Mini Project Test

Case: Pay Station

Indhold

[Introduction 3](#_Toc64814678)

[Exercises – System Development 4](#_Toc64814679)

[Exercise 1: Planning of Functional Testing at System Level (system testing) 4](#_Toc64814680)

[Example of System Test Cases 4](#_Toc64814681)

[Exercise 2: Planning and Design of Unit and Integration Tests 5](#_Toc64814682)

[Exercises - Programming 6](#_Toc64814683)

[Exercise 3: Database and Code Setup 6](#_Toc64814684)

[Exercise 4: Implementation of Test Cases 6](#_Toc64814685)

[Exercise 5: Make Changes and Test 7](#_Toc64814686)

[Submission and Evaluation 8](#_Toc64814687)

[Appendix 1: Description of The Pay Station 9](#_Toc64814688)

[Price and Calculation 9](#_Toc64814689)

[Appendix 2: Customer Interview 10](#_Toc64814690)

[Request 1a – Buy Parking Ticket Using Euros 10](#_Toc64814691)

[Request 1b – Buy Parking Ticket Using Danish Krone 10](#_Toc64814692)

[Request 1c – Buy Parking Ticket Using Both Euros and Danish Krone 10](#_Toc64814693)

[Request 2 – Cancel a Transaction 11](#_Toc64814694)

[Request 3 – Reject Illegal Coin 11](#_Toc64814695)

[Request 4 – Updating the Price 11](#_Toc64814696)

[Appendix 3: Use Cases and Domain Model 12](#_Toc64814697)

[Use Cases 12](#_Toc64814698)

[Domain Model: 15](#_Toc64814699)

[Appendix 4: SSD, Architecture and Interaction Overview 17](#_Toc64814700)

[SSD for Use Cases: *Buy Ticket and Update Price* 17](#_Toc64814701)

[Architecture and Database 17](#_Toc64814702)

[Design of the Interactions 18](#_Toc64814703)

[Appendix 5: Sequence Diagram after Changes 22](#_Toc64814704)

[References 23](#_Toc64814705)

# Introduction

You are employed in the testing department of a small company that makes software for payment machines.

Your company has been contacted by the company Easy Parking A/S, which wants to replace their outdated parking meters with newer electronic payment machines. Easy Parking A/S has asked your company to supply the software for the payment machines. Another company, Better Electronics A/S, will provide the hardware, their product is known as B-423E.

The developers have started to develop the system. It is agreed that the developer should only perform the tests necessary to ensure a sufficient quality of the critical code. But it is the testing department (you) that is responsible for testing. After a couple of meetings with Easy Parking A/S the developers have produced a description of the main Use Cases and a Domain Model. The developers have also documented the design while they were writing the code. The developers have *almost* finished their work, except for the implementation of small features. Due to time constraints, they have first sent the analysis and design to the test department (you). This means that you have not been able to start the planning and the specification of Test Cases just after the Use Cases was described, and the first design was ready!

Easy Parking A/S is pressing for getting the software completed, as it is the only thing they are missing before the payment machines can be used. The overall project is delayed and since the developers have tight deadlines on other critical projects , they have asked the developers to send the code produced to the test department for being tested **even if it is not yet finished**.

When you look at the requirements and design documents, you see that they are not very precise and there is no design for all the written code (something programmers often do). Since the developers are under pressure it is agreed that in addition to testing all the documentation and code, you (the test group) must also fix the errors found or complete the missing parts, even if it should the developers’ responsibility. Therefore, it is **important that it is clear in the code which modifications have been made by the test department**.

# Exercises – System Development

## Exercise 1: Planning of Functional Testing at System Level (system testing)

Please consider the following appendices:

* Appendix 1: Description of the pay station
* Appendix 2: Scenarios
* Appendix 3: Use Cases and Domain Model

Specify System Test Cases for the Use Cases: *Buy ticket* and *Update price* using the following approach:

1. Complete the specification of the Fully Dressed Use Cases adding the alternative flows of events which cover behaviour of an optional or exceptional character relative to normal behaviour, and variations of the normal behaviour.
2. Generate a full set of Use-Case Scenarios.
3. For each test scenario identify Test Cases.
4. Review and validate the Test Cases identified. Afterwards identify actual values to be used in implementing the final tests.

The Test Cases can be formulated based on the different coin combinations see Appendix 2 and the principle of Equivalence Classes. The table below shows an example of Test Cases for Scenario 1a (Appendix 2): remember also test for abnormal data combinations.

