Software Engineering Exercises

First-name Last-name[1]

April 24, 2017

[1]`www.example.com`

Dedicated to Calvin and Hobbes.

Contents

1

Requirements Engineering

1.1 Fidelity Program

Many companies offer fidelity programs, where customers obtain points each time they acquire a good / service, and can later exchange points for gifts. Let's focus on the case of a retailer, with hundreds of stores in Europe. A customer can enroll in the fidelity program, either online, accessing a web site, or in person, at a desk in a retail store. In both cases the customer receives a card, and credentials to access his/her account on the web, via PC or app on smartphone. At each purchase (both online or in a store) the customer can show the card and obtain points, at a certain rate (ex 10 euro – 1 point). The customer, at any time, can access online his account, check the history of purchases and corresponding points. Further, she can consult the list of available gifts, and possibly order one. A gift requires a certain number of points, that are deducted when the gift is ordered. Gifts can be received at home or collected in a retail store. In both cases the customer can follow the state of delivery online, using a tracking number. Delivery is subcontracted by the retailer to a logistic company. Ordering a gift may also involve a payment, by credit card.

In the following you should analyze and model the application that supports a fidelity program for the retailer company.

- Define the context diagram (including relevant interfaces);
- Define the glossary (key concepts and their relationships) with an UML class diagram;
- List the requirements in tabular form (do not forget to list important NF requirements);
- List the user level goal Use Cases;
- Select one use case from the point above and describe it.

Table 1.1: Relevant interfaces between actors and system

Actor	Physical Interface	Logical Interface
Customer	PC / Smartphone	GUI
Clerk of Retailer	PC	GUI
Administrator of Retailer	PC	GUI
Credit Card Circuit	Internet Connection	Web service (for payment)
Logistic Company	Internet Connection	Web service (parcel tracking)

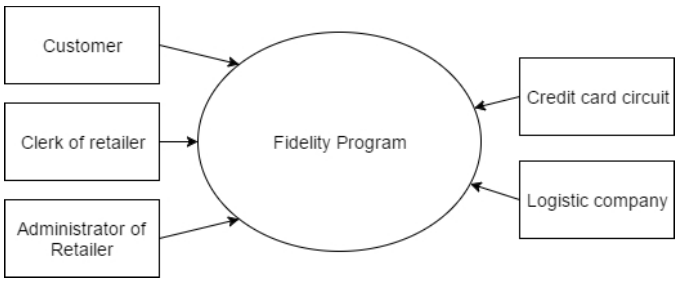


Figure 1.1: Fidelity Program Context Diagram - Sketch

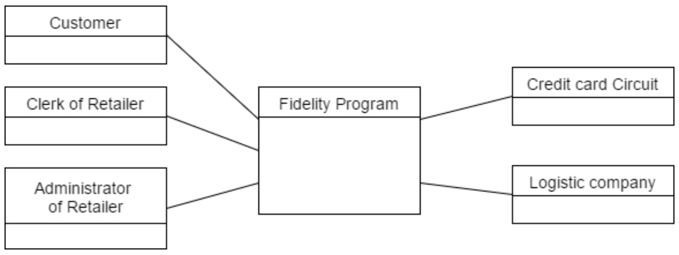


Figure 1.2: Fidelity Program Context Diagram - UML Class Diagram

Solution

In the Context Diagram, five different actors interact with the Fidelity Program system: the Customer, the Clerk and the Administrator of the retailer, the Credit card circuit, and the Logistic company in charge of shipping the selected goods.

In the Glossary, each user is identified by a Customer Descriptor, enrolled to a Fidelity Program and performing a set of Purchases. In the solution shown in the figure, the Customer can have at most one Fidelity Card. It could also be considered that a customer could have different more than one registered fidelity cards (for instance, in the case of loss or expiration of previous ones). This would change the cardinality of the relationship between Fidelity Card and Customer Descriptor classes (* instead of 0..1). Gifts are identified by a Gift Descriptor, and is linked to the Customer who orders it through a Gift Order class.

The Gift Order class could also be considered an association class; it is

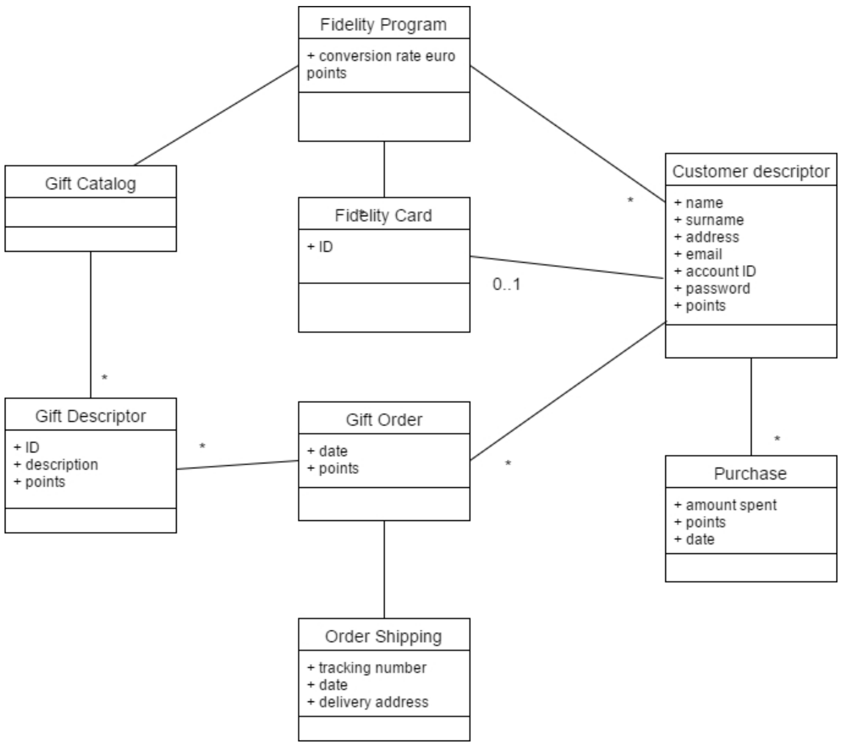


Figure 1.3: Glossary - UML Class Diagram

Table 1.2: Requirements in tabular form

ID	Type (F / NF)	Description
1	F	CRUD customer descriptor and account information
2	F	Login, Authenticate and authorize customer, logout
3	F	CRUD Gift Descriptor
4	F	CRUD Gift Order
5	F	CRUD Purchase
6	F	CRUD Order Shipping
7	F	Interact with logistic company
8	F	Interact with credit card system (send payment, receive feedback)
9	NF	Privacy: user should r/w only her data. Credit card info concealment
10	NF	Security: payments need authorization by card owner
11	NF	Performance: all functions to be completed in <1secs

however fundamental that is considered as a separate class from the Gift Descriptor, because it bears two additional attributes and the link to an Order Shipping object.

In the requirements, for what concerns the operations regarding purchases, they can be already present in retail store mgmt software, so they may not be strictly part of the Fidelity Program, but at least a proxy to them must be present. The same discussion can be made for the tracking of orders, because the tracking can actually be made on the site of the logistic company. The Non-Functional requirement related to performance may exclude the execution of the payment functions, since they are more likely to be completed in much time than 1sec.

Use Cases for the system are listed in the following:

Customer or clerk: Register;
Customer: Order Gift;
Customer: Track gift delivery;
Clerk: Manage gifts (add, modify);
Admin: Manage customer accounts (add, delete, modify).

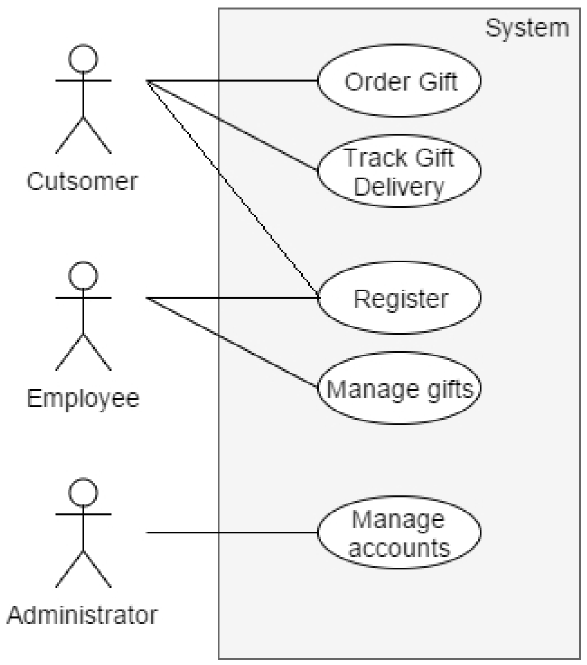


Figure 1.4: Use Case Diagram for the Fidelity Program

Steps for complex use cases (e.g., Order Gift) can be detailed with include directives, as in the following picture.
Among those use cases we select and detail Order Gift.

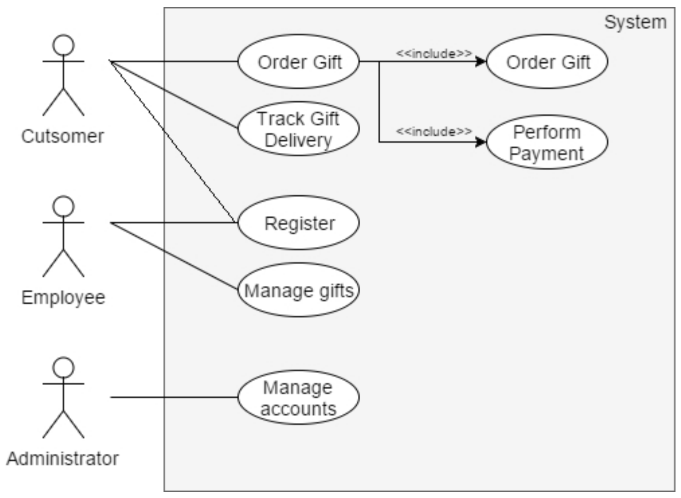


Figure 1.5: Use Case Diagram for the Fidelity Program (with inclusions)

Table 1.3: Description of the Order Gift use case.

Name	Order Gift
Scope	Fidelity Program Management
Level	User Goal
Intention	A customer wants to use her points to get a gift
Primary Actor	Customer
Precondition	Customer has a card and an account (has subscribed to fidelity program), has enough points to obtain a gift.
Main Success Scenario	1. Customer logs in, 2. Customer browses available gifts, 3. Customer selects gift. 4. System checks if customer has enough points. 5. System asks shipping address. 6. System confirms transaction and detracts points. 7. Customer logs out. 8. System starts procedure for packaging and shipping.
Extensions	3b. Gift not available. 4a. Not Enough Points. 4b. Payment needed. 4c. Payment failed.

Common mistakes and suggestions

A common mistake may be to not consider the Logistic Company as part of the Context Diagram: it must be included in it, since it performs the final operation of shipping gifts and must connected (through an interface) to the system.

Entities like the fidelity card or the credit card must not be considered as actors of the diagram. The Credit Card System must be part of it. In the Actor definition, the connection between the Logistic Company and the software system is considered: thus, the physical/logical interface used is a PC (or similar device) with a dedicated GUI, not another kind of physical interfaces for the logistic company to ship the orders.

Table 1.4: Wrong examples of actors and interfaces

Actor	Physical Interface	Logical Interface
Fidelity Card	Card Reader	Dedicated protocol
Credit Card	Card Reader	Dedicated protocol
Logistic Company	GPS	GPS Protocol

Not all actors that interact with the system need to be part of the class diagram. For instance, classes about the clerks or administrator of the retailer would be of no interest in the diagram. A hint about the fact that those classes are not relevant in the class diagram can be the fact that no particular attributes are specified for those actors, whereas the Customer has a full set of specific attributes.

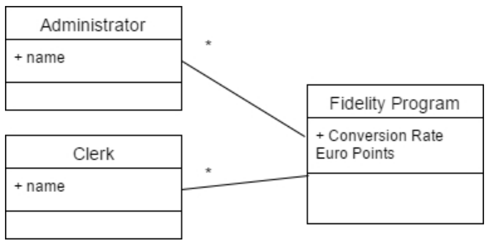


Figure 1.6: Wrong classes in Class Diagram

Another possible mistake would be to neglect the Purchase class, including purchases inside the customer descriptor. A hint to the necessity of a class to describe the individual purchase is the “points” attribute, given in the specification, that is different for each purchase.

The Order Shipping class, which contains attributes for the logistic company about a Gift Order object, must be kept separate from the Gift Order class, since it logically represents another entity in the system (even thought it has a one-to-one link with any Gift Order).

The Point to Euro conversion rate must be placed in the Fidelity Program, not in the Fidelity Card class nor the Customer Descriptor.

For what concerns Use Cases, a common possible error is considering the

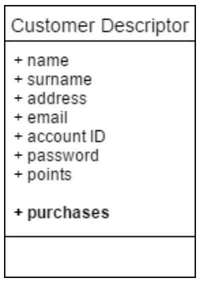


Figure 1.7: Wrong definition of Purchases

use cases in the use case diagram as a flow of operations to be performed by an actor. Instead, the only connections that can be made between different use cases in the diagram are include and extends connection, not logical prosecutions of the operations. For this reason, the use case in the following diagram is wrong.

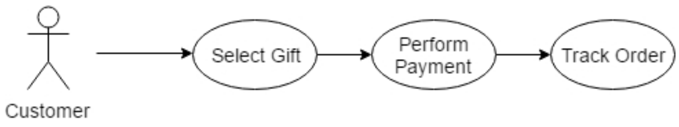


Figure 1.8: Example of wrong Use Case

A very relevant mistake is considering the system as an actor, and not just something with which the actors interact with. In the use case diagram for this particular application, it would be a serious mistake to put the Fidelity Program System amongst the actors, like in the following picture.

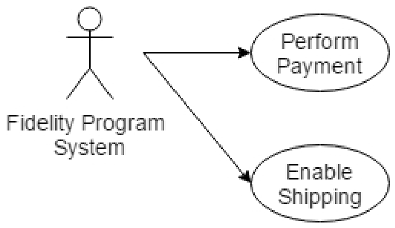


Figure 1.9: Example of wrong Use Case

Objects (classes) of the system must not be in the use cases. It makes no sense to connect those to the actions that are performed by actors (neither with the use of include), like in the following picture (errors are circled).

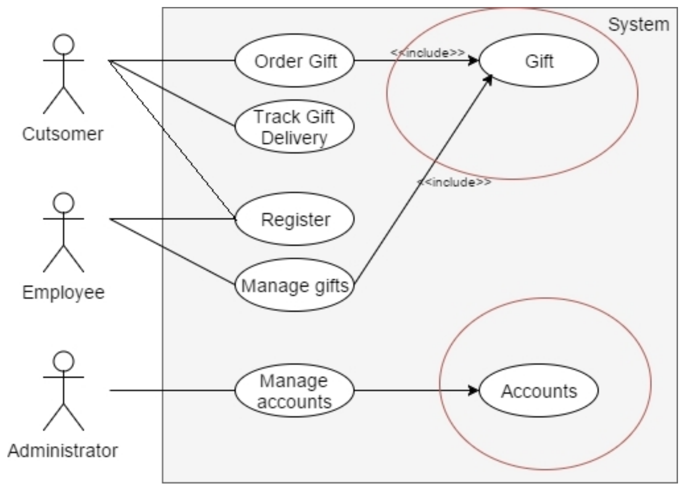


Figure 1.10: Example of wrong Use Case

1.2 Electric Power Billing

Each house connected to the electric network has a meter. The meter computes and stores how much electrical energy (kW hour) is used by the house. The meter sends regularly (say every 24 hours) the consumption over the electric wires (the electric wires of the power network can in fact be used as data transmission lines). A remote central server receives the data, stores it and computes a bill (say every two months), sends the bill to the customer. The bill can be sent in two different ways: electronically (email or web site accessible to the customer via username and password) or on paper. In the following you should analyze and model the application that collects power usage from each meter, computes the bill and sends it to the customer.

- Define the context diagram (including relevant interfaces);
- List the requirements in tabular form (do not forget to list important NF requirements);
- Define the glossary (key concepts and their relationships) with an UML class diagram;
- Define the system design (key hardware and software components of the application, UML class diagram) for the application. Describe the key design choices;
- Define one scenario describing the computation of a bill for a customer.

Solution

Five actors can be identified as interacting with the system: the customer/houseowner and the meter located in each house; the e-mail system, in charge of sending the bills to customers; the postal system, providing the alternative paper bills to the customers; finally, the banking system, needed for the execution of transactions through which the customers can pay their bills.

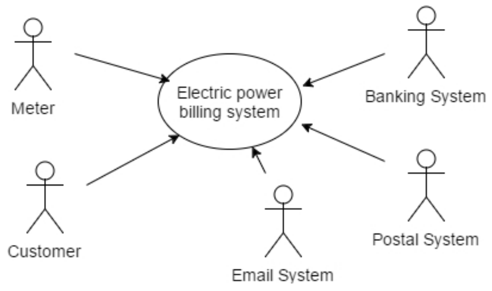


Figure 1.11: Electric Power Billing System Context Diagram - Sketch

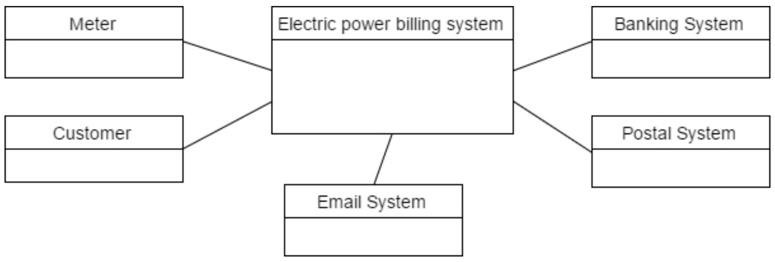


Figure 1.12: Electric Power Billing System Context Diagram - UML Class Diagram

Table 1.5: Electric Power Billing System - Interfaces

Actor	Physical Interface	Logical Interface
Customer	PC / Smartphone	GUI / Webpage
Postal System	PC	Web service offered by postal company
Banking System	PC	Web Service
E-mail System	PC	Web Service, POP, IMAP
Meter	Electrical Wires	Electric signals

In the Class Diagram, the customer of the system is represented by a Customer Record class. Each Customer Record can have multiple Contract Records, linked in a one-to-one relation to a Meter. The Contract Records is also linked to multiple Meter Readings (the measurements of the consumed energy) and to multiple bills that are sent to the customers to be paid.

