YEDITEPE UNIVERSITY
SOFTWARE ARCHITECTURE
TERM PROJECT 2021 FALL

DIGITAL PHARMACY PROJECT FOR AUTOMATING PHARMACIES

ATALAY ÖZMEN
SALİH EREN BİRİNCİOĞLU
METE ZENGİN

YEDİTEPE ÜNİVERSİTESİ

CONTENTS

1. Introduction

1.1. Background

In this report, we will explain the background, requirements and the architecture of our project. Our project is an automated pharmacy system with additional medical testing functionalities.

The biggest problem with the traditional pharmacies nowadays is that the pharmacies are open only until evening at a specific time, this time is 17:30 in Turkey for example. After this time, if a patient has a serious emergency and he/she is in need of a medicine, the patient has to search for night-time pharmacists which are rare and probably far away from the patient. Patients usually require some kind of motor transport in order to obtain medicine from the night-time pharmacists and for most people this is impossible to do, thus making the patient unable to obtain the medicine he/she needs and creating situations that can be serious and even fatal for the patient.

For illnesses that are common for everyone and don’t require a doctor’s visit, such as the flu, sore throat, headaches, finger cuts, mild infections etc. the patients may not know the optimal medicine they should be getting from the pharmacy and many times they might just keep using the same medicine that they know to be working. However, their option of medicine might not be totally accurate and need adjusting. Some of the time, the patients might not have any clue whatsoever to what medicine they should be using and they might need total guidance for choosing the right one. A way of accurately deciding on the correct medicine is needed in these situations.

1.2. Problem Statement

Based on the background information that was discussed, one of the biggest problems is the inaccessibility of the pharmaceutical goods during the night times and holidays when the pharmacies are closed. The pharmacies need to be open throughout the day without interruption in order to prevent any problems that might arise from the lack of needed medicine. In addition, the problem of finding the optimal medicine for illnesses that don't necessarily need a doctor's visit also has to be addressed and solved. Finally, we know that people have been going to pharmacies in order to get their blood pressure and blood sugar levels tested. While making the pharmacies automated, this process of checking blood pressure and blood sugar levels also need to be automated.

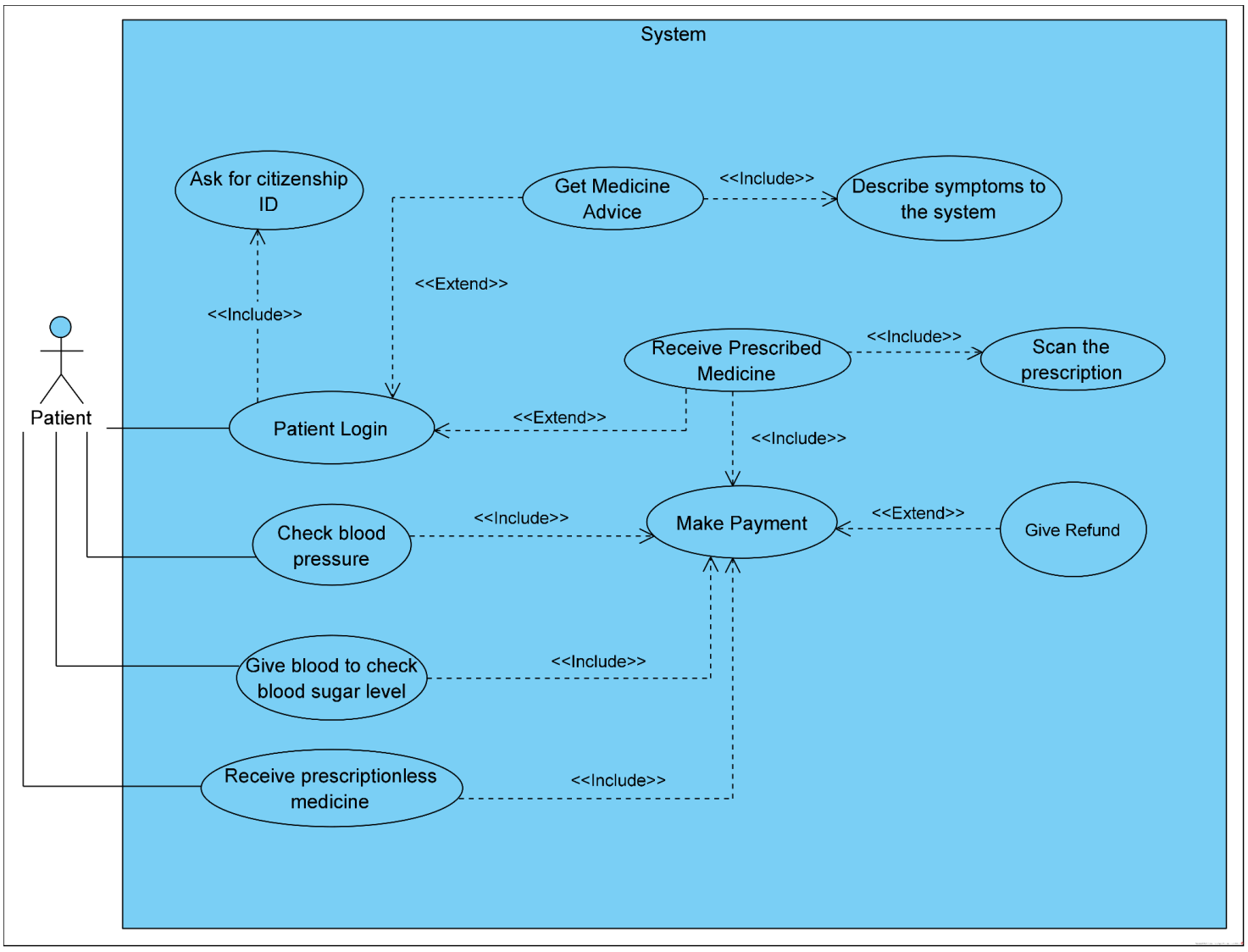
Figure 1.1: A disappointed patient looking at the time in front of a closed pharmacy

