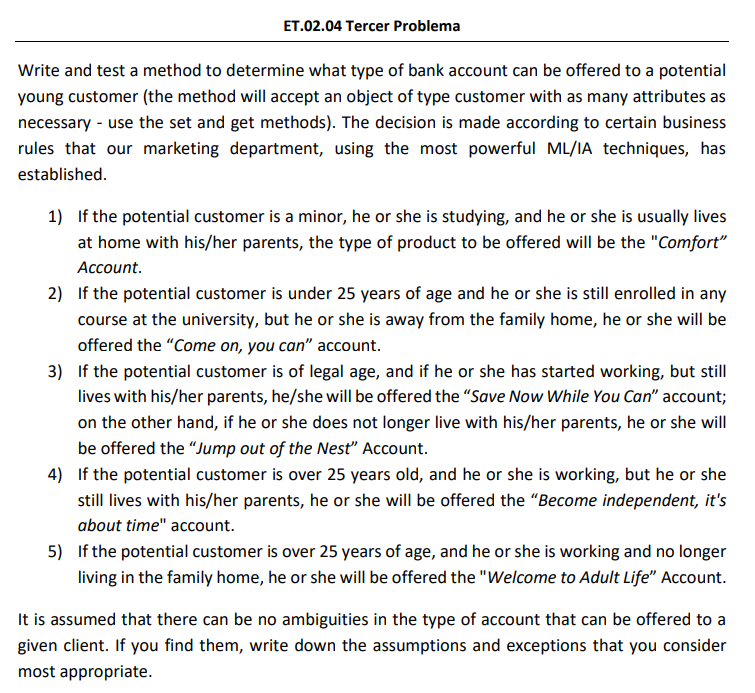
**Theoretical Exercise - Problem #3**

**Distribution of work**

*Pseudocode: Team 3: Marvin*

*Testing: Team 1: Alejandro, Alejandro*

**0 - Problem Description**



**1- Write, at least the pseudocode of the identified method.**

| import java.time.LocalDate;  class BirthDate {  private int year, month, day;  public BirthDate(int year, int month, int day) {  setYear(year);  setMonth(month);  setDay(day);  }  public int getAge() {  int currentYear = LocalDate.now().getYear();  int currentMonth = LocalDate.now().getMonth().getValue();  int currentDay = LocalDate.now().getDayOfMonth();  // Birthday of person within the current year already over  if (getMonth() > currentMonth ||  getMonth() == currentMonth && getDay() >= currentDay) {  return currentYear - getYear();  }  // Birthday of person within the current year not yet over  else  return currentYear - getYear() - 1;  }  public void setYear(int year) {  if (year >= 1800 && year <= LocalDate.now().getYear())  this.year = year;  else  throw new IllegalArgumentException("Illegal value");  }  public void setMonth(int month) {  if (month >= 1 && month <= 12)  this.month = month;  else  throw new IllegalArgumentException("Illegal value");  }  public void setDay(int day) {  if (day >= 1 && day <= 31)  this.day = day;  else  throw new IllegalArgumentException("Illegal value");  }  public int getYear() {  return year;  }  public int getMonth() {  return month;  }  public int getDay() {  return day;  }  } |
| --- |

| class Customer {  // as many attributes as possible  private BirthDate birthDate;  private boolean isStudent;  private boolean isWorking;  private boolean permanentResidencyWithParents;  public Customer(BirthDate birthDate, boolean isStudent, boolean isWorking, boolean permanentResidencyWithParents) {  setBirthDate(birthDate);  setStudent(isStudent);  setWorking(isWorking);  setPermanentResidencyWithParents(permanentResidencyWithParents);  }  public void setBirthDate(BirthDate birthDate) {  if (birthDate == null)  throw new IllegalArgumentException("Argument needs to be initialized");  this.birthDate = birthDate;  }  public void setStudent(boolean student) {  isStudent = student;  }  public void setWorking(boolean working) {  isWorking = working;  }  public void setPermanentResidencyWithParents(boolean permanentResidencyWithParents) {  this.permanentResidencyWithParents = permanentResidencyWithParents;  }  public boolean isStudent() {  return isStudent;  }  public boolean isWorking() {  return isWorking;  }  public boolean isPermanentResidencyWithParents() {  return permanentResidencyWithParents;  }  public BirthDate getBirthDate() {  return birthDate;  }  } |
| --- |

