PG3302-1 23H Software Design

Pizzabestillingsapplikasjon

av

Lagd av Danait, Line, Emilie og Anna.

Innholdsfortegnelse

Introduksjon

Introduksjon til programmet

Vi vil gjerne introdusere vår pizza applikasjon som er skapt med vektlegging på praktisk bruk og relevans i dagens samfunn. Med dette programmet kan du enkelt nyte en deilig pizza gjennom take away bestilling. Men frykt ikke, ved ytterligere vekst kan vi ta til vurdering å utvide til levering som en fremtidig mulighet.

Målsetting med programmet

Målet med pizza-applikasjonen vår er å gi en rask og problemfri brukeropplevelse. Den er designet for å være lett å navigere, slik at den er tilrettelagt for brukere i alle aldre. Vi har lagt vekt på å bruke god software design i programmet vårt, slik at det er mulighet for å utvide programmet uten å påvirke eksisterende funksjonalitet.

Beskrivelse av programmet

Vi har laget en applikasjon som lar brukeren bestille pizza som kan hentes på restauranten. Først får brukeren spørsmål om å logge inn eller legge til en ny bruker. Dersom brukeren forsøker å logge inn, men ikke er registrert i databasen, må hen registrere en bruker. Siden restauranten kun tilbyr at kunden henter selv, trenger brukeren kun å legge inn navn og telefonnummer. Når brukeren oppretter en ny bruker, får hen beskjed om å bekrefte at opplysningene er riktige. Dersom de ikke er det, får hen mulighet til å endre informasjonen. Etter dette er brukeren logget inn.

Nå får brukeren en meny med mulighet til å bestille pizza, logge ut eller håndtere brukerinformasjonen (endre brukerinformasjon eller slette brukeren fra databasen). Dersom brukeren velger å logge ut, avsluttes programmet. Dersom brukeren velger å slette brukeren sin, slettes brukeren fra tabellen og programmet avsluttes. Dersom hen velger å bestille pizza, dukker det opp en meny (ShowMenu) fra CartMenu klassen hvor brukeren kan velge mellom å se handlekurven sin, legge til ny pizza, fjerne pizza, endre mengden eller avslutte programmet. Hvis brukeren velger å legge til en ny pizza så starter menydelen av programmet. Her hentes pizzaene fra pizzatabellen og blir lagt til i en liste. Når brukeren har valgt hvilken type pizza de vil ha og hvilken topping de ønsker, så får man opp ShowMenu igjen hvor brukeren kan se handlekurven sin, endre mengde pizza, fjerne pizza, slette en pizza eller legge til en ny pizza. Hvis brukeren velger å legge til samme type pizza, så blir pizzaen lagt til i quantity og prisen dobles.

Til slutt får brukeren opp menyen igjen (showMenu) og kan velge å se orderen. Da sendes bestillingen inn, og kunden får en oppsummering av orderen. Her har vi lagt til funksjonalitet som gjør at en pizza tar 10 minutt å lage, ettersom vi er en liten bedrift. Så hvis brukeren har to pizzaer, så vil dette ta 20 minutter osv. Under ligger det et bilde eksempel fra en kunde som kjøper 3 pizzaer 05:30 og kan hente pizzaen sin klokken 06:00, 30 minutter senere.

```
Choose an action:
1. View shopping cart
2. Add a new pizza
3. Remove a pizza from the shopping cart
4. Change the quantity of a pizza in the shopping cart
5. Send order
6. Exit
5

Thank you for ordering from Pizza Factory! Here is your receipt:
Name: Test123

Your order:
1 Margarita
1 Hawaiian dream
1 Vegan deluxe

Total sum: 377 kr
Your order is ready for pick up at 30.11.2023 06:00:11
Welcome back another time!
```

Prosess

Idémyldring

Første dagen møttes vi fysisk, leste nøye gjennom oppgaveteksten og hadde idémyldring. Da kom vi frem til disse alternativene som tema til programmet vårt:

- Nettbutikk
- Pizzarestaurant, bestill mat
- Aksjon - bruke multithreading

Vi lente mot noe i form av butikk, da det virket underholdende og hadde stort potensial for utvidelse. Vi ble raskt enige om å lage en pizzabestillings-applikasjon. Da hadde vi en diskusjon om det skulle være sit down restaurant eller takeaway, og om det skulle være fokus på booking eller bestillingsdelen. Vi ble enige om å fokusere på bestillingsdelen av programmet. Vi gikk gjennom oppbyggingen av hva vi ville inkludere i programmet, og hva vi ville ha med i databasen.

Start-prosess

Så var det frem med tegnebrettet for å planlegge koden ved hjelp av klassediagrammer. Vi la en plan for å møtes fysisk ukentlig, ble enige om å bruke discord-gruppe for å kommunisere utenom og bruke github til versjonskontroll. For hver uke delegerte vi konkrete oppgaver som skulle gjennomføres til neste møte. De gangene vi møttes brukte vi til å diskutere eventuelle endringer i planen, komme med nye idéer og hjelpe hverandre med kode der vi stod fast. Vi har primært hatt ansvar for en eller flere klasser hver.

