Software Engineering II
Exercises

Christian Rossi

Academic Year 2023-2024

Abstract

The objective of the course is to teach the principals methods and processes of software engineering needed to develop complex and qualitative software.

The course covers the following arguments:

- Software process and its organization.
- Modelling languages.
- Requirements analysis and definition.
- Software development methods and tools.
- Approaches for verify and validate the software.

Contents

CHAPTER 1

Exercises session I

1.1 AdmissionManager

Your company has been tasked to develop a system that handles the admission applications that parents send, on behalf of their children, to the high schools of a metropolitan area. Parents can send admission applications to multiple schools. Before sending an application, they must register their child in the system; the registration includes login credentials (username and password), the personal data of the child (first name, last name, birthdate, etc.), the name of at least one parent, contact information (which must include an email address and a phone number), the name of the last school they attended, and the list of grades (which includes the obtained score, from 1 to 10, for each subject). Each application is assigned an identifier by the system, to allow parents to check its status after sending it (which can be "accepted", "rejected", or "not evaluated"). Parents can withdraw applications previously sent. They can also ask the system to be notified by email when the outcome of the evaluation of an application is available. School administrators use the system to check the applications sent to their schools and to approve/reject them. In particular, they can retrieve the list of applications sent to their schools that have yet to be evaluated; they can also leave comments on the applications, and they can decide to accept or reject the applications. Administrators can also set a preference to receive a notification, in the form of an email, when a new application is sent to their school.

1. Define the goals for the AdmissionManager system.
2. Select one of the goals defined in the previous point and define in natural language suitable domain assumptions and requirements to guarantee that the AdmissionManager system fulfills the selected goal.
3. Draw a UML Use Case Diagram describing the main use cases of the AdmissionManager system.
4. Pick one of the use cases, and define it.

Solution

1. The goals are world phenomena shared between the machine and the real world. They are problem of the real world that the AdmissionManager needs to address. We have four examples, which are:

- User sends an application.
- User withdraws an application.
- School administrator evaluates an application.
- User is notified about an application evaluation.

The problem with those goals is that they are only on world side, so they are not well formulated. The term user is ambiguous, it needs to be specified (parents and school administrator). The formulation can be changed to make them correct:

- Parents can manage (send and then monitor) applications to schools on behalf of their children.
- School administrators can manage (check and approve/reject) applications sent to their schools.

2. A domain assumption like "as soon as an application arrives to the system, a status needs to be assigned to it" is not correct because the status depend on a method in the program and not on something that is granted by the real world. Examples of correct domain assumption, that are not well formulated are:

   - Parents must be registered into the system to issue an application.
   - The system must allow parents to register by providing their email address and personal information.

   In the end, for the first goal we have the following domain assumptions:

   - AdmissionManager allows system administrators to open application windows for their schools.
   - AdmissionManager allows parents to register into the system and provide contact information and information about their child.
   - AdmissionManager allows parents to log into the system using the credentials input at registration time.
   - AdmissionManager allows parents to indicate in their profile that they want to be notified when the outcome of an application is available.
   - AdmissionManager allows parents to send an application to a school.
   - AdmissionManager assigns a unique identifier to each application received.
   - AdmissionManager allows parents to see the list of applications sent.
   - AdmissionManager allows parents to withdraw an application previously sent.

   The assumption is the following:

   - Parents provide correct information (in particular, contact information) when registering.