### Example of System Test Cases

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Test Case no. | Currency = EURO | | Currency = DKK | | Currency = NKK | | Expected result | |
| Cent | € | Øre | Kr. | Øre | Kr. | Per second | Per minute |
| Value | Value | Value | Value | Value | Value |
| 1:  Scenario 1a, EURO  One coin, Parking sec. round up | 1 |  | 0 | 0 | 0 | 0 |  |  |
|  |  |  |  |  |  | 24 | 1 |
| 2  Scenario 1a, EURO  More coins | 0 | 1 | 0 | 0 | 0 | 0 |  |  |
| 0 | 2 | 0 | 0 | 0 | 0 |  |  |
|  |  |  |  |  |  | 7200 | 120 |

## Exercise 2: Planning and Design of Unit and Integration Tests

For the planning and the design of Unit and Integration Testing you should consider Appendix 4 “SSD, Architecture and Interaction Overview”. Follow these steps:

1. Review each interaction diagram
2. Decide which methods you want to test
3. Consider whether it is a Unit or an Integration Test
4. For the selected methods, specify the Test Cases using the equivalence class partitioning and boundary value testing techniques.

It is expected that you as minimum create Test Cases for:

* *addPayment()* method (see interaction diagram for addPayment() )
* *insertParkingBuy()* method (see interaction diagram for buy() )
* *getPriceByZoneId()* method (see interaction diagram for updatePrice() )

In addition, consider other unit or components that you think should be tested (e.g. the connection to the database).

The following is an example of the above approach. The method *addPayment (*amount, currency, cointype*)* is an important method in *ControlPayStation*. Input for the test is: *coinValue, currency and coinType.*

Equivalence Classes for the currency of the coin can be the following:

* Valid currency = DKK and EURO.
* All other currencies are invalid.

Equivalence classes for the value of coins can be the following:

* Valid coins:
  + Euro:
    - 1, 2, 5, 10, 20, 50 cent
    - 1, 2 €
  + DKK:
    - 50 øre
    - 1, 2, 5, 10, 20 kr.

All other coins are invalid.

The *addPayment (amount, currency, cointype)* method adds each payment resulting in the total amount inserted in coins. The *payStation.getTimeBoughtInMinutes (): int* method returns the parking time in minutes based on the total amount inserted into the Pay Station.

The expected parking time should be greater than 0 minutes, if the input is valid; 0 if no coin was entered or if only invalid coins have been entered.

# Exercises - Programming

## Exercise 3: Database and Code Setup

Before implementing the Test Cases you need to set up the database and the code provided. Follow the instructions in the file Preparation.pdf.

## Exercise 4: Implementation of Test Cases

It is time to implement your Test Cases. In the code provided only few JUnit tests are made. You must write JUnit tests, and only slightly modify the Java code.

***Implement and run test cases for the use case: Buy Ticket***

Begin to implement the tests for the test cases you have found in the exercise 2 for the Buy Ticket use case. Check that the code behaves as expected. Otherwise, fix the code until the tests pass.

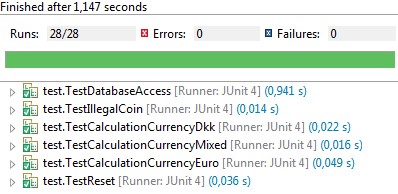
"Best practices" for JUnit test:

* Keep JUnit tests very simple
* Create small specific test methods (do one thing)
* Test method name shall indicate expected result
* Use the AAA pattern
  1. Arrange
  2. Act
  3. Assert

Below is a code example for the implementation of the Test Case no. 1 (see example in Exercise 2)

|  |
| --- |
| @Before  public void setUp() {  ps = new ControlPayStation();  }  /\*\*  \* Entering 5 cents should make the display report 2 minutes parking time  \*/  @Test  public void shouldDisplay2MinFor5Cents() throws IllegalCoinException {    // Arrange  int expectedParkingTime = 2; // In minutes  int coinValue = 5;  Currency.ValidCurrency coinCurrency = Currency.ValidCurrency.EURO;  Currency.ValidCoinType coinType = Currency.ValidCoinType.FRACTION;  // Act  ps.addPayment(coinValue, coinCurrency, coinType);  // Assert  assertEquals("Should display 2 min for 5 cents", expectedParkingTime, ps.readDisplay());  }  @After  public void cleanUp() {  ps.setReady();  } |

Continue to write and run tests until all have been tested, i.e. all Test Cases are implemented, run and there are no failures, something like this:



#### Implement and run Test Cases for the Use Case: Update Price

Again: Write and run tests until everything is tested (all Test Cases are implemented and report no errors).

## Exercise 5: Make Changes and Test

In this exercise you must refactor the class PPayStation, so some of the responsibility will be delegated to another class. This can be done by introducing a utility class Calculation, as "Meta Class", which will be responsible for performing the calculations. The changes are shown in the sequence diagram in Appendix 5.

#### Refactor Code and Run Tests

Modify the code so it matches the changes in the diagram. Then, run the tests - they would ideally go through.

