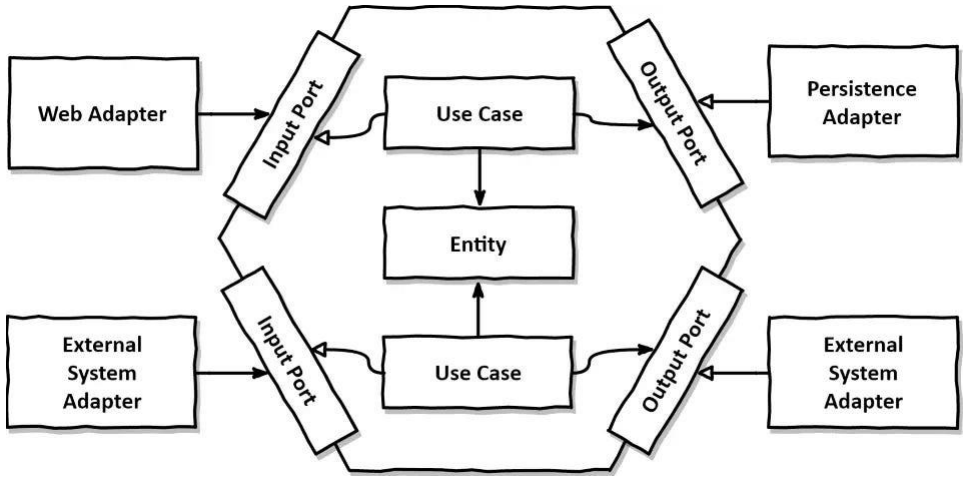
Heksagonal Arkitektur i Praksis

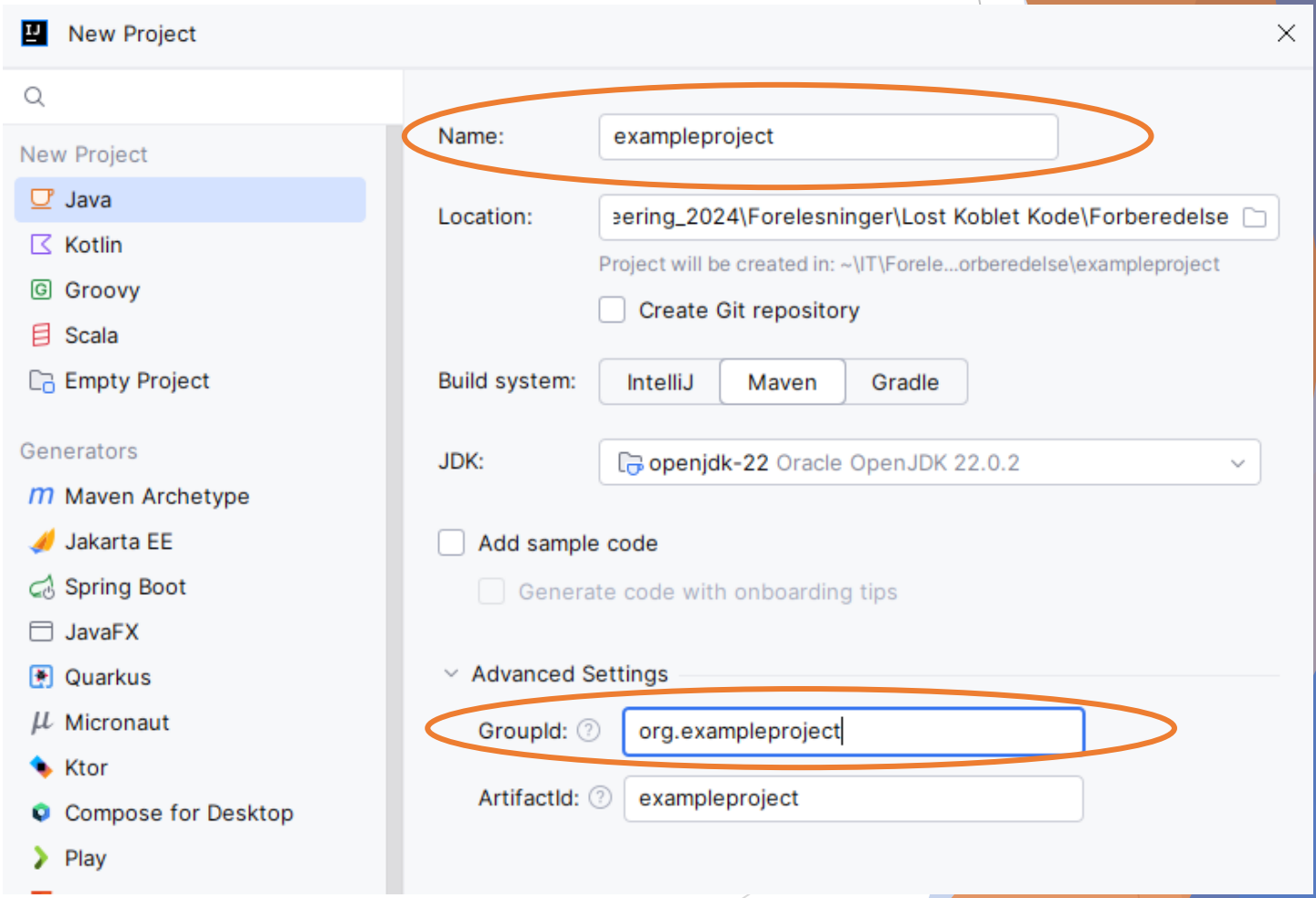
… to the best of my abilities …



https://miro.medium.com/v2/resize:fit:956/0*dZASpNdlFEaJLM83.png

Opprett maven prosjekt med domenenavn

- Opprett et maven-prosjekt og navngi det til noe passende ut ifra hva dere jobber med
  - Vanlig med små bokstaver
  - Dette blir på en måte «brand-navnet» deres
- Under Advanced Settings definer en GroupId
  - f.eks. org.<prosjektnavn>
  - eller com.<prosjektnavn>
- ... og ArtifactId
  - Typisk det samme som prosjektnavn



Prosjektet pom.xml fil

- Disse konfigurasjonene vil gjenspeiles i innholdet av prosjektets pom.xml fil:

```
<groupId>org.exampleproject</groupId>
<artifactId>exampleproject</artifactId>
<version>1.0-SNAPSHOT</version>