   And for the second goal we have the following domain assumptions:

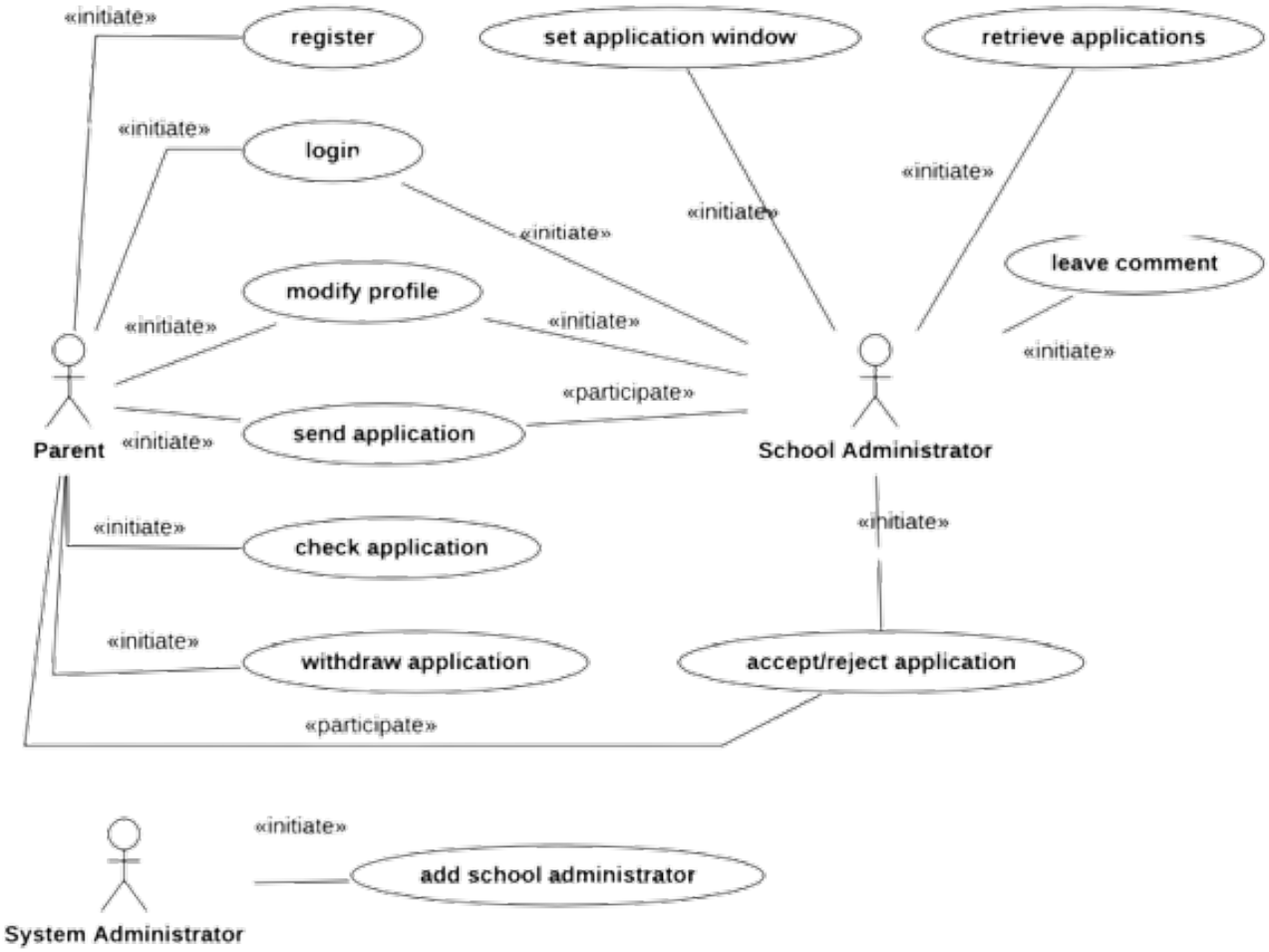
   - AdmissionManager allows system administrators to insert new administrators in the system and associate them with the corresponding school.

- AdmissionManager allows school administrators to log into the system using the credentials assigned to them by system administrators.
- AdmissionManager allows school administrators to indicate in their profile that they want to be notified when new applications for their schools are received.
- AdmissionManager allows school administrators to retrieve applications (related to their schools) that have yet to be evaluated.
- AdmissionManager allows school administrators to select an application yet to be evaluated and leave a comment in it.
- AdmissionManager allows school administrators to accept/reject an application.

