# 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

# Alloy examples

## 1.1 Address book

Imagine that we are asked to model a very simple address book. The books that contain a bunch of addresses linked to the corresponding names.

In this example we have three entities, which are: *Name*, *Addr* and *Book*. *addr* is linking *Name* to *Addr* within the context of *Book*. To indicate this relation we use the keyword *lone* that indicates that each *Name* can correspond at most one *Addr*. We can resume the previous statements as:

```
sig Name {}
sig Addr {}
sig Book {
    addr: Name -> lone Addr
}
```

This specification creates the following relations:

- Sets are unary relations.
- Scalars are singleton sets.
- The ternary relation involving the three predicates.

We can declare a new predicate with the keyword *pred*.

```
pred show {}
run show for 3 but exactly 1 Book
```

Where the second line indicates that we need to find at most three elements for every *Book*. The predicate *show* defined previously is empty and return always *true*. Now we can define a predicate with some argument, for example:

```
pred show [b:Book]{
    # b.addr > 1
}
run show for 3 but exactly 1 Book
```

The predicate (consistent) in the previous example adds a constraint on the number of *Address* relations in a given *Book*. The predicate (consistent) in the following example adds a constraint on the number of different *Address* that appears in the *Book*.

```
pred show [b:Book]{
    # b.addr > 1
    # Name.(b.addr) > 1
}
run show for 3 but exactly 1 Book
```

The predicate (inconsistent) in the following example contains the keyword *some* that indicates the existence of an element. In this case we have only one *Book* so the tool will say that no instances can be found.

```
pred add [b:Book]{
    # b.addr > 1
    some n:Name | # n.(b.addr) > 1
}
run show for 3 but exactly 1 Book
```

All the previous predicates are static because they doesn't change the signature. In Alloy there are also dynamic predicates for dynamic analysis. For example we can define a predicate that adds an *Address* and *Name* to a *Book* in the following way:

```
pred add [b,b':Book, n:Name,a:Addr]{
     b'.addr=b.addr + n -> a
}
pred showAdd [b,b':Book, n:Name,a:Addr]{
     add[b,b',n,a]
     #Name.(b'.addr) > 1
}
run showAdd
```

We can now define a predicate for the *Book* deletions.

```
pred del [b,b':Book, n:Name]{
     b'.addr=b.addr - n -> Addr
}
```

We can check if running a delete after an add returns us in the initial situation or not by using an *assertion*:

```
assert delRevertsAdd{
    all b1,b2,b3:Book,n:Name,a:Addr
    add[b1,b2,n,a] and del[b2,b3,n]
    implies b1.addr=b3.addr
}
```

While checking an assertion, Alloy searches for counterexamples. In this case we will find a counterexample so the assert will result $false$. To correct the assert we need to modify it in the following way:

```
assert delUndoesAdd{
    all b1,b2,b3:Book,n:Name,a:Addr |
    no n.(b1.addr) and add[b1,b2,n,a] and del[b1,b2,n]
    implies b1.addr=b3.addr
}
```

We can also need to get some signature. To do that we can use the Alloy functions. For example we can declare a function that search a certain *Book* and return a set of *Address*:

```
fun lookup[b:Book,n:Name]: set Addr{
    n.(n.addr)
}
```

## 1.2 Family relations

We now consider a family relationship tree. First of all we have to define a generic person, that can be a men or a woman.

```
abstract sig Person {
    father: lone Man
    mother: lone Woman
}
sig Man extends Person {
    wife: lone Woman
}
sig Woman extends Person {
    husband: lone Man
}
```

We have set that each $Person$ has at most one father and one mother (keyword *lone*) because we need a root for the tree. The person at the root needs to have no parents (for example they are unknown). The signature $Person$ is *abstract* because it needs to be specialized in one of the subsequent signatures, that are $Man$ or $Woman$. Signatures by using keyword *sig* represents

a set of atoms. Before this keyword we can define the number of entities that we need (*lone*, *one* or *some*).

**Definition**

> The *fields* of a signature are relations whose domain is a subset of the signature. The keyword *extends* is used to declare a subset of signature.

