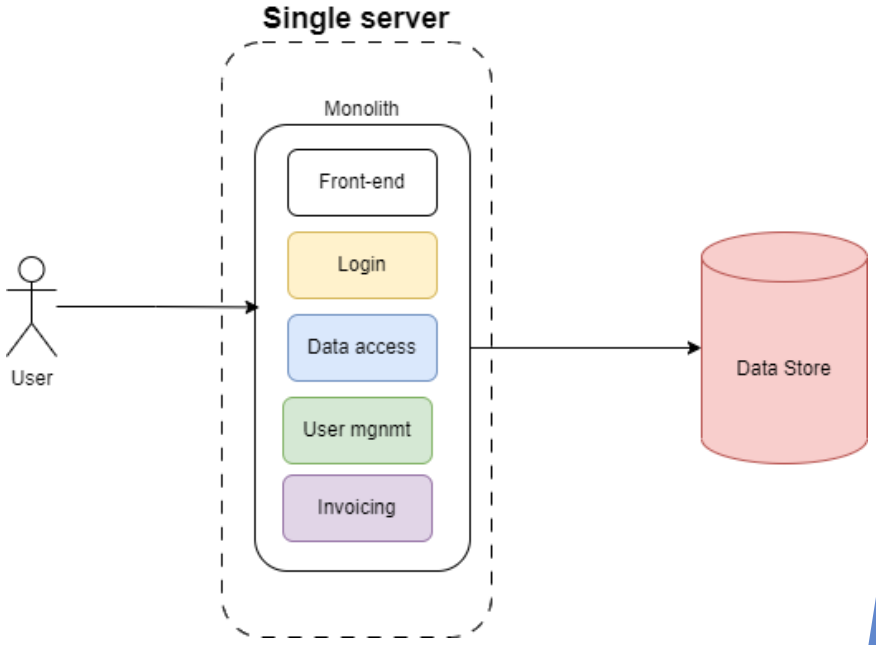
Arkitektur og løst koblet kode

Agenda

- Oppsummering fra sist
- Microservices arkitektur
- Skalering
- Heksagonal arkitektur
- Konseptuelt eksempel
- Kodeeksempel

Monolittisk arkitektur

Monolittisk arkitektur: All kode henger tett sammen og kan tenkes på som én "ting"	•Alle komponenter er nødvendige for å kunne kompilere og kjøre •Resulterer i én kjørbar fil (Ett prosjekt) •De fleste software starter som en monolittisk arkitektur •Alle komponenter deler samme database
Fordeler	•Enkelt å utvikle •Enkelt å feil**søke** (men ikke nødvendigvis fikse) •Gir i prinsipp god hastighet
Ulemper	•Vanskelig å gjøre store endringer •Vanskelig å oppgradere teknologi •Vanskelig å teste! (Vanskelig å isolere enheter) •Kan være tungt å slippe nye versjoner
Best egnet for	•Små enkle applikasjoner •Kort levetid •Rask utvikling



Løst koblet kode

Løst koblet kode

- Komponentene kan ofte bygges og kjøres individuelt
- Endringer i én komponent skal IKKE påvirke andre komponenter
- Komponenter kan typisk byttes ut enkelt

Fordeler

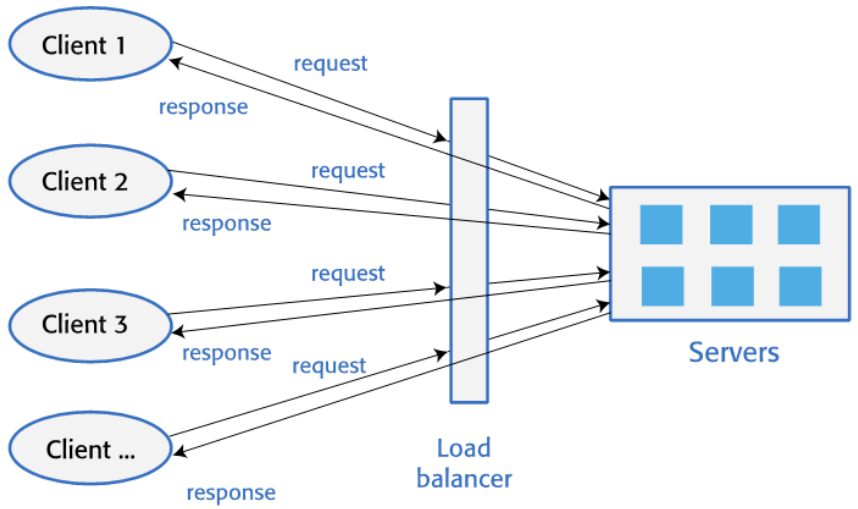
- Enkelt å opprettholde og videreutvikle
- Systemet kan tilpasses ved å bytte ut komponenter
- Komponenter kan enkelt gjenbrukes
- Meget skalerbart

Ulemper

- Tar lengere tid å utvikle
- Kan øke kompleksiteten
- Produktet kan potensielt ikke leveres som én «pakke»
- Øker potensielt prosesseringstid

Klient-server arkitektur

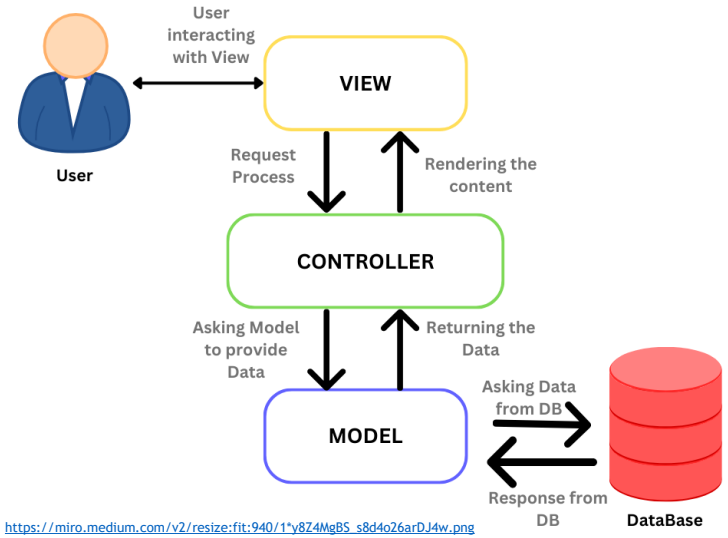
- Klient-server arkitektur: produkt-funksjonalitet skilles i klienter som snakker med delte servere
  - Klient – Er hovedsakelig ansvarlig for brukerinteraksjon og presentasjon av data
  - Servere – Er hovedsakelig ansvarlig for håndtering av data
  - Annen mer spesifikk logikk kan distribueres på enten klient eller servere
- Veldig vanlig i slikt som web-applikasjoner og mobil-applikasjoner
  - ... Egentlig generelt i applikasjoner med en eller flere delte databaser / servere



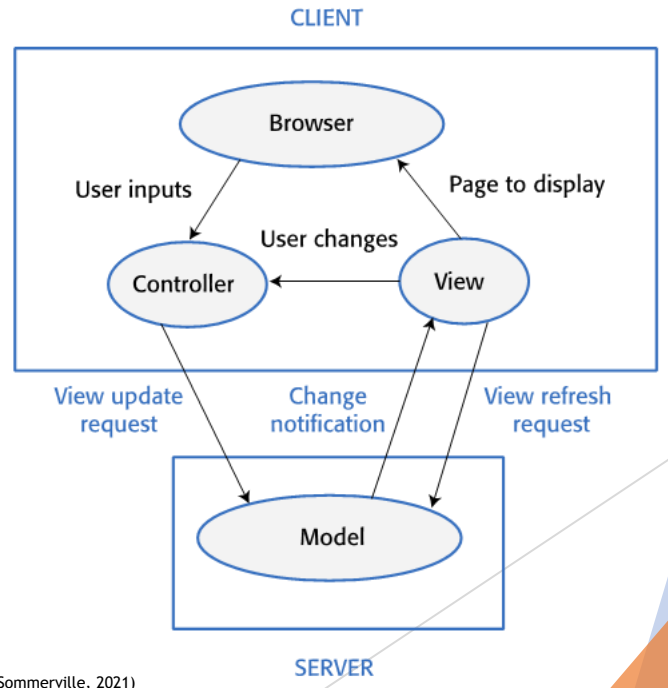
(Sommerville, 2021)

Model-View-Controller arkitektur

- Model-View-Controller (MVC) arkitektur: Ansvar deles i forskjellige lag
  - Model – Data / struktur av data
  - View – Visuell presentasjon av data
  - Controller – Ansvarlig for å håndtering av handlinger (events)
- Finnes flere varianter
  - Lagdelt
  - Trekant
  - Men felles: Model håndteres på serveren
- Brukes typisk sammen med klient-server arkitektur



