laboration 5

Viking Forsman

Version 1.0

Jonathan Vestin, Ghaith Shammout, Alexander Andersson

1 Antaganden

Denna laboration har en kravspecifikation som varierar beroende på vilket betyg man siktar på. Version 1 utgör grundversionen av applikationen, medan efterföljande versioner utökar funktionaliteten för att uppnå högre betyg. Jag siktar på betyg 5, vilket innebär att funktionaliteten från både version 2 och 3 ska inkluderas i koden.

Vissa av kraven specificerar hur produkterna i det lokala lagret ska synkroniseras med motsvarande produkter i onlinelagret, som är tillgängliga via ett API. Eftersom dessa krav kan tolkas på olika sätt har jag sammanställt tre tabeller som beskriver min tolkning. Dessa presenteras i Tabell 1, 2, 3 och 4 i appendix A. Klassdiagram över de nya och uppdaterade klasserna finns i appendix B.

2 Översikt

Detta är applikationens huvud funktionalitet:

K1: Användaren ska kunna trycka på en knapp för att synkronisera data.

K2: Hantera fel från API:et och visa felmeddelanden för användaren.

K3: Lagerstatus i centrallagret ska uppdateras vid försäljning av produkter.

K4: Applikationen ska automatiskt synkronisera med centrallagret varje minut.

K5: Applikationen ska kunna logga ändringar i produkternas pris och lagerstatus samt visualisera dessa på ett användarvänligt sätt.

3 Detaljerad beskrivning

Jag har försökt följa SOLID- och DRY-principerna i denna applikation, vilket innebär att jag strävar efter att skapa modulär och återanvändbar kod samt att undvika duplicering. Detta gör koden både lättare att underhålla och mer skalbar. Applikationen använder också MVVM-arkitekturen, vilket hjälper till att separera användargränssnittet från affärslogiken.

Överblickande beskrivning av klasserna i applikationen:

- `ApiService` Och dess interface `IApiService` ansvarar för kommunikationen med det API som används i laborationen. I interfacet definieras en metod för att hämta produkter från online-lagret samt en metod för att uppdatera lagerstatusen för en specifik produkt. Under huven används `HttpClient` för

att ladda ner information från API:et och `XDocument` för att tolka XML-svaret. Om ett fel inträffar returneras ett felmeddelande som kan användas för att informera användaren.

- `DataService` och dess interface `IDataService` har utökats med funktionalitet för att logga ändringar i de lokala produkternas pris och lagerstatus. Detta har implementerats genom en lyssnare som registrerar ändringar i dessa attribut. För att lagra informationen används `StockLog` och `PriceLog`, som är records – en immutable datatyp för att hantera oföränderliga datastrukturer. Applikationen lagrar dessa i två dictionaries, en för priser och en för lagerstatus, där produktens ID fungerar som nyckel för snabb åtkomst.

- `TimerService` och dess interface `ITimerService` används för att hantera en återkommande timer med ett konfigurerbart intervall. Tjänsten utnyttjar `DispatcherTimer` för att generera tidsbaserade händelser, vilket möjliggör att andra komponenter i applikationen kan reagera på tidsstyrda uppgifter. Klassen exponerar metoder för att starta, stoppa och uppdatera tidsintervallet, samt ett TimerTick-event som utlöses vid varje tidsintervall. I denna applikation används den för att synkronisera med online-lagret med ett intervall på en minut.

- `ViewModelBase`, en abstrakt klass, innehåller grundläggande funktionalitet för de två huvudvyerna i applikationen. Under denna laboration har den utökats med metoder för att hantera synkronisering, samt för att informera användaren om fel som kan uppstå vid kommunikation med online-lagret. Synkronisering sker genom två knappar, en för att ladda ner information och en för att ladda upp den, samt en metod som körs med jämna mellanrum. I båda klasserna används de nya tjänsteklasserna för att kapsla in funktionaliteten och därmed förenkla underhållet.

- `StoreViewModel` ansvarar för logiken kring försäljning av produkter, vilket i sin tur kan påverka deras lagerstatus. Därför har klassen utökats med funktionalitet för att synkronisera dessa ändringar med online-lagret. Vid försäljningstillfället hämtas det aktuella priset och dess lagerstatus. Om det fortfarande finns tillräckligt många produkter i lagret genomförs köpet, varefter online-lagret uppdateras för att spegla förändringarna. Detta för att säkerställa att lagersaldot hålls konsekvent mellan det lokala systemet och online-lagret.

