

Projekt BiljettShoppen

Grupp1 som avser att bedömas utifrån VG

Tom Bergman, William Mossberg, Kasper Nordlinder, Oskar Johansson,

Alexander Norberg & Camilla Tillander

Innehåll

User Stories	3
Funktionella och icke funktionella krav	4
Funktionella krav	4
Icke funktionella krav	4
UML – klassdiagram	5
Use Case – diagram.....	6
Who, What, When, Where och Why	6
APIE	7
Abstraktion	7
Polymorfism.....	7
Arv	8
Inkapsling.....	8
SOLID.....	8
Single Responsibility Principle.....	8
Open/Closed Principle	9
Liskov Substitution Principle	9
Interface Segregation Principle	9
Dependency Inversion Principle	9
Datastrukturer	10
Algoritmer.....	10
Design Patterns	11

User Stories

Som **kund** vill jag att bokningarna är tidsbestämda och att köparen måste slutföra sin betalning inom 10 minuter, annars förlorar de sin bokning.

Som **kund** vill jag att bokningssystemet ska vara konfigurerbart och klara av hög belastning utan att sidan kraschar.

Som **kund** vill jag kunna se hur många biljetter det finns kvar för ett evenemang så jag vet hur försäljningen går.

Som **köpare** vill jag få en bekräftelse att köpet har genomförts.

Som **köpare** vill jag kunna köpa biljetter till ett evenemang så att jag kan delta.

Som **köpare** vill jag kunna se en karta över arenan med lediga och upptagna platser.

Som **köpare** vill jag kunna välja att betala via faktura så jag kan betala för biljetten vid ett senare tillfälle.

Som **köpare** vill jag kunna se vilka evenemang som finns på olika arenor och vilka tider som dem händer, så jag kan bestämma mig vilket evenemang jag vill gå på.

Som **köpare** vill jag kunna se hur länge min bokning är reserverad innan jag behöver betala.

Som **administratör** vill jag kunna använda tjänsten för att skapa ett event för en specifik arena och tid.

Som **administratör** vill jag registrera en nybyggd arena till tjänsten så vi kan börja sälja biljetter till event i den nya arenan.

Som **administratör** vill jag kunna konfigurera platsuppsättningen för en specifik arena.

Som **administratör** vill jag kunna uppdatera ett evenemang.

Som **administratör** vill jag kunna ta bort ett evenemang.

Som **administratör** vill jag kunna ange hur många ingångar som är aktiva för eventet, så **köpare** vet vilken ingång de ska använda.

