UNIVERSITYOF HERTFORDSHIRE
School of Engineering and Computer Science

COURSEWORK ASSIGNMENT

Module Title: Programming for Software Engineers	**Module Code: 7COM1025 (Semester B)**
Assignment Title: Main Coursework	**Individual Assignment**
Tutor: Dr Hui Cheng	**Internal Moderator**: Dr Raimund Kirner

Student ID Number **ONLY**:	Year Code:
18056745	1025-0105

Marks Awarded %:	Marks Awarded after Lateness Penalty applied %:

Penalties for Late Submissions

- For each day (or working day for hard copy* submission only) for up to five days after the published deadline, coursework submitted late (including deferred coursework, but with the exception of referred coursework), will have the numeric grade reduced by 10 grade points until or unless the numeric grade reaches or is 50 (PG).
- Coursework (including deferred coursework) submitted later than five days (five working days in the case of hard copy* submission) after the published deadline will be awarded a grade of zero (0).
- Late submission of referred coursework will automatically be awarded a grade of zero (0).
- Please Note: Work that is submitted through StudyNet (Canvas) is subject strictly to the School policy on late submission – even one second late will be subject to the lateness penalty, with few acceptable reasons for late submissions being acceptable. You are strongly advised to submit your work at least one hour before the submission deadline, to give time to resolve difficulties.

Please refer to your student handbook for details about the grading schemes used by the School when assessing your work. Guidance on assessment will also be given in the Module Guide.

Guidance on avoiding academic assessment offences such as plagiarism and collusion is given at this URL: http://www.studynet.herts.ac.uk/ptl/common/LIS.nsf/lis/citing_menu

Section 1 - Problem Definition:
Introduction

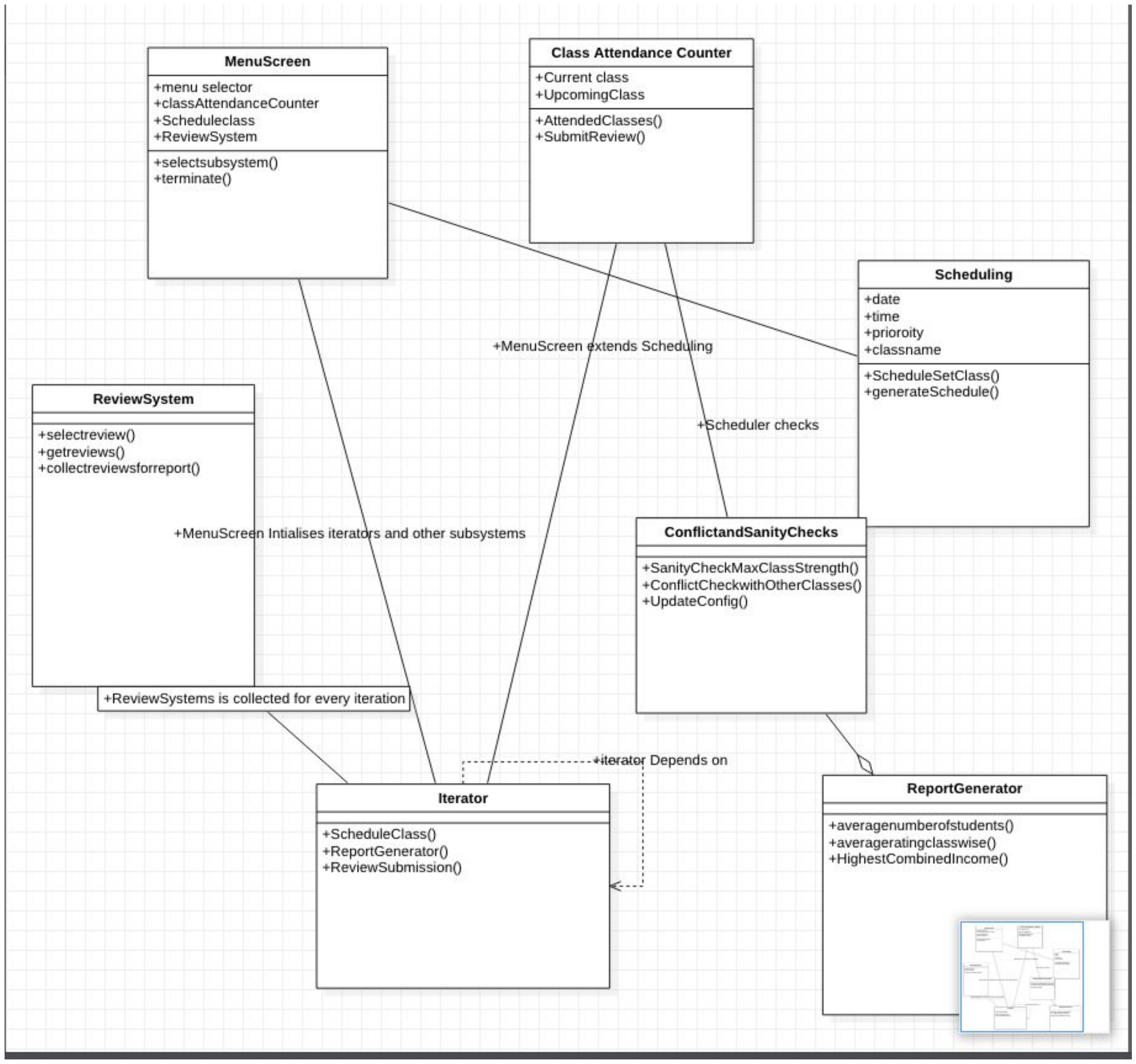
University Sports Centre (USC) needs software for managing the bookings of group exercise classes made by the students. The centre offers different group exercise classes on both Saturday and Sunday. The classes could be Yoga, Zumba, Aquacise, Box Fit, Body Blitz, etc. Each class can accommodate 4 students at most.