To get the set of grandpas of a given person we can define a function like this:

```
fun grandpas[p:Person]:set Person {
    p.(mother+father).father
}
pred ownGrandpa[p:Person] {
    p in p.grandpas[p]
}
```

We have also defined a predicate that checks if the person is in the set of grandpas returned by the function *grandpas*. The problem now is that we have not set constraints on relations. To do that we need to define two new operators for binary relations:

- Transitive closure: $\hat{}r = r + r.r + r.r.r + \ldots$
- Reflex transitive closure: $*r = iden + \hat{}r$

We can now define that no one can be the father/mother of himself:

```
fact {
    no p:Person | p in p.^(mother+father)
}
```

We have also to set a constraint that if X is husband of Y, then Y is the wife of X:

```
fact {
    all m:Man,w:Woman | m.wife=w iff w.husband=m
}
fact {
    wife = ~husband
}
```

The two facts are equivalent, but the second has been written using the transpose operator. The fact can contains multiple constraints. So the previous constraints can be written in one fact. The difference between $fact$ and $pred$ is that the first are global, while the second needs to be invoked.

# Chapter 2

# Exercises session I

### Exercise 1

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.

**Answer of exercise 1**

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.

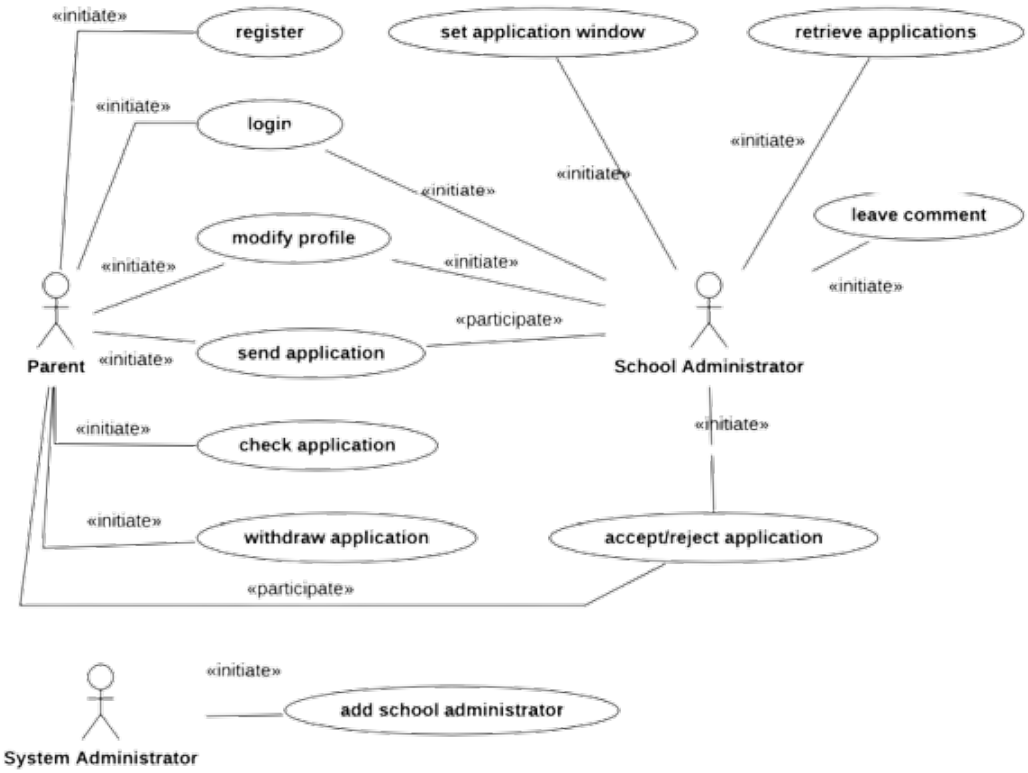
And for the second goal we have the following domain assumptions:

- 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<br>2. If required by the school, parent fills out additional information concerning child<br>3. Parent clicks "submit" button<br>4. System checks application and responds with application number<br>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.<br>School does no longer accept application (the application window has expired), so the application is immediately rejected |

## Exercise 2

The private security and event organisation 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.