In table 1.6 the requirements of the system are shown. In addition to the ones pertaining the functionalities of the system, three non-functional

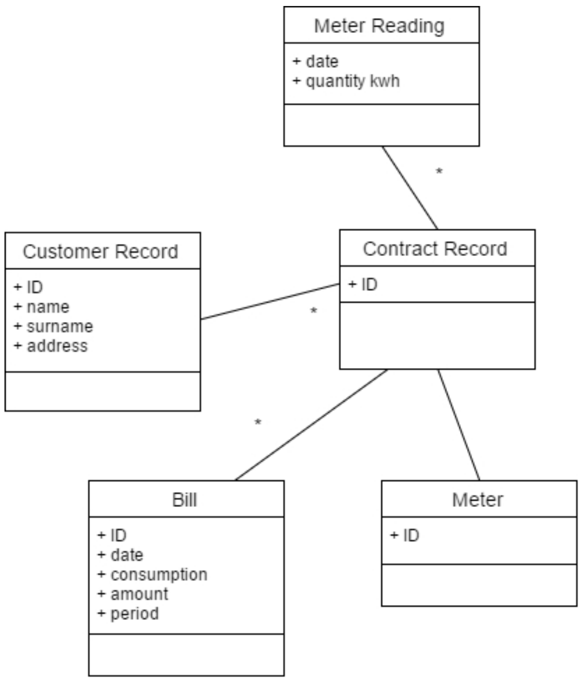


Figure 1.13: Electric Power Billing System - UML Class Diagram

Table 1.6: Electric Power Billing System - Requirements in tabular form

ID	Type	Description
1	F	Read consumption from meter
2	F	Storec consumption and date (meter reading)
3	F	Send meter reading to server
4	F	Request and receive meter reading from meter (on server)
5	F	Store meter reading for specific meter (on server)
6	F	Retrieve meter reading for specific meter (on server)
7	F	Compute bill (on server)
8	F	Send bill via postal system (on server)
9	F	Send bill via email (on server)
10	NF	Privacy, data of each customer should not be visible to others
11	NF	Domain Currency is euro
12	NF	Domain Electric Consumption is KWh

requirements have been formulated. The first one pertains privacy, that is fundamental in any kind of system that is treating personal data. In addition to that one, a non-functional requirement about security could be added, meaning that the personal data would be, for instance, encrypted when stored on the servers.

Requirements 11 and 12 are non-functional requirements pertaining to

Localization, i.e., the adaptation of a system to the units of measure and currencies of the environment in which it will be released.

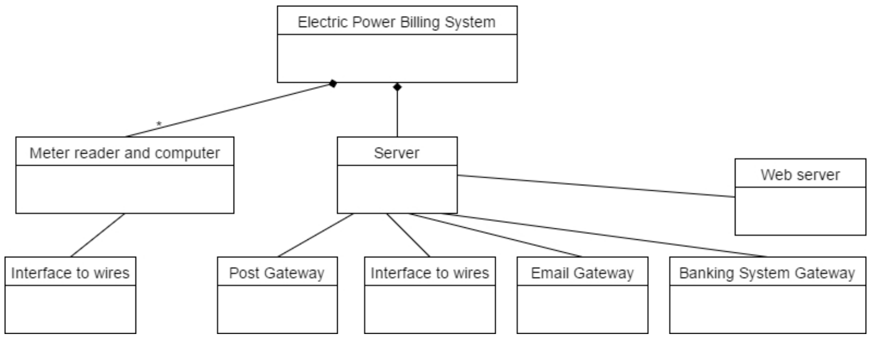


Figure 1.14: Electric Power Billing System - System Design, UML class diagram

In the graph in figure 1.14 it is made evident that the two main components of the system are a central server, to which all gateways are connected, and a metric reader for the measurements that must be used to compute the bills for the consumers.

Finally, in the following, the scenario for the operation of Bill Computation is described and linked to functional requirements.

Bill computation scenario:
Precondition: meter M, x months from last bill passed, all meter readings for meter M for last x months are available;
Postcondition: bill for meter M computed and sent.

Table 1.7: Bill computation scenario

Step	Description	Req ID
1	Retrieve all meter readings for meter M for last X months	6
2	Compute consumption in period	7
3	Apply fee and taxes	7
4	Produce bill	7
5	Send bill, according to consumer preference	8, 9

1.3 Restaurant Queue Management

At a busy restaurant in a mall, a system is used to manage the waiting queue. When a new customer arrives, the waiter, given the number of people in the party, checks on the system the expected waiting time to get a table; then the customer provides her name and mobile phone number, the system then sends a confirmation SMS to the customer's phone. At this point the customer has a table reserved. Then the customer is free to walk away and e.g. visit the shops in the mall. Every five minutes the system will send an SMS with the estimated residual waiting time. When the table for the customer gets available, a waiter sets the table as ready and the system sends an SMS to the customer asking her to get to the restaurant as soon as possible to get the reserved table. As the customer gets back to the restaurant she is seated at the reserved table. If the customer does not show up within a fixed time, then the next customer in the waiting queue is called. In the following you should analyze and model the queue management application.

- Define the context diagram (including relevant interfaces);
- List the requirements in tabular form (do not forget to list important NF requirements);
- Define the glossary (key concepts and their relationships) with an UML class diagram;
- Define the system design (key hardware and software components of the application, UML class diagram) for the application. Describe the key design choices;
- Define one scenario describing a customer that asks for a table at the restaurant.

Solution

Three actors can be found in the description of the system: the user, that uses the system through its smartphone, since he just interacts with the system through the SMS' he receives; the waiter, that directly interacts with the system in the restaurant by its user interface; the SMS gateway to which the system is connected, and that allows the sending of messages to the customers.

Table 1.8: Relevant interfaces between actors and system

Actor	Physical Interface	Logical Interface
User	Smartphone	GUI (SMS)
SMS Gateway	Internet Connection	Web Service
Waiter	Tablet/Smartphone	GUI

All the functional requirements are implemented through functions in the Restaurant Queue class. Information about tables (reserved or not) are stored in Table objects. The timeSetBusy variable represents the time at

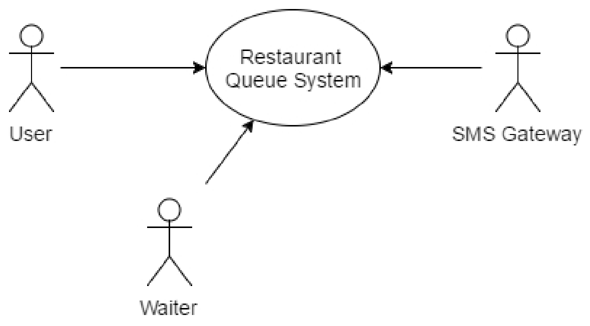


Figure 1.15: Restaurant Queue Management - System Context Diagram (Sketch)

Table 1.9: Requirements in tabular form

ID	Type (F / NF)	Description
1	F	Set up (number and capacity of tables, waiters)
2	F	Compute time to get table (with no reservation yet)
3	F	Ask to reserve table (create reservation)
4	F	Assign table, given reservation
5	F	Set table as busy
6	F	Set table as free
7	F	Communicate time to have table (given reservation)
8	F	Cancel reservation if not used within given time (e.g. 5 mins)
9	NF	Usability: GUI on smartphone usable by non IT experts within max 5 min training
10	NF	Privacy: Customers data not accessible

which a table becomes busy, and that is used to estimate when the table can become free, to send the information to the waiting customers.

The application (and hence all the classes represented in the class diagram: RestaurantQueue, Table, Reservation, Customer) runs on the the server. The Tablet/Smartphone provides the GUI for the employee/waiter of the restaurant, that must conform to the non-functional requirements listed in the table. The SMS gateway, connected by an internet link to the system, is in charge of sending messages about the status of tables to the waiting customers.

Table reservation scenario:
Precondition: no free tables;
Postcondition: Added Reservation R for customer C at time t0. Estimated free table for C at time t1.

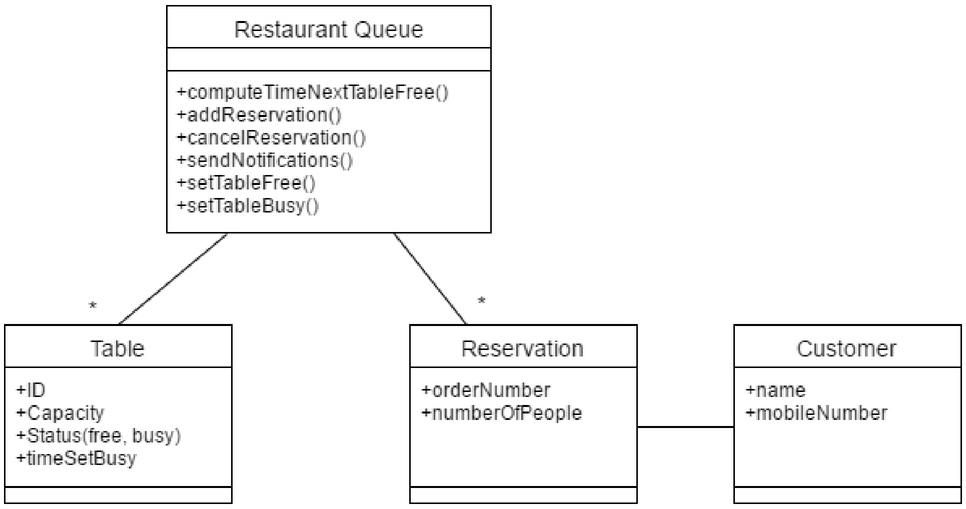


Figure 1.16: Restaurant Queue Management - UML Class Diagram

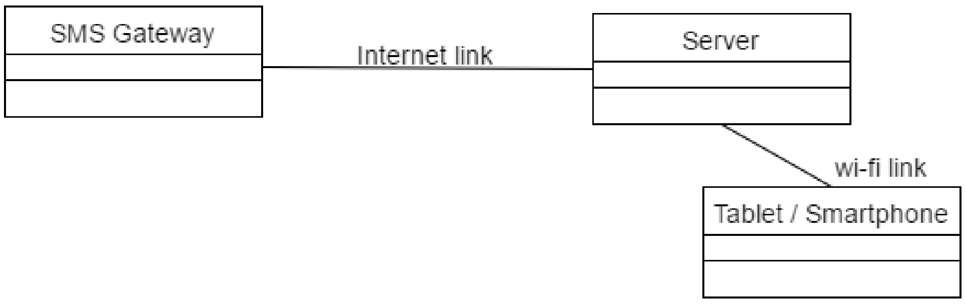


Figure 1.17: System Design - UML Class Diagram

Table 1.10: Table Reservation Scenario

Step	Description	Req ID
1	Customer asks for a table	-
2	Waiter checks waiting time and informs customer	2
3	Customer agrees to wait and asks for reservation	-
4	Writer creates reservation	3
5	System sends to customer notification of time to wait	7

1.4 Bike Sharing

A bike sharing system allows users to rent and use a bicycle in a city for a certain amount of time, for a small fee (cfr TO BIKE in Turin). First a user has to register in the system, giving his identity, and a credit card. Then the user can pay for the service (in the case of Turin a yearly fee plus an amount of money that corresponds to a certain amount of minutes available). The user receives at his address a specific RFID card to access the system. If the user already has a city transport card or other cards (ex the university student ID card), then these cards are used.

In the city many parking spots for bicycles are available. The parking spots are made of many stands, each controlling a bicycle. If the user has the card she can access a stand (equipped with an RFID card reader) and requests a bicycle. If authorized (credit available, no banning for previous damages, etc) the stand unlocks the bicycle and the user can use the bicycle. Then the user can return the bicycle to any other stand.

Administrators of the system must be able to monitor the system (bikes available per each parking spot, used bicycles, missing bicycles, definition of usage fees etc). Similar functions are available to users, via a web site, that shows the parking spots in the city and the number of bicycles available. The web site allows also administering one's account (check the history of usage for a user, check credit available, etc).

In the following consider the system for bike sharing (including bicycles, stands, web site).

- Define the context diagram (including relevant interfaces);
- Define the glossary (key concepts and their relationships) with an UML class diagram;
- List the requirements in tabular form (do not forget to list important NF requirements);
- Define the system design (key hardware and software components of the application, UML class diagram) for the application. Describe the key design choices.

Solution

Three actors can be identified in the described system: the user, which interacts with the system using his PC or smartphone, through a dedicated GUI; the Administrator of the system, which interacts to the system through a PC through a dedicated admin panel; the Credit Card System, which is connected to the system through internet connection and ad-hoc protocols.

In the context diagram, in addition to classes representing physical entities of the system (Bicycles, Stands, Parking Spots), classes are dedicated to describe the Users of the system (and their accounts), and the Usage of a bicycle performed by a particular account.

The relationship between the Bicycle and Account classes can be also represented as an association class. Since the relationship between User and

Table 1.11: Relevant interfaces between actors and system

Actor	Physical Interface	Logical Interface
User	PC (Smartphone)	GUI
Admin	PC	GUI
Credit card system	Internet connection	HTTPS, SSL, APIs...

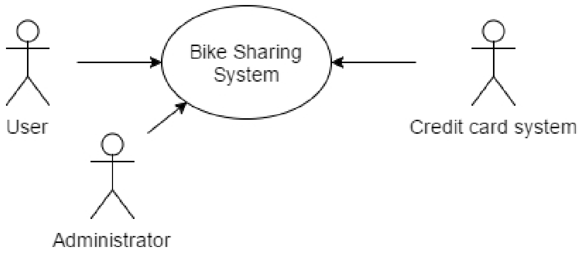


Figure 1.18: Bike Sharing - System Context Diagram (Sketch)

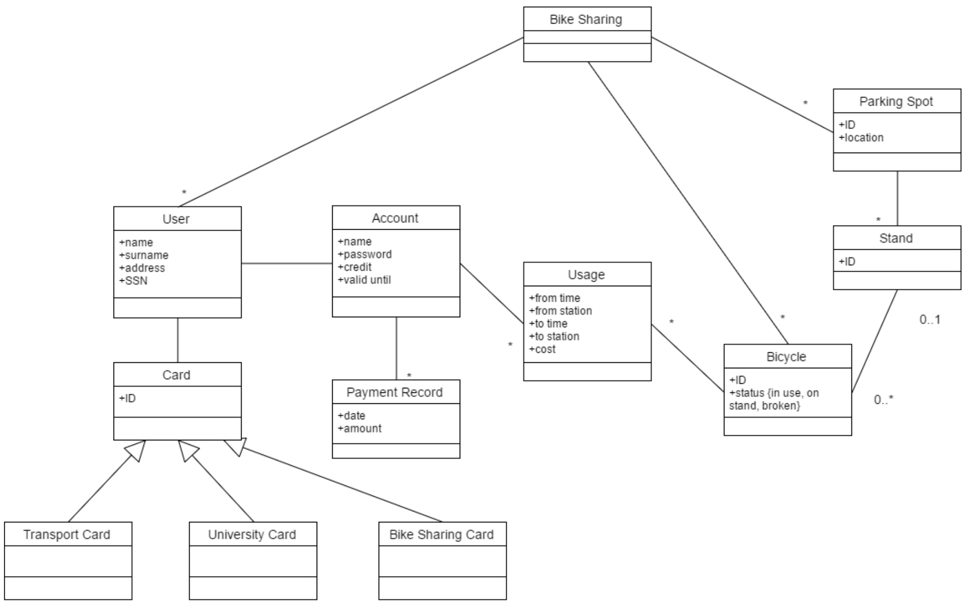


Figure 1.19: Bike Sharing - UML Class Diagram

Account, and Card and User, is one to one, the Card class can be connected to either of the two classes.

In table 1.12 the functional requirements of the system are shown: CRUD operations to configure the system and its users, statistics, monitoring and accounting operations. In addition to them, three plausible non-functional requirements are formulated, pertaining Privacy, Usability and Performance.

Table 1.12: Requirements in tabular form

ID	Type (F / NF)	Description
1	F	User Management CRUD user, CRUD account, CRUD card, attach card to user, attach card to account
2	F	System Configuration CRUD station, CRUD stand, CRUD bicycle, attach/detach stand to station, attach/detach station to system
3	F	Statistics Compute day/week/month usage per user/bicycle/station
4	F	Operative and accounting Authorize usage of bicycle, CRUD Usage, compute cost, attach usage to account, update fees per account, request payment, manage payment per account
5	F	Monitoring and maintenance Monitor status of bicycle/stand, ask and monitor repairs, Show status of stations/bicycles on map
6	NF	Privacy: User data should be visible only to user or admin
7	NF	Usability: A user familiar with internet tools (browser) should be able to use all functions on web site in at most 5 minutes, with no training
8	NF	Performance: Authorization (or not) to collect a bicycle should be granted in less than 0.5s after RFID card is read.

The system design models physical entities of the system (principally computers, devices and connections between them). A server is needed to manage all data and logic of the application. Each parking spot has a computer, to manage all the stands, and a computer is installed in each stand (for recognizing the bikes, reading RFID cards, locking and unlocking the bikes). Another possibility could be to have a single computer for each parking spot, directly connected to all the stands of the spot. Each stand, in either cases, is equipped with three devices: the RFID reader, a device to lock and unlock bicycles, and a bicycle interface to recognize the ID of the bicycles.