For either day (Saturday or Sunday), there are 3 classes for each day: 1 toward the beginning of the day, 1 toward the evening, 1 at night. The cost of each class is unique. The class cost for a similar exercise will continue as before regardless of whether they run at an alternate time.

An Student who needs to book a class needs to initially check the timetable and afterward select a class on a day. An Student can check the timetable by two different ways: one is by indicating the date and the other is by determining the activity name(Yoga, Zumba, Aquacise, Box Fit, Body Blitz). Students are permitted to change a booking, if there are as yet spaces accessible for the recently chosen class. An understudy can book the same number of classes as they need insofar as there is no time struggle.

After each gathering exercise class, students can compose a survey of the class they have joined in and give a numerical rating of the class extending from 1 to 5 (1: Very disappointed, 2: Dissatisfied, 3: Ok, 4: Satisfied, 5: Very Satisfied). The rating data will be recorded in the application

Section 2 – Design and Programming



UML Diagram
Classes and their Funtionallity

According to GoF definition, an iterator pattern provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation. It is behavioral design pattern.

As name implies, iterator helps in traversing the collection of objects in a defined manner which is useful the client applications. During iteration, client programs can perform various other operations on the elements as per requirements.

1. When to use iterator design pattern

Every programming language support some data structures like list or maps, which are used to store a group of related objects. In Java, we have `List`, `Map` and `Set` interfaces and their implementations such as ArrayList and HashMap.

A collection is only useful when it's provides a way to access its elements without exposing its internal structure. The iterators bear this responsibility.

So any time, we have collection of objects and clients need a way to iterate over each collection elements in some proper sequence, we must use iterator pattern to design the solution.

The iterator pattern allow us to design a collection iterator in such a way that –

    - we are able to access elements of a collection without exposing the internal structure of elements or collection itself.
    - iterator supports multiple simultaneous traversals of a collection from start to end in forward, backward or both directions.
    - iterator provide a uniform interface for traversing different collection types transparently.
- In Java, we have java.util.Iterator interface and it's specific implementations such as `ListIterator`. All Java collections provide some internal implementations of `Iterator` interface which is used to iterate over collection elements.

