Seminarium 4

Objektorienterad Design, IV1350

Kevin O’Flaherty, kevinof@kth.se

2022-05-18

Innehållsförteckning

1 Inledning

Seminarium 4 hade två uppgifter, med två deluppgifter vardera, och utgick ifrån arbete från tidigare seminarier. Den första uppgiften innebar att implementera undantagshantering i två fall:

Ett undantag skulle kastas ifall ett produktID matas in av användaren, som inte kunde matchas mot något i produktkatalagonen.

Ett annat undantag skulle kastas för att simulera ett nätverksbortfall eller liknande situation då det inte finns åtkomst till databasen. Då programmet saknar stöd för databashantering simuleras detta hårdkodat genom anrop till ett visst ”förbjudet” produkt-ID.

Som ett led i undantagshanteringen skulle undantagstillståndet skickas som ett meddelande till View-lagret och skrivas ut till användaren, och där det var tillämpligt loggas i en fil.

I båda fall skulle man följa best practices för undantagshantering (vilket behandlas vidare i diskussionskapitlet).

Till den andra uppgiften skulle Observermönstret implementeras och nu funktionalitet skrivas för ett användargränssnitt där den totala summan i kassan för avslutade försäljningar skulle kunna avläsas. Detta skulle också göras både som utskrift till skärmen, och som en fil. Begränsningen var att dessa klasser inte skulle kommunicera med Controllern, utan endast uppdateras via Observermönstret.

Valbart för högre poäng var också att implementera två ytterligare valfria mönster från Gang of Four.

Jag arbetade tillsammans med Patricia Lagerhult för att skriva om klasserna. Eftersom vi jobbade tillsammans i seminarie 3 hade vi väldigt snarlik kod och kunde därför implementera stort sett samma undantagshantering.

2 Metod

Vi diskuterade var vi skulle ha våra nya undantagsklasser. Det kändes lockande att sätta dem i ett eget paket utanför alla andra. I slutändan bestämde vi oss för att vara konsekventa och gjorde samma sak som vi gjort för våra DTO:er, och valde att ha undantagen i de lägsta lagren de används.

I tidigare implementation av programmet hade vi en null-värdes ItemDTO som ”bänkvärmare” för just saknade Items. Detta behövde vi ändra när vi istället kastade undantaget ItemNotFoundException.

I och med att alla metoder som nu potentiellt kunde kasta undantag ändrades det publika gränssnittet, och som konsekvens orsakade en ohygglig oreda bland våra tidigare skrivna tester för metoderna, som nu krävde try-catch-block. Själva undantagen fick i samma veva egna tester, som till stor del gick ut på att tvinga fram undantagstillstånden, och testa att ett undantag kastades tillbaka.

Ett util-paket skapades och fick bli hemmet för den nya klassen ExceptionLogger, som fick ansvar för att skriva ut en loggfil för de undantag som genererades vid ”nätverksbortfall med databasen”. (Denna kom i senare skede att bli en subklass.)

Till tidigare iteration i seminarium 3 hade vi redan skrivit i ”Register”-klassen i model-lagret en metod att hålla reda på hur mycket pengar som hade betalats sedan applikationen startat. Ett anrop från samma metod var det naturliga sättet att trigga Observers att göra sitt jobb. RegisterObserver-interfacet sattes i modellen och implementerades av TotalRevenueView (som naturligtvis fick höra hemma i view-lagret) och av TotalRevenueFileOutput som vi tyckte mest simulerade någon slags databas för loggning och därmed fick höra hemma i integrationslagret. Båda observers har @Override på metoden notifyObserversBalanceHasChanged(), som anropas med rådande totalsumma i kassan och därifrån gör anrop till de respektive klassernas egna metoder.

TotalRevenueFileOutput fick dessutom tillsammans med ExceptionLogger vara med för att bilda Template-mönstret, eftersom vi när vi skrev de respektive klasserna att de hade mycket gemensamt. I util-paketet skapades en superklass för dem vid namn FileOutputter som fick ta över det mesta av utskrifterna (inklusive try-catch-blocken) så att subklasserna kunde fokusera på vilka meddelanden som skulle skrivas till filerna.

