Seminarium 4

Objektorienterad Design IV1350

Paulina Huang
pauhua@kth.se
7 juni 2021

Innehåll

Introduktion

Seminarium 4 i kursen Objektorienterad Design handlar om undantagshantering, polymorfism och olika designmönster.

Undantag kastas på grund av att något har blivit fel under tiden som programmet körs. Felen kan bero på bruten affärslogik, alltså att användaren inte använder programmet som det är tänkt. Felet kan också bero på en bugg, som bör ha hittats under programutvecklingen. Ytterligare en anledning till att ett undantag kastas kan vara att något hindrar programmet att utföra sin uppgift, till exempel att den inte kan kopplas mot en server eller databas.

Polymorfism är det fjärde designmönstret som gås igenom i kursen. Den förbättrar dem tre tidigare designmönster som tagits upp under kursens gång; sammanhållning, låg koppling och inkapsling. Polymorfism handlar om att bryta ut det publika gränssnittet.

Det finns olika designmönster, typiska lösningar till vanliga problem, som är baserade på polymorfism. Designmönstren Observer, Singleton och Strategy implementeras i programmet.

Målet med seminarium 4 är att öva på att designa och koda undantagshantering, polymorfism och designmönster.

Metod

Undantagshantering

Arbetet började med att identifiera om dem undantag som skulle kastas enligt kravspecifikationen är av typen checked eller unchecked exception. Att ett undantag kastas för att indikera att sökningen är för ett itemID som inte finns i inventoryRegistry är checked exception, eftersom det beror på bruten affärslogik. Att ett undantag kastas för att man inte kan anropa databasen, ex. för att databasservern inte kan nås, beror varken på bruten affärslogik eller en bugg som kan elimineras under utveckling. Det tillhör samma kategori som en bugg, unchecked exception.

Därefter var det dags att bestämma om relevanta namn för undantagen. Det var önskvärt att finna en bra balans mellan beskrivande namn och generellt namn för att undvika ett alltför stort publikt gränssnitt vid användning av undantagen i programkoden.

Då undantag kastades skulle dem finnas i samband med try och catch-block.

Under utveckling av undantagshantering tänktes det också på att rätt abstraktionsnivå skulle användas. När ett ogiltigt itemIdentifier användes, som är ett checked exception och fick ett specifikt beskrivande namn, så skickades informationen om felet vidare till översta lagret vyn. När det var fel med databasen så skickades inte den detaljerade felmeddelandet upp till vyn utan stannade i Controller och vyn fick i stället ett mer generellt felmeddelande, att operationen misslyckades.

Eftersom båda felen inte beror på någon bugg som kan elimineras i utvecklingsfasen så är det av intresse för användaren att få ett meddelande om felsituationen. Ett felmeddelande skrivs även till loggen som är till utvecklaren. Felmeddelande skrivs till loggfilen endast när ett oväntat undantag har kastats.

Efter att undantag lagts till i relevanta metoder i källkoden uppdaterades javadoc kommentarer. Till sist uppdateras testerna.

Observer pattern

För att kunna implementera Observer pattern var det viktigt att förstå polymorfism och interface. Kunskap om polymorfism och interface inhämtades via kurslitteratur och föreläsningar.

Det beslutades om att TotalRevenueView skulle placeras i View paketet eftersom informationen skulle presenteras till användargränssnittet. TotalRevenueFileOutput placerades i ett inre paket, FileHandling, innanför paketet Integration. Klassen skulle skriva till en logg och hade inte något med dem redan existerande paketen att göra.

Observer klassen fick ett ganska allmänt namn, SaleObserver. En lista med objekt av typen SaleObserver placerades i CashRegister. Så fort CashRegister hade fått in kontanter blev Observer klasserna notifierade.

Både TotalRevenueView och TotalRevenueFileOutput har som uppgift att observera försäljningsomsättningen sen programmet startades. Eftersom TotalRevenueView skriver till användargränssnittet fick den ett mer grafiskt utseende. TotalRevenueFileOutput däremot som loggar till en fil har mer detaljerad information och visar respektive inkomst under olika försäljningar samt totala omsättningen sen programmet startade.

