Faculdade de Engenharia da Universidade do Porto

Mestrado Integrado em Engenharia Informática e Computação

Métodos Formais em Engenharia de Software

Distributed Printing Service

Authors:	*Student Number:*
Miguel Mano Fernandes	up201503538
Ventura de Sousa Pereira	up201404690

January 7, 2019

Contents

1 Introduction

1.1 System description

The purpose of this practical work is to develop, test and document an executable formal model of a high integrity software system in VDM++ using the Overture tool or the VDMTools.
Our team specifically selected the **company with a distributed printing service** topic which expects the development of a distributed printing queue depending on the type of document (black & white, color, A3, A4, ...), payment with balance, handling of printers placed in different places/clients, printer malfunction reports and solving plus employee sending and display reports with several metrics.
We identified three major actors in our system: **Student** in its user area, **Student** logged in a printer and **Service**.

1.2 List of requirements

Below follows a list of requirements categorized by main system actors.

1.2.1 Service

- R1 (Mandatory)
  A Service can hire Employees, whose sole purpose is to fix Malfunctions in Printers.
- R2 (Mandatory)
  A Service can fire Employees.
- R3 (Mandatory)
  A Service can assign a given Malfunction to an Employee.
- R4 (Mandatory)
  A Service can send all its Employees to fix their assigned Malfunction.

1.2.2 Student in user area

- R5 (Mandatory)
  A Student can add funds to its account so it can pay for document printing.

- R6 (Mandatory)
  A STUDENT can push a document to its personal printing queue.
- R7 (Mandatory)
  A STUDENT can delete a documents from its personal printing queue.
- R8 (Mandatory)
  A STUDENT can report a printer malfunction.

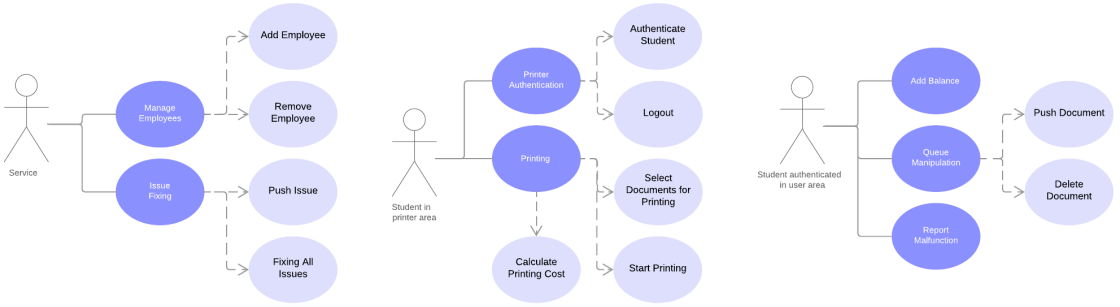
1.2.3 Student in printer area

- R9 (Mandatory)
  A STUDENT can authenticate itself on a PRINTER.
- R10 (Mandatory)
  A STUDENT can log out from a PRINTER.
- R11 (Mandatory)
  A STUDENT can select documents from its personal printing queue to be added to the printer's printing queue.
- R12 (Mandatory)
  A STUDENT can print every DOCUMENT on the PRINTER's printing queue.
- R13 (Mandatory)
  A STUDENT can request the total cost of the current printing queue.
- R14 (Mandatory)
  A STUDENT can check whether it has enough funds to print the current documents on the printing queue.

These requirements are directly translated onto use cases as shown next.

2 Visual UML Model

2.1 Use case model



The major use case scenarios (to be used later as test scenarios) are described next.

2.1.1 Manage Employees

Description: Normal scenario for managing employees.
Preconditions: For adding an employee to the company, it should already not be included in the set of employees and assignment map. Conversingly, for removing an employee, the precondition is being included in the set of employees and map.
Postconditions: For adding an employee, the employee should be included in the set of employees and assignment map in the end. Conversingly, for removing an employee, the postcondition is not being included in both data structures.
Steps: No steps defined.
Exceptions: No exceptions defined.

2.1.2 Issue Fixing

Description: Normal scenario for fixing an issue.
Preconditions: Whilst pushing a malfunction, it should not already be assigned to an employee.
Postconditions: There should be no malfunctions assigned to any employee after fixing every issue.
Steps: Firstly, a student reports a malfunction. The service automatically assigns a student to it. Lastly, it may decide to fix every pending malfunction.
Exceptions: No exceptions defined.

2.1.3 Printer Authentication

Description: Normal scenario for printer authentication.
Preconditions: No preconditions defined.
Postconditions: When authenticating, the student should be recorded whereas when logging out, the student is set to nonexistent. On both situations, the printer queue should be empty.
Steps: No steps defined.
Exceptions: No exceptions defined.

2.1.4 Printing

Description: Normal scenario for printing documents.
Preconditions: Selected document indexes must be higher than zero and the student should be authenticated at all points. At the moment of printing, the queue size should be different than 0.
Postconditions: When adding documents to the queue, the printer's queue size must be equal to its previous size plus the number of documents selected by the student.
Steps: Firstly, select the documents for printing. Then ask the printer to print.
Exceptions: No exceptions defined.

2.1.5 Add Balance

Description: Normal scenario for adding balance.
Preconditions: The value being added can't be zero.
Postconditions: The new balance is equal to the sum of the old balance plus the amount added.
Steps: No steps defined.
Exceptions: No exceptions defined.

2.1.6 Queue Manipulation

Description: Normal scenario for queue manipulation.
Preconditions: When deleting a document, it must be in the printer queue.
Postconditions: When pushing a document, it must then be inside the printer queue.

