Hendelser

INF101 forelesning 17. mars 2023

Torstein Strømme

Stikkord: funksjonelle grensesnitt, lambda, eventbuss

I dag

- Funksjonelle grensesnitt
- Lambda-uttrykk

- Eventbuss
  - Gjør ferdig applikasjon fra tirsdag

Funksjonelle grensesnitt

- Grensesnitt med én metode
- Syntaksen :: gjør at vi kan behandle funksjoner som objekter

```
this.timer = new Timer(model.getTimerDelay(), this::clockTick);
```

Funksjonelle grensesnitt

- Grensesnitt med én metode
- Syntaksen :: gjør at vi kan behandle funksjoner som objekter

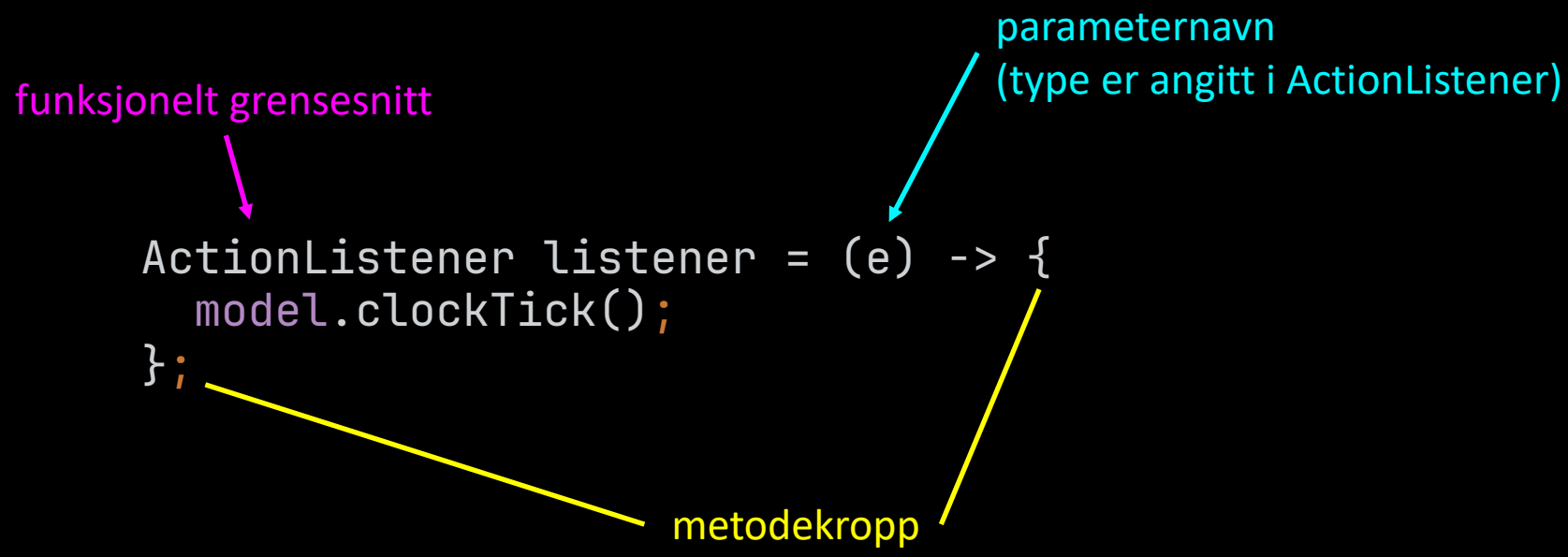
```
ActionListener listener = this::clockTick;
this.timer = new Timer(model.getTimerDelay(), listener);
```

```
public interface ActionListener {
    /**
     * Invoked when an action occurs.
     * @param e the event to be processed
     */
    public void actionPerformed(ActionEvent e);
}
```

```
public void clockTick(ActionEvent evt) {
    // ...
}
```

Lambda-uttrykk

- Oppretter objekter i et funksjonelt grensesnitt uten å navngi metoden
- «anonyme metoder»



