laboration 4

Viking Forsman

Version 1.0

Jonathan Vestin, Ghaith Shammout, Alexander Andersson

1 Antaganden

Under laborationen har två antaganden gjorts. Det första är att produktinformationen sparas i CSV-format när programmet avslutas och läses in vid uppstart, men att filen inte behöver uppdateras under programmets körning. Det andra antagandet är att applikationen i framtiden ska kunna komplettera befintliga produkter med data från ett online-lager, åtkomligt via ett API. Därför bör utvecklingen ske med denna integration i åtanke.

2 Översikt

Detta är applikationens grundkkrav utifrån ett användarperspektiv:

K1: Lägga till nya produkter. Varje produkt ska ha ett unikt ID/varunummer. Produkter får inte kunna ha felaktig information. Exempelvis avsaknad av obligatoriskt fält (namn och pris), negativa priser, bokstäver i varunumret, en speltid som är negativ eller inte en siffra.

K2: Ta bort produkter. Om någon i personalen försöker ta bort en vara ur sortimentet, och dess lagerstatus inte är noll, ska en dialogruta dyka upp om man verkligen vill ta bort varan. I dialogrutan ska det gå att svara Ja eller Nej.

K3: Lägga till en leverans av produkter från grossit.

K4: Försäljning av produkt till kund. Man ska inte kunna sälja varor som inte finns på lager.

K5: Programmet består av två huvud vyer. En vy för att hantera information om produkter, och en vy för att hantera försäljning av produkter.

Jag satsar på betyg 5, vilket innebär att vissa extrakrav ska uppfyllas:

K6: Alla produkter en kund har köpt ska skrivas ut på ett kvitto.

K7: Det ska gå att söka på attribut som exempelvis produkt, lagerstatus och artist.

K8: Kund ska kunna returnera en vara, varefter lagerstatusen ska ökas.

K9: Det ska gå att få ut topplistor på mest sålda produkter för varje månad eller år.

K10: Det ska även kunna hämta ut information om totalt antal produkter sålda varje månad och år.

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:

- `Produkt` är en abstrakt klass som definierar gemensam information för alla produkttyper, såsom ID, pris, namn och lagerantal. `Book`, `Movie` och `Game` är klasser som ärver från `Produkt` och innehåller egenskaper och funktionalitet som är specifika för respektive produktkategori. Vi använder dessutom klasserna `ProductFilterOptions`, `BookFilterOptions`, `GameFilterOptions`, `MovieFilterOptions` för att hantera användarens filterinställningar och sökpreferenser för varje produkttyp.

- `Receipt` är en klass som innehåller ett ID, ett datum och en lista med instanser av klassen `ReceiptItem`, vilka representerar de produkter som kunden har köpt. Varje `ReceiptItem` innehåller inte bara en referens till produkten, utan även dess namn, pris och antal enheter som kunden har köpt. Denna ytterligare information är viktig för att kunden ska kunna reklamera sina varor. Namnet och priset på produkten kan ändras, och därför lagras denna information också i `ReceiptItem` för att bevara korrekt information vid reklamation.

- `RelayCommand` är en flexibel klass som implementerar `ICommand` och används för att hantera kommandon i MVVM-arkitekturen i WPF. Den tillåter exekvering av åtgärder både med och utan parametrar och kontrollerar om kommandot kan exekveras baserat på valfria funktioner. Klassen gör det enklare att binda kommandon till UI-element utan att behöva skapa separata kommandoklasser, vilket bidrar till mer återanvändbar och underhållbar kod.

- Interfacet `IDataService` och klassen `DataService` ansvarar för CRUD operationer som hanterar produkter (såsom böcker, filmer och spel) samt kvitton i applikationen. Dessa objekt lagras i interna samlingar, och klassen tillhandahåller metoder för att lägga till, uppdatera och ta bort produkter och kvitton. En del av detta ansvar innebär att säkerställa att nya produkter får unika ID. Det är dock viktigt att notera att elementen i samlingen fortfarande är "mutable", vilket innebär att deras värden kan ändras utanför `DataService` klassen.

