**Homework T2 Submission Document**

You can work individually or in groups of 2.

|  |  |  |  |
| --- | --- | --- | --- |
| Names: | Joanne Wardell |  | KD Adkins |

**Test Derivation for *addEmp(e:Employee)***

We will use the category-partition method to derive a set of test cases for this method.

1. **Identify what criteria must hold true for each test case to pass**

The side-effects of this method are:

* numEmps becomes incremented by 1 if the operation succeeds in adding an employee to the store
* the getNumEmps method will return a different value if the operation succeeds in adding the employee
* the getTotalHours method could return a different value depending on the state of the Employee which might be added
* the getTotalPay method could return a different value depending on the state of the Employee which might be added
* the removeEmployee method could behave differently if the Employee is added successfully

1. **Identify inputs/parameters for each feature under test.**

The input parameters are:

* primitive data type numEmps: integer
  + the number of employees that a currently in the store
* reference data type emps: Employee[]
  + the container in which the emps are collected
* reference data type e: Employee
  + the employee that is trying to be added
* primitive data type added: Boolean
  + the indication of success or failure upon add attempt

1. **Identify the categories/characteristics for each input/parameter.**

Categories of numEmps:

* value

Categories of emps:

* length
* content

Categories of e:

* state

Categories of added:

* value

1. **Partition categories into choices**

The choices for the numEmps parameter are *positions remaining = PR*  and *no positions remaining = P*.

* PR – numEmps ∈ [0,19]
* P – numEmps = 20

The choices for the emps parameter are *full = F* and *not full = NF*.

* NF – emps array has indices [0,19] available for use
* F – emps array has no indices available for use

The choices for the e parameter are *null = N* and *non-null = NN*.

* NN – e is a reference to an Employee object
* N – e is the null data type
* The choices for the added parameter are *success = S* and *fail = FA*.

1. **Number of tests before constraints added**

|  |  |  |  |
| --- | --- | --- | --- |
| Input | Category | Choices | Coded Choices |
| numEmps | Value | Positions Remaining, No Positions Remaining | PR,P |
| emps | Length | Full, Not Full | F,NF |
| e | State | Null, Not Null | N,NN |
| added | Value | Success, Fail | T,FA |

By the counting principle, there are 2\*2\*2\*2 = 16 possible combinations before constraints are added.

PRFNT, PRFNFA, PRFNNT, PRFNNFA,

PRNFNT, PRNFNFA, PRNFNNT, PRNFNNFA,

PFNT, PFNFA, PFNNT, PFNNFA,

PNFNT, PNFNFA, PNFNNT, PNFNNFA

The following combinations are removed because they are invalid:

PRFNT – This means that there are no positions remaining but the employee was added successfully.

PFNT– This means that there are no positions remaining but the employee was added successfully.

PNFNT – This means that there are no positions remaining but the employee was added successfully.

PFNNT– This means that there are no positions remaining but the employee was added successfully.

PRFNT – This means that the array is full but there are positions remaining and the employee was added successfully.

PRFNNT – This means that the array is full but there are positions remaining and the employee was added successfully.

PRNFNT – This means that the array is full but there are positions remaining and the employee was added successfully.

This leaves 9 combinations.

1. **TSL Input**

The *TSL* input file is shown below.

NumEmps:

PositionsRemaining.

NoPositionsRemaining. [property max]

Emps:

Full. [property full]

NotFull.

E:

Null. [property null]

NotNull.

Added:

Success. [if !full && !max && !null]

Fail.

Justification for the constraints:

|  |  |
| --- | --- |
| **Constraint** | **Justification** |
| Employees can only be added if there is room in the emps array and if the number of current employees is less than 20. | Employees can only be added if the emps array has indices available for use and numEmps is suggests that positions are remaining. |
| Only references to Employee objects can be added, no null data can be added. | Null objects are not handled, can cause null pointer exceptions, and cause failures. |

1. **TSL Output – Test Frames**

The 9 test frames are show below:

Test Case 1 (Key = 1.1.1.2.)

NumEmps : PositionsRemaining

Emps : Full

E : Null

Added : Fail

Test Case 2 (Key = 1.1.2.2.)