Eventbuss

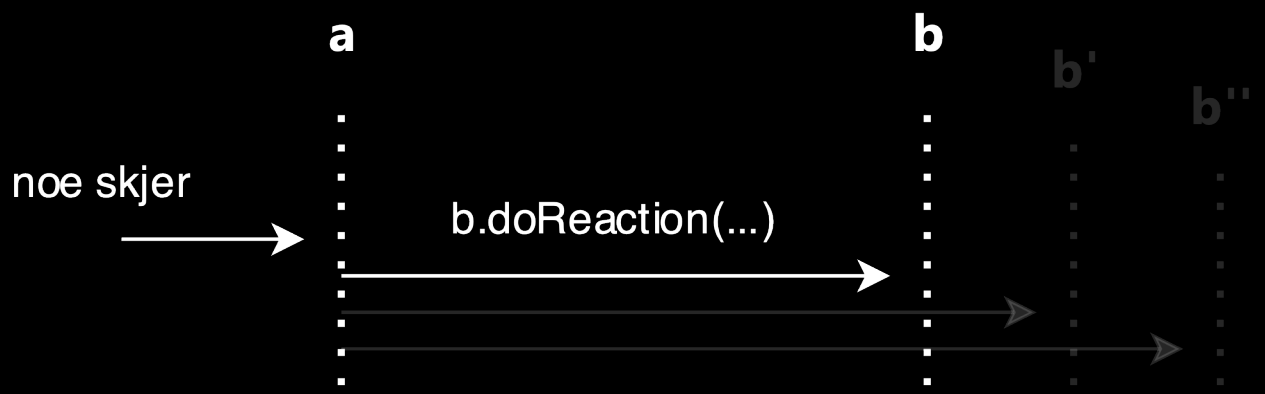
- Kommunikasjon mellom ulike deler av et program som opprettholder høy modularitet

- Benyttes når «en hendelse i a skal føre til en hendelse i b»

Vi har sett før: a kaller på b

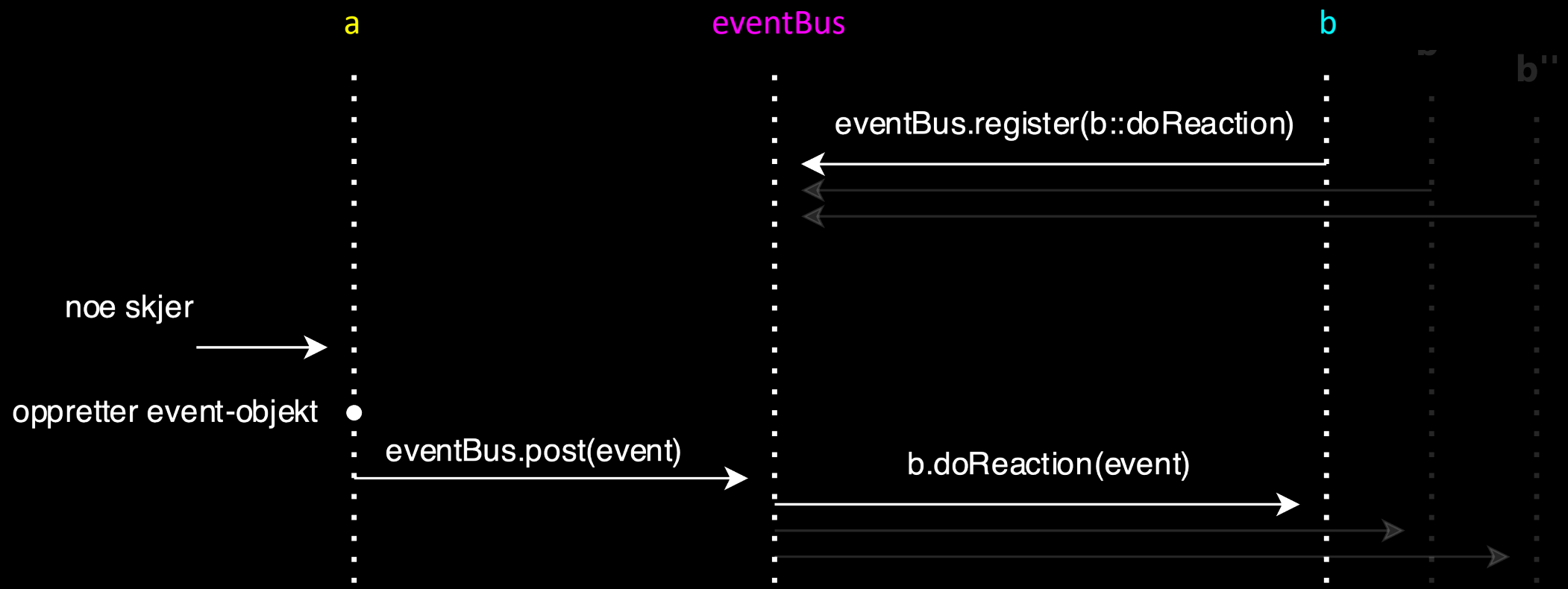

```java
public class A {
  private B b;

  public A(B b) {
    this.b = b;
  }

  public void somethingHappens(String msg, int num) {
    System.out.println(this + " method called with args: " + msg + ", " + num);
    this.b.doReaction(msg, num);
  }
}
```

