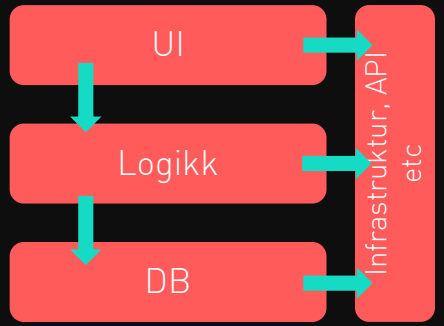
Intro til Clean Architecture

Henrik Wingerei
Espen Ekvang

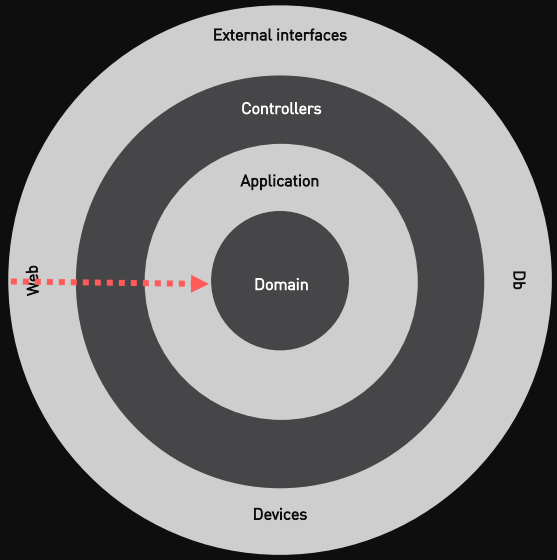
Utfordringer med tradisjonell lagdelt arkitektur

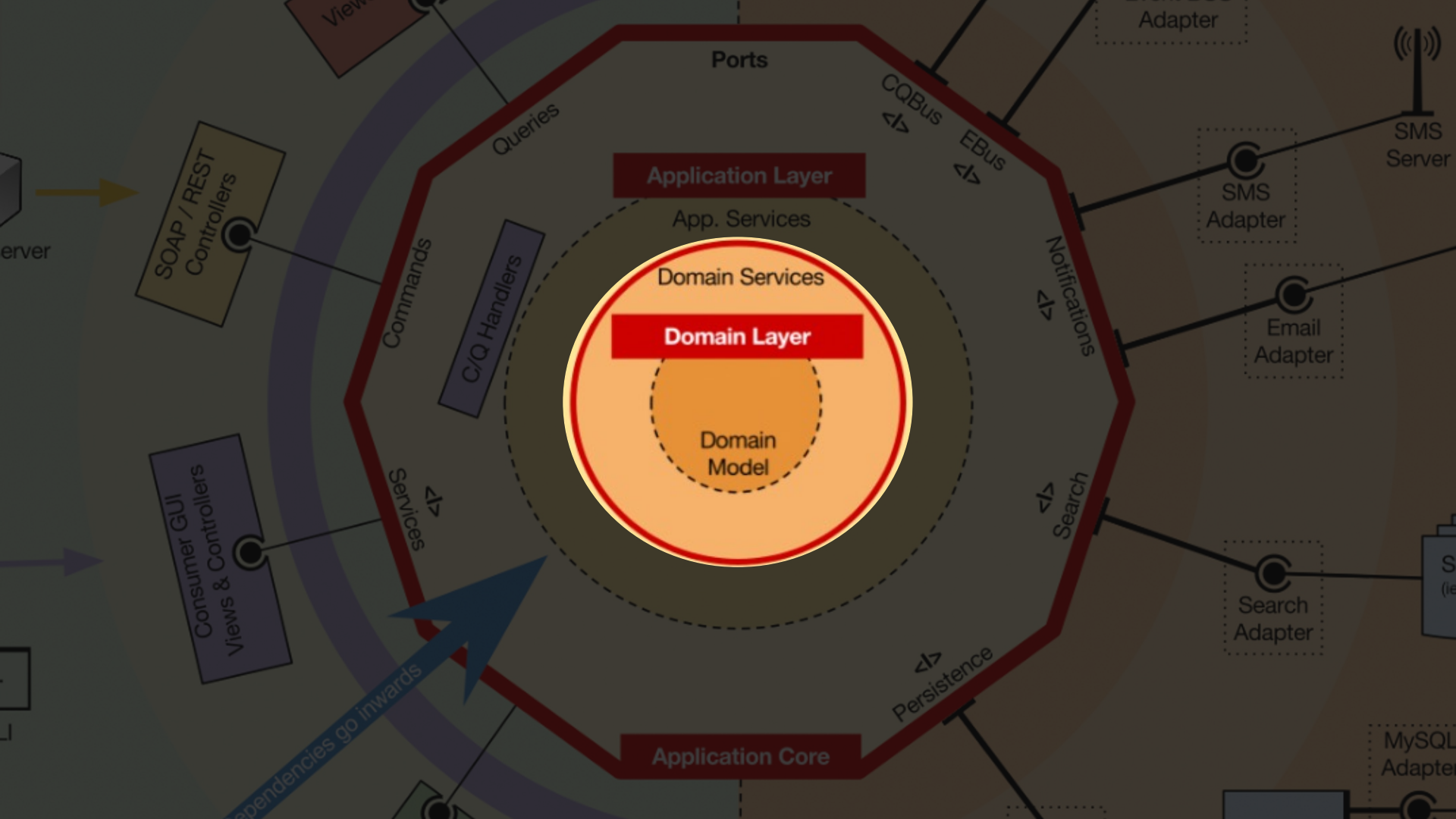
- I utgangspunktet ikke et tydelig skille mellom kjernen til applikasjonen (domene og forretningsregler) og «alt annet»
- "Detaljer" som DB, API osv. er ofte sentrale deler av arkitekturen
- Kan fort bli sterke koblinger til infrastruktur, database, 3. parts rammverk osv.
- Alle avhengigheter peker nedover til DB
- Kan være utfordrende å teste logikken isolert