Funktionella och icke funktionella krav

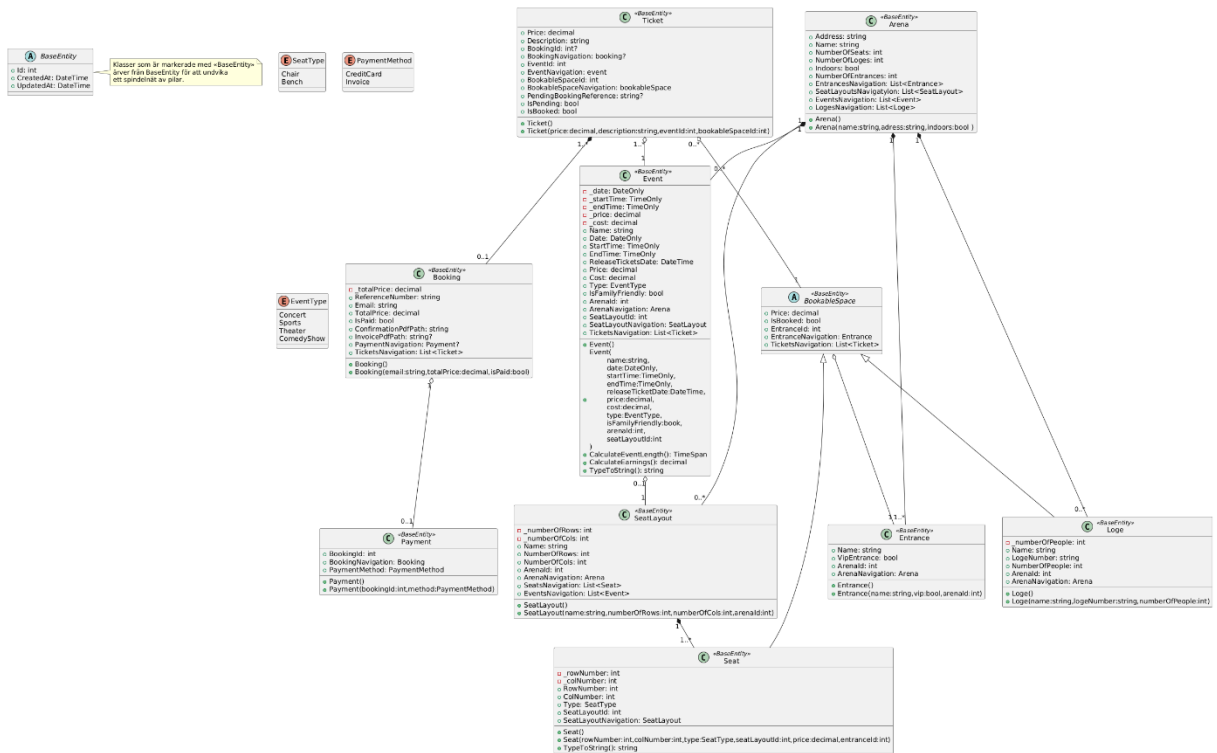
Funktionella krav

- Systemet ska kunna lagra information om evenemang.
- Systemet ska kunna hantera ett fast antal biljetter per evenemang.
- Systemet ska endast tillåta försäljning av biljetter efter ett visst datum och klockslag.
- Systemet ska kunna visa arenans sittplatser grafiskt.
- Systemet ska visa lediga respektive upptagna platser.
- Systemet ska stödja tre typer av sittplatser.
- Systemet ska kunna hantera olika prissättningar per sittplatstyp.
- En användare ska endast kunna boka 1 – 5 biljetter för vanliga evenemang.
- Systemet ska tillåta att boka fler än 5 biljetter vid familjeevenemang.
- Om användaren inte slutför köpet inom 10 minuter ska platserna automatiskt frigöras.
- När en bokning påbörjas ska platserna reserveras i 10 minuter.
- Systemet ska erbjuda betalning direkt med kort eller via faktura.
- Systemet ska validera kortuppgifterna för att säkerställa att de är äkta.
- Systemet ska validera att betalningen genomförts innan bokningen bekräftas.
- Systemet ska generera och skicka en bekräftelse via E-post vid lyckad bokning.
- Systemet ska beräkna pris utifrån sittplatstyp, tidpunkt för bokning och evenemangstips.
- Administratörer ska kunna:
 - Skapa, uppdatera och ta bort evenemang.
 - Skapa, uppdatera och ta bort arena.
 - Skapa ny stolslayout för en arena.

