Miniprojekt Test

Case: Betalingsautomat

**Indhold**

[Indledning 2](#_Toc507097673)

[Opgaver i systemudvikling 2](#_Toc507097674)

[Opgave 1: Planlægning af funktionelle tests på systemniveau (accept- og systemtest) 2](#_Toc507097675)

[Opgave 2: Planlægning og design af unit- og integrationstest 3](#_Toc507097676)

[Opgaver i programmering 5](#_Toc507097677)

[Opgave 3: Installation af program 5](#_Toc507097678)

[Opgave 4: Implementer test cases 5](#_Toc507097679)

[Opgave 5: Lav ændringer og test 6](#_Toc507097680)

[Aflevering og evaluering 7](#_Toc507097681)

[Bilag 1: Beskrivelse af betalingsautomaten 8](#_Toc507097682)

[Bilag 2: Scenarier 10](#_Toc507097683)

[Bilag 3: Use cases og domænemodel (krav) 11](#_Toc507097684)

[Use cases 11](#_Toc507097685)

[Domænemodel: 13](#_Toc507097686)

[Bilag 4: SSD, arkitektur og design af interaktion 14](#_Toc507097687)

[SSD for use casene: *Køb billet* og *Opdater pris* 14](#_Toc507097688)

[Arkitektur og database 14](#_Toc507097689)

[Design af interaktion 15](#_Toc507097690)

[Bilag 5: Sekvensdiagram efter ændring 20](#_Toc507097691)

[Kilder 20](#_Toc507097692)

# Indledning

I skal forestille jer, at I er ansat i en testafdeling i en et mindre firma, som laver software til betalingsautomater.

Jeres firma er blevet kontaktet af P-firmaet Nem Parkering A/S, som ønsker at udskifte deres forældede parkometre med nyere elektroniske betalingsautomater. Hardwaren består af automater fra Better Electronics A/S, nærmere bestemt deres model B-423E. I skal levere softwaren til betalingssystemet.

Udviklerne er godt i gang med at udvikle systemet. Undervejs har de holdt et par møder med Nem Parkering A/S, som har resulteret i, at de vigtigste use cases er blevet beskrevet, og der er lavet en domænemodel. Udviklerne har desuden dokumenteret designet samtidigt med, at de har kodet systemet. Det er aftalt, at udviklerene selv skal lave de tests, der er nødvendige for at sikre, at kritisk kode ikke fejler, men ellers er det testafdelingen (jer), der har ansvaret for at teste. Udviklerne er næsten færdige, idet der kun er småting der mangler. På grund af tidspres har de ikke fået sendt deres kravspecifikation og design til testafdelingen (jer), som det var aftalt. Det betyder, at I ikke har kunnet påbegynde planlægning og udledning af testcasene, som normalt kan starte, når use casene er blevet beskrevet, og det første design var klar!

Nem Parkering A/S har i længere tid presset på for at få softwaren leveret, da det er det eneste, de mangler for, at automaterne kan tages i brug. De mister penge for hver dag der går! Ledelsen i jeres firma føler sig presset både af Nem Parkering A/S , men også af andre projekter, der venter på at blive igangsat. De har derfor bedt udviklerne, om at sende det de har til test, så hurtigt det kan lade sig gøre.

Ved jeres første gennemgang af krav- og designdokumenterne viser det sig, at de ikke er særlig præcise og at der er kodet mere end vist i diagrammerne. ”Noget programmører ofte gør”. Da udviklerne er presset af de andre projekter der skal igangsættes, er det derfor aftalt, at I (testgruppen) foruden al testdokumentationen og skrivning af testene, også skal rette i dokumentation og kode, hvis den er mangelfuld eller fejlagtig. Det sidste er normalt udviklernes ansvar. Derfor er det også vigtigt, at det tydeligt fremgår i koden, hvilke rettelser testafdelingen har foretaget.

# Opgaver i systemudvikling

## Opgave 1: Planlægning af funktionelle tests på systemniveau (accept- og systemtest)

Tag udgangspunkt i følgende beskrivelser fra bilagene:

* Bilag 1: Beskrivelse af betalingsautomat
* Bilag 2: Scenarier
* Bilag 3: Use cases og domænemodel

Udform systemtestcases for use casene: *Køb billet* og *Opdater pris* ved brug af følgende fremgangsmåde:

1. Tag udgangspunkt i use case beskrivelserne for *Køb billet* og *Opdater pris*, tilføj alternative flow og acceptkriterier
2. Definer testscenarier der dækker både de normale forløb og de alternative flows:
3. For hvert testscenarie:
   1. Opstil testcases
   2. Tilføj de data det skal bruges i testcasen

## Opgave 2: Planlægning og design af unit- og integrationstest

Som grundlag for planlægning og design af unit- og integrationstests skal I tage udgangspunkt i designet fra bilag 4: SSD, arkitektur og design af interaktion. Følg følgende fremgangsmåde:

1. Gennemgå de enkelte interaktionsdiagrammer
2. Afgør hvilke metoder I vil teste
3. Overvej om det er en unit- eller integrationstest
4. For de udvalgte metoder opstilles testcases med udgangspunkt i princippet om ækvivalente klasser, grænseområder og testscenarier

Det forventes at I som et minimum laver testcases til:

* *addPayment* metoden (køb billet, kontrol af mønter og beregning af købt tid)
* *insertParkingBuy* metoden (køb billet, skrivning af et køb til databasen)
* *getPriceByZoneId* metoden (opdater pris, aflæsning af pris fra databasen)

Desuden skal I overveje øvrige mere tekniske test, der med udgangspunkt i jeres erfaringer bør testes fx connection til databasen (jf arkitekturafsnit)

I det følgende gives et eksempel baseret på metoden *addPayment.*

Metoden *addPayment(coinValue, currency, cointype)* er en central metode,der kaldes på *ControlPayStation -* som så samarbejder med andre klasser i udførelsen af metoden.