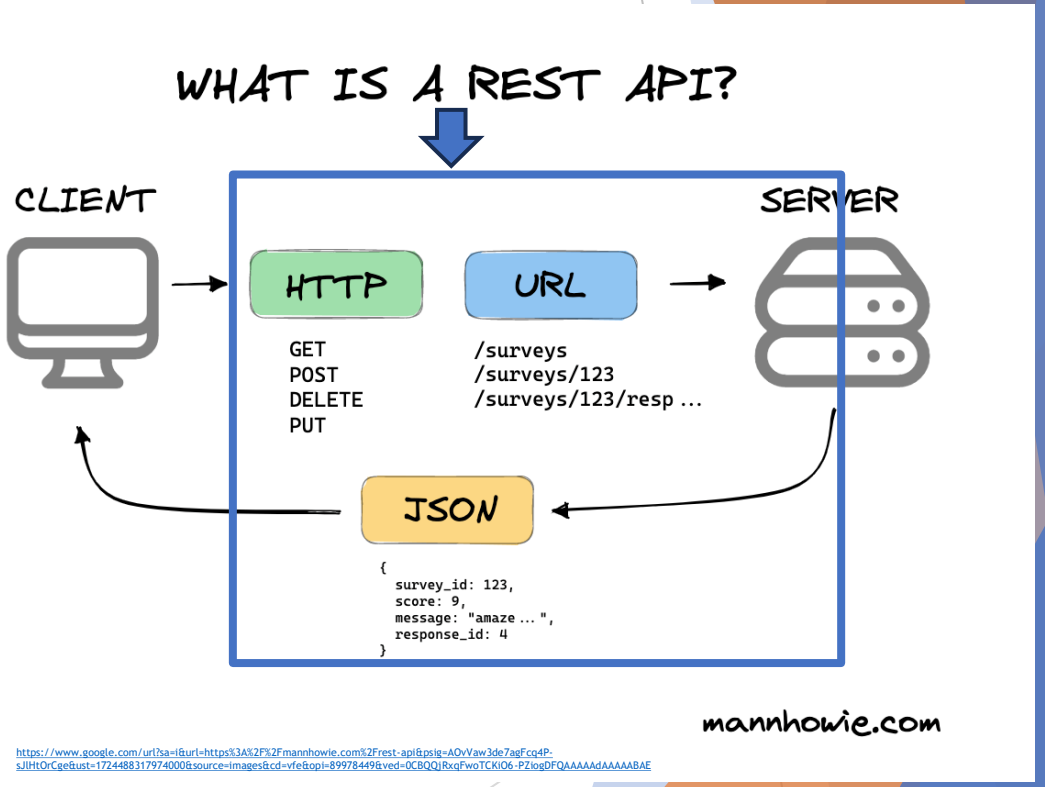
https://miro.medium.com/v2/resize:fit:940/1*y8Z4MgBS_s8d4o26arDJ4w.png



(Sommerville, 2021)

(Web-)API-er

- Application Programming Interface (API) – En type system-komponent som er ansvarlig for å håndtere kommunikasjon mellom andre komponenter
  - Kan tenkes på som en bro mellom komponenter (f.eks. Brukergrensesnitt og server/database)
- Kommunikasjonen foregår gjennom HTTP-forespørsler for abstrakte handlinger
  - Gi meg …, Legg til …, Oppdater …, Slett …, Utfør …
  - Kalles via URL-er
- Komponentene trenger altså ikke å ha direkte insikt i andre komponenters implementasjon
  - Bare hvilke handlinger som kan etterspørres gjennom API-et



https://www.google.com/url?sa=i&url=https%3A%2F%2Fmannhowie.com%2Frest-api&psig=AOvVaw3de7aqFcq4P-sJlHtOrCge&ust=1724488317974000&source=images&cd=vfe&opi=89978449&ved=0CBQQjRxqFwoTCKiO6-PZ1ogDFQAAAAAdAAAAABAE

Serialisering

- Vi kan ikke sende objekter (i tradisjonelt format) over nett
- HTTP-kommunikasjon er TEKST
  - ... altså må sendte objekter også være tekst

- Serialisering – Transformere et objekt til tekst
- Deserialisere – Transformere tekst tilbake til et objekt

- Typisk JSON-format
  - Alternativt XML, men JSON er mer lettvektig og ofte lettere å lese

- Det finnes mange biblioteker for serialisering/deserialisering
  - Jackson (kan anbefales)
  - Gson
  - Finnes typisk alternativer i alle språk

```
public class Person {
    private String name;
    private int age;
    private List<String> hobbies;

    // Getters and setters
}
```

```
{
    "name": "John Doe",
    "age": 30,
    "hobbies": [
        "Reading",
        "Hiking"
    ]
}
```

Data Transfer Object (DTO)

- Et Data Transfer Object (DTO) er en klasse/objekt som BARE benyttes i forbindelse med oversendelse av data (over nett)
  - Inneholder bare ren data (variabler med get-/set-metoder)
  - Kan samle data fra flere typer objekter i én struktur
  - Disse er typisk hva som blir serialisert/deserialisert
  - Typisk unike DTOs for spesifikke kontekster (views, typer komponenter, osv.)
  - Benyttes ved API-kall (hente og endre/sette inn)
  - DTO-klassene hører til modellene og ligger på serveren(e)

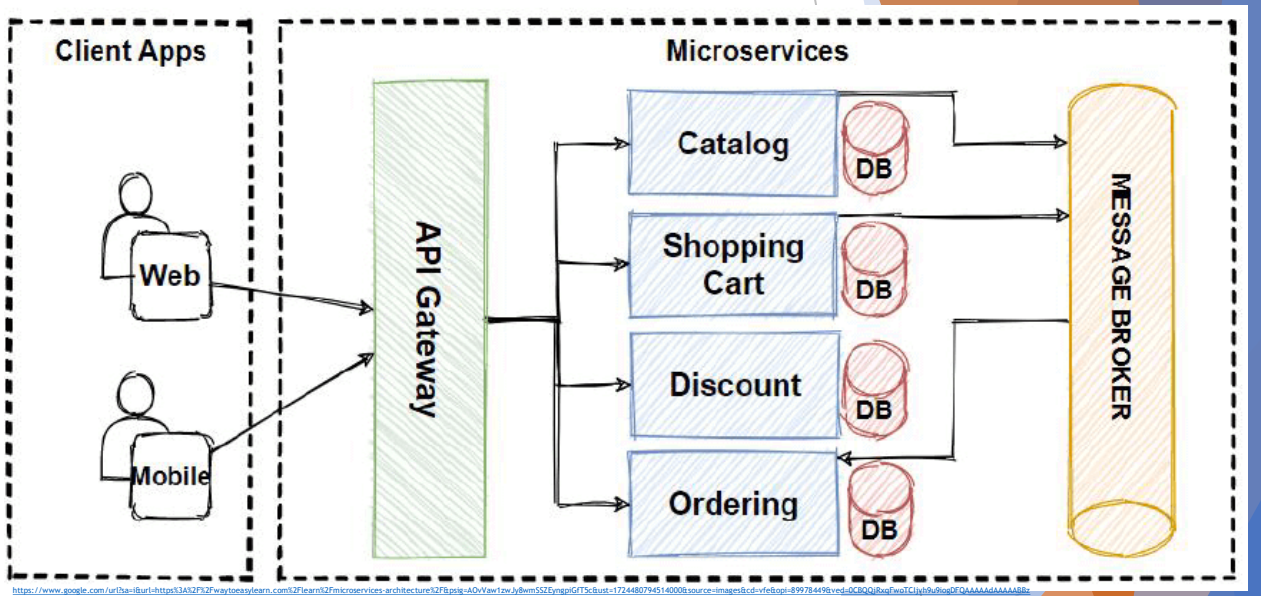
- Fordeler med å DTOs
  - Komponenter blir uavhengige av den originale datastukturen (løsere kobling)
  - Vi kan samle data og begrense oss til de vi trenger (optimalisere hastighet og redusere kompleksitet)
  - Mer standardisert → Enklere å teste

```
public class ProductViewDTO {
    private Long id;
    private String name;
    private String description;
    private BigDecimal price;
    private String imageUrl;
    private boolean isAvailable;

    // Getters and setters
}
```

```
public class OrderSummaryDTO {
    private Order order;
    private Customer customer;
    private List<OrderItemDTO> items;

    // Getters and setters
}
```

Microservices-arkitektur

- En microservices-arkitektur er det motsatte av en monolittisk arkitektur
- Alle features i systemet blir laget som en egen dedikert «microservice»
  - Hostes på egne servere
  - Kan være basert på unik teknologi og språk
  - Har potensielt sine egne databaser
  - Kan kommunisere med andre komponenter/microservices
- Microservicene aksesseres gjennom en API-gateway
- Microservicene kan enkelt byttes ut eller skaleres opp
- Arkitekturen er typisk brukt i (store) cloud-tjenester