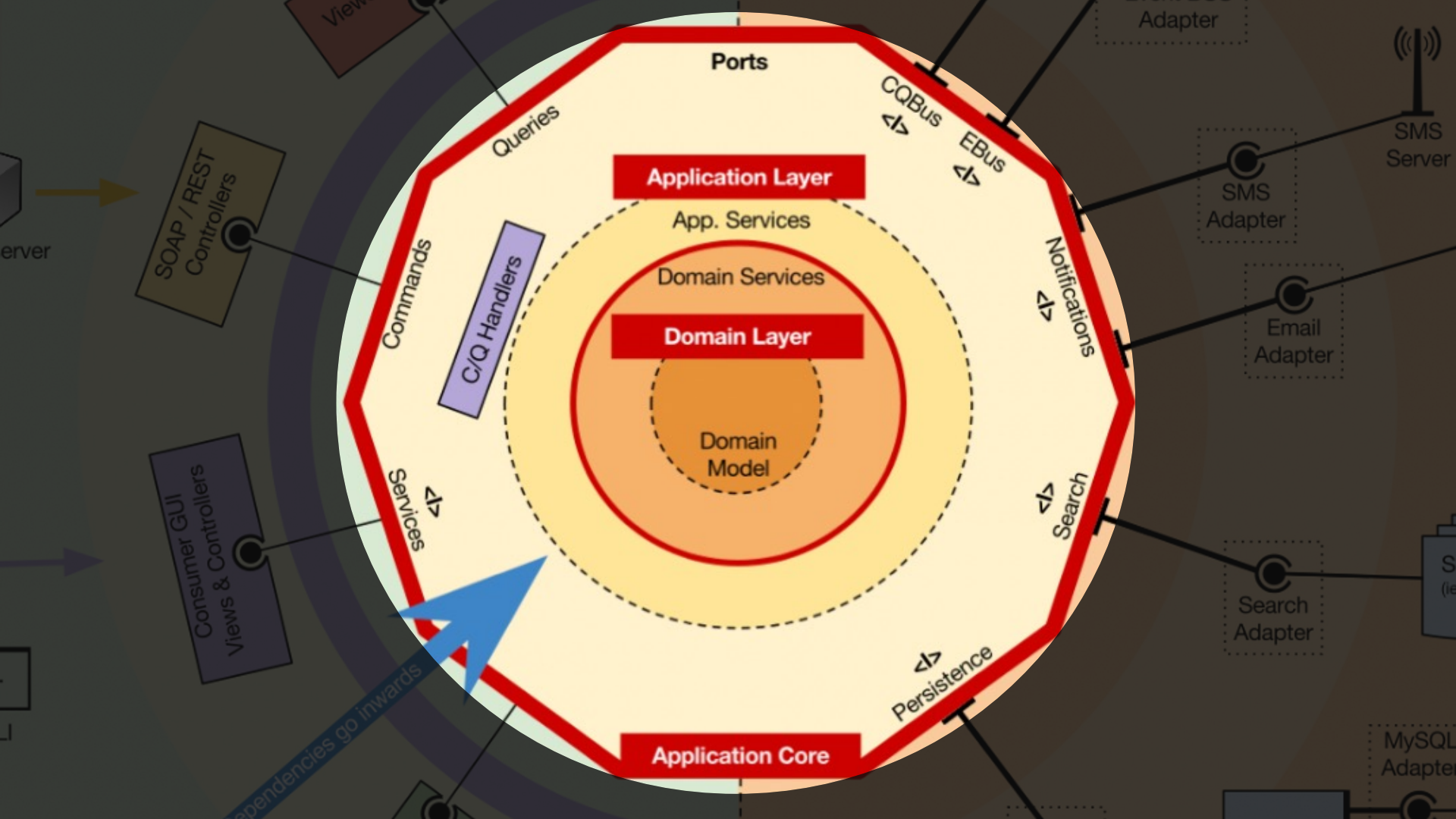


Clean architecture

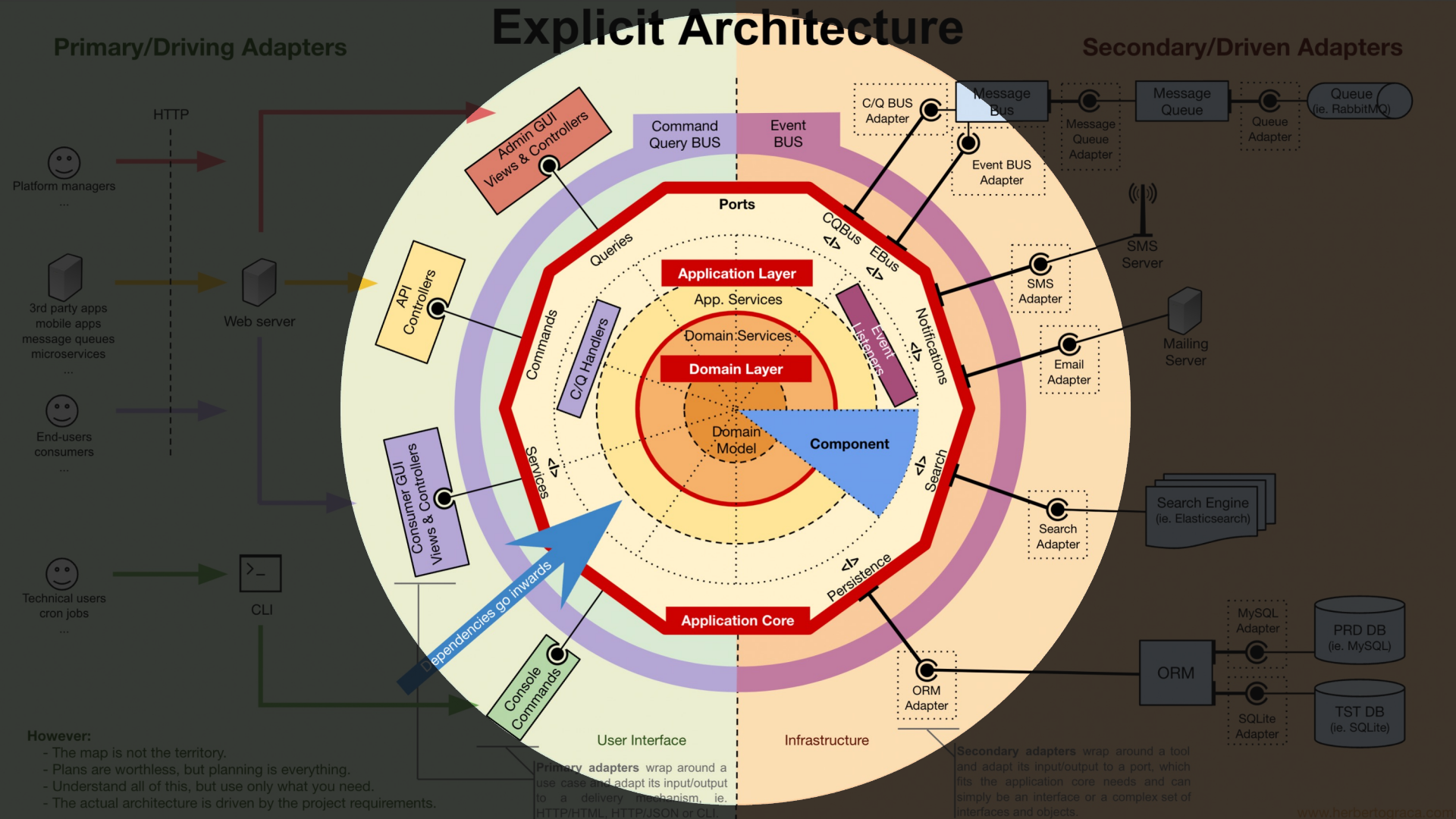
- Tydelig skille mellom de ulike deler av applikasjonen (separation of concern)
- Domene og forretningslogikk i midten
- Alt annet (DB, REST API, Tjenester) er «detaljer» og det kjenner ikke domene og forretningslogikken til
- *The Dependency Rule:* Avhengigheter kan kun peke innover til det mest sentrale: Domene og forretningslogikken
- «Screaming architecture» – man skal ganske umiddelbart kunne se hva en applikasjon faktisk gjør