- Interfacet `IPrintService` och klassen `PrintService` ansvarar för att hitta en skrivare på datorn och skriva ut den angivna texten på den. Om inte en skrivare kan hittas skrivs istället kvittot till en PDF i enlighet med kravspecen. I denna applikation används klassen för att skriva ut kvitton.
- Interfacet `IStorageService` och klassen `StorageService` hanterar sparande och inläsning av applikationens data till en CSV-fil, som enligt labbspecifikationen lagras i samma mapp som applikationen körs från. I en verklig applikation vore det dock mer lämpligt att använda appens lokala datalagring, vanligtvis `C:\Users\användarnamn\AppData\Local`. Denna plats är avsedd för applikationsspecifik data och erbjuder fördelar som bättre säkerhet och åtkomst utan administratörsrättigheter. I .NET kan denna mapp nås dynamiskt.
- `ViewModelBase` är en abstrakt klass som innehåller funktionalitet som är gemensam för de två huvudvyerna i applikationen. Den hanterar bland annat logiken för att välja och filtrera objekt i listorna som visar produkter. `InventoryViewModel` och `StoreViewModel` är däremot specifika för sina respektive vyer. Den förstnämnda av dessa ansvarar för CRUD-operationer på produkterna, medan den andra fokuserar på försäljning och reklamation av produkter.
- `ReturnReceiptDialogViewModel` är en klass som ansvarar för logiken gällande retur av produkter. Klassen `ReturnItem` innehåller referenser till produkten och kvittot, samt hur många enheter som ska returneras. Antalet enheter som returneras kan inte vara negativt eller mer än antalet köpta enheter enligt kvittot, och produkternas lagersalsdo ändras i enlighet med antalet återlämnade produkter.
- `PopularProductDialogViewModel` är en klass som hanterar logiken för att sammanställa information om populära produkter. Användaren kan ange ett datumintervall samt ett heltal för att generera en rapport över de mest sålda produkterna inom perioden, där heltalet anger antalet unika produkter som ska inkluderas. För att presentera produkterna används klassen `PopularProductItem`. Utöver den vanliga produktinformationen innehåller den även beräknade värden för det totala antalet sålda enheter samt deras sammanlagda försäljningsvärde.
- `ProductDialogViewModel` är en abstrakt klass som ärvs av `BookDialogViewModel`, `GameDialogViewModel` och `MovieDialogViewModel`. Dessa klasser ansvarar för logiken för att skapa eller uppdatera existerande produkter. Utöver detta finns även klassen `StockDialogViewModel` som ansvarar för att fylla på lagersaldot på produkter.

4 Problem

En begränsning i den nuvarande implementationen är lagringen av data i en CSV-fil. En relationsdatabas skulle vara mer fördelaktig eftersom den hanterar data säkrare och mer strukturerat, med bättre prestanda vid sökning och uppdatering. Dessutom erbjuder databaser funktioner som DELETE CASCADE för att förhindra inkonsekvent data. Ett alternativ till CSV skulle kunna vara serialisering till JSON, som även om det inte är lika effektivt för stora mängder data, möjliggör representation av mer komplex information, till exempel objekt som innehåller andra objekt.

Jag försökte göra klassen för permanent lagring via CSV mer generell genom att använda reflektion i en abstrakt Repository-klass, där varje modellklass kunde skapa sitt eget repository och implementera ett specifikt interface. Detta användes sedan i en UnitOfWork-klass för enhetlig hantering. När Receipt-klassen introducerades, som innehöll en lista med instanser av en annan klass, blev lösningen för komplicerad. Jag refaktorerade därför klassen och hårdkodade lagringen för varje modell. Även om detta minskar flexibiliteten, är det enklare och mer lättförståeligt, vilket underlättar underhåll och vidareutveckling.

Ett specifikationskrav som kan vara problematiskt i en verklig kontext är förbudet mot att använda tredjepartsbibliotek i applikationen. Denna implementation följer MVVM-principen, men WPF har inte fullt stöd för denna designprincip i sin standard. Hantering av kommandon, beroendeinjektion och notifieringar kräver ofta extra kod när endast inbyggda funktioner används. Tredjepartsbibliotek som Prism och MVVM Toolkit förenklar processen genom att erbjuda färdiga lösningar för DelegateCommand, Event Aggregator och automatisk hantering av INotifyPropertyChanged. Detta gör MVVM-implementeringen mer effektiv och lättare att underhålla. Prism erbjuder också Dependency Injection, vilket förenklar hantering av beroenden mellan komponenter och gör applikationen mer modulär och testbar genom att automatiskt injicera rätt instanser i ViewModels och andra delar av applikationen.

5 Sammanfattning

Detta är första gången jag använder MVVM i WPF på ett djupare plan, och jag tycker att det på många sätt liknar MVC-applikationer i .NET. Dock är WPF inte lika strikt när det gäller var applikationslogiken ska hanteras, vilket ger flera olika sätt att lösa problem. Detta är både en fördel och en nackdel, då flexibiliteten möjliggör anpassade lösningar men också kan skapa osäkerhet och göra koden svårare att hålla konsekvent och underhållen.