#### Consider New Test Cases

Consider Test Cases for Calculation class. Specify the Unit Test Cases and implement them. Continue until all tests are implemented and run without failures.

# Submission and Evaluation

One report per group. The report must contain the following elements:

* The revised Fully Dressed Use Cases with alternative flows and acceptance criteria
* System Test Cases
* Unit- and Integration Test Cases:
  + Choose 4-8 Test Cases you think best contribute to assure the quality of the software:
    - Give the motivation
    - Describe each test case and show the implementation
  + Describe the use of JUnit annotations as @Before, @BeforeClass, @After, @AfterClasstests in your test classes and give examples

Remember the formalities in the front page of the report, including your names.

**The complete report is submitted in Teams (see the Introduction document for the exact deadline and hand-in details).**

**The evaluation will take place in the first System Development lesson after the workshop. The whole class will be present.**

# Appendix 1: Description of The Pay Station



The pay station accepts Euros and Danish krone coins. When the first coin is inserted, the display shows the total parking time in minutes (bought so far) as well as two buttons marked resp. BUY and CANCEL.

Technically, the model B-423E has a 256 KB memory chip and has a built-in hardware so that it can communicate with other devices, for example a database server via Wireless Session Protocol (WSP).

However, the model does not receive payment by debit card, but Easy Parking has ensured that the pay stations can later be expanded with this facility. As is often the case when existing technology is replaced by electronics, the initial requirements are basically to provide the same basic behaviour as the old mechanical parking meters.

### Price and Calculation

The pay station accepts payment in two currencies: Danish krone and Euro. All the valid coins in the two above mentioned currencies are valid for the payment of the parking, so the following coins are valid:

* Euro
  + 1, 2, 5, 10, 20, 50 cents
  + 1, 2 €
* DKK
  + 50 øre
  + 1, 2, 5, 10, 20 kr.

The price for parking is given as the number of parking seconds one can park for 1 cent. For example, 1 cent gives a parking time of 24 seconds– and 5 cents gives a parking time of 120 seconds (2 minutes). More examples in Exercise 2.

Prices are always in Euro. If the driver pays in Danish coins, the value is converted into Euro. More precisely the following is valid:

* A driver can pay with a mix of valid coins (both Euro and Danish krone coins).
* The total parking time is always calculated based on the total amount entered in the pay station converted into cents. The amount is a decimal number to avoid rounding inaccuracy. For example, 1 Euro and 5 DKK gives ca. 166.6666667 cents (with a rate of 7.50 DKK/euro).
* If the calculated parking time is not a whole number of minutes, then rounded up to the next whole minute. For instance, 166,6666667 cents give a parking time of 4000 seconds (with a price of 24 seconds / 1 cent). This gives ca. 66,66666667 minutes, which will be rounded up to 67 minutes parking time.
* Each pay station is situated in a parking zone (1, 2 or 3) and the parking price depends on the pay station parking zone.

# Appendix 2: Customer Interview

In collaboration with Easy Parking and Better Electronics the following requests have been identified.

### Request 1a – Buy Parking Ticket Using Euros

A German car driver walks to the pay station to buy parking time. He enters several valid Euro coins as payment.

On the pay station’s display, he can see how much parking time he has bought so far. Once he is satisfied with the amount of them, he presses the button marked BUY. He receives a printed parking ticket stating the number of parking minutes he has bought. The purchase is saved in the remote database. The display is cleared to prepare for another transaction.

### Request 1b – Buy Parking Ticket Using Danish Krone

A Danish car driver walks to the pay station to buy parking time. He enters several valid Danish coins as payment.

On the pay station’s display, he can see how much parking time he has bought so far. Once he is satisfied with the amount of them, he presses the button marked BUY. He receives a printed parking ticket stating the number of parking minutes he has bought. The purchase is saved in the remote database. The display is cleared to prepare for another transaction.

### Request 1c – Buy Parking Ticket Using Both Euros and Danish Krone

A Danish car driver has just come home from vacation abroad. He walks to the pay station to buy parking time. He enters several valid Danish and Euro coins as payment.

On the pay station’s display, he can see how much parking time he has bought so far. Once he is satisfied with the amount of them, he presses the button marked BUY. He receives a printed parking ticket stating the number of parking minutes he has bought. The purchase is saved in the remote database. The display is cleared to prepare for another transaction.

### Request 2 – Cancel a Transaction

A driver has entered several coins but realizes that the accumulated parking time shown in the display exceeds what she needs. She presses the button marked CANCEL and her coins are returned. Nothing is saved in the database. The display is cleared to prepare for another transaction.

### Request 3 – Reject Illegal Coin

A driver has entered 50 cents in total and the display reads 20 minutes. By mistake, he inserts 1 Norwegian Krone which is not a recognized coin. The pay station rejects the coin and the display is not updated.