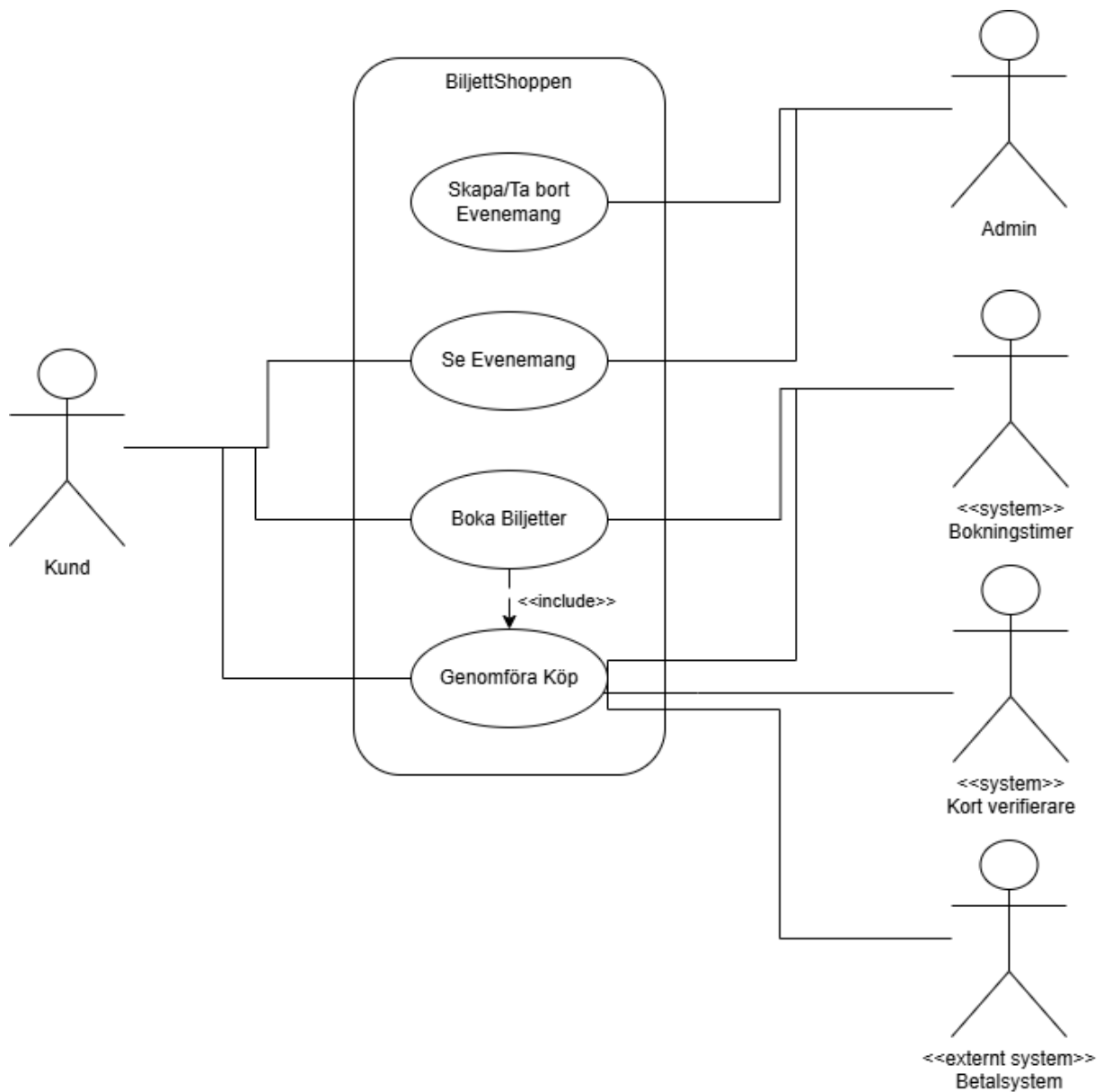
Icke funktionella krav

- Systemet ska klara av hög belastning utan att krascha.
- Systemet ska följa GDPR för hantering av personuppgifter.
- Endast behöriga ska ha tillgång till administratörspanelen.
- Hemsidans gränssnitt ska vara lättanvänt för nya användare.
- Felmeddelanden ska vara tydliga.
- Koden ska vara utvecklingsbar och följa god objektorienterad design.

UML – klassdiagram



Use Case – diagram



Who, What, When, Where och Why

Programmet som byggts har byggts till en kund, kunden hade begärt ett biljettbokningssystem med funktionalitet som att boka biljett, kunna betala biljett, biljetthantering, ha en adminpanel där man ska kunna skapa arenor, hantera evenemang. Kunden har satt sin deadline klockan 16:00 den 5 november 2025. Programmet är skapat som en webbapplikation och kommer köras på en hemsida. Varför programmet skapats är för att kunden efterfrågan en effektivare lösning på ett biljettbokningssystem som håller högre prestanda och enkelt kan utvecklas och byggas ut.

APIE

Abstraktion

Genom att använda `BaseEntity` som en abstrakt klass kan man definiera gemensamma egenskaper och metoder som entitetsklasserna ärver ifrån. Det gör att man slipper upprepa samma kod i subklasserna. Klassen används som grund för de entiteter som ska motsvara tabellrader i databasen.

Subklasserna ärver alla tillgängliga medlemmar från `BaseEntity`, men använder endast det som är relevant för deras egen funktion. `BaseEntity` hjälper på så sätt till att organisera och standardisera koden, men används indirekt och är dold bakom de specifika entitetsklasserna.