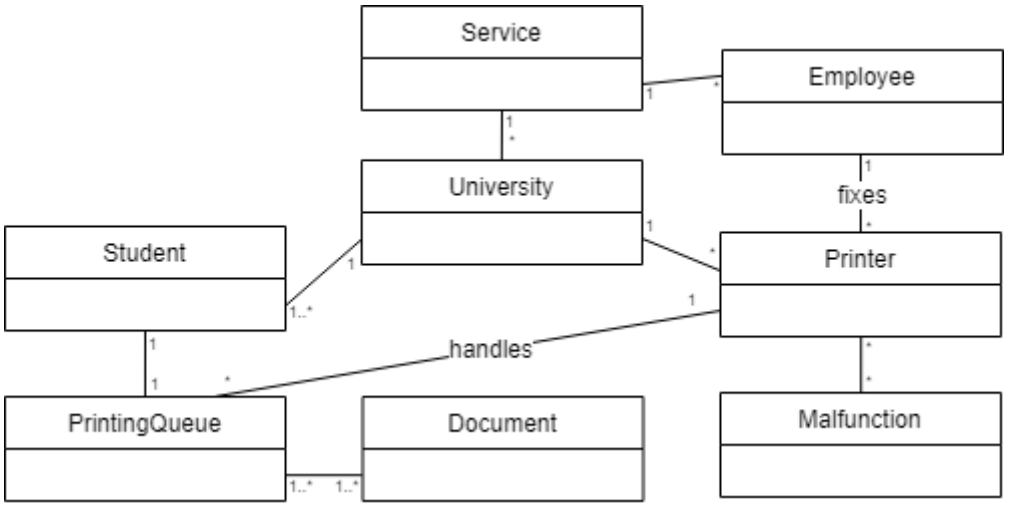
Steps: No steps defined.
Exceptions: No exceptions defined.

2.1.7 Report Malfunction

Description: Normal scenario for reporting malfunctions.
Preconditions: No preconditions defined.
Postconditions: No postconditions defined.
Steps: No steps defined.
Exceptions: No exceptions defined.

2.2 Class diagram

The following UML is the provisional diagram made by hand by the elements of the team. It was thoroughly updated during the development of the project.



We also took the liberty to generate a class on the Overture tool and the UML file *class-uml.uml* is available in the deliverable zip. Also, a simple class description is avaliable on the *feup-mfes-coverage.pdf*.

3 Formal VDM++ Model

Full inspection of the source code is available on the file *feup-mfes-coverage.pdf* included in the deliverable folder. Every class, operation, function and instance variable should be properly documented and a wide array of types was suitably used. Invariants were added and both preconditions and postconditions were applied to every single method (except when not deemed necessary).

4 Model Validation

An extensive file *feup-mfes-coverage.pdf* containing every implemented class alongside its coverage table is available in the deliverable folder. Perfect coverage was reached on every file.

5 Model Verification

5.1 Example of domain verification

Overture generated a total of 72 proof obligations. Below we'll underline an example. We selected number 49 and 50 as they're pretty extensive and cover a couple of types. The title reads `Service.fix_all_issues())`. Let's take a closer look at the code:

```
public fix_all_issues: () ==> ()
fix_all_issues() == (
    for all emp in set dom assigned do
    if assigned(emp) <> nil then (assigned(emp).fix();
    assigned := assigned ++ {emp |-> nil});
)
post rng assigned = {nil};
```

Proof obligation #49 is of type **legal map operation** and its view cites:

```
(emp in set (dom assigned))
```

That means we're only extracting employees which have malfunctions assigned. Relative to proof obligation #50 which tackles the same method, is of type **operation establishes postcondition**. Its view reads:

```
((rng (assigned ++ {emp |-> nil})) = {nil})
```

Therefore, after fixing every issue, every employee mustn't have any malfunction assigned but its key should remain in the assigned map. Each employee must map to nil object at the end.

5.2 Example of invariant verification

Relative to invariant verification, we looked at number #53 of type **state invariant holds**. The first line is the relevant piece of code and the second is the proof view:

```
1. inv balance >= 0.0;
2. (forall idd:Student`String & (assert_id_syntax(idd)
   => (balance >= 0.0)))
```

A student may never be in debt towards the service and in fact every student registered in the system must have a positive balance at all points.

6 Code Generation

After some basic modelling of the VDM++ classes, the group attempted to convert to Java using the provided Code Generation tool on the Overture software.
Unfortunately, compiling through the Windows terminal proved itself too convoluted so we resorted to yet another version of Eclipse.
After fixing other random issues like package linking and main arguments definition, the project compiled and ran smoothly.

7 Conclusions

7.1 Results achieved

The model developed covers all the requirements proposed. We successfully created a company with a distributed printing service. Documents may be printed in different colors and paper dimension and are paid with student balance. Printers are placed in different clients – in our abstraction, universities.
Printer malfunctions may be reported by students and employees are sent by the printing service company to fix said issues.
Each printer also tracks the total number of documents and pages printed.

7.2 Possible improvements

One possible improvement could be the fact that if a malfunction is reported and there's no available employee to fix said failure (possibly every employee is already assigned to other machine) a simple error message pops up.
A welcomed improvement would be creating a sort of malfunction list employees could take on after they were done with their currently assigned issue instead of simply discarding it.

7.3 Division of effort

Work was evenly distributed between both elements of the group.
The project took about two full days of development given the need for extensive VDM++ documentation checking and lack of resources online.