**Testing Example**

***Category-Partition Method***

This example will use the spirit of ECT and CPM to develop several unit tests.

**Problem Statement**

Consider this problem from CS 1302. We will construct unit tests for the constructor and the *getPay* method.

|  |
| --- |
| You will write a class named *Employee* that keeps track of the hours worked on each day of the week. The class will have the following members:1. hours – a double array with 7 elements. This holds the hours worked on each day of the week. Monday (index=0) is the first day of the week.
2. name – a string containing the name of the employee
3. payRate – a double which represents the pay rate ($/hr)
4. Employee(name:String, payRate:double) – constructor that accepts a name for the employee and their pay rate ($/hr). The name must have a length greater than zero and the pay rate must be greater than zero; otherwise, a *RuntimeException* is thrown.
5. getHours(i:int) – returns the number of hours worked on day i.
6. getName – returns the name
7. getNumDaysWorked – returns the number of days worked.
8. getPay – returns the total pay for the week computed in the following way:
* Weekday hours (Mon-Fri) are paid at the pay rate. Any hours over 40 during weekdays are paid at time-and-a-half.
* Weekend hours are paid at double-time, no matter whether the workday hours exceed 40.
* Working 7 consecutive day earns a bonus of $50.00, no matter how many total hours worked.
1. getPayRate – returns the pay rate
2. getTotalHours – returns the total number of hours worked for the week
3. getWeekdayHours – returns the total number of hours worked during weekdays (Mon-Fri)
4. getWeekendHours – returns the total number of hours worked during the weekend (Sat-Sun)
5. mergeEmployee – accepts an *Employee* object and merges the hours. You can assume the input employee has the same name and pay rate. For example if *e1* had hours: [8,8,8,2,0,0,0] and *e2* had hours: [0,0,0,4,10,0,0]. Then, when *e1.mergeEmployee(e2)* is excecuted, the *e1* will have hours: [8,8,8,6,10,0,0].
6. newWeek – starts the week over by setting 0 hours for each day
7. setHours(i:int,num:double) – sets the number of hours worked on day i.
8. toString – returns a string that is formatted like this…
 |

**Test Derivation for *Employee(name:String, payRate:double)***

1. **Identify independently testable feature (use case, method)**

We will test the *Employee* constructor.

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

There are two parameters: *name* and *payRate.*

1. **Identify the categories/characteristics for each input/parameter.**
2. **Partition categories into choices**
3. **Identify constraints among choices**

Here, we combine these three steps as the situation is straight forward. There would be 6 (2\*3) tests before adding the constraints. With the constraints there will be 4 tests.

|  |  |  |  |
| --- | --- | --- | --- |
| **Input** | **Category** | **Choice** | **Constraint** |
| *name* | length | =0 | Error |
|  |  | >0 |  |
| *payRate* | Value | <0 | Error |
|  |  | =0 | Error |
|  |  | >0 |  |

1. **Produce and evaluate test case specifications (frames) using TSL as needed.**

We did not use the TSL program as we could easily specify the four tests.

|  |  |  |  |
| --- | --- | --- | --- |
| Num | Name | Pay rate | Constraint |
| 1 | =0 | >0 | Error |
| 2 | >0 | <0 | Error |
| 3 | >0 | =0 | Error |
| 4 | >0 | >0 |  |

1. **Test Cases**

The four test cases are shown below.

|  |  |  |  |
| --- | --- | --- | --- |
| Num | Name | Pay rate | Expected Result |
| 1 | “” | 50.0 | Exception |
| 2 | “Markus” | -10.0 | Exception |
| 3 | “Markus” | 0.0 | Exception |
| 4 | “Markus” | 50.0 | Getters produce correct result |

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

The side-effects of this method are:

1. The *name* and *payRate* are initialized if the *payRate* is greater than zero and the *name* length is greater than zero. Otherwise, a *RuntimeTimeException* is thrown.
2. **JUnit Tests**

@DisplayName("Constructor: Empty string")

@Test

**void** testConstructorEmptyString() {

 Assertions.*assertThrows*(RuntimeException.**class**, () -> {**new** Employee("",50.0);});

}

@DisplayName("Constructor: Payrate <0")

@Test

**void** testConstructorPayrateLess0() {

 Assertions.*assertThrows*(RuntimeException.**class**, () -> {**new** Employee("Markus",-10.0);});

}

@DisplayName("Constructor: Payrate =0")

@Test

**void** testConstructorPayrateEqual0() {

 Assertions.*assertThrows*(RuntimeException.**class**, () -> {**new** Employee("Markus",0.0);});

}

@DisplayName("Constructor: saves instance vars")

@Test

**void** testConstructorNormal() {

 Employee e = **new** Employee("Markus", 50.0);

 *assertEquals*("Markus",e.getName());

 *assertEquals*(50.0,e.getPayRate());

}

**Test Derivation for *getPay():double***

1. **Identify independently testable feature (use case, method)**

We will use the category-partition method to derive a set of test cases for this method. We note that there are four sources of pay that an employee can receive.

