|  |  |  |
| --- | --- | --- |
| CS3343-LA3 2014/15 Sem A December 5, 2014 | B.T.  Refactor Report | |
| [Type the abstract of the document here. The abstract is typically a short summary of the contents of the document. Type the abstract of the document here. The abstract is typically a short summary of the contents of the document.] | | Prepared by  B.T. |

Contents

[Refactor #01: Create an boundary class IO to store respective methods 2](#_Toc405144548)

[Situation 2](#_Toc405144549)

[Plan 2](#_Toc405144550)

[Changes 2](#_Toc405144551)

[Schedule 2](#_Toc405144552)

[TestSchedule 2](#_Toc405144553)

[Refactor #02: Create an entity class Timetable 3](#_Toc405144554)

[Situation 3](#_Toc405144555)

[Plan 3](#_Toc405144556)

[Changes 3](#_Toc405144557)

[Schedule 3](#_Toc405144558)

[RequiredConstraint 4](#_Toc405144559)

[IO 5](#_Toc405144560)

[BuildingConstraint 5](#_Toc405144561)

[TimeConstraint 6](#_Toc405144562)

[TimeGapConstraint 6](#_Toc405144563)

[Changes 7](#_Toc405144564)

[Schedule 7](#_Toc405144565)

[Timetable 7](#_Toc405144566)

[TimeGapConstraint 8](#_Toc405144567)

[Refactor #03: Create a new control class Utilities to store all helper functions 9](#_Toc405144568)

[Situation 9](#_Toc405144569)

[Plan 9](#_Toc405144570)

[Changes 9](#_Toc405144571)

[Schedule 9](#_Toc405144572)

[Utilities 12](#_Toc405144573)

# Refactor #01: Create an boundary class IO to store respective methods

## Situation

All methods are in the Schedule class (where the main function is), in order to preserve better meaning of different classes, an entity class (‘IO’) is added such that all input- and output-related functions can be located.

## Plan

Create a class called IO and move all related functions to it. Modify all corresponding caller-callee relationships.

## Changes

### Schedule

Functions readTimeslots and printSchedule moved to IO. Removed printScheduleHeader.

#### Before

In main function:

|  |
| --- |
| …  readTimeslots(timeslots, inputFile);  …  if (rc.isFulfilled()) {printSchedule(validPermutatedUniqueCourseTimeslotsList.get(i)); break; }  … |

#### After

In main function:

|  |
| --- |
| …  IO.readTimeslots(timeslots, inputFile);  …  if (rc.isFulfilled()) {IO.printSchedule(validPermutatedUniqueCourseTimeslotsList.get(i)); break; }  … |

### TestSchedule

#### Before

|  |
| --- |
| Schedule.readTimeslots(timeslots, "CS3343\_data.txt"); |
| Schedule.printSchedule(timeslots); |

#### After

|  |
| --- |
| Import schedule.IO; |

|  |
| --- |
| @Test  **public** **void** testPrintScheduleHeader() {  } |

|  |
| --- |
| IO.readTimeslots(timeslots, "CS3343\_data.txt"); |
| IO.printSchedule(timeslots); |

# Refactor #02: Create an entity class Timetable

## Situation

Semantically, a set of timeslots is a timetable. However, it was implemented as an array list of class timeslot.

## Plan

To create a new entity class called Timetable and move corresponding methods into it. Respective caller-callee relationships should also be modified.

Please note that, not all array list of timeslots are refactored because of the difference in meaning.

## Changes

Refactor caller-callee relationships.

### Schedule

#### Before

In main function:

|  |
| --- |
| ArrayList<ArrayList<ArrayList<Timeslot>>> allPerm = GeneratePermutations(permutatedUniqueCourseTimeslotsList);  ArrayList<ArrayList<Timeslot>> validPermutatedUniqueCourseTimeslotsList = **new** ArrayList<ArrayList<Timeslot>>();  **for** (ArrayList<Timeslot> i : allPerm.get(0)) {  **boolean** overlap = **false**;  **for** (Timeslot j : i) {  **for** (Timeslot k : i) {  **if** (j.equals(k)) {  **break**;  }  **if** (j.overlap(k)) {  overlap = **true**;  **break**;  }  }  }  **if** (!overlap)  validPermutatedUniqueCourseTimeslotsList.add(i);  }  **int** numValidCombinations = validPermutatedUniqueCourseTimeslotsList.size();  **for** (**int** i=0; i < validPermutatedUniqueCourseTimeslotsList.size(); i++) {  ArrayList<Timeslot> l = validPermutatedUniqueCourseTimeslotsList.get(i);  RequiredConstraint rc = **new** RequiredConstraint(l, listOfCrns);  **if** (rc.isFulfilled())  {  IO.*printSchedule*(validPermutatedUniqueCourseTimeslotsList.get(i));  **break**;  }  } |

In sortByStartTime function:

|  |
| --- |
| **public** **static** **void** sortByStartTime(ArrayList<Timeslot> timetable, ArrayList<Timeslot> result) |

#### After

In main function:

|  |
| --- |
| ArrayList<Timetable> validPermutatedUniqueCourseTimeslotsList = **new** ArrayList<Timetable>();  **for** (ArrayList<Timeslot> i : allPerm.get(0)) {  **boolean** overlap = **false**;  **for** (Timeslot j : i) {  **for** (Timeslot k : i) {  **if** (j.equals(k)) {  **break**;  }  **if** (j.overlap(k)) {  overlap = **true**;  **break**;  }  }  }  **if** (!overlap){  Timetable timetable = **new** Timetable(i);  validPermutatedUniqueCourseTimeslotsList.add(timetable);  }    } |

In sortByStartTime function:

|  |
| --- |
| **public** **static** **void** sortByStartTime(Timetable timetable, ArrayList<Timeslot> result) |

### RequiredConstraint

#### Before

|  |
| --- |
| **public** RequiredConstraint(ArrayList<Timeslot> l, ArrayList<String> listOfCrns) {  **boolean** found = **true**;  **for** (String i : listOfCrns) {  **boolean** foundi = **false**;  **for** (Timeslot s : l)  **if** (i.equals(s.getCrn())) {  foundi = **true**;  }  found &= foundi;  }    **this**.fulfilled = found;  } |

#### After

|  |
| --- |
| **public** RequiredConstraint(Timetable l, ArrayList<String> listOfCrns) {  **boolean** found = **true**;  **for** (String i : listOfCrns) {  **boolean** foundi = **false**;  **for** (Timeslot s : l.getTimeslots())  **if** (i.equals(s.getCrn())) {  foundi = **true**;  }  found &= foundi;  }    **this**.fulfilled = found;  } |

### IO

#### Before

|  |
| --- |
| **public** static void printSchedule(ArrayList<String> timetable) |

#### After

|  |
| --- |
| **public** **static** **void** printSchedule(Timetable timetable) |

### BuildingConstraint

#### Before

|  |
| --- |
| **public** BuildingConstraint(Timetable timeslots, ArrayList<String> listOfBldgs) {  **for** (String i : listOfBldgs) {  **for** (Timeslot s : timeslots {  **if** (i.equals(s.getBuilding())) {  System.*out*.println("Found " + i);  **this**.fulfilled = **false**;  **break**;  }  }  }  } |

#### After

|  |
| --- |
| **public** BuildingConstraint(Timetable timeslots, ArrayList<String> listOfBldgs) {  **for** (String i : listOfBldgs) {  **for** (Timeslot s : timeslots.getTimeslots()) {  **if** (i.equals(s.getBuilding())) {  System.*out*.println("Found " + i);  **this**.fulfilled = **false**;  **break**;  }  }  }  } |

### TimeConstraint

#### Before

|  |
| --- |
| **public** TimeConstraint(ArrayList<Timeslot> timetable, HashMap<Integer,ArrayList<Double>> daytimeExcluded) {  **for** (Timeslot i : timetable) {  **if** (daytimeExcluded.containsKey(i.getDay())) {  **for** (**double** j : daytimeExcluded.get(i.getDay())) {  **if** (j < i.getFinishTime() && j >= i.getStartTime()) {  **this**.fulfilled = **false**;  **break**;  }  }  }  }  } |