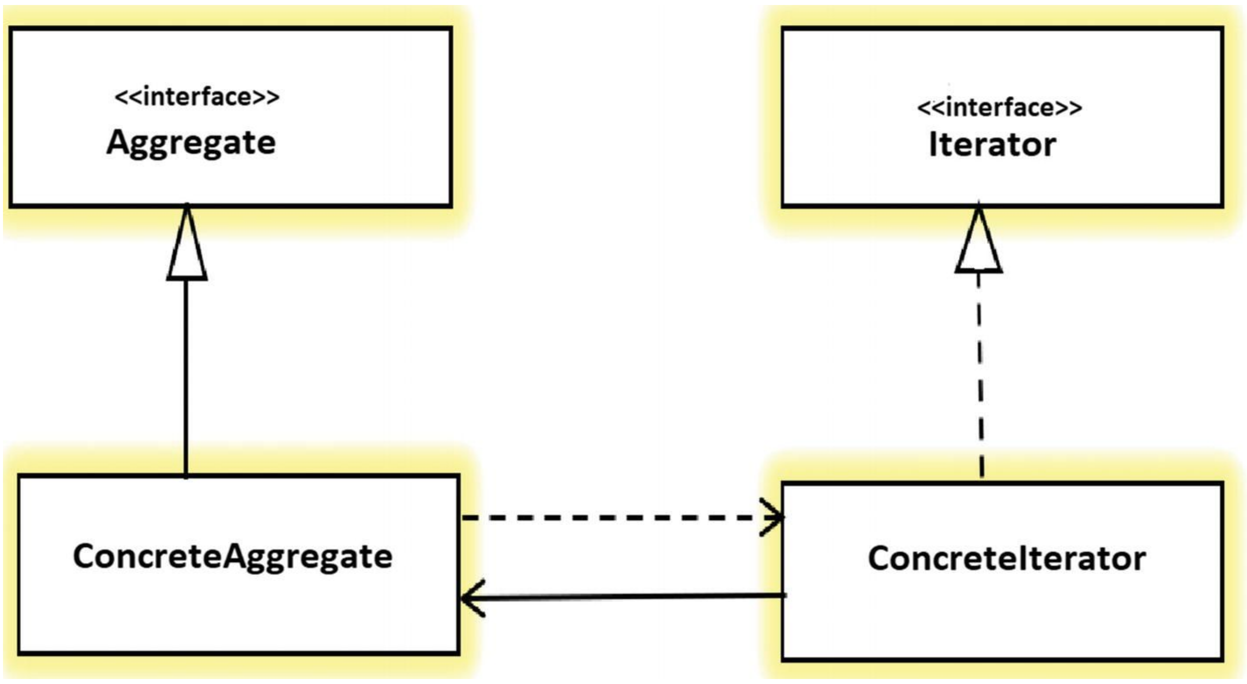
```
    List<String> names = Arrays.asList("alex", "brian", "charles");

    Iterator<String> namesIterator = names.iterator();

    while (namesIterator.hasNext())
    {
       String currentName = namesIterator.next();

       System.out.println(currentName);
 }
```

- In media players, we have a list of songs listed and we can play the song by traversing to the songs list and select desired song. It’s also an iterator example.



Data Structures and Algorithms (genetic Algorithm for scheduling)

In this chapter, we will create a genetic algorithm that schedules classes for a college timetable. We will examine a couple of different scenarios in which a class-scheduling algorithm may be used, and the constraints that are usually implemented when designing a timetable. Finally, we will build a simple class scheduler, which can be expanded to support more complex implementations.

In artificial intelligence, *class scheduling* is a variation of the constraint satisfaction problem. This category of problem relates to problems, which have a set of variables that need to be assigned in such a way that they avoid violating a defined set of constraints.

```
Project Code
```

```
public class Class { public String name; public int Students; public String
NameOfLecturer; public ClassVenu addVenues; public Subject(String argName,
String argNameofLecturer, int argStudents) { name = argName;
NameOfLecturer=argNameofLecturer; Students=argStudents; } public String
toString() { return name;}}
```

```
Code Block for Class Intialisation
```

```
public class Days extends Hours{ public Days(Subject a, Subject b,Subject c,
Subject d,Subject e, Subject f,Subject g,Subject h, Subject j) {int i=0;
TimeSlot[i]=a; TimeSlot[i+1]=b; TimeSlot[i+2]=c; TimeSlot[i+3]=d;
TimeSlot[i+4]=e; TimeSlot[i+5]=f; TimeSlot[i+6]=g; TimeSlot[i+7]=h;
TimeSlot[i+8]=j; }
```

`Schdeuling algorithms`

Personal Info: Hello everyone, I'm a computer science student; who (unfortunately) has to work on this NP-Complete problem for my Final Year Project. I am not so experienced in programming beyond my assignments and things I learned in University. So I apologies for any ignorant questions I might ask.