2. Architecture Steps

2.1. Requirements Specification

2.1.1. Functional Requirements

Use Case Diagram:



2.1.2. Non-Functional Requirements

Volere Templates:

Requirement ID: 3001 **Requirement Type: Non-Functional**

Requirement: Reliability
Description: The pharmacist kiosk shall be online and active for 7/24. The patient should be able to get what he/she wants even if there is a power cut.
Rationale: This requirement is needed because this is a vital service for sick people and they should be able to reach medicine 7/24.
Fit Criteria: An external device like a generator can be set to eliminate this kind of risk.

Requirement ID: 3002 **Requirement Type: Non-Functional**

Requirement: Hygiene
Description: Pharmacist kiosk and its functionalities shall be hygienic to use.
Rationale: This requirement is needed because services we provide need high level hygiene for human health.
Fit Criteria: All the equipment that interacts with humans should disinfect itself or should be replaceable after use.

Requirement ID: 3003 **Requirement Type: Non-Functional**

Requirement: Efficiency/Response Time
Description: Pharmacist kiosk shall respond fast to patients. Patient shall be able to finish the job? between 2 and 8 minutes.
Rationale: This requirement is needed because the kiosk has to be time efficient. Time is important when it comes to health.
Fit Criteria: We can use a cache system for providing most used medicines also our machine can deliver critical medicines faster.

Requirement ID: 3004 **Requirement Type: Non-Functional**

Requirement: Usability
Description: Pharmacist kiosk's interface shall be user friendly. Users shall understand controls easily.
Rationale: Control system is essential for this type of service. Any patient should be able to use it.
Fit Criteria: The interface will be tested by people to get feedback from them.

Requirement ID: 3005 **Requirement Type: Non-Functional**

Requirement: Security
Description: Pharmacist kiosk shall be secure. Patients can only get the medicines they have the right to buy.
Rationale: Patients shouldn't buy controlled substances without prescription.
Fit Criteria: The process of buying prescription medicines can be checked with multiple steps and secured authorizations.

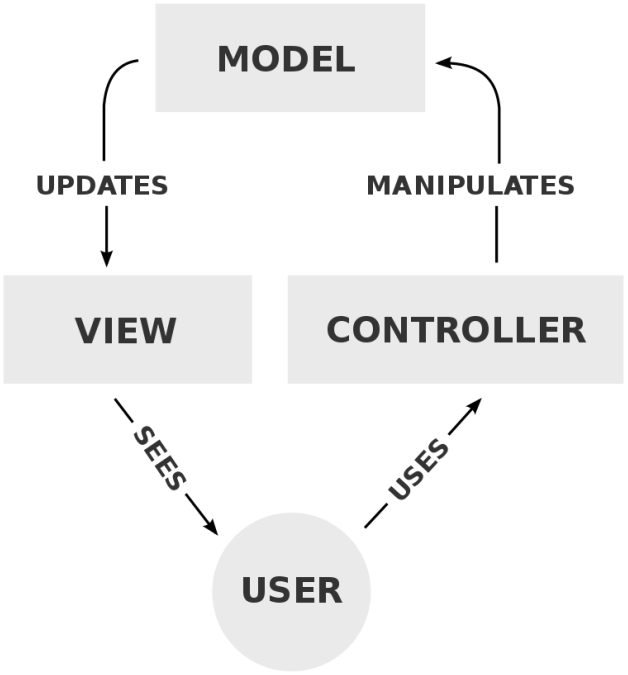
Requirement ID: 3006 **Requirement Type: Non-Functional**
Requirement: Maintainability
Description: The software shall be easy to change, improve and understand to other developers.
Rationale: This requirement is needed because when a new update or improvement is necessary the software code can be modified easily without wasting time.
Fit Criteria: The maintainability can be measured by the common amount in the code and looking at the complexity of the code.

Requirement ID: 3007 **Requirement Type: Non-Functional**
Requirement:Portability
Description: This is an alternative for pharmacies so it shall be fast to deploy.
Rationale: This requirement is needed because medicines are essential for humans and we need these pharmacy kiosks in every neighborhood.
Fit Criteria: It can be tested how pharmacy kiosks work in different environments and how fast we can set up.

Requirement ID: 3008 **Requirement Type: Non-Functional**
Requirement:Affordability
Description:We will need a lot of pharmacy kiosks so it shall be cheap.
Rationale: This requirement is needed because medicines are essential for humans and we need these pharmacy kiosks in every neighborhood.
Fit Criteria: The components will be made reusable and maintainable in order to reduce engineering overhead and reduce costs.

