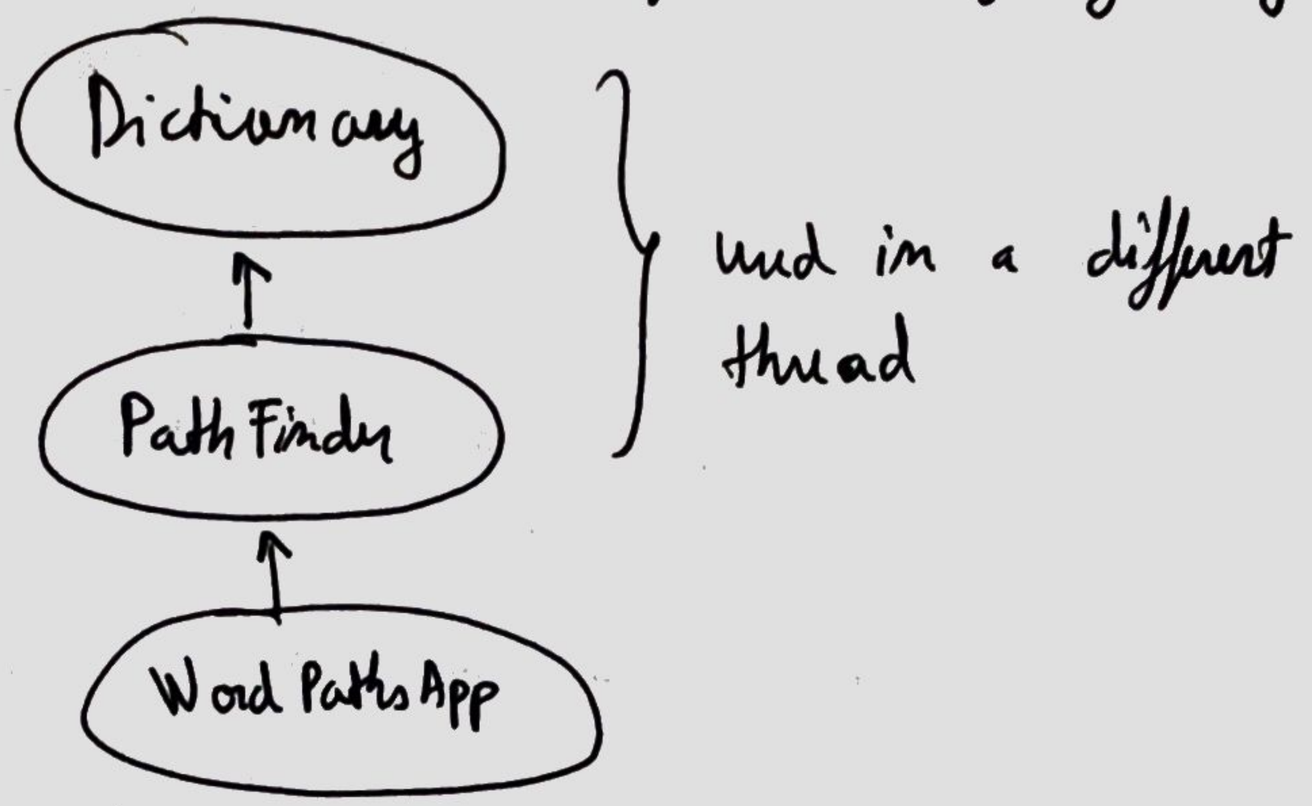
Tutor Matt Kusner

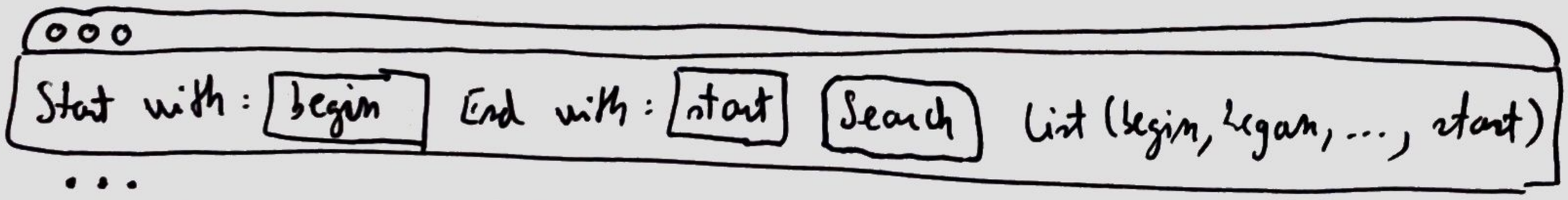
IP 3
Sheet 3
Alex Tatomir

Question 1

The quantity of code was too much to write by hand with no error. You can find all files at the end of this script.

Below you can find some sketches useful in designing my






```
...
val finder = new PathFinder (new Pichionary ("small-words"))
class Worker extends Thread {
    override def run {
        val answer = finder.find-path (SourceText.text, TargetText.text)
        println (answer)
        val message = if (answer == Nil) "No Solution" else answer.toString
        Swing.onEDT {solutionLabel.text = message}
    }
}
listenTo (SearchButton).
reactions += {
    case ButtonClicked (SearchButton) => (new Worker).start
}
...
```

Question 2

```
class MySet[T](elements: Set[T]) extends Set[T] {
    private val data: List[T] = elements.toList
    def contains(key: T): Boolean = {
        for (v <- this; if (key == v)) return true
        false
    }
    def iterator: Iterator[T] = data.iterator
    def +(elem: T): MySet[T] = new MySet(data.toSet + elem)
    def -(elem: T): MySet[T] = new MySet(data.toSet - elem)
    override def empty: MySet[T] = new MySet(Set())
}
```

Question 3

(a) First of all, we have to force our class to extend the Partial Order trait:

> class MySet[T](elems: Set[T]) extends Set[T] with PartialOrder[MySet[T]]

and define:

```
def <=(that: MySet[T]): Boolean = {
    for (v <- this)
        if (!that.contains(v))
            return false
    true
}
def lub(that: MySet[T]): MySet[T] = {
    var answer = this
    for (v <- that) answer = answer + v
    answer
}
```

Note that the method

> for (v <- that)

can be used because the method "iterator" is defined before.

(b)

```
class UpSet[T <: PartialOrder[T]](_elements : Set[T]) {
  /** Represent the set using a 'minimal' standard set */
  private var elements = minimal(_elements)