The assumption is the following:

- School administrators periodically evaluate applications and guarantee to explicitly accept/reject all applications arrived within the notification window.

3. The UML use case diagram is the following:



4. We select send application case. We have that:

Name	Send Application
Actor	Parent, School Administrator
Entry condition	Parent has registered child and is logged in with the corresponding account. He/she has all necessary information
Event Flow	1. Parent selects school to which application must be sent 2. If required by the school, parent fills out additional information concerning child 3. Parent clicks “submit” button 4. System checks application and responds with application number 5. If administrator of selected school has asked to receive a notification of the application, email is sent to school administrator
Exit Condition	Application is received by the system, and email is sent to school administrator if he/she asked to be notified
Exceptions	Data provided in application is invalid or missing, user is notified that it should be fixed. School does no longer accept application (the application window has expired), so the application is immediately rejected

1.2 PaasPopCoin

The private security and event organization company HSG from the Netherlands wants to build an application (PaasPopCoin) that handles the coin emission and transactions in the scope of a medium-size music festival they are organizing. The goal of the system is to allow festival-goers and operators to spend an allotted amount of money in relative safety and without the need to bring wallets and other assets around the event. The software in question needs to handle at least three scenarios:

- Emission of coins in exchange for money through appropriate cashier desks and ATMs.
- Cash-back, that is, exchange of coins with cash in the same locations (we assume that people at the festival may be willing to receive back the money corresponding to the coins they have not used).
- Tracking of coin expenditure transactions at the various festival shops.

In the scope of the above scenarios, there are several special conditions to be considered. First, in the scope of coin emissions, there exist four classes of coin buyers:

a. VIPs who receive a 30% discount on the coins they buy.

b. Event organization people who receive a 50% discount.

c. Event ticket holders class A, who receive a 20% discount.

d. Regular ticket holders who receive no discount.

When buying coins, users first need to authenticate themselves by inserting their own ID card in the ATM or by giving it to the cashier; this allows the system to determine the class to which each coin buyer belongs. After authentication, buyers get the coins upon inserting into the ATM or giving to the cashier the corresponding amount of money. Second, also in the context of cash-back, users need to authenticate with their ID card to make sure the appropriate amount of money is given back, considering their role and privileges. Third, during the event, every shop clerk keeps track through the PaasPopCoin system of the sales of products and the coins received. PaasPopCoin relies on a third-party analytics service to periodically check whether the festival is earning money or not (cost-benefit analysis). Such check is performed with respect to costs of products being sold during the event, as well as the overhead to cover all event organization and management expenses.

1. With reference to the Jackson-Zave distinction between the world and the machine, identify the relevant world phenomena for PaasPopCoin, including the ones shared with the machine, providing a short description if necessary. For shared phenomena specify whether they are controlled by the world or the machine. Focus on phenomena that are relevant to describe the requirements of the system.

2. Describe through a UML Class Diagram the main elements of the PaasPopCoin domain.

3. Define a UML Use Case Diagram describing the relevant actors and use cases for PaasPopCoin. You can provide a brief explanation of the Use Case Diagram, especially if the names of the use cases are not self-explanatory.