Appendix A: Applikationsdesign

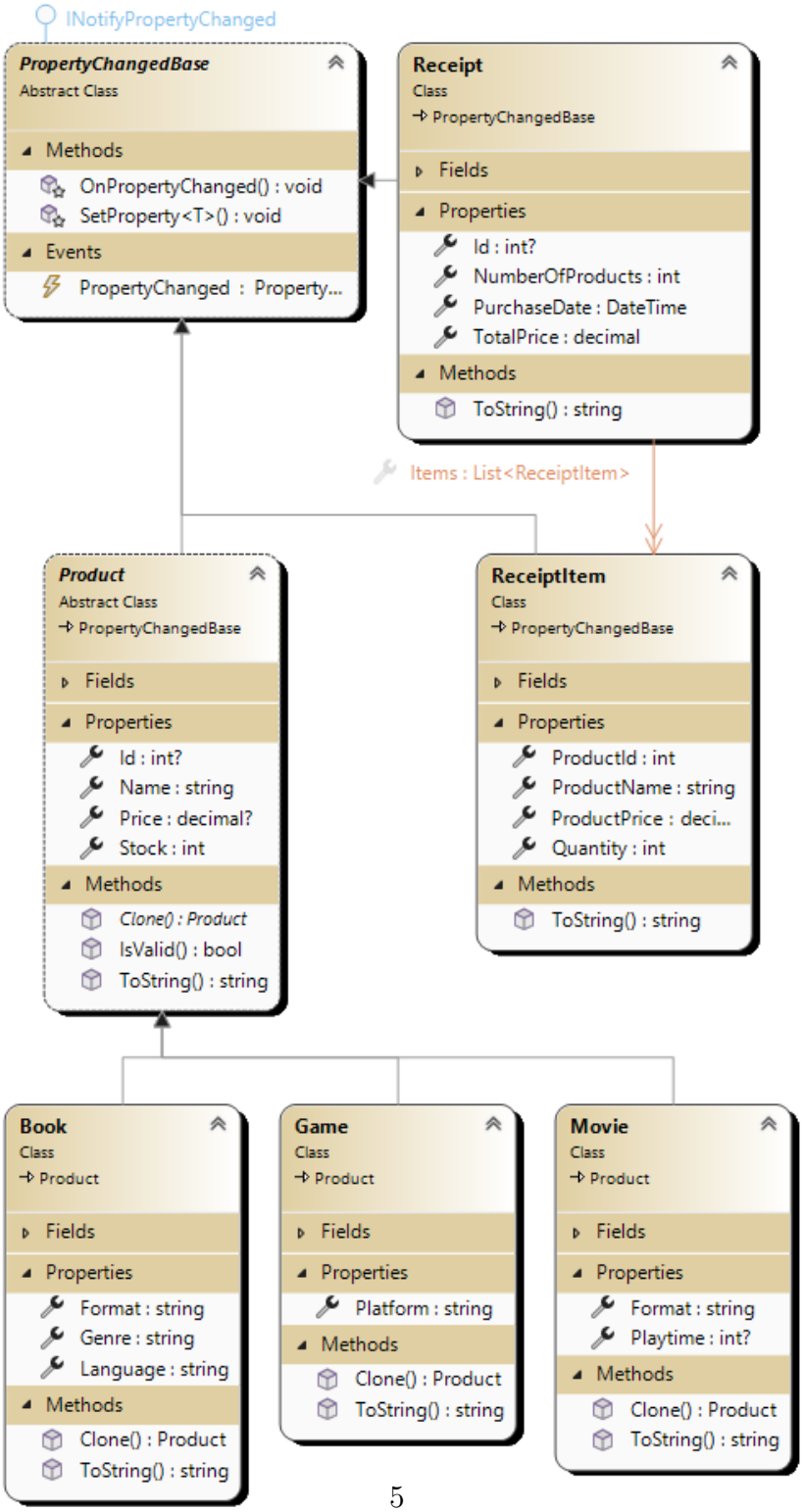

Figure 1 – Klassdiagrammet visar modellerna i applikationen, Dessa använder sig av `PropertyChangedBase` för att informera det grafiska interfacet om ändring av värden.