#### After

|  |
| --- |
| **public** TimeConstraint(Timetable timetable, HashMap<Integer,ArrayList<Double>> daytimeExcluded) {  **for** (Timeslot i : timetable.getTimeslots()) {  **if** (daytimeExcluded.containsKey(i.getDay())) {  **for** (**double** j : daytimeExcluded.get(i.getDay())) {  **if** (j < i.getFinishTime() && j >= i.getStartTime()) {  **this**.fulfilled = **false**;  **break**;  }  }  }  }  } |

### TimeGapConstraint

#### Before

|  |
| --- |
| **public** TimeGapConstraint(ArrayList<Timeslot> t, **double** timeDifference) {  ArrayList<Timeslot> r = **new** ArrayList<Timeslot>();  Schedule.*sortByStartTime*(t, r);    **for** (**int** i=0; i<r.size()-1; i++) {  **if** ((r.get(i).getDay() == r.get(i+1).getDay()) && (r.get(i+1).getStartTime() - r.get(i).getFinishTime() > timeDifference)) {  **this**.fulfilled = **false**;  **break**;  }  }  } |

#### After

|  |
| --- |
| **public** TimeGapConstraint(Timetable timetable, **double** timeDifference) {  ArrayList<Timeslot> r = **new** ArrayList<Timeslot>();  Schedule.*sortByStartTime*(timetable, r);    **for** (**int** i=0; i<r.size()-1; i++) {  **if** ((r.get(i).getDay() == r.get(i+1).getDay()) && (r.get(i+1).getStartTime() - r.get(i).getFinishTime() > timeDifference)) {  **this**.fulfilled = **false**;  **break**;  }  }  } |

## Changes

Moved function sortByStartTime from Schedule to Timetable.

### Schedule

#### Before

|  |
| --- |
| public void sortByStartTime(Timetable timetable, Timetable result) {  … //omitted  } |

### Timetable

#### After

|  |
| --- |
| **public** **void** sortByStartTime(Timetable result) {  **if** (**this**.size() == 1) {  result.add(**this**.get(0));  **return**;  }  **double** min = Double.*MAX\_VALUE*;  **int** minIdx = 0;  **for** (**int** i=0; i<**this**.size(); i++) {  **if** (**this**.get(i).getStartTime()+(**this**.get(i).getDay()-1)\*24 < min) {  min = **this**.get(i).getStartTime()+(**this**.get(i).getDay()-1)\*24;  minIdx = i;  }  }  result.add(**this**.get(minIdx));  **this**.remove(minIdx);  **this**.sortByStartTime(result);  } |

### TimeGapConstraint

#### Before

|  |
| --- |
| **public** TimeGapConstraint(Timetable timetable, **double** timeDifference) {  Timetable sorted = **new** Timetable();  Schedule.sortByStartTime(timetable, sorted);    **for** (**int** i=0; i<sorted.size()-1; i++) {  **if** ((sorted.get(i).getDay() == sorted.get(i+1).getDay()) && (sorted.get(i+1).getStartTime() - sorted.get(i).getFinishTime() > timeDifference)) {  **this**.fulfilled = **false**;  **break**;  }  }  } |

#### After

|  |
| --- |
| **public** TimeGapConstraint(Timetable timetable, **double** timeDifference) {  Timetable sorted = **new** Timetable();  timetable.sortByStartTime(sorted);    **for** (**int** i=0; i<sorted.size()-1; i++) {  **if** ((sorted.get(i).getDay() == sorted.get(i+1).getDay()) && (sorted.get(i+1).getStartTime() - sorted.get(i).getFinishTime() > timeDifference)) {  **this**.fulfilled = **false**;  **break**;  }  }  } |

# Refactor #03: Create a new control class Utilities to store all helper functions

## Situation

All helper functions are in the Schedule class, where the main function is located. It would better to relocate them to a newly created class (“Utilities”) which is specifically responsible for storing such functions.

## Plan

To move all helper functions from Schedule to a new class, Utilities, and modify corresponding caller-callee relationships.