2.2. Architecture Specifications

The model-view-controller architecture design will be used in making of this system. The reason for this decision is that patients need a user interface to interact with the system. The “view” part of the model-view-controller is the user interface part. The services that our system uses is the “controller” part. It is where the logic of our system lies. This controller will communicate with the database and system variables. Therefore, the database and system variables are the “model” part. When the model part changes, it updates our view in order to show the user the results of the services that they have used.



Our system model that we designed for the digital pharmacy project has processes that consist of three steps. First of all, the patient interacts with the kiosk and pays for the service. Patients can access multiple services like buying medicine, getting medicine advice,blood sugar testing and blood pressure testing. If the patient wants to buy prescribed medicine, the patient has to login to the system. Otherwise the patient does not need to login to use the other services. If a patient logs in to the system, the system saves its information to the pharmacy database.

Cashier component checks if the payment is sufficient or not. If the payment is not sufficient the money is given back to the patient. If the payment is sufficient, the cashier sends signals to the components that the patient requested service from. The cashier component will have multiple user interfaces in order to provide service to more than one patient at the same time. This component acts like a middleman between the patient and the service components.

All the service components that are described below are designed in such a way that it can provide more than one user interface.

Medicine selling component will handle providing medicine service to the patient. It takes a signal from the cashier in order to work.

Medicine adviser component works by taking symptom input from the patient and deciding on the most probable illness that causes these symptoms. After that it gives the optimal medicine advice for that illness.

After the cashier approves payment, the blood sugar test service applies the test to the patient. Later it returns the result to the patient. Blood pressure test works similarly to the blood sugar test. After payment is approved it applies the test and returns the result to the patient.

