

# Extensible Records i Elm

- Tryggere updates og røddigere kode (som er mye enklere å teste)

# Del 1: Update-funksjoner

## I en `update` nær deg

...Fungerer enkle updates ganske bra:

```
update : Msg -> Model -> (Model, Cmd Msg)
update msg model =
  case msg of
    {-diverse-}

    -- Lettlest: ny model = gammel model, med nytt `name`-felt
    ChangeName newName ->
      ( {model | name = newName }, Cmd.none)

    {-diverse-}
```

# Men Proto har ikke bare enkle updates

Denne er mindre lettlest, men får plusspoeng for å være eksplisitt i hva som skjer:

```
GeneralContextMenuOnSubmit ->
  case model.generalMenuContext of
    DetailOption _ menuOption ->
      case menuOption of
        AddNotificationSubscription ->
          ( model, Effect.none )

        AddToSelection ->
          let
            fieldId =
              Document.ContextMenu.AddToSelection.customEntryTitleFieldID

            validatedTitleField =
              model.form
              |> Form.parse fieldId Form.stringParser
              |> Form.validateMaxLength (Input.inputCharLengthToInt Input.ExtraShort)

            validatedModel =
              { model | form = Form.persistErrors validatedTitleField model.form }
          in
            case Form.toResult validatedTitleField of
              Err _ ->
                ( validatedModel
                  , Effect.focusElement <|
                    Form.firstErrorID [ fieldId ] validatedModel.form
                )

              Ok _ ->
                case model.addToSelectionContext of
                  Context.MostRecentlyUsedSelection ->
                    submitAddToMostRecentlyUsedSelection
                      (Selection.lastEditable shared.selections)
                      (Implementation.getSearchTerm model.legalSourcesModel)
                      route.url
                      validatedModel

                  Context.OtherSelection ->
                    ( { validatedModel
                      | generalMenuContext = DetailOption (Modal.Succeeding Modal.Third) AddToOtherSelections
                    }
                      , Selections.fetchEditableSelections
                    )

                  Context.NewSelection ->
                    ( { validatedModel | generalMenuContext = DetailOption (Modal.Succeeding Modal.Third) AddToNewSelection }
                      , Effect.none
                    )

                AddToOtherSelections ->
                  submitAddToOtherSelections
                    shared.settings
                    (Implementation.getSearchTerm model.legalSourcesModel)
                    route.url
                    model

                AddToNewSelection ->
                  submitToNewSelection
                    (Implementation.getSearchTerm model.legalSourcesModel)
                    route.url
```

## Så da lager vi funksjoner. Eller?

```
update : Msg -> Model -> (Model, Cmd Msg)
update msg model =
  case msg of
    [...] diverse]

    ComplexMsg complexPayload ->
      -- Hva er model etter `handleComplexity`?
      model |> handleComplexity complexPayload

    [...] diverse]

handleComplexity : ComplexData -> Model -> Model
handleComplexity somethingComplex model =
  Debug.todo "implement brilliantly"
```

Typesignaturen til `handleComplexity` forteller oss at den tar en `Model` inn og spytter en `Model` ut igjen. Hvordan forholder vi oss til dette?

## Alternativ 1?

Vi kan håpe at neste utvikler som rører koden 1) holder tunga rett i munnen og 2) sjekker implementasjonen manuelt på `handleComplexity` sånn at hun vet hvilke felter som kan endres av en `ComplexMsg` (JavaScript Style™).

Bare litt synd at det begynner å bli mange `Msg` å sjekke etterhvert...

## Alternativ 2?

Vi kan slutte med delegate-funksjoner inni `update`, sånn at alle modell-endringer vises inline helt eksplisitt uten noe hokus-pokus.

Men `elm-review` maser allerede om cognitive complexity, og den `update`-blokka begynner å bli *lang*...

## Alternativ 3!?

Vi kan spille på lag med Elm, som ikke bare er funksjonelt og "sterkt typet" ( `Model -> Model` ), men som også støtter **Extensible Records**.

En "extensible record" er **en record som har "minimum disse gitte feltene"**.

```
type alias SomethingWithAge a =  
  { a  
    | age : Int  
  }  
  
type alias SomethingWithSize a =  
  { a  
    | height : Int  
    , width : Int  
  }
```

## Tilbake til delegate-funksjonen vår

```
handleComplexity : ComplexData -> Model -> Model
handleComplexity somethingComplex model =
  let
    complexity1
      = {- se for deg noe sykt komplekst -}
    complexity2
      = {- se for deg noe enda mer komplekst -}

    scrollPosition = multiply complexity1 complexity2
  in
  -- Selve _endringen i modellen_ er ikke kompleks i det hele tatt:
  ( { model | scrollPosition = scrollPosition } , Cmd.none )
```

# Extensible Record Upgrade™

```
-- JavaScript style: her må man lese gjennom implementasjonen
handleComplexity : ComplexData -> Model -> Model
handleComplexity somethingComplex model =
    {-diverse-}
    ( { model | scrollPosition = scrollPosition } , Cmd.none )

-- Elm Style! Vi spiller på lag med typene:
type alias HasScrollPosition a =
    { a -- `a` her er Hva Som Helst™, inkludert en tung `Model` med mye greier
      | scrollPosition : Int
    }

handleComplexity : ComplexData -> HasScrollPosition a -> HasScrollPosition a
handleComplexity somethingComplex model =
    {-diverse-}
    ( { model | scrollPosition = scrollPosition } , Cmd.none )
```

## Wins (ikke-uttømmende)

- i en `update` kan man se på typesignaturen til en delegate hvilken del av modellen som berøres (slipper å skumme en potensielt lang og kompleks implementasjon)
- hvis neste utvikler plutselig begynner å endre flere felter enn det typesignaturen tilsier, vil compileren si ifra
- `handleComplexity` **kan testes uten å måtte mocke en potensielt stor `Model`**
- enkelt: i praksis kan man ofte fjerne typesignatur, og la LSP komme med forslag basert på hvilke felter som berøres!