Utkast for oppbyggingen programmet

Pizzarestaurant (bestille pizza)

Vi lagde et lite oppsett på hvordan vi tenkte programmet vårt skulle se ut:

- logge inn / registrere hvem som skal bestille
- se meny
- velge pizza, størrelse og antall
- fjerne/legge til topping
- velge tidspunkt for levering (lage køsystem)
    - sjekke om pizzaen blir levert innenfor åpningstiden
    - legge til lock så ikke to bestiller på samme tidspunkt
- betaling
- håndtere at flere bestiller på samme tidspunkt (lock)
- antar at bruker kun legge inn én adresse?

Databaser

- Customer
    - kundeId
    - navn
    - adresse
    - telefonnr.
- Bestillingshistorikk?
    - knyttet til kunde
    - ordreId
    - readyToPickup
- Pizza
    - pizzaId
    - navn
    - beskrivelse
    - price

Oppgavefordeling og dager vi møttes:

Tors 26.10

- **Person 1**
    - klassen ordre
- **Person 2**
    - klassen kunder
    - laget database
    - legge inn kunde i databasen
- **Person 3**
    - klassen seeMenu og UML
- **Person 4**
    - klassen seeMenu og UML

Tors 2.11

- **Person 1**
  - Klassen Ordre
  - Køsystem
- **Person 2**
  - lage databaseoppsett
  - entities
  - Login
    - lage egen metode for å sjekke om brukeren finnes i databasen
    - sjekke at brukeren legger inn valid input
  - HandleCustomer
    - kunne slette bruker
- **Person 3**
  - Klassene MenuCategory (superklasse)
    - lage en subklasse Pizza (subklasse)
- **Person 4**
  - ShoppingCart

Tors 9.11

- **Person 1**
  - ferdigstille klassen PizzaQueue
    - hente antall fra userInput
    - beregne ny tid
  - SOLID-prinsippene
    ++
- **Person 2**
  - HandleOrdre
    - skrive ut bestillingen (hva som er bestilt, kunde, hentetid)
    - legger ordre til tabellen Order
  - HandleCustomer
    - legge inn telefonnummer med 8 siffer?
  - fikse loopen i programmet
  - databaser - legge til andre anotasjoner?
  - Layering
  - UML i visual

  - Login
    - navn må være bokstaver
    - telefonnummer må være tall
  - HandleCustomer
    - EditCustomer
      - håndtere at brukeren legger inn bokstav som menyalternativ og telefonnummer
- **Person 3**
  - Menu-klassene
    - legge til en ny meny (subklasse)
    - legge til størrelse p[pizza

    - Decorating
      ++
- **Person 4**
    - klassen chart
    - opprette testing
      ++

Tors 16.11

- ca ferdigstilt eksamen

uke 47

- fokus på Frontend eksamesøving

Fre 24.11

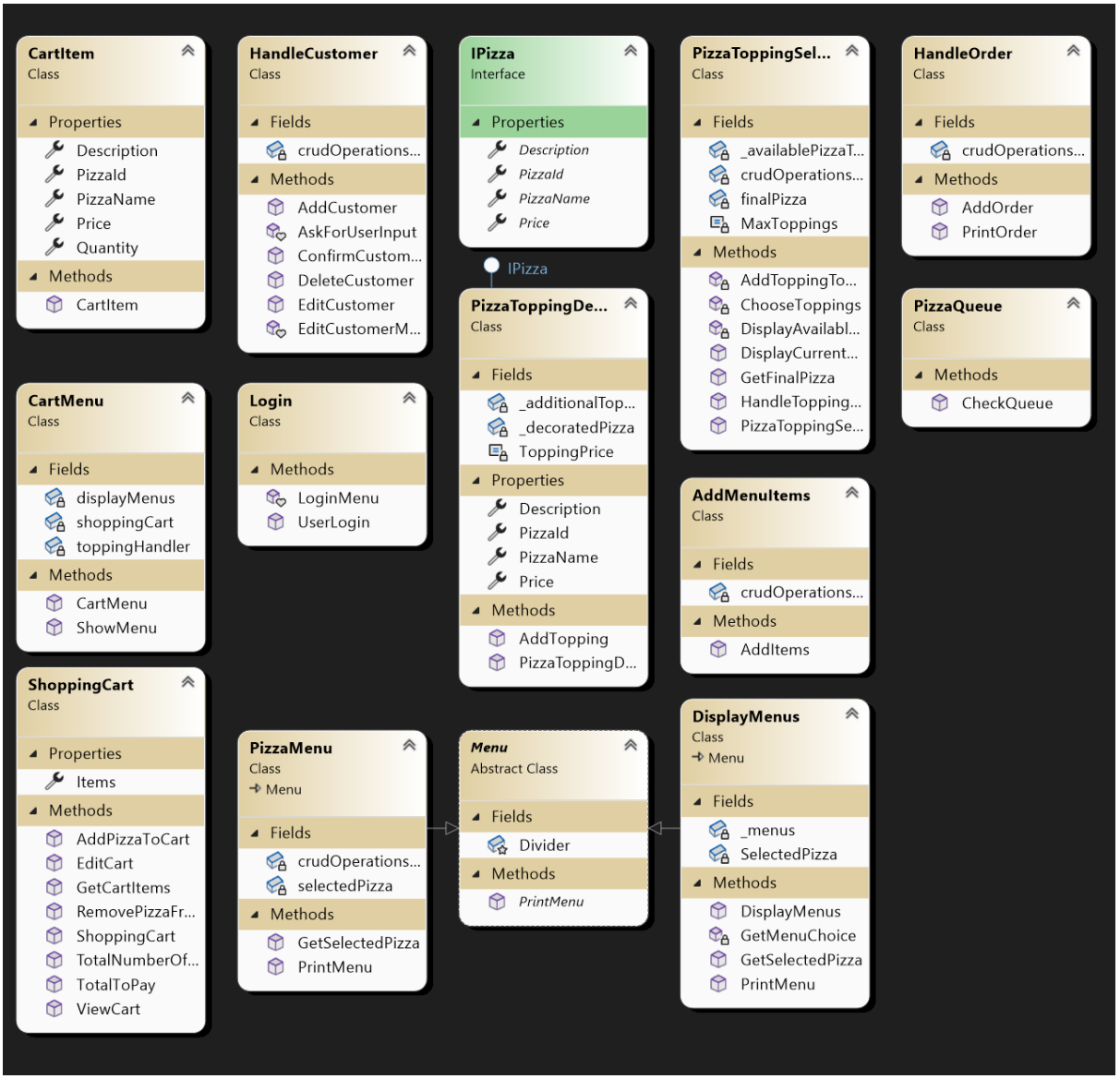
- testing
- flette sammen klassene
- dokumentasjon
- kommentere egen kode i visual studio
- UML (klasser, database)