NumEmps : PositionsRemaining

Emps : Full

E : NotNull

Added : Fail

Test Case 3 (Key = 1.2.1.2.)

NumEmps : PositionsRemaining

Emps : NotFull

E : Null

Added : Fail

Test Case 4 (Key = 1.2.2.1.)

NumEmps : PositionsRemaining

Emps : NotFull

E : NotNull

Added : Success

Test Case 5 (Key = 1.2.2.2.)

NumEmps : PositionsRemaining

Emps : NotFull

E : NotNull

Added : Fail

Test Case 6 (Key = 2.1.1.2.)

NumEmps : NoPositionsRemaining

Emps : Full

E : Null

Added : Fail

Test Case 7 (Key = 2.1.2.2.)

NumEmps : NoPositionsRemaining

Emps : Full

E : NotNull

Added : Fail

Test Case 8 (Key = 2.2.1.2.)

NumEmps : NoPositionsRemaining

Emps : NotFull

E : Null

Added : Fail

Test Case 9 (Key = 2.2.2.2.)

NumEmps : NoPositionsRemaining

Emps : NotFull

E : NotNull

Added : Fail

1. **Test Cases**

The test cases we derived are shown below.

PRFNFA, PRFNNFA, PRNFNFA, PRNFNNT, PRNFNNFA, PFNFA, PFNNFA, PNFNFA, PNFNNFA

Notice that the highlighted cases are not valid for logical reasons.

PRFNFA – It is impossible for there to be positions remaining with the emps array being full.

PRFNNFA – It is impossible for there to be positions remaining with the emps array being full.

PRNFNNFA – If there are positions remaining and the emp is not null, then it should be successfully added.

PNFNFA – It is impossible for there to be no positions remaining while the emps array is not full.

PNFNNFA – It is impossible for there to be no positions remaining while the emps array is not full.

We are left with 4 test cases.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test** | **Value [PR / P]** | **Length [F / NF]** | **State [N / NN]** | **Value [T / FA]** |
| 1 | PR | NF | N | FA |
| 2 | PR | NF | NN | T |
| 3 | P | F | N | FA |
| 4 | P | F | NN | FA |

**Test Derivation for *removeEmployee(pos:int):Employee***

We will use the category-partition method to derive a set of test cases for this method.

1. **Identify what criteria must hold true for each test case to pass**

The side-effects of this method are:

1. **Identify inputs/parameters for each feature under test.**
2. **Identify the categories/characteristics for each input/parameter.**
3. **Partition categories into choices**
4. **Number of tests before constraints added**
5. **TSL Input**

The *TSL* input file is shown below.

Justification for the constraints:

|  |  |
| --- | --- |
| **Constraint** | **Justification** |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

1. **TSL Output – Test Frames**

The \_\_ test frames are show below:

1. **Test Cases**

The test cases we derived are shown below.

[Remove unused columns]

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Test** | **[Put name of characteristic 1 here]** | **[Put name of characteristic 2 here]** | ***Etc.*** |  |  |  |  |  |
| 1 |  |  |  |  |  |  |  |  |
| 2 |  |  |  |  |  |  |  |  |
| 3 |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |

Following the 6 Steps of the CPM:

1. The feature being tested is the AddEmp method, which can be abstractly thought of as the algorithmic feature responsible for “adding an Employee object to the Store”.
2. The **input parameters** that this algorithm requires are the Employee object which is trying to be added to the Store, the array used to contain Employees, and the numEmps variable representing the number of Employees that are “currently in the Store”. For each input-parameter, we will describe the **characteristics**. So, the most simple input is the numEmps variable. It is an integer so it ranges from -232 to 232 , but we will only use a subset of those values, namely, values from [0, 20]. In addition, another characteristic of the integer data type is that it does not contain fractional parts, so it is a whole number. So, our range is from zero to twenty inclusive and is discrete not continuous. Next we can talk about the data structure holding the Employees, the array. It is an array so that means it’s size is immutable. It’s size will always be 20. The array is an Object so it is a reference type. In addition, the array is of type Employee. The array contains data of type Object, so it contains references, not primitive data. It contains references to Employee objects. Something interesting to note is that the array is sensitive because it’s size cannot change. The array can get full. The array can only have indices from 0 to 19. So any time the numEmps variable is used in conjuction with accessing this array or using it somhow, it will need to be decriminted by a discrete amount ranging from [1-19]. Any negative index will not work. The array will have indices of interest: some indices will point to elements which are references to Employee objects. Some indices will point to elements that contain references to null objects. The special incides are always subject to change because emps can be added, emps can be removed, emps can be overwritten with emps, emps can be overwritten with null. If I add an emp, then the interesting indices contain one more element, namely, one larger than the previously last element. Since we’ve touched base on the array, we can talk about the Employee data type which is also an input parameter. The Employee data type is a subclass of Object. It is a reference type which means that any variable that I make for this Employee object is a reference to the Employee’s references to primitive data type. AKA, the Employee object is a reference to a group of primitive data types and other reference types. The Employee object has attributes which means that it has state. The employee’s state can vary from instant to instant. The state of the employee is initialized upon construction [instantiation] of the Employee object. The employee is not an immutable object because it has setters and behaviors that can alter its state. There are no conditions that prohibit when these behaviors/actions can be performed so the state of an employee is subject to change at any time. So, the state of the employee depends on the behaviors that it performs. The state of the employee is also subject to change depending on actions that the store class performs on the Employee object that can change it’s state. The state of the employee is changed when the newWeek method is called. The hours array is reset to all zeros. The state of the employee is changed when the mergeEmployee method is called because the array’s values are altered because they are combined with another employee’s hours. The state of the employee is changed when the set hours method is called because the values in the hours array are overwritten in the specified index with the specified hours. From what I can tell, the Store cannot alter the state of the Employee. Why: because the Employee class does not have an association with the store. So, even though the Emp’s position, for example, is changed when the store adds/rems employees, the emp is oblivious to this since it does not contain a reference to the store class. ….. Those are just a few characteristics of the Employee input parameter. Now we need to establish the **categories** in which the three input-parameters belong. The three parameters belong to distinct categories because the CPM argues that input parameters can be separated into categories. We will first identify the categories to which the numEmps integer can belong to. The categories are only consisting of values that are in the input domain of relevant cases. So, the int numEmps can be a positive whole number from 0 to 20 inclusive. We will split this range into a variety of categories. By inspection, three distinct categories are emerging for the numEmps variable. 1: if numEmps = 0, then there are no employees in the store. This is a distinct case because ………. there is nothing to work with so other meathods which are expecting emps and attributes/state from them will be affected…. Another category for the numEmps variable is if numEmps = [1,19). This case might also include 20 for the value of numEmps. Another reason why this is an important category is because the emps array only contains references to null objects, this meanst that none of the indices are special and none of them will result in the return / retrieval of a non-null object. This could cause some compoennents/behaviors of the system to fail and could lead to run time errors/excpetions. This is a distinct case in and of itself and from numEmps = 0 because there are now at least one Employees in the store meaning that there is at least some state/emp-data to play with, for the system to use, for the store/methods to perform calculations on. Now, the emps array contains at least one reference to a non null object, so there are at least one interesting/special indices in the emps array meaning that it is less likely to use an index which leads to a null object. The final distinct category for the numEmps parameter is the category where numEmps = 20. This is a distinct category because it means that the array is full. This means that if another emp wants to be added, an emp must be overwritten with another emp, an emp must be removed, or an emp must be overwritten by a null. Also, it could maybe cause the array index to go out of bounds.? probably not. The next parameter to create categories for is the hours array. The hours array varies in the number of Employees that it contains and the state of the Employees. The hours array mainly is concerned with the number of hours that it contains indirectly through the Employees that it contains. The hours array can be split into three categories: empty, non-empty, and full. Empty category: there are only null references contained in the array, Non-Empty category: there is one or more references to Employee objects contained in the array, Full category: there are exactly twenty references to Emps contained in the array. … why do we care.. Finally, the Employee parameter can be split into categories. Where the add emp method is concerned, the algorithm does not depend on the state of the emp. It only requires that the emp is not null. So, the emp parameter can be divied into to categories null or non-null. This categorization might be trivial.