Input til test af *addPayment*: coinValue, currency og cointype

Ækvivalensklasser er fx følgende:

Valid valuta (currency) = DKK og EURO. Øvrig valuta er ugyldig.  
Valide mønter (coinValue og coinType):

|  |  |  |
| --- | --- | --- |
|  | Euro: |  |
|  | o | 1, 2, 5, 10, 20, 50 cent |
|  | o | 1, 2 € |
|  | DKK: |  |
|  | o | 50 øre |
|  | o | 1, 2, 5, 10, 20 kr |

Øvrige mønter er ugyldige.

Valide mønter accepteres og påvirker derfor beregning af købt parkeringstid:

*timeBought=GetTimeBoughtInMinutes(insetedSoFar, usePrice): int*

Det forventede resultat for parkeringstid er *TimeBought*: Større end 0 minutter ved gyldigt input. 0 før mønt-indkast eller hvis der kun er benyttet ugyldige mønter.

Testcasene formuleres med udgangspunkt i scenarier om forskellige møntkombinationer jf. bilag 2 samt princippet om ækvivalente klasser. Nedenstående tabel viser et eksempel på testcases for scenarie 1a (bilag 2): Husk også test for unormale datakombinationer!

*Eksempler på testcases*

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Testcase nr | Currency = EURO | | Currency = DKK | | Currency = NKK | | Forventer resultat: | |
| Cent | Euro | Øre | Kr. | Øre | Kr. | P sek. | P min.  TimeBought |
| Value | Value | Value | Value | Value | Value |
| 1:  Scenarie 1a,  EURO  En mønt,  P sek rundes op | 1 |  | 0 | 0 | 0 | 0 |  |  |
|  |  |  |  |  |  | 24 | 1 |
|  |  |  |  |  |  |  |  |  |
| 2  Scenarie 1 a,  EURO  Flere mønter | 0 | 1 | 0 | 0 | 0 | 0 |  |  |
| 0 | 2 | 0 | 0 | 0 | 0 |  |  |
|  |  |  |  |  |  | 7200 | 120 |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |

# Opgaver i programmering

## Opgave 3: Installation af program

Når I er klar til at kode jeres testcases skal i først installere program og database. Følg anvisningerne i installationsvejledningen *’Forberedelse til: Implementering af Test Cases’* (fil: Preparation.pdf).

## Opgave 4: Implementer test cases

I skal nu implementere jeres test cases.

I den netop installerede kode er næsten al Java programmeringen færdig, mens kun nogle få JUnit test er lavet. I skal altså fortrinsvist skrive JUnit tests, og kun i mindre grad anden Java kode.

Begynd fx med at implementere testene for de testcases i har fundet frem til i opgave 2 vedr *Køb billet*. Tjek at koden opfører sig som forventet. Ellers ret indtil at JUnit testene er fejlfri.

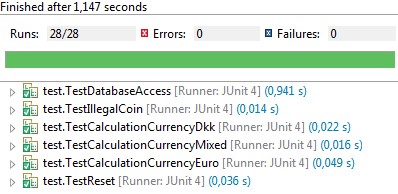
”Best practices” for JUnit test:

* Keep Junit tests very simple
* Create small specific test methods (do one thing)
* Test method name indicates expected result
* Consider using the AAA pattern (Arrange-Act-Assert)

Eksempel på kode til implementering af testcase nr. 1 (se eksemplerne på testcases, opgave 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();  } |

Fortsæt med at kode og teste indtil alt er testet. Alle testcases er implementerede og der meldes ingen fejl. Sådan eller lignende:



***Implementer test cases fra use casen: Opdater pris***

Igen: Kod og test indtil alt er testet. Alle JUnits test er implementerede og der meldes ingen fejl.

## Opgave 5: Lav ændringer og test

I denne opgave skal I *refactor* klassen *ControlPayStation,* så noget af ansvaret blive udelegeret til andre klasser. Det gøres ved at indføre en klasse *Calculation,* som en *”Metaclass*”, der skal have ansvaret for at udføre beregningerne. Ændringerne er vist i sekvensdiagrammet i bilag 5.

***Refactor koden og kør testene***

Ændre koden således den passer til ændringerne i diagrammet. Kør herefter testene – de skulle helst gå igennem!

#### Overvej nye test cases

Overvej testcases til *Calculation* klassen. Opstil unit testcases og implementer! Fortsæt til alle JUnits test er implementerede og der ikke meldes nogen fejl.

# Aflevering og evaluering

Der afleveres følgende dokumentation i en samlet rapport pr gruppe:

* De reviderende use cases med alternative flows og acceptkriterier
* Systemtest cases
* Unit- og integrationstestcases:
  + Udvælg de 4-8 testcases I mener, der har bidraget mest til kvalitetssikringen
  + Begrund
  + Beskriv hver testcase og vis JUnit implementering
  + Beskriv brugen af JUnit annotationer som @Before, @BeforeClass, @After, @AfterClasstests i jeres testklasser, og giv eksempler

Husk forside med formalia, herunder jeres navne.

# Bilag 1: Beskrivelse af betalingsautomaten



Automaten fungerer på den måde, at for hver mønt der indsættes i automaten, vises på skærmen den samlede parkeringstid i minutter.