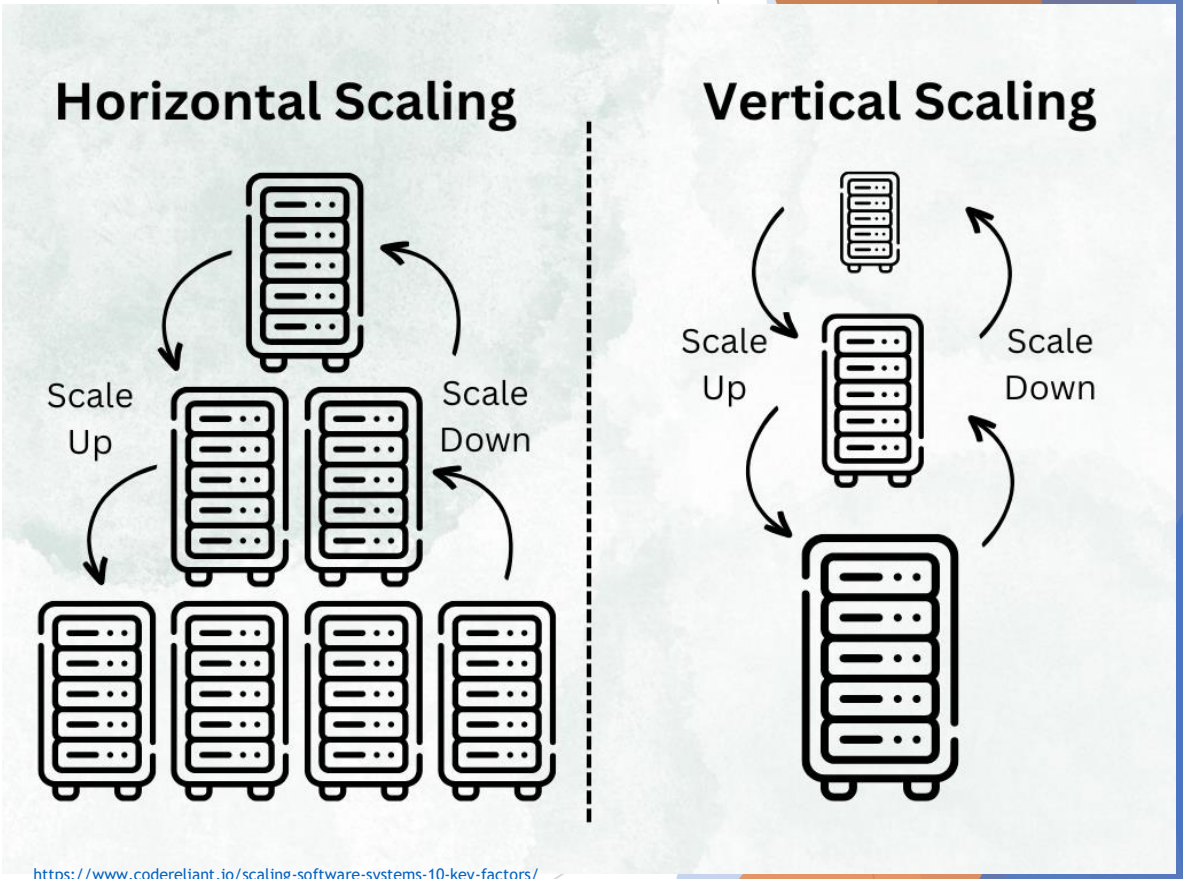
NETFLIX

Spotify®

amazon

Skalering

- Servere er også datamaskiner
  - Begrenset prosesseringskraft
  - Store mengder forespørsler kan kvele en server (nettbutikk-salg, DDoS-angrep, osv.)
- Vi må kunne ha muligheten til å oppgradere prosesseringskraften for å kompensere
  - Dette kalles skalering
- To typer skalering
  - Vertikal: Oppgradere hardware i serveren
  - Horisontal: Sette opp flere instanser av samme server og fordele trafikk
- For å kunne gjøre horisontal skalering må systemet være løst koblet
- Skalering er vel så relevant for microservices

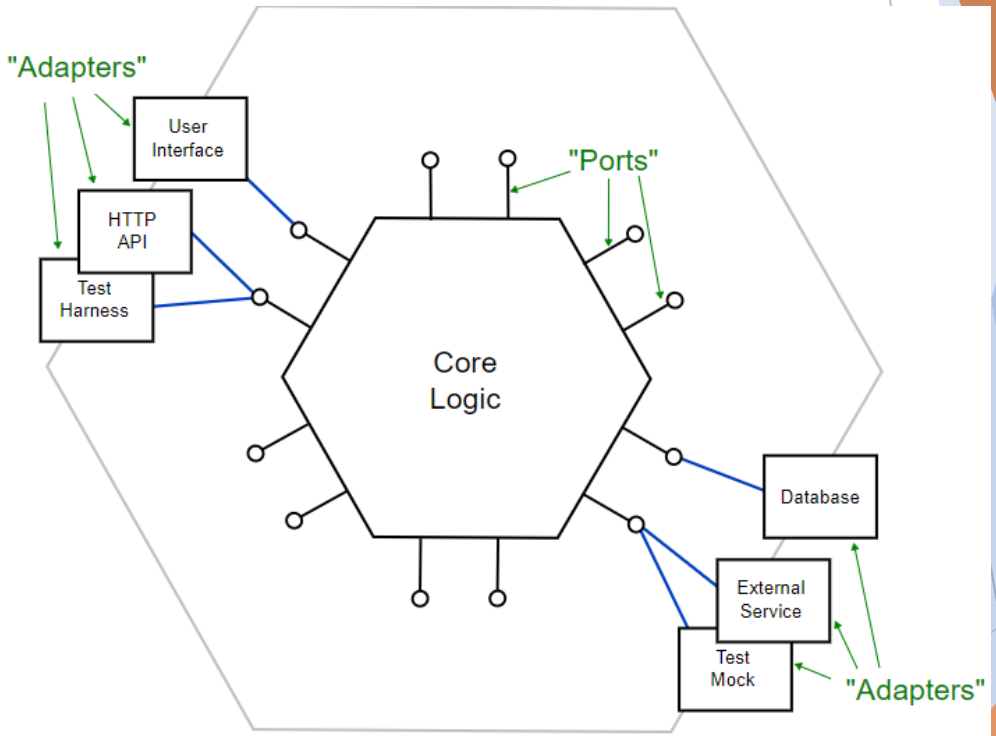


https://www.codereliant.io/scaling-software-systems-10-key-factors/

Litt nærmere koden

Heksagonal arkitektur (Ports and Adapters)

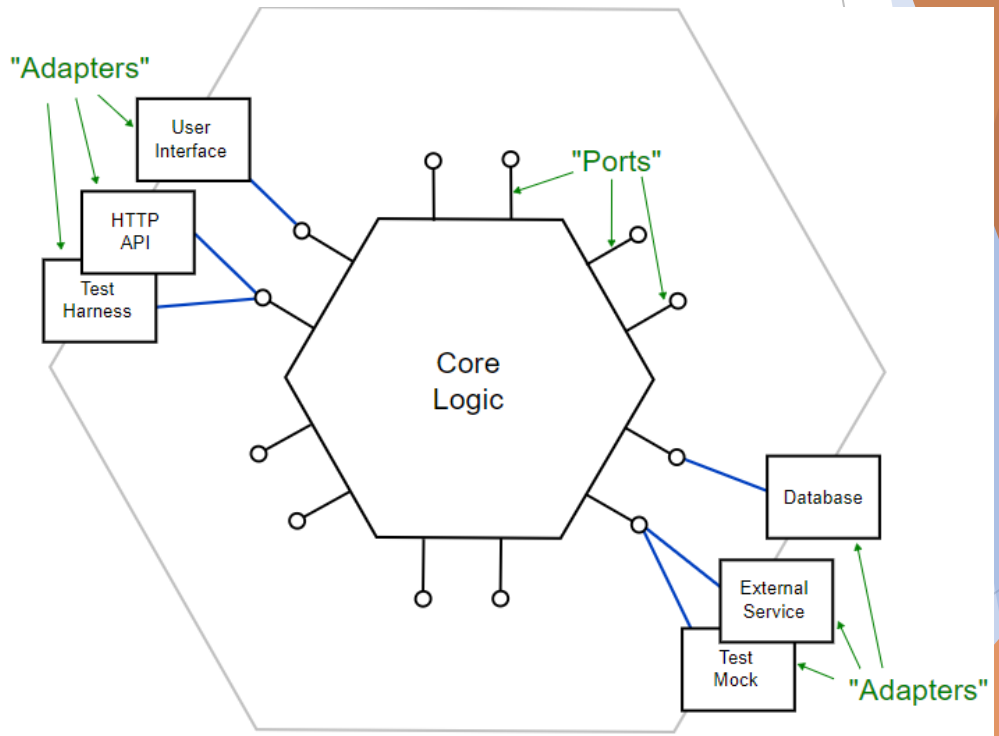
- Heksagonal arkitektur: Skiller «kjernekoden» og «alt annet»
  - «Alt annen» blir løst koblede komponenter som kan kobles TIL kjernen (men ikke motsatt)
- Kjernekode – Den mest fundamentale logikken i systemet
  - Typisk slikt som data-klasser og hvordan objektene av disse håndteres (*features*)
  - Helt uavhengig av konkret *implementasjon* (GUI, Database, osv.)
  - Kjernen skal IKKE ha noen avhengigheter
- Kjernekoden skal kunne «vare evig»
  - Vi skal kunne bytte implementasjon uten å måtte endre kjernen
  - Tenk hvordan individuelle banksystemer har blitt opprettholdt ...



https://timwise.co.uk/images/blog/hexagonal/hexagonal.svg

Heksagonal arkitektur (Ports and Adapters)