| enum BankAccountType {  Comfort, ComeOnYouCan, SaveNowWhileYouCan, JumpOutOfTheNest,  BecomeIndependentItsAboutTime, WelcomeToAdultLife  } |
| --- |

| public class BankLogic {  public static BankAccountType determineAccountTypeYoungCustomer(Customer c) {  // (1) is minor, studying, residency with parents -> Comfort account  if (c.getBirthDate().getAge() < 18 &&  c.isStudent() &&  c.isPermanentResidencyWithParents())  return BankAccountType.Comfort;  // (2) under 25, studying, residency not with parents -> ComeOnYouCan account  else if (c.getBirthDate().getAge() < 25 &&  c.isStudent() &&  !c.isPermanentResidencyWithParents())  return BankAccountType.ComeOnYouCan;  // (3.1) 18 and above, working, residency with parents -> SaveNowWhileYouCan account  else if (c.getBirthDate().getAge() >= 18 &&  c.getBirthDate().getAge() < 25 &&  c.isWorking() &&  c.isPermanentResidencyWithParents())  return BankAccountType.SaveNowWhileYouCan;  // (3.2) 18 and above, working, residency not with parents -> SaveNowWhileYouCan account  else if (c.getBirthDate().getAge() >= 18 &&  c.getBirthDate().getAge() < 25 &&  c.isWorking() &&  !c.isPermanentResidencyWithParents())  return BankAccountType.JumpOutOfTheNest;0  // (4) 25 and above, working, residency with parents -> BecomeIndependentItsAboutTime account  else if (c.getBirthDate().getAge() >= 25 &&  c.isWorking() &&  c.isPermanentResidencyWithParents())  return BankAccountType.BecomeIndependentItsAboutTime;  // (5) 25 and above, working, residency not with parents -> WelcomeToAdultLife account  else if (c.getBirthDate().getAge() >= 25 &&  c.isWorking() &&  !c.isPermanentResidencyWithParents())  return BankAccountType.WelcomeToAdultLife;  // This code is only executed, if the customer is NOT applicable for a Young Account  throw new IllegalArgumentException(“Account is not applicable for a young account.”);  }  } |
| --- |

**2 - Identify the variables that must be considered to test the method.**

In this problem we have our result with 4 inputs, one of them being a number (integer) and the other three are a true or false statement (boolean).

| c.getBirthDate().getAge() : int |
| --- |
| c.isStudent() : boolean |
| c.isPermanentResidencyWithParents() : boolean |
| c.isWorking() : boolean |

These input variables are going to be the main focus of our testing and are found in the BankLogic class.

**3 - Identify the test values for each one of the variables previously identified, specifying the technique used to obtain each of those values).**

| Parameter | Equivalence Class | Values for equivalence class | Boundary values (heavy variant) | Error guessing values |
| --- | --- | --- | --- | --- |
| c.getBirthDate().getAge() | (-∞,0) [0,18) [18,25) [25,+∞) | -8, 5, 20, 40 | -1, 0, 1, 17, 18, 19, 24, 25, 26 | -1000  1000 |
| c.isStudent() | True/False | True/False | - | - |
| c.isPermanentResidencyWithParents() | True/False | True/False | - | - |
| c.isWorking() | True/False | True/False | - | - |

**4 - Calculate the maximum possible number of test cases that could be generated from the test values.**

In order to calculate the maximum number of test cases, we consider that an input to this problem (c.getBirthDate().getAge()), is the only one needed in this decision. Because it is the only one that can really change. So we are going to consider the following values: (-1, 0, 1, 17, 18, 19, 24, 25, 26, -1000, 1000).