Yet another design has a single interaction point per parking spot (a screen, a keyboard and a card reader): the user interacts with it to operate on bikes, then a bike is unlocked and can be used. This design is cheaper, but provides minor usability to users.

Suggestions and common mistakes

Bicycles, Stands and Cards are considered a part of the system, so it

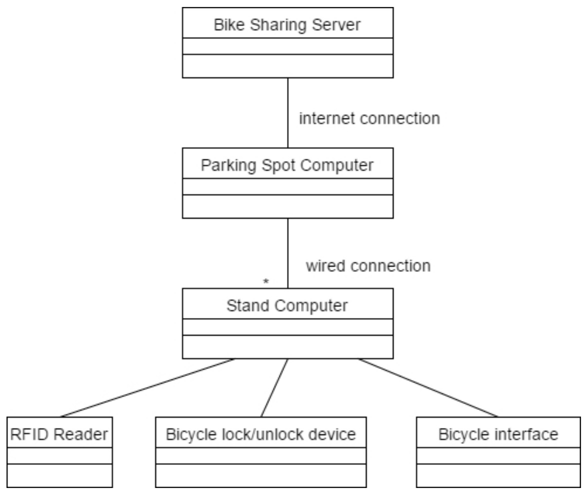


Figure 1.20: Bike Sharing - System Design Model (UML Class Diagram)

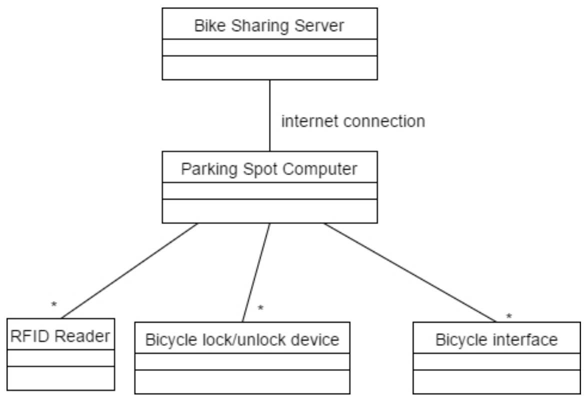


Figure 1.21: Bike Sharing - Alternative System Design Model (UML Class Diagram)

would be an error to place them as actors in the system diagram, as in the following image. A possible additional actor can be a logistic company (e.g., postal service or similar) in charge of shipping cards to users. Another possible actor could be an e-mail gateway, in charge of sending e-mails to registered users.

Elements of the physical system (like the credit card system, the RFID reader) or the website of the application must not be put as separate classes inside the Class Diagram, since they are considered to be part of the Bike Sharing system, or of the individual stands.

Another error would be to put the card field inside the User class. This would made impossible to made a distinction between different kinds of

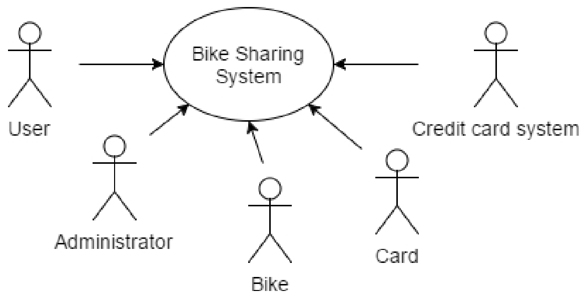


Figure 1.22: Bike Sharing - Wrong System Context Diagram (Sketch)

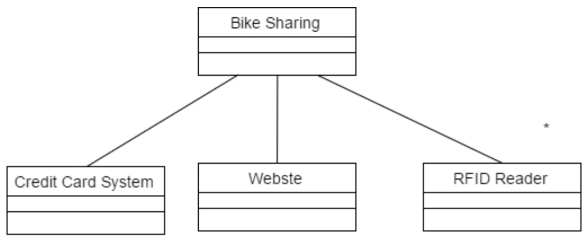


Figure 1.23: Bike Sharing - Wrong Classes in Class Diagram

RFID cards, that need to be modeled using a hierarchy of classes.

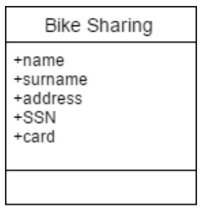


Figure 1.24: Bike Sharing - Wrong Classes in Class Diagram

Person and Account must be considered as separate classes, since they carry different information: the physical user of the system, and the way he/she is registered inside it. A possibility (not necessary in the solution) is to consider also an Admin class in the system, and define a specialization of a parent Person class.

In the System Design Model, it is an error to consider the Credit Card System, which is not part of the system (it has an interface to it), or RFID cards (which are not physical elements of the system).

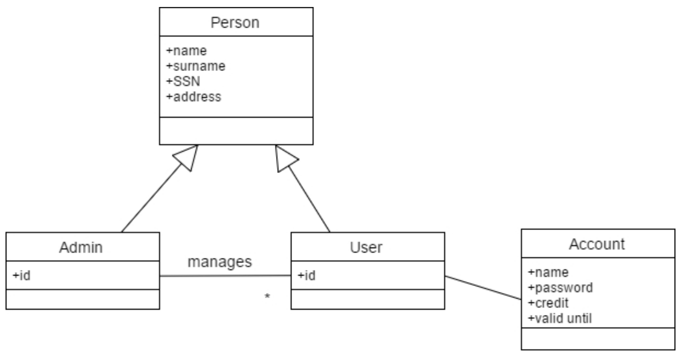


Figure 1.25: Bike Sharing - Admin - User specialization in Class Diagram

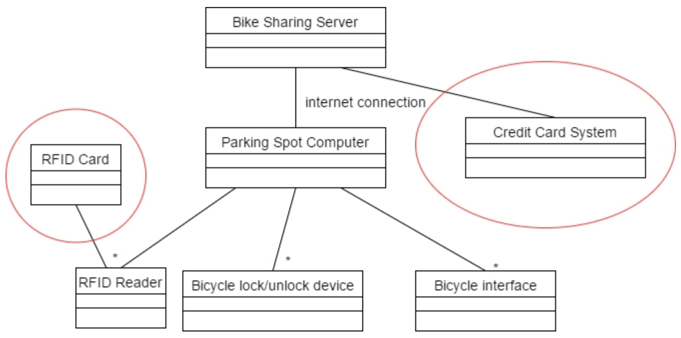


Figure 1.26: Bike Sharing - Wrong System Design Model

1.5 Car Maintenance Management

Car maintenance management During the lifetime of a car many maintenance and repair interventions are needed, some scheduled, some not. It is important for the owner of the car to keep track of these interventions. On the car producer side, it is important to keep track of these interventions both for safety issues (recalls due to defects) and for customer care. Many car producers or car dealers have developed applications to handle car maintenance.

In the following you should analyze and model the application that supports web based car maintenance management.

Key roles to be considered are the car owner, the car manufacturer (defines maintenance schedules and jobs), the maintenance shop (records maintenance interventions).

Key high level functions to be considered are:

- Define the regular maintenance jobs for a car (typically performed by car manufacturer). Remark that jobs (both type and schedule) depend on the car model;

- Record a set of interventions on a car, due to normal maintenance or not (typically performed by a maintenance shop). Record also effort spent and cost for each job, to be used both for owner records and for payment;
- Remind owner about a scheduled maintenance;
- Browse and analyze jobs for a car (typically performed by car owner or maintenance shop).

Considering the requirements above, in the following you should:

- Define the context diagram (including relevant interfaces);
- Define the glossary (key concepts and their relationships) with an UML class diagram;
- List the requirements in tabular form (do not forget to list important NF requirements);
- Define the system design (key hardware and software components of the application, UML class diagram) for the application. Describe the key design choices;
- Define one scenario describing a maintenance recall (invite owner of car for a scheduled maintenance job, for a number of times or until job done).

Solution

The four actors that need to interact to the system are the Car Owner requiring for maintenance, the Car needing it, the Car Manufacturer defining the maintenance jobs, the Maintenance Shop handling the actual maintenance of the car.

All actors have an internet connection as physical interface to the system, and a GUI within the browser with specific windows and menus as logical interface to the system. Also the car, for more recent models (those built in last 20 years), must be considered as an actor in the system. If it is the case, it can mount an interface (typically RS232 or CAN) between its main ECU (Electronic Control Unit) and the maintenance system. This kind of access is typically restricted to authorized maintenance shops. The logical interface is made of functions to read data, and data formats for the information exchange.

In the UML Class Diagram, the class ‘Car’ describes an actual car. The Car has an ID (the chassis number) that never changes, and a tag that

Table 1.13: Relevant interfaces between actors and system

Actor	Physical Interface	Logical Interface
Car Manufacturer	PC (Internet Connection)	GUI
Car Owner	PC (Internet Connection)	GUI
Maintenance Shop	PC (Internet Connection)	GUI
Car	CAN	OBD-II Protocols

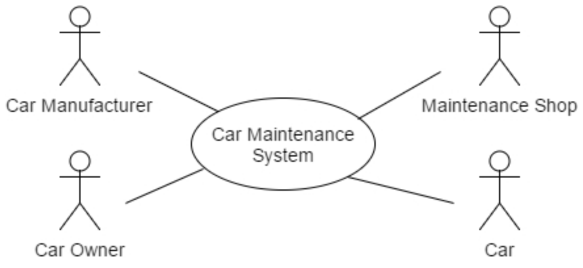


Figure 1.27: Car Maintenance Management - System Context Diagram (Sketch)

can be changed throughout its lifetime. 'Car Model' describes characteristics that are common to cars of the same model. The Car Manufacturer is linked to its catalogue of models.

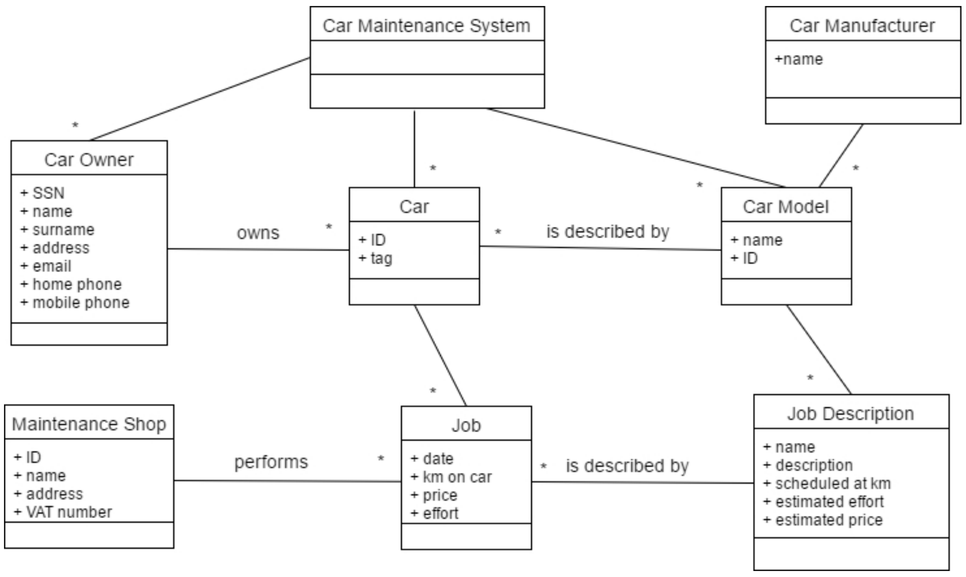


Figure 1.28: Car Maintenance Management - UML Class Diagram

Similarly to the 'Car' class, the 'Job' models an actual Job performed on a real car, whereas the 'Job description' describes a generic job, with estimated cost and required effort. Therefore, 'Job description' is attached to 'Care Model', and 'Job' to 'Car'. Job Descriptions allow defining the schedule of Jobs for a specific car (the attribute 'scheduled at km'). Every job is performed by a specific 'Maintenance Shop'; multiple Maintenance Shops are part of the Car Maintenance System. The information about the car owner must be kept in a separate 'Car Owner' class and not in the 'Car' class itself, since a single person can own multiple vehicles.

Requirements for the system are shown in table 1.14. The functional requirements have been grouped in six groups, each pertaining an element of the system (Car, Car Owner, Car Model, Job Description, Job). Similar requirements should have been formulated also for the management of Maintenance Shops and Car Manufacturer registered in the system.

An example of Privacy Non Functional Requirement (req. 9) is also provided. Similar requirements can be formulated for the management of each element of the system, since every element carries sensible information that should be known only by the respective Maintenance Shop / Car Owner / Car Manufacturer.

Table 1.14: Requirements in tabular form

ID	Type (F / NF)	Description
1	F	Car Management
1.1,1.2,1.3	F	Add a car, modify a car, attach car to owner
2	F	Car Model Management
2.1,2.2	F	Add a model, modify a model
3	F	Job Description Management
3.1,3.2	F	Add job description, attach job description to car model
4	F	Car Owner Management
4.1,4.2	F	Add owner, modify owner
5	F	Job Management
5.1,5.2,5.3,5.4	F	Add job, modify job, attach job to job description, attach job to car
6	F	Scheduled maintenance management
6.1,6.2	F	Compute scheduled maintenance for a car, send reminder to owner
7	NF	Performance: all functions should have response time < 0.5sec
8	NF	Privacy: all functions and data about car model accessible only by car manufacturer (similarly for other functions)

In the following, the scenario describing a maintenance call is formulated.

Precondition: car C has job description JB to be performed in less than 14 days from today.

Postcondition: job J corresponding to job description JB performed on car C.

Table 1.15: Maintenance Recall Scenario

Step	Description	Req ID
1	Send Reminder to CO (Car Owner of Car C) about job JB	6.2
	One week passes	
2	Send Reminder to CO (car owner of car C) about job JB	6.2
	Three more days pass	
3	CO brings car C to maintenance shop and requires the performance of the job	
4	Shop records job J corresponding to JP on car C	5.1,5.3,5.4

2

Static Diagrams

2.1 Sport Support Application

A sport support application allows a user to log her sport activities, to keep track of them, and analyze performances, and share with friends. It runs on smartphones equipped by GPS, while the analysis of performance part can be done both on the smartphone or through a browser on a web site.

The basic unit managed by the application is the training. A training has a start time, a stop time, an attached sport (cycling, jogging, walking, skiing etc). It also has a geographical route, that is collected through the GPS. The training can be split in parts. So a typical scenario is the following: the user starts a new training, and when done he stops it; or the user starts a new training, stops a part and starts another part, one or more times, then stops the training (the typical usage of parts is to measure laps when running on a circular track).

In some cases the user can take pictures during the training. The picture is geo-referenced and also attached to the training.

If the user has a heart rate monitor (a sensor put on the chest, and linked via Bluetooth to the smartphone) the heart rate log is also attached to the training.

On the analysis side, the user can: see a list of all her trainings, select a training, see charts (speed chart, altitude chart, heart rate chart), see metrics (average speed, max speed, per training and per part, etc), see the route followed on a map, see pictures taken, on the same map.

On the social side, a user can define a list of friends, and share with them one or more trainings.

Solution

The central element of the Sport Support Application Class Diagra is the Training class, which represents a training started by the user. A Training object can be subdivided in multiple parts, and can be attached to multiple

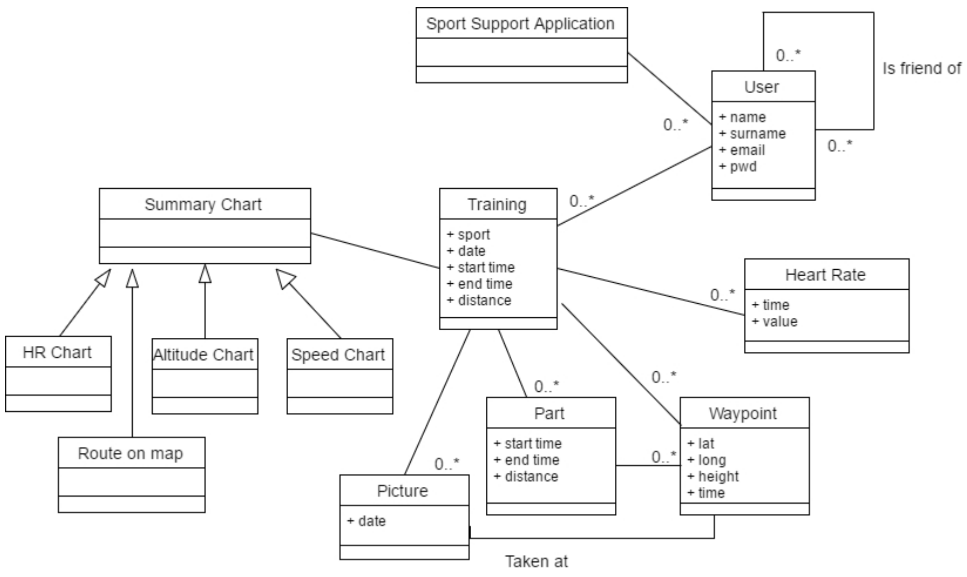


Figure 2.1: Sport Support Application - UML Class Diagram

pictures. To symbolize the possibility to geo-reference a picture and the starting point of a training (part), the class Waypoint is used (it includes coordinates and time).

The heart rate can be attached to the training (the fact that it is optional is symbolized by the 0...* multiplicity in the relationship).

The types of Summary Charts that can be attached to the training are all represented by a parent-children relationship between a generic Summary Chart class and specialized ones.

Variable elements that are collected in trainings cannot be considered as internal arrays (see following figure) of the “training” object: they must be considered as individual classes, linked with the Training class with a “many” cardinality.

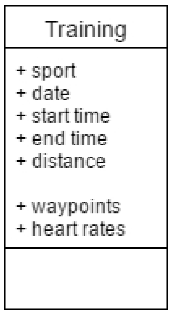


Figure 2.2: Sport Support Application - Wrong Training Class

Physical objects that are described in the specification of the application (Smartphone, GPS Sensor, Server, Camera) are not necessary in the conceptual diagram, even though it would not have been an error to include them.

2.2 Trip Sharing Application

New technologies allow to transform any car / driver in a taxi/ taxi driver.