### Request 4 – Updating the Price

Every night at 3 AM the pay station retrieves the actual price for the parking zone from the remote database.

# Appendix 3: Use Cases and Domain Model

## Use Cases

The following Use Case Diagram shows the Use Cases for the Pay Station system:



The Use Case Buy Ticket is the main use case of the system. The actor is a customer that wants to buy a parking ticket. In the Use Case Update Price, the actor is an automated “Timer”, which gets the price from the central parking system every night at 3 o’clock. Both Use Cases are associated with the actor represented by the central parking system, that provides the services for the registration of the purchase and the retrieving of the price.

The following are the Fully Dressed Use Cases from which you must generate System Test Cases.

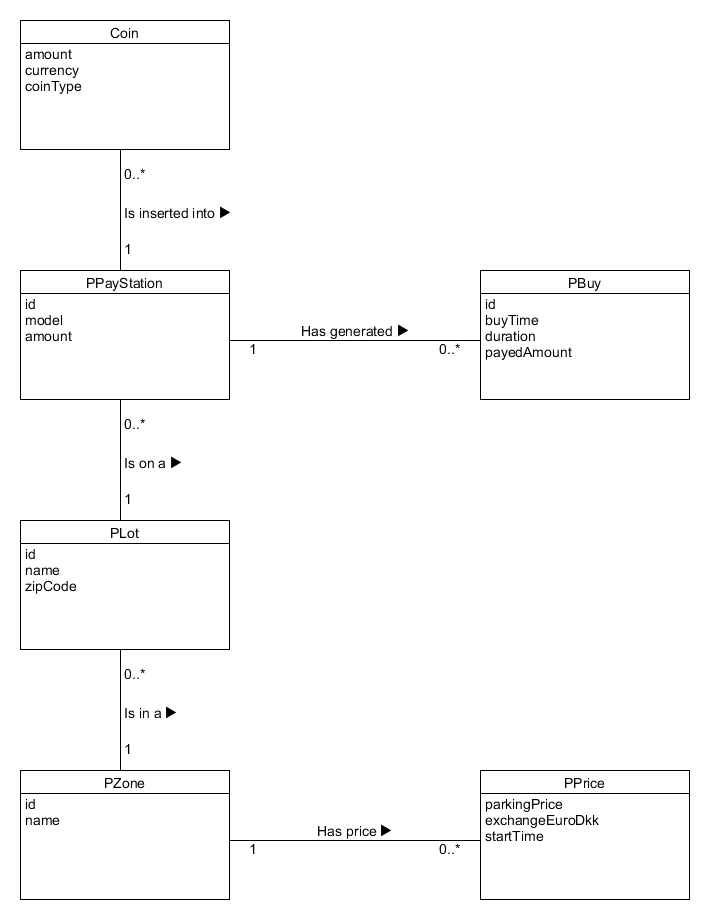
#### Use Case: Buy Ticket – Fully Dressed

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Use Case name | **Buy Ticket** | | | |
| Actors | Customer | | | |
| Pre-conditions | None | | | |
| Post-conditions | Coins are in the pay station and the parking ticket with the parking time is printed. | | | |
| Frequency | Max 200 / hour | | | |
| Flow of events |  | **Actor** |  | **System** |
| 1. | A customer walks to the pay station to buy parking time. |  |  |
| 2. | The customer enters the first coin. | 3.  4. | Checks if the coin is valid  Registers payment and updates the display which shows how much time he has bought so far. |
|  |  |  | 5. | Displays the buttons BUY and CANCEL. |
| 6. | The customer enters the next coin. | 7.  8. | Checks if the coin is valid  Registers payment and updates the display which shows how much time he has bought so far. |
|  | *The customer repeats step 6 -8 until he is satisfied with the time bought* | | |
| 9. | The customer presses the button marked BUY | 10. Registers the purchase in the central database and prints the ticket. | |
|  |  | 11. The display is cleared to prepare for another transaction. | |
| Alternative flows |  |  | | |
|  |  |  | |
|  |  |  | |
|  |  |  | |
|  |  |  | |
| Special Requirements |  |  |  | |
|  |  |  | |
|  |  |  | |

#### Use Case: Update Price – Fully Dressed

|  |  |  |  |
| --- | --- | --- | --- |
| Use Case name | **Update Price** |  |  |
| Actors | Timer |  |  |
| Pre-conditions | None |  |  |
| Post-conditions | Prices are updated in the pay station |  |  |
| Frequency | One time every night, at 3 AM |  |  |
| Flow of events | **Actor** |  | **System** |
| 1. The Use-case starts when it is 3 AM. |  |  |
|  | 2. | The current prices from the parking system are retrieved and saved locally in the pay station. |
| Alternative flows |  |  |  |
|  |  |  |
|  |  |  |
| Special Requirements |  |  |  |

## Domain Model:

The pay stations, their location and the different purchases are recorded in the central parking system. The Domain Model looks like this:

**NOTE**: The developers did not have time to implement the class Plot yet.

**NOTE**: For now, there is a Coin class only and no sub-classes (EuroCoin, DkkCoin).

The above Domain Model captures the contents of the entire parking system.

