**Databas-design.**

*customers*: Alla kunder har en profil med persondata samt lösenord för inloggning.   
*stock*\_*account*: Varje kund kan ha ett flera konton för aktiehandel. På kontot har man sin saldobalans.   
*account*\_type: Det finns olika typer av konton. I dagsläget så är det olika egenskaper för beskattning. Stocks: Detta är tabellen för de olika aktierna som går att handla med.   
*listing*: Det finns olika marknadsplatser där aktien är listad. Det som skiljer dem är hur banken väljer att ta betalt för courtaget. I dagsläget finns det ett fast pris och ett rörligt.   
*stocks\_to\_account*: Detta är ”kopplingstabellen” som vilken aktie och antalet till ett aktiekonto.   
*active\_orders*: Själva marknadsplatsen. Här sparas köp eller säljordern med det antal och pris som kunden lagt in.   
*stock\_transactions*: När en köporder och en säljorder matchar så sparas datan som en transaktion.

**Funktionalitet**: Programmets huvudsakliga syfte är att kunder ska kunna skapa ordrar och när en order är lagd så kollar programmet att en matchande order finns i databasen. Då skapas en transaktion samt aktier och pengar byter ägare.   
Programmet hämtar in aktuella order-priser samt kan även räkna ut aktiernas marknadsvärde baserat på den senaste slutförda transaktionen.

**Djupdyk**.

1. *Antalet på köp-order stämmer inte med antalet på säljordern.* Säg att en kund har en säljorder ute på 500 aktier. Köper någon endast 100 av dessa ska ordern inte makuleras. Utan den aktiva ordern som ligger kvar behöver ändras till 400. Enkel lösning men var inte helt nöjd med detta. Kunden hade ju faktiskt lagt en säljorder på 500 men nu står det bara 400. Hur jag löste detta var att programmet skapar en kopia av säljordern på 100 aktier men som direkt sätts till inaktiv. Aggregerar vi nu antalet aktier som en kund ville sälja vid exakt det klockslaget får vi nu rätt antal även om ordern blir uppdelad till flera kunder. Så här i efterhand så inser jag att en vidareutveckling på denna lösning skulle vara att ha en ny kolumn i active\_orders tabellen som heter original\_order\_id så att uppdelningen av en order kan spåras till en originalorder.

Utöver det så vill man nog sätta upp rensningsrutiner och se till att inte spara inaktiva orders i active\_orders tabellen för länge då Stock\_transactions skall ses som den tabell som vi ska långtidslagra data i.

1. *En rad per aktie i kopplingstabellen.* Då antalet av en aktie finns med som en kolumn i kopplingstabellen blir det knepigt kodmässigt att flytta aktier till en annan kund om de ligger utspridda på flera olika rader och behöver räknas ihop först. Detta löste jag genom att skriva en ganska enkel metod (StockTransactionManager rad 49) Då kollar programmet först ifall kunden redan har aktien på en rad i databasen och då istället ändrar antalet på raden.

**Vidareutveckling**

Databas: För att minimera risken för korrupt data skulle man kunna använda sig av transaktioner här.   
Då det är många steg och databas-anrop för att slutföra en order så blev det för stort. Jag skulle även kunnat utforska att använda triggerfunktioner för t.ex pengatransaktionen.

Kod/funktionalitet: Skapa funktionalitet för courtage-priser. Jag designade databasen att stödja både ett fast pris per order men även ett pris baserat på procent av summan. Skulle också behöva utveckla ett sätt att kunden endast betalar ett courtage per order. I nuläget om en kund med en stor order handlar med flera kunder med mindre ordrar så blir det ett courtage per varje uppdelad order. Bra för banken men mindre roligt för kunden. Som förslaget i djupdyk 1 så skulle man nog behövt ha koll på vilken som är originalordern och ta betalt när den är uppfylld.