Mycket av det vi använder kommer från externa bibliotek och ramverk, till exempel från Microsoft, för att vår kod ska fungera ihop med deras struktur. En klass som `BookingTimer` ärver från en abstrakt basklass (`BackgroundService`) i Microsofts bibliotek som `Microsoft.Extensions.Hosting`. Vilket är nödvändigt för att timern ska kunna köras som en `BackgroundService` i web appen. Eller våran `ApplicationDbContext` som ärver från `IdentityDbContext<IdentityUser>` som kommer från `Microsoft.AspNetCore.Identity.EntityFrameworkCore` detta gör att vi kan bygga upp en databas med Entity Framework Core och få inloggnings funktionalitet på köpet.

Polymorfism

I denna applikation används `BookableSpace` som är en abstrakt klass, vilket definierar de metoder och egenskaper som samtliga bokningsbara entiteter ska implementera. Klasserna `Loge` och `Seat` implementerar detta och tillhandahåller sina respektive versioner av den gemensamma funktionaliteten.

Anropar metoderna via `BookableSpace` utan att behöva känna till den underliggande objekttypen. Om objektet är av typen `Seat` används dess specifika implementation, exempelvis för att markera om en plats är upptagen eller för att avgöra om den representerar en stol eller en bänk.

Genom denna användning av polymorfism kan biljetthanteringen förenklas, eftersom koden inte längre behöver förlita sig på villkorssatser såsom `if` eller `switch` för att särskilja olika objekttyper. I stället avgörs beteendet dynamiskt vid körning beroende på objektets faktiska typ. Detta bidrar till en mer flexibel och skalbar kodstruktur som är enklare att underhålla och vidareutveckla.

Arv

Genom att de klasser som ska göras om för att passa databaser ärver från `BaseEntity` minskar behovet av upprepad kod. Den gemensamma funktionaliteten definieras endast en gång i `BaseEntity`, och alla ärvande klasser får tillgång till denna automatiskt utan att den behöver åter implementeras. I de flesta fallen använder vi komposition över arv, då det bidrar till en mer flexibel lösning. Komposition minskar beroendet mellan klasser, vilket i sin tur stärker projektets koppling till single responsibility principle.

De lyxloger som planeras att byggas finns redan fördefinierade i systemet och databasen, redo att aktiveras när de är färdigplanerade. När de ärver de från `BookableSpace` och integreras automatiskt i systemet, vilket möjliggör omedelbar försäljning så snart de tas i bruk.

Inkapsling

Inkapsling ger ett extra skydd för objektens tillstånd och ger full kontroll över hur och när data ändras. Många properties är publika, särskilt i klasser som ärver från `BaseEntity`, eftersom dessa motsvarar tabeller i databasen och dessa kan behöva uppdateras. Däremot kan subclasser av `BaseEntity` ha private properties (backing fields) för att kunna ange logik i våra setters för att till exempel förhindra att starttiden infaller efter sluttiden för ett evenemang, eller att ett biljettpreis inte kan vara negativt. Övriga klasser använder för det mesta privata fält och publika properties endast där det är nödvändigt. Det gör det också möjligt att ändra den interna implementationen av en klass utan att påverka den externa koden, så länge de publika metoderna och egenskaperna förblir oförändrade. Detta bidrar till ökad flexibilitet och bättre underhållbarhet av koden.

SOLID

Single Responsibility Principle

En klass ska bara ha ett jobb eller ansvarsområde. Om varje klass bara gör en sak så blir det:

- Enklare att felsöka: om ett fel uppstår till exempel att fel antal biljetter visas så vet vi vart i koden man ska titta.
- Lätt att bygga ut: om man vill ändra hur en arena skapas, gör man det lätt i `CreateArenaHandler`. Risken att ändringen förstör något annat är lika med noll eftersom klasserna är helt separata.
- Hög testbarhet: Varje klass ansvarsområde kan testas separat. Vilket gör det enkelt att verifiera att varje del av systemet funkar på ett korrekt sätt.

Open/Closed Principle

Med open/closed principle menas det att programmet ska vara byggt på det sättet att de är öppet för utveckling men stängt för modifiering. Ett exempel hur vi har använt det är genom att använda decorator pattern för att prissätta biljetterna. Eftersom om du vill ändra kraven för prissättningen så gör du de genom att lägga till nya klasser och inte ändra på de befintliga.