<properties>
   <maven.compiler.source>22</maven.compiler.source>
   <maven.compiler.target>22</maven.compiler.target>
   <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>
```

Del opp prosjektet i moduler

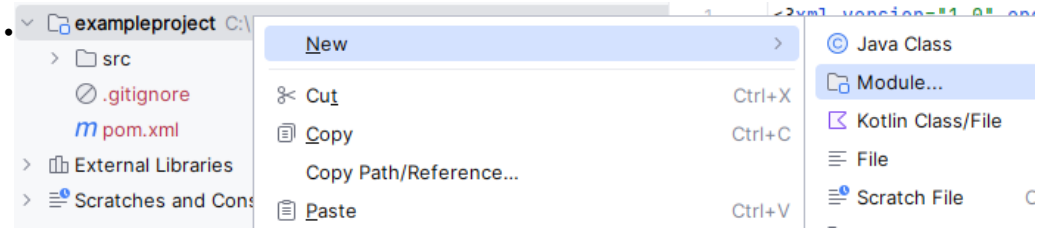
- Prosjektet vil bli overordnet modul for å samle andre prosjekt-moduler med oppdelt ansvar

- **Forslag** til prosjektmoduler
  - core – kjernekoden
  - api - REST API-et (hvis relevant)
  - web – web-applikasjonen (hvis relevant)
  - infrastructure – Implementasjon for database, melding-systemer, eksterne tjenester osv.
  - tests – samler testene
  - common – samler det som er felles over modulene (f.eks. DTOs, Exception-klasser, Utility-klasser, osv)
    - Disse er egentlig deler av core, men kan skilles ut i en egen modul