Solution

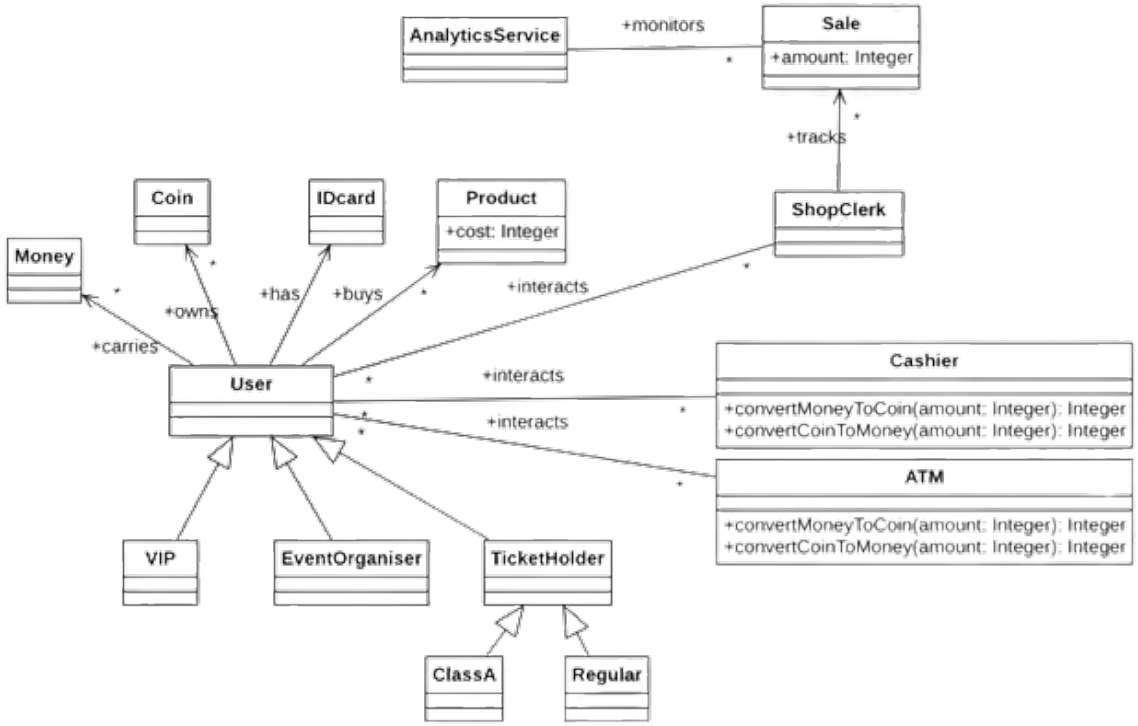
1. The world-only phenomena can be:

   - User buys Class A ticket.
   - User buys regular ticket.
   - VIP is contracted for event.
   - Event organization is started and contractors registered.
   - Event starts.
   - User gives money to cashier (to be converted in coins).
   - User gives coins to cashier (to be converted in money).
   - User gives ID card to cashier.
   - User buys some product at festival.
   - The external analytics service checks the success of an event.

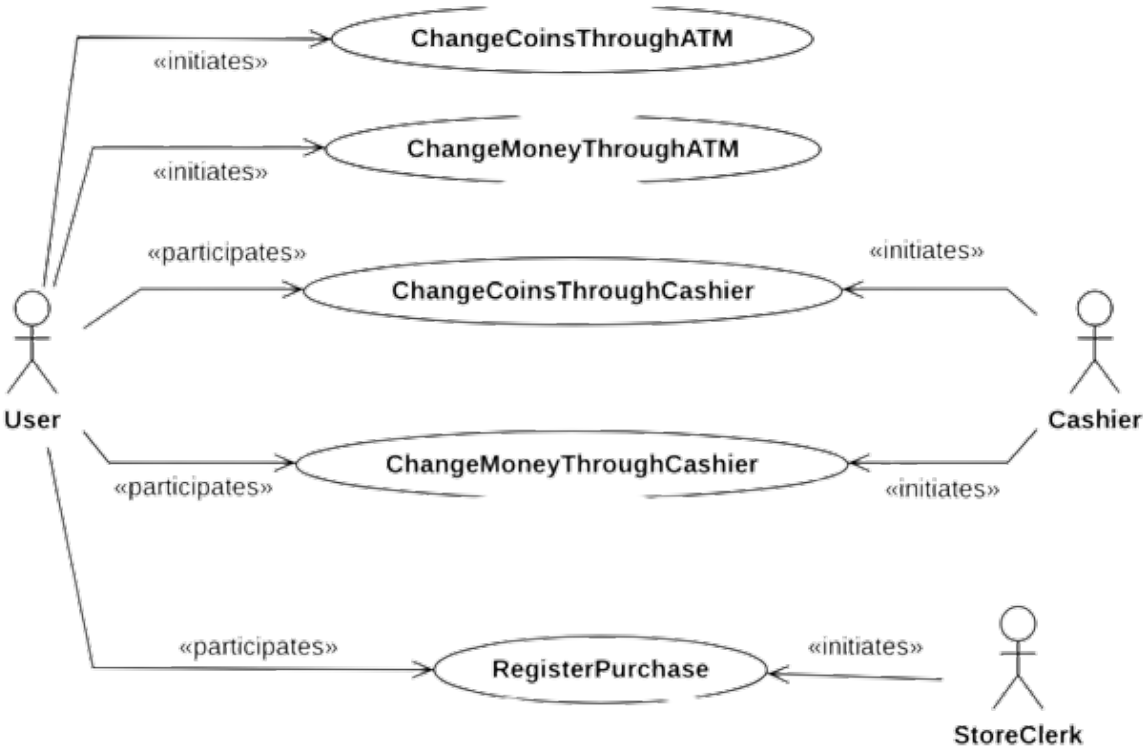
   The shared phenomena can be the following:

   - User inserts money into an ATM machine.
   - ID Card is inserted into ATM.
   - User inserts coins into an ATM.
   - Cashier inserts in the system an ID card number.
   - Cashier inserts in the system the amount of money handed by a certain user.
   - Cashier inserts in the system the amount of coins returned by a certain user.
   - Store clerk inputs in system the amount of coins spent by user in shop.
   - The system enables coin emission after checking ID card and inserted amount of money.
   - The system enables cash-back after checking ID card and inserted number of coins.
   - The system sends data about purchases to the external analytics service.

2. The UML diagram of the given problem is:



3. The UML use case diagram of the given problem is:



CHAPTER 2

Exercise session II

2.1 Alloy

Consider construction cubes of three different sizes, small, medium, and large. You can build towers by piling up these cubes one on top of the other respecting the following rules:

- A large cube can be piled only on top of another large cube.
- A medium cube can be piled on top of a large or a medium cube.
- A small cube can be piled on top of any other cube.
- It is not possible to have two cubes, A and B, simultaneously positioned right on top of the same other cube C.

1. Model in Alloy the concept of cube and the piling constraints defined above.
2. Model also the predicate canPileUp that, given two cubes, is true if the first can be piled on top of the second and false otherwise.
3. Consider now the possibility of finishing towers with a top component having a shape that prevents further piling, for instance, a pyramidal or semispherical shape. This top component can only be the last one of a tower, in other words, it cannot have any other component piled on it. Rework your model to include also this component. You do not need to consider a specific shape for it, but only its property of not allowing further piling on its top. Modify also the canPileUp predicate so that it can work both with cubes and top components.

Solution

1. The concept of cube and its constraints can be defined in the following way.

```
abstract sig Size{}

sig Large extends Size{}

sig Medium extends Size{}

sig Small extends Size{}
```

```
sig Cube {
    size: Size,
    piledOn: lone Cube
}{piledOn ≠ this}

fact noCircularPiling {
    no c: Cube | c in c.^piledOn
}

fact pilingUpRules {
    all c1, c2: Cube | c1.piledOn = c2 implies (
    c2.size = Large or
    c2.size = Medium and (c1.size = Medium or c1.size = Small) or c2.size = Small and c1.
        ↪ size = Small)
}

fact noMultipleCubesOnTheSameCube {
    no disj c1, c2: Cube | c1.piledOn = c2.piledOn
}
```

2. The predicate canPileUp can be defined as follows.

```
pred canPileUp[cUp: Cube, cDown: Cube] {
    cUp.piledOn = cDown and
    (cDown.size = Large or
    cDown.size = Medium and (cUp.size = Medium or cUp.size = Small) or 2
    cDown.size = Small and cUp.size = Small)
}

pred show {}

run show

run canPileUp
```