Potential users access a web site (from mobile devices or PCs) and register in the system. When they need a taxi they log in and request for a ride (detailing start point and destination).

Potential drivers also access the web site and register once. Then, when they are willing to share their car for a ride, they log in, and describe the ride they are doing (or they are planning to do).

The system then tries to match rides requested by users and rides offered by drivers. The match is evaluated basically on start point, start time, destination. In case the match is possible the system contacts the potential driver and asks him / her if he is available for the specific ride. If so, then the system puts in contact the user and the driver, establishing also a price for the ride.

At this point, always through the application, user and driver negotiate the final details (exact point for meeting, destination). If the negotiation succeeds, the driver picks up the user and performs the ride.

Both for security and for efficiency, each ride is tracked: start point and start time, path followed, end point and end time are all registered. This is also made through the app on the smartphones of both driver and user.

At the end of the ride, and if all is right, the user pays the ride. No money is exchanged directly between driver and user: the exchange is made on the accounts associated to driver and user and managed by the application. A percentage of the price goes to the company providing the sharing service.

The system supports also a peer evaluation system. At the end of the ride the user evaluates the driver, and vice-versa.

Solution

It is important to represent separately Ride Request, Ride Offer and the actual ride, which is represented by the Ride Log class. This has to be done because those individual objects will be arguments of algorithms capable of performing matching between them (e.g., comparing start/end points, time ranges and expiration times).

Users and Drivers are considered as subclasses of a generic Person class. It is important to differentiate between the two roles in the system, because they are linked to different classes (respectively, ride requests and ride offers); in addition to that, the Driver is linked to a Car object, which is not necessary for an User who is just requesting rides.

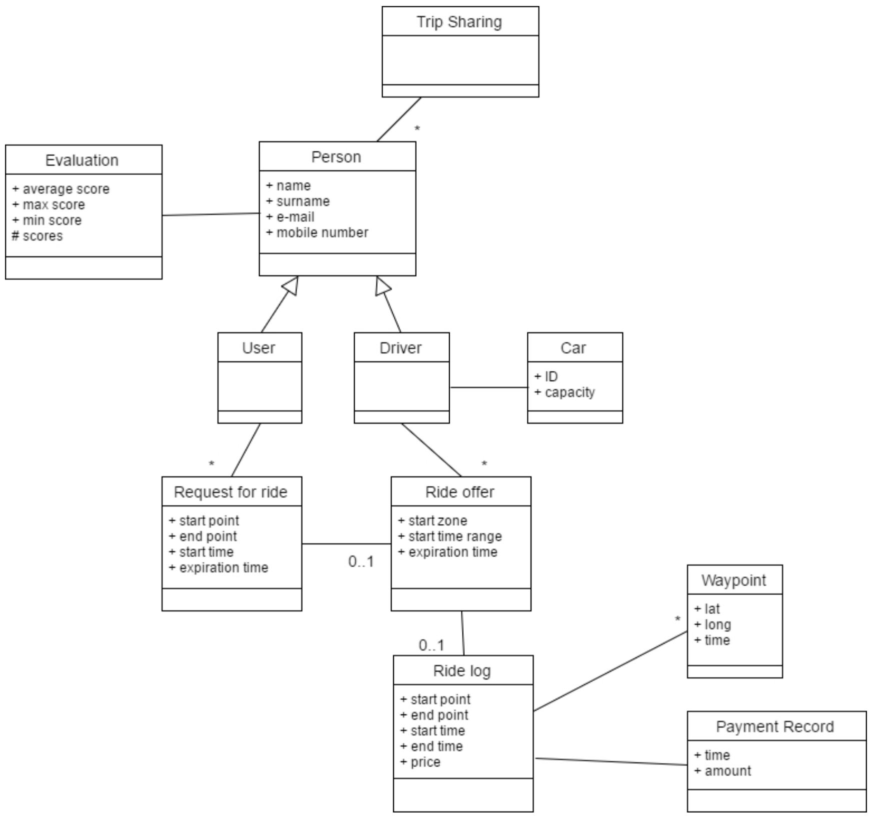


Figure 2.3: Trip Sharing Application - UML Class Diagram

To track and monitor rides, a Ride Log class is used, attached to the Ride Offer (if and only if it has been coupled to a Ride Request). The Ride Log contains a set of Waypoints (featuring coordinates about places through which the ride has passed) and the Payment Record certifying that the payment has been performed at the end of the ride.

The peer evaluation feature of the system is represented by the Evaluation class, that contains the average score for each Person enrolled in the system, computed each time he /she performs a drive.

2.3 Virtual Money Account

A vacation resort has many places where a customer can buy things or services for small amounts of money (ex. drinks, ice creams, towels, etc.). Having to bring cash or credit cards for paying is inconvenient for customers.

A virtual money system replaces cash or credit cards with ‘virtual money’. Each customer, upon arrival in the resort, receives a bracelet with a unique ID written on the bracelet as a bar code. The customer can then buy (using cash or credit card) virtual money to be attached to the ID. Then he can spend, buy, virtual money, exchange it back for real money. In all cases he has just to show the ID.

In the resort, each point of sale of a service or good must be equipped with a register that reads the ID on the bracelet and performs a sale transaction withdrawing virtual money from the bracelet / ID.

In the resort some specific points (exchange points) must be equipped for buying /selling virtual money versus cash or credit cards.

Solution

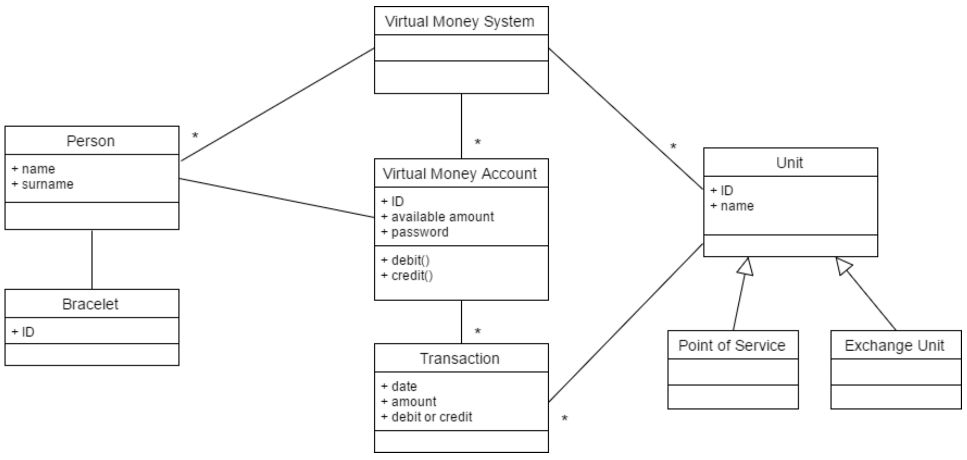


Figure 2.4: Virtual Money Account - UML Class Diagram

It is important that the physical customer (represented by the Person class) is not coincident with the logical user of the system (represented by the Virtual Money Account class). There is no need of direct link between the Bracelet class and the Virtual Money Account class, since they are both reachable through a Person object.

Transactions cannot be considered as an array reported in a Virtual Money Account object, but they need a class to be represented. Each transaction can be performed by a specific Unit, which can be a Point of Service or an Exchange Unit. Since there is no difference in the way Points of Service and Exchange Units are linked with Transactions and the Virtual Money System, they can be considered as subclasses of the Unit class. No attributes are evident from the system description given; however, a possible attribute or Exchange Unit should be ”change rate”, meaning how many virtual credits can be obtained for an unit of cash (e.g., 1eu.).

2.4 Bus Information

Consider a metropolitan area with many bus lines and bus stops. A display at each bus stop shows the estimated waiting time for all the bus lines stopping there.

Besides, a user can obtain the same information sending an sms to a certain number, with the ID of the bus stop.

In the following you should analyze and model the application that provides waiting time information for buses.

Solution

A bus line is a sequence of bus stops (e.g., line 13 stops in bus stops 10, 3 and 24). A bus serves on a certain bus line, and at any time has a certain position. The bus line defines the path a bus must follow and is therefore essential to estimate arrival times.

The simplest algorithm that can be used is speed = distance / time and therefore estimated time = remaining distance to next bus stop / average bus speed between bus stop x and bus stop y.

The stop at a certain bus stop is modeled through an Association class, containing the estimated waiting time (computed using the algorithm above) and the actual stop time, measured on board by the system.

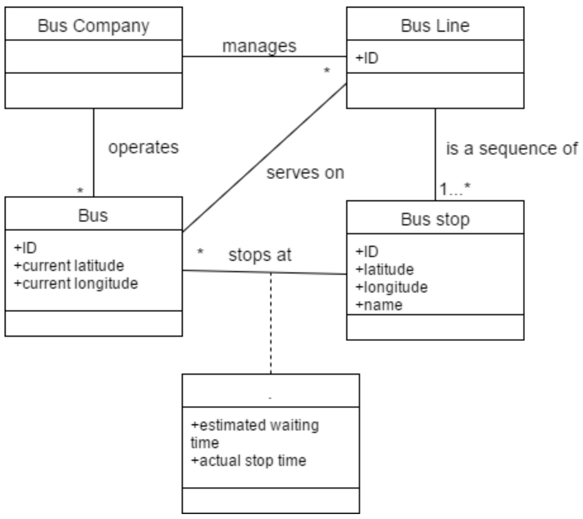


Figure 2.5: Bus Information - UML Class Diagram

2.5 Web Wedding Wishlist

In weddings, a group of people aggregate to select, buy and send gift(s) to the couple. This process was paper based: for instance a couple going to be married selected gifts in a specific shop (the wishlist), friends came to the shop, decided what gift to send, and paid for it; the shop maintains the wishlist, receives payments, and finally sends the gifts.

Nowadays this can be web based. The main functions to be provided are:

- Management of the process (start and end dates, close / open group of friends, etc);
- Management of wishlist;
- Management of groups (couple, friends);
- Payments.

In the following you should analyze and model the web wishlist system.

Solution

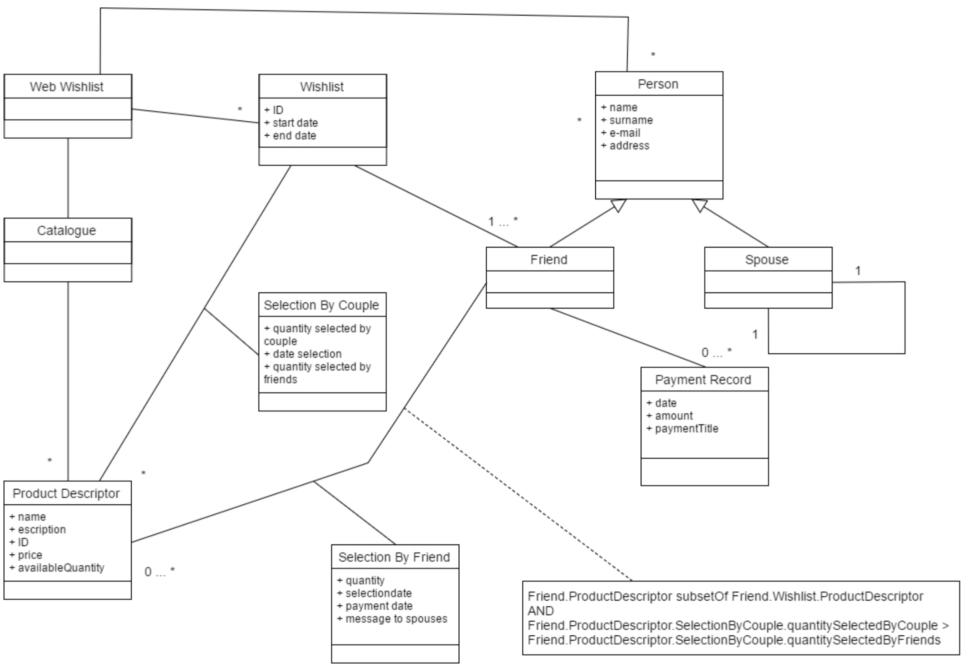


Figure 2.6: Web Wedding Wishlist - UML Class Diagram

The key elements of the solution are: a Catalogue of different items, each one described by a ProductDescriptor object; the Wishlist, which contains a number of product descriptors, and that is linked to a couple of spouses; the Person class, having as subclasses the Spouses, in charge of creating the wishlist, and the Friends, in charge of performing transactions to buy elements of a wishlist.

The link between the Product Descriptors, Wishlists and Friends are described by two UML association classes: Selected By Couple and Selected By Friend. The former one gives info about the number of products selected by the couple in their wishlist, and the amount of those products bought by friends. The latter gives information about the products bought by a friend, about the payment method and date and a possible message to the spouses. The Selection By Friend association is ruled by two conditions, guaranteeing that the item is part of the wishlist and that the amount of products bought by friends are less than the amount requested by the couple in first instance.

To each friend, one or more payment records are associated.

2.6 Eating Support Application

Many applications are already available to support people in monitoring their eating.

Each time the user eats something he/she can use the application to log what she has eaten. There are at least three ways of doing this.

- The user looks for the food eaten (ex: hamburger and fries, or pasta carbonara etc) in a predefined list; then the user can customize the quantity of the food eaten (ex 150g pasta carbonara).
- The user enters the quantities of basic ingredients eaten (butter 10grams, sugar 15 grams, etc); the user can customize the lists, adding foods and ingredients.
- Finally, the user can select a meal. A meal is made of a number of foods (ex hamburger and fries + muffin, or pasta carbonara + green salad). The application provides a list of meals. The user can also define and add complete meals to the list of meals.

Having logged what he has eaten over time, the user can monitor his eating habits, in various ways: calories eaten per week, per day, per meal; number of meals per day and time interval between them; basic nutrients (carbohydrates, saturated fat, non saturated fat, proteins, vitamins, minerals, etc) per week, per day, per meal.

Besides the above monitoring function, the application allows the user to compare her eating habits with some predefined habits. A habit is defined in term of RDA (Recommended Dietary Allowance) per calories and per basic nutrient (ex RDA protein = 150 g, RDA saturated fat 20g, etc). Again the application provides some predefined good eating habits customized in function of sex, age, height and living style (athlete, manual worker, non manual worker etc); plus the capability of defining new habits.

The application could also allow the user to define a monthly /weekly/ daily diet. A diet is made of foods. Then the user can compare her eating habits with the defined diet.

Define the key concepts and entities and their relationships (UML class diagram) for the application.

Solution

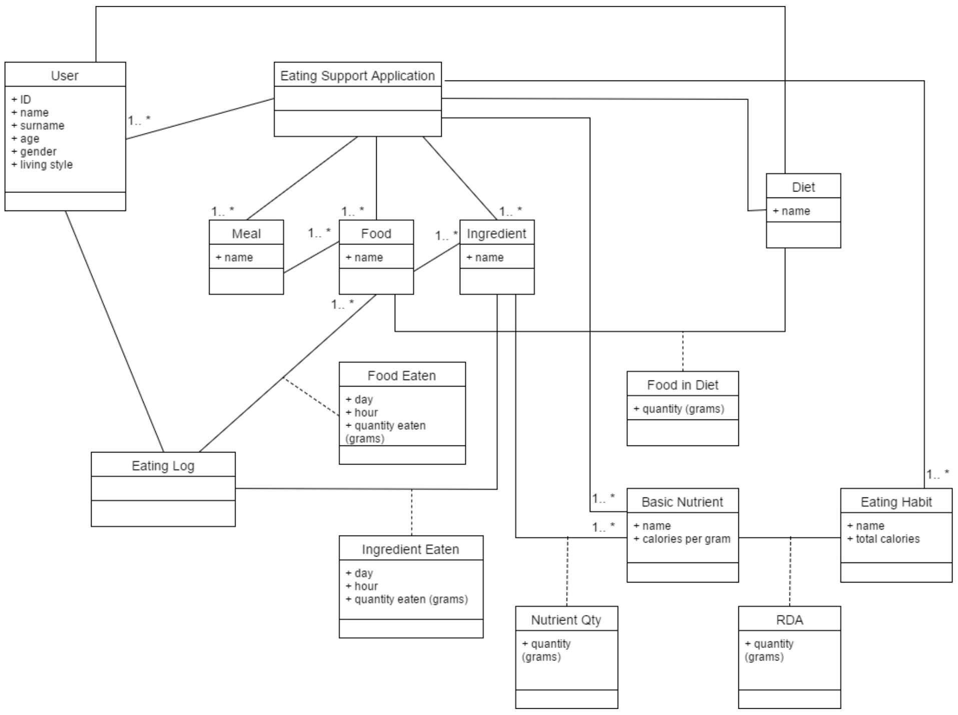


Figure 2.7: Eating Support Application - UML Class Diagram

The basic classes are Meal, Food, Ingredient and Basic Nutrient. They contain what is possible to eat, at different levels of detail, the lowest being the Basic Nutrient. The relationship between one level and the next details the quantities that are involved (e.g., if Ingredient = Butter, 100g butter contain 90g saturated fat and 10g water). For the sake of simplicity, this information -which could have been represented using UML association classes- is not shown in the Class Diagram. Thanks to these associations, given a meal, a food or an ingredient, and a quantity, it is possible to calculate the basic ingredients contained in them.

The Eating Log class is used to take trace of what is eaten by a user, in terms of Foods and Ingredients. The association class between Eating Log and Food/Ingredient reports the quantity eaten. In alternative to the association class, the relationship could have been modeled as an intermediate class. It is important that the quantity eaten is not considered as an attribute of the Ingredient or Food class.

The RDAs are an Association Class between Eating Habit and Basic Nutrient, since the person's habits (e.g., healthy or fit life style) require different sets of basic nutrients to be eaten.

The Diet class contains a list of Foods that can be eaten. Even though it has not been shown to keep the UML Class Diagram simpler, also in this case an association class would be needed to represent the quantity of each food considered in a Diet. In addition to the relationship with Food, Diet could also be connected with Meals and individual Ingredients. Also in this case, the relationships have not been shown to keep the diagram simpler.