## Changes

### Schedule

#### Before

*Functions to be moved to Utilities are omitted. Please refer to the part “After”.*

|  |
| --- |
| **public** **static** **void** main(String[] args) {  ArrayList<Timeslot> timeslots = **new** ArrayList<Timeslot>();      String inputFile = args[0];    IO.*readTimeslots*(timeslots, inputFile); //Extract method    ArrayList<String> uniqueCourses = allCourses(timeslots);  HashMap<String,HashMap<String,ArrayList<Timeslot>>> uniqueCourseTimeslots = **new** HashMap<String,HashMap<String,ArrayList<Timeslot>>>();  **for** (String i : uniqueCourses) {  ArrayList<Timeslot> t = extractTimeslotsByCode(timeslots, i);  ArrayList<Timeslot> allLectures = extractTimeslotsByType(t, "Lecture");  ArrayList<Timeslot> allTutorials = extractTimeslotsByType(t, "Tutorial");  HashMap<String,ArrayList<Timeslot>> slot = **new** HashMap<String,ArrayList<Timeslot>>();  **if** (allLectures.size()>0)  slot.put("Lecture", allLectures);  **if** (allTutorials.size()>0)  slot.put("Tutorial", allTutorials);  uniqueCourseTimeslots.put(i, slot);  }  HashMap<String,ArrayList<ArrayList<Timeslot>>> permutatedUniqueCourseTimeslots = **new** HashMap<String,ArrayList<ArrayList<Timeslot>>>();  ArrayList<ArrayList<ArrayList<Timeslot>>> permutatedUniqueCourseTimeslotsList = **new** ArrayList<ArrayList<ArrayList<Timeslot>>>();  **for** (String i : uniqueCourses) {  permutatedUniqueCourseTimeslots.put(i, permutate(uniqueCourseTimeslots.get(i).get("Lecture"), uniqueCourseTimeslots.get(i).get("Tutorial")));  permutatedUniqueCourseTimeslotsList.add(permutate(uniqueCourseTimeslots.get(i).get("Lecture"), uniqueCourseTimeslots.get(i).get("Tutorial")));  }  ArrayList<ArrayList<ArrayList<Timeslot>>> allPerm = GeneratePermutations(permutatedUniqueCourseTimeslotsList);    ArrayList<Timetable> validPermutatedUniqueCourseTimeslotsList = **new** ArrayList<Timetable>();  **for** (ArrayList<Timeslot> i : allPerm.get(0)) {  **boolean** overlap = **false**;  **for** (Timeslot j : i) {  **for** (Timeslot k : i) {  **if** (j.equals(k)) {  **break**;  }  **if** (j.overlap(k)) {  overlap = **true**;  **break**;  }  }  }  **if** (!overlap){  Timetable timetable = **new** Timetable(i);  validPermutatedUniqueCourseTimeslotsList.add(timetable);  }    }  **int** numValidCombinations = validPermutatedUniqueCourseTimeslotsList.size();  **if** (numValidCombinations == 0)  System.*out*.println("There is no possible combination i.e. You should remove at least 1 course.");  **else**  System.*out*.println("There are " + numValidCombinations + " possible combinations.");  ArrayList<String> listOfCrns = **new** ArrayList<String>();  listOfCrns.add("60002");  listOfCrns.add("50005");  **for** (**int** i=0; i < validPermutatedUniqueCourseTimeslotsList.size(); i++) {  Timetable l = validPermutatedUniqueCourseTimeslotsList.get(i);  RequiredConstraint rc = **new** RequiredConstraint(l, listOfCrns);  **if** (rc.isFulfilled())  {  IO.*printSchedule*(validPermutatedUniqueCourseTimeslotsList.get(i));  **break**;  }  }  } |