Finally, we need a component that acts as an intermediary between our system and the government system and also stores log information. It will be a database communicator that authorizes the patient logins and verifies the prescriptions that the patient scans to the system. So it will use both the local database and the government system to provide security.

```
Enum Symptoms = [SoreThroat, RunnyNose, ..., BurntSkin]
Enum Payment : = {None,NotSufficient,Sufficient};

component Patient{
     MedicineObject medicineAdvice;
     Symptoms[] patientSymptomInputs;
     bool requestForBloodSugarTest := false;
     bool requestForBloodPressureTest:=false;
    bool requestForMedicine := false;
    bool requestLogin := false;
    bool requestForPrescriptionScan := false;
    bool prescriptionVerified = true;
    Payment payment := None;
    String loginTc := null;
    String loginDevlet := null;
    MedicineObject boughtMedicine;
    bool loginSuccesful;
    Prescription prescription;
    int refundTaken = 0;

    required port pdr{
        @functional{
              requires: true;
              ensures: medicineAdvice := \result;  }
        MedicineObject getMedicineAdvice();
    }
    provided port pdp{
        @functional{
            requires: !patientSymptomInputs.isEmpty()
            ensures: result := patientSymptomInputs;  }
        SymptomObject[] getSymptoms();
    }
required port loginPort{
    @functional{
         promises: inputLoginTc := loginTc;  inputLoginDevlet :=
loginDevlet;
         requires: true;
         ensures: loginSuccesful = \result;  }
    bool patientLogIn(String inputLoginTc, String
inputLoginDevlet);
}

 emitter port paymentForMedicine{
    @functional{
        promises:payment \in [NotSufficient, Sufficient];
        ensures: requestForMedicine:=true;}
    paymentForMedicine(int payment);
}
  emitter port paymentforBloodSugarTest{
    @functional{
        promises:payment\in [NotSufficient, Sufficient];
        ensures: requestForBloodSugarTest:=true; }
    paymentforBloodSugarTest(int payment);
}
```

```
 emitter port paymentforBloodPressureTest{
    @functional{
        promises:payment\in [NotSufficient, Sufficient];
        ensures: requestForBloodPressureTest:=true; }
    paymentforBloodPressureTest(int payment);
}

required port BloodSugarTestResults{
    @functional{
        requires:true;
         ensures: requestForBloodSugarTest:=false;
                  sugar_level := \result;}
    String BloodSugarTestResult();
}

required port BloodPressureResults{
    @functional{
        requires:true;
        ensures: requestForBloodPressureTest:=false;
                 pressure_level := \result}
     String BloodPressureTestResult()
}

required port getMedicine{
    @functional{
      requires: true;
      ensures: requestForMedicine := false;
               boughtMedicine:=\result;
    }
    MedicineObject giveMedicine();
  }

required port scanPrescription{
     @functional{
         requires: loginSuccesful == true;
         ensures:
             promises: inputPrescription := prescription;
             requires: prescriptionVerified := \result; }
     bool verifyPrescription(Prescription inputPrescription);
}

consumer port getRefund{
      @functional{
             requires: true;
             ensures: refundTaken := refundTaken + refundAmount;
      }
      giveRefund(int refundAmount);
}
```

```
component DataBaseCommunicator{
    Model pharmacyDatabase;
    Model governmentDatabase;
    boolean loginSuccesful = false;

    provided port scanPrescription{
        @functional{
            requires: governmentDatabase.verify(inputPrescription) == true;
            ensures: \result := true;
          otherwise:
            requires: governmentDatabase.verify(inputPrescription) == false;
            ensures: \result := false;
        }
        boolean verifyPrescription(Prescription inputPrescription);
    }

    provided port loginPort{
        @functional{
            requires: governmentDatabase.verifyLogin(loginTc, loginDevlet) == true;
            ensures: pharmacyDatabase.addLog(loginTc, loginDevlet)
                            \result = true;
          otherwise:
            requires: governmentDatabase.verifyLogin(loginTc, loginDevlet) == false;
            ensures: \result = false;
        }
        boolean patientLogin(String loginTc, String loginDevlet);
    }
}
```

```
component MedicineSelling(){
  MedicineObject Medicines[];
  bool cashierTransactionSuccesful[N]  := false;
  bool medicinePaymentIsTaken[N]:=false;

  provided port giveMedicine{
    @functional{
      requires:true;
      ensures:\result := Medicines[@];
    }
    MedicineObject giveMedicine();
  }

consumer port fromCashier{
    @functional{
         ensures:
            medicinePaymentIsTaken[patientID] := true;
            cashierTransactionSuccesful[patientID] := true;
    }
        approveMedicinePayment(ID patientID);
}
```

```
component cashier(ID N := 5){

    bool bloodPressurePaymentIsTaken[N]:= false;
    bool bloodSugarPaymentIsTaken[N]:=false;
    bool medicinePaymentIsTaken[N] := false;
    bool refundNeeded[N] := false;
    int refundAmount = null;

    consumer port BloodPressure[N]{
        @functional{
           requires:payment == Sufficient;
           ensures:
               bloodPressurePaymentIsTaken[@]:=true;
           otherwise:
                requires: payment == NotSufficient;
                ensures: refundNeeded[@] := true;
                              refundAmount = payment;
        }
     paymentforBloodPressureTest(int payment);
    }
    consumer port BloodSugar[N]{
            @functional{
               requires:payment == Sufficient;
               ensures:
                   bloodSugarPaymentIsTaken[@]:=true;
           otherwise:
               requires: payment == NotSufficient;
               ensures: refundNeeded[@] := true;
                        refundAmount = payment;
        }
     paymentforBloodSugarTest(int payment);
    }

 consumer port paymentForMedicine{
          @functional{
                requires: payment == Sufficient;

ensures:medicinePaymentIsTaken[@]:=true
             otherwise:
                  requires: payment == NotSufficient;
                  ensures: refundNeeded[@] := true;
                           refundAmount = payment;}
          paymentForMedicine(int payment);
 }
```

```
emitter port  toBloodPressureTest{
        @functional{
            promises:patientID \in[0, N-1];
            ensures:
              bloodPressurePaymentIsTaken[patientID] :=
false;
         }
         doBloodPressureTest(ID patientID);
}

 emitter port toBloodSugarTest{
        @functional{
            promises:patientID \in[0, N-1];
            ensures:
              bloodSugarPaymentIsTaken[patientID] := false;
         }
        doBloodSugarTest(ID patientID);
}

emitter port toMedicineSelling{
        @functional{
            promises:patientID \in[0, N-1];
            ensures:
              bloodSugarPaymentIsTaken[patientID] := false;
         }
       approveMedicinePayment(ID patientID);

}

emitter port refundToPatient{
            @functional{
                 requires: refundAmount != null;
                 ensures: refund := refundAmount;
                          refundNeeded[@] == false;
                          refundAmount = null;
             }
            giveRefund(int refund);
}
```

```
component MedicineAdviser{
    SymptomObject symptomList[];
    boolean symptomsAreSet = false;
    MedicineAdviseGenerator adviser;
    MedicineObject medicineAdvice = null;

    required port pmr{
        @functional{
            promises: \nothing;
            ensures: symptomList := \result;
                           medicineAdvice := adviser.generate(symptomList);
        }
        SymptomObject[] getSymptoms();
    }

    provided port pmp {
        @functional{
            requires: medicineAdvice != null;
            ensures: \result = medicineAdvice
        }
        MedicineObject getMedicineAdvice()
    }
}
```

```
Enum BloodPressureLevel := {High, Normal, Low}

component BloodPressureTest(ID N:=5){
    bool bloodPressurePaymentIsTaken[N]:=false;
    BloodPressureLevel pressure_level;
    bool bloodPressureResultsReturned:=false;

provided port BloodPressureTestResults[patientID] {
    @functional {
          requires :bloodPressurePaymentIsTaken != false;
          ensures:resultsReturned:=false;
                \result := pressure_level in\ {High, Normal, Low}
    }
    String BloodPressureTestResult();

consumer port fromCashier {
    @functional{
        requires:true;
        ensures:bloodPressurePaymentIsTaken[N]:=true;
            bloodPressureResultsReturned:=true;
      }
      ReturnResults(ID patientID ,pressure_level);
}
```

```
Enum BloodSugarLevel := {High, Normal, Low}

component BloodSugarTest(ID N:=5){
    bool bloodSugarPaymentIsTaken[N]:=false;
    BloodSugarLevel sugar_level;