- `PriceAndStockLogsDialogViewModel` ansvarar för att visualisera ändringar i produkternas lagerstatus och pris. Detta görs i en dialogruta med hjälp av OxyPlot, där användaren kan välja en specifik produkt och ange en tidsperiod för att visa relevanta loggar. Vid valet av bibliotek övervägdes även

`LiveCharts2`, som erbjuder ett modernt gränssnitt med stöd för animerade diagram. Dock föll valet på `OxyPlot`, eftersom det är mer lättviktigt, har bättre prestanda och är väl dokumenterat. Dessa faktorer gjorde det till det mest lämpliga alternativet för denna applikation enligt min bedömning.

- `LabelledComboBox` är en UserControl-klass som låter användare välja ett av flera förutbestämda alternativ från en lista. Likt de andra egendefinierade klasserna för användarinput har jag lagt till en etikett (label) som kan användas i användargränssnittet för att tydligare beskriva kontrollens syfte. Detta är något som inte stöds som standard i WPF för textboxes och liknande inmatningskontroller. Därför ansåg jag att introduktionen av denna typ av kontroll förbättrar tydligheten genom att ge mer kontext i gränssnittet. Kontrollen har en hjälpkklass, `ComboBoxItemModel`, som används för att representera alternativen som användaren kan välja bland. Varje alternativ består av en textsträng, som utgör det visade namnet, och ett värde av typen `object`, vilket gör det möjligt att använda boxing för att stödja valfri datatyp. Detta ger större flexibilitet och möjliggör återanvändning av komponenten i olika sammanhang där olika typer av data behöver representeras i en lista. I denna applikation används kontrollen specifikt för att låta användaren välja vilken produkt som ska visas i diagrammen som illustrerar förändringar i pris och lagerstatus över tid.

4 Problem

Ett problem som uppstod i version tre av laborationen var kravet på att visualisera förändringar i produkternas lagerstatus och pris. Att samla in denna information var relativt enkelt och löstes genom att lägga till lyssnare i `DataService`-klassen, vilka loggade ändringar av dessa attribut. Däremot var själva visualiseringen mer utmanande. Detta eftersom WPF saknar inbyggt stöd för att rita grafer utan att använda tredjepartsbibliotek (ett av kurs kraven), medan WinForms erbjuder grafritning via `Microsoft Chart`.

En möjlighet var att använda `Canvas` och `Polyline` för att rita grafen manuellt utan externa verktyg, men det skulle också kräva mer kod att skriva och underhålla. Ett annat alternativ var att bädda in en WinForms-kontroll i WPF. Efter att ha rådfrågat läraren fick jag dock tillåtelse att göra ett undantag och använda ett tredjepartsbibliotek. Jag valde då `OxyPlot`, eftersom det erbjuder ett smidigt och kraftfullt sätt att skapa grafer i WPF.

5 Sammanfattning

Jag är överlag nöjd med applikationen, som enligt mig uppfyller de funktionalitetskrav som anges i laborationsbeskrivningen på ett genomtänkt sätt. Det finns dock utrymme för förbättringar vid en eventuell vidareutveckling. Till exempel skulle man kunna lägga till en funktion för att visuellt jämföra pris- och lagerstatusutvecklingen för flera produkter samtidigt. I den nuvarande implementationen visas endast en produkts utveckling i taget.

Ett annat område för potentiell förbättring är hur `DataService` hanterar ändringar av produkter och kvitton. För närvarande går det inte att lägga till eller ta bort objekt ur de datastrukturer de förvaras i, men själva objekten är mutable. Detta är en svaghet eftersom det innebär att programmerare av misstag kan ändra värden som inte borde kunna ändras, exempelvis produktens ID. En möjlig lösning skulle kunna vara att modifiera den property som ansvarar för detta värde så att den enbart kan sättas en gång. Dock skulle detta göra den initiala processen att lägga till objekt med unika ID mer komplicerad.