Det sista mönstret vi valde att använda var att göra vår Register till en Singleton. Detta dels för att vi tänkte att en Point-of-Sale inte skulle ha mer än en kassa, men också för att vi av en händelse råkade introducera en bugg i programmet där vi hade två olika Registers. Detta orsakade en hel del bekymmer för att räkna ut dagens total, och ett olyckligt samtida felkonstruerat test för just denna funktion ledde till ett par timmar av frustrerat felsökande. Singleton-mönstret var här inte bara lämpligt, det ger mig en trygghet att aldrig kunna göra om samma misstag. Max en instans av Register räcker gott för denna applikation!

3 Resultat

Både tester och källkod finns att hitta på följande länk:
https://github.com/kaytaffer/kevPOS

I nuvarande iteration av programmet finns det ingen interaktiv vy, utan den kör några hårdkodade anrop som skickar in fem stycken ItemDTO, varav den fjärde inte har ett matchande ID i ”databasen” och kastar ett ”InvalidInputException” och det sista med ID:t 404 med flit kastar undandtaget ”ConnectionException” för att simulera att databasen inte kan nås.

```
--------------------< se.kth.iv1350.kevpos:ood-pos >--------------------
Building ood-pos 1.0-SNAPSHOT
--------------------------------[ jar ]---------------------------------

--- exec-maven-plugin:3.0.0:exec (default-cli) @ ood-pos ---
------------------------------
Latest scanned item: Book
Description: It is cool
Running total: 53.0
including VAT: 3.0
------------------------------
Latest scanned item: Chocolate
Description: It is tasty
Running total: 75.4
including VAT: 5.4
------------------------------
Latest scanned item: Book
Description: It is cool
Running total: 128.4
including VAT: 8.4
------------------------------
ERROR: You've entered something wrong. Please try again.
ERROR: Please reconnect and try again.
```

Figur 3.1. Utskrift när de hårdkodade värdena körs.

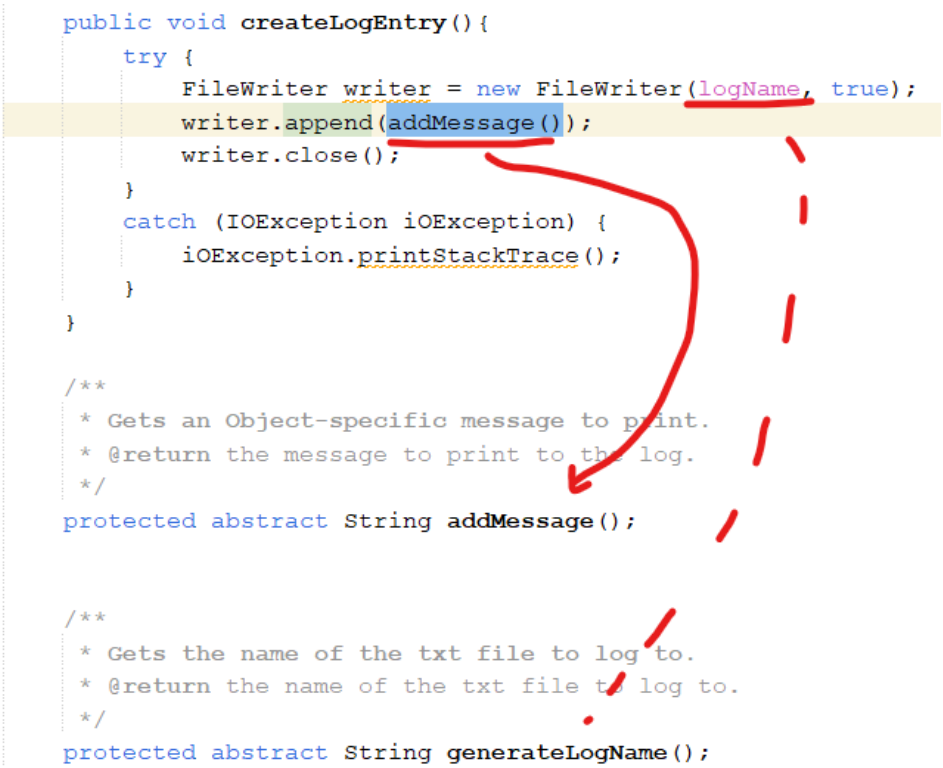
För de items som kan matchas skrivs deras information ut, och för de övriga skrivs undantagets meddelande.