    bool bloodSugarResultsReturned:=false;

 provided port BloodSugarTestResults[patientID] {
     @functional {
     requires :bloodSugarPaymentIsTaken != false;
     ensures:bloodSugarResultsReturned:=false;
          \result := sugar_level in\ {High, Normal, Low};
    }
     String BloodSugarTestResult();

   consumer port fromCashier {
    @functional{
        requires:true;
        ensures:bloodSugarPaymentIsTaken[N]:=true;
            bloodSugarResultsReturned:=true;
            sugar_level = input_sugar_level;
        }
        ReturnResults(ID patientID ,input_sugar_level);
}
```

```
connector patient2Cashier(patientRole{paymentForMedicine, paymentForBloodSugarTest, paymentForBloodPressureTest},
                          cashierRole{paymentForMedicine, paymentForBloodSugarTest,
paymentForBloodPressureTest}){

       role patientRole{
       emitter port paymentForMedicine{
               @interaction{waits: requestForMedicine && prescriptionVerified; }
                paymentForMedicine(int payment);
       }
       emitter port paymentforBloodSugarTest{
              @interaction{waits: !requestForBloodSugarTest;}
                     paymentforBloodSugarTest(int payment);
       }

       emitter port paymentforBloodPressureTest{
                     @interaction{waits: !requestForBloodPressureTest;}
                      paymentforBloodPressureTest(int payment);
        }

        consumer port getRefund{
            @interaction{waits: true;}
            giveRefund(int refundAmount);
        }
       }


        role cashierRole{
           consumer port paymentForMedicine{
                 @interaction{accepts:medicinePaymentIsTaken[@] == false;}
                 paymentForMedicine(int payment);
                   }

             consumer port paymentforBloodSugarTest{
                 @interaction{accepts:bloodSugarPaymentIsTaken[@] == false; }
                  paymentforBloodSugarTest(int payment);
             }

             consumer port paymentforBloodPressureTest{
                 @interaction{accepts:bloodPressurePaymentIsTaken[@] == false;
                  paymentforBloodPressureTest(int payment);
             }

             emitter port refundToPatient{
                 @interaction{waits: true; }
                 giveRefund(int refundAmount);
             }
    }

     connector link1( patientRole{paymentForMedicine}, cashierRole{paymentForMedicine});
    connector link2( patientRole{paymentForBloodSugar}, cashierRole{paymentForMedicine});
    connector link3( patientRole{paymentForBloodPressure}, cashierRole{paymentForMedicine});
    connector link4( patientRole{getRefund}, cashierRole{refundToPatient});
}
```

```
connector patient2DatabaseCommunicator(patientRole{scanPrescription, loginPort},

databaseCommunicatorRole{scanPrescription, loginPort}) {

      role patientRole{
            required port scanPrescription{
            @interaction{
               waits: requestForPrescriptionScan == true;
            }
            verifyPrescription(Prescription inputPrescription)
       }

       required port loginPort{
           @interaction{
               waits:loginTc != null && loginDevlet != null;
           }
           bool patientLogIn(String inputLoginTc, String inputLoginDevlet);
       }

      }

      role databaseCommunicatorRole{
         provided port scanPrescription{
           @interaction{
               waits: true;
           }
           boolean verifyPrescription(Prescription inputPrescription);
        }

         provided port loginPort{
            @interaction{
               waits: true;
            }
            bool patientLogin(String loginTc, String loginDevlet);
        }
      }

      link1( patientRole{scanPrescription}, databaseCommunicatorRole(scanPrescription});
      link2( patientRole{loginPort}, databaseCommunicatorRole(loginPort});
}
```

```
connector Cashier2BloodSugarTest(cashierRole{toBloodSugarTest},
BloodSugarTestRole{fromCashier}){
       role cashierRole{
              emitter port toBloodSugarTest{
                  @interaction{waits:bloodSugarPaymentIsTaken[@] == true;}
                  doBloodSugarTest(int payment);
             }
       }
       role BloodSugarTestRole{
              consumer port fromCashier{
                  @interaction{waits:!bloodSugarPaymentIsTaken[patientID];}
                  ReturnResults(ID patientID ,input_sugar_level);
             }
       }

      link1( cashierRole{toBloodSugarTest}, BloodSugarTestRole{fromCashier});
}
```

```
connector Cashier2BloodPressureTest(cashierRole{toBloodPressureTest},
BloodSugarTestRole{fromCashier}){
       role cashierRole{
              emitter port toBloodPressureTest{
                  @interaction{waits:bloodPressurePaymentIsTaken[@] == false;}
                  doBloodPressureTest(int payment);
              }
      }

       role BloodPressureTestRole{
              consumer port fromCashier{
                  @interaction{waits:!bloodPressurePaymentIsTaken[patientID];}
                  ReturnResults(ID patientID ,pressure_level);
             }
       }
      link1( cashierRole{toBloodPressureTest}, BloodPressureTestRole{fromCashier});
}
```

```
connector BloodSugarTest2Patient(BloodSugarTestRole({BloodSugarTestResults},
PatientRole{BloodSugarTestResults}){
   role BloodSugarTestRole{
      provided port BloodSugarTestResults {
            @interaction{accepts: bloodSugarResultsReturned[@]==true;}
            String BloodSugarTestResult();
         }

    }
    role PatientRole{
      required port BloodSugarTestResults{
            @interaction{waits:requestForBloodSugarTest==true;}
             String BloodSugarTestResult();
        }
    }
    link1( BloodSugarTestRole{BloodSugarTestResults},PatientRole{BloodSugarTestResults});
}
```

```
connector BloodPressureTest2Patient(BloodPressureTestRole({BloodPressureTestResults},
    PatientRole{BloodPressureResults}){
     role BloodPressureTestRole{
              provided port BloodPressureTestResults{
                    @interaction{ accepts: bloodPressureResultsReturned[@]==true; }
                    String BloodPressureTestResult();
      }

    }
    role PatientRole{
      required port BloodPressureResults{
            @interaction{waits:requestForBloodPressureTest==true;}
              String BloodPressureTestResult();
    }

    link1( BloodPressureTestRole{BloodPressureTestResults},
                                          PatientRole{BloodPressureResults});

}
```

```
connector cashier2MedicineSelling(cashierRole{toMedicineSelling}, medicineSellingRole{fromCashier}){
    role cashierRole{
        emitter port toMedicineSelling{
            @interaction{waits:medicinePaymentIsTaken[@] == true;}
            approveMedicinePayment(ID patientID);
        }
    }

    role medicineSellingRole{
        consumer port fromCashier{
            @interaction{waits: !medicinePaymentIsTaken[patientID];}
            approveMedicinePayment(ID patientID);
        }
    }

    link(cashierRole{toMedicineSelling}, medicineSellingRole{fromCashier});
}
```

```
connector MedicineSelling2Patient(MedicineSellingRole{giveMedicine}, PatientRole{getMedicine}){
    role MedicineSellingRole{
       provided port giveMedicine{
                   @interaction{accepts: cashierTransactionSuccesful[patientID] == true}
                    MedicineObject giveMedicine();
       }
}

    role Patient Role{
      required port getMedicine{
       @interaction{waits:requestForMedicine == true;}
        MedicineObject giveMedicine();
  }
}

    link(MedicineSellingRole{giveMedicine}, PatientRole{getMedicine});
}
```

```
component DigitalPharmacy{

	component Patient patient1();

	component MedicineAdviser medAdvIns();
	component BloodSugarTest bloodSugarTestIns();
	component BloodPressureTest bloodPressureTestIns();
	component MedicineSelling medicineSellingIns();
	component Cashier cashierIns();
	component DatabaseCommunicator databaseCommIns();

	connector patient2Cashier x1(patient1{paymentForMedicine, paymentforBloodSugarTest,
paymentforBloodPressureTest, getRefund}, cashierIns{paymentForMedicine, paymentforBloodSugarTest,
paymentforBloodPressureTest, refundToPatient});

	connector patient2DatabaseCommunicator{patient1{scanPrescription, loginPort},
databaseCommIns{scanPrescription, loginPort});

	connector Cashier2BloodSugarTest(cashierIns{toBloodSugarTest}, bloodSugarTestIns{fromCashier});

	connector Cashier2BloodPressureTest(cashierIns{toBloodPressureTest},
bloodPressureTestIns{fromCashier});

	connector Cashier2MedicineSelling(cashierIns{toMedicineSelling}, medicineSellingIns{fromCashier});

connector BloodSugarTest2Patient(bloodSugarTestIns{BloodSugarTestResults},
patient1{BloodSugarTestResults});

	connector BloodPressureTest2Patient(bloodPressureTestIns{BloodPressureTestResults},
patient1{BloodPressureResults});

	connector MedicineSelling2Patient(medicineSellingIns{giveMedicine}, patient1{getMedicine});

}
```