Vi har sett før: a kaller på b

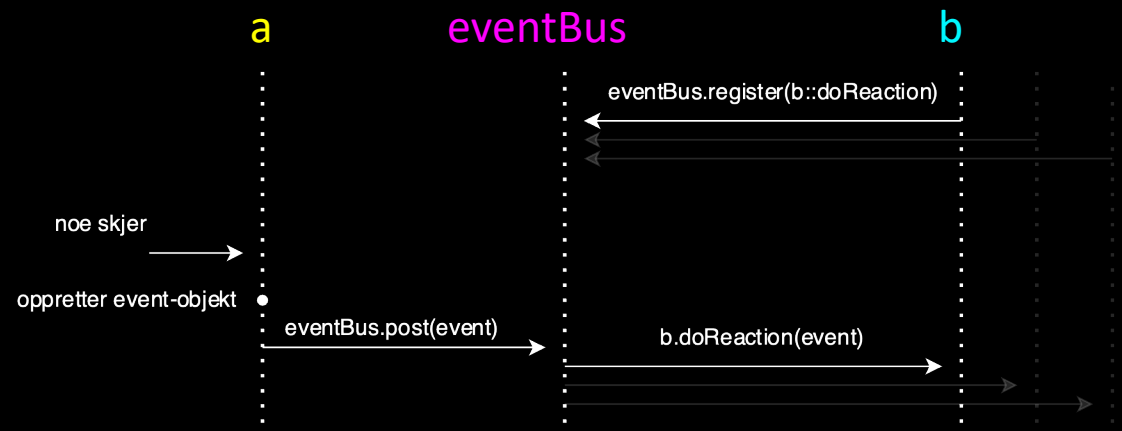


- Ulemper:
  - a må vite b sin type og hvilken metode som skal kalles → lav modularitet
  - metoden i b som skal kalles må være public → dårlig innkapsling
  - a må vite hvor mange b'er det er

Eventbuss


a
eventBus
b
b''
eventBus.register(b::doReaction)
noe skjer
oppretter event-objekt
eventBus.post(event)
b.doReaction(event)

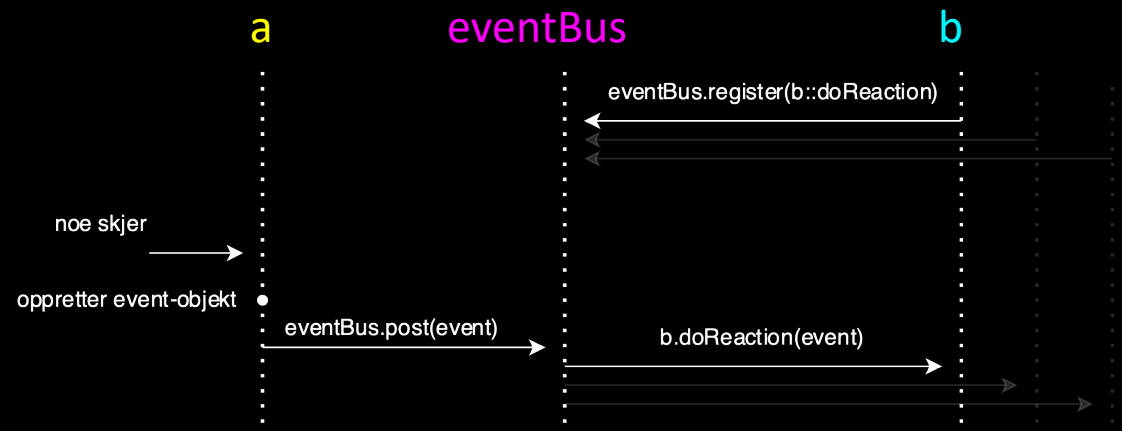
Eventbuss




```java
public record MyEvent(String msg, int num) {}
```

```java
public class A {
  private MyEventBus eventBus;

  public A(MyEventBus eventBus) {
    this.eventBus = eventBus;
  }

  public void somethingHappens(String msg, int num) {
    System.out.println(this + " method called with args: " + msg + ", " + num);
    this.eventBus.post(new MyEvent(msg, num));
  }
}
```