UML

UML er et standardisert modelleringsspråk som kan brukes til å lage skisser av et program. Disse skissene ble endret på underveis i prosessen da vi kom på ny funksjonalitet vi ønsket å legge til applikasjonen.

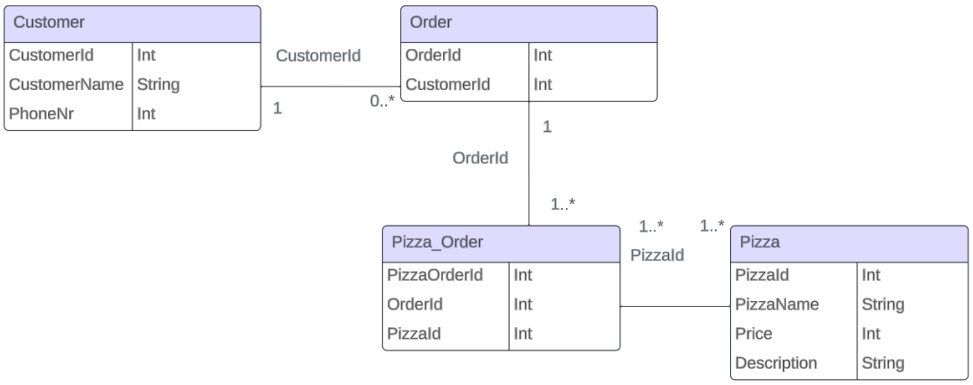
Klassediagram

Vi tegnet klassediagram i oppstartsfasen for å se hvilke metoder vi tenkte var aktuelle å implementere i applikasjonen vår. Klassediagrammet ligger som en fil i prosjektet vårt, som også kan sees på bildet under.



ER-diagram

Vi benyttet også UML for å lage en skisse av databasen, og til dette formålet brukte vi Lucidchart. Vi tok denne beslutningen basert på vår erfaring med programmet fra tidligere år, og følte oss komfortable med å anvende det. Dette er en entitet-relasjonsmodell som beskriver strukturen til vår database.



Tabellene i databasen og relasjonen mellom dem

- Customer
  Tabellen representerer kundene, og her blir kundens informasjon lagret med autoinkremert customerId, customerName og phoneNr. Her er customerId primærnøkkelen, og det er helt unikt for hver kunde.

- Order
  Tabellen representerer bestillingene. OrderId er primærnøkkelen som er unik for hver bestilling. Fremmednøkkelsen CustomerId som refererer til CustomerId i Customer-tabellen. På denne måten har vi etablert en relasjon mellom customer og order.

  Forholdet mellom tabellene customer og order er at en kunde kan ha null til mange bestillinger, mens en bestilling kan bare ha en kunde.

- Pizza_Order
  Dette er en koblingstabell som inneholder orderId, PizzaId og PizzaOrderId, hvor sistnevnte er primærnøkkelen. Denne koblingstabellen brukes for å koble bestillinger med spesifikke pizzatyper.

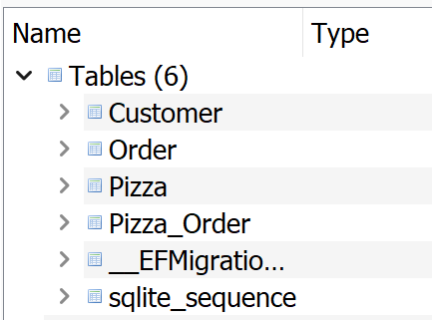
  Vi rakk ikke å ta denne tabellen i bruk i koden vår, så den blir ikke oppdatert på noe tidspunkt.

  Forholdet mellom tabellene order og pizza_order er at en bestilling kan inneholde en til mange pizzatyper, men en pizza bestilling kan bare ha en bestilling(kvittering).