  /** check if new value is in the set */
  def contains(x:T): Boolean = {
      for (v <- elements; if (v <= x)) return true
      false
  }

  /** intersect two UpSets */
  def intersection(that: UpSet[T]): UpSet[T] = {
      var answer = Set[T]()
      for (a <- elems; s <- that.elems) answer = answer + (a lub s)
      new UpSet(minimal(answer)) // or UpSet(answer)
  }

  /** a private function that finds the minimal subset */
  private def minimal(data: Set[T]): Set[T] = {
      var answer: Set[T] = data
      for (a <- data) {
          var ok = true
          for (b <- data; if (a != b && b <= a)) ok = false
          if (!ok) answer = answer - a
      }
      answer
  }
}
```

(c) First of all, let's enhance my Upset with PartialOrder[Upset[T]]

> class Upset[T <: PartialOrder[T]](_elems: Set[T]) extends
PartialOrder[Upset[T]]

and define methods <= and lub:

```
/** compare this with that */
def <= (that: Upset[T]): Boolean = {
    for (v ← elems; if (!that.contains(v))) return false
    true
}
/** find the lub of two sets (actually i.e. Union) */
def lub (that: Upset[T]): Upset[T] = {
    var answer = elems
    for (v ← that.elems) answer = answer + v
    new Upset(answer)  // automatically finds the minimal subset
}
```

Question 4

It can be observed that the class parameter of Bag is a function f: T -> IN. Therefore, the class can be defined as:

```
class Bag[T](private val f: (T => Int)) {
    def add(x: T): Bag[T] = new Bag[T]( v => {
        if (x == v) f(v) + 1 else f(v)
    } /** remove an elem if it is in the bag */
    def remove(x: T): Bag[T] = new Bag[T] (v => {
        if (x == v) Math.max(0, f(v) - 1) else f(v)
    } // do nothing if not in the bag
    def count(x: T) = f(x)                                  // or count
    def union(that: Bag[T]): Bag[T] = new Bag(v => f(v) + that.f(v))
}
```

All implementation is straightforward from the definition of each method.