2.3. Traceability Between Requirements and Architecture

All of the functional requirements have been addressed in the XCD architecture design.

The system can handle more than one patients at a time. Therefore, each Cashier, MedicineSelling, BloodPressureTest and BloodSugarTest components have N ports each.

A system that takes symptoms and returns an optional medicine is also created in our XCD model. Patient and MedicineAdviser components have a reciprocal communication where Patient sends their list of symptoms to the MedicineAdviser component, and MedicineAdviser component returns an optional medicine to the **Patient** component. This algorithm that infers possible illnesses from a list of symptoms isn't explicitly shown in the model as our aim is just to show the general architecture of the system.

Process of buying a medicine isn't limited to just describing a symptom and getting advice from the system. Patients can also pick a medicine of their choice and buy it.
There are two possible scenarios in this situation.

1. The medicine can be a controlled substance, in this case, a prescription is needed. Prescription is given by the patient and scanned. This scanned result is then sent to the government database in order to be verified by a mediator component that is called **DatabaseCommunicator**. This component handles the communication between our system and the government system(in Turkey, this system is called Medula). If the prescription can be verified, the patient's payment is accepted into the **Cashier** component and if the payment amount is sufficient enough, Cashier component emits a signal to the **MedicineSelling** component, indicating that the payment is successful and medicine can be given to the patient.

2. The other scenario is where the medicine that is needed by the patient isn't a controlled substance. In this case, no verification is required and the patient can make their payment, and if the payment amount is sufficient, the transaction is successful.

Patients can also have their blood pressure and blood sugar tested via **BloodPressureTest** and **BloodSugarTest** components. Normally these tests are done freely, but in order to prevent a lot of people occupying our digital pharmacy, these tests are now done by payment. Just like the process in the MedicineSelling component, the patient first requests a test and makes the payment to the **Cashier** component. If the payment is sufficient enough, the Cashier component emits a signal to **BloodSugarTest** or **BloodPressureTest** components and the testing process starts. After the testing ends, the results are returned to the **Patient** component.