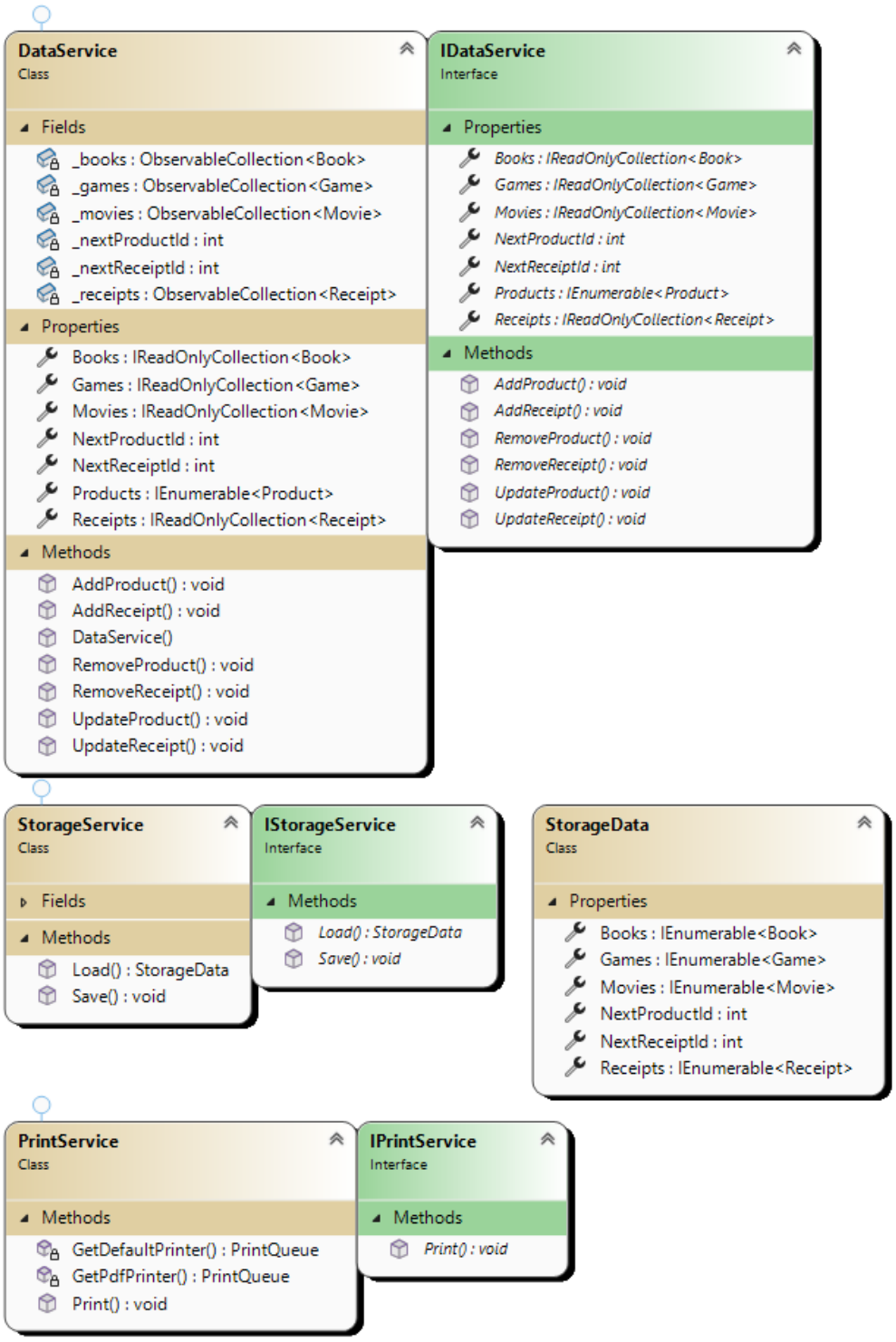

Figure 2 – Klassdiagrammet visar service klasserna i applikationen. Dessa är globalt tillgängliga genom sina interface, vilket möjliggör att den underliggande klassen lättare kan bytas ut vid behov.