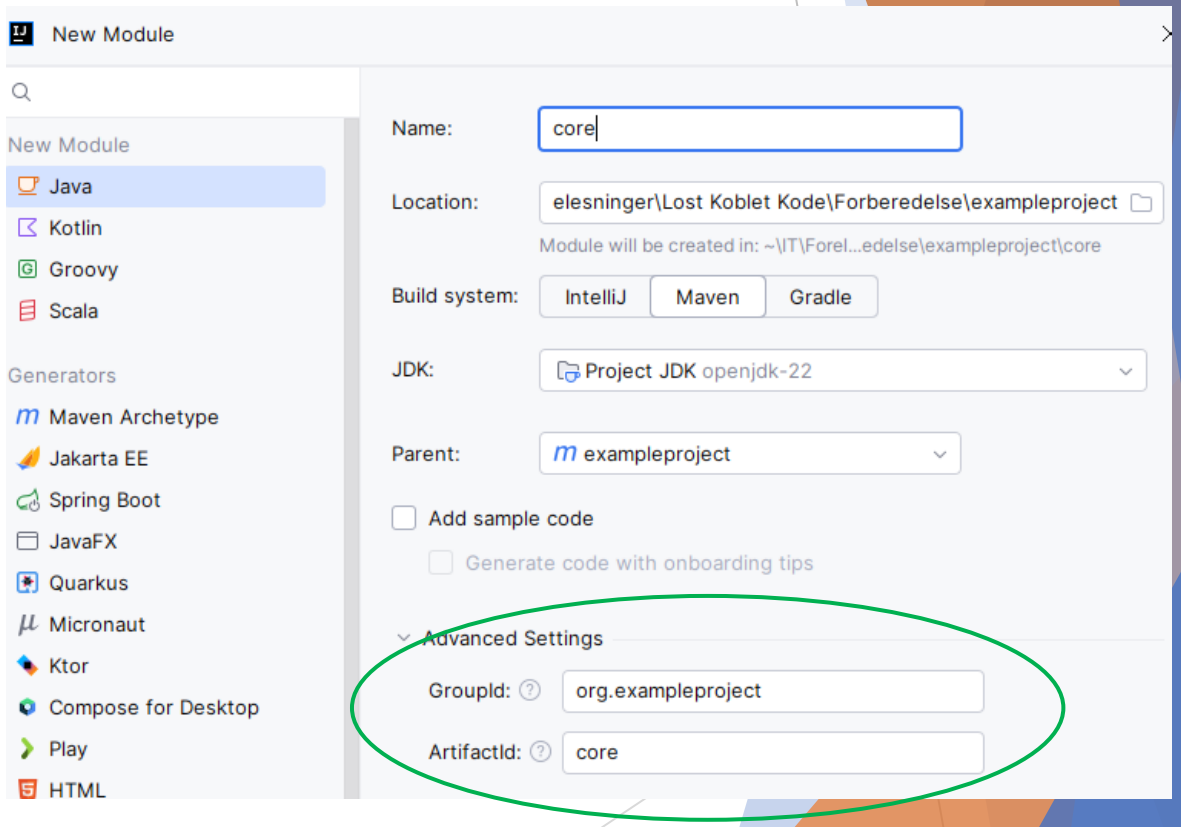
Opprett modul

- Prosjekt-modulene kan enkelt opprettes ved å høyreklikke på prosjekt-mappen

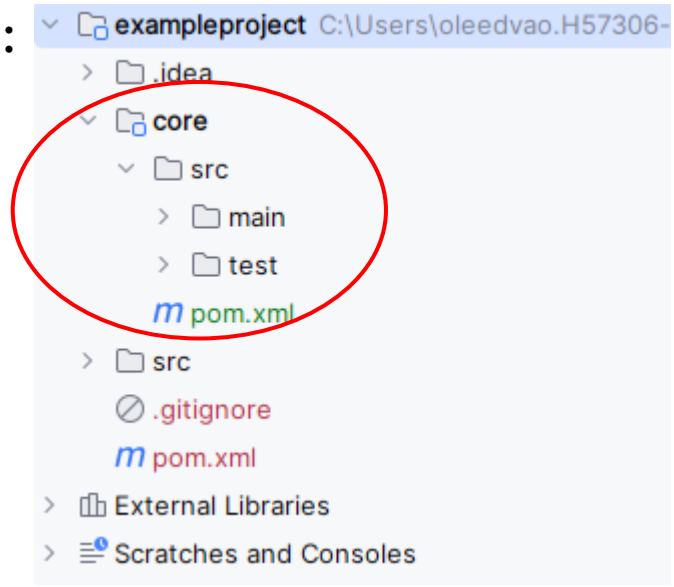
1.



2.



Resultat:



Typisk automatisk fylt inn. La Stå.

Pom.xml ved moduler

- Pom.xml filene blir automatisk oppdatert til å reflektere modul-strukturen

exampleproject:

```
<groupId>org.exampleproject</groupId>
<artifactId>exampleproject</artifactId>
<version>1.0-SNAPSHOT</version>
<packaging>pom</packaging>
<modules>
   <module>core</module>
</modules>
```

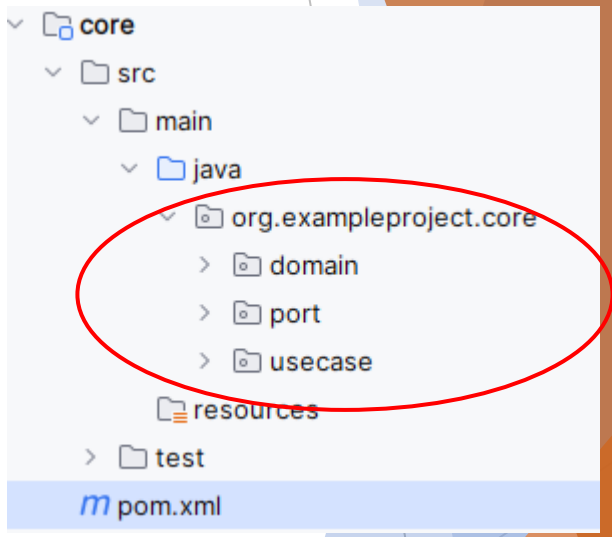
core:

```
<parent>
   <groupId>org.exampleproject</groupId>
   <artifactId>exampleproject</artifactId>
   <version>1.0-SNAPSHOT</version>
</parent>

<artifactId>core</artifactId>
```

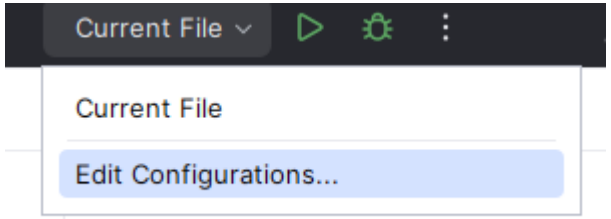
Lag en pakkestruktur i hver modul

- Lag en pakkestruktur under src/main/java i hver modul
  - Oppkall denne etter prosjektets domenenavn (samme som definert i pom.xml)
- For eksempel org.<prosjektnavn>.<modulnavn>
  - org.exampleproject.core
  - org.exampleproject.api
  - ...
- Legg alle klassefiler under denne pakkestukturen
  - Du kan fint ha flere underpakker for å strukturere klasser/interfaces
- Dette er for å oppnå gode og beskrivende import statements
  - import org.exampleproject.core.domain.SomeDataStructure
  - import org.exampleproject.core.usecase.CreateSomeDataStructureUseCase