At every process where there is a payment, we check if the payment is sufficient or not sufficient. If the **Patient** component sends an amount that is not sufficient, that amount will be refunded back to the **Patient** component by emitting a signal from **Cashier** component to **Patient** component.

One of the most important non-functional requirements we have is security. To make our system secure, we require each user to be logged in before requesting to buy a prescribed medicine. In addition to this, every login is logged in the pharmacy database to trace any possible system attacks or system errors. Buying prescribed medicines requires scanning the prescription, getting verified by the government database and finally verifying if the payment for the medicine is sufficient enough. By having all these steps in the process, our system provides a secure environment.

The other two most important non-functional requirements are maintainability and portability. In our XCD model, we used connectors in order to define the interaction protocols between components. This means that the interaction part is independent from the components itself. This makes our components protocol independent and reusable. If we want to use the same component in a different context, we can just configure the connector in a different way and have no problems. Thus, we can say that our system components are highly portable and maintainable. Also, the fact that our components are reusable means that there won't be any engineering overhead while building this system for different customers. This decreases the engineering costs dramatically and makes the system more affordable too.

The system can serve N patients at the same time. This number can change according to the pharmacy system that is going to be built using this system. It can go up or down according to the needs of the customer. Having a system that can handle concurrent processes like this increases the efficiency and response time of our pharmacy. Therefore, we can say that our system is highly efficient in terms of response time.

In terms of reliability, we designed our system to be able to handle processes from different patients at the same time by having more than one port. This prevents any deadlock from happening. Unlike traditional pharmacies, our pharmacy will run 24/7 simply because it’s an electronic machine and will help the patients all the time.

Rest of the non-functional requirements such as hygiene and usability cannot be proven by looking at the XCD modeling. The usability part depends on how human-friendly the user interface will be. Unfortunately we cannot show how the user interface will look like here. In terms of hygiene, the system should disinfect itself after every blood test, but it is hard to show this type of thing on an architecture model.

3. Model-to-Code Transformation Algorithm Specifications

In this part, all the components that were defined in the XCD model will be written as pseudocodes.

The most complex algorithm that we have is the algorithm that takes symptoms as an input and gives out a possible illness. This is in the MedicineAdviser component. In order to do this, we need a pre-made data pool that has symptoms and illnesses linked to each other. WebMD's symptom checker algorithm is a great example of this.

A symptom can be an indicator of more than one illness, sometimes it can be the main symptom of the illness, sometimes a minor symptom. For example, fatigue can be a strong symptom of malnutrition, but it also can be a minor symptom of cancer. If you have a fatigue symptom, it's more likely that it's linked to malnutrition and less likely to be linked with cancer. Therefore, each illness will have a weight value. If a symptom is a major symptom of said illness, it will have more weight and if it's a minor symptom it will have a smaller weight value.

In addition to this, we also need a mapping of illnesses to the optimal medicines. This is also a pre-made data pool.

```
class MedicineAdviser{
	HashMap<Symptom, Illness[]> symptomIllnessHM();
	HashMap<Illness, Medicine> illnessMedicineHM();

	Medicine giveAdvise(Symptom[] symptomList) {
		List<Illness> possibleIllnesses;

		for symptom in symptomList
			possibleIlnesses.add(symptomIllnessHM[symptom]);
			increment possibleIllnesses weight by the symptoms weight value;

		Illness mostProbableIllness = possiblelllnesses.findTheMaximumWeightedIllness();

		return illnessMedicine.get(mostProbableIllness); //returns optional medicine
	}
}
```

The DatabaseCommunicator class helps the communication between the government system and our system. It also logs the patients who logged in to use it after for security purposes.

```
class DatabaseCommunicator{

	URL governmentPrescriptionVerifyApi = government.com/verifyPrescription;
	URL governmentLoginAuthorizationApi = government.com/authorizeLogin;
	PharmacyRepository pharmacyRepository;


	bool scanPrescription(Prescription prescription) {
		result = sendHTTPRequest(prescription, governmentPrescriptionVerifyApi);
		if (result == successful )
			return true;
		else
			return false;
	}

	bool patientLogin(Username username, Password password)
	{
		result = sendHTTPRequest(username, password, governmentLoginAuthorizationApi)
		if (result == successful )
		{
			pharmacyRepository.addLog(username, password);
			return true;
		}
		else
			return false;
	}

}
```