Singleton

Singleton mönstret garanterar att det bara finns en instans av en klass, och att det är den instansen som används överallt i programmet.

Singleton implementerades i databaserna InventoryRegistry och AccountingRegistry. RegistryCreator kändes onödig efter implementationen, initieringen i main-metoden togs därför bort och klassen användes aldrig i programmet. RegistryCreator har också tagits bort från parameterlistan i Controller konstruktorn.

Singleton implementeras även i LogHandler, som loggar oväntade buggar under programexekveringen. Det är inte önskvärt att den existerande loggen tas bort varje gång varje gång ett nytt objekt av LogHandler skapas.

Strategy

Strategy mönstret gör det möjligt för en klass att ändra sitt beteende och algoritm under programexekveringen.

Strategy har implementerats i ErrorMessageHandler och LogHandler, när det ska loggas något till antingen användargränssnittet eller till en loggfil. Eftersom det ibland var önskvärt att skicka det meddelande som finns i undantagen och ibland var önskvärt att skicka ett anpassat meddelande fick interface Logger två metoder, logException som tar in ett Exception i parameterlistan och printMessage, som tar in en sträng i parameterlistan.

Implementering av algoritmerna görs av en ny klass som agerar som klient åt Logger. Klassen får namnet LoggerClient. Det var svårt att bestämma om den nya klassen skulle placeras i paketet view eller i paketet util. Eftersom dess metoder anropas från view, bestämdes det att LoggerClient skulle placeras där. På det sättet kan klassen LoggerClient och dess metoder vara paketprivata.

Resultat

Undantagshantering

Programmet imiterar en försäljningsprocess, i exempelvis en butik. En försäljning startar. Varor skannas in i systemet. Felmeddelanden genereras om användare försöker skanna ett ogiltigt artikelnummer eller om databasen inte kan nås. Försäljningen avslutas och till sist skrivs ett kvitto ut med information om försäljningen. Relevanta enhetstester har skrivits för klasserna i programmet. En exempelexekvering av programmet visas i figur 1 och figur 2.

Observer pattern

Ett interface, SaleObserver, har lagt till i model. SaleObserver implementeras av klasserna TotalRevenueView och TotalRevenueFileOutput. Klasserna initialiseras i View respektive Main via Controller som skickar dessa vidare till CashRegister. I CashRegister sparas dem i en lista, och Observer i listan notifieras när betalning har registrerats i kassaregistret.

Två olika försäljningar startas i View för att simulera verkligheten. Observer klasserna har blivit korrekt anropade och omsättning från programstart visas i konsolen samt i en logg fil.

Singleton

För att implementera Singleton har klasserna InventoryRegistry, AccountingRegistry samt Controller där dem anropas ändrats. Eftersom RegistryCreator kändes onödig efter implementationen av Singleton mönstret har användandet av RegistryCreator tagits bort från programmet, dvs från Main och Controller.

Strategy pattern

För att implementera Strategy mönstret har ett nytt interface lagts till, Logger. ErrorMessageHandler och LogHandler, som skapades i samband med undantagshanteringen i den tidigare uppgiften i seminarium 4, har ändrats. Dem implementerar nu interfacet Logger. Testerna klasserna som använder Strategy mönstret har uppdaterats. Även View, som använder sig av Logger, har fått förändringar i källkoden.

Övrigt

Då hela programmet blev granskad igen under programutvecklingen av undantagshantering så har en allmän uppdatering gjorts för att få ännu bättre sammanhållning, låg koppling och inkapsling. RunningTotal och latestRegisteredItem har satts ihop till ett eget objekt, SaleInfoDTO, för att förbättra designen och smidigare kunna presentera informationen till vyn. Fler tester har skrivits. Vissa variabelnamn har fått ännu mer beskrivande namn samt fått keyword final. Fler metoder har hittats som kunnat ändrats till privata eller paket-privata.

Då två försäljningar görs för att se uppdatering av kassaregistret innebar det att exekveringar som fanns i View blev duplicerade. Det har därför skapats privata metoder för exekveringar som förekommer i View när en försäljning sker. Även Receipt har fått en del privata metoder för att förbättra sammanhållningen.