University Timetable Scheduling Project using Genetic Algorithm:

.

Regarding my methodology and the way that I want to create this application, I have decided to use two main Artificial Intelligence techniques to solve the problem; Genetic Algorithm and Constraint satisfaction. I came up with some rules and divided them in to two main groups; soft constraints and hard constraints. So the algorithm should work this way:

Assuming that we have our inputs (classrooms and their capacities, Subject names, lecturers and the amount of students signed up for them). The application starts with generating random timetables. However I decided to Implement my hard constraints,

one that cannot be ignored (such as having same lecturer teaching more than one classes at same time or classroom being occupied with more than one subject at any given time) at this stage. So basically we are generating random timetables but as we are creating them, we apply the hard constraints to them. I would like to call it guided randomly generated timetables. The next stage is to find the best possible timetable. Using the previous technique I talked about; We already know the generated timetables are acceptable in terms of hard constraints now using genetic algorithm I will find the best possible (most optimal) timetable and get that as the result. So soft constraints (such as having break between classes, avoiding same subject having more than one class everyday and such) would be used in my GA's fitness function.

Most important question: 1. How to create timetable? Here is my attempt on creating timetable and Review Generation:

```
public class Subject { public String name; public int Students; public String NameOfLecturer; public ClassVenu addVenues; public Subject(String argName, String argNameofLecturer, int argStudents) { name = argName; NameOfLecturer=argNameofLecturer; Students=argStudents; } public String toString() { return name;}}
```

```
public class ClassVenu { public String ClassName; public int ClassCapacity; public ClassVenu(int argClassCapacity, String argClassName) { ClassName = argClassName; ClassCapacity = argClassCapacity; } }
```

```
public class Days extends Hours{ public Days(Subject a, Subject b,Subject c, Subject d,Subject e, Subject f,Subject g,Subject h, Subject j) {int i=0; TimeSlot[i]=a; TimeSlot[i+1]=b; TimeSlot[i+2]=c; TimeSlot[i+3]=d; TimeSlot[i+4]=e; TimeSlot[i+5]=f; TimeSlot[i+6]=g; TimeSlot[i+7]=h; TimeSlot[i+8]=j; }
```

```
package fyp_small_timetable; public class Hours { public Subject [] TimeSlot = new Subject[9];

}
```

```
package fyp_small_timetable; public static void main(String[] args) { //creating Subjects Subject A = new Subject("Human Computer Interaction", "Batman" , 49); Subject B = new Subject("Artificial Intelligence", "Batman" , 95); Subject C = new Subject("Human Computer Interaction", "Batman" ,180);
Subject D = new Subject("Human Computer Interaction", "Batman" , 20); Subject
E = new Subject("Empty", "No-One", 0); //Creating Class Venue ClassVenu Class1
= new ClassVenu(100,"LecturerTheater1"); ClassVenu Class2 = new
ClassVenu(50,"LecturerTheater2"); ClassVenu Class3 = new
ClassVenu(200,"LecturerTheater3"); //Creating Days Days Day1 =new Days(A, A,
E, B, B, E, E, A, A); Days Day2 =new Days(C, C, E, D, D, E, E, A, A); Days
Day3 =new Days(E, E, E, B, B, E, E, A, A); Days Day4 =new Days(C, C, E, C, C,
E, E, A, A); Days Day5 =new Days(A, A, E, B, B, E, E, A, A); //creating
Timetable TimeTable T1 = new TimeTable(Day1, Day2, Day3, Day4, Day5);
List<Subject> answer = T1.ShowTimetable(T1); System.out.println(answer); } }
```

Section 3 – Testing(Junit Testing)