Patient class doesn't provide much functionality. It's rather a user of the already existing functionalities that other components provide.

```
class Patient{
	Symptom[] symptomList; MedicineObject medicineAdvice;
	Payment payment; Medicine boughtMedicine;
	Prescription prescription; boolean prescriptionVerified = false;
	Money changeTaken = 0;
	boolean loginSuccesful = false; boolean paymentSucceeded = false;
	String username; String password;
	String bloodSugarLevel; String bloodPressureLevel;
	BloodSample bloodSample; BodySignal tensionSignal;

ServiceRequest serviceRequest = null;
	MedicineSelling medicineSelling; MedicineAdviser medicineAdviser;
	DatabaseCommunicator databaseCommunicator; Cashier cashier;

	Symptom[] getSymptoms() {
		return symptomList;
}

	void scanPrescription(){
		this.prescriptionVerified = databaseCommunicator.scanPrescription(prescription);
	}

void testBloodSugar() {
		if(paymentSucceded && serviceRequested == bloodSugarTest)
this.bloodSugarLevel = bloodSugarTest.BloodSugarTestResult(bloodSample);
}

void testBloodPressure() {
		if(paymentSucceded && serviceRequested == bloodPressureTest)
			this.bloodPressureLevel =
					bloodPressureTest.BloodPressureTestResult(tensionSignal);
	}

	void getMedicine() {
		if(paymentSucceded && prescriptionVerified && serviceRequested == medicine)
		      this.boughtMedicine = medicineSelling.getMedicine(medicineRequest);
	}

void makePayment() {
		MoneyAndBoolean obj = Cashier.makePayment(payment, serviceNeeded);
		if(obj.Boolean == true){
			paymentSucceeded = true;
			this.changeTaken = changeTaken + obj.Money;
		} else{
			paymentSucceeded = false;
			this.changeTaken = changeTaken + obj.Money;
		}
	}

	void login() {
		loginSuccessful = databaseCommunicator.patientLogin(username, password);
	}
}
```

This is the cashier class. It is much more basic than the one that we modeled with XCD. It takes a payment and the type of service. If the payment is enough for the given service, it returns change(if payment is more than the service cost). If the payment is not enough, it just returns the given money.

```
class Cashier {
      BloodSugarTest bloodSugarTestComponent;
      BloodPressureTest bloodPressureTestComponent;
      MedicineSelling medicineSellingComponent;

      HashMap<Service, Money>  listOfServices = [{bloodSugarTest,5.00TL}, {Aspirin,
15.00TL},.. ]

      MoneyAndBoolean makePayment(Money moneyObject, Service service){

            if(moneyObject.amount >= listOfServices[service].cost) {
                  return {moneyObject.amount -  listOfServices[service].cost , true};
            }
            else {
                  return {moneyObject.amount, false};
            }
      }
}
```

Here are the service components: MedicineSelling, BloodSugarTest and BloodPressureTest.

```
class MedicineSelling() {
	MedicineObject Medicines[];
	bool cashierTransactionSuccessful[N]  := false;
	bool medicinePaymentIsTaken[N]:=false;

	MedicineObject getMedicine(MedicineRequest medicineRequest){
		if Medicines.contains(medicineRequest)
			return Medicines.get(medicine);
	}
}
```

```
class BloodSugarTest() {
	String bloodSugarLevel;
	BloodSugarTestDevice device;

	String BloodSugarTestResult(BloodSample bloodSample){
		return device.measureBloodSugarLevel(bloodSample);
	}
}
```

```
class BloodPressureTest() {
	String bloodPressureLevel;
	BloodPressureTestDevice device;

	String BloodPressureTestResult(BodySignal tensionSignals){
		return device.measureBloodSugarLevel(tensionSignals);
	}
}
```

4. Lessons Learned

What is an architecture decision?

It is a decision about how we should model the system we are developing. This design model shows every stage and layer of the system and it helps a lot in the later developing stages.

Importance of architecture decision.

In today's world almost every technological device has a system. The development process can change according to the system’s complexity. Some systems are basic and can be built by one person. Some are so big they have other systems inside their system. Development stage of these kinds of systems must be meticulous and with multiple substages. Model based designs are an essential and important part of the development of systems. Engineers decide models according to the needs of the system and how it will work, how it will interact with the outside world. Decided architecture design guides the rest of the project. These models help engineers from all fields to understand systems and guide them to implement the correct needs. Architecture is where your system is converted into model and this model shows all the needs of the system. All the systems will be built over this. While it seems so complex if you decide and implement a correct architecture and follow it, the rest of the project will be flawless.

How did we use it.

Our digital pharmacy project is a complex system. It will be used globally and in many environments. It must be portable and maintainable. If we want to provide these requirements must be tested and bugs can be fixed easily. Getting inspiration from the gas station model for our payment system helped us create our system. We also used connectors for re-usability.

5. Conclusion

While making this project, we gave importance to model based software architecture. Software developers act on software architecture. Software architecture guides them and makes their work easier. By following these steps in our software architecture models, both customers and developers understand more clearly what we want.Especially big projects definitely need software architecture because big projects are wide-ranging and have big teams. These big teams create various communication problems when handling a project. Software architecture creates a barrier to these various problems. In addition, the software architecture saves us time and money. It allows us to have an idea before starting the project and to answer the questions in our minds.And that's why we need software architecture before we start implementing our project.

6. Definitions, Acronyms and Abbreviations

XCD: A novel architecture modelling language. Provides enhanced re-usability via first-class connectors, design-by-contract based modelling, guaranteed realisability of software architectures and automated formal verification.

Model-view-controller: Model–view–controller (MVC) is a software design pattern commonly used for developing user interfaces that divide the related program logic into three interconnected elements. These elements are model, view and controller.

Software Component: Software components are **parts of a system or application**. Components are a means of breaking the complexity of software into manageable parts.

Connector: Connector is a means of making the components protocol independent. They connect the components and hold protocol information.