2.7 Garden Watering Controller

A watering controller is a device that opens / closes a water valve when defined by the user. The water valve sections a water pipe that waters a garden (or anything else). In the following you should analyze and model a controller with these characteristics:

- The user interacts via a simple user interface (a few buttons for input, 3-4 inches LCD black and white screen);
- The controller controls 2 zones, in other words controls 2 independent water valves;
- The user can, for each zone: open / close directly the valve; use a default watering program (open 10 minutes per day at 10 pm, every day); use a simple watering program (one watering period per day, the user defines the start time and the duration); use a custom water program (x watering periods per day, every y hours, each of duration z); use a weekly custom water program (same as above but each day in the week can have different parameters);
- The controller can be connected to an external rain sensor (bipolar connector, 0-3V, if closed no rain). In case of rain no watering program is applied.
- The controller is battery operated.

Define the glossary (key concepts and their relationships) using an UML Class Diagram.

Solution

The two different water valves are modeled in the Class Diagram by the Zone class, each one attached to a current program.

Different programs (default, simple, custom daily, custom weekly) are considered as sublcasses of the parent class Predefined Program.

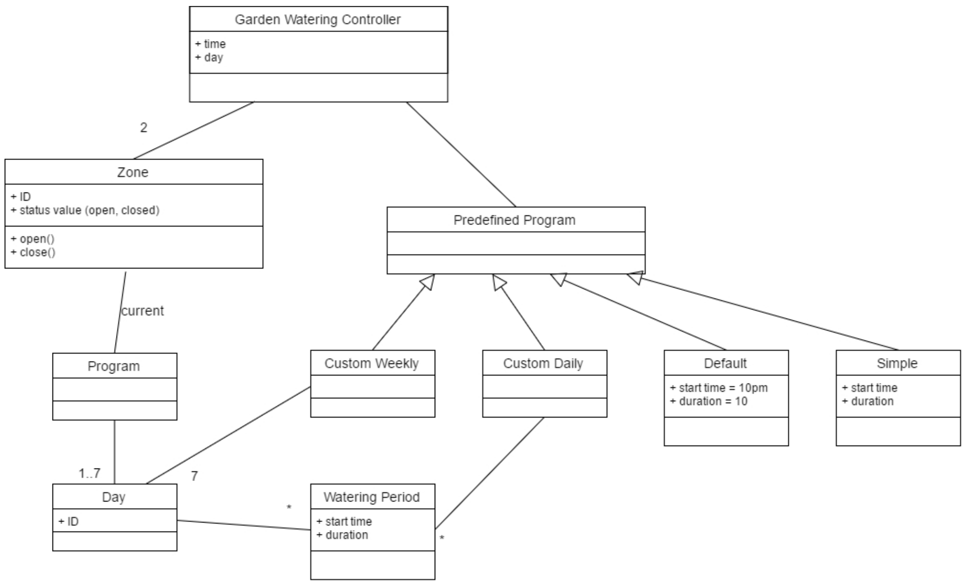


Figure 2.8: Garden Watering Controller - UML Class Diagram

Hardware elements (like the buttons and the screen of the User Interface, the external rain sensor, and the batteries) are not needed in the logical Class Diagram, even though it would not be a mistake to include them. All those hardware elements belong in fact in the system design model (that can also be represented using a UML class diagram).

2.8 Parcel Handling and Tracking

A logistic company (e.g. DHL, UPS) handles shipping of parcels all over the world. A key function is tracking the position and status of each parcel, from reception to delivery.

Each parcel is associated to a waybill, that contains name and address of sender, name and address of addressee, (the person to whom the parcel is sent) and a unique ID (also called tracking number). The waybill (and notably the ID written as bar code) is printed and attached to the parcel.

The parcel, from origin to destination, is routed through a number of intermediate points. In term of graphs, the parcel follows a path (sequence of nodes) on a graph. Routing the parcel means deciding the path. Tracking the parcel means collecting the actual sequence of nodes followed by the parcel, with the time of reception in each node. This information is collected through a specific device, that, upon reception of the parcel at a node, reads the bar code, and sends to a server time, bar code, location (i.e. node). The device is custom made, but in fact it can be seen as a modified smartphone (GPS for the location, bar code reader, internet connection to a server). The device also has a touch screen used to collect the signature of the addressee who receives the parcel at delivery (this signature will be the proof of delivery). The tracking system has two types of users: employees of the company in charge of handling parcels, and end users who access the system via a web site. On the web site a user, entering a tracking number, can follow the shipping of a parcel, until delivery. Tracking information for each parcel is kept for 90 days.

Define the glossary (key concepts and their relationships, using a UML class diagram) for the system.

Solution

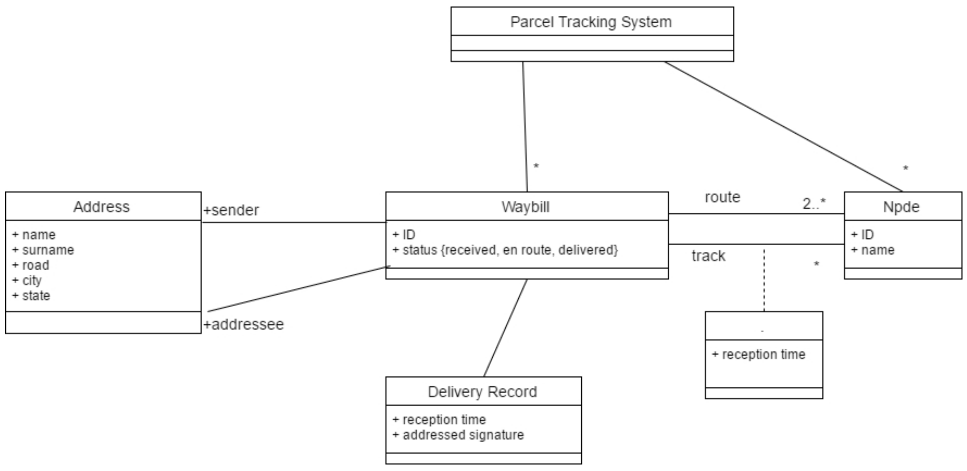


Figure 2.9: Parcel Handling and Tracking - UML Class Diagram

Each Node object represent a node of the network of the shipping company.

For any parcel, the planned route is made by an ordered sequence of nodes (at least 2). The actual track (that may be different from the planned one) followed by a parcel is also an ordered sequence of nodes. Therefore, two different relationships must be present between the Waybill class and the Node class. The association between Waybill and Node for the composition of the actual percurred track requires an additional piece of information, the reception time. Hence, a UML association class is needed.

A similar double relationship happens between the Waybill and the Address class. Both sender and addressee are objects of the Address class.

3

Software Testing

3.1 Black Box Testing

3.1.1 Triangle Type

Define black box tests for the following function, using equivalence classes and boundary conditions.

int triangleType(int side1, int side2, int side3);

This function receives three integers that represent the lenght of the three sides of a triangle. It returns 1 if the three sides define an equilateral triangle, 2 if isosceles, 3 if scalene, -1 if the three sides cannot be composed in a triangle.

Examples:
triangleType(1, 1, 1) → 1
triangleType(1, 1, 3) → 2
triangleType(3, 4, 5) → 3
triangleType(1, 1, 2) → -1

Solution

Since the input values represent the lengths of the three sides of a triangle, they have to be strictly positive integers. Thus, we can define two equivalence classes for each of the variables: $[minint, 0]$ and $[1, maxint]$. We do not have any upper limit for the variable values. We need eight different test cases to test all the possible combinations of input equivalence classes.

All combinations of equivalence classes are invalid test cases except for all three inputs belonging to $[1, maxint]$. In this case, we need to provide additional valid test cases to cover any of the possible outputs of the function: *equilateral* (1), *isosceles* (2), *scalene* (3), and *impossible* (-1). In the case of the *isosceles* output, we can provide additional test cases for different

side1	side2	side3	Not a triangle	# equal sides	Position of equals	V/I	Test cases
[minint, 0]	[minint, 0]	[minint, 0]	-	-	-	I	(-1,-1,-2) err B:(0,-1,-2) err
”	”	[1, maxint]	-	-	-	I	(-1,-1,2) err
”	[1, maxint]	[minint, 0]	-	-	-	I	(-1, 1, -2) err
”	”	[1, maxint]	-	-	-	I	(-1, 1, 2) err
[1, maxint]	[minint, 0]	[minint, 0]	-	-	-	I	(1, -1, -2) err
”	”	[1, maxint]	-	-	-	I	(1, -1, 2) err
”	[1, maxint]	[minint, 0]	-	-	-	I	(1, 1, -2) err
”	”	[1, maxint]	T F F F F F	- 0 3 2 2 2	- - - A A C A C A C A A	V V V V V V	(1, 1, 2) -1 (3, 4, 5) 3 (1, 1, 1) 1 (2, 2, 3) 2 (2, 3, 2) 2 (3, 2, 2) 2

Table 3.1: Solution of exercise 3.1.1

positions of the two equal values among the three inputs of the function.

Boundary conditions should be provided with values equal to 0 and 1 for any of the input values.

3.1.2 Compute Calories

Define black box tests for the following class, using equivalence classes and boundary conditions.

double computeCaloriesFood(int weightProteins, int weightFats, int weight-Carbohydrates);

The function receives the weight in grams of proteins, fats, carbohydrates in a food and computes the total amount of calories given. 1g protein = 4 calories, 1g carbohydrates = 4 calories, 1g fats = 9 calories.

Example: computeCaloriesFood(1, 2, 3) $\rightarrow$ 1*4 + 2*9 + 3*4 = 34 calories.

Solution

The values provided as input to the function represent quantities (in grams) of nutritional elements in foods. Thus, they must be greather or equal than zero. For each of the variables, two equivalence classes can be defined: $[minint, -1]$ and $[0, maxint]$. Eight test cases (2*2*2) are needed to explore all the possible combination of these equivalence classes. Only the test case with all the values of the variables belonging to the range $[0, maxint]$ should produce a valid output. All other ones should produce an error.

Test cases must be provided for boundary conditions also. -1 and 0 are boundary values for all three variables. In the table, a (not exhaustive) set

weightProteins	weightFats	weightCarbo-hydrates	Valid / Invalid	Test cases
[minint, 0[	[minint, 0[	[minint, 0[	I	T(-10,-10,-10) err B: T(-1,0,0) err
"	"	[0, maxint]	I	T(-10,-10,100) err
"	[0, maxint]	[minint, 0[	I	T(-10,10,-10) err
"	"	[0, maxint]	I	T(-10,10,10) err
[0, maxint]	[minint, 0[	[minint, 0[	I	T(10,-10,-10) err
"	"	[0, maxint]	I	T(10,-10,10) err
"	[0, maxint]	[minint, 0[	I	T(10,10,-10) err
"	"	[0, maxint]	V	T(1,2,3) 34 B: T(0,0,0) 0 B: T(1,0,0) 4

Table 3.2: Solution of exercise 3.1.2

of boundary test cases is provided.

3.1.3 Light Controller

Define black box tests for the following class, using equivalence classes and boundary conditions.

```
Class LightController{
    private boolean onOff;
    private int intensity;
    private void dimUp();
    private void dimDown();
    public void dimDown();
    public void on();
    public void off();
    public void getIntensity(); //getter for Intensity
    public boolean getOnOff(); //getter for onOff
}
```

The class controls a light, that can be on or off. When "on", its intensity can range from 1 to 100. When "off", its intensity is 0. At creation, the light is off. The on() command sets the light on, at intensity 50. Functions dimOn() and dimOff() respectively add / reduce intensity by 10. When a sequence of dimOff() reduces the intensity to zero, the light is switched off.

Solution

onOff is a boolean variable. It can be described with two equivalence classes, off and on. Intensity values can be split in four equivalence classes: $[minint, 0[$, 0, $[1, 100]$, $]100, maxint]$.

To test all the possible combinations of the equivalence classes, at least eight test cases must be provided. Values belonging to the first and fourth classes always generate invalid test cases, since, according to the definition of the class behaviour, the intensity must be positive and cannot be greater than 100. Valid test cases are generated when light is “off” and intensity is 0, and when light is “on” and intensity belongs to $[1, 100]$.

Since we are not testing a single method but a whole class with setters and getters, each individual test case must be composed of a set of consecutive method calls performed on an instance of the class. Only test cases T2 and T7 are valid.

Boundary test cases should be provided for intensity equal to 0, 1 and 100. T2 and T6 serve as boundary test cases for Intensity = 0. T8 serves as boundary test case for Intensity = 100. A test case cannot be provided for intensity equal to 1, since operating with dimDown() and dimUp() it is not possible to obtain an intensity value that is not multiple of 10.

3.1.4 Compute Fee

Define black box tests for the following function, using equivalence classes and boundary conditions.

double computeFee(int duration, int minRate, int minRate2);

This function computes (in euros) the fee for a bicycle rental, using the following parameters:
duration: minutes the bicycle has been used
minRate: cost per minute, in cents of euro
minRate2: cost per minute, in cents of euro

The fee iscomputed as follows: free for the first 30 minutes; minRate per minute for the first hour exceeding the first 30 minutes (i.e., 30 to 90 minutes); minRate2 per minute after 90 minutes.

Examples:
computeFee(35, 10, 20)→ = (35-30)*10 = 50
computeFee(65, 10, 20) → = (65-30)*10 = 350
computeFee(95, 10, 20)→ = 60*10 + (95-90)*20 = 700

Solution

Since we have three different ways to compute the fee based on the number of minutes of the loan, at least three equivalence classes are needed for the variable *duration*: [1, 30], [31, 90] and [90, maxint]. In addition to those, an equivalence class must include negative integers, which lead to invalid results of the function.

No specifications are given about minRate and minRate2. Therefore, it is possibile to suppose just two equivalence classes for them: $[minint, 0]$ and $[1, maxint]$.

To explore all the possible combinations of equivalence classes, at least 16 test cases are needed. Three combinations provide valid result: the ones with duration, minRate and minRate2 being all greater or equal than 0.

Light on, off	Intensity <0, 0, 1-100, >100	Valid	Test Case

Off	[minint, 0[	I	T1: L = new LightController();
			L.getIntensity() == 0;
			L.dimDown(); L.dimDown();
			L.getIntensity() == 0; // fail
			L.getOnOff() == off;

”	0	V	T2: L = new LightController();
			L.getIntensity() == 0;
			L.off();
			L.getIntensity() == 0;
			L.getOnOff() == off;

”	[1, 100]	I	T3: L = new LightController();
			L.getIntensity() == 0;
			L.dimUp();
			L.getIntensity() == 0; //fail
			L.getOnOff() == off;

”	[1, 100]	V	T3b: L = new LightController();
			L.on();
			L.getIntensity() == 50;
			repeat 5 times L.dimDown()
			L.getIntensity() == 0; //impossible to get 1
			L.getOnOff() == off;

”	]100, maxint]	I	T4: L = new LightController();
			L.getIntensity() == 0;
			repeat 20 times L.dimUp();
			L.getIntensity() == 0; //fail
			L.getOnOff() == off;

On	[minint, 0[	I	T5: L = new LightController();
			L.on();
			L.getIntensity() == 50;
			L.getOnOff() == on;
			repeat 20 times L.dimDown()
			L.getIntensity() == 0 //fail;
			L.getOnOff() == off //fail;

”	0	I	T6: L = new LightController();
			L.on();
			repeat 20 times L.dimDown()
			L.getIntensity() == 0;
			L.getOnOff() == off; // fail

”	[1, 100]	V	T7: L = new LightController();
			L.on();
			L.getIntensity() == 50;
			L.getOnOff() = on;

”	]100, maxint]	I	T8: L = new LightController();
			L.on();
			L.getIntensity() == 50;
			L.getOnOff() == on;
			repeat 20 times L.dimUp()
			L.getIntensity() == 100; // fail
			L.getOnOff() == on;

Table 3.3: Solution of exercise 3.1.3

Boundary conditions must be provided for duration = {0, 1, 30, 31, 90, 91}, minRate = {0, 1}, minRate2 = {0, 1}. All test cases with variables equal to zero are invalid.

Comments and possible alternatives

Either duration, minRate or minRate2 can be considered valid if equal to 0. In that cases the equivalence classes and the validity of test cases change. For instance, the equivalence classes for both minRate and minRate2 become [minint, 0[and [0, maxint]. Boundary test cases Tb(60, 10, 0) and Tb(60, 0, 10) become valid.

In a real application, minRate2 could be constrained to be greater than minRate. In such case, 16 test cases are not enough to explore all the possible combinations of inputs. For each combination of equivalence classes, it should be verified whether minRate2 > minRate or not. This would result in 16 additional test cases to be defined. Since this condition is not given explicitly in the description of the function, it is not necessary to discuss it in the solution of the exercise.

duration	minRate	minRate2	Valid	Test Cases
[minint, 0]	[minint, 0]	[minint, 0]	I	T(-10, -10, -20) err
”	”	[1, maxint]	I	T(-10, -10, 20) err
”	[1, maxint]	[minint, 0]	I	T(-10, 10, -20) err
”	”	[1, maxint]	I	T(-10, 10, 20) err Tb(0, 10, 20) err
[0, 30]	[minint, 0]	[minint, 0]	I	T(10, -10, -20) err
”	”	[1, maxint]	I	T(10, -10, 20) err
”	[1, maxint]	[minint, 0]	I	T(10, 10, -20) err
”	”	[1, maxint]	V	T(10, 10, 20) = 0.0 Tb(1, 10, 20) = 0.0 Tb(30, 10, 20) = 0.0
[31, 90]	[minint, 0]	[minint, 0]	I	T(60, -10, -20)
”	”	[1, maxint]	I	T(60, -10, 20) err
”	[1, maxint]	[minint, 0]	I	T(60, 10, -20) err Tb(60, 10, 0) err Tb(60, 0, 10) err
”	”	[1, maxint]	V	T(60, 10, 20) (60-30)*10 = 3.0 Tb(31, 10, 20) = 0.1 Tb(90, 10, 20) = 6.0
[91, maxint]	[minint, 0]	[minint, 0]	I	T(100, -10, -20) err
”	”	[1, maxint]	I	T(100, -10, 20) err
”	[1, maxint]	[minint, 0]	I	T(100, 10, -20) err
”	”	[1, maxint]	V	T(100, 10, 20) 60*10 + 10*20 = 8.0 Tb(91, 10, 20) = 0.1 Tb(100, 1, 1) = 60*1 + 10*1 = 0.7

Table 3.4: Solution of exercise 3.1.4

3.1.5 Compute Water Timing

Define black box tests for the following unction, using equivalence classes and boundary conditions.

int times[] computeWaterTiming(int start, int duration, int nTimes)

This function programs a timer to water gardens. The timer has an internal clock and opens/closes a valve. The timer considers only a one day range. The first parameter defines the time (in minutes) when the valve opens, the second parameter the duration (in minutes) the valve remains open, and the third parameter the number of times the cycle is repeated. nTimes can be only 1, 2 or 3. If the value is 1 the cycle is repeated every 24 hours, if 2 every 12 hours, if 3 every 8 hours. The function returns an array with the specific start / end times, in minutes, when the valve remains open. It returns an error in case the values passed are not meaningful.