Det optimala vore att enbart tillåta ändringar i objekt via en metod i `DataService`-klassen. Detta skulle ge mer kontroll över hur och när objekt ändras, och dessutom skulle det vara enklare att föra logg på förändringar i pris och lagerstatus. För att förhindra att objekten ändras utanför klassen skulle immutable versioner av modelklasserna behöva skapas. Men applikationen innehåller redan många klasser, och införandet av ännu fler skulle potentiellt kunna påverka underhållet och översikten av applikationen negativt.

Appendix A: Tolkning av synkroniseringskraven

Användaren köper en produkt.		
Version	**Lokala produkter påverkas?**	**Online produkter påverkas?**
1	Lagerstatus uppdateras	Nej
2	Lagerstatus uppdateras	Lagerstatus uppdateras baserat på lokala produkten
3	Lagerstatus uppdateras	Lagerstatus uppdateras baserat på lokala produkten

Table 1 – Översikt över hur de olika versionerna av applikationen hanterar lagerstatus för lokala produkter och online produkter vid köp.

Användaren trycker på "Sync Upload" knappen		
Version	**Lokala produkter påverkas?**	**Online produkter påverkas?**
1	Inte implementerat	Inte implementerat
2	Nej	Lagerstatus uppdateras baserat på online produkten
3	Nej	Lagerstatus uppdateras baserat på online produkten

Table 2 – Översikt över hur de olika versionerna av applikationen hanterar synkronisering baserat på lokala produkter.

Användaren trycker på "Sync Download" knappen		
Version	**Lokala produkter påverkas?**	**Online produkter påverkas?**
1	Pris och lagerstatus uppdateras baserat på online produkten. Om produkten saknas i det lokala lagret läggs den automatiskt till.	Nej
2	Pris och lagerstatus uppdateras baserat på online produkten. Om produkten saknas i det lokala lagret läggs den automatiskt till.	Nej
3	Pris och lagerstatus uppdateras baserat på online produkten. Om produkten saknas i det lokala lagret läggs den automatiskt till.	Nej

Table 3 – Översikt över hur de olika versionerna av applikationen hanterar synkronisering baserat på online produkter.

Periodisk synkronisering en gång i minuten		
Version	**Lokala produkter påverkas?**	**Online produkter påverkas?**
1	Inte implementerat	Inte implementerat
2	Inte implementerat	Inte implementerat
3	Pris och lagerstatus uppdateras baserat på online produkten	Nej

Table 4 – Översikt över hur periodisk synkronisering påverkar lagerstatus och priser för lokala produkter.