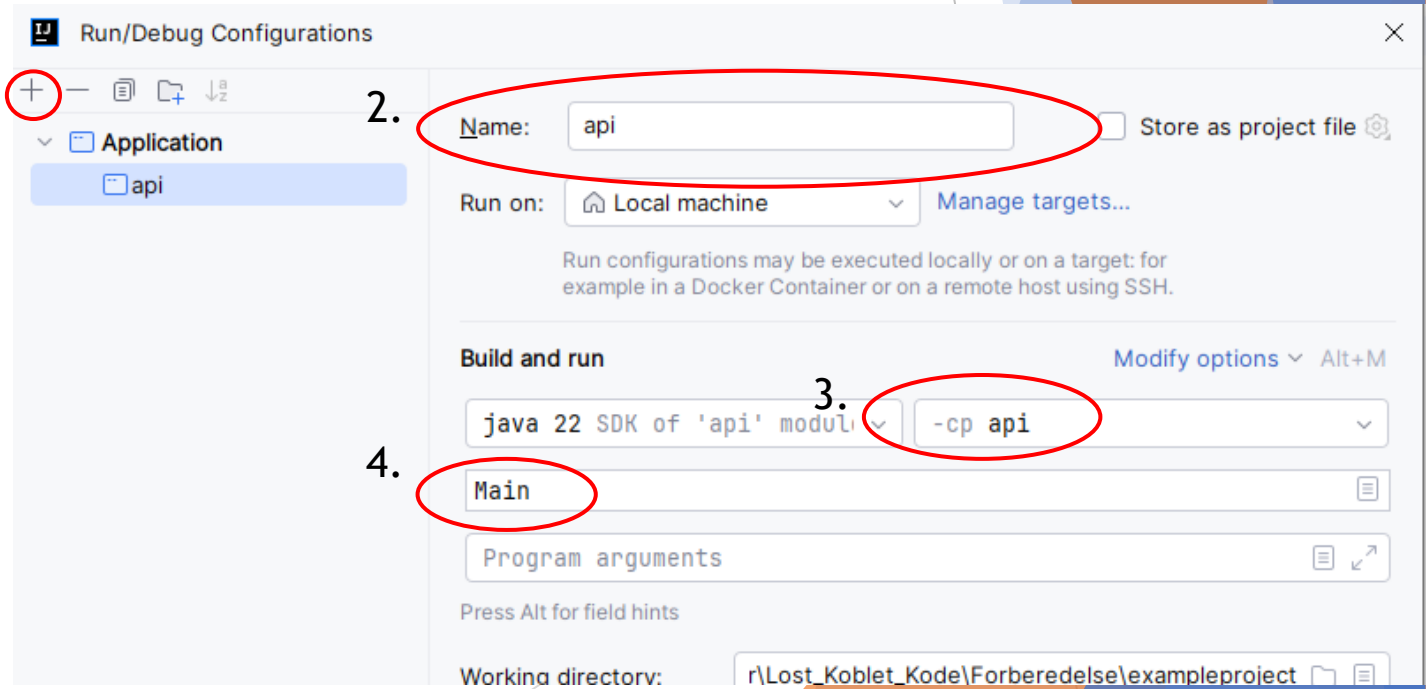


Kjøring av flere moduler

- Åpne «Edit Configurations»