1. Regular Pay(R) – hours worked during the week whose total number was 40 or less
2. Over-time Pay(O) – hours worked during the week in excess of 40 hours
3. Double-time Pay(D) – hours worked during the weekend
4. 7-Day Bonus Pay(7) – worked all 7 days in a week.

There are six different scenarios, that as we derive the tests we need to make sure are covered:

1. R
2. R,O
3. R,D
4. R,O,D
5. R,D,7
6. R,O,D,7
7. **Identify inputs/parameters for each feature under test.**

Although there are no formal parameters for this method, there are two effective parameters:

1. The *hours* array which is an instance variable.
2. The *payRate*

However, *payRate* is not really a factor since we already tested that it is greater than zero. Thus, a single pay rate will suffice for testing, say $10/hour.

1. **Identify the categories/characteristics for each input/parameter.**
2. **Partition categories into choices**

We combine steps 3 and 4. It was decided that there are three characteristics that should be considered for the input. The choices and justification are shown in the table below. It should be noted that the number of hours worked during the weekend is not a characteristic because the number of days worked during the weekend (WEDW) is sufficient. In other words, any work during the weekend receives double-time pay.

|  |  |  |  |
| --- | --- | --- | --- |
| **Input** | **Category/Characteristic** | **Choices** | **Comment** |
| *hours* | Number of hours worked during the week, *e.g.* Monday-Friday (WDH) | <40 | Regular pay only, no overtime |
|  |  | =40 | Regular pay only, no overtime, boundary case |
|  |  | >40 | Regular pay and overtime |
|  | Number of days worked during the week, *e.g.* Monday-Friday (WDW) | 0 | No days worked during the week |
|  |  | 4 | Worked during the week, but not all days. Could have chosen 1,2, or 3 instead of 4. |
|  |  | 5 | Worked all week days |
|  | Number of days worked during the weekend, *e.g.* Saturday & Sunday (WEDW) | 0 | No days worked during weekend, thus no double-time nor 7-day bonus |
|  |  | 1 | 1 day worked during weekend, thus double-time, but no 7-day bonus |
|  |  | 2 | Both weekend days worked, thus double-time and possible 7-day bonus |

There are 3 choices for each of the three characteristics. Thus, there are a total of 27 (3\*3\*3) tests specified before constraints are added.

1. **Identify constraints among choices**

The *TSL* input file is shown below.

#Week Days Worked

WDW:

 0. [property WDW=0]

 4.

 5. [property WDW=5]

#Weekend Days Worked

WEDW:

 0. [property WEDW=0]

 1.

 2. [if WDW=5]

#Week Day Hours

WDH:

 <40.

 =40. [if WDW=5 && WEDW=0]

 >40. [if !WDW=0]

Justification for the constraints:

|  |  |
| --- | --- |
| **Constraint** | **Justification** |
| WEDW=2 [if WDW=5] | It is sufficient to test working both weekend days only when the employee has worked all 5 workdays as this will apply the 7-day bonus.  |
| WDH=40 [if WDW=5 && WEDW=0] | This is a border case. The result of this constraint is the same as adding [single], which was the desire. However, by specifying this constraint we make it explicit exactly the condition we want to test when an employee works exactly 40 hours during the week. In this case, we choose achieving those 40 hours over all 5 days during the week and no weekend work. |
| WDH>40 [if !WDW=0] | This eliminates an impossible case: working more than 40 hours during the week, but not working any days during the week. |

1. **Produce and evaluate test case specifications (frames) using TSL as needed.**

The 13 test frames are show below:

|  |  |  |
| --- | --- | --- |
| Test Case 1 (Key = 1.1.1.) WDW : 0 WEDW : 0 WDH : <40Test Case 2 (Key = 1.2.1.) WDW : 0 WEDW : 1 WDH : <40Test Case 3 (Key = 2.1.1.) WDW : 4 WEDW : 0 WDH : <40Test Case 4 (Key = 2.1.3.) WDW : 4 WEDW : 0 WDH : >40Test Case 5 (Key = 2.2.1.) WDW : 4 WEDW : 1 WDH : <40Test Case 6 (Key = 2.2.3.) WDW : 4 WEDW : 1 WDH : >40Test Case 7 (Key = 3.1.1.) WDW : 5 WEDW : 0 WDH : <40 |  | Test Case 8 (Key = 3.1.2.) WDW : 5 WEDW : 0 WDH : =40Test Case 9 (Key = 3.1.3.) WDW : 5 WEDW : 0 WDH : >40Test Case 10 (Key = 3.2.1.) WDW : 5 WEDW : 1 WDH : <40Test Case 11 (Key = 3.2.3.) WDW : 5 WEDW : 1 WDH : >40Test Case 12 (Key = 3.3.1.) WDW : 5 WEDW : 2 WDH : <40Test Case 13 (Key = 3.3.3.) WDW : 5 WEDW : 2 WDH : >40 |

1. **Generate test cases from test case specifications which should be validated and verified.**

You will do this for your homework: HW T1

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

The side-effects of this method are:

The method returns the correct pay for given input.