The Bag is represented as a function and function should be contravariants (in the domain), i.e we can use g: A -> C instead of f: B -> C as long as B ⊆ A. Therefore, the bag should be contravariant. This change can be achieved by defining the generic type T with a "-" sign

▷ class Bag[-T](private val f: (T => Int)) {...}

Now, everything works as expected, except the union function. Note that if we have two functions f: B->C, g: A->C with B ⊆ A, the union (f ∪ g) should be defined on B -> C. Therefore, the correct definition of union is:

▷ def union[X <: T](that: Bag[X]): Bag[X] = new Bag[X](v => f(v) + that.count(v))

Question 5

As it can be seen in lecture notes, mutable objects/classes should not be covariant or contravariant because unexpected exceptions may happen. Suppose that A <: B and Set[A] <: Set[B] (mutable sets).

```
var a : Set[A] = new Set[A]
var b : Set[B] = new Set[B]
b = a  // covariance allows this operation
b += new B  // exception because in b can now be added
            only objects of type A or its subtypes
```

and the contravariant situation: A <: B and Set[A] >: Set[B]

```
var a : Set[A] = new Set[A]
var b : Set[B] = new Set[B]
a = b  // contravariance allows this operation
a += new B
```

now works even though it shouldn't because B is not a subtype of A.

However, it may provide more flexibility if it is handled very carefully. For general purpose, it is safe to treat mutable classes as invariants.

Question 6

A full set of documentation for the AutoSnail case study can be easily generated by running the command

```
> scaladoc <path to sources>
```

In my case, I created a new folder "docs" in the same place where the "src" folder is and ran:

```
> scaladoc ../src/*
```

A lot of html files were created, including index.html which is the main entry to the documentation. In the AppFrame class can be found the following methods inherited from scala.swing.UIElement:

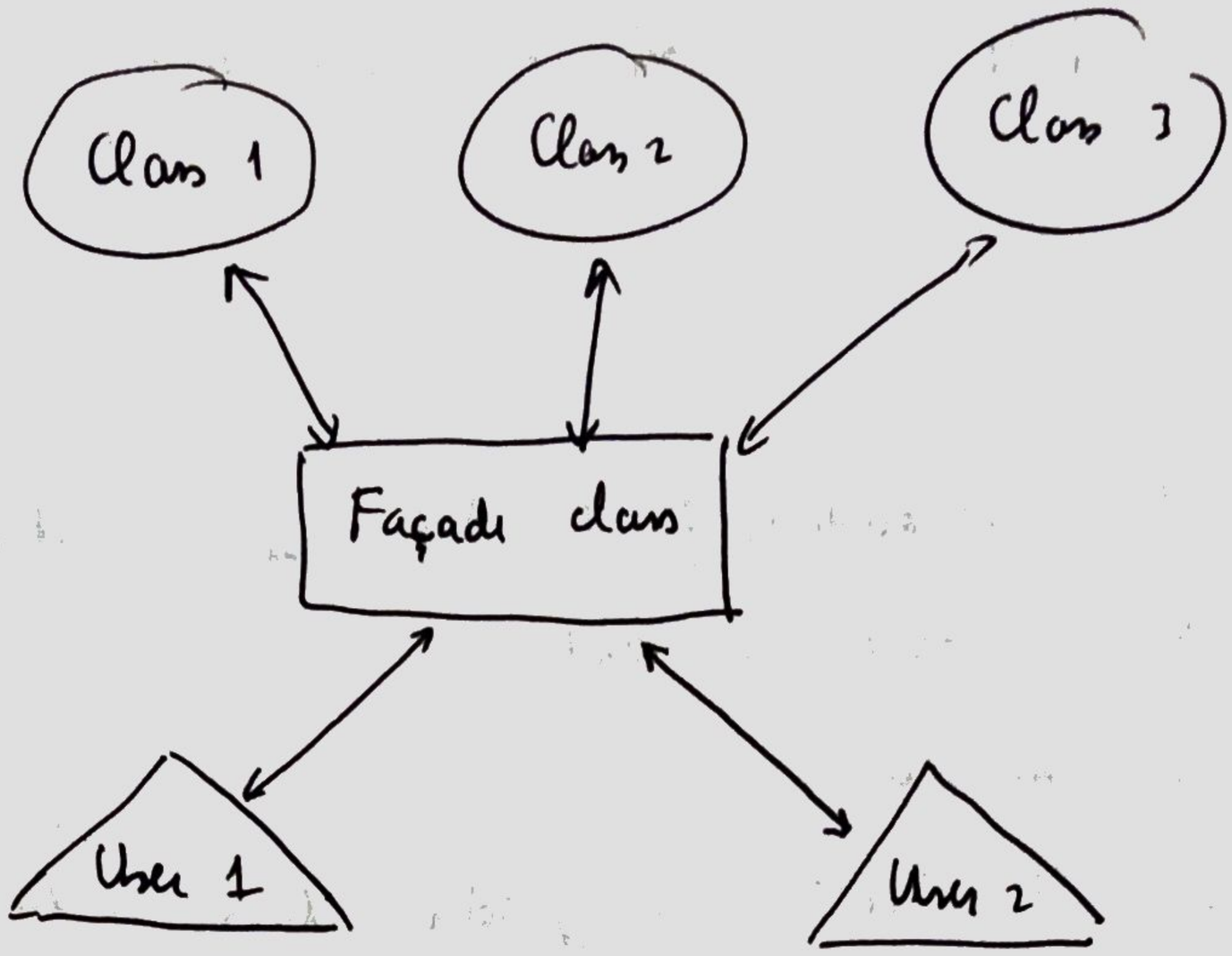
```
> def background : Color
> def background_=(c: Color): Unit      (... and many more ...)
> def toolkit : Toolkit
```

Question 7

The Façade pattern supports a simple interface to larger code. It is mainly used to simplify a set of complex classes / structured code by a front-facing easy-to-use interface. For example, the AppFrame implements the Façade pattern to make the access to the GUI easier (only through this class). It also increases the ~~tooody~~ loose coupling between all components.

The responsibility of the AppFrame is to aggregate all GUI elements into one window, also managing the events that happen.

You can find a descriptive sketch of the Façade pattern.



Question 8

First of all, we have to create a new button that can be used to change the color:

```
> private val ChangeColorButton = new Button { text = "Change ma color" }
```

and make the AppFrame a reactor to this button

```
>   listenTo (ChangeColorButton)
>   reactions += {
>       case ButtonClicked (ChangeColorButton) => {
>           val default_color = viewer.background
>           val option_color = ColorChooser.showDialog (this, "Choose ma color",
>                                                       default color)
>           option_color match {
>               case None => println ("No color chosen")
>               case Some (new_color) =>
>                   Swing.onEDT {
>                       viewer.background = new_color
>                       viewer.repaint
>                   }
>           }
>       }
> }
```

may be included directly i.e without "Swing.onEDT."

Another essential thing to do is to add the button to the design:

```
> val elems = Array[Component](...
>                 ...
>                 ...)
>                 (changeColorButton)
```

Now, the new feature is completely functional and can be used.

Questions

The method 'findTower' is too slow. How can we make it faster? We can sort all towers in increasing order of their x coordinate and binary search to find the first candidate. In this way, we eliminated a lot of candidates from the prefix of the array.