Ports
Queries
CQBus
EBus
Application Layer
App. Services
Domain Services
Domain Layer
Domain Model
Commands
C/Q Handlers
Notifications
Services
Search
Persistence
Application Core
SOAP / REST Controllers
Consumer GUI Views & Controllers
Adapter
SMS Server
SMS Adapter
Email Adapter
Search Adapter
dependencies go inwards


Ports
Queries
CQBus
EBus
Application Layer
App. Services
Domain Services
Domain Layer
Domain Model
Commands
C/Q Handlers
Notifications
Services
Search
Persistence
Application Core
SOAP / REST Controllers
Consumer GUI Views & Controllers
SMS Server
SMS Adapter
Email Adapter
Search Adapter
Adapter
Dependencies go inwards

Explicit Architecture



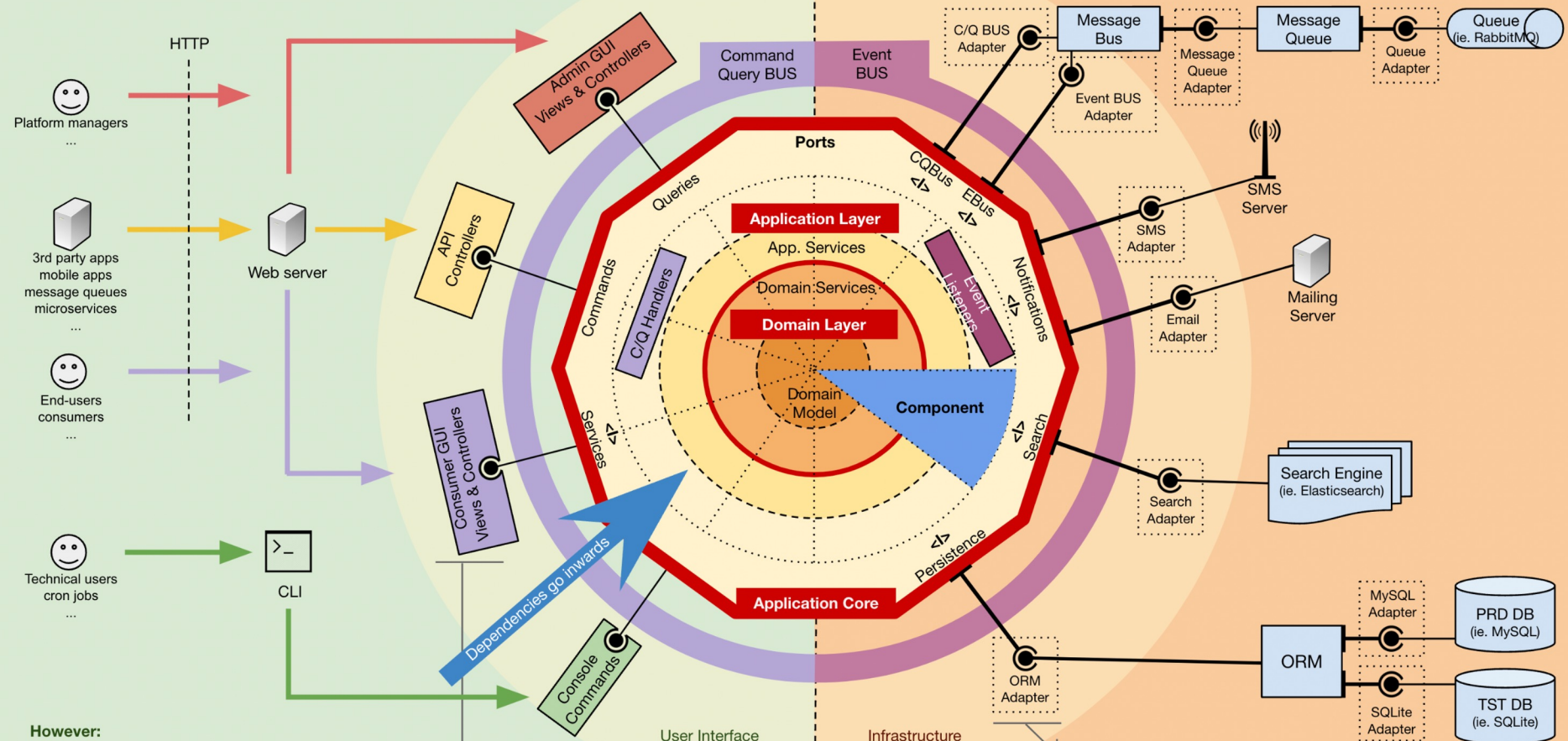
However:

- The map is not the territory.
- Plans are worthless, but planning is everything.
- Understand all of this, but use only what you need.
- The actual architecture is driven by the project requirements.

Explicit Architecture

Primary/Driving Adapters

Secondary/Driven Adapters



However:

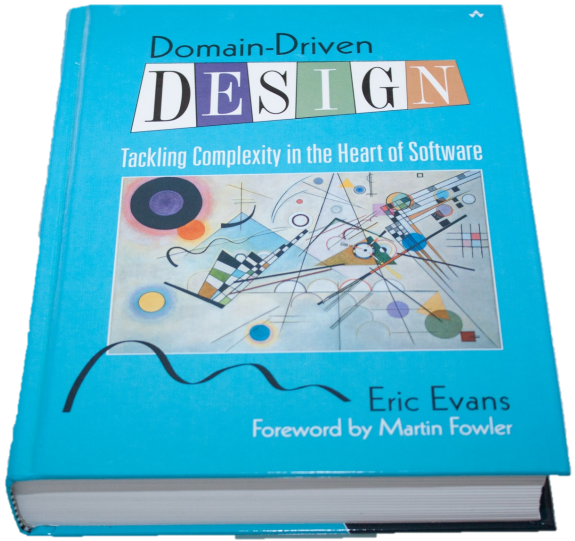
- The map is not the territory.
- Plans are worthless, but planning is everything.
- Understand all of this, but use only what you need.
- The actual architecture is driven by the project requirements.

Primary adapters wrap around a use case and adapt its input/output to a delivery mechanism, ie. HTTP/HTML, HTTP/JSON or CLI.

Secondary adapters wrap around a tool and adapt its input/output to a port, which fits the application core needs and can simply be an interface or a complex set of interfaces and objects.

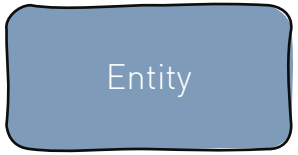
Domain Driven Design

Domain Driven Design



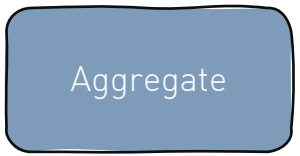
- Introdusert av Eric Evans i 2003
- En metode for å utvikle software med hovedfokus på domenet, forretningsprosesser og –regler
- En domenemodell inneholder både data og oppførsel
- Domenemodellen kan bidra til et felles språk (*ubiquitous language*) mellom utviklere og forretning