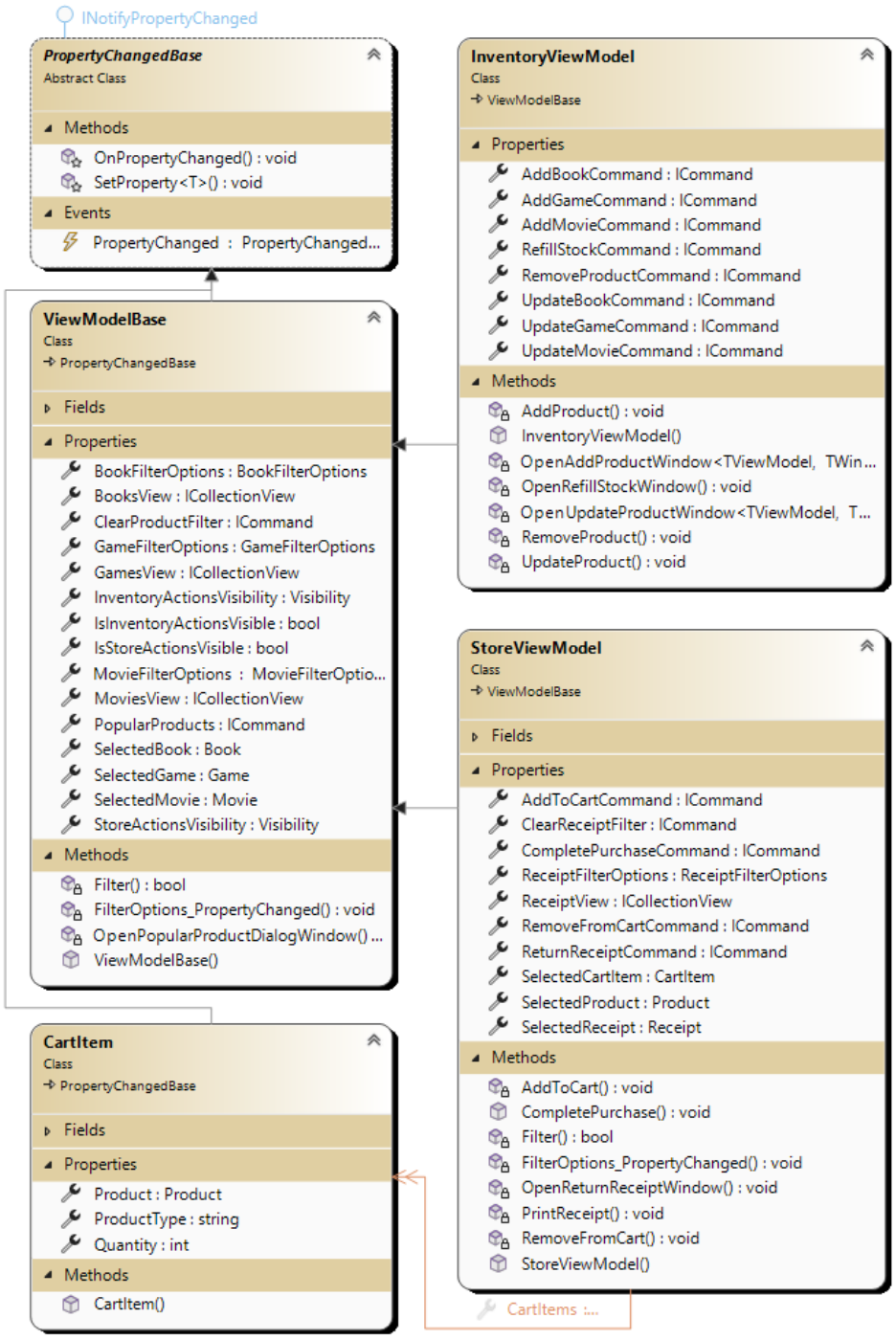

Figure 3 – Klassdiagrammet visar viewmodels i applikationen. Applikationen har två huvud vyer, vilka båda ärver från en abstract klass som innehåller gemensam funktionalitet.