  Forholdet mellom pizza_order og pizza er at en bestilling (pizza_order) kan ha en til mange pizzatyper. En pizza kan også ha en til mange bestillinger.

- Pizza

Pizza representerer de ulike pizzaene vi har tilgjengelig. Her får brukeren valg om hvilken pizza de ønsker. Primærnøkkelen her er PizzaId.



Prosessen over hvordan vi så for oss bestillingen ville skje:
Kunden legger inn en bestilling på programmet vårt ved hjelp av telefonnummer, navn og id. De velger hvilken pizza de vil ha ved hjelp av id/number. Etter de har valgt får de et unikt bestillingsnummer. Bestillingene vil dermed bli lagret i order tabellen.

Versjonskontroll og par-programmering

Vi benyttet Github som vår plattform for versjonskontroll for å samarbeide effektivt på det samme prosjektet. Vi lagde ulike branches og utførte push til main når vi var tilfredsstilt med de ulike delene av programmet. I løpet av prosjektet støtte en av oss på tekniske utfordringer med macbook. Det sannsynligvis skyldes at Visual Studio er planlagt å bli avviklet for Mac-plattformen, men en god løsning på dette ble parprogrammering. Dette tillot grundig diskusjon og sikrere kode, noe som styrket vår sikkerhet og tillit til koden.

Vi brukte også bare grener (branches) for koding, og vi samlet endringene til hovedgrenen (main) når vi var klare. Dette gjorde arbeidet vårt enklere og mer pålitelig. Hvis vi jobbet i separate grener, kunne vi til og med kopiere grenen og jobbe individuelt på egne datamaskiner, spesielt når vi ikke jobbet sammen på samme sted.

Pensum

SOLID

S. *Single-responsibility principle*
Handler om ansvarsbegrensning. Dette vises i koden ved at hver klasse og metode har sitt ansvarsområde, og kun en oppgave.

I pizzaOrderingApp finnes klassen “program” som ikke inneholder metoder for beregning, men kaller på metoder fra de andre klassene (arbeidsutdeling)**.** Dette er gjort i flere av klassene og fører til at hver klasse omhandler et enkelt tema, med metoder som hører til. Ved bruk av slik spesialisering blir strukturen i koden lettere å forstå, endre og teste.

Dette prinsippet har vi fulgt gjennom programmet vårt. I for eksempel cartHandler har ShoppingCart klassen handlinger relatert til selve handlekurven, og CartMenu klassen har i oppgave å vise menyen til handlekurven. CartItem klassen brukes til å representere individuelle elementer i handlekurven. PizzaId, quantity, pizzaName og price er properties i CartItem klassen, og disse beskriver detaljene til hver vare(element) som legges i handlekurven.

O. *Open Closed principle*
Klassene skal være åpne for utvidelser, men lukket for modifikasjoner.

Open Closed prinsippet blir fulgt i mesteparten av koden. Arv og underklasser er brukt slik at man ikke trenger å endre eksisterende kode om man ønsker å utvide prosjektet. Menu klassen benytter dette, og man kan lage flere subclasser om man ønsker å legge til nye menyer feks drikkemeny eller lignende. Pizza klassen er også åpen for utvidelser, samtidig som den er lukket for direkte endringer, noe som er i tråd med SOLID prinsippene.

Prinsippet blir derfor fulgt i disse klassene, men for at brukeren skal kunne se den nye menyen som har blitt lagt til og samtidig følge SOLID er man nødt til å endre den eksisterende koden.

Console.WriteLine er hardkodet i DisplayMenu klassen (se bilde under), og vil derfor ikke følge open/closed prinsippet. For å forbedre koden slik at den også følger SOLID bør man lagre menyene i en database tabell og benytte seg av en “for-each loop” som printer ut alle menyene som er lagret i databasen. Dette er grunnen for at vi valgte å ha en DisplayMenu klasse, da det skal være lettest mulig å implementere en ny menu (feks. brus meny, fingermat meny osv). Disse må da bli lagt ny klasse til som arver fra Menu klassen for at dem skal bli printet ut riktig.

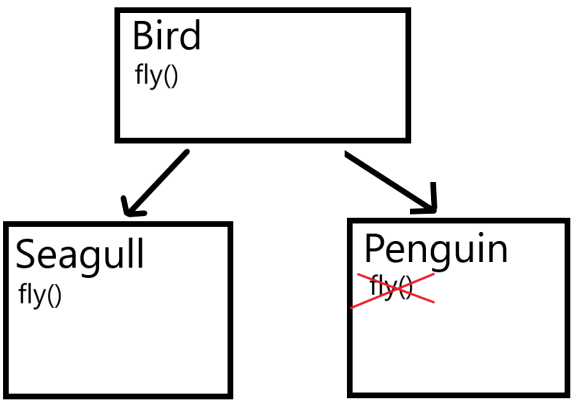
```
public override void PrintMenu()
{

    Console.WriteLine($"\n{Divider}\nSelect a menu:\n{Divider}");
    Console.WriteLine("1. Pizza");
```

Vi valgte å beholde PrintMenu, til tross for at den er hardkodet, ettersom det gir oss en mer lesbar kode som passer bra til mindre komplekse programmer. Vi prioriterte heller å utvikle funksjonalitet på bestillingsprosessen. Men dette er noe som kan oppfattes som en potensiell svakhet. En alternativ tilnærming kunne vært å ha menyen lagret i en egen database. Noe som ville ført til bedre skalerbarhet og vedlikehold. Imidlertid må vi vurdere at når vi legger til en pizza, skal flere toppinger også legges til, og denne sammenhengen vises i den nåværende tilnærmingen.