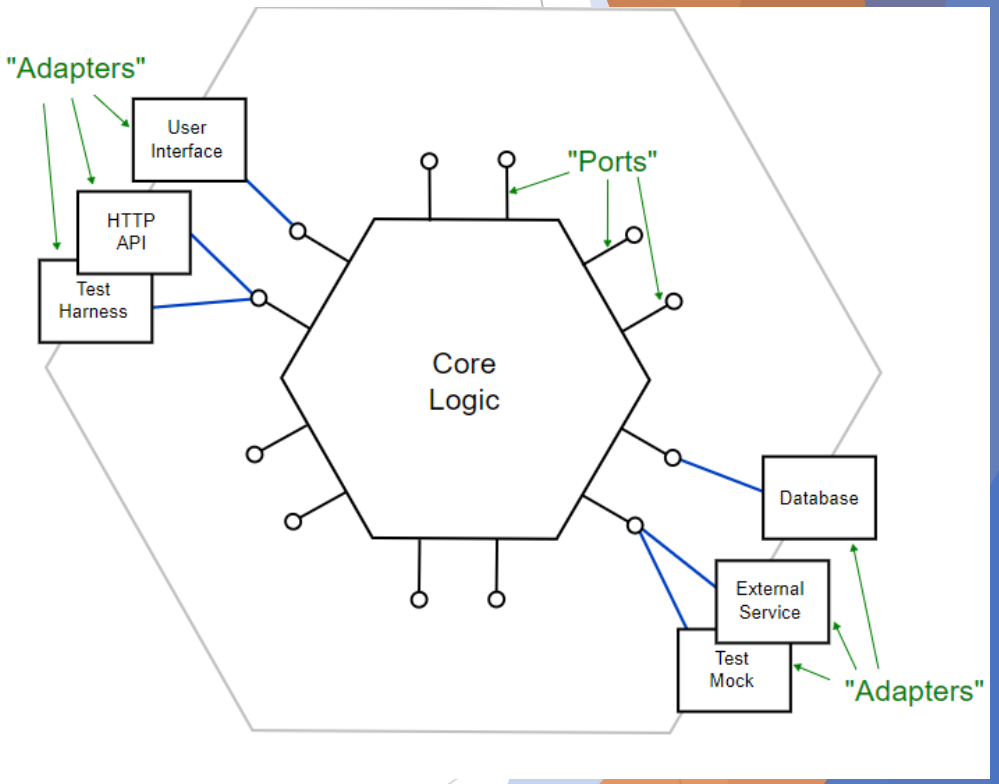
- For å kunne koble implementasjoner til kjernen definerer vi «ports» og «adapters»
- Ports – Definerte muligheter for interaksjon med kjernen (input / output)
  - Abstrakte metoder definert via Interfaces i kjernen
- Adapters – Spesifikk implementasjon
  - Implementerer iterfaces/ports fra kjernen
  - Ligger i egne moduler med kjernen som avhengighet



https://timwise.co.uk/images/blog/hexagonal/hexagonal.svg

Heksagonal arkitektur (Ports and Adapters)

- Typiske eksempler på port (interfaces) og adapters (implementasjon):
  - SQLDatabaseRepository implements RepositoryPort
  - APIProductController implements ProductControllerPort
  - JSONFileHandler implements FileHandlerPort
  - MessageBroadcaster implements MessageBroadcasterPort
  - EventLogger implements EventLoggerPort



Kjernekode

- Vi deler gjerne opp kjernekoden ut ifra produktets domene (Hva det omhandler)
  - Data, konsepter, oppførsel og regler, forhold osv.
  - Generelle Use cases – Handling → Effekt

- Hva defineres som kjernekode?
  - Entiteter (domene-modeller) – Hovedobjekter (Har typisk en unik ID) – kunde, produkt, osv.
    - Kan inneholde regler for opprettelse og manipulasjon, samt produksjon av andre resultater
  - Aggregatrøtter – Entiteter bestående av andre entiteter og verdityper – Ordre med kunde, ordrelinje og produkter
  - Klasser som representerer use cases - (typisk hente, opprette, endre, slette, osv.)
  - Generelle exception knyttet til use cases - (typisk feil ved henting, opprettelse, osv.)
  - Verdityper – Objekter som representerer typer verdier –
    - Penger, farger, størrelser osv.
  - Utility-klasser - Abstrakte klasser med statiske metoder eller konstanter
  - Porter (Interfaces) – Abstrakte metoder for interaksjon til eller fra kjernen
  - DTOs – Objekter for oversending av data – Beskriver i grunn kommunikasjon
    - Kan eventuelt skilles i en egen modul på siden av kjernen

Kjernekode

- Husk!
  - Kjernelogikk skal ikke være avhengig av eksterne biblioteker eller rammeverk
    - I så fall bare bittesmå eller veldig trygge avhengigheter
  - Klasser innenfor kjernen kan likevel være avhengig av hverandre og portene (interfaces)
  - Vi kan bruke kjernen som en avhengighet i andre moduler

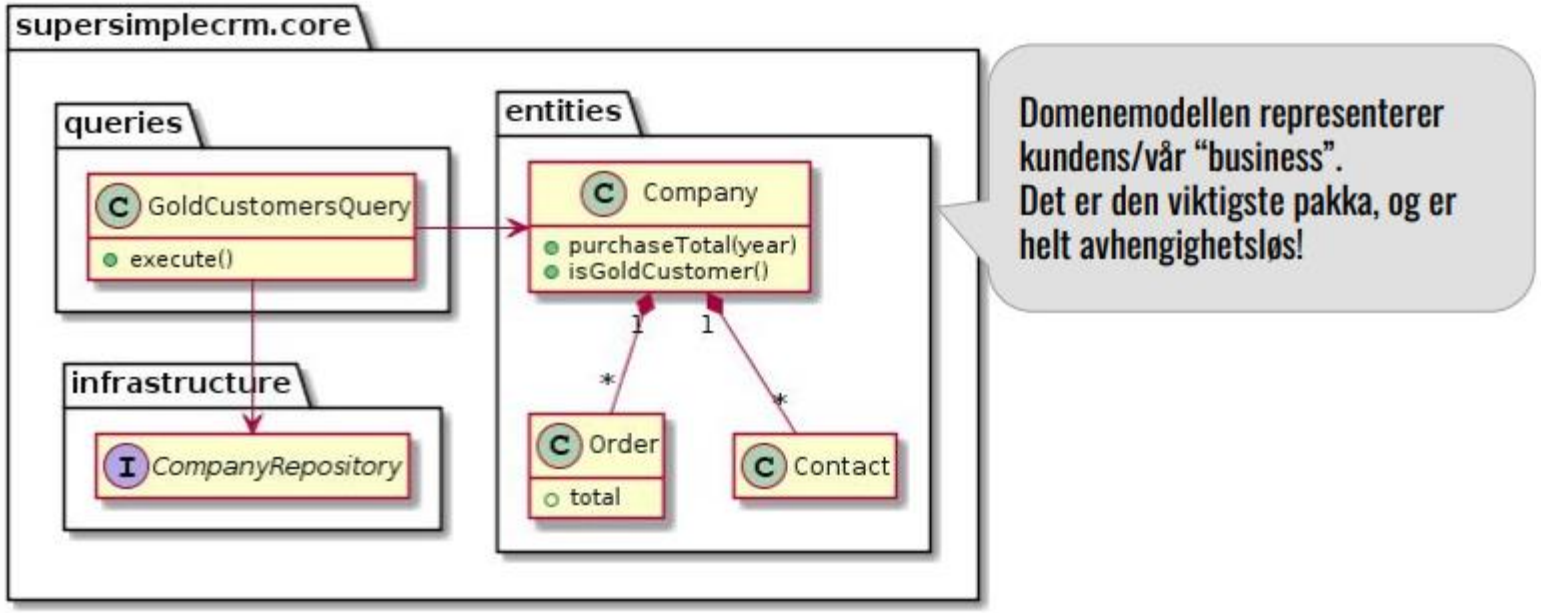
Knytning med testing

- Hvis vi strukturerer koden med denne arkitekturen blir det veldig enkelt å teste
  - Kjernekoden er ikke avhengig av noe og kan enkelt testes i isolasjon
  - Vi kan mocke portene (interfacene) for implementasjon
    - Forfalske respons fra andre komponenter