Efter den første mønt er indsat vises desuden en ’Annuller’ og en ’Køb’ knap på skærmen.

Teknisk set er der i model B-423E indbygget hardware, så betalingsautomaten kan kommunikere med f.eks. en database server via Wireless Session Protocol ( WSP ). Der er desuden en 256 KB memory chip i automaten.

Derimod kan modellen ikke modtage betaling via betalingskort, men Nem Parkering har sikret sig, at betalingsautomaterne senere kan udbygges med denne facilitet.

### *Pris og beregning*

Betalingsautomaten kan modtage betaling i to valutaer DKK og Euro.

Alle gældende mønter i de to valutaer er gyldige som betaling for parkering. Altså er følgende mønter gyldige:

* Euro

|  |  |  |
| --- | --- | --- |
|  | o | 1, 2, 5, 10, 20, 50 |
|  | o | 1, 2 € |
|  | DKK |  |
|  | o | 50 øre |
|  | o | 1, 2, 5, 10, 20 kr |

Prisen angives som det antal sekunder 1 cent giver. F.eks. 1 cent giver 24 sekunders parkeringstid – svarende til, at 5 cent giver en parkeringstid på 120 sekunder == 2 minutter. Flere eksempler under scenarier.

Prisen er altid i Euro. Hvis der betales med danske mønter omregnes beløbet til Euro.

Mere specifikt gælder:

* Der kan betales med et vilkårligt miks af gyldige mønter (altså både Euro og DKK mønter)

* Den samlede parkeringstid beregnes altid på grundlag det samlede indbetalte beløb omregnet til cent. Beløbet er et decimaltal, for at undgå afrundings-unøjagtighed.

Fx giver 1 Euro og 5 kr ca. 166,6666667 cent (ved en kurs på 7,50 kr / Euro).

* Hvis den beregnede parkeringstid ikke er et helt antal minutter, så rundes op til næste hele minuttal.

Fx giver de ca. 166,6666667 cent en parkeringstid på 4000 s. (ved en pris på 24 sekunder / 1 cent). Det giver ca. 66,66666667 minutter, som så rundes op til 67 minutters parkering.

* Hver betalingsautomat har en parkeringszone (1, 2 eller 3). Så prisen afhænger af automatens parkeringszone.

# Bilag 2: Scenarier

I samarbejde med Nem Parkering og Better Electronics er der opstillet disse scenarier:

### Scenarie 1a – Køb parkeringsbillet (Euro)

En tysk bilist går hen til betalingsautomaten, for at betale for sin parkering. Han indsætter en række tilladte Euro mønter som betaling. Han kan se på skærmen hvor lang tid han har betalt for ind til videre. Når han er tilfreds med tiden trykker han på ’Køb’ knappen. Der udskrives en parkeringsbillet med det antal minutter han har betalt for og købet gemmes i en remote database.

Automaten og skærmen nulstilles, så den er klar til næste kunde.

### Scenarie 1b – Køb parkeringsbillet (DKK)

En dank bilist går hen til betalingsautomaten, for at betale for sin parkering. Han indsætter en række tilladte danske mønter som betaling. Herfra som scenarie 1a.

### Scenarie 1c – Køb parkeringsbillet (Euro og DKK)

En dansk bilist er lige kommet hjem fra ferie – han går hen til betalingsautomaten, for at betale for sin parkering. Han indsætter en række tilladte danske og Euro mønter som betaling. Herfra som scenarie 1a.

### Scenarie 2 – Fortryd betaling

En bilist har indsat en række tilladte mønter, men opdager at den parkeringstid der vises langt overskrider den tid han har betalt for. Han trykker på ’Annuller’ knappen og får sine penge retur. Der gemmes intet i databasen. Automaten og skærmen nulstilles, så den er klar til næste kunde.

### Scenarie 3 – Ugyldig mønt

En bilist har i alt indsat 50 cents og skærmen viser derfor 20 (minutter) i parkeringstid. Ved et uheld indsætter han en norsk 1 krone (en ugyldig mønt). Betalingsautomaten tager ikke imod mønten, og skærmen opdateres ikke.

**Scenarie 4 – Opdatering af pris**

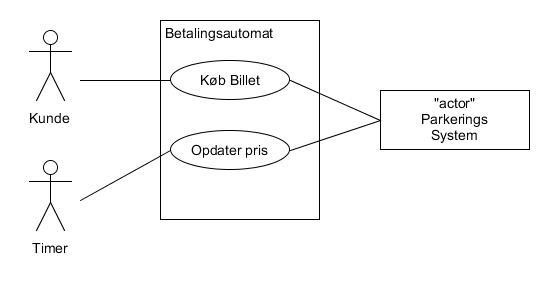
Hver nat kl. 03 henter betalingsautomaten den aktuelle pris for parkeringszonen i den remote database.

# Bilag 3: Use cases og domænemodel (krav)

## Use cases

Ud fra kravene til systemet som er beskrevet i bilag 1 og 2, er der fundet frem til følgende use cases:

Use case diagram:



Use casen: *Køb billet* er den centrale use case i systemet. Aktøren en kunde, der gerne vil købe en parkeringsbillet. Use casen: *Opdater pris* , der henter prisen over på automaten fra det centrale parkeringssystem startes automatisk af en ”Timer” aktør kl 03 hver nat. Begge use cases bruger *services* fra det centrale parkeringssystem til registrering af et køb samt hentning af pris.