3. The now model become:

```
abstract sig Size{}

sig Large extends Size{}

sig Medium extends Size{}

sig Small extends Size{}

abstract sig Block{
    piledOn: lone Cube
}

sig Cube extends Block {
    size: Size
}{piledOn ≠ this}

sig Top extends Block { }

fact noCircularPiling {
    no c: Cube | c in c.^piledOn
}

fact noMultipleBlocksOnTheSameCube {
    no disj b1, b2: Block | b1.piledOn = b2.piledOn
}

fact pilingUpRules {
    all c1, c2: Cube | c1.piledOn = c2 implies (
    c2.size = Large or
    c2.size = Medium and (c1.size = Medium or c1.size = Small) or c2.size = Small and c1.
        ↪ size = Small)
}

pred canPileUp[bUp: Block, cDown: Cube] {
```

```
    bUp.piledOn = cDown and (bUp in Top or
    (cDown.size = Large or
    cDown.size = Medium and (bUp.size = Medium or bUp.size = Small) or cDown.size = Small
        ↪  and bUp.size = Small))
}

pred show {}

run show

run canPileUp
```

2.2 Alloy

The company TravelSpaces decides to help tourists visiting a city in finding places that can keep their luggage for some time. The company establishes agreements with small shops in various areas of the city and acts as a mediator between these shops and the tourists that need to leave their luggage in a safe place. To this end, the company wants to build a system, called LuggageKeeper, that offers tourists the possibility to: look for luggage keepers in a certain area; reserve a place for the luggage in the selected place; pay for the service when they are at the luggage keeper; and, optionally, rate the luggage keeper at the end of the service.

Given the scenario above, consider the following world phenomena:

- Every user owns various pieces of luggage.
- Every user can carry around various pieces of luggage.
- Each piece of luggage can be either safe, or unsafe.
- Small shops store the luggage in lockers, where each locker can store at most one piece of luggage.

Consider also the following shared phenomena:

- Each locker is opened with an electronic key that is associated with it (the electronic key is regenerated at each use of the locker; also, a locker that does not have an electronic key associated with it is free).
- Each user can hold various electronic keys.

Formalize through an Alloy model:

1. The world and machine phenomena identified above.
2. A predicate capturing the domain assumption D1 that a piece of luggage is safe if, and only if, it is with its owner, or it is stored in a locker that has an associated key, and the owner of the piece of luggage holds the key of the locker.
3. A predicate capturing the requirement R1 that a key opens only one locker.
4. A predicate capturing the goal G1 that for each user all his/her luggage is safe.
5. A predicate capturing the operation GenKey that, given a locker that is free, associates with it a new electronic key.

Solution

The model requested is:

```
abstract sig Status{}

one sig Safe extends Status{}

one sig Unsafe extends Status{}

sig Luggage{
    luggageStatus : one Status
}

sig EKey{}

sig User{
    owns : set Luggage, carries : set Luggage, hasKeys : set EKey
}

sig Locker{
    hasKey : lone EKey, storesLuggage : lone Luggage
}

sig Shop{
    lockers : some Locker
}

pred D1 {
    all lg : Luggage | lg.luggageStatus in Safe
    iff
    all u : User | lg in u.owns implies ( lg in u.carries or some lk : Locker | lg in lk.
        ↪ storesLuggage and lk.hasKey ≠ none and lk.hasKey in u.hasKeys )
}

pred R1 {
    all ek : EKey | no disj lk1, lk2: Locker | ek in lk1.hasKey and ek in lk2.hasKey
}

pred G1 {
    all u : User | all lg : Luggage | lg in u.owns implies lg.luggageStatus in Safe
}

pred GenKey[lk, lk' : Locker] {
    lk.hasKey = none
    lk'.storesLuggage = lk.storesLuggage
    one ek : EKey | lk'.hasKey = ek
}
```