Konseptuelt eksempel

Eksempel - Kjerne

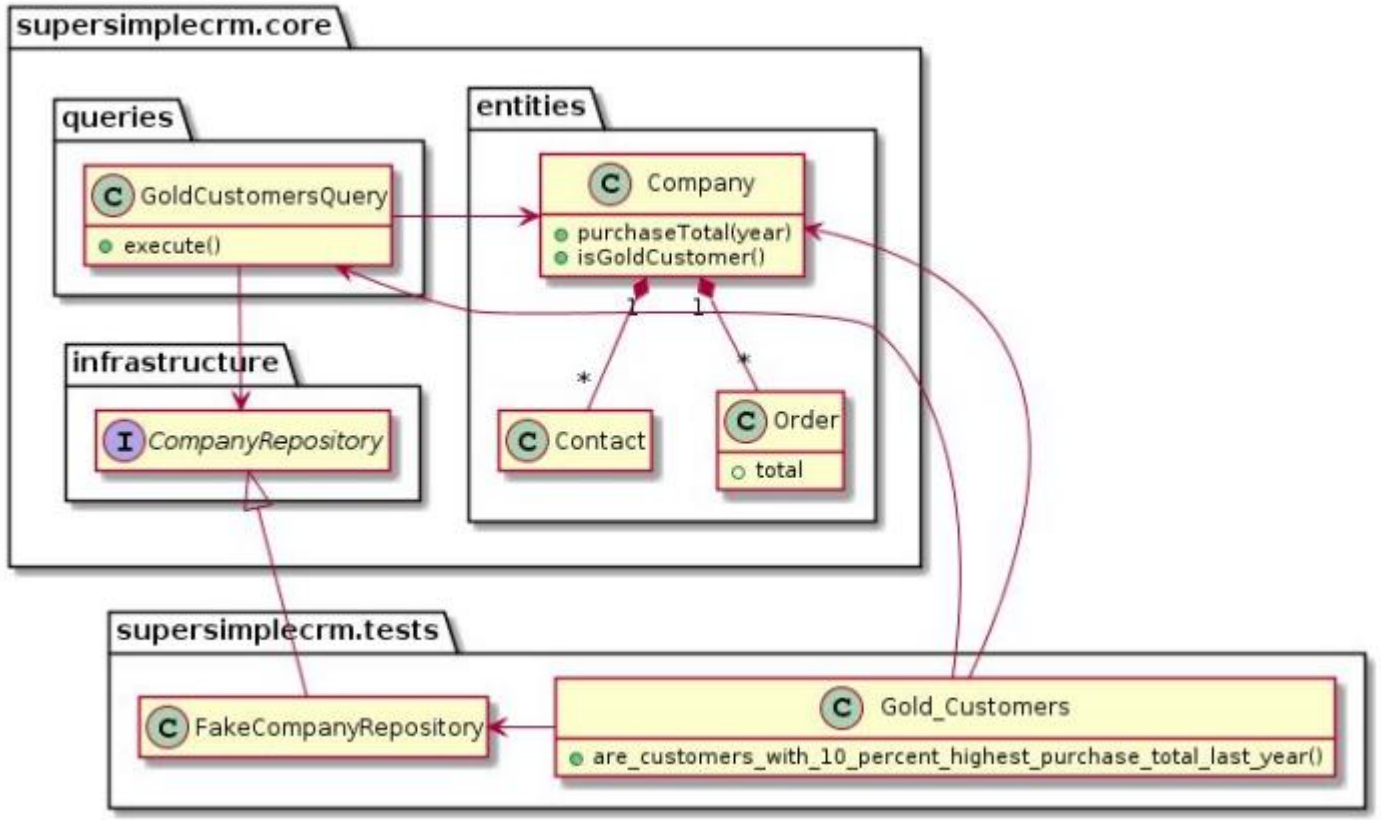
- Dette er et eksempel på en kjerne for et enkelt Customer Relationship Managment system (CRM)
  - Merk Interfacet CompanyRepository – Dette er en port for lagring



(Aabech, 2023)

Eksempel – Testing av Kjerne

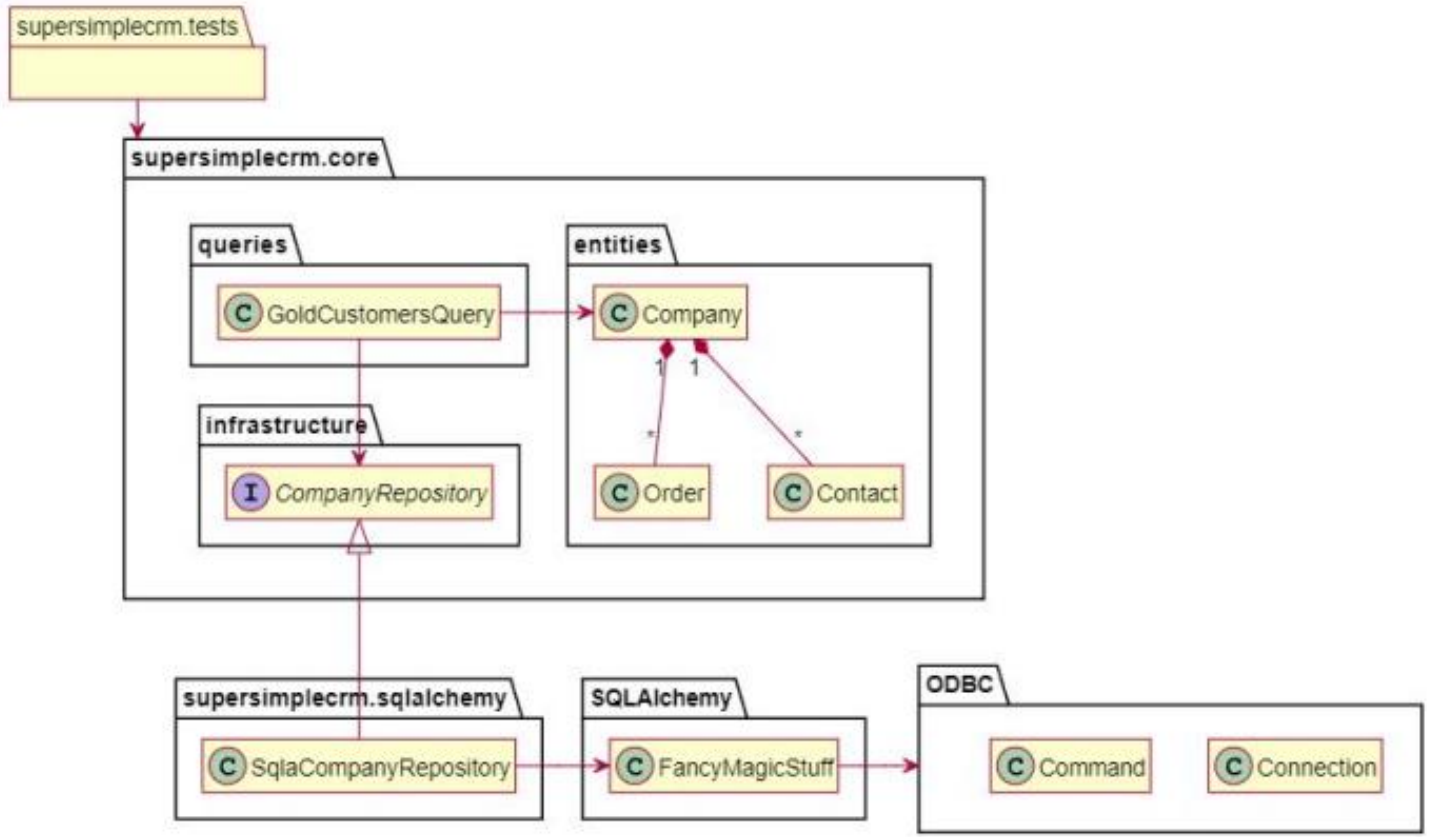
- Vi tester ofte i en separat modul fra kjernen
- Vi står fritt til å teste alt i kjernen enten i modulen eller som en egen modul



(Aabech, 2023)

Eksempel - Databaseimplementasjon

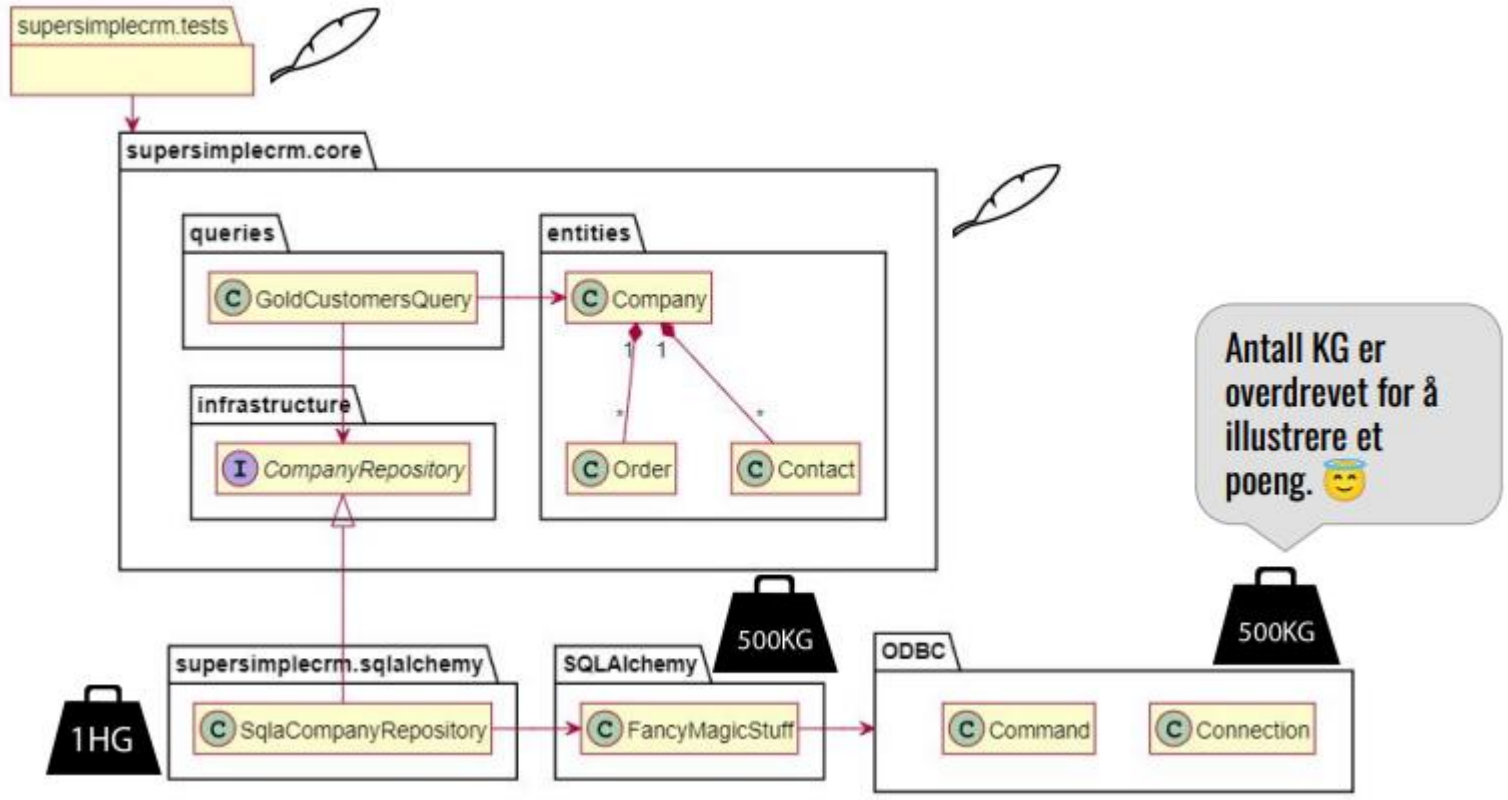
- Når vi skal implementere noe (f.eks. en database) bruker vi kjernen som en avhengighet i «implementasjons-modulen»
  - Enkelt å lage en ny implementasjon senere uten å endre kjernen