Liskov Substitution Principle

Objekt av en basklass ska kunna ersättas med objekt av dess subklass utan att programmets funktionalitet påverkas. Hur vi använder liskov substitution principle i vårt program är exempelvis att `Loge` – klassen kan ersätta `BookableSpace` utan att programmet skulle förstöras.

Interface Segregation Principle

Klasser ska inte tvingas implementera metoder de inte använder. Ett exempel i vår kod är att `BookableSpace` hanterar endast bokningsbarheten och inte betalningen eller evenemangshanteringen. Våra interfaces är väldigt specifika och har endast ett ansvarsområde så följer de i sig single responsibility principle vilket gör att interface segregation principle följs väldigt enkelt. Det gör att klasserna kan implementera hela interfacet och allt i interfacet används sedan av den klassen.

Dependency Inversion Principle

Moduler på hög nivå ska inte vara beroende av moduler på lägre nivå. Båda ska i stället vara beroende av abstraktioner som exempelvis ett interface. Abstraktioner ska inte vara beroende av konkreta implementationer. De konkreta implementationerna ska vara beroende av abstraktioner. Ett exempel på vart vi använder dependency inversion principle är att våra handlers inte har någon implementation utan det är mediator som har hand om den. Programmet är även byggt på ASP.NET Core som använder sig av dependency injection, som i sin tur följer dependency inversion principle. Eftersom de beroenden som en klass behöver specificeras som privata attribut och registreras i konstruktorn. Dessa klasser registreras med hjälp av ett interface, vilket gör att klasserna inte är beroende av en implementation utan i stället bara berättar vilket interface de behöver för att utföra sin uppgift. ASP.NET Cores Dependency Injection hittar sedan dessa implementationer i bakgrunden automatiskt utifrån de klasser man har registrerat i sin DI container. Detta gör att projektet följer sig av Dependency Inversion rakt igenom.

Datastrukturer

Projektet har utförts med hjälp av Git och ett Github repository. Dessa gör det enkelt att arbeta parallellt och backa upp koden till tidigare versioner. Pull requests ökar även kvaliteten på den kod som används i slutprodukten, då koden kan granskas innan den slås ihop med projektet (main).

`BookingTimer` skapades för att hantera tidsbegränsning vid bokning. Bokningar hanteras i samma följd som de kommer in, bokningarna lagras i en `ConcurrentDictionary<string, Booking>`. Denna datastruktur är vald på grund av att den är byggd för att hantera att flera trådar vill ha åtkomst till den samtidigt. Detta är viktigt eftersom boknings processen måste kunna hantera hög trafik. Algoritmen för att rensa bokningar är $O(n)$ eftersom vi måste gå igenom hela datastrukturen för att se vilka bokningar som ska rensas bort, eftersom vi inte kan garantera att bokningarna ligger i ordning på grund av nätverks delay kan vi inte hoppa ur loopen tidigt. Däremot med denna datastruktur får vi $o(1)$ för att hitta en bokning vilket gör att den är väldigt effektiv på att plocka ur bokningar som har betalats. En dictionary behöver inte heller skifta sina element när de plockas in och tas bort nya.

Listor används för att lagra objekt i en ordnad struktur. I projektet används listor i till exempel `BrowseAllEventsHandler`, där det kan vara viktigt att gå igenom hela listan. Det är även en flexibel datastruktur som dynamiskt kan öka respektive minska vid behov.

Vi har även skapat upp en egen datastruktur `PaginatedList<T>` vilket skapar upp en lista med paginerings funktionalitet. För att förenkla implementation av paginering på hemsidan. Denna data struktur hanterar utplockning av rätt objekt beroende på vilket sidnummer man har angett automatiskt. Men även lite ytterligare information som till exempel total antal objekt i tabellen, hur många objekt som visas för denna sida, och totalt antal sidor.

Enums är fördefinierade värden, exempelvis status (Pending, Confirmed, Cancelled) Enums är minneseffektiva och därför användbara för värden som går att fördefiniera. Enums används i Enums-foldern och används bland annat för `EventType` och `SeatType`.

Algoritmer

En algoritm kan jämföras med ett recept. Algoritmer är det som urskiljer ett effektivt och välfungerande program mot ett betydligt mindre effektivt.