Eventbuss

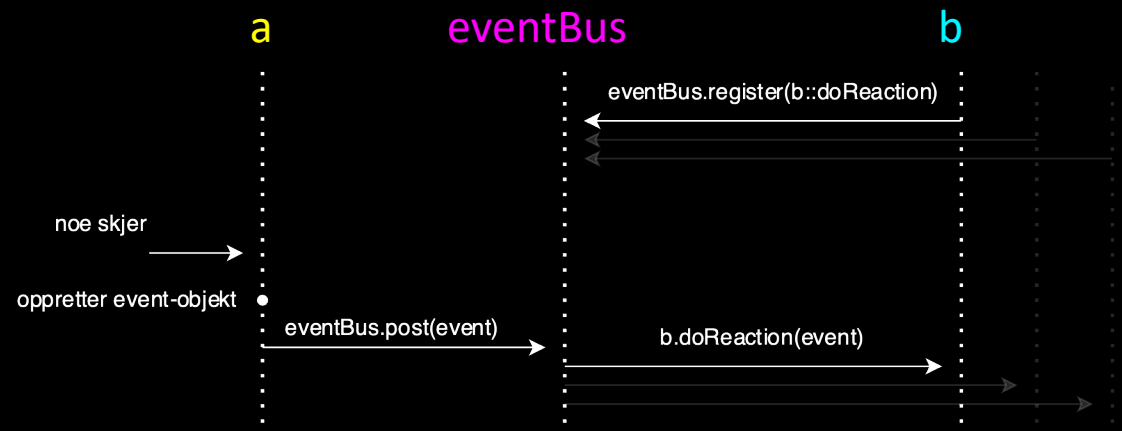



```java
public record MyEvent(String msg, int num) {}
```

```java
public class A {
  public class MyEventBus {
    private List<MyEventHandler> eventHandlers = new ArrayList<>();

    public void register(MyEventHandler eventHandler) {
      this.eventHandlers.add(eventHandler);
    }

    public void post(MyEvent event) {
      for (MyEventHandler eventHandler : this.eventHandlers) {
        eventHandler.handle(event);
      }
    }
  }
}
```

```java
@FunctionalInterface
public interface MyEventHandler {
  void handle(MyEvent event);
}
```

Eventbuss

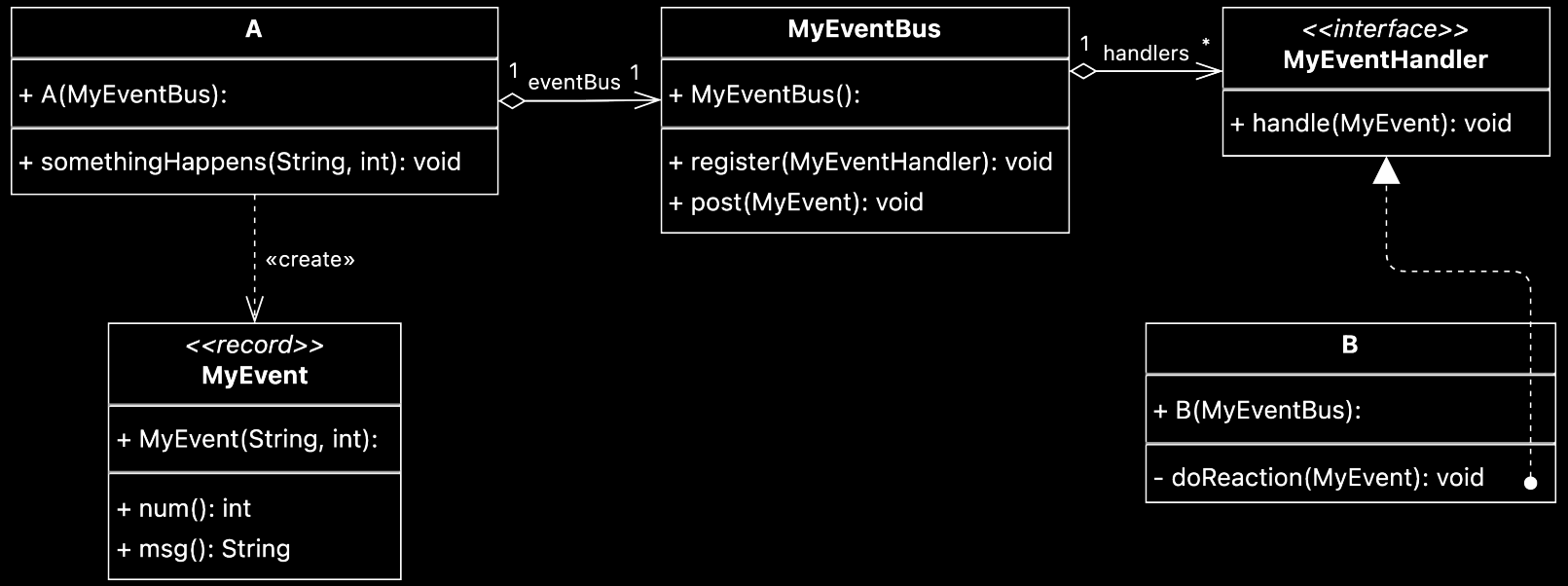



```java
public record MyEvent(String msg, int num) {}
```

```java
public class A {
  public class MyEventBus {
    public class B {

      public B(MyEventBus eventBus) {
        eventBus.register(this::doReaction);
      }

      private void doReaction(MyEvent event) {
        String msg = event.msg();
        int num = event.num();
        System.out.println(this + " reacts to event w/info: " + msg + ", " + num);
      }
    }
  }
}
```

```java
@FunctionalInterface
public interface MyEventHandler {
  void handle(MyEvent event);
}
```