Domain Driven Design – Begreper



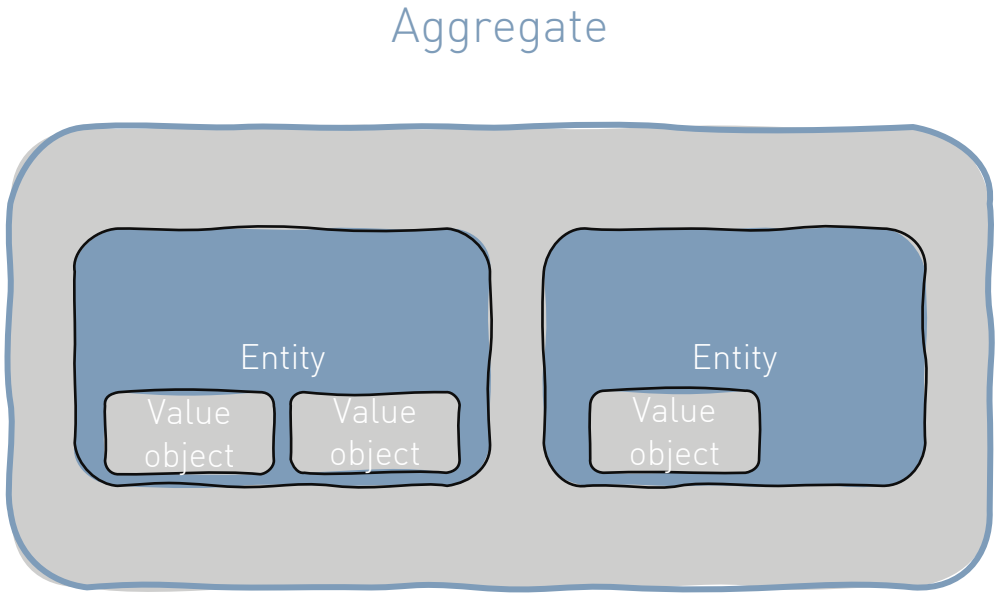
- Et objekt som inneholder verdier som kan endres over tid.
- Kan identifiseres med en id/nøkkel
- Eksempler: Kunde, Bil, Ansatt

- Et objekt som inneholder verdier som ikke kan endres.
- Identifiseres med verdiene i objektet og har ikke en unik id/nøkkel
- Eksempler: Navn, Penger, Land



- Inneholder en eller flere entities (og value objects)
- Fungerer typisk som et interface/inngangsportal for å jobbe på og manipulere entitetene

