**SE401: Software Quality Assurance & Testing**

**Activity 7.1: Black Box Testing**

The activity is about a class called WorkSchedule. The purpose of the class is to manage the schedule for the employees of a company. The time is divided in units of 1 hour and the hours are simply identified by integers (Note that this is an unrealistic simplification). For each hour, the schedule stores the number employees which is needed at that time. It could for instance be that during week-days the need is that 5 employees work at the same time, but less during nights and weekend. For each hour the schedule also stores the names of the employees who have been assigned to work at that hour.

The constructor and methods of the class are:

**public** WorkSchedule(**int** size) { ... }

creates a schedule which contains the hours 0,1,2,...,size - 1 where for each hour the number of needed employees is set to zero and there are no employees assigned to it

**public** Hour readSchedule(**int** time) { ... }

gives back an object of the class Hour, which has two fields:  
requiredNumber of type int is the required number of employees working at hour time.  
workingEmployees of type String[] is the names of the employees who have **so far** been assigned to work at hour time.

**public** **void** setRequiredNumber(**int** nemployee, **int** starttime, **int** endtime) { ... }

sets the number of required working employees to nemployee for all hours in the interval starttime to endtime.

**public boolean** addWorkingPeriod(String employee, **int** starttime, **int** endtime) { ... }

schedules employee to work during the hours from starttime to endtime.

**public** String[] workingEmployees(**int** starttime, **int** endtime) { ... }

returns a list of all employees working at some point during the interval starttime to endtime.

**public** **int** nextIncomplete(**int** currenttime) { ... }

returns the closest time starting from currenttime for which the required amount of employees has not yet been scheduled.

The constructor (WorkSchedule) and the methods readSchedule, setRequiredNumber, nextIncomplete, are trusted to be correct and do not need to be tested. Note that readSchedule can be used to examine the state of a class instance.

A compiled implementation of the class can be found in the following Java archive:

> WorkSchedule.jar

**Your task:** For each specification provided below (i.e. specification for the methods addWorkingPeriod, and workingEmployees):

1. Specify (the domain and) the input space for each argument to the method (including the object of that method).
2. Divide this input space into partitions, based on the specification.
3. Implement at least one test case for each partition, and possibly additional border cases.,
4. For each test case, give a comment which describes what partition it tests.

**a) addWorkingPeriod**  
*requires:*  
  employee is a non-null string  
*ensures:*  
  if starttime < 0 or endtime >= size or starttime > endtime then  
    returns false and the schedule is unchanged  
  otherwise  
    if for any hour in the interval starttime to endtime the length of workingEmployees is equal to requiredNumber then  
      returns false and the schedule is unchanged  
    otherwise  
      if for any hour in the interval starttime to endtime there is a string in workingEmployees which equal employee then  
        returns false and the schedule is unchanged  
      otherwise  
        returns true,  
        for i between starttime and endtime, workingEmployees contain a string equal to employee and  
        the rest of the schedule is unchanged

**b) workingEmployees**  
*requires:*  
  starttime >= 0 and endtime < size  
*ensures:*  
  if starttime <= endtime then  
    returns an array with distinct strings -- a string appears in the return array if and only if  
    it appears in the workingEmployees of at least one hour in the interval starttime to endtime  
  otherwise  
    returns an empty array  
  and in either case the schedule is unchanged

**Reporting test cases**  
  
In a report file, you **must** explain and motivate why you divided the test cases in the way you did. You **must** show how you defined the input space, and how you partitioned it.

**Example:** Given the method:

public String m(int x);

With the specification:

requires: x != 0

ensures: if x > 0 then

the method returns the String "High"

otherwise

the method returns the String "Low"

Your report should read something like:

Input space: x != 0 (or: "x" in { MIN\_INT, ..., -1, 1, ..., MAX\_INT } )

Partition #1: x > 0 (or: "x" in { 1, ..., MAX\_INT } )

test\_m\_part1: input: "x = 155", expected: "m(x) = "High""

Partition #2: x < 0 (or: "x" in { MIN\_INT, ..., -1 } )

test\_m\_part2: input: "x = -18", expected: "m(x) = "Low""

Border cases:

test\_m\_border1: input: "x = MIN\_INT", expected: "m(x) = "Low""

test\_m\_border2: input: "x = -1", expected: "m(x) = "Low""

test\_m\_border3: input: "x = 1", expected: "m(x) = "High""

test\_m\_border4: input: "x = MAX\_INT", expected: "m(x) = "High""

Found bugs: None.