L. *Liskov Substitution Principle*

Objekter skal kunne bli erstattet med subtypene deres uten å endre de ønskelige egenskapene til programmet. Koden har ingen klasser som har subclasser som ikke bruker alle metodene fra klassen den arver fra. PizzaMenu arver fra menu, men bruker alle metodene i menu, i liket med alle subklassene som det hadde vært naturlig å lage om man ønsket å utvide programmet. Det er ikke nødvendig for en sub klasse å override en metode i klassen den arver fra (feks hvis en metode var logisk feil å ha i en sub klasse (se bilde under) hadde man vært nødt til å bruke override og brutt prinsippet). Programmet bryter dermed ikke Liskov Substitution Prinsippet.



I. *Interface Segregation Principle*

Klienter burde ikke bli tvunget til å bruke metoder som de ikke bruker. Meningen med prinsippet er å unngå at interfacer inneholder metoder som de ikke bruker eller trenger, men heller lage interfacer som er spesialiserte, slik at klientene bare implemeterer metodene som er relevante. Dette gjør arkitekturen mer fleksibel og lettere å endre eller utvide koden. Dette prinsippet blir også fulgt siden det ikke er noen interfacer med for mange metoder.

Ved å implementere IPizza grensesnittet sikret vi at klassene kun måtte forholde seg til de metodene som er nødvendige for deres spesifikke rolle.

D. *Dependency Inversion Principle.*
High-level moduler (business logic) og low-level moduler (database og andre eksterne ressurser) skal ikke være avhengig av hverandre, men heller bruke abstraksjon. Dette gir programmet mange av de samme fordelene som ved bruk av interface segregation principle i tillegg til å gjøre testing lettere. Her kunne vi også forbedret koden vår ved å ta i bruk interfaces for å implementere database. Slik at koden hadde vært mer fleksibel om man skulle endret databasetype til feks SQL Server, Cosmos DB, Postgres eller en annen.

Design patterns

Vi har tatt i bruk decorator-pattern. Dette er et pattern som gir oss evnen til å etterligne arv ved kjøretid. Med andre ord gir det oss muligheten til å legge til ekstra egenskaper og funksjoner mens programmet kjører, uten å måtte endre på hele koden på forhånd. Dette implementerte vi i PizzaToppingDecorator klassen, og det ga oss evnen til å legge til ekstra funksjonalitet i IPizza objekter.
kilde: PG3302_07_Multithreading_DP-Factory-Decorator.pdf. Sandnes. 2023

Properties er en viktig del av denne implementeringen. I vår PizzaToppingDecorator klasse utnytter vi properties for å definere detaljer om den dekorerte pizzaen. PizzaId er propertien som gir tilgang til den opprinnelige pizzaen, og henter deretter verdier fra IPizza objektet. PizzaName referer også til IPizza objektet og gir oss en tilgang på navnet til pizzaen. Price tar grunnprisen fra det dekorerte IPizza objektet og legger til prisen for de ekstra toppingene. Description kombinerer en beskrivelse av den opprinnelige pizzaen med en liste over de ekstra toppingene.

Dette gjorde at vi hadde muligheten til å tilpasse pizza objekter med forskjellige toppings, og oppdatere pris, mens vi samtidig opprettholdt fleksibilitet i koden, og bevarte strukturen til den opprinnelige Pizza klassen.

Multithreading

Vi valgte å ikke implementere multithreading i vår applikasjon, fordi vi tok utgangspunktet i at kun en bruker kan bestille pizza om gangen, og derfor kreves det ikke parallell-utførelse. Dette har gjort koden vår ren og lettforståelig, og minsker risikoen for bugs. En annen grunn var også at vi valgte å fokusere på andre deler av pensum.

Testing

Unit testing brukes for å teste individuelle metoder/klasser/funksjoner, mens integration testing brukes mer for å teste om disse komponentene fungerer sammen i en hel løsning.
kilde: PG3302_11_EFCore-p3_MoreTesting.pdf. Sandnes. 2023

Vi prioriterte å teste metodene som ble ansett som de mest essensielle i programmet. Vi tok ikke i bruk Test driven development, da vi ønsket å fokusere på å få applikasjonen til å fungere før vi begynte med test-delen. Unit testing har vi tatt i bruk i programmet vårt, det har vi brukt til å teste individuelle komponenter for å sikre forventet resultat. Vi valgte å kjøre

noen tester på metoder i CrudOperationsCustomer og ShoppingCart klassen. I Tillegg har vi også implementert tester for diverse dekorasjons-utfall, og tester for å legge til pizza i databasen.

Vi tok beslutningen om å utelate bruk av mocking i vårt testprosjekt grunnet påviste utfordringer og tidsbegrensninger. Beslutningen om å utelate dette ble tatt på bakgrunn av oppståtte problemer, og de gitte tidsrammene. Imidlertid ville vi ideelt sett inkludert mocking hvis vi hadde hatt mer tid.