#### After

|  |
| --- |
| ArrayList<Timeslot> timeslots = **new** ArrayList<Timeslot>();  String inputFile = args[0];    IO.*readTimeslots*(timeslots, inputFile); //Extract method    ArrayList<String> uniqueCourses = Utilities.*allCourses*(timeslots);  HashMap<String,HashMap<String,ArrayList<Timeslot>>> uniqueCourseTimeslots = **new** HashMap<String,HashMap<String,ArrayList<Timeslot>>>();  **for** (String i : uniqueCourses) {  ArrayList<Timeslot> t = Utilities.*extractTimeslotsByCode*(timeslots, i);  ArrayList<Timeslot> allLectures = Utilities.*extractTimeslotsByType*(t, "Lecture");  ArrayList<Timeslot> allTutorials = Utilities.*extractTimeslotsByType*(t, "Tutorial");  HashMap<String,ArrayList<Timeslot>> slot = **new** HashMap<String,ArrayList<Timeslot>>();  **if** (allLectures.size()>0)  slot.put("Lecture", allLectures);  **if** (allTutorials.size()>0)  slot.put("Tutorial", allTutorials);  uniqueCourseTimeslots.put(i, slot);  }  HashMap<String,ArrayList<ArrayList<Timeslot>>> permutatedUniqueCourseTimeslots = **new** HashMap<String,ArrayList<ArrayList<Timeslot>>>();  ArrayList<ArrayList<ArrayList<Timeslot>>> permutatedUniqueCourseTimeslotsList = **new** ArrayList<ArrayList<ArrayList<Timeslot>>>();  **for** (String i : uniqueCourses) {  permutatedUniqueCourseTimeslots.put(i, Utilities.*permutate*(uniqueCourseTimeslots.get(i).get("Lecture"), uniqueCourseTimeslots.get(i).get("Tutorial")));  permutatedUniqueCourseTimeslotsList.add(Utilities.*permutate*(uniqueCourseTimeslots.get(i).get("Lecture"), uniqueCourseTimeslots.get(i).get("Tutorial")));  }  ArrayList<ArrayList<ArrayList<Timeslot>>> allPerm = Utilities.*GeneratePermutations*(permutatedUniqueCourseTimeslotsList);    ArrayList<Timetable> validPermutatedUniqueCourseTimeslotsList = **new** ArrayList<Timetable>();  **for** (ArrayList<Timeslot> i : allPerm.get(0)) {  **boolean** overlap = **false**;  **for** (Timeslot j : i) {  **for** (Timeslot k : i) {  **if** (j.equals(k)) {  **break**;  }  **if** (j.overlap(k)) {  overlap = **true**;  **break**;  }  }  }  **if** (!overlap){  Timetable timetable = **new** Timetable(i);  validPermutatedUniqueCourseTimeslotsList.add(timetable);  }    }  **int** numValidCombinations = validPermutatedUniqueCourseTimeslotsList.size();  **if** (numValidCombinations == 0)  System.*out*.println("There is no possible combination i.e. You should remove at least 1 course.");  **else**  System.*out*.println("There are " + numValidCombinations + " possible combinations.");  ArrayList<String> listOfCrns = **new** ArrayList<String>();  listOfCrns.add("60002");  listOfCrns.add("50005");  **for** (**int** i=0; i < validPermutatedUniqueCourseTimeslotsList.size(); i++) {  Timetable l = validPermutatedUniqueCourseTimeslotsList.get(i);  RequiredConstraint rc = **new** RequiredConstraint(l, listOfCrns);  **if** (rc.isFulfilled())  {  IO.*printSchedule*(validPermutatedUniqueCourseTimeslotsList.get(i));  **break**;  }  } |