Eventbuss

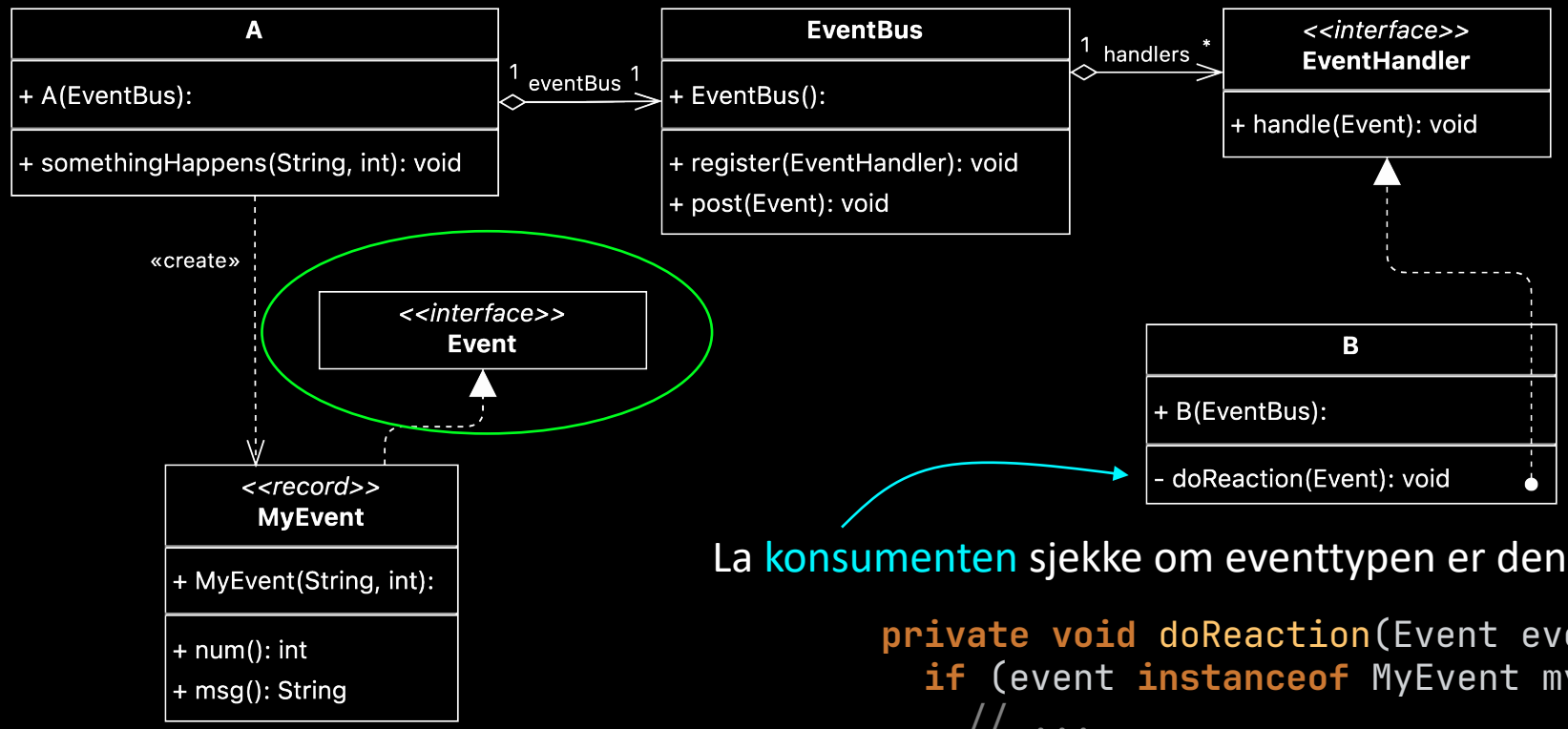

A
+ A(MyEventBus):
+ somethingHappens(String, int): void
1
eventBus
1
MyEventBus
+ MyEventBus():
+ register(MyEventHandler): void
+ post(MyEvent): void
1
handlers
*
<<interface>>
MyEventHandler
+ handle(MyEvent): void
«create»
<<record>>
MyEvent
+ MyEvent(String, int):
+ num(): int
+ msg(): String
B
+ B(MyEventBus):
- doReaction(MyEvent): void