Lagdeling

Lagdeling handler om hvordan koden struktureres. En av hensiktene med lagdeling er at en del av koden skal kunne endres uten at det er nødvendig å endre på annen kode. Den mest grunnleggende måten å dele koden opp i er lagene brukergrensesnitt, applikasjonens logikk og tekniske tjenester. Brukergrensesnittet er den delen av koden som brukeren interagerer med, applikasjonens logikk har applikasjonens funksjonalitet og tekniske tjenester er for eksempel hvordan data lagres. I vår kode er Program-klassen brukergrensesnittet. Input fra brukeren sendes videre til applikasjonens logikk hvor den bearbeides og eventuelt sendes videre til tekniske tjenester. I vårt tilfelle bruker vi database SQLite til å lagre data.

Eksempel: Brukeren kommuniserer med brukergrensesnittet med ønske om å logge inn. Applikasjonens logikk sjekker om brukeren er registrert i databasen ved å sende forespørsel til tekniske tjenester. Data som sendes tilbake fra databasen håndteres av applikasjonens logikk. Dersom brukeren finnes i databasen blir hen logget inn ellers får hen beskjed om å registrere seg som ny bruker. Til slutt får brukeren en velkomstmelding. Nå er brukeren logget inn så lenge programmet kjører.

EF Core

For lagring av data har vi brukt rammeverket EFCore til å integrere databasen SQLite med kode-først-tilnærming hvor vi bruker LINQ metode-syntax. Vi valgte kode-først-tilnærming fordi hvis man ser at man må endre noen av entitetsklassene, så oppdaterer EF core tabellen når vi kjører migrations. Det gjør det lettere for oss hvis vi plutselig oppdager en feil i logikken. Vi har en egen mappe for tabellene/domeneobjektene. I disse klassene lages objektene som skrives til/fra databasen/koden. Vi synes at metode-syntax er lettlest og gir oss mulighet til å bruke lambda som gjør det enklere å hente ut data fra databasen.

Ved å bruke EF Core ble databasen definert av entitets klassene og en DbContext klasse. Tabellene har dermed blitt lagd av EF core basert på disse klassene når vi kjørte migration. kilde: PG3302_09_EfCore-part1.pdf. Sandnes. 2023

Gjennom koding har vi etablert tabellene Order, Pizza, Pizza_Order og Customer som vist i samsvar med UML for databasen. Dette bidro til en klar oversikt over hele bestillingsprosessen. Vi bruker imidlertid ikke order-tabellen, men har valgt å beholde den som en del av software-designet. Dette gir en forutseende tilnærming, slik at hvis systemet utvides i fremtiden, kan en order tabell være nyttig å ha.

I prosjektet vårt finner man en migrations-mappe som illustrerer hvordan EF Core har generert våre tabeller basert på entitetene våre, som vist i koden nedenfor:

```
namespace PizzaOrderingApp.Entities {
    public class Customer {

        public int CustomerId { get; set; }
        public string CustomerName { get; set; } = string.Empty;
        public int PhoneNr { get; set; }

        public ICollection<Order>? Order { get; set; }
    }
}
```

```
namespace PizzaOrderingApp.Entities {

    public class Pizza : IPizza {
        public int PizzaId { get; set; }
        public string PizzaName { get; set; } = string.Empty;
        public int Price { get; set; }
        public string Description { get; set; } = string.Empty;
        //public char Size { get; set; }

        public ICollection<Pizza_Order>? Pizza_Order { get; set; }
    }
}
```

```
    namespace PizzaOrderingApp.Entities {

        public class Pizza_Order {
            [Key]
            public int PizzaOrderId { get; set; }
            public int OrderId { get; set; }
            public int PizzaId { get; set; }


            public Order? Order { get; set; }
            public Pizza? Pizza { get; set; }
        }
    }
```

```
namespace PizzaOrderingApp.Entities {
    public class Order {
        public int OrderId { get; set; }
        public int CustomerId { get; set; }
        public Customer? Customer { get; set; }


        public ICollection<Pizza_Order>? Pizza_Order { get; set; }
    }
}
```

For å inkludere nye pizzatyper i vår pizza-tabell, implementerte vi en metode som lagret elementene i en liste. Kilde: Forelesning PG3302_09_EfCore-part1.pdf, Sandnes. 2023

```
namespace PizzaOrderingApp.MenuHandler
{
    public class AddMenuItems
    {
        //crud
        CrudOperationsMenu crudOperationsMenu = new CrudOperationsMenu();

        //Metode for å legge til pizzaer i databasen
        public void AddItems()
        {
            try
            {
                    List<Pizza> pizzas = new List<Pizza>
                {
                    new Pizza { PizzaName = "Margarita", Price = 99, Description = "Tomato sauce, cheese" },
                    new Pizza { PizzaName = "Pepperoni", Price = 149, Description = "Tomato sauce, cheese, pepperoni" },
                    new Pizza { PizzaName = "Vegan deluxe", Price = 129, Description = "Tomato sauce, vegan cheese, peppers, olives"},
                    new Pizza { PizzaName = "Hawaiian dream", Price = 149, Description = "Tomato sauce, cheese, pianpple, ham"}
                };

                    //crud
                    crudOperationsMenu.AddPizzas(pizzas);

                    Console.WriteLine("(Pizzas added to the database successfully.)");
            }
            catch (Exception ex)
            {
                Console.WriteLine("Error while adding pizzas to the database:");
                Console.WriteLine(ex.Message);
            }
        }
    }

}
```