Appendix B: Applikationsdesign

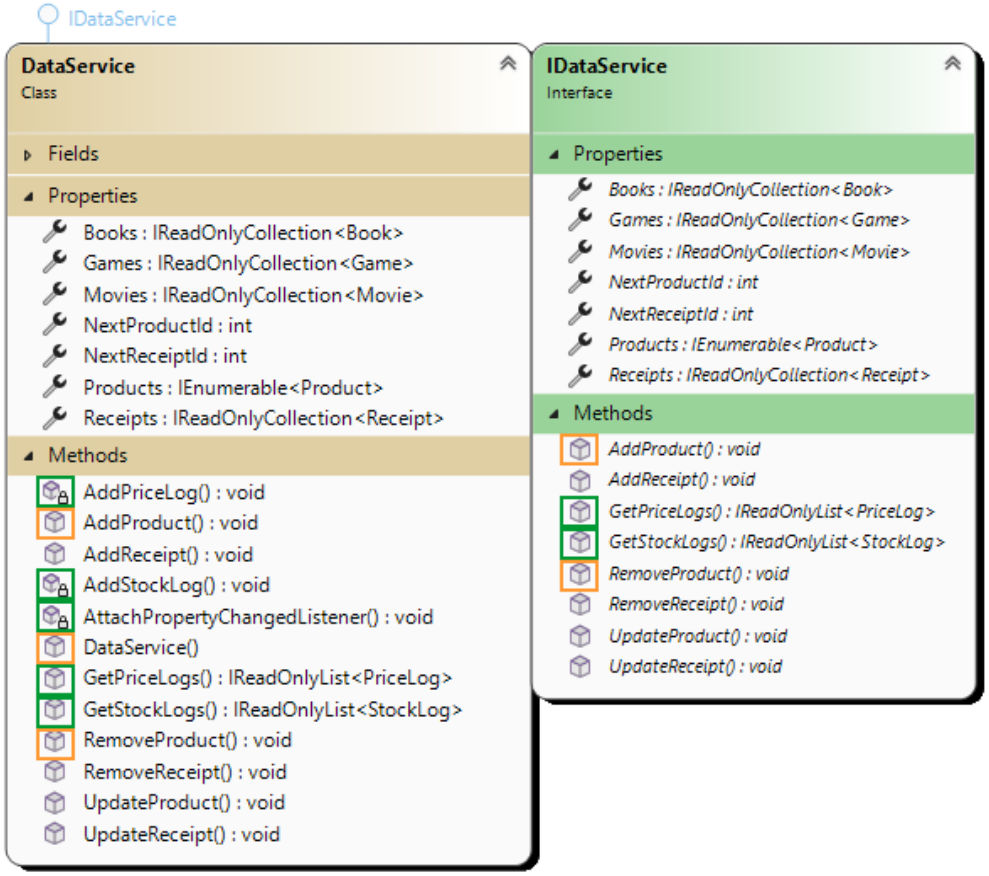

Figure 1 – Diagrammet visar den uppdaterade `DataService`-klassen och dess tillhörande interface. Uppdateringen inkluderar nya metoder och datastrukturer för att hantera loggar om produkternas priser och lagerstatus från ett data perspektiv. För tydlighetens skull har nya metoder markerats med en grön markering, medan uppdaterade metoder har fått en gul markering.