(Aabech, 2023)

Eksempel – «Vekt»

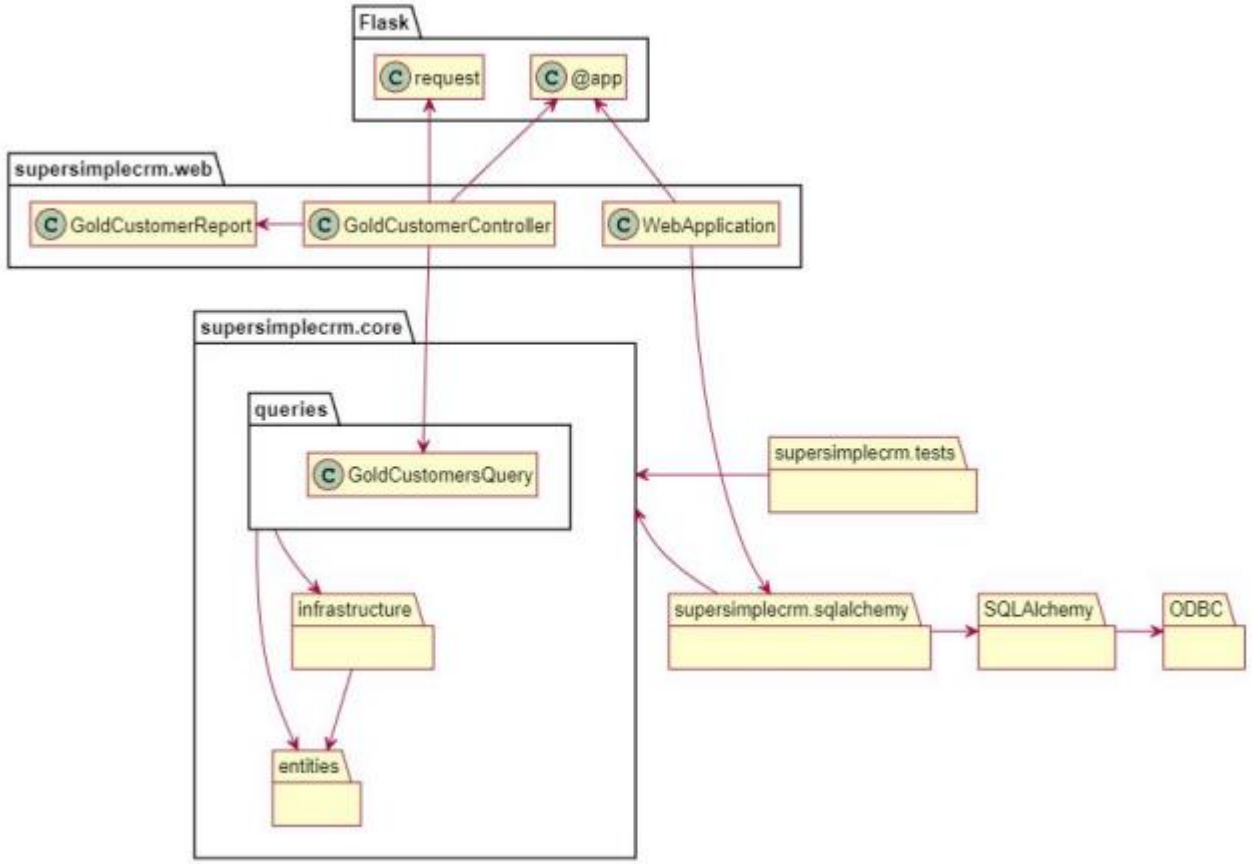
- Ved å implementere går systemet opp i «vekt»



(Aabech, 2023)

Eksempel - GUI

- GUI kan også implementeres i egne moduler ved å benytte funksjonalitet i kjernen
- (Burde kanskje også ha interfaces som definerer minimum for Controllere)



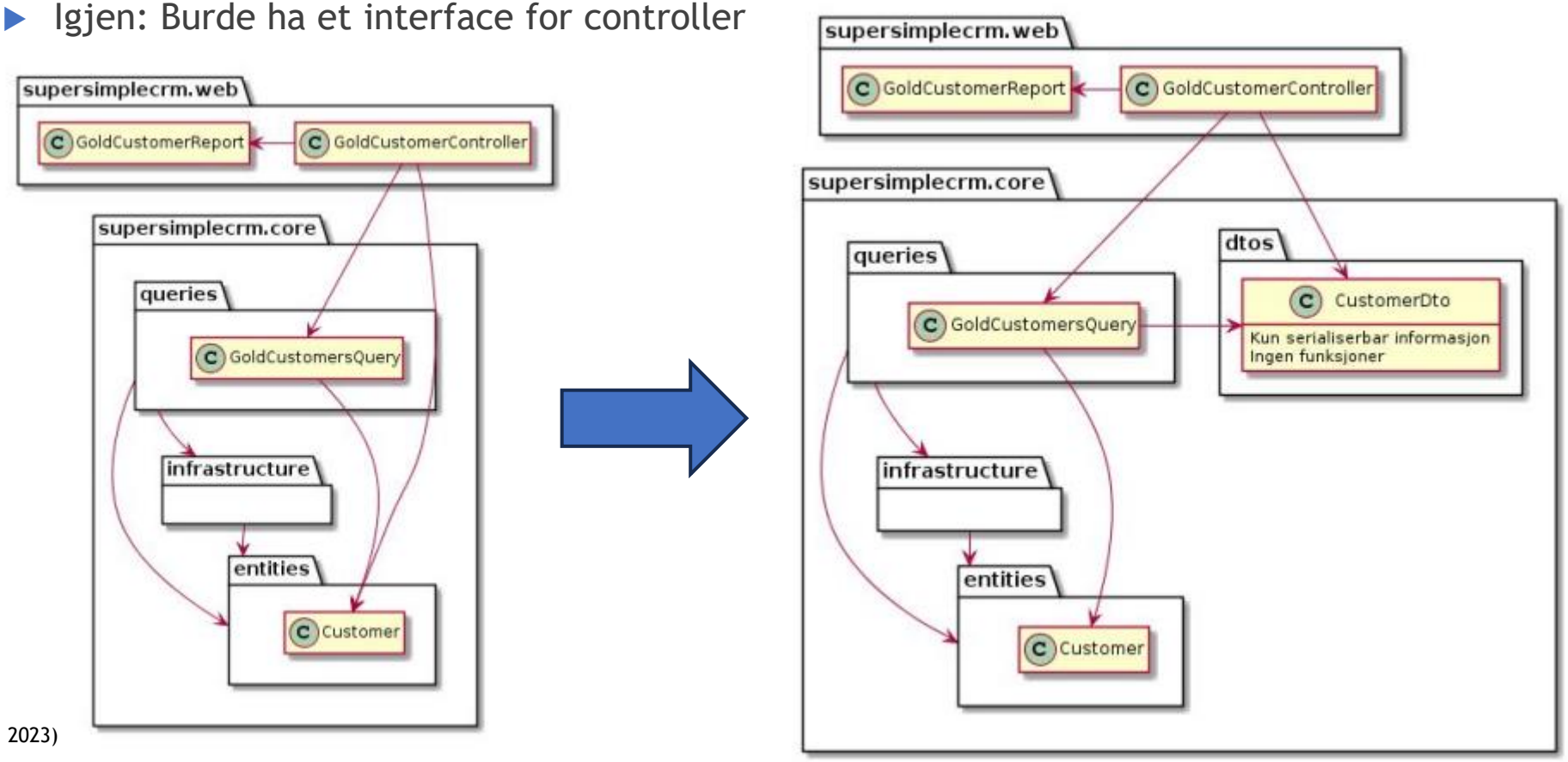
(Aabech, 2023)