Det grafiska användargränssnittet har förbättrats; utskriften av programexekveringen har blivit finare, bland annat har kvittot fått ett annat format med streckkod i slutet.

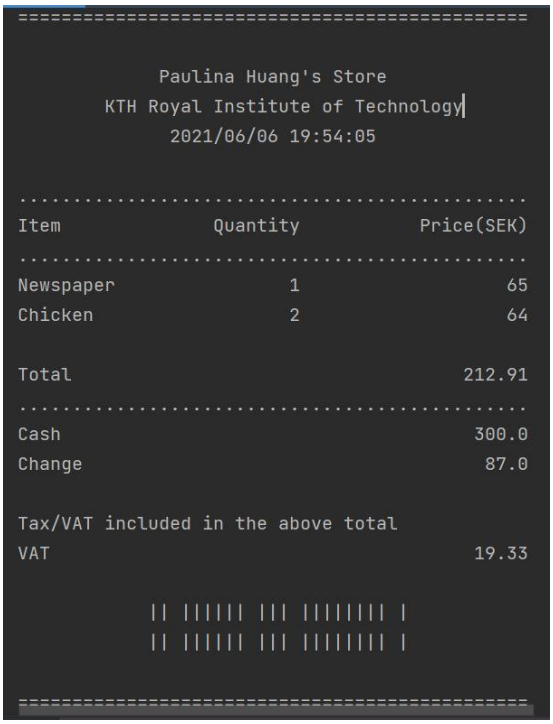
Se kommande figur 1 – figur 7 för utskrift av programexekvering. Se figur 8 för loggfilen av omsättning efter två försäljningar. Länk till källkoden kommer efter figurerna.

```
================ A NEW SALE HAS BEEN STARTED ================

---------------- Start of scanning Procedure ----------------

Scanning item...
Name: Newspaper, Price: 65.0, VATRate: 0.06
Running total: 68.9 SEK

Scanning item...
Name: Chicken, Price: 64.29, VATRate: 0.12
Running total: 140.9 SEK

Scanning item...
Name: Chicken, Price: 64.29, VATRate: 0.12
Running total: 212.91 SEK

Trying to scan an invalid identifier...
ERROR | Correctly failed to scan a nonexistent identifier.
06 Jun 2021 19:54:05

Trying to scan when database can not be called...
ERROR | Correctly failed to call the database.
06 Jun 2021 19:54:05
```

Figur 1. Utskrift av programexekvering

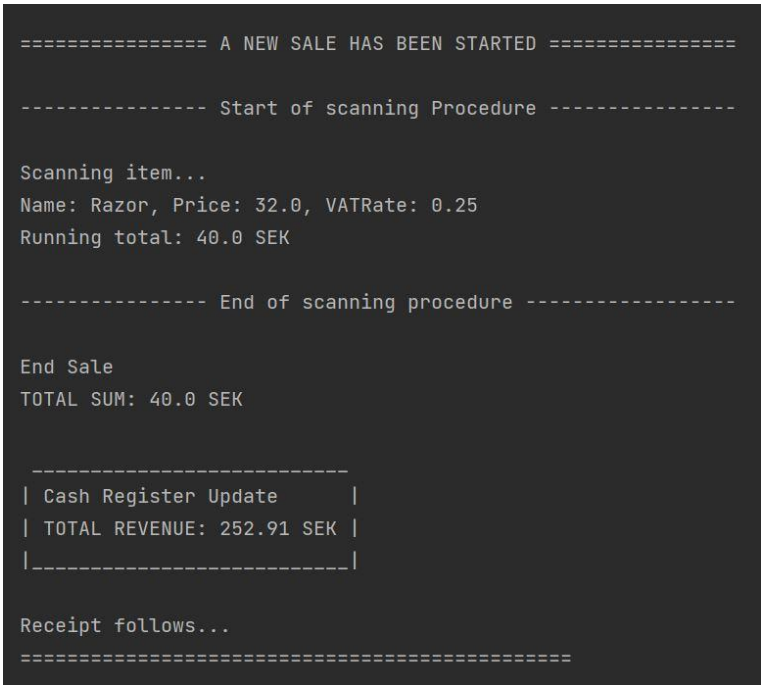
```
---------------- End of scanning procedure ------------------

End Sale
TOTAL SUM: 212.91 SEK


 ____________________________
| Cash Register Update       |
| TOTAL REVENUE: 212.91 SEK  |
|____________________________|

Receipt follows...
```