***Use case beskrivelser:***

I det følgende er vist fully dressed beskrivelser af de use cases, der skal laves test for.

#### Use case: Køb billet – fully dressed

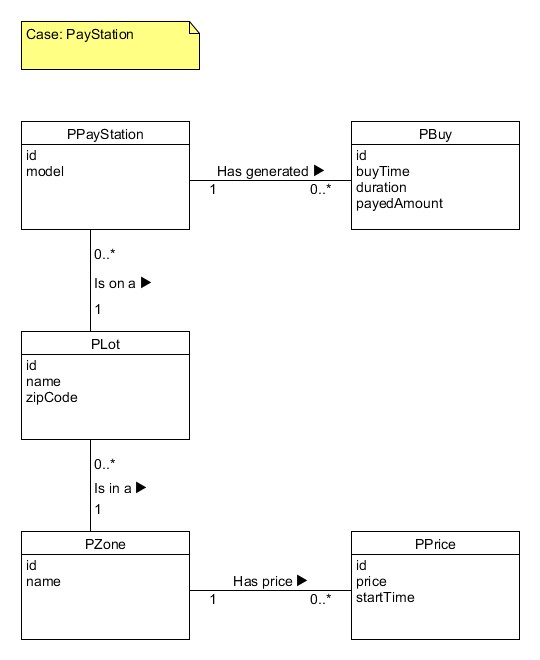
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Use case navn | **Køb billet** | | | |
| Aktører | Kunde | | | |
| Præbetingelse | Ingen | | | |
| Postbetingelse | Penge er kommet i automaten og billetten med parkeringstid er udskrevet | | | |
| Frekvens | Max 200 / time | | | |
| Flow of events |  | **Aktør** |  | **System** |
| 1. | En kunde ankommer til en parkeringsautomat for at købe en billet |  |  |
| 2. | Kunden kommer første mønt i automaten | 3.  4. | Tjekker om mønten er lovlig Registrere betalingen og opdaterer ”købt tid”, som vises i displayet |
|  |  |  | 5. | Panelets ”Køb” og ”Annuller” knapper gøres aktive |
| 6. | Kunden kommer næste mønt i  automaten | 7.  8. | Tjekker om mønten er lovlig Registrere betalingen og opdaterer ”købt tid”, som vises i displayet |
|  | *Trin 6 til 8 gentages* | | |
| 9. | Kunden angiver at billet ønskes købt | 10. Systemet udskriver billetten og registrer købet i den centrale database | |
|  |  | 11. Automaten nulstilles så det er klar til næste kunde | |
| Alternative flow |  |  | | |
|  |  |  | |
|  |  |  | |
|  |  |  | |
|  |  |  | |
| Specielle krav |  |  |  | |
|  |  |  | |
|  |  |  | |

#### Use case: Opdater pris – fully dressed

|  |  |  |  |
| --- | --- | --- | --- |
| Use case navn | **Opdater pris** |  |  |
| Aktører | Timer |  |  |
| Præbetingelse | Ingen |  |  |
| Postbetingelse | Priserne er opdateret i automaten |  |  |
| Frekvens | 1 gang om natten (kl 03) |  |  |
| Flow of events | **Aktør** |  | **System** |
| 1. Use-casen starter med at klokken er blevet 03 |  |  |
|  | 2. | De aktuelle priser fra parkeringssystemet hentes over og gemmes i lokalt på automaten |
| Alternative flow |  |  |  |
|  |  |  |
|  |  |  |
| Specielle krav |  |  |  |

## Domænemodel:

Automaterne, deres placering og de forskelige køb registreres i det centrale parkeringssystem. Domænemodellen ser således ud:



Ovenstående domænemodellen afspejler indholdet af hele parkeringssystemet.

De billetter der er blevet købt på den enkelte paystation registreres i i *PBuy*  der tilknyttes den *PPayStation,* hvor de er købt*.* Der registres endvidere hvilken parkeringsplads *PLot* en *PPayStation* er placeret på, samt hvilken parkeringszone: *PZone* parkeringspladsen *PLot* befinder sig i. Endelig er det enkelte prisobjekt *PPrice* tilknyttet den parkeringszone *PZone* det gælder for.

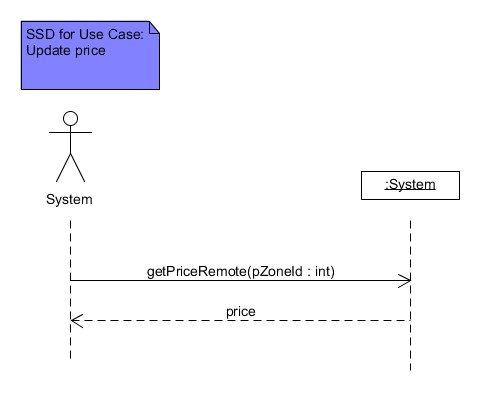
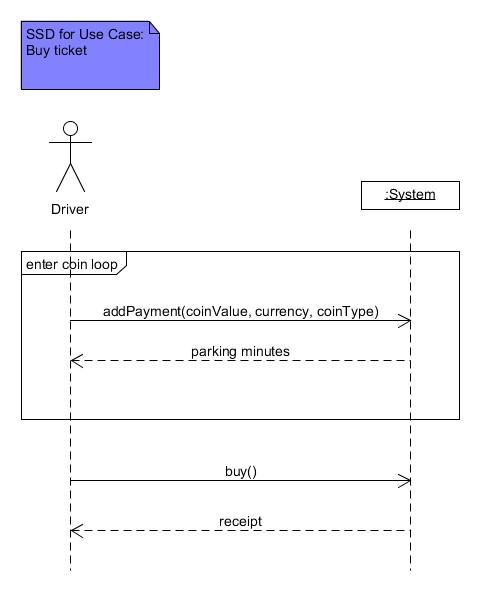
Lokalt på de enkelte automater gemmes:

* *PPrice* objekt der hentes over fra det centrale parkeringssystem (kl 03 hver nat).
* *PZone object med ident på parkeringszonen.*
* *PBuy* objekt – danner grundlag for gem i parkeringssystemet.
* *PPayStation* objekt – med automatens data, fx hvilken model det er.
* *PReceipt objekt –* Beregnet købt tid *buytime* gemmes på et receipt objekt, der kun eksisterer indtil et køb er afsluttet.