The purchases are represented by the class *PBuy*. Each purchase is done at a specific pay station *PPayStation*. A Pay Station is situated in a parking lot *Plot,* which is located in a parking zone *PZone*. A price, represented by *PPrice,* is associated to the parking zone it is valid for.

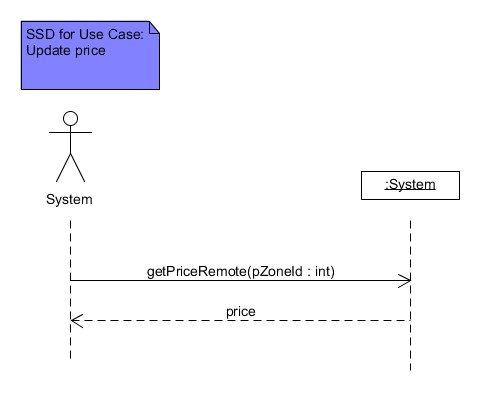
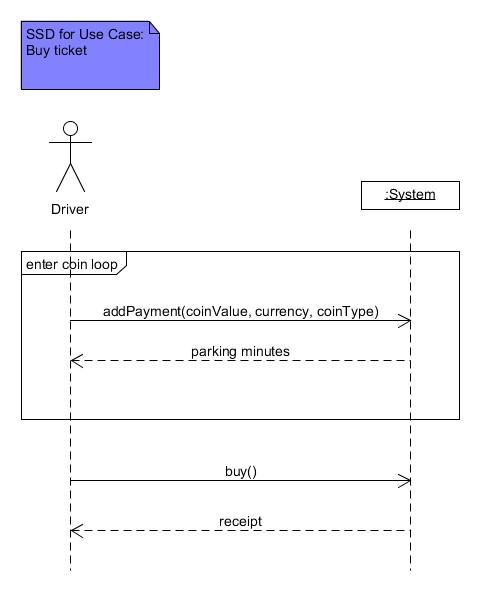
Locally, on each pay station, the following are saved:

* *PPrice* object which is retrieved every night from the central parking system.
* *PZone* object with identification of the parking zone*.*
* *PBuy* object- form the basis for the save in the parking system.
* *PPayStation* object – with the information about the pay station.
* *PReceipt* object *–* shows the time bought.

# Appendix 4: SSD, Architecture and Interaction Overview

## SSD for Use Cases: *Buy Ticket and Update Price*

Based on fully dressed descriptions of the above Use Cases, the following System Sequence Diagrams are made.



## Architecture and Database

It has been decided to build the system in a 3-layered architecture.

In general, there is a controller class per Use Case.

When the classes do too many different things helper classes can be used to maintain "high cohesion" and promote reuse. The application contains a single utility/helper class at the moment. The methods in the helper class are "static". This is shown in UML by using the stereotype "metaclass".

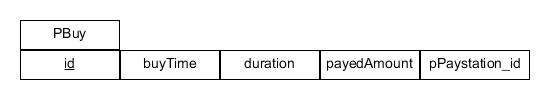
These are the following interfaces, in the Database Layer :

* DbPBuy
* DbPPrice

Data for the purchase is saved in the central database. Connection to the database is a Singleton:

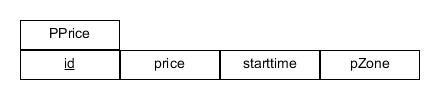
Connection con = DBConnection.getInstance().getDBcon();

The table PBuy looks like:



The primary key id is autogenerated. The attribute pPaystation\_id is a foreign key referring to the id of the PayStation.

The table PPrice looks like:



pZone is a foreign key referring to the id in PZone.

On each Pay Station, only the price is retrieved every night and saved in a local PPrice object. The Pay Station is configured with a fixed rate of 7.50 DKK / Euro.