Liten digresjon - GUI

- Ekstremet viktig at kjernen kan leve uavhengig av GUI
  - GUI-er er nesten alltid rammeverk
  - Hver applikasjon har ofte behov for flere GUI-implementasjoner (web, PC, mobil...)
  - GUI er ofte det som må oppgraderes hyppigest
- Frikobling
  - Lage «use-case»-klasser i kjernen som definerer interaksjon (her GoldCustomersQuery)
  - La GUI-controllere (eller API-controllere) jobbe mot disse «use-case»-klassene

Eksempel - DTO

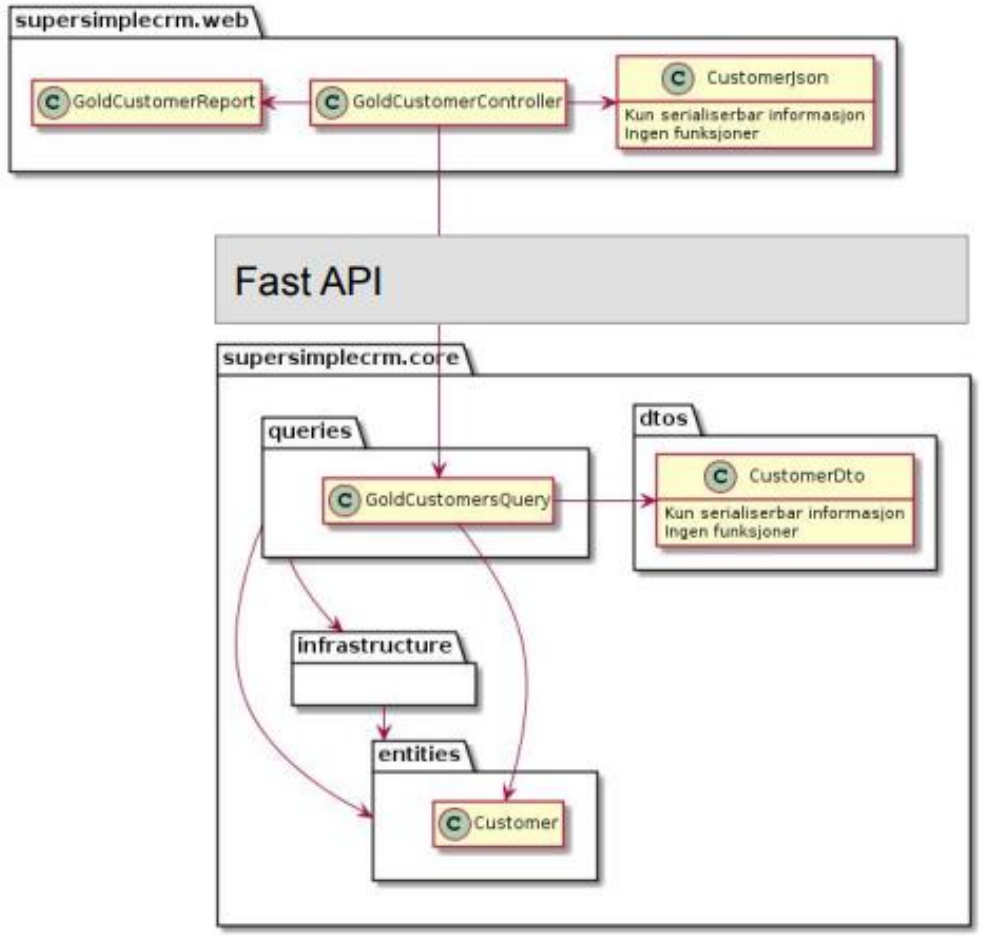
- GUI-modulene burde ikke aksessere domene-modellene direkte (tett kobling)
- I stedet burde vi lage egne DTOs for slik interaksjon (løsere kobling)
  - Igjen: Burde ha et interface for controller



(Aabech, 2023)

Eksempel - API

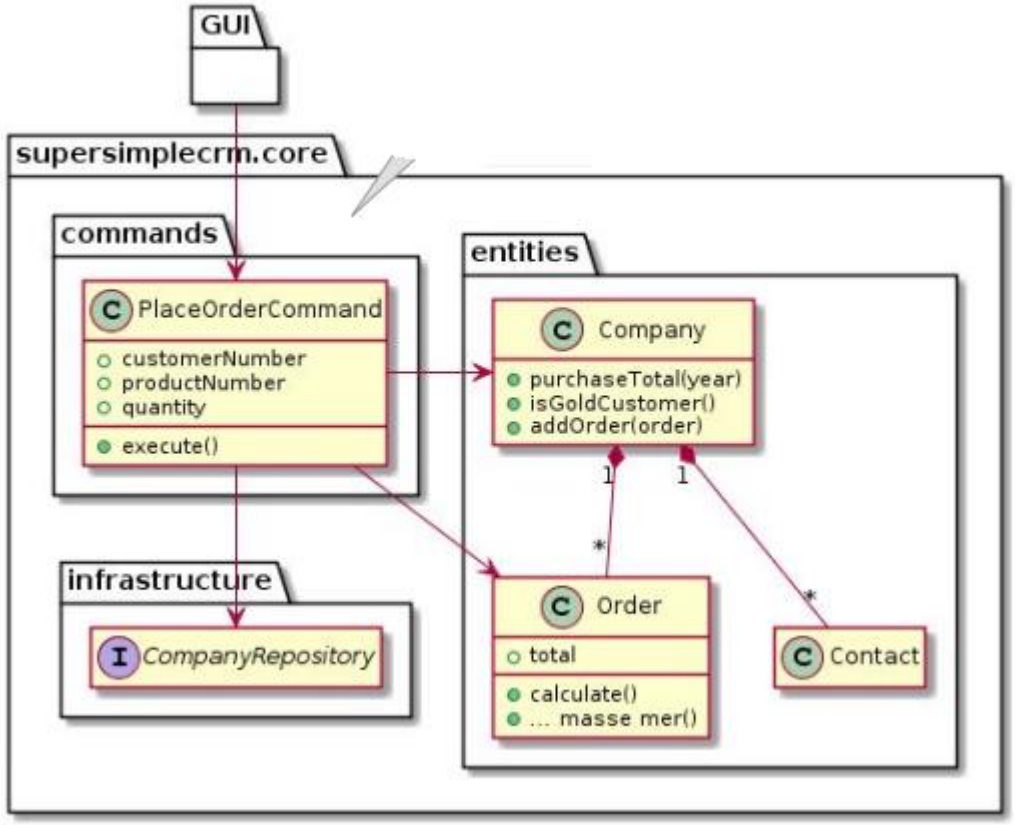
- For å lage en god webapplikasjon kan vi legge et API mellom web-GUI og kjernen (serveren)



(Aabech, 2023)

Eksempel - Kommandoer

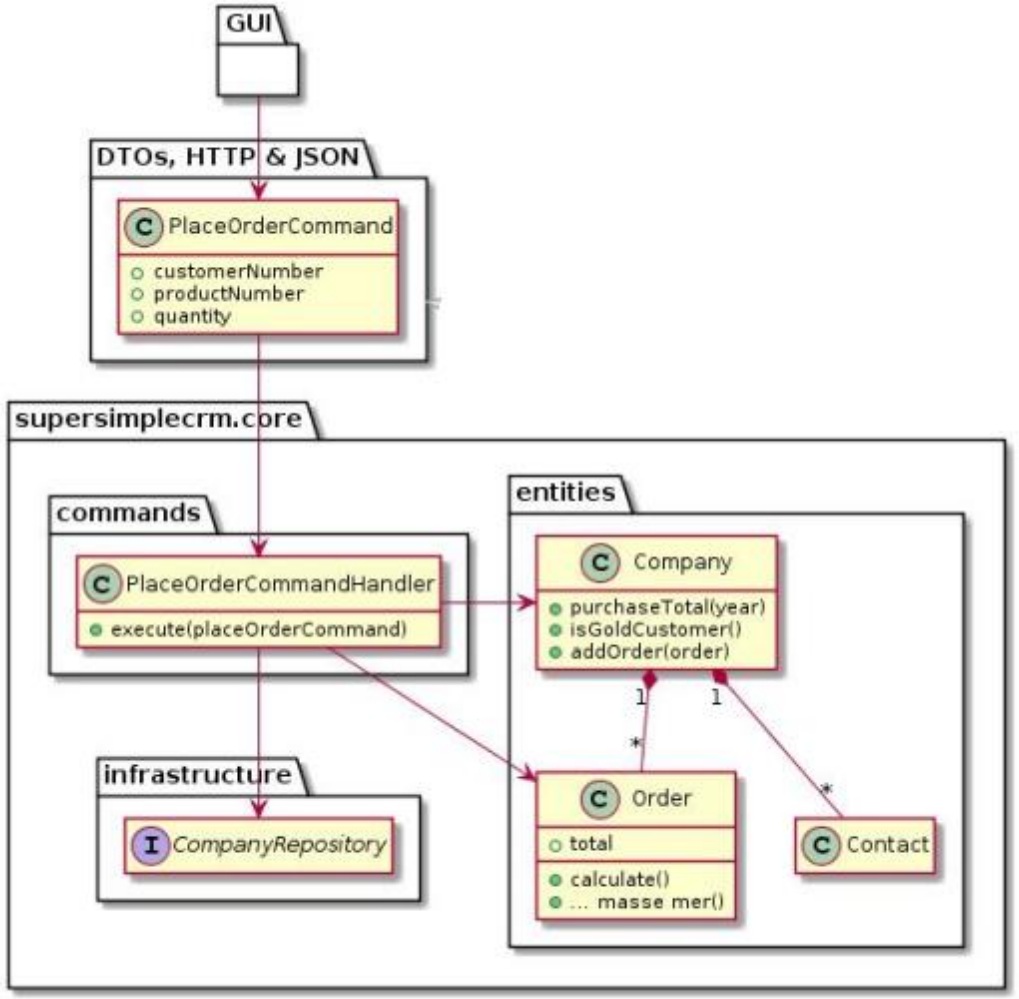
- For kall på handlinger i kjernen kan vi definere standardiserte Command-klasser
  - Handlings-use case
  - (Burde også kalles av et API)