# Bilag 4: SSD, arkitektur og design af interaktion

## SSD for use casene: *Køb billet* og *Opdater pris*

Med udgangspunkt i fully dressed beskrivelserne af ovenstående use cases er der lavet nedenstående systemsekvensdiagrammer, hvor input-/og outputhændelser til systemet er navngivet og der er tilføjet en parameterliste



## Arkitektur og database

Som grundlag for designet er det besluttet at opbygge systemet i en 3-lagsarkitektur.

I udgangspunktet er der en controllerklasse per use case. I tilfælde hvor controllerklasserne laver mange forskellige ting, som fx både tjek samt forskellige beregninger anvendes ”hjælpe” klasser i kontrollerlaget for at bibeholde ”high cohesion” og genbrugsmuligheder.

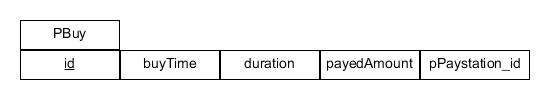
Nogle af kaldene er ”static”. Dette vises i UML ved at bruge typen ”MetaClass”. Kaldes fx metoden *testCoin(….)* på *”MetaClass” Coin* betyder det, at *testCoin(…)* er et statisk metodekald på en instans af en metaklassen: *Coin* I databaselager benyttes følgende interfaces:

* DbPBuy
* DbPPrice

Data for købet skal gemmes i den centrale database. Forbindelsen til databasen er en singleton:

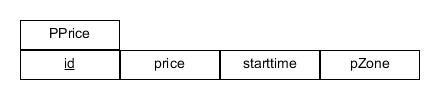
Connection con = DBConnection.getInstance().getDBcon();

Tabellen *PBuy* ser således ud:



Primærnøglen id er autogenereret. pPaystation\_id er en FK der peger på id i Paystation.

Tabellen *PPrice* ser således ud:



pZone er en FK der peger på id i PZone.

På den enkelte automat er det alene prisen, der hentes over hver nat og gemmes i et lokalt PPrice objekt. Automaten er konfigureret med en fast kurs på 7,50 kr / Euro.

## Design af interaktion

#### Use case: Køb billet

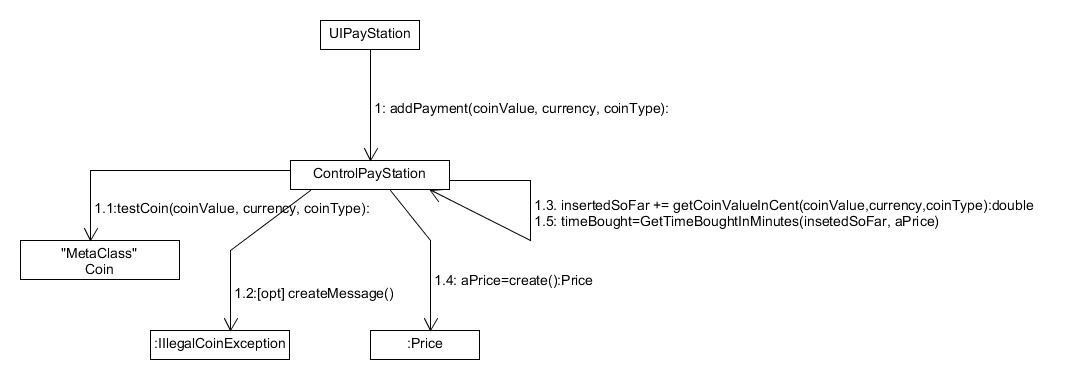
Der er valgt at lave et interaktionssdiagram for hver systemhændelse i SSD’en. Interaktionsdiagrammerne er vist som både kommunikationsdiagram og sekvensdiagram.

