**Media Shop**

Av: Fabian Fröding

Version: 5

# 1. Antaganden

* I klassen Product representeras varunumret av fältet ”id”.
* Hantering av id/varunummer sköts automatiskt. Det är ingenting användaren bör tänka på.
* Antal varor i lagret minskas redan när varan läggs i varukorgen. Detta för att förhindra att varan läggs till fler gånger än antalet i lagret. Om användaren stänger ner programmet medans varor finns i varukorgen läggs dessa tillbaka i lagret.
* Kvitto-funktion och återköp-funktion är endast tillgänglig i kassavyn.
* Vid återköp av en produkt uppdateras lagerstatus automatiskt, kassaperonsal/användaren behöver inte göra det manuellt.
* När alla produkter från ett tidigare köp/kvitto har returnerats/återköpts, tas kvittot bort från databasen.
* Vid återköp av en vara som tagits bort från lagret, så får användaren bekräftelse på återbetalning, men varan ”åter-registreras” inte till lagret eftersom den tidigare tagits bort.
* Produkter som tagits bort från lagret eller som returnerats via återköp tas inte med i beräkningen i statistik-funktionerna (top 10 mest sålda varor, total försäljning av en viss produkt).
* Programmet klarar av att hantera två olika produkter med exakt samma namn (olika id genereras), men det kan bli rörigt för användaren.

# 2. Översikt

MediaShop tillåter en användare att hantera olika produkter. Programmet har två flikar, där en del funktionaliteter visas på båda medans andra är unika för de två vyerna. T.ex. kan användaren lägga produkter i en varukorg och sälja produker i kassavyn, medans tillägning och borttagning av produkter endast kan utföras i lagringsvyn.

Programmet tillåter även användaren att generera grafer över produkter baserat på antal sålda enheter eller hur många gånger en specifik produkt sålts under ett visst tidsintervall. Användaren kan också filtrera listan med produkter baserat på olika sökkriterier.

När en produkt säljs genereras ett kvitto, som kan användas i kassavyn senare för att returnera varor från ett specifikt kvitto. I lagringsvyn kan använaren även uppdatera lagerstatus med ett angivet antal (leverans från grossist).

Båda vyerna tillåter användaren att se en lista med alla produkter i lagret.

# 3. Detaljerad beskrivning

Programmet använder sig av en MVC-liknande arkitektur.

* Models består av domän-enheter som t.ex. Product, Receipt.
* Views består av Windows Forms, i.e. “interfaces”.
* Controller-lagret är uppdelat i två ytterligare lager:
  + Controllers: Hanterar vilken data som skcikas vidare till vyerna från Repositories och hanterar även en del logik mellan dessa klasser.
  + Repositories: Hanterar lagring och hämtning av data från .txt-filer.

Programmet består av följande klasser:

* Domän-enheter:
  + Product: representationen av en fysisk produkt i lagret. Förutom grundläggande egenskaper som id, namn etc. så har en product även en ”type” som berättar vad för typ av produkt det är.
    - ProductType: En enum för att avgöra vilken kategori en Product tillhör.
  + Receipt: En klass som representerar ett kvitto. Klassen innehåller en lista av produkter och ett datum med formatet yyyymmddhhmmssfff. De tre sissta siffrorna i datumet genereras för att göra så att varje kvitto är unikt.
* Datalagring:
  + ”Repository”-klasser är det understa lagret som hanterar sparande och hämtning av data. Repository-klassens ansvar och funktionalitet är baserat på CRUD-operationer (Create, read, update, delete).
  + ”Controller”-klasser använder datan som skickas från Repository-lagret för att sedan skicka den vidare till vyerna som visas för användaren.
  + Repositories är baserade på ”Interfaces” för att bevara det grundläggande CRUD-ansvaret som klassen har.
  + Data sparas i en extern textfil.
* Vyer:
  + MainForm: Innehåller två vyer (kassa och lager) där olika funktioner kan användas.
  + NewProductForm: En popup-form för att fylla i uppgifter för en ny produkt som ska läggas till i lagret.
  + RefundForm: En vy som tillåter användaren att välja ett kvitto, och sedan välja en prdukt från det kvittot som önskas returneras (återköp).
  + StatsForm: En popup-form för att visa grafer över statistisk-funktioner.
* Grundläggande funktioner som getters/setters och liknande är inte inkluderat i klassdiagrammet. Detta för att hålla diagrammet minimalistiskt och begripligt.
* Metoder som ärvs från parent-klasser eller interfaces är inte representerade i den ärvande klassen. Detta för att reducera redundans.
* Domän-enheter (t.ex. Product, Receipt) är POCOs (Plain Old C-Object), som är klasser som endast håller information.
* Somliga metoder som återger en bool gör detta för att indikera om operationen lyckades eller inte. T.ex. om metoden ”Add(Product)” återger *true* indikerar det att tilläget av produkten lyckades medans *false* hade indikerat att tilläget misslyckades.

Klassdiagram

A close up of text on a white background

Description automatically generated

# 4. Problem

Från början var det tänkt att ha ytterligare ett ”Service”-lager mellan Controller och Repository, men detta togs bort för att undvika over-engineering då lagret inte kändes så användbart (service-layers är till för att öka separation of concerns mellan Controller och Repository).

Det var också tänkt att ha en domän-enhet User, och funktionalitet för registrering och inloggning för olika användar-konton. Men efter planering insåg jag att detta inte är nödvändigt för att uppfylla MVP-beskrivningen (minimal viable product) som anges i instruktionerna.

Under programmets utveckling gjordes många ändringar. Först var kassa och lagrings-vyerna uppdelate i olika WindowsForms, men efter mer experimentering så slutade det med att de olika vyerna representerades av flikar.

För övrigt var det inga stora problem med programmets utveckling. En sak som jag är ganska missnöjd med är MainForm-klassens storlek och komplexitet (som man tydligt kan se i klassdiagrammet). Klassens storlek kan argumenteras vara p.g.a. den mängd knappar och Form-elements, men jag hade gärna försökt modulera klassen till mindre delar om möjligt.

En annan sak som jag hade tänkt implementera (men inte hann) var att ha mer logik i Controller-klasserna, så att Forms-klasserna endast innehöll UI-funktioner. Detta för att försöka följa Separation of Concerns-principen.

# 5. Sammanfattning

Utveckling av programmet arbetades på ett visst antal timmar per dag, istället för att vänta och ta allting på en gång. Trots detta blev jag klar med programmet bara några dagar innan deadline. Programmets utveckling har varit mycket lärorikt, speciellt för användandet av Windows Forms. Den implementerade MVC-lösningen användes främst för att programmet är byggt runt en domän-enhet (Product), och jag tyckte därför denna lösning passade uppgiften. Inga alternativa mjukvaru-arkitekturer planerades att användas.

Mest tid lades på att hantera den datan från repository och controller-lagret i Forms-lagret. Det var ibland väldigt klurigt att få in datan i rätt format i de olika Forms-elementen, som t.ex. att lista Product-typ enum i en ComboBox, eller produkterna från ett visst kvitto i en ListView.

Generellt sätt är jag hyfsat nöjd med projektets gång.