Examples
computeWaterTiming(60, 20, 1) → [60, 80, -1, -1, -1, -1]
//open the valve 1 time per day, starting at 1am, for 20 minutes
//first open period from minute 60 to 80

computeWaterTiming(60, 20, 2) → [60, 80, 780, 800, -1, -1]
//open the valve 2 times per day, starting at 1am, for 20 minutes
//first open period from minute 60 to 80, second period from 780 to 800

computeWaterTiming(60, 20, 3) → [60, 80, 540, 560, 1020, 1040]
//open the valve 3 times per day, starting at 1am, for 20 minutes
//first from 60 to 80, second from 540 to 560, third from 1020 to 1040

(Minutes in one day = 60*24 = 1440)

Solution

Since a day is composed by 1440 minutes, a valid value for the *start* variable must be a positive integer lower than 1440. Therefore we define three equivalence classes: [minint, 0[, [0, 1440], [1441, maxInt]. We do the same also for duration, since for the behaviour of the function any watering period cannot be longer than a full day.

Regarding the *Ntimes* variable, the function behaves differently for values 1, 2 and 3. So, we define separate equivalence classes for each of the three values, and two additional ones ([minint, 0] and [4, maxint]) to consider invalid values.

An additional condition must be posed on *duration* and *nTimes*, so that the last watering time does not exceed the lenght of a day. Such condition ($start + duration * nTimes \leq 1440$) is indicated in the solution table as C1.

Valid test cases can be obtained with start and duration belonging to the [0, 1440] interval, nTimes = {1,2,3}, and for C1 = true. All other test cases fail.

Boundary values that should be tested are *start* = {-1, 0, 1440, 1441} (same for duration) and *nTimes* = {0, 4}. A not exhaustive set of boundary test cases is given in the solution.

Comments and possible alternatives

To give an exhaustive coverage of all the possible combinations of the equivalence classes, and the condition on duration and nTimes, 90 test cases (3*3*5*2) should be provided. In this and other cases in which the exhaustive numbers of test cases is very large, it makes sense to operate a pruning of the combinations of equivalence classes. Since a negative value for *start* and *duration* invalidates all possible test cases (no matter which value other variables are given) it is possible to stop the exploration when the [minint, 0[case is considered.

Since the ranges of the variables *start* and *duration* are not explicitly defined, it is also possible to consider 0 as an invalid value for both. In this case, obviously, the equivalence classes for both would change to [minint, 0], [1, 1440] and [1441, maxint].

In this case, the condition C1 could also be considered in the equivalence classes for *duration*, making them dependant from the values for the variable *start*. However, adding a column to the table makes the discussion of test cases simpler and more readable.

start	duration	nTimes	C1	V	Test Cases
[minint, 0[	*	*	*	I	T(-60, 10, 1) err
[0, 1440]	[minint, 0[	*	*	I	T(60, -10, 1) err
”	[0, 1440]	[minint, 0]	*	I I	T(60, 10, -1) err Tb(60, 10, **0**) err
”	”	1	T	V V I	T(60, 10, 1): [60, 70, -1, -1, -1, -1] Tb(**0**, 10, 1): [0, 10, -1, -1, -1, -1] Tb(**1440**, 10, 1) err
”	”	”	F	I I	T(60, 1400, 1) err Tb(1, **1440**, 1) err
”	”	2	T	V V	T(60, 10, 2): [60, 90, 780, 790, -1, -1] Tb(60, **0**, 2): [60, 60, 780, 780, -1, -1]
”	”	”	F	I	T(60, 1400, 2) err
”	”	3	T	V	T(60, 10, 3): [60, 90, 540, 550, 1020, 1030]
”	”	”	F	I	T(60, 1400, 3) err
”	”	[4, maxint]	*	I I	T(60, 10, 10) err Tb(60, 10, **4**) err
”	[1441, maxint]	*	*	I	T(60, 1500, 1) err
[1441, maxint]	*	*	*	I	T(1500, 10, 1) err

Table 3.5: Solution of exercise 3.1.5

3.1.6 Show Picture

Define black box tests for the following class, using equivalence classes and boundary conditions.

int showPicture(int lat, int long, char* filename)

This C function receives the full pathname of a file in Windows format (filename) that contains a picture in jpg format. It receives also latitude and longitude of the picture, expressed as integers between -1 and 1. If the parameters are acceptable the function shows the picture on screen and returns 1, otherwise returns 0.

Examples
showPicture(0, 0, C:/Pictures/mountain.jpg) → 1
showPicture(-2, 0, C:/Pictures/mountain.jpg) → 0

Solution

This function receives as parameter a string containing a filename. In this case a simple solution is to consider just two equivalence classes: a string containing a currect filename, and a string containing a wrong one.

Since *latitude* and *longitude* lie between -1 and 1, three equivalence classes must be considered for both the variables: [minint, -1[, [-1, 1] and]1, maxint]. Combined with the ones for *filename*, these classes create 18 (3*3*2) combinations for which a test case must be provided. Only one combination provides a valid test case.

Boundary values to be tested for *latitude* and *longitude* are -1 and 1 (valid test cases) and -2 and 2 (invalid test cases).

lat	long	filename	Valid	Test case
[minint, -1[	[minint, -1[	not correct	I	T(-10, -10, XDE#mountain.jpg) 0
”	”	correct	I	T(-10, 10, C:/Pictures/mountain.jpg) 0
”	[-1, 1]	not correct	I	T(-10, 0, XDE#mountain.jpg) 0
”	”	correct	I	T(-10, 0, C:/Pictures/mountain.jp)0
”	]1, maxint]	not correct	I	T(-10, 10, XDE#mountain.jp) 0
”	”	correct	I	T(-10, 10, C:/Pictures/mountain.jp) 0
[-1, 1]	[minint, -1[	not correct	I	T(0, -10, XDE#mountain.jp) 0
”	”	correct	I	T(0, -10, C:/Pictures/mountain.jp) 0
”	[-1, 1]	not correct	I	T(0, 0, XDE#mountain.jp) 0
”	”	correct	V	T(0, 0, C:/Pictures/mountain.jp) 1 Tb(**-1**, 0, C:/Pictures/mountain.jp) 1 Tb(**1**, 0, C:/Pictures/mountain.jp) 1 Tb(0, **-1**, C:/Pictures/mountain.jp) 1 Tb(0, **1**, C:/Pictures/mountain.jp) 1
”	”	not correct	V	T(0, 0, XDE#mountain.jp) 0
”	]1, maxint]	not correct	I	T(0, 10, XDE#mountain.jp) 0
”	”	correct	I	T(0, 10, C:/Pictures/mountain.jp) 0
]1, maxint]	[minint, 1]	not correct	I	T(10, -10, XDE#mountain.jp) 0
”	”	correct	I	T(10, -10, C:/Pictures/mountain.jp) 0
”	[-1, 1]	not correct	I	T(10, 0, XDE#mountain.jp) 0
”	”	correct	I	T(10, 0, C:/Pictures/mountain.jp) 0
”	]1, maxint]	not correct	I	T(10, 10, XDE#mountain.jp) 0
”	”	correct	I	T(10, 10, C:/Pictures/mountain.jp) 0

Table 3.6: Solution of exercise 3.1.6

Comments and possible alternatives

To make the discussion of the *filename* more complete, it is possible to define the following equivalence cases taking into account additional possibilities:

Wrong / right syntax;
File exists / not exists;
File is jpeg / is not jpeg;

3.1.7 Can Participate

Define black box tests for the following function, using equivalence classes and boundary conditions. The function computes if a researcher can participate to a selection to become professor, in function of certain conditions.

boolean can_participate(int teaching_hours, int different_courses, intNA, int EA)

Input:
teaching_hours: number of hours taught;
different_courses: number of different courses where the person has taught;
NA: number of articles published;
EA: experience acquired, as number of years in doing research or teaching.

Output:
TRUE if teaching_hours $>$ 1500 and different_courses $>$ 3 and NA_normalized $>$ 40 (with NA_normalized = NA if EA $\leq$ 10, NA_normalized = NA*10/EA if EA $>$ 10)   FALSE otherwise

Examples:
can_participate(1600, 4, 60, 11) = TRUE
can_participate(1000, 4, 60, 11) = FALSE
can_participate(1600, 2, 60, 11) = FALSE
can_participate(1700, 7, 30, 5) = FALSE
can_participate(1600, 3, 30, 8) = FALSE

Solution

Teaching hours must be a positive integer for the test cases to be valid. Among positive integers, a distinction can be made between values smaller or bigger than 1500, that determine the result provided by the function. Therefore three equivalence classes are defined for the variable *teaching_hours*: [minint, 0[, [0, 1500], [1501, maxint]. The same reasoning can be done for *different_courses*, defining the equivalence classes [minint, 0], [1, 3] and [4, maxint].

NA is the number of articles published and must be a positive (or null) integer for the function to work, the equivalence classes [minint, 0] and [1, maxint] are considered for it.

Based on the value of EA (smaller or greater than 10) the output of the function changes. EA must be a positive integer for the test case to be vlid. Therefore, three different equivalence classes are defined for *EA*: [minint, 0], [1, 10], [11, maxint].

NA_normalized is a derived parameter from EA and NA. It cannot be a negative number, so only two equivalence classes need to be considered: [1, 40] and [41, maxint].

The total number of combinations of equivalence classes is 3*3*3*2*2 = 108. For this reason some combinations are pruned from the table at the first invalid range encountered. After the pruning, 31 combinations are obtained. Among them, 16 yield valid test cases. For reasons of available space, test cases are detailed outside the table.

Boundary test cases must be provided for *teaching_hours* = {0, 1, 1500, 1501}, *diff_courses* = {0, 1, 3, 4}, *NA* = {0, 1}, *NA_norm* = {1,40,41}. A not exhaustive set of boundary test cases is provided in the table.

Test cases:
T1: can_participate(-100, 2, 5, 5) err;
T2: can participate(100, -2, 5, 5) err;
T3: can participate(100, 2, 5, -5) err;
T4: can_participate(100, 2, -5, 5) err;
T5: can_participate(100, 2, 20, 1) false;
T6: can_participate(100, 2, 50, 1) false;
T7: can_participate(100, 2, -5, 15) err;
T8: can_participate(100, 2, 35, 15) false;
T9: can_participate(100, 2, 75, 15) false;
T10: can_participate(2000, 5, 5, -5) err;
T11: can_participate(2000, 5, -5, 5) err;
T12: can_participate(2000, 5, 20, 1) false;
T13: can_participate(2000, 5, 50, 1) false;
T14: can_participate(2000, 5, -5, 15) err;
T15: can_participate(2000, 5, 35, 15) false;
T16: can_participate(2000, 5, 75, 15) false;
T17: can participate(2000, -2, 5, 5) err;
T18: can participate(2000, 2, 5, -5) err;
T19: can_participate(2000, 2, -5, 5) err;
T20: can_participate(2000, 2, 20, 1) false;
T21: can_participate(2000, 2, 50, 1) false;
T22: can_participate(2000, 2, -5, 15) err;
T23: can_participate(2000, 2, 35, 15) false;
T24: can_participate(2000, 2, 75, 15) false;
T25: can_participate(2000, 5, 5, -5) err;
T26: can_participate(2000, 5, -5, 5) err;

Teaching hours	Diff courses	EA	NA	NA norm	V/I	Test case
[minint, 0]	*	*	*	*	I	T1 → err Tb1 → err
[0, 1500]	[minint, 0]	*	*	*	I	T2 → err Tb2→ err
”	[1, 3]	[minint, 0]	*	*	I	T3 → err Tb3 → err
”	”	[1, 10]	[minint, 0]	*	I	T4 → err Tb4 → err
”	”	”	[1, maxint]	[1, 40]	V	T5 → F Tb5 → F Tb6 → F
”	”	”	”	[41, maxint]	V	T6 → F
”	”	[11, maxint]	[minint, 0]	*	I	T7 → F
”	”	”	[1, maxint]	[1, 40]	V	T8 → F
”	”	”	”	[41, maxint]	V	T9 → F
”	[4, maxint]	[minint, 0]	*	*	I	T10 → err
”	”	[1, 10]	[minint, 0]	*	I	T11 → err
”	”	”	[1, maxint]	[1, 40]	V	T12 → F
”	”	”	”	[41, maxint]	V	T13 → F
”	”	[11, maxint]	[minint, 0]	*	I	T14 → err
”	”	”	[1, maxint]	[1, 40]	V	T15 → F
”	”	”	”	[41, maxint]	V	T16 → F
[1501, maxint]	[minint, 0]	*	*	*	I	T17 → err
”	[1, 3]	[minint, 0]	*	*	I	T18 → err
”	”	[1, 10]	[minint, 0]	*	I	T19 → err
”	”	”	[1, maxint]	[1, 40]	V	T20 → F
”	”	”	”	[41, maxint]	V	T21 → F
”	”	[11, maxint]	[minint, 0]	*	I	T22 → F
”	”	”	[1, maxint]	[1, 40]	V	T23 → F
”	”	”	”	[41, maxint]	V	T24 → F
”	[4, maxint]	[minint, 0]	*	*	I	T25 → err
”	”	[1, 10]	[minint, 0]	*	I	T26 → err
”	”	”	[1, maxint]	[1, 40]	V	T27 → F
”	”	”	”	[41, maxint]	V	T28 → T
”	”	[11, maxint]	[minint, 0]	*	I	T29 → err
”	”	”	[1, maxint]	[1, 40]	V	T30 → F
”	”	”	”	[41, maxint]	V	T31 → T

Table 3.7: Solution of exercise 3.1.7

T27: can_participate(2000, 5, 20, 1) false;
T28: can_participate(2000, 5, 50, 1) true;
T29: can_participate(2000, 5, -5, 15) err;
T30: can_participate(2000, 5, 35, 15) false;
T31: can_participate(2000, 5, 75, 15) true;

Some examples of boundary test cases:
Tb1: can_participate(**0**, 2, 5, 10) err;
Tb2: can_participate(100, **0**, 5, 10) err;
Tb3: can_participate(100, 2, 5, **0**) err;
Tb4: can_participate(100, 2, **0**, 10) err;
Tb5: can_participate(100, 2, 1, 5) false; (NA_norm = **1**)

Tb6: can_participate(100, 2, 40, 5) false; (NA_norm = **40**)

Comments and possible alternatives

It is also possible to make a preliminary pruning of all the test cases with variables that ensure invalidity. They can all be specified at the end of the table.

Teaching hours	Diff courses	EA	NA	NA norm	V/I	Test case
[minint, 0]	*	*	*	*	I	-
*	[minint, 0]	*	*	*	I	-
*	*	[minint, 0]	*	*	I	-
*	*	*	[minint, 0]	*	I	-
...	...	...	...	...	...	...

After an header like the described one, only valid test cases will be described inside the table.

3.1.8 Luggage

Define black box tests for the following function, using equivalence classes and boundary conditions.

```
boolean acceptLuggage(int nLuggage, int length1, int width1, int depth1,
        int weight1, int length2, int width2, int depth2, int weigth2);
```

The function decides if luggage for a flight is accepted. The rules are:
- max 2 pieces of luggage;
- for each piece of luggage, sum of the three dimensions must be $\leq$ 300cm;
- the total weight of all luggage must be $\leq$ 30kg;

The parameters are as follows:
- nLuggage: number of pieces of luggage of the passenger;
- length1, width1, depth1: dimensions of first luggage, in cm;
- length2, width2, depth2: dimensions of second luggage, in cm;
- weight1, weight2: weight of luggage, in kg.

Examples:
acceptLuggage(2, 100, 50, 10, 15, 110, 60, 15, 10) $\rightarrow$ accept
(total weight 25, dimensions 160 and 185)
acceptLuggage(2, 100, 50, 10, 28, 110, 60, 15, 10) $\rightarrow$ not accept
(total weight 38, dimensions ok)

Solution

All test cases are invalid if any variable is not positive. For the *nLuggage* variable, four ranges can be defined: [minint, 0], 0, 1, 2 and [3, maxint], with the first and last one yielding invalid results. For all other variables, two ranges are defined: [minint, 0] and [0, maxint].

In addition to the ranges for each of the variables, three conditions must be considered: $totalDim1 \leq 300$, $totalDim2 \leq 300$, $totalWeight \leq 30$. The second one must be checked only if $nLuggage = 2$. On the other hand, if $nLuggage = 0$, none of the conditions have to be checked.

If $nLuggage = 1$, it is not necessary to take into account all the possible combinations for values of variables pertaining the second luggage. If $nLuggage = 0$, it is not necessary to take into account any of the other variables.