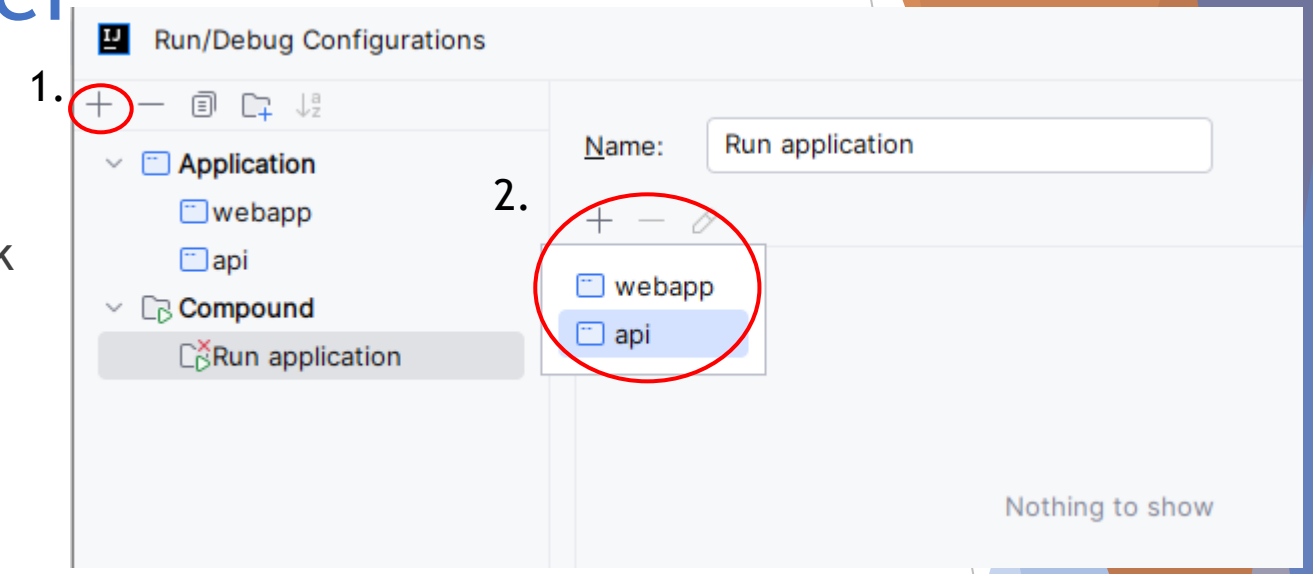
- 1. trykk på +, velg «Application» og trykk på «Add new run configuration»
- 2. Gi konfigurasjonen et passende navn. F.eks. det samme som modulen
- 3. velg modul
- 4. Velg inneholdt klasse med main-metode