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

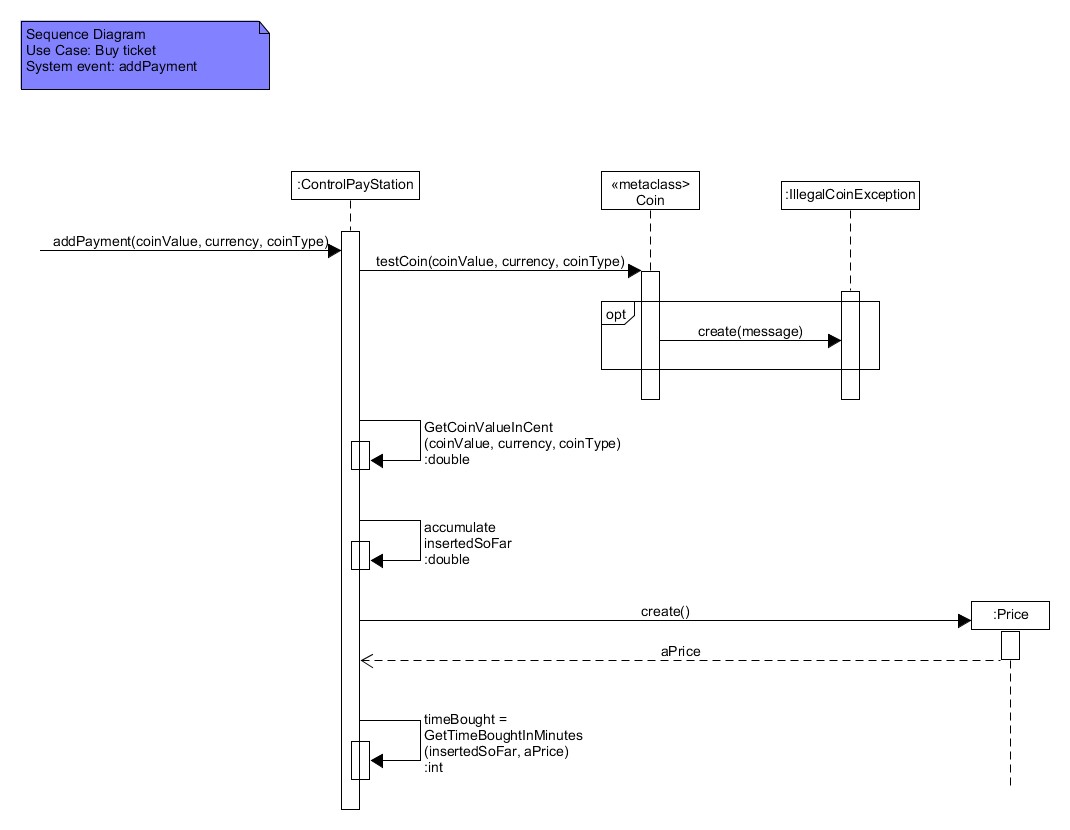
Det er besluttet at der foruden controllerklassen *ControlPayStation* til at modtage *addPayment(coinValue, currency, cointype)* også skal være en metaklasse:

 *<metaclass> Coin’s* som har ansvaret at teste, om det er en legal mønt der anvendes.

Kommunikationsdiagram:



Sekvensdiagram:



Først tjekkes om det er en lovlig mønt ved kald af den statiske metode: *testCoin* på *Coin*. Hvis den ikke er lovlig kastes en exception.

Er mønten lovlig forøges det indbetalte beløb *insertedSoFar* med værdien af mønten *GetCoinValueInCent(coinValue, currency, coinType)*. Endelig beregnes hvor meget tid der er købt: *timeBought =*

*GetTimeBoughtInMinutes(insertedSoFar, new Price()).* Disse metoder er interne private metoder på *ControlPayStation.* For at kunne beregne prisen kaldes metoden *getParkingPrice* på *aPrice:PPrice*.

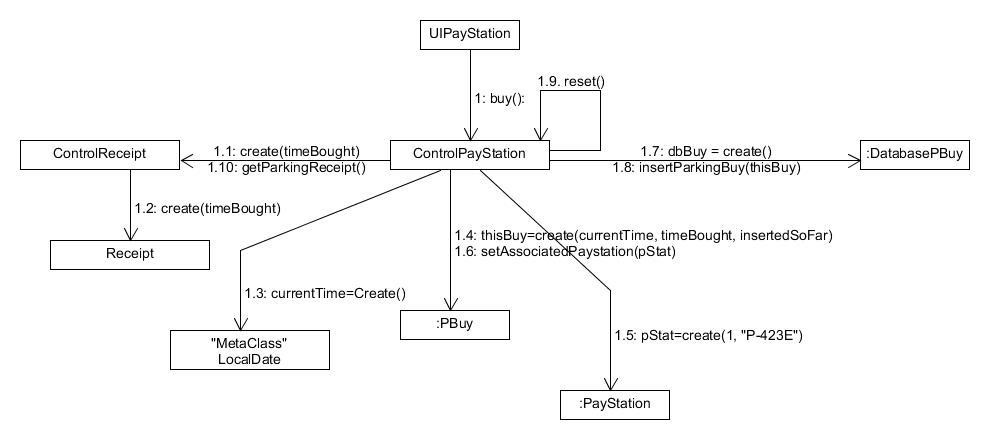
Med hensyn til *Price* er præbetingelsen, at der eksisterer et *Price* objekt for den pågældende parkeringsautomat, som er dannet ud fra det centrale prisregister. Det er alene prisen der hentes over. Lokalt er valgt at lægge kursværdien ind for at kunne omregne til EURO. Lokalt på automaten ser prisobjektet således ud:



#### Kommunikationsdiagram for: Buy()

Her vises interaktionen for operationen: buy(). ”Metaclass” LocalDate benyttes til at finde det aktuelle tidspunkt ud fra pc’ens tid.

Kommunikationsdiagram:



Detførste der sker, er, at variablen *timebought* som indeholder den købte tid og som blev beregnet af *AddPayment* metoden gemmes midlertidig i et *Receipt* objekt, så kvitteringen kan genereres senere.

Herefter samles oplysningerne om købet i *thispBuy* der oprettes som en ny instans af *PBuy.*  Der mangler dog oplysninger om købstidspunktet, som findes ved at kalde Localtime fra java biblioteket. Efter oprettelsen af *thispBuy* tilføjes associeringen til *PayStation* (jf domænemodel) ved først at oprette en instans *pStat* af *Paystation*, som herefter sættes som referenceattribut på *thispBuy*.