The total number of combinations of the mentioned ranges is 5 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 = 1280. This number would be made bigger by the consideration of the conditions about the total dimension and total weight. Therefore, the solution table is pruned any time an invalid value is first encountered.

To reduce the extension of the table, *n* has been used in place of *nLuggage* and *l, w, d, k* in place of length, width, depth and weight. *td1, td2 and tw* stands respectively for *totalDim1*, *totalDim2*, *totalWeight*.

Boundary values to be tested are 0 and 3 for *nLuggage*, and 0 for any other variable. Test cases must also be provided to test boundary values for the combined values *totalDim1*, *totalDim2* and *totalWeight*: 300, 301, 30 and 31. Some examples (without exhaustive combinations of them) are given in the table.

Test cases:
T1: acceptLuggage(-10, 10, 10, 10, 10, 10, 10, 10, 10) → error
T2: acceptLuggage(0, 0, 0, 0, 0, 0, 0, 0, 0) → accept
T3: acceptLuggage(1, -10, 10, 10, 10, 10, 10, 10, 10) → error
T4: acceptLuggage(1, 10, -10, 10, 10, 10, 10, 10, 10) → error
T5: acceptLuggage(1, 10, 10, -10, 10, 10, 10, 10, 10) → error
T6: acceptLuggage(1, 10, 10, 10, -10, 10, 10, 10, 10) → error
T7: acceptLuggage(1, 100, 20, 5, 25, 0, 0, 0, 0) → accept
T8: acceptLuggage(1, 100, 20, 5, 35, 0, 0, 0, 0) → not accept
T9: acceptLuggage(1, 100, 150, 70, 25, 0, 0, 0, 0) → not accept
T10: acceptLuggage(1, 100, 150, 70, 35, 0, 0, 0, 0) → not accept
T11: acceptLuggage(2, -10, 10, 10, 10, 10, 10, 10, 10) → error
T12: acceptLuggage(2, 10, -10, 10, 10, 10, 10, 10, 10) → error
T13: acceptLuggage(2, 10, 10, -10, 10, 10, 10, 10, 10) → error
T14: acceptLuggage(2, 10, 10, 10, -10, 10, 10, 10, 10) → error
T15: acceptLuggage(2, 10, 10, 10, 10, -10, 10, 10, 10) → error

n	l1	w1	d1	k1	l2	w2	d2	k2	td1	td2	tw	V	TC
<0	-	-	-	-	-	-	-	-	-	-	-	I	T1: E
0	-	-	-	-	-	-	-	-	-	-	-	V	T2: A
1	≤ 0	-	-	-	-	-	-	-	-	-	-	I	T3: E
”	> 0	≤ 0	-	-	-	-	-	-	-	-	-	I	T4: E
”	”	> 0	≤ 0	-	-	-	-	-	-	-	-	I	T5: E
”	”	”	> 0	≤ 0	-	-	-	-	-	-	-	I	T6: E
”	”	”	”	> 0	-	-	-	-	≤ 300	-	≤ 30	V	T7: A Tb1: A
”	”	”	”	”	-	-	-	-	≤ 300	-	> 30	V	T8: NA Tb2: A
”	”	”	”	”	-	-	-	-	> 300	-	≤ 30	V	T9: NA Tb3: NA
”	”	”	”	”	-	-	-	-	> 300	-	> 30	V	T10: NA
2	≤ 0	-	-	-	-	-	-	-	-	-	-	I	T11: E
”	> 0	≤ 0	-	-	-	-	-	-	-	-	-	I	T12: E
”	”	> 0	≤ 0	-	-	-	-	-	-	-	-	I	T13: E
”	”	”	> 0	≤ 0	-	-	-	-	-	-	-	I	T14: E
”	”	”	”	> 0	≤ 0	-	-	-	-	-	-	I	T15: E
”	”	”	”	”	> 0	≤ 0	-	-	-	-	-	I	T16: E
”	”	”	”	”	”	> 0	≤ 0	-	-	-	-	I	T17: E
”	”	”	”	”	”	”	> 0	≤ 0	-	-	-	I	T18: E
”	”	”	”	”	”	”	”	> 0	≤ 300	≤ 300	≤ 30	V	T19: A Tb4: A Tb5: A
”	”	”	”	”	”	”	”	”	≤ 300	≤ 300	> 30	V	T20: NA
”	”	”	”	”	”	”	”	”	≤ 300	> 300	≤ 30	V	T21: NA Tb6: NA
”	”	”	”	”	”	”	”	”	≤ 300	> 300	> 30	V	T22: NA
”	”	”	”	”	”	”	”	”	> 300	≤ 300	≤ 30	V	T23: NA
”	”	”	”	”	”	”	”	”	> 300	≤ 300	> 30	V	T24: NA
”	”	”	”	”	”	”	”	”	> 300	> 300	≤ 30	V	T25: NA
”	”	”	”	”	”	”	”	”	> 300	> 300	> 30	V	T26: NA
>2	-	-	-	-	-	-	-	-	-	-	-	I	T27: E

Table 3.8: Solution of exercise 3.1.8

T16: acceptLuggage(2, 10, 10, 10, 10, 10, -10, 10, 10) $\rightarrow$ error
T17: acceptLuggage(2, 10, 10, 10, 10, 10, 10, -10, 10) $\rightarrow$ error
T18: acceptLuggage(2, 10, 10, 10, 10, 10, 10, 10, -10) $\rightarrow$ error
T19: acceptLuggage(2, 80, 150, 10, 15, 70, 70, 10, 10) $\rightarrow$ accept
T20: acceptLuggage(2, 80, 150, 10, 15, 70, 70, 10, 20) $\rightarrow$ not accept
T21: acceptLuggage(2, 80, 150, 10, 15, 170, 170, 10, 10) $\rightarrow$ not accept
T22: acceptLuggage(2, 80, 150, 10, 15, 170, 170, 10, 20) $\rightarrow$ not accept
T23: acceptLuggage(2, 180, 150, 10, 15, 70, 70, 10, 10) $\rightarrow$ not accept
T24: acceptLuggage(2, 180, 150, 10, 15, 70, 70, 10, 20) $\rightarrow$ not accept
T25: acceptLuggage(2, 180, 150, 10, 15, 170, 170, 10, 10) $\rightarrow$ not accept
T26: acceptLuggage(2, 180, 150, 10, 15, 170, 170, 10, 20) $\rightarrow$ not accept
T27: acceptLuggage(3, 10, 10, 10, 10, 10, 10, 10, 10) $\rightarrow$ error

Boundary test cases:
Tb1: acceptLuggage(1, 100, 20, 5, 30, 0, 0, 0, 0) $\rightarrow$ accept
Tb2: acceptLuggage(1, 100, 20, 5, 31, 0, 0, 0, 0) $\rightarrow$ not accept
Tb3: acceptLuggage(1, 100, 100, 101, 25, 0, 0, 0, 0) $\rightarrow$ not accept
Tb4: acceptLuggage(1, 100, 100, 100, 10, 10, 10, 10, 10) $\rightarrow$ accept
Tb5: acceptLuggage(1, 10, 10, 10, 10, 100, 100, 100, 10) $\rightarrow$ accept
Tb6: acceptLuggage(1, 10, 10, 10, 10, 100, 100, 101, 10) $\rightarrow$ not accept

Comments and possible alternatives

In case of nLuggage = 0 the function should always return accept, but the value of the other parameters is not defined. In principle, any value could be acceptable. This is a defect in the requirements. In the table, it is assumed that the values for other parameters should be zero.

In case of nLuggage = 1, the values of parameters for the second luggage is not defined. This is another defect in the requirements. In the table it is assumed that the values for the other parameters should be zero.

For length, width and depth, two ranges only (negatives and positives) have been defined. A better option is [minint, 0], [1, 300],]300, maxint]. However, this would increase the number of classes. Similarly, three classes can be defined for the weight of luggages.

3.2 White Box Testing

3.2.1 While Loop

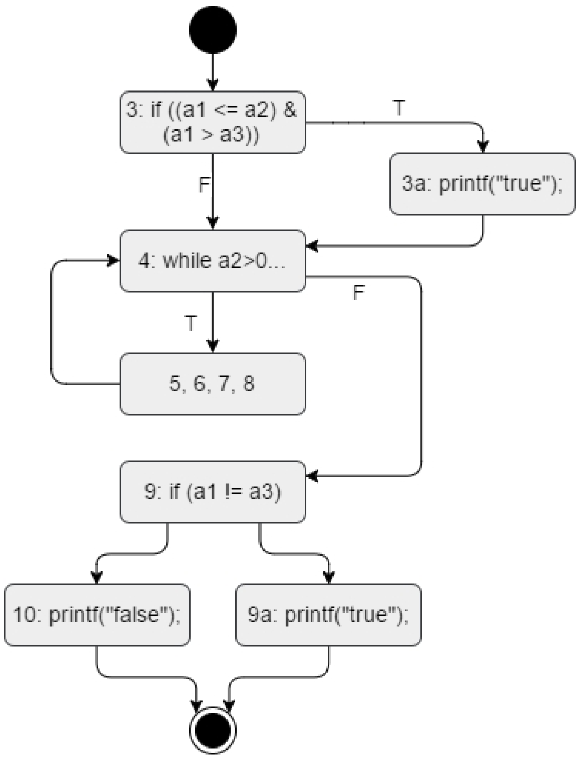
For the following function define the control flow graph, and define test cases to obtain the highest possible node coverage, edge coverage, multiple condition coverage, loop coverage, path coverage. For the test cases, **write only the input value**.

```
01   int f (int a1, int a2, int a3)
02      {
03      if ((a1 <= a2) && (a1 > a3)) printf("true");
04      while (a2 < 0)
05         {
06         a1 = a1 + 1;
07         a2 = a2 + 1;
08         }
09      if (a1 != a3) printf("true");
10      else printf("false");
11      }
```

Discussion

There are two possible alternatives in considering test cases to cover the different conditional statements. If if line 3, if line 9, and while are considered independently, four test cases are needed. Otherwise, if the same test case is considered for the coverage of nodes linked to all the conditional statements, just two test cases are needed. In the table, test cases are provided for a combined consideration of the conditional statements: with T1, the condition of the first if is true, and the condition of the second if is true. With T2 the conditions are both false. Node 5, 6, 7, 8 are covered by both test cases. Thus, T1 and T2 guarantee node coverage. They also guarantee 100% edge coverage.

Path coverage depends on the value of a2. If a2 >= 0, 4 paths are needed. If a2 < 0, 4*|a2| paths are needed. a2 can vary from minint to -1, but an algorithm can be written to generate nearly all test cases (see definition of test cases $Ta2_i$ below, so path coverage is feasible.



Coverage type	TC for 100% coverage	Coverage obtained	Test cases defined
Node	1+1+2 considered statements independently, 2 otherwise	100	T1: 3, 3a, 4, 6, 7, 9, 9a; T2: 3, 4, 6, 7, 9, 10
Edge	2+1+2 considering statements independently, otherwise 2	100	T1: all except 3-4 and 9-10; T2: 3-4 and 9-10
Multiple condition line 3	4	100	T1: TT; T2: TF; T3: FT; T4: FF
Loop line 4	3	100	T1, T2: many times; T3: no enter; T5: enter once
Path	4 if a2 >= 0, 4*|a2| if a2<0; finally 4 + 4*|a2|, for each a2	100	for each (a2 from -1 to minint) produce 4 test cases: $Ta2_1$, $Ta2_2$, $Ta2_3$, $Ta2_4$

Test cases:
T1(-2, -1, -22)
T2(-2, -1, -1)
T3(2, 1, 0)
T4(2, 1, 3)
T5(1, -1, 1)

$Ta2_1$(a2-1, a2, a2-2)
$Ta2_2$(a2-1, a2, -1)
$Ta2_3$(a2+1, a2, 0)
$Ta2_4$(a2+1, a2, 1)

3.2.2 For Loop

For the following function define the control flow graph, and define test cases to obtain the highest possible node coverage, edge coverage, multiple condition coverage, loop coverage, path coverage. For the test cases, **write only the input value**.

```
int function(int x, int z) {
    int k; int j;
    int value = 0;

    if (x < 0) return -1;
    for ( k = 0; k < x; k++) {
        if ( z > x || z < 20)
        {
            for ( j = 0; j < k; j++ ) {

                value ++;
            }
        }
    }
    return value;
}
```

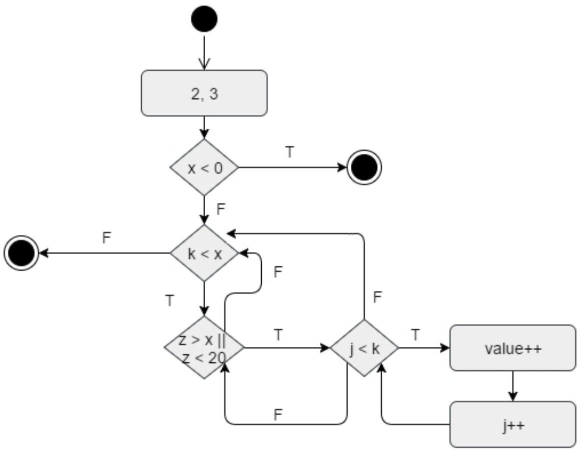
Discussion

There are two if statements in the function: the first one, if the condition x < 0 is met, makes the function return. The other one makes the function proceed through the nested for loop or not. Since there is no else in the second if clause, it is not necessary to explore it with two different test cases to obtain node coverage. Thus, to obtain 100% node coverage two test cases are needed: one with x < 0 and one with x > 0, so that both branches after the first if are taken.

To obtain edge coverage, it is necessary to provide a test case for which the second if statement is true, and a test case for which it is false (always assuming that the outer for loop is entered at least one time).

This function contains two different loops that need both to be tested. The exercise asks for the internal loop (line 9). The number of executions of the loop depends on the value of k, which also depends on the current iteration of the outer for loop. If x = 1, the inner for loop is not executed. With x = 2, the inner for loop is executed just one time (at the second enter of the outer for loop). Finally, with x = 3, the inner for loop is executed multiple times (two times, in correspondence of the second enter of the outer for loop).

Path coverage, in this case, is not feasible because it is proportional with x^2, since the value of x defines the number of executions of both the nested for loops.



Coverage type	TC for 100% coverage	Coverage obtained	Test cases defined
Node	2	100	T1, T2
Edge	3	100	T1, {T2 or T4 or T5}, T6
Multiple condition line 7	4	100	T2, T3: TT; T4: FT; T5: TF; T6: FF
Loop line 9	3	100	T3: no enter; T2: one time; T4: many times
Path	O(x^2)	Not feasible	-

Test cases:
T1(-1, 10)
T2(2, 3)
T3(1, 3)
T4(3, 3)

T5(2, 21)
T6(22, 21)

3.2.3 Array

For the following function define the control flow graph, and define test cases to obtain the highest possible node coverage, edge coverage, multiple condition coverage, loop coverage, path coverage. For the test cases, **write only the input value.**

```
01  int function(int[] array) {
02      int temp;
03      int k = array.length -1;
04      for (int i = 0; i < array.length -1 && i<MAXINT; i++) {
05          if (array[i] > array[i+1]) {
06              temp = array[i];
07              array[i] = array[k];
08              array[k] = temp;
09          }
10      }
11  }
```

Discussion

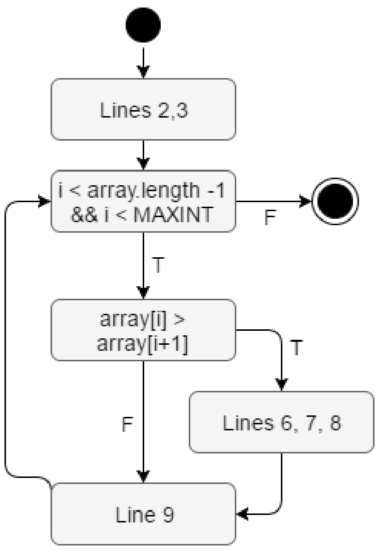
The described function exchanges two consecutive elements inside an array if the first one is bigger than the second one. A single pass is performed on the array, and is managed by the outer for loop, that is executed array.length - 1 times. Inside the for loop, an if statement determines whether the swap of two elements has to be performed or not.

Node and edge coverage are simply obtainable with an array that has length > 2, and that in one case triggers the if statement, and in one case does not. T1({ 2, 1, 3 }) serves for this purpose: in the first iteration of the for loop, array[0] > array[1], hence the "true" edge is taken. In the second iteration, array[1] (now 1) < array[2], hence the "false" edge is taken. All the nodes are also covered by such test cases (obtaining node coverage is possible also with a simpler test case with an array of two elements).

Theoretically, the multiple condition in line 4 would require the test of four different combinations. Since the condition is evaluate for each iteration of the for loop and multiple different results are obtained during the execution of each test case, less than four test cases are needed. T1, defined before, covers the true-true (at the first and second check) and false-true (at the third and final check) possibilities. To cover true-false, an array that has a length greater than MAXINT should be fed to the function. Similarly, to cover false-false, an array with a length equal to MAXINT should be used as input. These two last combinations may be unfeasible in practice, so only 50% of coverage is guaranteed.

The for loop in line four can be covered simply by three different test cases with arrays of different length: an array of length equal to 1 makes the function never enter the loop; with an array of length = 2 as input the function enters the loop once; an array of length > 2 makes the function enter the loop multiple times.

The path coverage in this case would require a huge amount of different test cases, since for any possible array length $2^{array.length-1}$ possible paths are generated. With the test cases defined for the other types of coverage, the path coverage is close to zero.