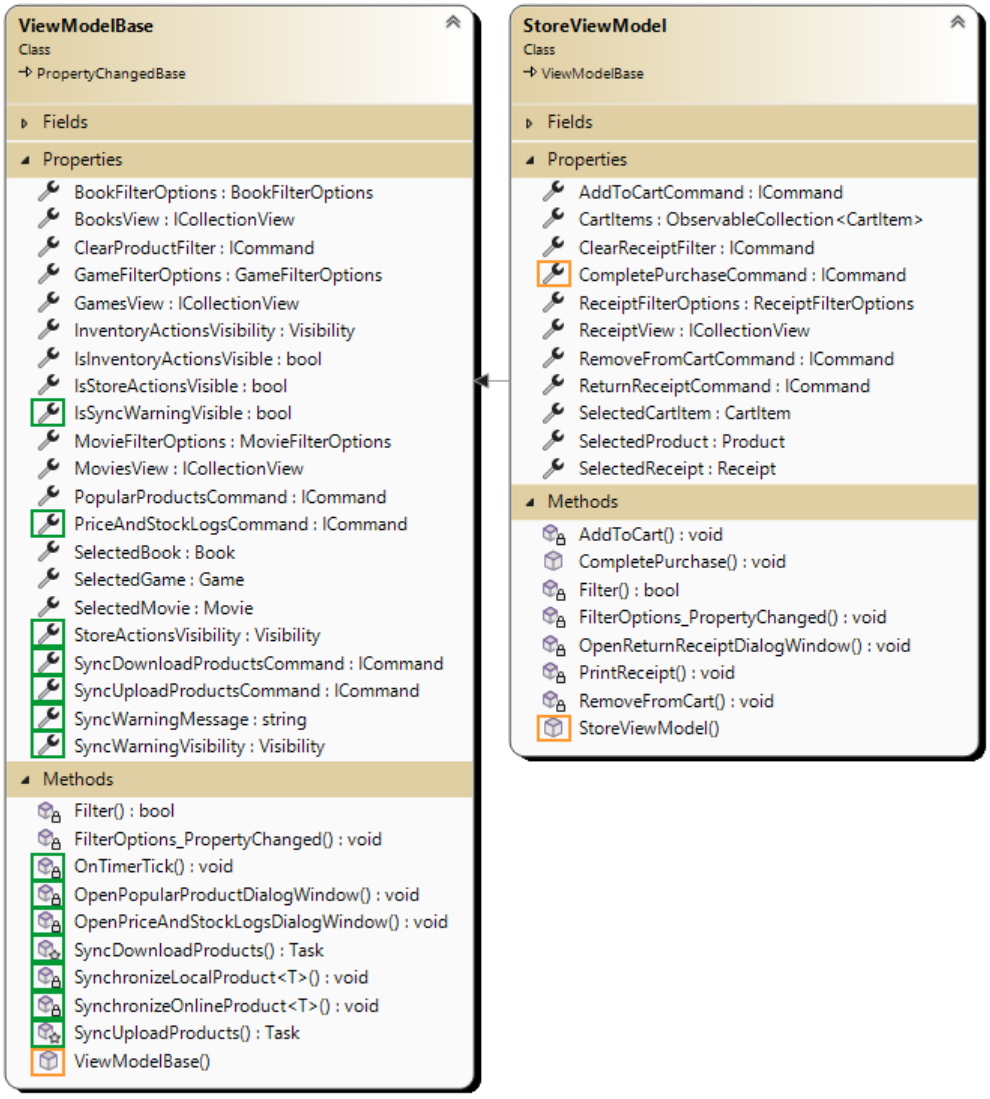

Figure 2 – Uppdaterat klassdiagram för `ViewModelBase` och `StoreViewModel`. Nya metoder har lagts till för att hantera synkronisering vid knapptryck, vid regelbundna intervall och vid försäljning, samt hantering av fel vid API-kommunikation. För tydlighetens skull har nya metoder markerats med en grön markering, medan uppdaterade metoder har fått en gul markering.

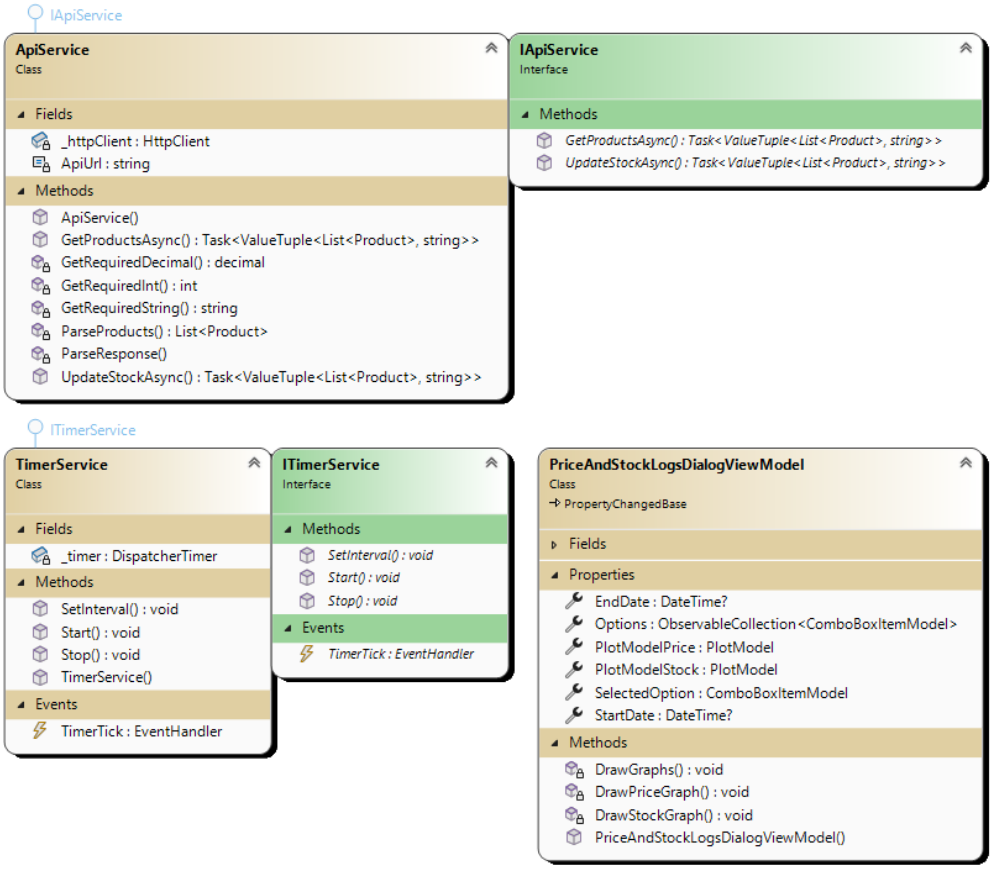

Figure 3 – Diagrammet visar de nya klasserna i applikationen (med undantag för Records eftersom de inte stöds i Visual Studios diagram ritar verktyg). Jag har försökt att hålla tjänsteklasserna generella, och istället forma deras output efter mina behov i viewmodel klasserna. Detta för att förbättra möjligheten för återanvändning av tjänsterna i andra kontexter.