Below is my approach to write testcases for various scheduling test cases in Junit Testcases

```
public class ScheduleTest { Date mon = new Date(2014, 0, 5); Date tue = new
Date(2014, 0, 6); Date wed = new Date(2014, 0, 7); Date thu = new Date(2014,
0, 8); Date fri = new Date(2014, 0, 9); Date sat = new Date(2014, 0, 10); Date
sun = new Date(2014, 0, 11); Date firstOf2014 = new Date(2014, 0, 1); Date
lastOf2014 = new Date(2014, 11, 31); Date today =
GregorianCalendar.getInstance().getTime(); Date tomorrow = new
Date(today.getTime() + (1000L * 3600 * 24)); Date nextMonth = new
Date(today.getTime() + (1000L * 3600 * 24 * 31)); Date prevMonth = new
Date(today.getTime() - (1000L * 3600 * 24 * 31)); Schedule
scheduleTodayTomorrow = new Schedule(today, tomorrow); @Test public void
testDateStartBeforeDateStop(){ try{ scheduleTodayTomorrow = new
Schedule(tomorrow, today); Assert.fail("Should throw IllegalArgumentException
when dateStart after dateStop."); } catch (IllegalArgumentException ex){ //OK
} } @Test public void testDateStopCanBeNull(){ Schedule schedule = new
Schedule(today, null); } @Test public void testDefensiveCopy(){ long todayTime
= today.getTime(); today.setTime(1000); Assert.assertEquals(todayTime,
scheduleTodayTomorrow.getDateStart().getTime()); long tomorrowTime =
tomorrow.getTime(); tomorrow.setTime(1000); Assert.assertEquals(tomorrowTime,
scheduleTodayTomorrow.getDateStop().getTime()); } @Test public void
testDateValidity(){
Assert.assertTrue(scheduleTodayTomorrow.checkDateValid(today));
Assert.assertTrue(scheduleTodayTomorrow.checkDateValid(tomorrow));
Assert.assertFalse(scheduleTodayTomorrow.checkDateValid(prevMonth));
Assert.assertFalse(scheduleTodayTomorrow.checkDateValid(nextMonth)); Date
inDate = new Date(today.getTime() + 1000L);
Assert.assertTrue(scheduleTodayTomorrow.checkDateValid(inDate)); } @Test
public void testDailyRepetition(){ Schedule schedule = new Schedule(today,
nextMonth); schedule.setRepetitionType(Schedule.REPETITION.DAILY);
Assert.assertTrue(schedule.checkDateValid(today));
Assert.assertTrue(schedule.checkDateValid(tomorrow));
Assert.assertTrue(schedule.checkDateValid(nextMonth)); } @Test public void
testMonthRepetition(){ Schedule schedule = new Schedule(firstOf2014,
lastOf2014); schedule.setRepetitionType(Schedule.REPETITION.MONTHLY); // As
the scheduleTodayTomorrow starts on 1/1, then every first day of months should
be valid for (int i=1; i<=11; i++){ Date date = new Date(2014, i, 1);
Assert.assertTrue("Valid date with month " + i + " treated as invalid.",
schedule.checkDateValid(date)); } // Every other days of every month are
invalid for (int i=1; i<=11; i++){ Date date = new Date(2014, i, 2);
Assert.assertFalse("Invalid date with month " + i + " treated as valid.",
schedule.checkDateValid(date)); } } @Test public void
testWeekRepetitionNoValidDateAsDefault(){ Schedule schedule = new
Schedule(firstOf2014, lastOf2014);
schedule.setRepetitionType(Schedule.REPETITION.WEEKLY);
Assert.assertFalse(schedule.checkDateValid(mon));
Assert.assertFalse(schedule.checkDateValid(tue));
Assert.assertFalse(schedule.checkDateValid(wed));
Assert.assertFalse(schedule.checkDateValid(thu));
Assert.assertFalse(schedule.checkDateValid(fri));
Assert.assertFalse(schedule.checkDateValid(sat));
Assert.assertFalse(schedule.checkDateValid(sun)); } @Test public void