När kunden vill se platser för evenemang så räcker det inte att ha en osorterad lista, därför använder vi i `ViewSeatsHandler` en sorteringsalgoritm som är inbyggd i C# LINQ för att sortera platserna. Detta var också nödvändigt eftersom om en plats uppdateras i databasen ändras ordningen i databasen vilket gör att de alltid måste sorteras vid inhämtning.

```
.Include(e => e.SeatLayoutNavigation)
```

```
.ThenInclude(sl => sl.SeatsNavigation)

.OrderBy(s => s.RowNumber)           // Sorterar radnummer

.ThenBy(s => s.ColNumber)           // Sorterar platsnummer
```

När vi ska ta reda på hur många biljetter som finns kvar så måste vi räkna ut det med hjälp av en algoritm då vi inte lagrar hur många biljetter som finns kvar. Detta gör vi i `RemainingTicketsHandler`

- Vi hämtar det specifika evenemanget (`theEvent`) och relevanta data som `SeatLayoutNavigation` och `SeatsNavigation`.
- Sen beräknar vi den totala kapaciteten med hjälp av `theEvent.SeatLayoutNavigation.SeatsNavigation.Count()`
- Därefter beräknar vi antalet sålda biljetter genom att köra en `CountAsync()` operation på `Tickets` tabellen, där vi filtrerar på `EventId` och att `BookingId != null`
- Slutligen så subtraherar vi `soldTickets` från `totalCapacity` för att få fram värdet för `remainingTickets`

Vi har även använt oss av Luhns algoritm vilket är en algoritm för att enkelt kunna validera om ett kortnummer är korrekt eller inte. Den funkar genom att ta det sista numret i kreditkortsnumret. Du ska sedan dubbla värdet av det sista och sen fortsätta vartannat nummer och dessa ska också dubblas. Om ett av dessa nummer som dubblas skulle vara högre än 9, Så summeras siffrorna av produkten, till exempel om det skulle bli 12, så tar vi $1 + 2 = 3$ om det skulle vara 15 så blir det $1 + 5 = 6$. Sedan ska alla nummer summeras. Om vi sedan tar (summan % 10) och svaret är 0, betyder det att vi har ett korrekt kortnummer. Viktigt att komma ihåg är att detta bara beräknar om kortet möjligtvis är ett riktigt kortnummer. Algoritmen garanterar såklart inte att kortet faktiskt existerar eller har möjlighet att betala. Men vi kan åtminstone veta att de användaren har angett bör vara korrekt och gör att vi kan gå vidare i betalningsprocessen.

Design Patterns

Genom att hålla oss till olika designmönster så underlättar det att underhålla koden över tid eller om någon ny ska sätta sig in i projektet. Det är en viktig del när det kommer till att lämna över ett projekt till en kund som sedan ska vidareutveckla programmet med sitt team. Några designmönster vi har använt oss av är Mediator, Command Query Responsibility Segregation (CQRS) och Decorator Pattern.

- Mediator har vi använt då det är ett välkänt designmönster som hjälper dig att minska oordnade beroenden mellan olika objekt. Det mönstret gör är att begränsa direktkommunikationen mellan olika objekt och tvingar de att samarbeta via ett förmedlarobjekt.

- CQRS har vi använt då det är ett väldigt effektivt mönster för att skapa skalbara mjukvarusystem. Det separerar ansvaret på att läsa och skriva data i olika databaser. Det medför även att det är väldigt enkelt och smidigt att jobba flera i samma projekt och väldigt nära varandra i projektet. Eftersom vi hade en tajt deadline och ett krav att kunden skulle ta över projektet och fortsätta utveckla de så passar CQRS perfekt för att göra det så enkelt som möjligt att fortsätta utveckla det men även göra det simpelt att arbeta flera samtidigt i projektet. Eftersom alla features då delas upp i sin egen klass och en persons ändringar kommer inte påverka någon annans ändringar.
- Decorator Pattern är ett mönster som låter utvecklaren lägga till ny funktionalitet till ett objekt dynamiskt. Vi har använt de i vår prissättning för att sätta pris på biljetter utifrån flera olika parametrar dynamiskt. Det gör att vi enkelt kan lägga till nya sittplatser exempelvis utan att behöva ändra något annat. Men också nya prissättnings krav kan väldigt enkelt läggas till eller tas bort utan att behöva modifiera befintlig kod.