(Aabech, 2023)

Eksempel - Lagdeling

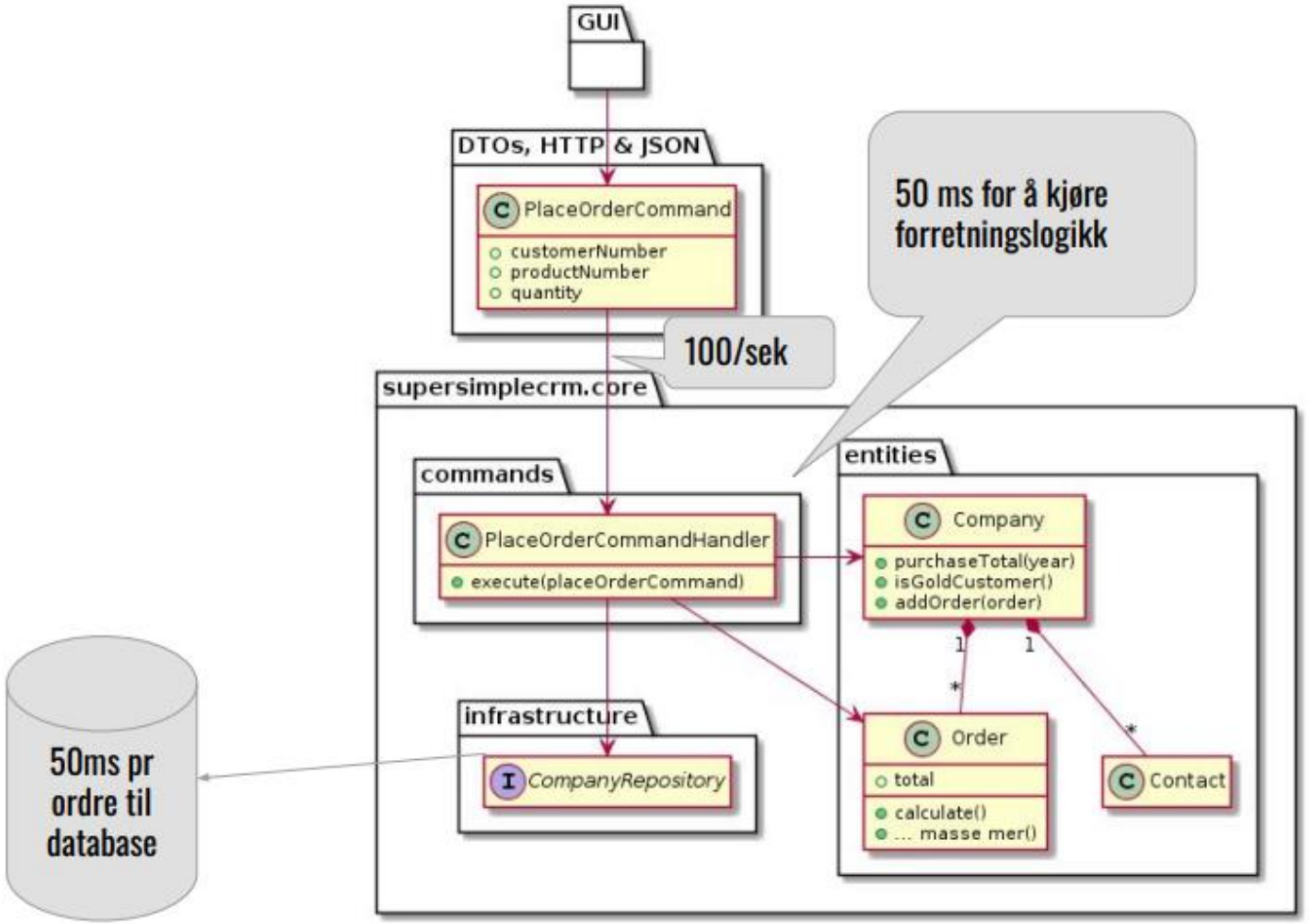
- Dette kan eventuelt lagdeles enda mer (modul med DTOs for kommunikasjon)



(Aabech, 2023)

Eksempel - Skalering

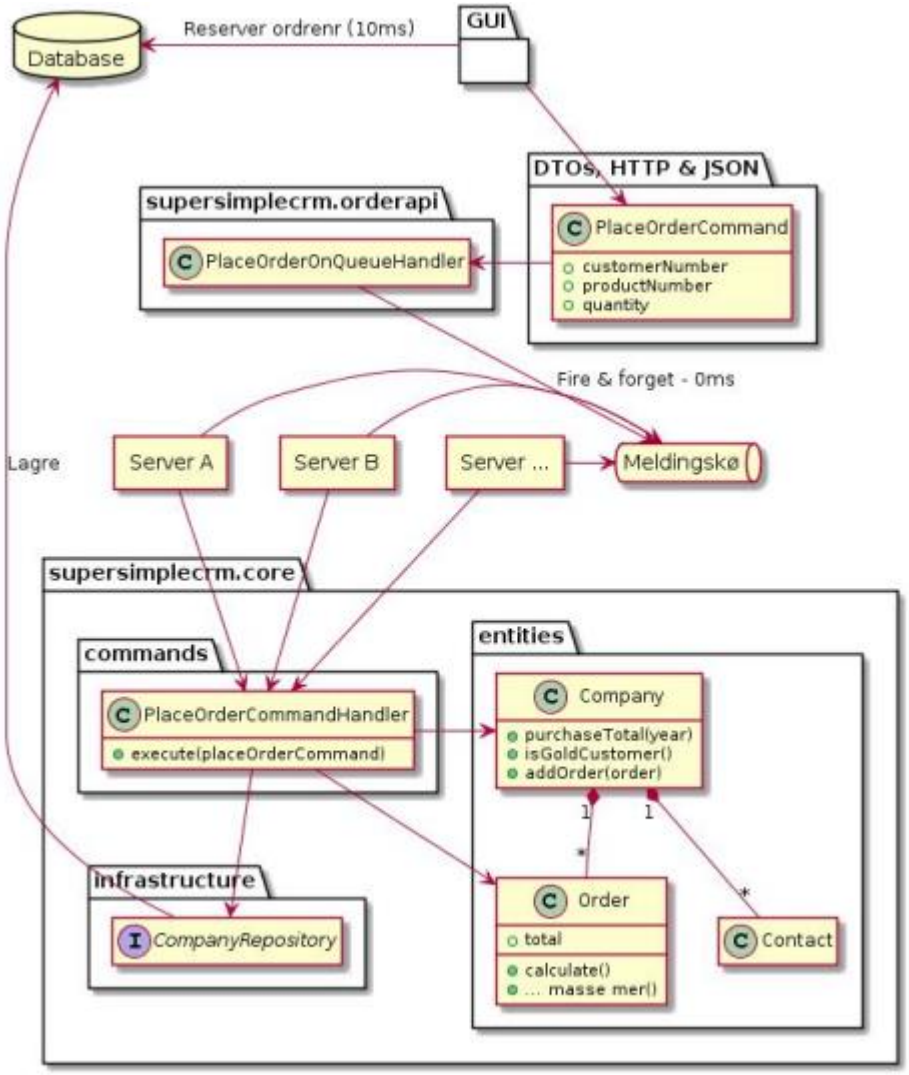
- Hvis bare én server kan systemet lett drukne i forespørsler



(Aabech, 2023)

Eksempel - Skalering

- Vi kan altså bruke skalering for å lage flere servere med kildekoden, samt en kø for å distribuere trafikk



(Aabech, 2023)

Hvilket nivå av løs kobling

- Vi kan se fordelene med løst koblet kode
  - Enklere å teste og fleksibel arkitektur
  - Enklere å skalere
  - Vi kan utvikle på forskjellige komponenter uavhengig av hverandre (så lenge kommunikasjonen mellom dem er standardisert)
  - Komponenter kan ha unike rammeverk, plattformer og løsninger
  - Komponenter kan hostes individuelt (microservices)

- Men bør vi alltid gå for løst koblet kode?
  - Mer komplekst
  - Større prosesseringstid

- Monolitter kan være vel så bra for små applikasjoner som ikke skal endres betydelig etter release
- Det finnes også teknikker for å lage monolitter med god lagdeling og mer modulære elementer

Mine anbefalninger til dere
Hva dere bør vurdere

- Jeg vil anbefale å forsøke på en heksagonal arkitektur
  - Støtter kursets fokus på testing
  - Tillater gruppen å jobbe på flere komponenter samtidig
  - Kul måte å strukturere på ...
- Vurder også slikt som
  - Model-View-Controller (og klient-server) arkitektur
  - API og DTOs