Coverage type	TC for 100% coverage	Coverage obtained	Test cases defined
Node	1	100	T1
Edge	1	100	T1
Multiple condition line 4	4 or less	50%, 100% may be unfeasible	T1: TT and FT, T4: TT and TF, T5: TT and FF
Loop line 4	3	100	T1, T2, T3
Path	$2^{array.length-1}$	100% not feasible	-

Test cases:
T1(2, 1, 3)
T2(1)
T3(2, 1)
T4(an array longer than MAXINT)

T5(an array as long as MAXINT)

3.2.4 Compute Tax

For the following function define the control flow graph, and define test cases to obtain the highest possible node coverage, edge coverage, multiple condition coverage, loop coverage, path coverage. For the test cases, **write only the input value.**

```
01   private int compute_tax(int wage) {
02
03       int ranges[] = {5000, 15000, 30000};
04       int amount_due = 0;
05       int level = 0;
06
07       for(int i=0; i<3; i++) {
08           if (wage > ranges[i])
09               level++;
10       }
11       if (level == 1)
12           amount_due = 500;
13       else if (level == 2)
14           amount_due = 1500;
15       else if (level == 3)
16           amount_due = 3000;
17       return amount_due;
18   }
```

Discussion

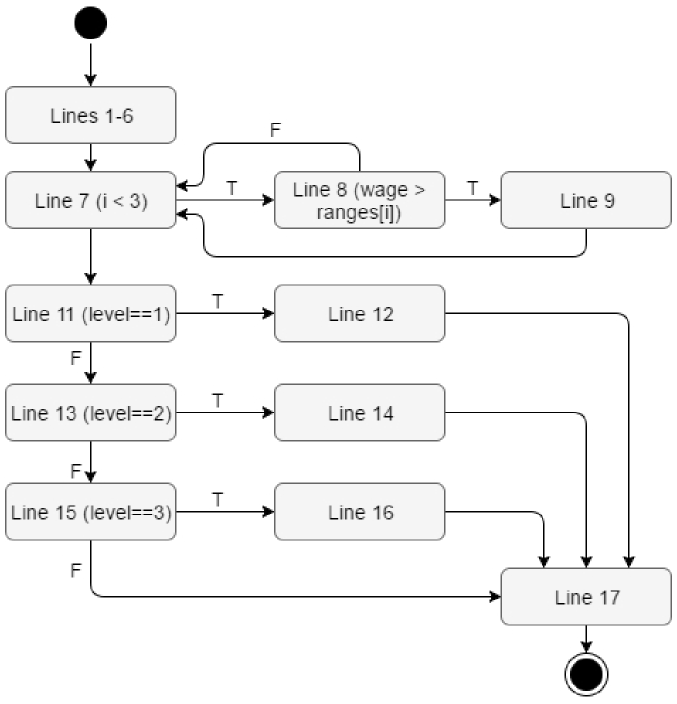
The loop at line 7 does not depend on the input variables, and will be executed three times in any execution of the function. Thus, it is not possible to obtain full loop coverage, but just 33%.

Three test cases are needed to cover with a true condition each of the if else statements, and therefore to obtain 100% node coverage. All the three test cases will have wage at least equal to 5000, and thus all of them will cover node 7, 8 and 9. To guarantee full edge coverage, it is needed a T0 (with a wage $<$ 5000) to cover the edge between line 15 and line 17 (see control flow graph).

Theoretically the paths are 32: 2^3 (if inside the for) x 4 (if-else chain). However, only 4 are feasible, i.e. the ones corresponding to the four test cases defined later. That is because the paths depend on the value of parameter 'wage':
wage $<$ 5000

$5000 < \text{wage} \leq 10000$
$10000 < \text{wage} \leq 15000$
$15000 < \text{wage} \leq 30000$
$\text{wage} > 30000$
So with the same tests defined for node and edge coverage it is also possible to cover all paths.

There are no multiple conditions in the function to be covered.



Coverage type	TC for 100% coverage	Coverage obtained	Test cases defined
Node	3	100	T1
Edge	4	100	T1
Multiple condition	NA	-	-
Loop line 7	1	33% (1/3)	T0
Path	4	100%	T0, T1, T2, T3

Test cases:
T0: compute_tax(1000)
T1: compute_tax(6000)
T2: compute_tax(16000)
T3: compute_tax(32000)

3.2.5 String copy

For the following function define the control flow graph, and define test cases to obtain the highest possible node coverage, edge coverage, multiple condition coverage, loop coverage, path coverage. For the test cases, **write only the input value.**

```
int main() {
    int i, j;
    char str[10][50], temp[50];
    printf("Enter 10 words:");
    for ( i = 0; i < 10; ++i )
        gets(str[i]);
    for( i = 0; i < 9; ++i )
        for( j = i + 1; j < 10 ; ++j){
            if ( strcmp( str[i] , str[j] ) > 0)
            {
                strcpy(temp, str[i]);
                strcpy(str[i], str[j]);
                strcpy(str[j], temp);
            }
        }
    return 0;
}
```

Discussion

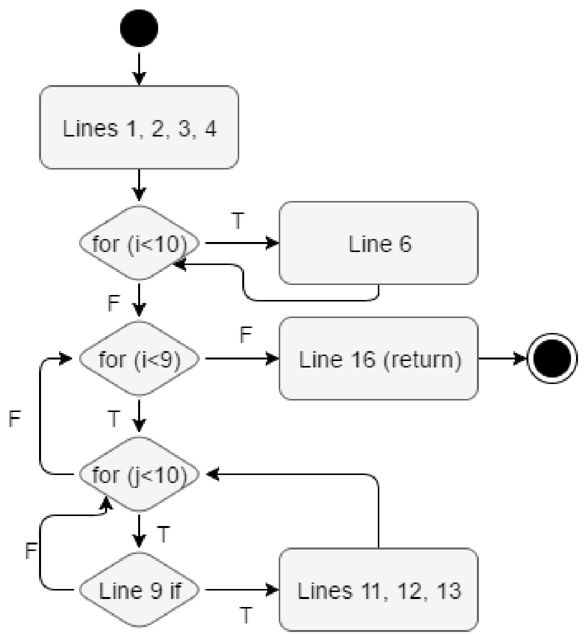
The function takes as input 10 words that are written by the user at runtime. After the for loop used for the acquisition, two nested for loops are present: the outer ones cycle on the strings from 1 to 9, the inner one on the remaining strings that compose the vector. Each time the strings are not in alfabetical order, they are swapped using the strcpy function.

A single test case is sufficient to obtain node and edge coverage: this test case should provide some words to be exchanged and some not. This wayu, all edges are covered.

There is no multiple condition to take into account.

All loops in the function are not controllable, because they depend on indices i and j that are not influenced by the strings that are given in input by the user. The first and second for loop are executed always many times (respectively, 10 and 9). At the last iteration of the outer for loop, the inner for loop is executed just one time; at any other iteration, it is executed multiple times. Hence, 33%, 33% and 66% coverage can be guaranteed, respectively, for the three for loops.

Path coverage is unfeasible with a limited amount of test cases, since the total amount of possible paths is 2^{9*9} (2 possible choices for each of the iteration of the for cycle).



Coverage type	TC for 100% coverage	Coverage obtained	Test cases defined
Node	1	100	T1
Edge	1	100	T1
Multiple condition	NA	-	-
Loop	3*3 = 9	Line 5 1/3 (many), Line 7 1/3 (many), Line 8 2/3 (1, many)	T1
Path	$2^{array.length-1}$	100% not feasible	-

Test case:
T1({BABZ, AABC, BABZ, BABZ, BABZ, BABZ, BABZ, BABZ, BABZ, BABZ})

3.2.6 Grades

For the following function define the control flow graph, and define test cases to obtain the highest possible node coverage, edge coverage, multiple condition coverage, loop coverage, path coverage. For the test cases, **write only the input value.**

```
01   double average (int grade1, int grade2, int grade3, int grade4, int
grade5, int grade6) {
02       int grades[6]; grades[0]= grade1; grades[1]= grade2;
03       grades[2]= grade3; grades[3]= grade4;
04       grades[4]= grade5; grades[5]= grade6;
05       double sum = 0.0; double min =33; double max = 0;
06       for (int i=0; i<6 || grade6 > 34; i++) {
07           sum = sum + grades[i];
08           if (grades[i] < min) min = grades[i];
09           if (grades[i] > max) max = grades[i];
10       }
11       return (sum –max - min)/4;
12   }
```

Discussion

The function takes six values as input, puts them in an array and then cycles over the array to find the minimum and maximum of the given grades.

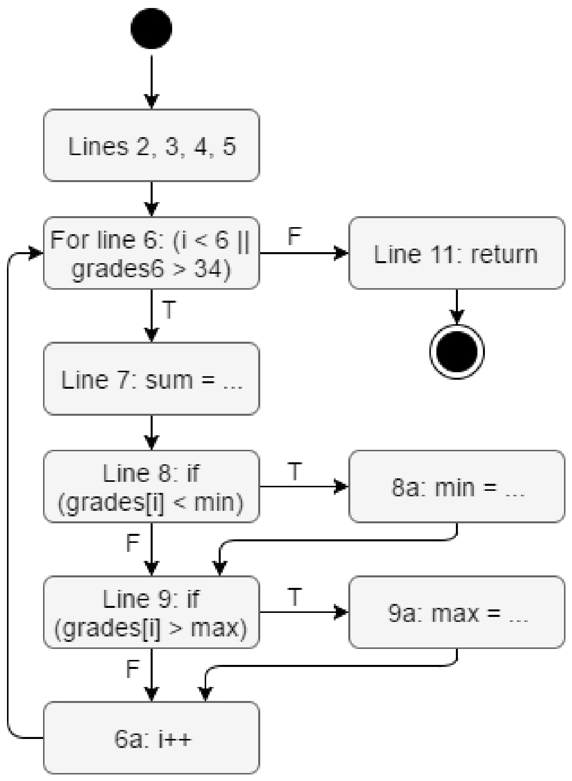
To obtain 100% coverage for both nodes and edges, a single test case T1 is sufficient. In particular, this test case should have a first grade that is set as minimum and maximum (e.g., 20), then a value that is set as new maximum (e.g., 30); then at least a value between the min and max (so to have false in bot the internal if statements).

Multiple condition in line 6 should generally be tested with four different test cases. In this case, just two test cases are sufficient to test all the combinations, in different iterations of the for loop. The test case T1 described before covers the combinations TF (iterations 1 to 6) and FF (seventh iteration). A second test case with grade6 $>$ 34 can be defined in order to cover the combinations TT (seventh iteration).

Loop at line 6 is not controllable: there is no way to force one iteration or zero iterations only, and any test case will perform at least 6 iterations. Therefore, 33% coverage can be obtained.

For what concerns path coverage, if grade6 $\leq$ 34 the loop iterates 6 times; in this case the possible paths are $\leq 4^6 = 4096$. If grade6 $>$ 34 then the loop never ends. However if the language has array bound control (e.g., Java) an exception will be raised at iteration 7 and the loop will stop. So in the best case the paths are $\leq$ 4096, not impossible but still challenging.

Test cases:
T1(20, 30, 21, 21, 21, 21)
T2(20, 30, 21, 21, 21, 35)


Lines 2, 3, 4, 5
For line 6: (i < 6 || grades6 > 34)
F
Line 11: return
T
Line 7: sum = ...
Line 8: if (grades[i] < min)
T
8a: min = ...
F
Line 9: if (grades[i] > max)
T
9a: max = ...
F
6a: i++

Coverage type	TC for 100% coverage	Coverage obtained	Test cases defined
Node	1	100%	T1
Edge	1	100%	T1
Multiple condition line 6	4 in general, 2 in this case	100	T1: TF; T2: TT; T1: FF (it. 7); T2: FT (it. 7)
Loop line 6	3	33%	T1 (6 it.)
Path	$2^{1}2$ if grade6 $\leq$ 34. Loop never ends otherwise	-	-

3.2.7 Heapsort

For the following function define the control flow graph, and define test cases to obtain the highest possible node coverage, edge coverage, multiple condition coverage, loop coverage, path coverage. For the test cases, **write only the input value.**

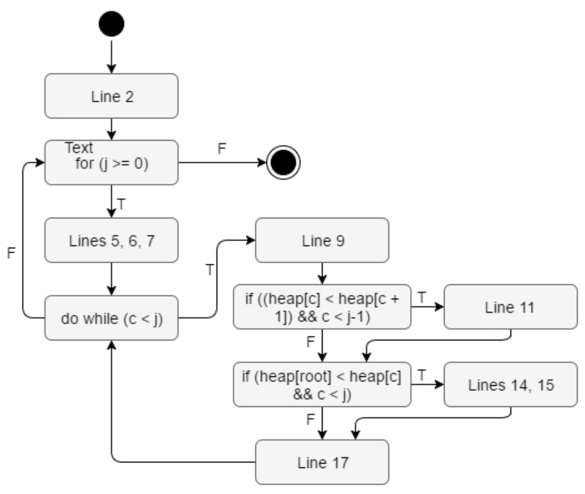
```
01    void heapsort(int heap[], int no){
02        int i, j, c, root, temp;
03        for (j = no - 1; j >= 0; j–)
04        {
05            temp = heap[0]; heap[0] = heap[j];
06            heap[j] = temp; root = 0;
07          do
08          {
09              c = 2 * root + 1;
10              if ((heap[c] < heap[c + 1]) && c < j-1)
11                  c++;
12              if (heap[root]<heap[c] && c<j)
13              {
14                  temp = heap[root]; heap[root] = heap[c];
15                  heap[c] = temp;
16              }
17              root = c;
18          } while (c < j);
19      }
20  }
```

Discussion

A single test case is sufficient to obtain both node and edge coverage. The same goes for the multiple condition in line 10: its four different cases can be covered at various iterations of the same test case T1, detailed later.

For what concerns the loop at line 3, three different test cases are needed, since the number of iterations of the loop depends on the input parameter *no*. A test case with $no = 0$ yields zero iterations, with $no = 1$ a single iteration, with $no > 1$ multiple iterations.

The number of executions of the do while loop depends on j, which is derived from *no*. Therefore, both the nested loops depends on no, and the number of possible paths grows exponentially with *no*. Hence, it is not possible to give a set of test cases able to guarantee 100% path coverage for the function.



Coverage type	TC for 100% coverage	Coverage obtained	Test cases defined
Node	1	100%	T1
Edge	1	100%	T1
Multiple condition line 10	4 or less	100	T1
Loop line 3	3	100%	T1 many; T2 zero; T3 one
Path	Not feasible	-	-

Test cases:
T1: heap[5, 2, 3, 6], 4
T2: heap[], 0
T3: heap[1], 1

3.2.8 Binary search

For the following function define the control flow graph, and define test cases to obtain the highest possible node coverage, edge coverage, multiple condition coverage, loop coverage, path coverage. For the test cases, **write only the input value.**

```
01   int binarySearchArray(int* array, int n, int search) {
02       first = 0; last = n - 1; middle = (first+last)/2;
03
04       while( first <= last )
05       {
06         if ( array[middle] < search )
07             first = middle + 1;
08         else if ( array[middle] == search )
09         {
10             return (middle+1);
11         }
12         else
13         { last = middle - 1; }
14          middle = (first + last)/2;
15       }
16      if ( first > last )
17         return -1;
18   }
```

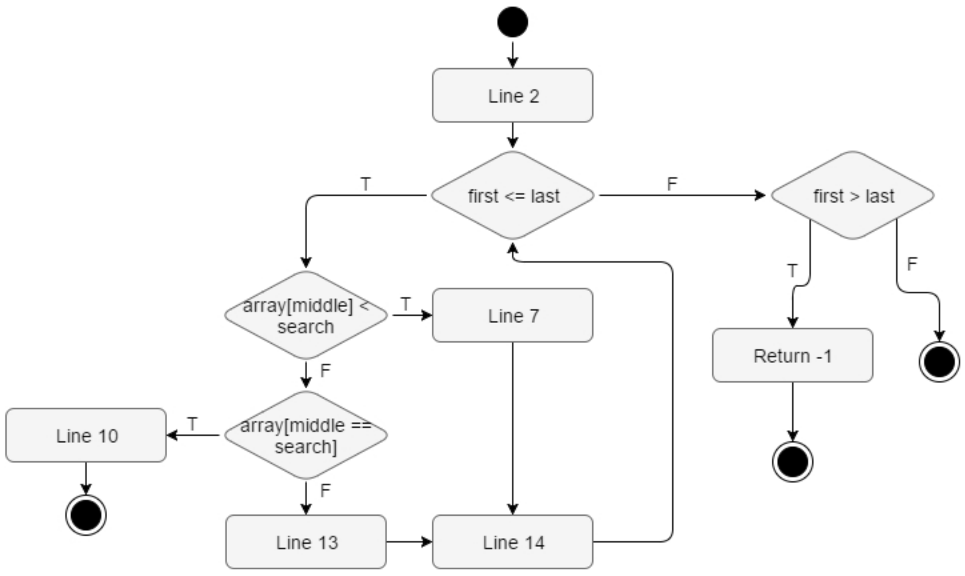
Discussion
The function performs a binary search inside an array; the inputs given to the function are the array itself, the number of elements of the array and the number to be searched inside it.

It is possible to have node and edge coverage with two test cases. Inside the while cycle, a test case must trigger, in different iterations, each of the three possible branches created by the if - else - else statement. A test case must cover the return in line 10, and another one must cover the return in line 17.

Three test cases are needed to obtain all the possible returns of the function (the one from line 10, and the two ones reachable from the if statement at line 16), and therefore guarantee 100% edge coverage.

Three test cases are needed to cover the loop in line 4. The three different coverages of the loop are feasible because the loop execution depends on input variables to the function.

The path coverage is not feasible in practice, since the vector can have a length as big as MAXINT, and multiple paths are triggerable inside each iteration of the while loop.



Coverage type	TC for 100% coverage	Coverage obtained	Test cases defined
Node	<= 3	100%	T1, T2
Edge	<= 4	100%	T1, T2
Multiple condition	NA	-	-
Loop	3	100%	T3 zero; T4 one; T5 many;
Path	100% not feasible in practice	-	-

Test cases:
T1({1,3,5,7,8,9,11}, 7, 8): found
T2({1,3,5}, 3, 99): not found

T3({1}, 0, 2)
T4({1}, 1, 2)
T5({1,3,5}, 3, 2)