## Case Study: AISearch.elm:update

Merk: Man kan gjøre det samme med `Shared.Model` og `Settings` osv (selv om det er viktigst med `Model` fordi `Model` er det som returneres.)

## Del 2: Views

```
type alias Settings =  
  { apiUrl : ApiUrl.ApiUrl  
  , basePath : String  
  , colorScheme : ColorScheme.ColorScheme  
  , device : Device  
  , language : Language.Language  
  , translations : I18Next.Translations  
  , envSettings : EnvSettings  
  , features : FeatureFlags  
  , isForgePreview : Bool  
  , initialResizeAdjustment : Maybe Int  
  , initialTocOpen : Bool  
  }  
-- ...eller bare  
type alias Settings a =  
  { a | translations : I18Next.Translations }
```

## Lag typer for det du trenger

Kan `prettyPrintName` være mer spesifikk og mindre kravstor?

```
prettyPrintName : Settings -> Model -> Html msg
prettyPrintName { language } { firstName, lastName} =
  Html.span [] [
    Html.text case language of
      Language.Formal ->
        lastName ++ ", " ++ firstName
      Language.Casual ->
        firstName ++ " " ++ lastName
  ]
```

# Med extensible records

```
type alias HasLanguage a =  
  { a  
    | language : Language.Language  
  }  
  
type alias HasName a =  
  { a  
    | firstName : String  
    , lastName : String  
  }  
  
-- ny, smalere typesignatur:  
prettyPrintName : HasLanguage a -> HasName b -> Html msg  
prettyPrintName { language } { firstName, lastName} =  
  -- Implementasjon som før
```

## Del 3: Phantom types med Extensible Records

```
main : Html msg
main =
    DomainList.new [ "one word", "bad word", "other word" ]
        |> DomainList.sort -- kan vi få compileren til å enforce sortering?
        |> DomainList.sanitize -- hva med det her; kan det være non-optional?
        |> DomainList.render

-- Spoiler: Ja.
```

# DomainList del 1: Typer

```
type DomainList a -- `a` er en "phantom type": brukes kun på venstre siden av `=`  
    = DomainList (List String)
```

```
type alias Sorted a =  
    { a  
      | sorted : ()  
    }
```

```
type alias Sanitized a =  
    { a  
      | sanitized : ()  
    }
```

```
type alias SafeForRender a =  
    { a  
      | sorted : ()  
      , sanitized : ()  
    }
```

## DomainList del 2: Implementasjon

```
new : List String -> DomainList {}
new list =
    DomainList list

sort : DomainList a -> DomainList (Sorted a)
sort (DomainList list) =
    DomainList (List.sort list)

sanitize : DomainList a -> DomainList (Sanitized a)
sanitize (DomainList list) =
    DomainList (list |> List.filter (\entry -> entry /= "bad word"))

render : DomainList (SafeForRender a) -> Html msg
render (DomainList list) =
    Html.div []
        [ Html.h1 []
          [ Html.text "The following is oh-so-safe, both sorted and Sanitized. The Compiler guarantees it!" ]
          , Html.ul []
            (list |> List.map (\entry -> Html.li [] [ Html.text entry ]))
          ]
```

# Compile-time garanti 1: listen må være sortert

```
-- TYPE MISMATCH ----- /Users/cekrem/code/talks/elm/extensible-records/src/Main.elm
```

This function cannot handle the argument sent through the (`|>`) pipe:

```
9|      DomainList.new [ "one word", "bad word", "other word" ]
10|      -- let's skip sorting |> DomainList.sort
11|      |> DomainList.sanitize
12|      |> DomainList.render
      ^^^^^^^^^^^^^^^^^^^^^^^
```

The argument is:

```
DomainList.DomainList (DomainList.Sanitized {})
```

But (`|>`) is piping it to a function that expects:

```
DomainList.DomainList { a | sorted : (), sanitized : () }
```

## Compile-time garanti 2: listen må være sanitized

```
-- TYPE MISMATCH ----- /Users/cekrem/code/talks/elm/extensible-records/src/Main.elm
```

This function cannot handle the argument sent through the (`|>`) pipe:

```
9|      DomainList.new [ "one word", "bad word", "other word" ]
10|      |> DomainList.sort
11|      -- let's skip sanitizing: |> DomainList.sanitize
12|      |> DomainList.render
      ^^^^^^^^^^^^^^^^^^^^^^^
```

The argument is:

```
DomainList.DomainList (DomainList.Sorted {})
```

But (`|>`) is piping it to a function that expects:

```
DomainList.DomainList { a | sorted : (), sanitized : () }
```

**That's all.** 😊