Figur 2. Utskrift av programexekvering



Figur 3. Utskrift av programexekvering

Figur 4. Utskrift av programexekvering



Figur 5. Utskrift av programexekvering

Figur 6. Utskrift av programexekvering

Figur 7. Utskrift av programexekvering

```
SALES REVENUE
Income: 212.91 SEK

Total revenue: 212.91 SEK

SALES REVENUE
Income: 212.91 SEK
Income: 40.0 SEK

Total revenue: 252.91 SEK
```

Figur 8. Logg fil av total omsättningen

Länk till programmets källkod och enhetstester
https://github.com/originalpau/FantasticPOS

Diskussion

Undantagshantering

Undantagshanteringen följer de riktlinjer som bland annat finns beskrivna under avsnitt Metod.

Observer

Objekten TotalRevenueView och TotalRevenueFileOutput får reda på när en betalning har gjorts och registrerats i kassaregistret – utan att ha ett beroende till andra klasser utöver interfacet SaleObserver.

Referensen av Observer skickas via Controllern direkt till det objekt som ska observeras. Eftersom Controller är den klass som initialiserat och har en referens till det objekt som ska observeras sen tidigare så blir det varken sämre koppling eller sammanhållning.

Det objekt som skulle observeras är CashRegister. På det viset får inte Observer klasser notifiering förrän en insättning av kontanter de facto har registrerats och eliminerar risken att Observer får datainformation när det blivit en bugg i programmet. Vidare blir det även bättre sammanhållning och lägre koppling än ifall andra klasser valts, då Controller hade en referens till det objekt som skulle observeras, som det beskrivits i stycket ovan.

Det är en primitiv datatyp som skickas från det objekt som ska observeras till Observer. Alternativet skulle vara att skicka med hela objektet i parametern vilket skulle ge Observer klassen mer flexibilitet om mer datainformation krävs. Att skicka med objektet som observeras till Observer skulle däremot öka kopplingen och beroenden. Av den anledningen valdes det att endast skicka med totala kostnaden av den aktuella försäljningen i parameterlistan.

Singleton

Det har gjorts en korrekt implementation av Singleton. Det finns endast en instans av databaserna AccountingRegistry och InventoryRegistry. Designmönstret har valts då det bara är ett objekt av respektive databas som ska användas och det är det objektet som alla andra klasser ska använda.

Strategy

Designmönstret Strategy har valt för funktionen av att kunna skriva till något, må så vara en loggfil eller konsolen. Designmönstret har valts för att det finns ett behov av att byta algoritm för en liknande funktion. View använder sig av interface typen Logger för att använda dess funktioner när den behöver skriva till konsolen eller en loggfil då ett undantag kastas.

Författaren känner dock att det är oklart om det blev en bättre design efter implementationen av Strategy. Det upplevdes som att källkoden fick mer kod, utan att kopplingen blev lägre som fallet var med Observer. I och med skapandet av ett nytt objekt när klassen som implementerar Logger ska användas så blev det ändå ett beroende till View, precis som det var innan Strategy. Det blev inte heller bättre inkapsling jämfört med innan Strategy. Dessutom skapades ett nytt objekt av klasserna som implementerade Logger varje gång något skulle loggas, vilket kändes onödigt. Det finns en förklaring i kurslitteraturen för hur skapandet av ett nytt objekt kunde undvikas, dock verkade det göra situationen ännu mer komplicerad för författaren att få en förståelse för varför Strategy inte verkar ha gjort så att programmet får bättre design, vilket gjorde att författaren beslutade sig för att inte ge sig in i det.

För att sänka kopplingen och öka sammanhållningen introduceras en ny klass, LoggerClient, vars publika metoder minskar inkapslingen. LoggerClient är en klient till Logger och anropas från View.