Eventbuss

- Ulemper
  - Litt mer kompleksitet (én gang)
- Fordeler
  - Litt mindre kompleksitet (potensielt mange ganger)
  - Høyere modularitet
  - Bedre innkapsling

Generell eventbuss

- La «Event» være en supertype over alle typer eventer

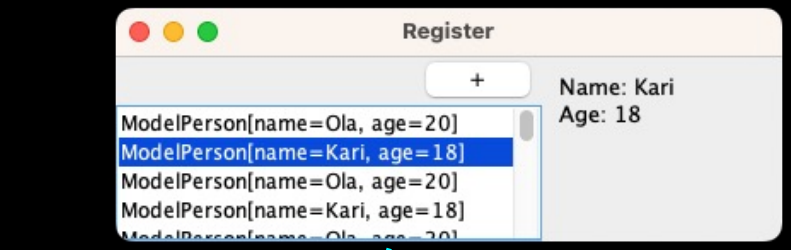


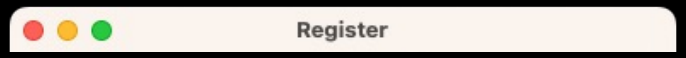
La konsumenten sjekke om eventtypen er den som skal reageres på

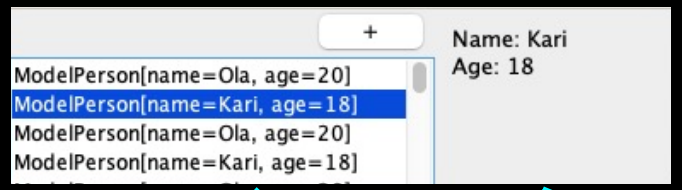
```
private void doReaction(Event event) {
  if (event instanceof MyEvent myEvent) {
    // ...
  }
}
```