Refaktorering

- Vi startet med å plassere spørringer i klassene i Application layer, men senere flyttet dem til en mappe i Technical layer.
- I begynnelsen hadde metodene flere oppgaver, men vi reorganiserte dem ved å flytte flere av oppgavene ut i egne metoder.
- Vi observerte at vi hadde tre nesten identiske klasser: IPizza, som fungerer som hovedpizzaobjektet; Entity, som arver fra IPizza og representerer pizzaentiteten; og CartItem, som deler betydelige likheter med pizzaobjektet. For å unngå å ha to nesten identiske klasser, valgte vi å la CartItem bruke IPizza-objektet i stedet for å gjenta nesten de samme feltene. Det eneste vi trengte å legge til her var "quantity". Dette resulterte i en mer effektiv og vedlikeholdbar implementering.

Utfordringer

Planlegging

Vi støtte på utfordringer i planleggingsprosessen. Vi startet med å utarbeide grundige planer for fordeling av ansvar hver uke. Imidlertid ble vi konfrontert med en annen eksamen midtveis i dette prosjektet, noe som gjorde det utfordrende å balansere prioriteringene våre og sørge for at vi fikk tilstrekkelig fremgang i prosjektet før vi måtte sette av tid til øving for den andre eksamenen. Selv med denne utfordringen, klarte vi å løse det.

EF core

I starten var alt nytt, og vi måtte sette oss grundig inn i det. Vi opplevde noen kommunikasjonsproblemer mellom ulike parter i gruppen, spesielt når det gjaldt å lage tabeller i databasen. Det oppsto ulike perspektiver på hvordan dette skulle håndteres. Imidlertid, etter å ha investert tid i å forstå problemet og komme til enighet om løsningsmetoden, gikk prosessen ganske smidig.

Versjonskontroll

Vi støtte på noen problemer med versjonskontrollen på GitHub, spesielt med merge conflicts. Dette oppsto for eksempel når to personer arbeidet i samme klasse, men skrev forskjellige ting. Problemer oppstod også hvis noen ikke hadde pushet kode på en stund, eller ikke hadde commitet eller pullet. Imidlertid løste dette seg raskt når vi ble flinkere til å gjøre disse operasjonene oftere. Vi organiserte også ansvarsfordelingen bedre for å unngå å jobbe for mye i samme klasse. Men dette ble heldigvis lettere og lettere hvor mer vi brukte git. Google document ble som versjonskontroll til rapporten, veldig praktisk da alle kan dokumentere i samme fil i sanntid.

Bugs og mangler

Manglende funksjonalitet

Ved å gjennomgå programmet vårt har vi identifisert litt manglende funksjonalitet.
Hvis man har for eksempel har 15 margaritha og velger remove pizza altså case 3 og du velger å fjerne margaritha så vil alle margaritaene bli fjernet. Men samtidig kan man argumentere for at hvis man ønsker å trappe ned mengden på en pizza kan man istedenfor velge change quantity. Da vil det ønskede utfallet forekomme.

Vi tok en uheldig snarvei: Logikken mellom menyen og handlekurven oppdaterer ikke prisen når man legger til en topping. Dermed vil prisen for en topping bli lagt på den opprinnelige prisen i handlekurven. Så når brukere legger til ekstra toppinger på en pizza, oppdateres ikke prisen på pizzaen i handlekurven (shopping cart) tilsvarende. I stedet vises den opprinnelige prisen på pizzaen, uten tillegg for toppingene. Så om brukeren ønsker å legge til topping(s) vises feilaktig totalpris i handlekurven.
Problemet synes å ligge i hvordan PizzaToppingDecorator-objektet (som inneholder den oppdaterte prisen med toppingene) håndteres og overføres til ShoppingCart. Det ser ut til at den oppdaterte pizzaen med tilleggstoppinger ikke riktig kommuniseres eller lagres i handlekurven.
Vi valgte å prioritere andre kritiske deler av applikasjonens logikk og funksjonalitet fremfor å løse dette problemet. Selv om vi forstår viktigheten av å inkludere tilleggstoppingene i priskalkulasjonen, har vi besluttet at utviklingen og forbedringen av andre funksjoner og komponenter i systemet har høyere prioritet akkurat nå.

Kjente bugs i koden

Vi oppdaget ingen bugs i koden.

Annet

Exeption handling

Vi har implementert exception handling i betydelige deler av menylogikken for å effektivt identifisere og håndtere spesifikke feilsituasjoner. Dette ga oss en bedre oversikt over potensielle feil og tillater en mer målrettet respons når unntak oppstår.

Internal og public funksjoner

Vi har implementert både interne og offentlige funksjoner i systemet vårt. For eksempel brukte vi interne funksjoner i klassen HandleCustomer for å demonstrere kompetanse. Dette valget ble motivert av at disse metodene ikke kan benyttes i andre prosjekter. Siden disse metodene samler inn data om kundene, er det svært fornuftig med tanke på å forhindre lekkasje av personlig informasjon.