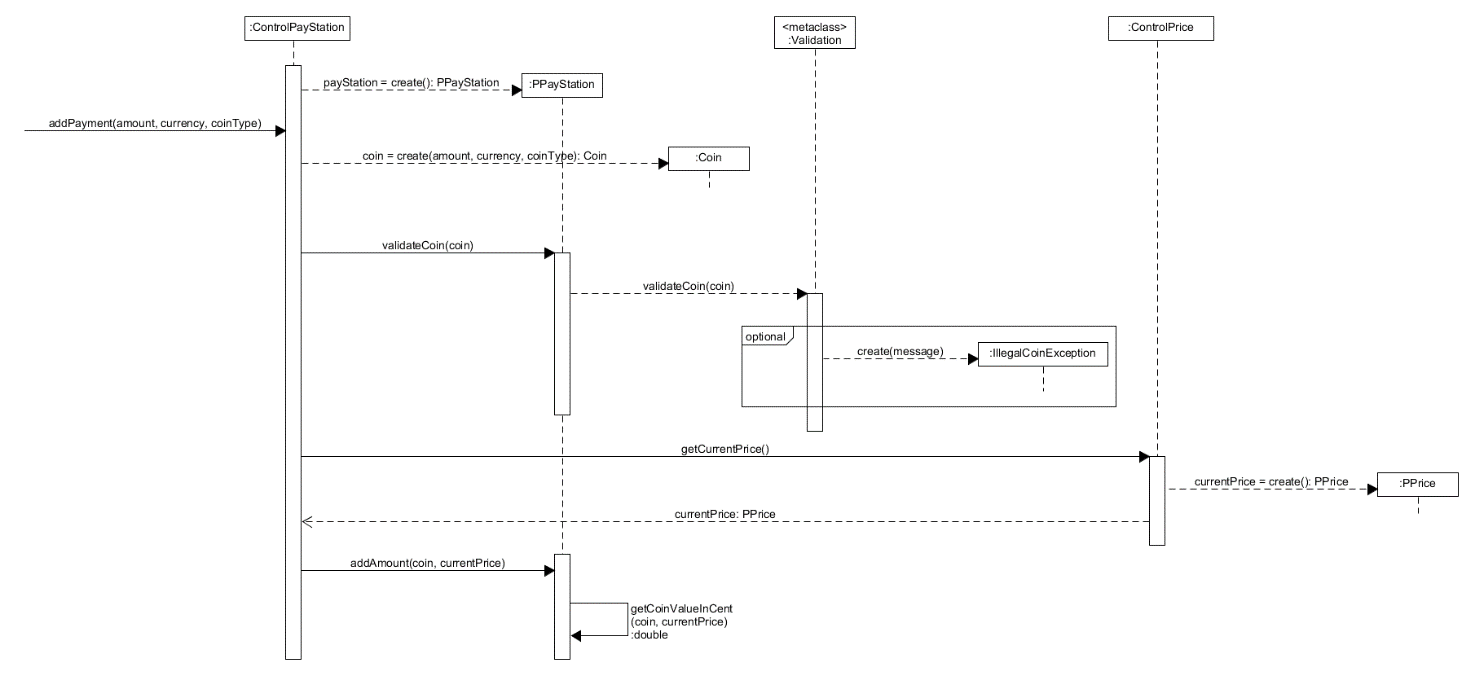
### Design of the Interactions

The developers have done an Interaction Diagram for each system event in the SSD.

#### Use Case: Buy Ticket

#### addPayment(coinValue, currency, cointype)

The Sequence Diagram for addPayment() is shown below:



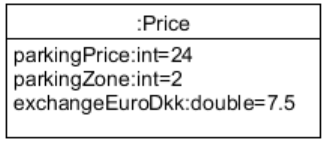
The overall steps involved in buying a parking ticket is as follows:

1) The customer initiates buying a parking ticket by inserting a coin into the Pay Station. This invokes the *addPayment()* method in the *Pay Station controller*. The parameters entered (amount, currency, coinType) is used to create an instance of the *Coin* class.

2) The coin is then validated as the Pay Station only accepts certain coin types. If the coin is invalid an *IllegalCoinException* is thrown.

3) If the coin is valid, then the coin amount is added to the total amount added to the Pay Station, so far. The total amount so far is in Euro cent. Since the total amount is in Euro cent, we need to convert Danish coins to Euro cents. Therefore, we call the *Price* controller to get the current price.

The PPrice object returned looks like this:



#### buy()

The diagram below shows the interactions for the operation buy(). The "MetaClass" LocalDate is used to find the current time from the machine time.

1) After the customer is finished inserting coins into the Pay Station, the Buy button is pressed. This invokes the *bu*y() method in the *Pay Station controller.*

2) The current time of purchase is registered by using *LocalDate* from the Java library.