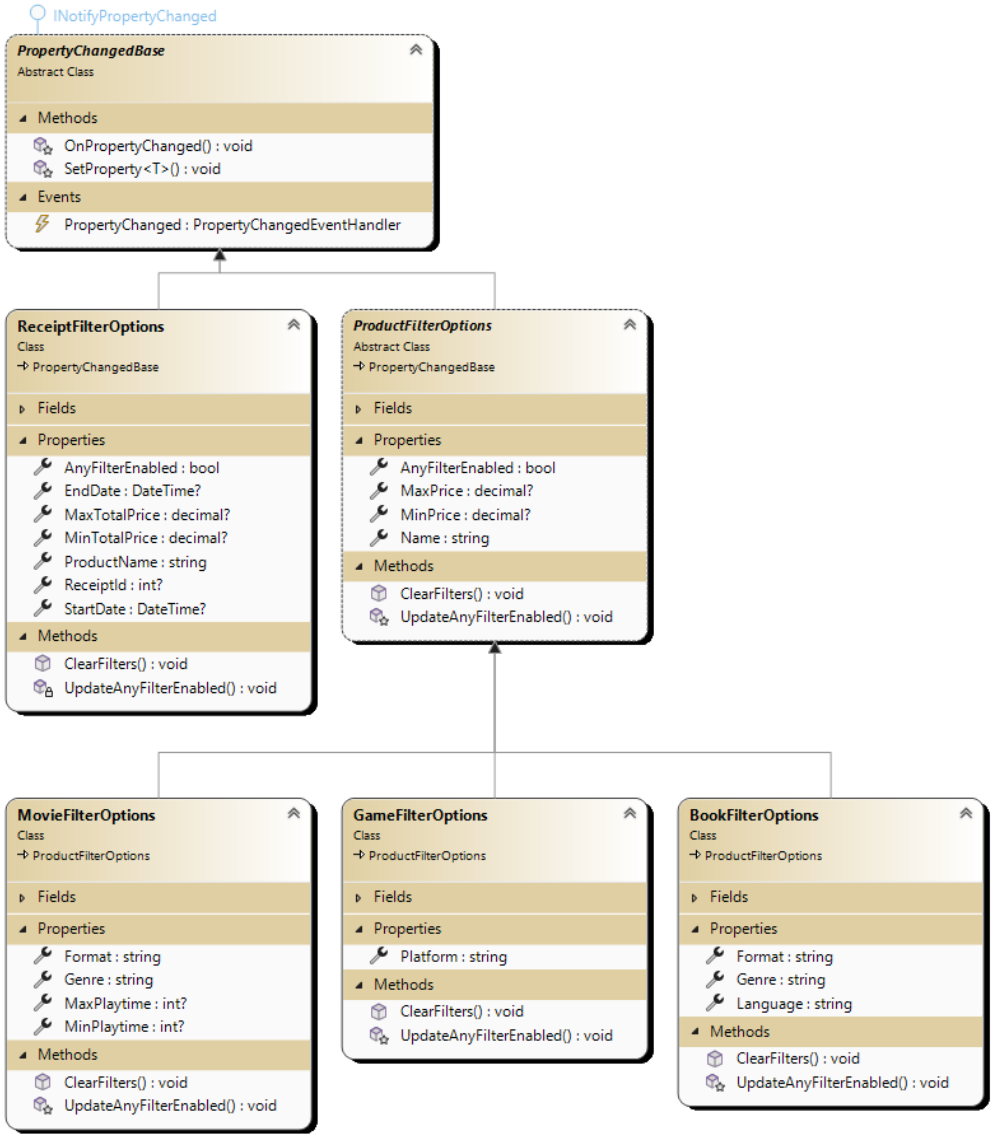

Figure 4 – För att underlätta filtreringsfunktionen har jag skapat en separat klass för varje modell (filmer, böcker, spel och kvitto). Samtliga produkter ärver från en abstrakt klass som innehåller gemensam information. Textsträngarna i klasserna används för att identifiera om det finns element där någon del av värdet matchar söksträngen, oberoende av versaler och gemener, medan övriga datatyper såsom int, DateTime och decimal filtreras baserat på intervall.