So the possible amount of combinations is: 11 \* 2 \* 2 \* 2 = 88.

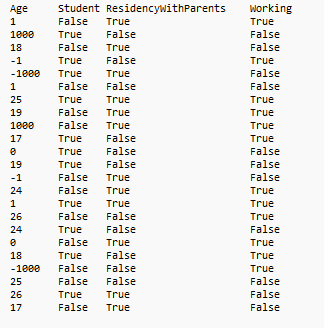
**5 - Define some test suites using each use**

In this step we want to check if every operation works, if the person who wants the account is valid and if it is the type of the account.

| **Test suit** | **Variables** |
| --- | --- |
| 1 | c.getBirthDate().getAge() = 17  c.isStudent() = true  c.isPermanentResidencyWithParents() = true  c.isWorking() = false |
| 2 | c.getBirthDate().getAge() = 19  c.isStudent() = true  c.isPermanentResidencyWithParents() = false  c.isWorking() = false |
| 3 | c.getBirthDate().getAge() = 19  c.isStudent() = false  c.isPermanentResidencyWithParents() = true  c.isWorking() = true |
| 4 | c.getBirthDate().getAge() = 24  c.isStudent() = false  c.isPermanentResidencyWithParents() = false  c.isWorking() = true |
| 5 | c.getBirthDate().getAge() = 26  c.isStudent() = false  c.isPermanentResidencyWithParents() = true  c.isWorking() = true |
| 6 | c.getBirthDate().getAge() = 26  c.isStudent() = false  c.isPermanentResidencyWithParents() = false  c.isWorking() = true |

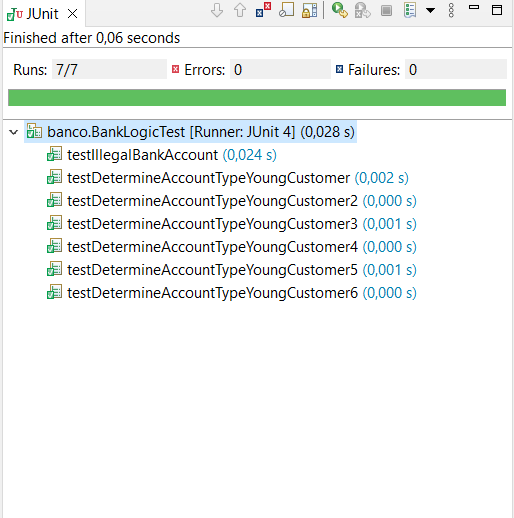
**6 - Define test suits to achieve pairwise coverage by using the proposed algorithm in lectures. You can check the results by means of the software PICT1**

As we had a total combination of 88 possible cases in order to get the pairwise coverage we used the tool PICT from Microsoft. In the end we finished with 23 different test suits.



**7 - For code snippets that include decisions, propose a set of test cases to achieve coverage of decisions.**

To create the requested set of test cases, we can use the list defined in section 5 one to one. The decisions made within the code consist of the different types of bank accounts we want to detect - Therefore, we need to choose test cases with values for each detectable bank accounts type. We also need to check for possible illegal values, as this is also a type of decision (throw an expectation or don’t) the code should be able to make. After defining these tests, details provided at [BankLogicTestDecisions.java](https://github.com/AlbertoCazallasMonje/ISO2-2022-A03/blob/4e070ce2fda2ef45889b775796f47bff91981e64/resources/theoreticalwork/tw2-problem2/BankLogicTestDecisions.java), we see all tests are passed.



Looking at the class BankLogic code coverage of 98.1%, we see that only the main public class is not used by the internal logic. The important decisions, however, are covered. The full report is available at: [DecisionCoverageReport.pdf](https://github.com/AlbertoCazallasMonje/ISO2-2022-A03/blob/4e070ce2fda2ef45889b775796f47bff91981e64/resources/theoreticalwork/tw2-problem2/DecisionscoverageReport.pdf).