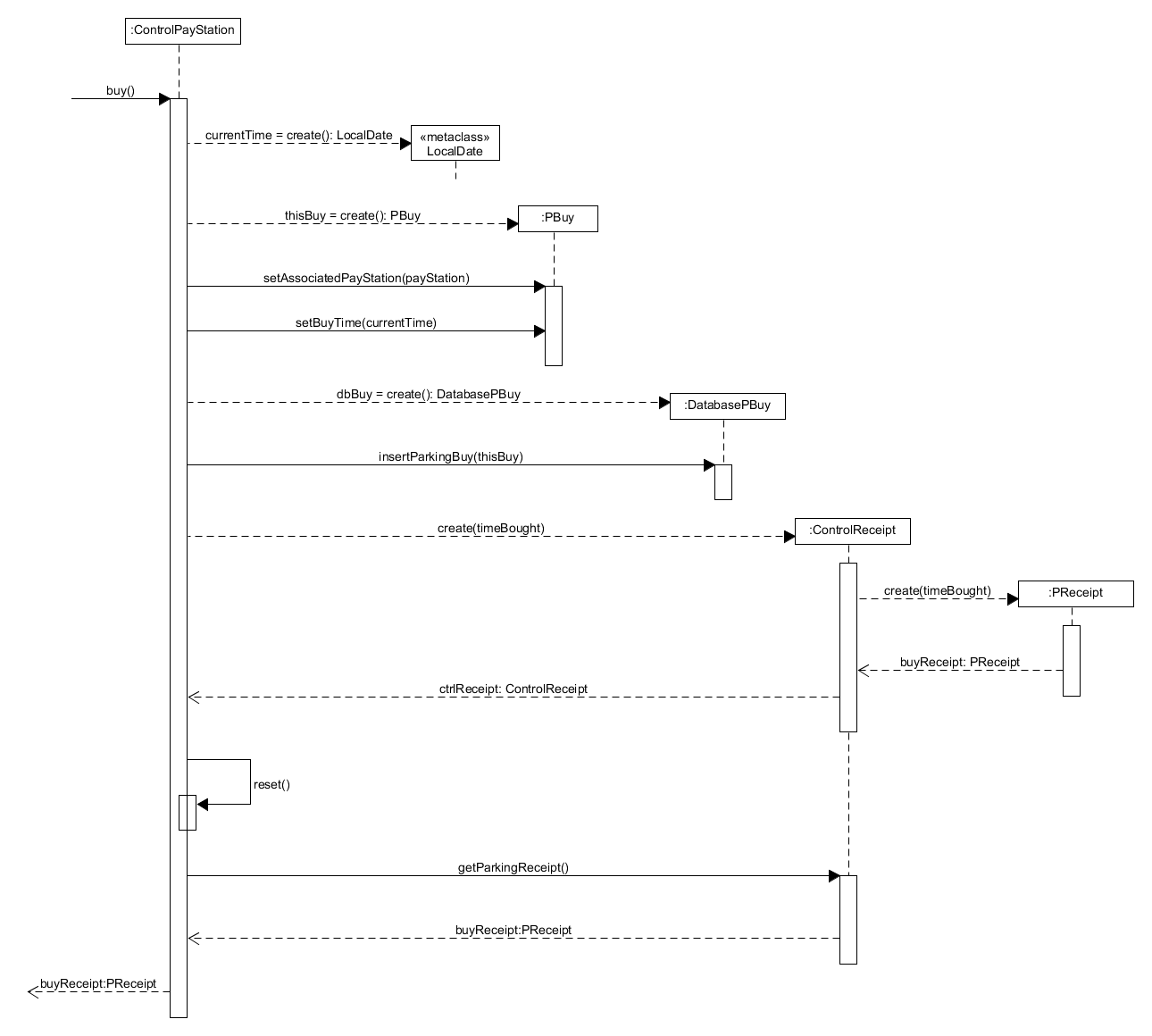
3) The buy process then generates an instance of the *PBuy* class (*thisBuy*). The current time and the specific Pay Station are then set on the *PBuy* instance.

4) Saving to the database is done by passing the object *thisBuy* to the *insertParkingBuy()* method of an instance of the DB layer class *DatabasePBuy*. The primary key is autogenerated. Issues with saving to the database are handled through exceptions.

5) After having saved *thisBuy* in the database, a receipt needs to be generated. This is done by having the Pay Station controller call the Receipt controller. The Receipt controller creates an instance of the PReceipt class using the parking duration.

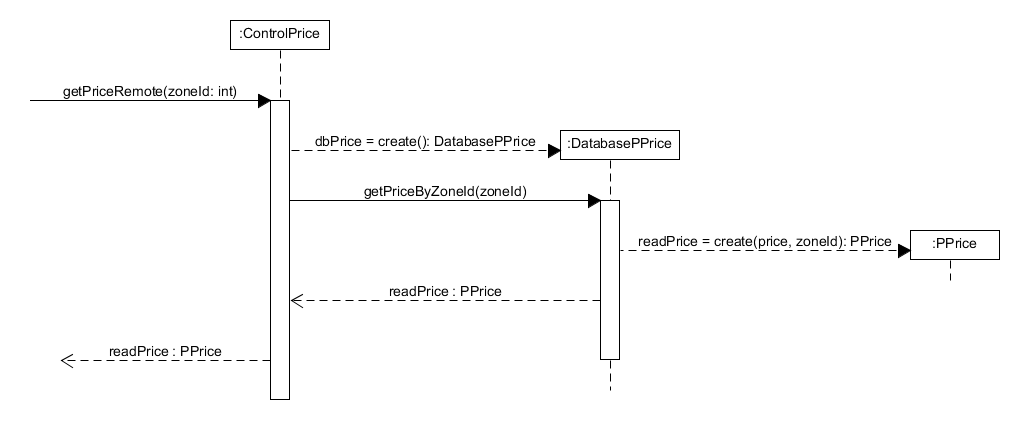
6) The total amount is the reset on the PPayStation instance.