Similarly, we can stop our search when all remaining towers have their x coordinate too large.

```
/** init the ordredTowers array in ascending order of x */
  var countTowers = 0
  var ordredTowers : Array[RoadMap.Towers] = null

  sortTowers

private def sortTowers = {
    ordredTowers = map.towers.toArray.sortBy(_.location.x)
    countTowers = ordredTowers.length
}

/** find the first position with x ≥ limit - in log time */
def findFirstPosition(limit: Float): Int = {
    var l = 0; var r = countTowers
    while (l < r) {
        val mid = (l+r)/2
        if (transform(orderedTowers(mid).location).getX < limit)
            l = mid+1
        else r = mid
    }
    l
}
```

```
/** Find the tower that contains a given point, or return null */
protected def findTower(p: Point2D): RoadMap.Tower = {
  var tower: RoadMap.Tower = null
  var searched = 0
  val lower_limit: Float = p.getX.toFloat - tol.toFloat
  val upper_limit: Float = p.getX.toFloat + tol.toFloat
  for (pos <- findAllPointsIn(lower_limit, upper_limit, towerTree)) {
    searched += 1
    val t = towerAt(pos)
    val q = translate(t.location)
    if (p.distance(q) <= tol) {
      println(searched)
      return t
    }
    if (translate(t.location).getX > upper_limit) {
      println(searched)
      return tower
    }
  }
  println(searched)
  tower
}
```

Using the commented code above we can find the number of candidate towers used in the search. This value is very small (in general ≤ 3) so the overall complexity of findTower is now $O(\log(\text{countTowers}))$ not $O(\text{countTowers})$ as before (in the worst case).

```
/** The Dictionary class - Dictionary.scala*/
class Dictionary(fname: String) {
    private val words = new scala.collection.mutable.HashSet[String]
    initDict

    /** Check if a word is in the dictinary */
    def isWord(w: String) = words.contains(w)

    /** Initialize the dictionary */
    private def initDict = {
        val allWords = scala.io.Source.fromFile(fname).getLines
        def include(w: String) = w.forall(_.isLower)
        for (w <- allWords; if include(w)) words += w
    }
}


/** WordPathsApp.scala */

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

object WordPathsApp extends SimpleSwingApplication {
    def top = new MainFrame {
        object SourceText extends TextField {columns = 10}
        object TargetText extends TextField {columns = 10}
        object SearchButton extends Button{text = "Search"}
        var solutionLabel = new Label {text = "No solution"}
        contents = new FlowPanel {
            contents += new Label{text = "Start with: "}
            contents += SourceText
            contents += new Label{text = " End with: "}
            contents += TargetText
            contents += SearchButton
            contents += solutionLabel
        }

        /** the class that finds the path */
        val finder = new PathFinder(new Dictionary("knuth_words"))

        /** a thread that finds the specified path */
        class Worker extends Thread {
            override def run {
                val answer = finder.find_path(SourceText.text, TargetText.text)
                println(answer)

                val message = if (answer == Nil) "No solution" else
answer.toString
                Swing.onEDT {solutionLabel.text = message}
            }
        }

        listenTo(SearchButton)
        reactions += {
            case ButtonClicked(SearchButton) => (new Worker).start
        }
    }
}


/** PathFinder.scala */
/** A class that finds paths in a dictionary */
class PathFinder(dict: Dictionary) {
    private val path = new scala.collection.mutable.HashMap[String,
```

```
List[String]]
    private val queue = new scala.collection.mutable.Queue[String]

    /** Find a path from source to target */
    def find_path(source: String, target: String): List[String] = {
        if (!dict.isWord(source) || !dict.isWord(target) || source.length !=
target.length)
            return Nil

        path.clear
        queue.clear
        path.update(source, List(source))
        queue.enqueue(source)
        while (!queue.isEmpty && !path.contains(target)) {
            val word = queue.dequeue
            val len = word.length
            for (i <- 0 until len; c <- 'a' to 'z') {
                if (word.charAt(i) != c) {
                    val new_word = word.take(i) + c + word.drop(i + 1)
                    if (!path.contains(new_word) && dict.isWord(new_word)) {
                        path.update(new_word, new_word :: path(word))
                        queue.enqueue(new_word)
                    }
                }
            }
        }

        if (path.contains(target))
            path(target).reverse
        else
            Nil
    }
}
```