Programmet skriver också till projektpaketet en txt-fil med tid då ”nätverksförlusten” kastades. Samma sak görs till en annan fil varje gång en försäljning slutförs, som loggar den senaste totalintäkten. Detta skrivs också till konsolen.

```
ERROR: Please reconnect and try again.
Today's total revenue is: 128.4 money units.

-------------------------------
---Receipt---
2022-05-19T17:58:24.137138500
2.0  Book  cool  100.0
1.0  Chocolate  tasty  20.0

Total price: 128.4
Discount: 0.0
Received Payment: 200.0
Change: 71.6
-------------------------------
Latest scanned item: Book
Description: It is cool
Running total: 53.0
including VAT: 3.0
-------------------------------
Today's total revenue is: 181.4 money units.

-------------------------------
```

Figur 3.2. Utskrift från TotalRevenueView

Följande kod hämtas ur FileOutputter, vår Template för filutskrifter. Subklasserna har tömts på den duplicerade koden, och svarar bara för innehållet i det som i bilden markerats med rött.



```
public void createLogEntry(){
    try {
        FileWriter writer = new FileWriter(logName, true);
        writer.append(addMessage());
        writer.close();
    }
    catch (IOException iOException) {
        iOException.printStackTrace();
    }
}

/**
 * Gets an Object-specific message to print.
 * @return the message to print to the log.
 */
protected abstract String addMessage();


/**
 * Gets the name of the txt file to log to.
 * @return the name of the txt file to log to.
 */
protected abstract String generateLogName();
```

Figur 3.3. FileOutputter Template-mönstrets implementation.

Slutligen, i följande figur syns implementationen av vår singleton, som vi valde att göra vår Register till:

```
private static final Register INSTANCE_OF_REGISTER = new Register();

/**
 * Creates a <code>Register</code> singleton.
 */
private Register() {
    this.balance = 0;
    this.registerObservers = new ArrayList<>();
}

/**
 * Returns the only existing instance of this singleton.
 * @return the <code>Register</code>.
 */
public static Register getRegister(){
    return INSTANCE_OF_REGISTER;
}
```

Figur 3.4. En och bara en Register

3 Diskussion

Att alla tester av publika metoder med nya undantag gick sönder är ett varnande exempel vad som kan hända om man är oförsiktig och ändrar det publika gränssnittet utan vidare.

I och med att alla metoder som nu potentiellt kunde kasta undantag ändrades det publika gränssnittet, och som konsekvens orsakade en ohygglig oreda bland våra tidigare skrivna tester för metoderna, som nu krävde try-catch-block.

Själva undantagsklasserna var svåra att testa direkt eftersom de till stor del bara har konstruktorer. Istället får vi lägga krut på de metoder som kastar undantag och att try-catch-hanteringen i metoderna som tillämpar dem.

I controller omvandlas undantagen från djupare lager till mer generiska former med användarvänliga meddelanden för att ha rätt abstraktionsnivå, men behåller sina ursprungliga causes utifall man skulle vilja implementera något slags adminläge i vyn.

Ett bekymmer med vår Template är hur vi valt att lösa att varje FileOutputter-subklass ska skriva till en egen txt-fil. Detta görs genom att i konstruktorn tvinga objekten att ange ett namn på denna fil genom generateLogName-metoden. Finns det inte en vettig @Override i subklassen fångas detta av ett catch all-namn ("poorlyImplementedObjectOfSuperclassFileOutputter.txt”) som kommer vara där utskriften hamnar, men i denna iteration är det fullt möjligt att fler olika FileOutputters skriver till samma log. Detta är inte en svaghet med Template-mönstret, utan med vår implementation, och behöver nödvändigtvis inte vara en nackdel, men är i varje fall problematiskt.

Jag tycker att kassan är ett väldigt bra objekt att göra till en singleton. Hade jag börjat med det hade det spart mig tid i längden, så egentligen ägnade jag inte andra kandidater något övervägande. Nu i efterhand känns även Controller, var och en av ”databasernas” hanterare, och olika loggningklasser som även de skulle kunna vara singletons.