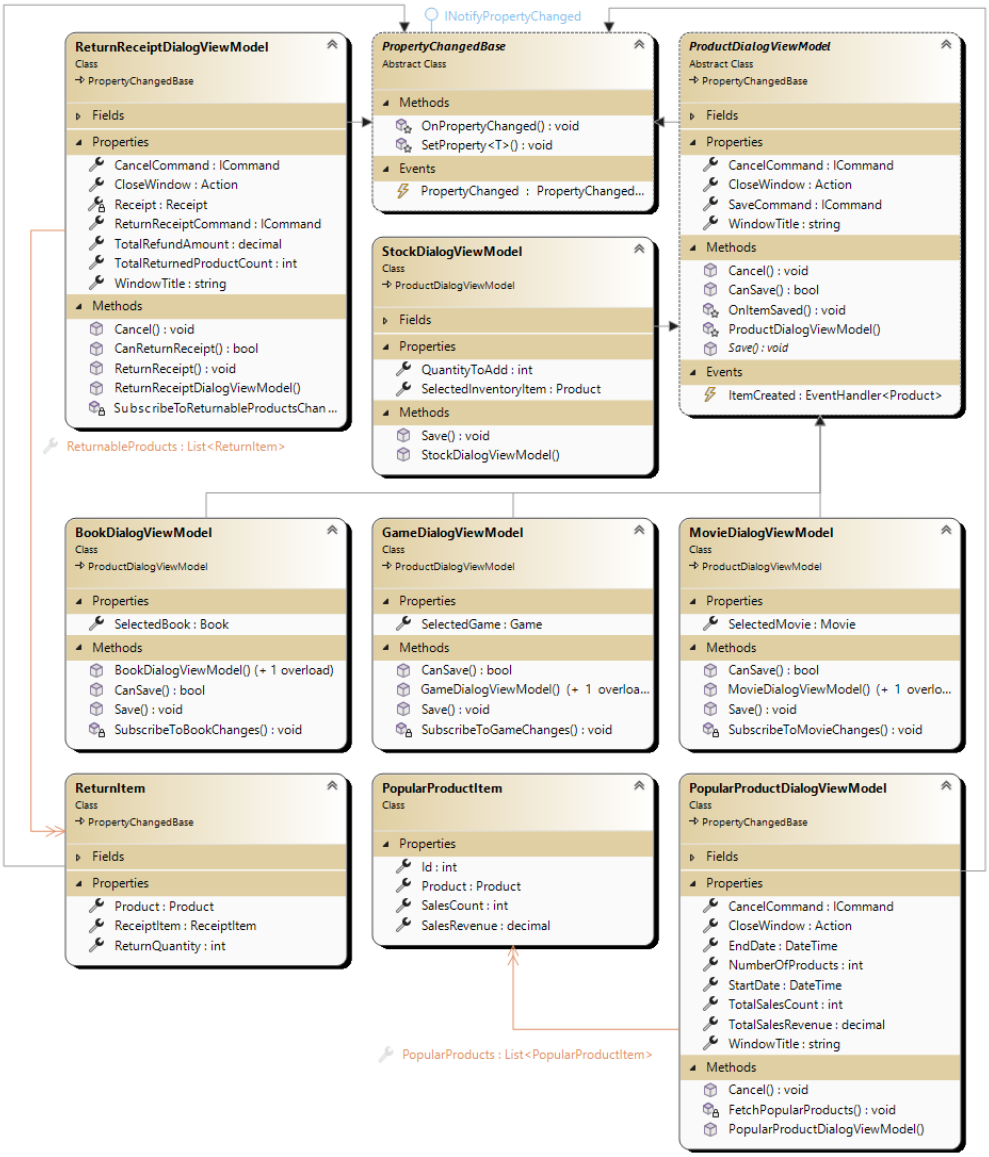

Figure 5 – Klassdiagrammet visar viewmodels för dialogrutor i applikationen. Jag använder dessa dialogrutor eftersom de kapslar in specifik funktionalitet. Dessutom bidrar de till ett mer avskalat gränssnitt genom att endast visa information när användaren efterfrågar den. Det finns dialogrutor för att skapa och ändra produkter, fylla på lagersaldo, visa försäljningsinformation samt hantera retur av produkter.

Appendix B: Inledande applikationsdesign

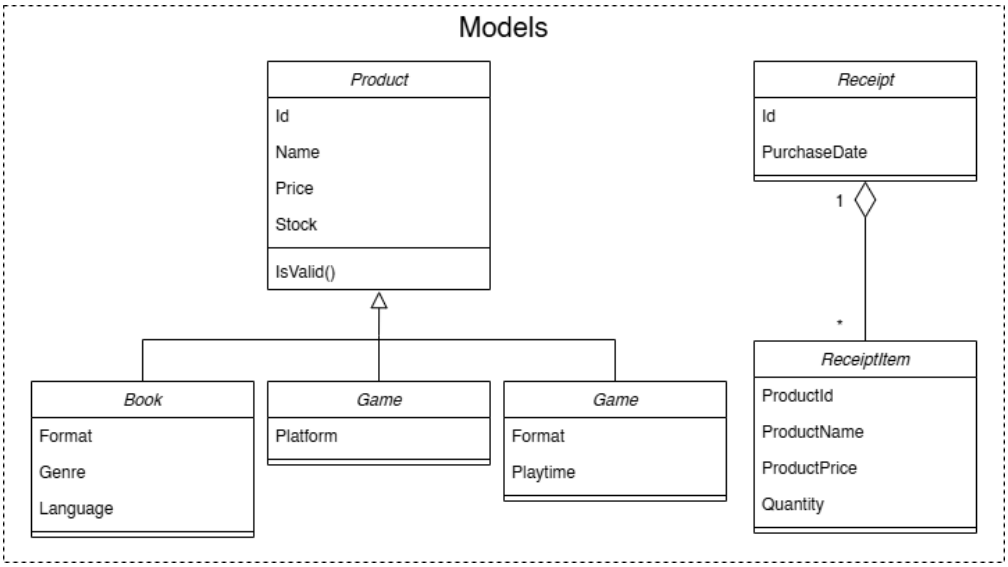

Figure 6 – Inledande design av modellerna i applikationen.