Domain Driven Design – Begreper



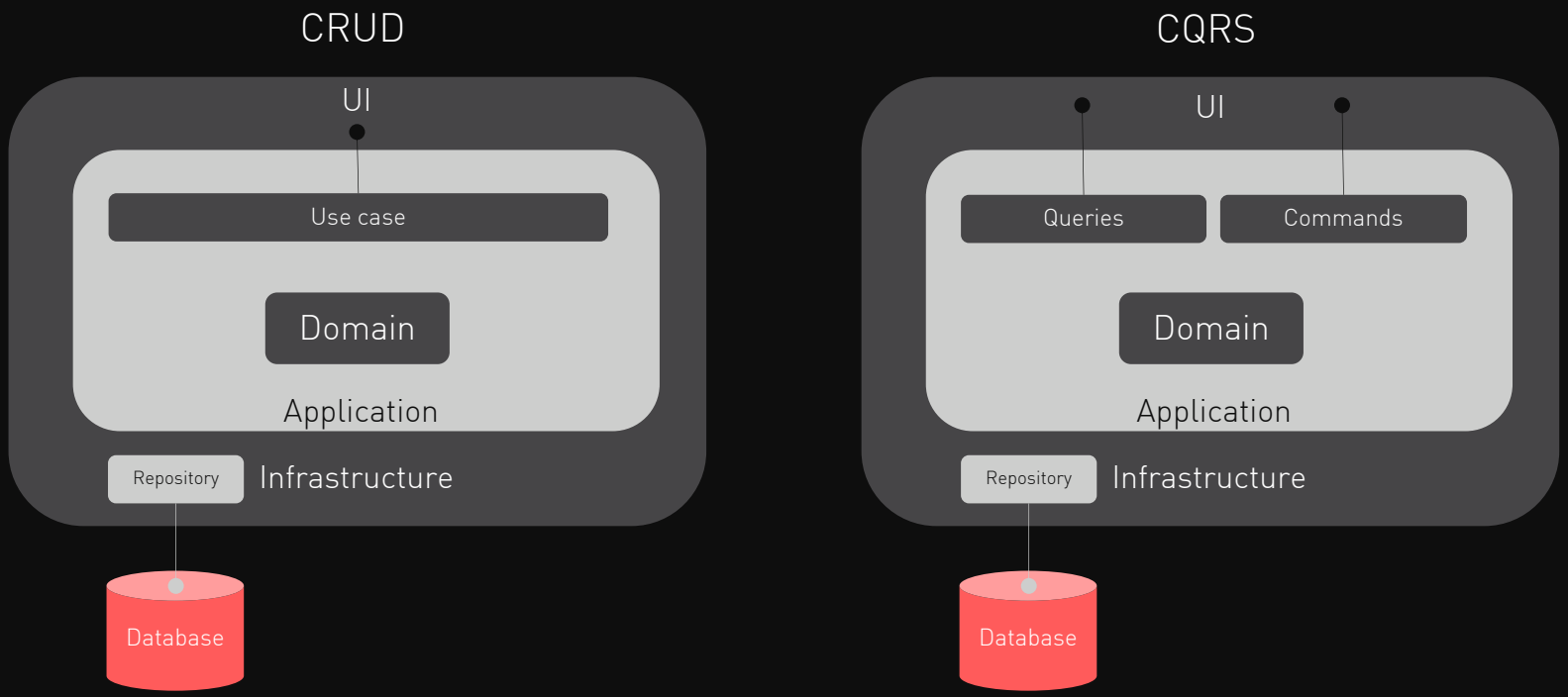
Domain Driven Design – Begreper

- Inneholder metoder for å hente, lagre og oppdatere domeneobjekter
- Eksisterer typisk for aggregater
- Kun interface i selve domenemodellen, implementeres i infrastruktur

Command and Query Responsibility Segregation (CQRS)

CQRS

- Pattern beskrevet av Greg Young
- Skiller tydelig mellom **skriving** og **lesing** av data
- *Commands* brukes for å **skrive** og **oppdatere** data (endrer tilstand)
- *Queries* brukes for å **lese** data (returnerer data, men endrer ikke tilstand)
- Brukes ofte sammen med Event Sourcing, men kan brukes uten
- Kan lese og skrive fra forskjellige datakilder
- Passer ikke for alle – i mange tilfeller kan det introdusere for mye kompleksitet


CRUD
UI
Use case
Domain
Application
Repository
Infrastructure
Database
CQRS
UI
Queries
Commands
Domain
Application
Repository
Infrastructure
Database

Hvordan er det modellert i workshopen?

Workshop

Case

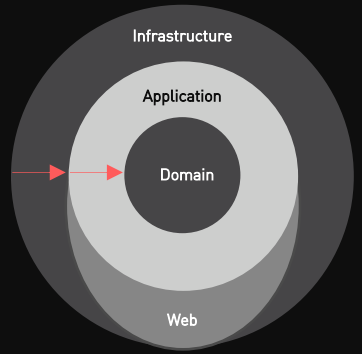
- Vi skal jobbe med å lage et lite system for å håndtere kunder og strøm
- Vi har laget en liten domenemodell med aggregates, entities og valuetypes
- Jobben deres er å implementere et sett med use caser ved bruk av clean architecture

- Kickstartere i Kotlin og C# tilgjengelig på GitHub

Struktur

Følgende struktur er opprettet i repoene:

- *Domain*
  Entiteter, valuetyper, aggregater
- *Application*
  Applikasjonslogikk, commands og query handlere
- *Infrastructure*
  Database, eksterne tjenester ++
- *Web*
  API-controllere, dependency injection setup etc



HUSK: Avhengigheter kan kun gå innover mot midten av arkitekturen

Entities & Valuetypes

Aggregates

Customer

- CustomerEntity

Entities

CustomerEntity

- Name
- ID
- Country

MeteringPointEntity

- MeteringPointID
- Name
- Address
- Power Zone

Valuetypes

- MeteringPointID (18 digits)
- Address
- Country
- CustomerID
- CustomerName
- MeteringPointName
- Period
- PowerZone (NO1, NO2, NO3, NO4 og NO5)
- UnitOfMeasurement

Oppgaver

Postman-collection finnes i repoene

- (En kunde skal kunne opprettes fra et navn, legal id, legal country.)
- (En kunde skal kunne hentes vha id)
- Alle kunder skal kunne hentes ut
- En kunde skal kunne få lagt til målepunkter (id, navn, adresse, strømsone).
- En kunde skal kunne si opp et målepunkt og beholde eventuelle andre målepunkter.
- En kunde skal kunne se detaljer om alle målepunktene sine (strømsone, adresse, et egendefinert navn f.eks. «hytta», status, type).
- En kunde skal kunne se hva forbruket har vært på et gitt målepunkt i et gitt tidsrom.

Oppsummering