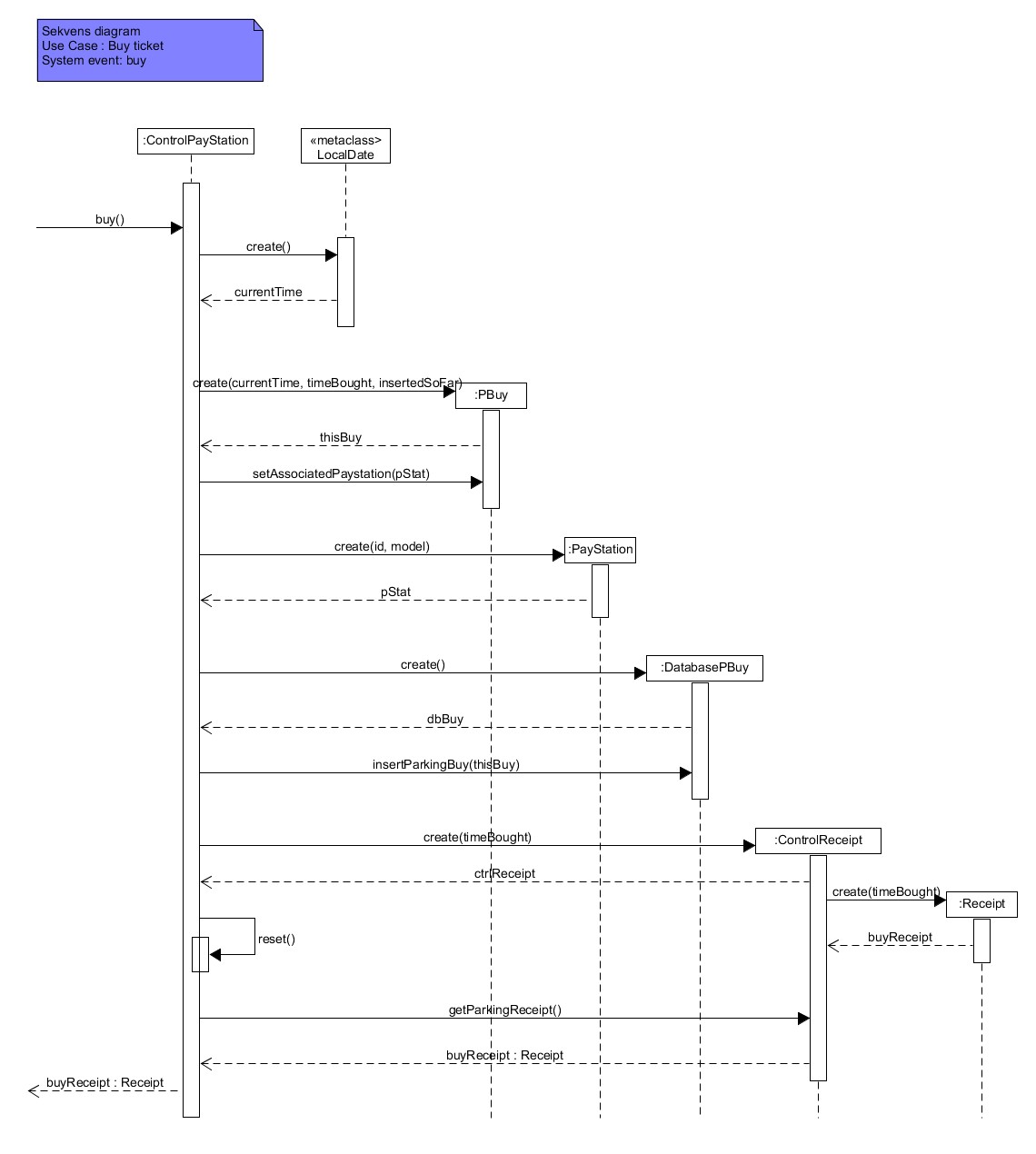
Skrivningen til databasen sker ved at objektet *thisBuy* med købsoplysningerne sendes til en instans af db klassen

*DatabasePBuy*. Først oprettes instansen *dbBuy*. Derefter indsættes data fra *thisBuy* ved kald af *insertparkingBuy*.

Primærnøglen autogenereres. Problemer med skrivningen til databasen håndteres gennem en række exceptions.

Til sidst udskrives kvitteringen.

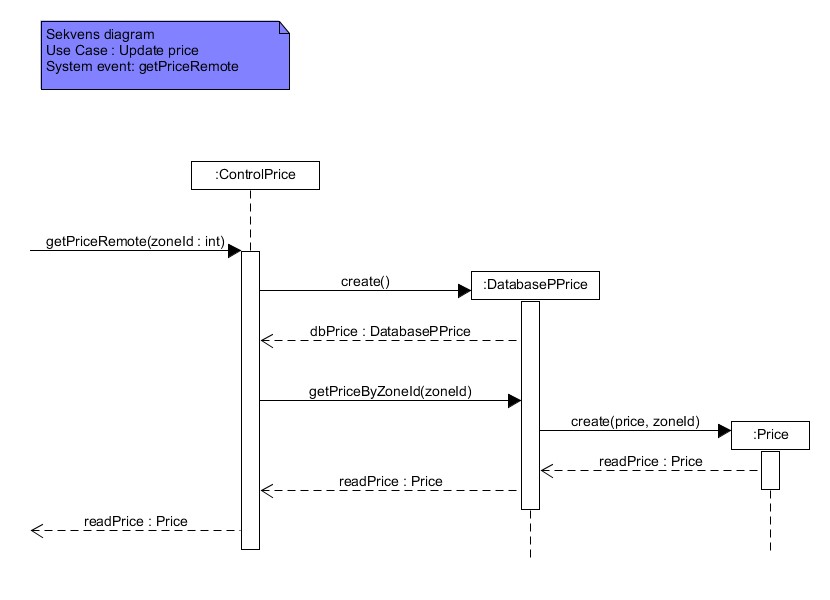
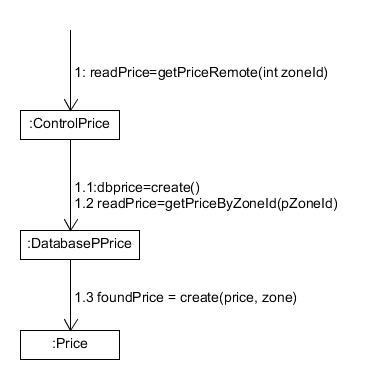
Sekvensdiagram:



#### Kommunikationsdiagram for Use casen: Opdater pris

Her er der kun en systemhændelse: *ReadPrice* der skal designes.

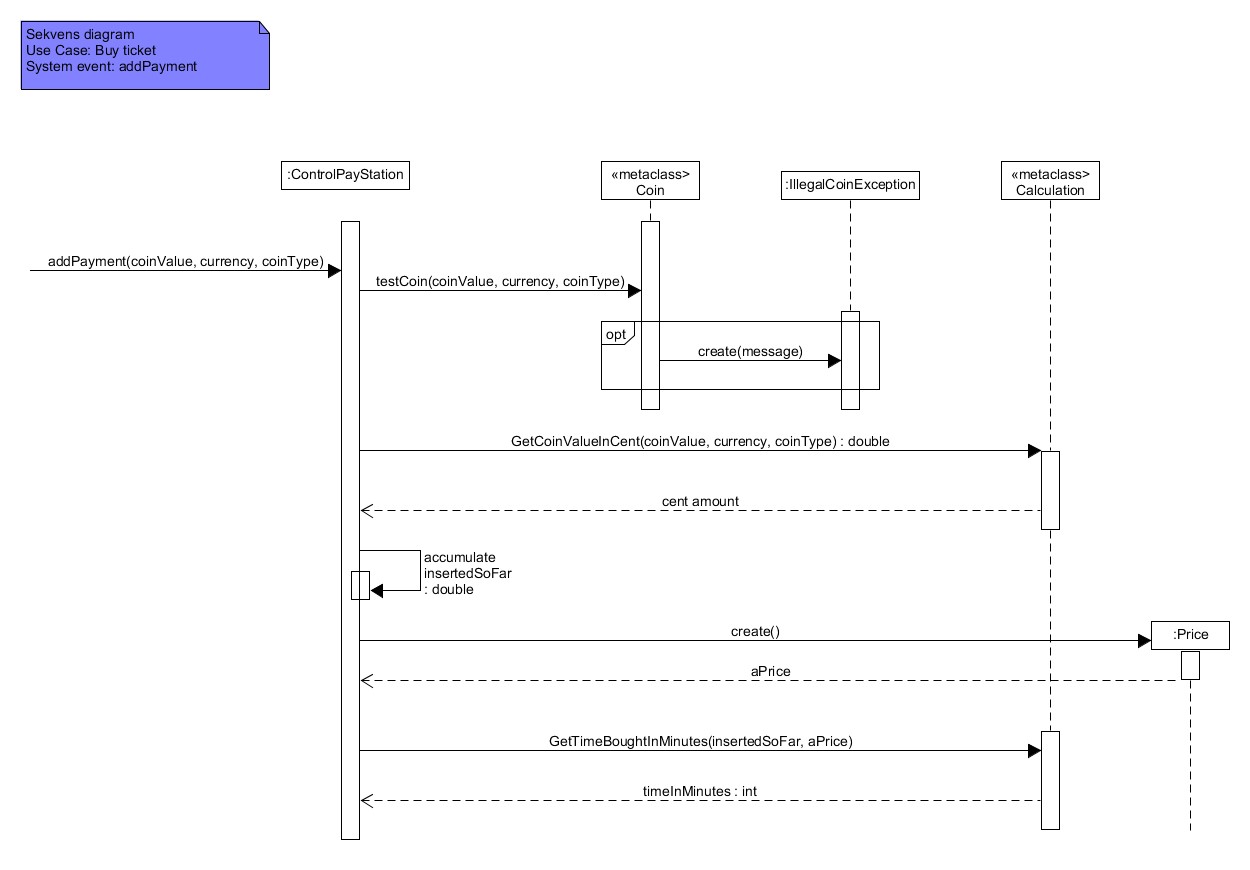
Kommunikationsdiagram: Sekvensdiagram:



Systemhændelsen *getPriceRemote* sendes som et metodekald til *ControlPrice,* som kalder metoden:

*getPriceByZoneID(pZoneId) på dbprice,* der er en instans af db klassen *Databaseprice.* I *getPriceByZoneID(pZoneId)* metoden læses informationer fra databasen, som samles i *foundPrice (readprice),* der er en instans af Price. Det dannede prisobjektet returneres til *ControlPrice*.

# Bilag 5: Sekvensdiagram efter ændring



# Kilder

1. Henrik Bærbak Christensen: Flexible, Reliable Software. Taylor and Francis Group, LLC 2010.

1. Foto: http://fdcgates.com/st/26-356-thickbox/federal-apd-universal-pay-stations-for-parking-facilities.jpg (Set januar 2017)