**Answer of exercise 2**

1. The world-only phenomena can be:

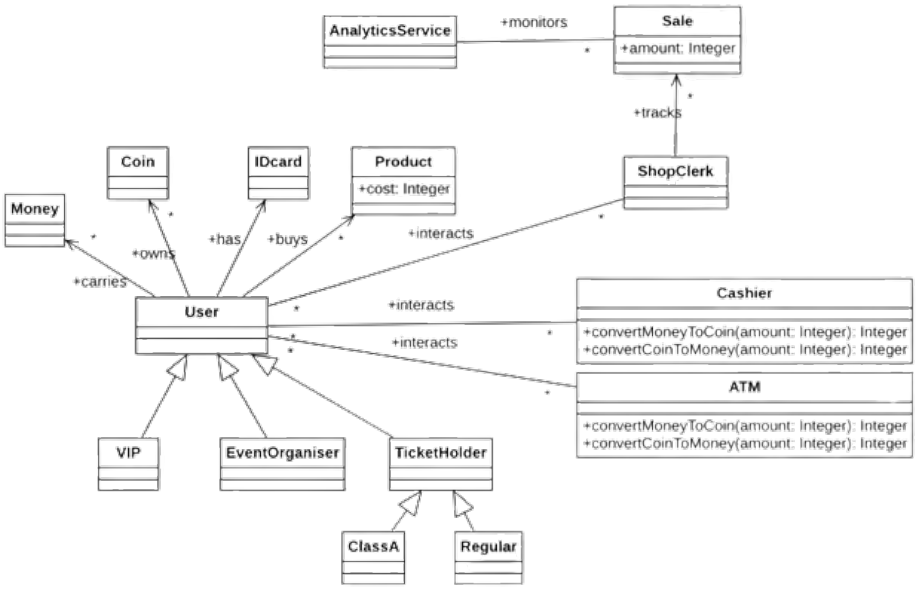
   - User buys Class A ticket.
   - User buys regular ticket.
   - VIP is contracted for event.
   - Event organization is started and contractors registered.
   - Event starts.
   - User gives money to cashier (to be converted in coins).
   - User gives coins to cashier (to be converted in money).
   - 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:

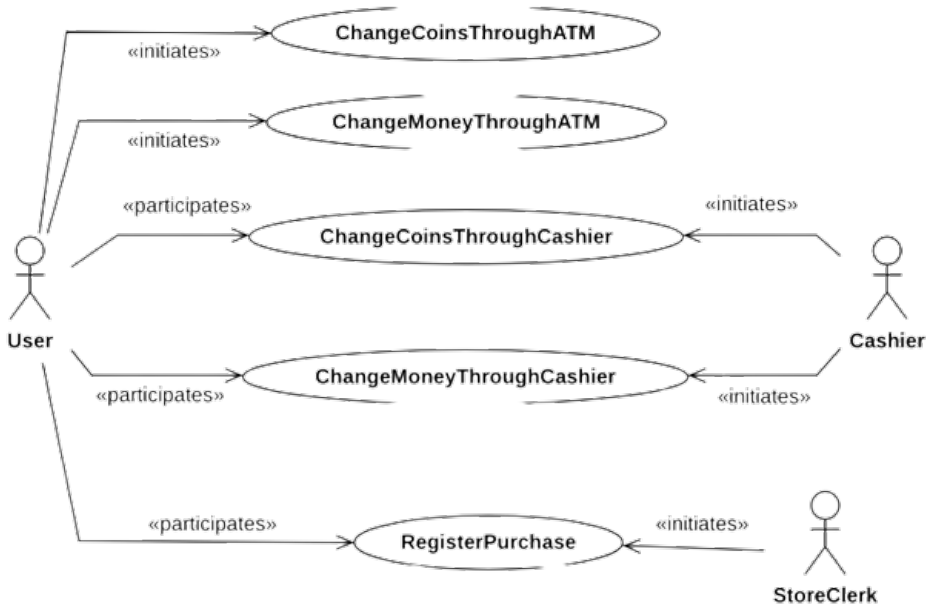
   - User inserts money into an ATM machine.
   - ID Card is inserted into ATM.
   - User inserts coins into an ATM.
   - Cashier inserts in the system an ID card number.
   - Cashier inserts in the system the amount of money handed by a certain user.

- 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 diagram of the given problem is:


ChangeCoinsThroughATM
«initiates»
ChangeMoneyThroughATM
«initiates»
«participates»
ChangeCoinsThroughCashier
«initiates»
User
Cashier
ChangeMoneyThroughCashier
«participates»
«initiates»
«participates»
RegisterPurchase
«initiates»
StoreClerk