testWeekRepetitionAddMonday(){ Schedule schedule = new Schedule(firstOf2014,
lastOf2014); schedule.setRepetitionType(Schedule.REPETITION.WEEKLY);
schedule.addDayOfWeek(Schedule.DAY_OF_WEEK.MONDAY);
Assert.assertTrue(schedule.checkDateValid(mon));
Assert.assertFalse(schedule.checkDateValid(tue));
Assert.assertFalse(schedule.checkDateValid(wed));
Assert.assertFalse(schedule.checkDateValid(thu));
Assert.assertFalse(schedule.checkDateValid(fri));
Assert.assertFalse(schedule.checkDateValid(sat));
Assert.assertFalse(schedule.checkDateValid(sun)); } @Test public void
testWeekRepetitionAddTuesday(){ Schedule schedule = new Schedule(firstOf2014,
lastOf2014); schedule.setRepetitionType(Schedule.REPETITION.WEEKLY);
schedule.addDayOfWeek(Schedule.DAY_OF_WEEK.TUESDAY);
Assert.assertTrue(schedule.checkDateValid(tue));
Assert.assertFalse(schedule.checkDateValid(mon));
Assert.assertFalse(schedule.checkDateValid(wed));
Assert.assertFalse(schedule.checkDateValid(thu));
Assert.assertFalse(schedule.checkDateValid(fri));
Assert.assertFalse(schedule.checkDateValid(sat));
Assert.assertFalse(schedule.checkDateValid(sun)); } @Test public void
testWeekRepetitionAddWednesday(){ Schedule schedule = new
Schedule(firstOf2014, lastOf2014);
schedule.setRepetitionType(Schedule.REPETITION.WEEKLY);
schedule.addDayOfWeek(Schedule.DAY_OF_WEEK.WEDNESDAY);
Assert.assertTrue(schedule.checkDateValid(wed));
Assert.assertFalse(schedule.checkDateValid(mon));
Assert.assertFalse(schedule.checkDateValid(tue));
Assert.assertFalse(schedule.checkDateValid(thu));
Assert.assertFalse(schedule.checkDateValid(fri));
Assert.assertFalse(schedule.checkDateValid(sat));
Assert.assertFalse(schedule.checkDateValid(sun)); } @Test public void
testWeekRepetitionAddThursday(){ Schedule schedule = new Schedule(firstOf2014,
lastOf2014); schedule.setRepetitionType(Schedule.REPETITION.WEEKLY);
schedule.addDayOfWeek(Schedule.DAY_OF_WEEK.THURSDAY);
Assert.assertTrue(schedule.checkDateValid(thu));
Assert.assertFalse(schedule.checkDateValid(mon));
Assert.assertFalse(schedule.checkDateValid(tue));
Assert.assertFalse(schedule.checkDateValid(wed));
Assert.assertFalse(schedule.checkDateValid(fri));
Assert.assertFalse(schedule.checkDateValid(sat));
Assert.assertFalse(schedule.checkDateValid(sun)); } @Test public void
testWeekRepetitionAddFriday(){ Schedule schedule = new Schedule(firstOf2014,
lastOf2014); schedule.setRepetitionType(Schedule.REPETITION.WEEKLY);
schedule.addDayOfWeek(Schedule.DAY_OF_WEEK.FRIDAY);
Assert.assertTrue(schedule.checkDateValid(fri));
Assert.assertFalse(schedule.checkDateValid(mon));
Assert.assertFalse(schedule.checkDateValid(tue));
Assert.assertFalse(schedule.checkDateValid(wed));
Assert.assertFalse(schedule.checkDateValid(thu));
Assert.assertFalse(schedule.checkDateValid(sat));
Assert.assertFalse(schedule.checkDateValid(sun)); } @Test public void
testWeekRepetitionAddSaturday(){ Schedule schedule = new Schedule(firstOf2014,
lastOf2014); schedule.setRepetitionType(Schedule.REPETITION.WEEKLY);
schedule.addDayOfWeek(Schedule.DAY_OF_WEEK.SATURDAY);
```

Section 5 – References

These are various sources used in the project

1. https://codereview.stackexchange.com/questions/46282/schedule-class-exercise-for-test-driven-development
2. https://javahungry.blogspot.com/2013/04/scheduling-algorithm-first-come-first.html

Section 6 – Appendix