Flashback

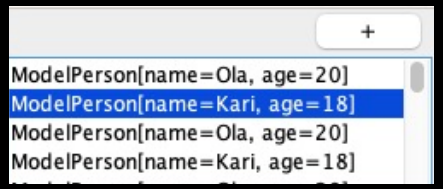
JFrame





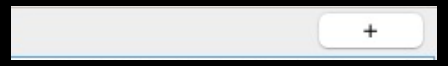


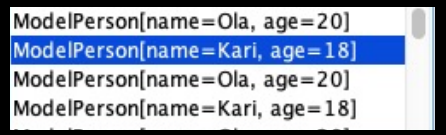
ViewMain



ViewPerson

ViewCreatePerson

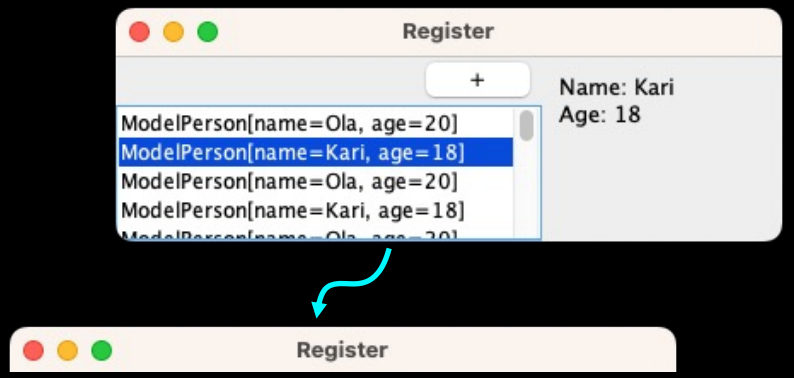




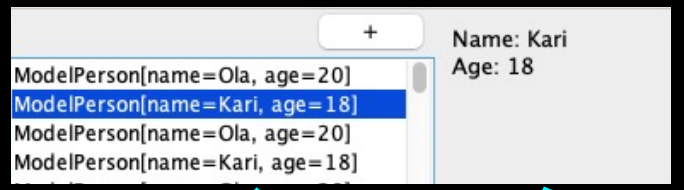
ViewListOfPeople

Flashback

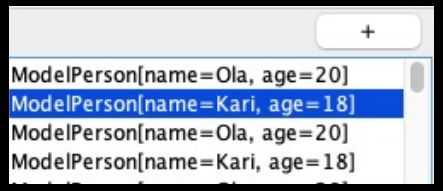
JFrame



for å endre hva som vises, må vi skrive kode her



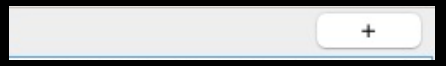
ViewMain

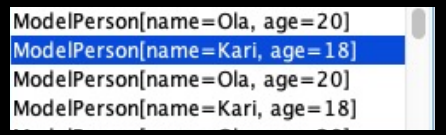


klikk fra brukeren kan vi lytte til her

ViewPerson

ViewCreatePerson

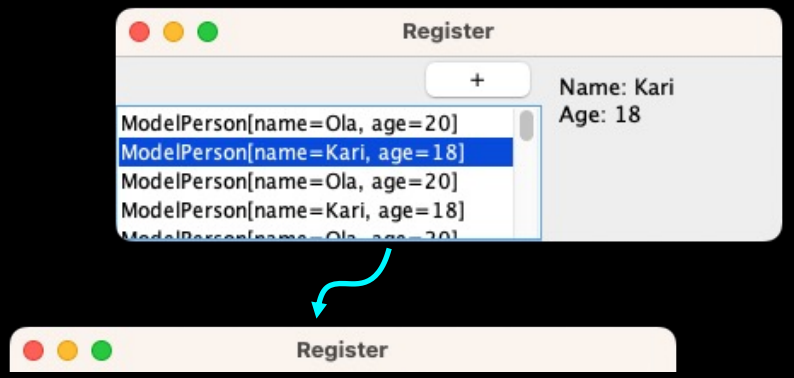




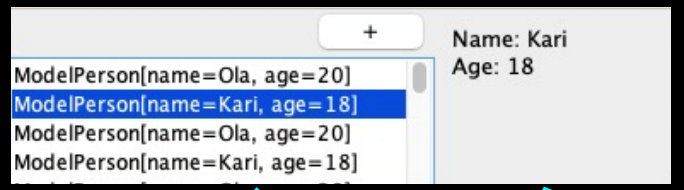
ViewListOfPeople

Flashback

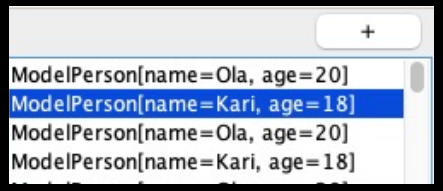
JFrame



konsumerer hendelse

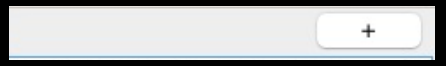


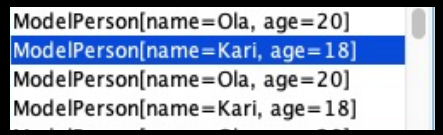
ViewMain



produserer hendelse