Vi har brukt public funksjoner også i store deler av prosjektet, da dette gir mulighet for ekstern tilgang og integrasjon i systemet. Dette bidrar til smart software design med tanke på alltid å ha muligheten for utvidelse.

Topping funksjonalitet

For å hente tilgjengelige toppings fra databasen, har vi implementert en funksjon kalt GetAvailablePizzaToppings. Inne i denne funksjonen oppretter vi en instans av PizzaOrderingDbContext for å få tilgang til databasen.

Vi starter med å hente beskrivelsene av alle pizzaene i databasen ved hjelp av db.Pizza.Select(p => p.Description).ToList(). Dette gir oss en liste over beskrivelser som inkluderer alle toppings for hver pizza.

Deretter bruker vi SelectMany for å splitte hver beskrivelse basert på komma (,) og fjerne eventuelle tomme elementer. Dette resulterer i en liste over alle toppings som er tilgjengelige for pizzaene i databasen.

Vi bruker Distinct for å sikre at hver topping vises kun én gang, uavhengig av hvor mange pizzaer som bruker den. Til slutt trimmer vi hver topping for å fjerne eventuelle ekstra mellomrom rundt den.

Dette gir oss en endelig liste over unike toppings som er tilgjengelige for brukere å velge fra når de legger til toppings på pizzaene sine. Denne tilnærmingen sikrer at toppings ikke blir duplisert, og at de vises nøyaktig en gang, selv om de brukes på flere pizzaer. Dette gir en optimal opplevelse for både brukere og utviklere når det kommer til å legge til og velge toppings.

```
namespace PizzaOrderingApp.Application_logic.Decorators {

    // PizzaToppingDecorator enhances an IPizza object with additional toppings
    5 references
    public class PizzaToppingDecorator : IPizza {
        private readonly IPizza _decoratedPizza;
        private readonly List<string> _additionalToppings = new List<string>(); // // List to hold extra toppings
        private const int ToppingPrice = 30; // price pr topping

        2 references
        public PizzaToppingDecorator(IPizza decoratedPizza) {
            _decoratedPizza = decoratedPizza;
        }

        // Properties, add additional logic for toppings
        7 references
        public int PizzaId => _decoratedPizza.PizzaId;
        6 references
        public string PizzaName => _decoratedPizza.PizzaName;
        7 references
        public int Price => _decoratedPizza.Price + _additionalToppings.Count * ToppingPrice;
        8 references
        public string Description => $"{_decoratedPizza.Description}, {string.Join(", ", _additionalToppings)}";

        5 references
        public void AddTopping(string topping) {
            _additionalToppings.Add(topping);
        }

    }
}
```

Her kan du se at alle har ", " mellom seg så systemet vett ka den ska henta

```
using PizzaOrderingApp.Entities;
using PizzaOrderingApp.Technical_services.CRUD;

namespace PizzaOrderingApp.MenuHandler {
    2 references
    public class AddMenuItems {
        CrudOperationsMenu crudOperationsMenu = new CrudOperationsMenu();

        //Method to add pizzas to the database
        1 reference
        public void AddItems() {
            try {
                List<Pizza> pizzas = new List<Pizza>
                {
                    new Pizza { PizzaName = "Margarita", Price = 99, Description = "Tomato sauce, cheese" },
                    new Pizza { PizzaName = "Pepperoni", Price = 149, Description = "Tomato sauce, cheese, pepperoni" },
                    new Pizza { PizzaName = "Vegan deluxe", Price = 129, Description = "Tomato sauce, vegan cheese, peppers, olives"},
                    new Pizza { PizzaName = "Hawaiian dream", Price = 149, Description = "Tomato sauce, cheese, pianpple, ham"}
                };

                crudOperationsMenu.AddPizzas(pizzas);

                // Can uncomment this to see if items are added to database
                //  Console.WriteLine("(Pizzas added to the database successfully.)");
            } catch (Exception ex) {
                Console.WriteLine("Error while adding pizzas to the database:");
                Console.WriteLine(ex.Message);
            }
        }
    }

}
```

Kilder

PG3302_07_Multithreading_DP-Factory-Decorator.pdf
PG3302_11_EFCore-p3_MoreTesting.pdf
PG3302_09_EfCore-part1.pdf
PG3302_06_SOLID_DP-DependencyInjection-Factory.pdf
PG3302_04_CodeExample-CardDeck.zip
PG3302_09_EfCoreExample (zip)
PG3302_05_Debugging_OperatorOverloading_UnitTesting.pdf
Av Tomas Sandnes, 2023
https://learn.microsoft.com/en-us/ef/core/modeling/inheritance
https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.selectmany?view=net-8.0
https://learn.microsoft.com/en-us/dotnet/api/system.environment.exit?view=net-8.0https://learn.microsoft.com/en-us/dotnet/api/system.string.join?view=net-8.0
https://dev.to/tamerlang/understanding-solid-principles-dependency-inversion-1b0f#:~:text=The%20Dependency%20Inversion%20Principle%20(DIP,should%20not%20depend%20on%20details
Kilder brukt November 2023

Vi har brukt en sammensatt forståelse av pensum vi har lært i forelesningene og kunnskap vi hadde fra før av.