7) Finally, the getParkingReceipt() method is called to get the receipt, which gets returned to the customer.



#### Interaction Diagram for the Use Case: Update Price

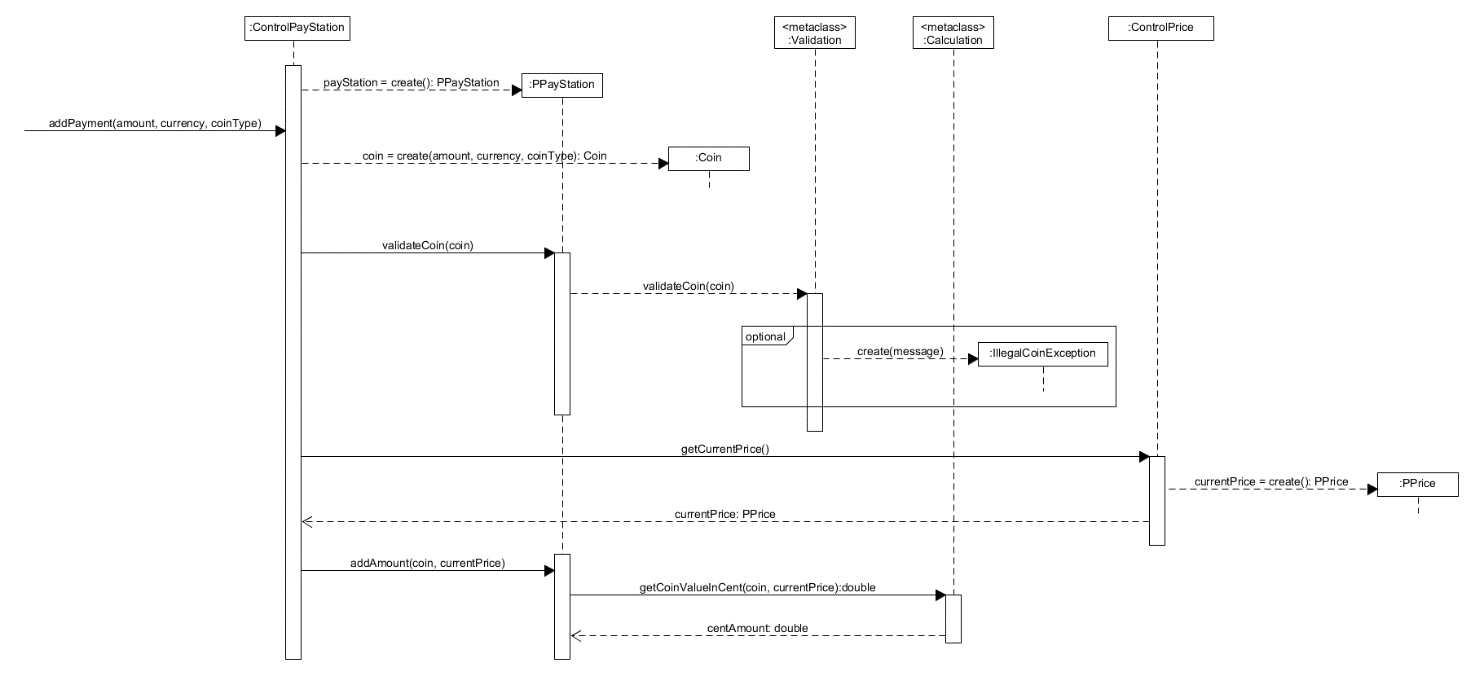
The Sequence Diagram is shown below:



The only system operation that need to be designed is getPriceRemote(…).

System operation getPriceRemote() invokes getPriceByZoneID(zoneId) on dbPrice which is an instance of DatabasePPrice. This method reads the information from the database and create an instance of the class Price, which is returned.

# Appendix 5: Sequence Diagram after Changes



# References

1. Henrik Bærbak Christensen: Flexible, Reliable Software. Taylor and Francis Group, LLC 2010.
2. Photo: http://fdcgates.com/st/26-356-thickbox/federal-apd-universal-pay-stations-for-parking-facilities.jpg (accessed January 2017)