**8 - For code snippets that include decisions, propose test case sets to achieve MC/DC coverage.**

In this test, we need to produce all different outcomes (Bank account types) using all logically possible combinations. This test will first define the different logic checks in a table, looking at the method: determineAccountTypeYoungCustomer(Customer c), and then define tests in JUnit. Since we already know of the absence of checks for illegal values, we want to test the logic for determining all six different bank account types only.

A = Age.

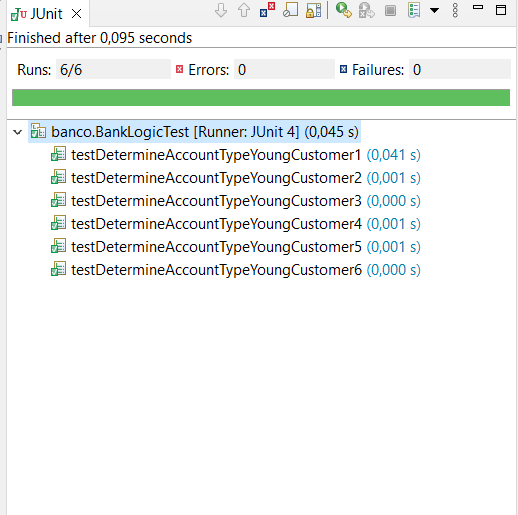
B = IsStudent.

C = IsPermanentResidencyWithParents.

D = IsWorking

| A | B | C | D | Logic | Dominant Condition | Expect output value |
| --- | --- | --- | --- | --- | --- | --- |
| 27 | 0 | 0 | 1 | (A ∧ ¬C)∧ D ) | A,C,D | “Welcome Adult life” |
| 20 | 0 | 0 | 1 | (A ∧ ¬C)∧ D ) | A,C,D | “Jump out of the nest” |
| 20 | 0 | 1 | 1 | (A ∧ C∧ D ) | A,C,D | “Save now” |
| 27 | 0 | 1 | 1 | (A ∧ C∧ D ) | A,C,D | “Become independent” |
| 17 | 1 | 0 | 0 | (A ∧ B) | A, B | “Come on you can” |
| 17 | 1 | 1 | 0 | (A ∧ B ∧ C) | A,B,C | “Comfort” |

Running the newly defined test cases, available at [BankLogicTestMCDCcoverage.java](https://github.com/AlbertoCazallasMonje/ISO2-2022-A03/blob/4e070ce2fda2ef45889b775796f47bff91981e64/resources/theoreticalwork/tw2-problem2/BankLogicTestMCDCcoverage.java), we see that all test cases pass.



**9 - Comment on the results of the number of test cases obtained in section 4, 5, and 6, as well as the execution of the oracles: what could be said about the coverage achieved?**

In the section number four, we can see that with this code we just have one int variable and the rest were boolean variables, so we just need to evaluate the limits values of the int variable (getBirthDate().getAge()) with the differents combinations of the boolean numbers, the result it’s 88 possible combinations.

From this section we could extract that with these 88 possible combinations, we could test the code with all the necessary values.

In the section number five, we want to test if all the operations works, for this task, we use a different combinations for the methods values, with these tests suits we really achieve to detect that all the combinations for the variable (determineAccountTypeYoungCustomer(Customer c)) works and with the section number 7 we really test all the combinations and we detect that all the methods works.

In the section number 6 we have have to achieve the pairwise coverage and for this task we have used the tool PICT from Microsoft, this software it´s very useful to reduce the number of test, for example, at first we had 88 possible cases, and after use the software, we have only 23 different tests suits, with these combinations, we can test all the possibilities for the method, with the exceptions and all the possible values.

About the coverage achieved, we have to say the same as we said in the exercises 7 and 8, all the tests were passed, and with a high coverage percentage, we can resume that all the tests were passed correctly and therefore, the code it's apparently correct.