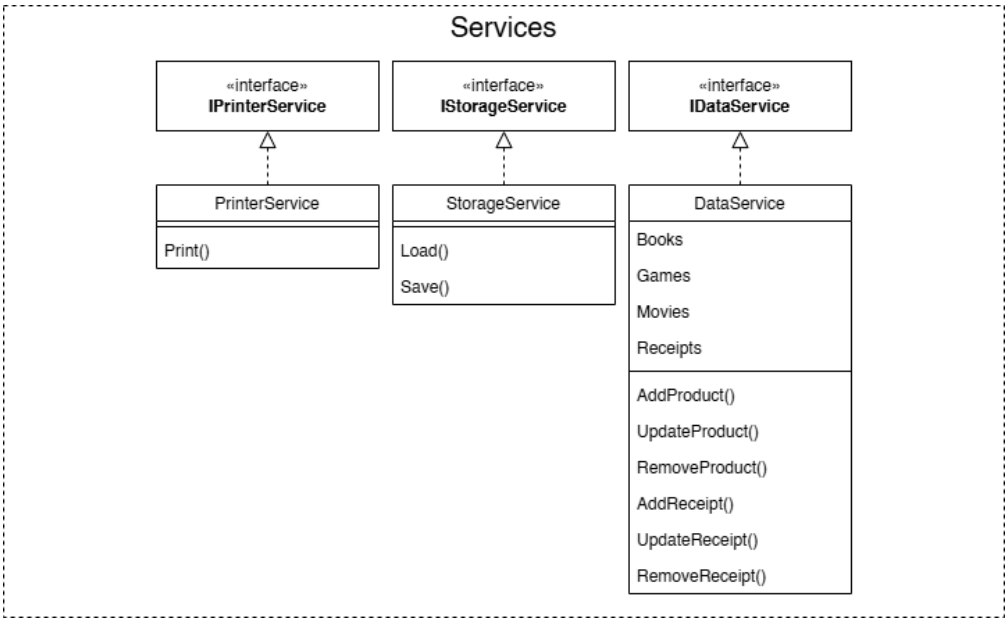

Figure 7 – Inledande design av service klasserna i applikationen.

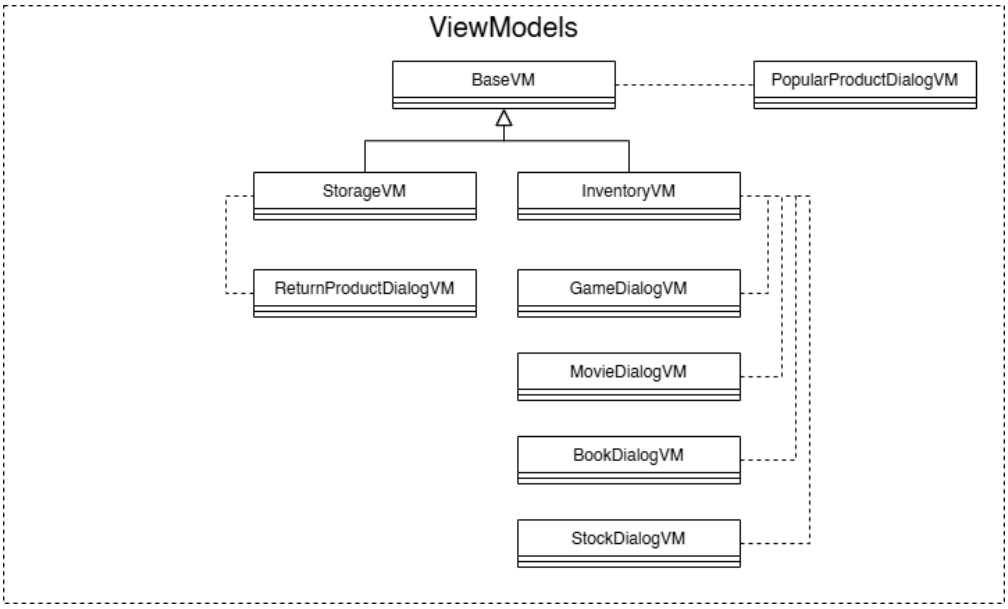

Figure 8 – Inledande design av viewmodel klasserna i applikationen.