### Utilities

#### After

|  |
| --- |
| /\*\* The first session's start time. \*/  **public** **static** **double** *firstTime* = 8.0;    /\*\* The last session's finish time. \*/  **public** **static** **double** *lastTime* = 23.0;  /\*\*  \* All courses.  \*  \* @param timeslots the timeslots  \* @return the array list of all courses  \*/  public static ArrayList<String> allCourses(ArrayList<Timeslot> timeslots) {  ArrayList<String> courses = new ArrayList<String>();  for (Timeslot j : timeslots) {  boolean contains = false;  for(String i: courses){  if(j.getCode().equals(i)){  contains = true;  break;  }  }  if (!contains)  courses.add(j.getCode());  }  return courses;  }  /\*\*  \* Extract timeslots by day.  \*  \* @param timeslots the timeslots  \* @param day the day  \* @return the array list of courses of a particular day  \*/  public static ArrayList<Timeslot> extractTimeslotsByDay(ArrayList<Timeslot> timeslots, Weekday day) {  ArrayList<Timeslot> t = new ArrayList<Timeslot>();  for (Timeslot i : timeslots)  if (i.getDay() == day.getDay())  t.add(i);  return t;  }  /\*\*  \* Extract timeslots by course code.  \*  \* @param timeslots the timeslots  \* @param code the course code  \* @return the array list of courses of a particular course  \*/  public static ArrayList<Timeslot> extractTimeslotsByCode(ArrayList<Timeslot> timeslots, String code) {  ArrayList<Timeslot> t = new ArrayList<Timeslot>();  for (Timeslot i : timeslots)  if (i.getCode().equals(code))  t.add(i);  return t;  }  /\*\*  \* Extract timeslots by session type (lecture/tutorial).  \*  \* @param timeslots the timeslots  \* @param type the type of session  \* @return the array list of courses of a particular type of session  \*/  public static ArrayList<Timeslot> extractTimeslotsByType(ArrayList<Timeslot> timeslots, String type) {  ArrayList<Timeslot> t = new ArrayList<Timeslot>();  for (Timeslot i : timeslots)  if (i.getType().equals(type))  t.add(i);  return t;  }  /\*\*  \* Permutate.  \*  \* @param list1 the first array list to permutate  \* @param list2 the second array list to permutate  \* @return the array list of permutating list1 with list2  \*/  public static ArrayList<ArrayList<Timeslot>> permutate(ArrayList<Timeslot> list1, ArrayList<Timeslot> list2) {  ArrayList<ArrayList<Timeslot>> res = new ArrayList<ArrayList<Timeslot>>();  for (Timeslot i: list1) {  for (Timeslot j: list2) {  ArrayList<Timeslot> t = new ArrayList<Timeslot>();  t.add(i);  t.add(j);  res.add(t);  }  }  return res;  }  /\*\*  \* Permutate array list of array list.  \*  \* @param list1 the first array list of array list to permutate  \* @param list2 the second array list of array list to permutate  \* @return the array list of permutating list1 with list2  \*/  public static ArrayList<ArrayList<Timeslot>> permutateArrayList(ArrayList<ArrayList<Timeslot>> list1, ArrayList<ArrayList<Timeslot>> list2) {  ArrayList<ArrayList<Timeslot>> res = new ArrayList<ArrayList<Timeslot>>();  for (ArrayList<Timeslot> i: list1) {  for (ArrayList<Timeslot> j: list2) {  ArrayList<Timeslot> t = new ArrayList<Timeslot>();  t.addAll(i);  t.addAll(j);  res.add(t);  }  }  return res;  }  /\*\*  \* Generate permutations by splitting an array list by course code and session type recursively.  \*  \* @param list the list storing all sessions  \* @return the array list of all permutated sessions  \*/  public static ArrayList<ArrayList<ArrayList<Timeslot>>> GeneratePermutations(ArrayList<ArrayList<ArrayList<Timeslot>>> list)  {  if (list.size() == 1)  return list;  ArrayList<ArrayList<Timeslot>> t = permutateArrayList(list.get(0), list.get(1));  list.add(t);  list.remove(1);  list.remove(0);  return GeneratePermutations(list);  }  /\*\*  \* To generate an array list of time before a given time t.  \*  \* @param t the given time t  \* @return the array list of time NOT before time t  \*/  public static ArrayList<Double> beforeTime(double t) {  ArrayList<Double> listOfExcludedTime = new ArrayList<Double>();  for (double i = firstTime; i < t; i++) {  listOfExcludedTime.add(i);  }  return listOfExcludedTime;  }  /\*\*  \* To generate an array list of time after a given time t.  \*  \* @param t the given time t  \* @return the array list of time NOT after time t  \*/  public static ArrayList<Double> afterTime(double t) {  ArrayList<Double> listOfExcludedTime = new ArrayList<Double>();  for (double i = t; i < lastTime; i++) {  listOfExcludedTime.add(i);  }  return listOfExcludedTime;  }  /\*\*  \* To generate an array list of time between 2 given times t1 and t2. (t1, t2]  \*  \* @param t1, t2 the given times  \* @return the array list of time NOT between t1 and t2  \*/  public static ArrayList<Double> betweenTime(double t1, double t2) {  ArrayList<Double> listOfExcludedTime = new ArrayList<Double>();  for (double i = t1; i < t2; i++) {  listOfExcludedTime.add(i);  }  return listOfExcludedTime;  } |