ViewPerson





ViewListOfPeople

ViewCreatePerson

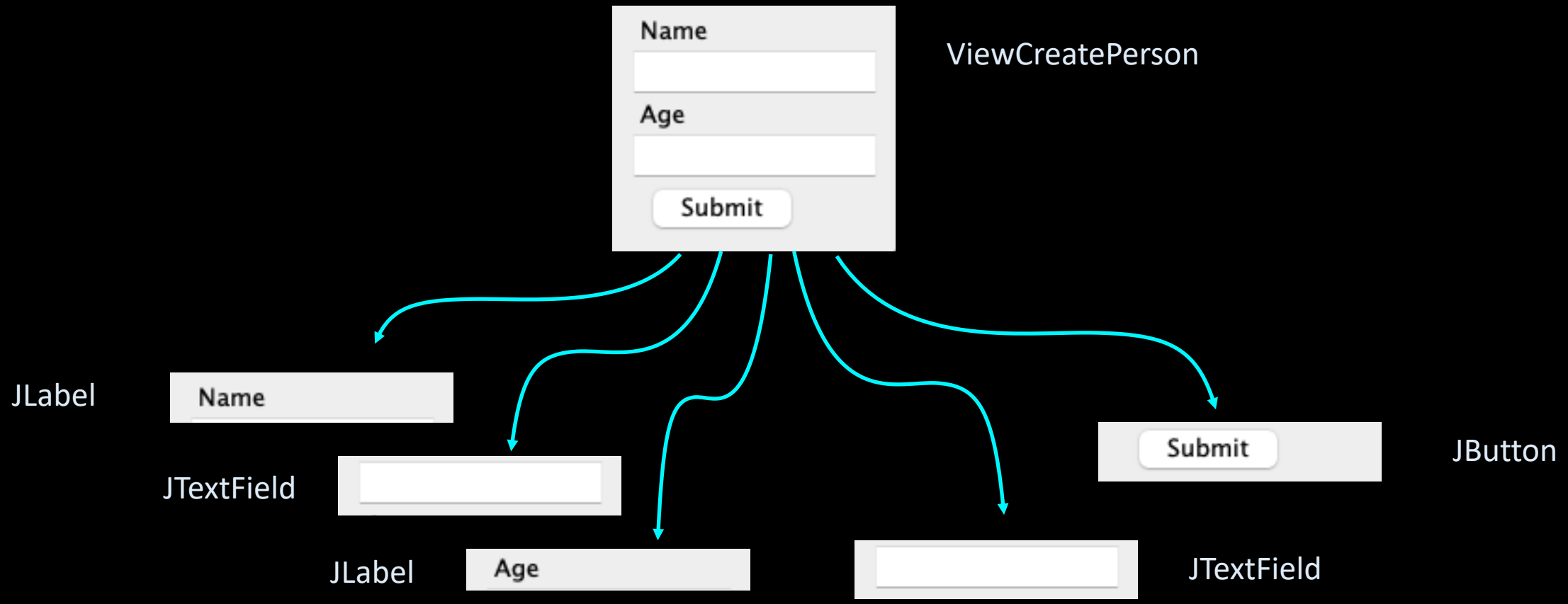
Plan

- Opprett en EventBus i Main, og la ViewMain og ViewListOfPeople få den gitt som argument ved konstruksjon

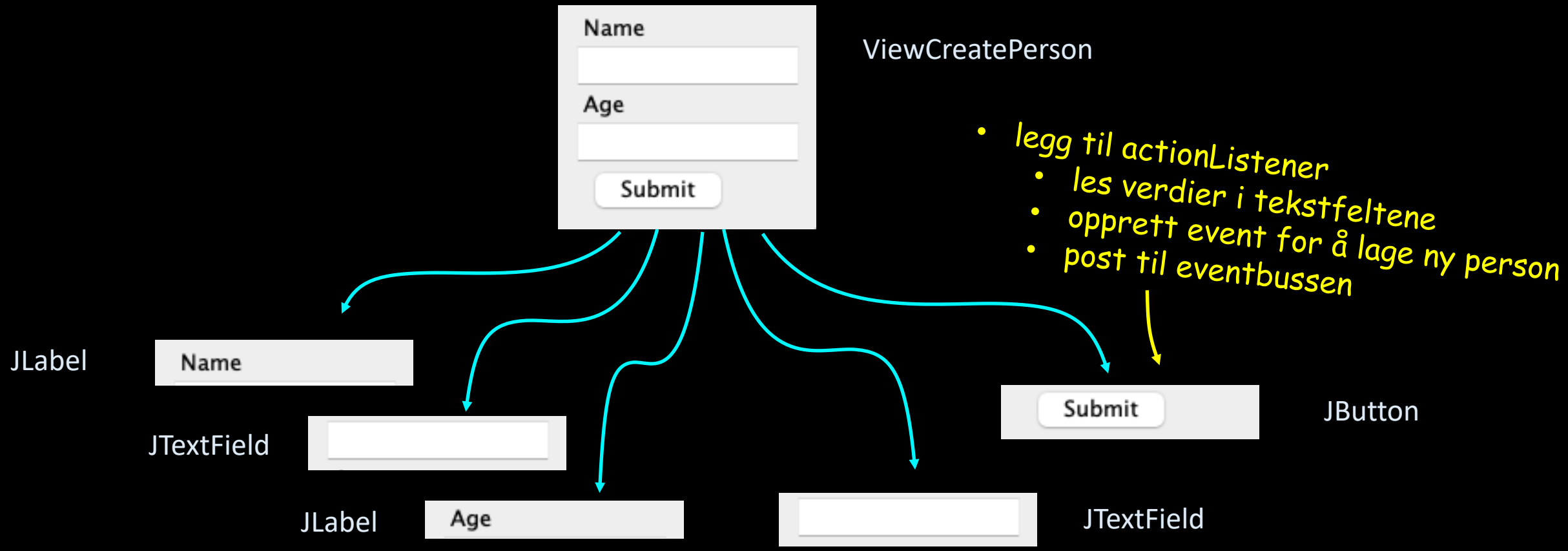
- Opprett en eventtype for at en person i listen blir valgt av brukeren
- Legg til en lytter for museklikk på JList-objektet
  - Hent ut hvilket Person-objekt som er valgt fra JList-objektet
  - Opprett en event for dette, og post den i eventbussen

- La ViewMain registrere seg som konsument, og håndter eventen slik at panelet byttes

Panel for å legge til nye personer



Panel for å legge til nye personer



Neste steg

- Legg til en knapp som tar oss til ViewOrCreatePerson –skjermen
- Opprett en kontroller som konsumerer eventene og oppdaterer modellen

- For å oppdatere listen som viser modellen: observer

Legg til knapp i ViewMain