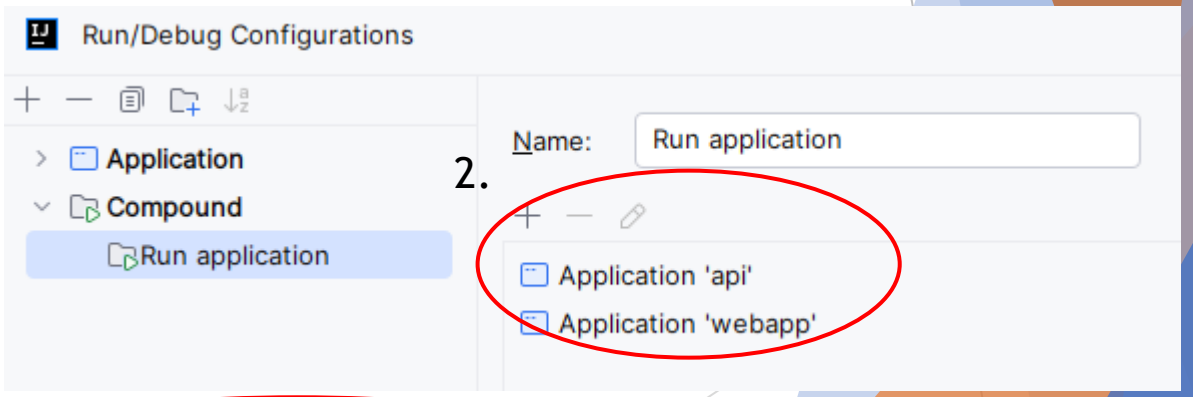


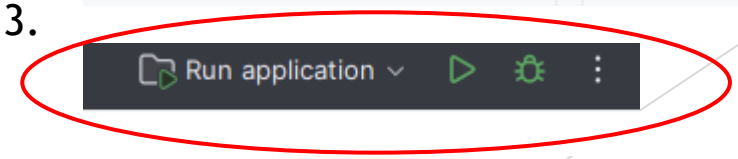
Kjøring av flere moduler

- 1. Etter å ha laget en unik run configuration for hver kjørbar modul trykk på + og velg «Compound» og gi den et passende navn
- 2 Trykk på + og velg de run configurations du ønsker å kjøre samtidig i Compounden
- 3. Etter dette kan modulene kjøres samtidig ved å kjøre den opprettede Compounden

- **Et alternativ for alt dette er å starte module-konfigurasjonene i rekkefølge manuelt**
  - **Kan potensielt være en bedre løsning**







Domene-modeller

- Det første man gjør er typisk å starte med å definere datatypene i systemet som en del av kjernen
  - Det man skal kunne opprette objekter av og behandle
  - Dette kalles ofte domene-modeller
- Man vil ofte avdekke flere domene-modeller mens man programmerer

Eksempel

- Opprette prosjekt «musicas»
  - Skal fungere som et register for artister, sanger, album osv.
- Del opp i forskjellige moduler
  - Start med core
- Definer en god pakkestruktur
- Lage noen domenemodeller

Use Cases

- Vi kan definere interaksjon med kjernen som egne use case-klasser

- Hente data
  - Fra persistent lagring
  - Generert av domene-modellene
  - ...
- Gjøre handling
  - Legge/endre/slette data
  - Sette igang en annen prosess (f.eks. publisere en melding)
  - ...

- Disse tar typisk DTOs som input
  - Dataene som er nødvendige for å utføre use caset
- Kaller evt. Porter for å «iverksette» implementasjonsspesifikke handlinger

Use Cases - Porter og Dependency Injections

- Porter benyttes typisk i use case-klasser ved bruk av Dependency Injection
- Fancy begrep, men betyr egentlig bare at klassen har en interface-variabel som instansieres i konstruktøren
  - Og som benyttes i kodelogikk...
- Typisk use case *struktur* med dependency injection av en port og en DTO for input

```
public interface SomeRepository {
    public void doSomething(DoSomethingDTO doSomethingDTO);
}
```

```
package org.exampleproject.core.usecase;

import org.exampleproject.core.dto.DoSomethingDTO;
import org.exampleproject.core.port.SomeRepository;

public class DoSomethingUseCase {

    private final SomeRepository someRepository;

    public DoSomethingUseCase(SomeRepository someRepository) {
        this.someRepository = someRepository;
    }

    public void execute(DoSomethingDTO doSomethingDTO) {

        someRepository.doSomething(doSomethingDTO);

    }
}
```

Eksempel

- Lag use cases for å
  - Opprette ny artist
  - Hente en artist
- Benytt dependency injection for porter
- Benytt DTOs for å standardisere input

Avhengigheter mellom moduler

- Hvis vi har en modul som trenger å aksessere noe i en annen intern modul definerer vi dette som en avhengighet i pom.xml
  - Typisk moduler som er avhengig av core

```
<parent>
    <groupId>org.exampleproject</groupId>
    <artifactId>exampleproject</artifactId>
    <version>1.0-SNAPSHOT</version>
</parent>

<artifactId>tests</artifactId>

...

<dependencies>
    <dependency>
        <groupId>org.exampleproject</groupId>
        <artifactId>core</artifactId>
        <version>1.0-SNAPSHOT</version>
    </dependency>
</dependencies>
```

Testing

- Kan være lurt å lage en egen modul for testing
  - ... i stedet for å gjøre dette direkte i de andre modulene
  - Samler alle tester å ett sted
- Krever at test-modulen har definerte avhengigheter til de andre modulene

Eksempel

- Lag en modul tests
- Legg til core som en avhengighet
- Test use casene