

Internet Applications Design and Implementation

(Lecture 9 - React/Redux meets IFML)

**MIEI - Integrated Master in Computer Science and Informatics
Specialization block**

João Costa Seco (joao.seco@fct.unl.pt)

Jácome Cunha (jacome@fct.unl.pt)

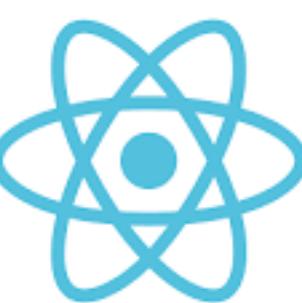
João Leitão (jc.leitao@fct.unl.pt)

Internet Applications Design and Implementation

(Lecture 9 - Part 1 - State Management.
Flux&Redux: Design Patterns to the Rescue)

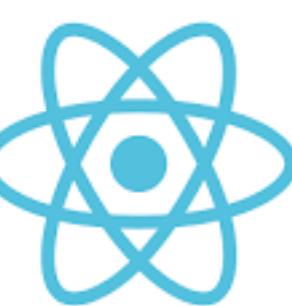
**MIEI - Integrated Master in Computer Science and Informatics
Specialization block**

João Costa Seco (joao.seco@fct.unl.pt)
Jácome Cunha (jacome@fct.unl.pt)
João Leitão (jc.leitao@fct.unl.pt)



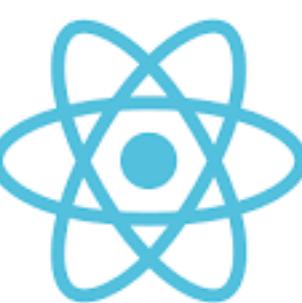
React

- React implements a presentation layer
- React Pros:
 - Highly structured and stable applications
 - Highly Composable and reusable UI components
 - Loose coupling of components (through binding and render)
 - Great match with specification languages like IFML.
- React Cons:
 - There is not a great way to connect to a back-end.
 - Poor scalability with relation to the size of state and number of actions needed (coupling rises).



(Recall) Identify the UI minimal complete state

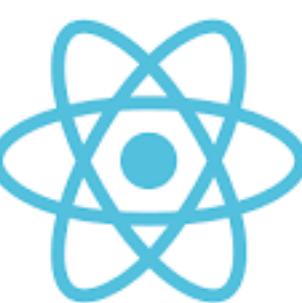
- Is it passed in from a parent via props? If so, it probably isn't state.
- Does it remain unchanged over time? If so, it probably isn't state.
- Can you compute it based on any other state or props in your component? If so, it isn't state.



(Recall) Find the state component by Q&A

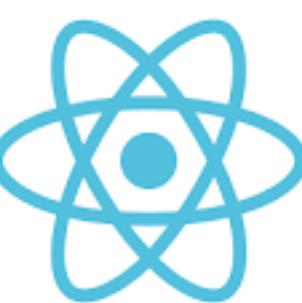
For each piece of state in your application:

- Identify every component that renders something based on that state.
- Find a common owner component (a single component above all the components that need the state in the hierarchy).
- Either the common owner or another component higher up in the hierarchy should own the state.
- If you can't find a component where it makes sense to own the state, create a new component simply for holding the state and add it somewhere in the hierarchy above the common owner component.



Problem?

- The main model entities for a React application will be placed on the state of the application.
 - Application class will be cluttered with Layout, State and State changing and sync code (controllers?)
 - The whole application will be (re)rendered on every state change (efficiency problem?)
 - Solution 1: functions passed through props.
 - Solution 2: controller object(s) with groups of functions will be passed through props.
 - These are “indirect pointers” in the ownership hierarchy!
 - Although React creates new objects and clones state well, the code coupling does not scale well, needs a lot of lifecycle management code.



Problem?

- Problem 1: Prop drilling

Properties that need to be accessed in many different places (views) in the application tend to be passed through props in “all” components. This makes reuse a lot harder...

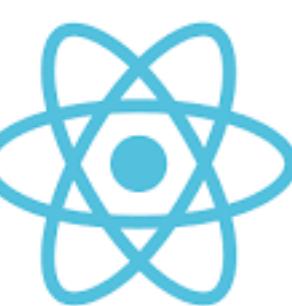
Solution 1: Use children components to structure your application

Solution 2: Use render props to compute the components in advance

Solution 3: Use React.Context to store and distribute global information

- Problem 2: Remote connections and Callback hell

When state is kept at the top level (or near), loading of state and state updates are encoded in many callbacks. This makes reuse a lot harder. One must resort to more sophisticated patterns/libraries/frameworks.



Example

```
const App = () => {
  const username = "mary"
  const roles = ["admin", "editor"]
  return <div>
    <Header username={username}/>
    <Container roles={roles}/>
  </div>
}

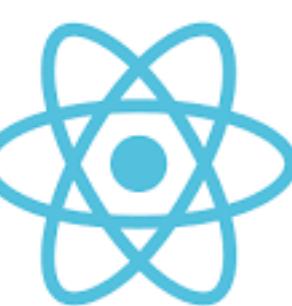
/* This header cannot be reused in other places without a username and userbox!! */

const Header =
  ({ username }: { username: string }) =>
  <div style={{ border: "1px solid" }}>This is the header <UserBox username={username} /></div>

const UserBox = ({ username }: { username: string }) =>
  <div> <p>The username is {username}</p> </div>

/* This container cannot be reused in other places without an array of roles and the children components below!! */

const Container = ({roles}:{roles:string[]}) =>
  <div>
    <div>
      {roles.includes("admin") && <p> This is a ADMIN part of the container </p>}
      {roles.includes("editor") && <p>This is an Editor part of the container </p>}
    </div>
  </div>
.
```



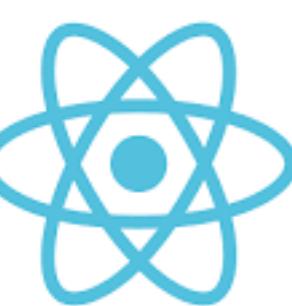
Component Composition/Inclusion

```
const App = () => {
  const username = "mary"
  const roles = ["admin", "editor"]
  return <div>
    <Header>
      <UserBox username="username"/>
    </Header>
    <Container>
      <div>
        {roles.includes("admin") && <p> This is a ADMIN part of the container </p>}
        {roles.includes("editor") && <p>This is an Editor part of the container </p>}
      </div>
    </Container>
  </div>
}

const Header : React.FunctionComponent<{}> =
  (props) => <div style={{ border: "1px solid" }}>This is the header { props.children }</div>

const UserBox = ({username}:{username:string}) =>
  <div> <p>The username is {username}</p> </div>

const Container: React.FunctionComponent<{}> = (props) =>
  <div>
    { props.children }
  </div>
```



Render props

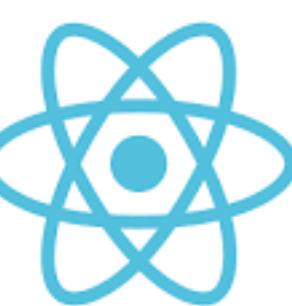
```
const App = () => {
  const username = "mary"
  const roles = ["admin", "editor"]

  const userBox = () => <div> <p>The username is {username}</p> </div>

  const content = () => <div>
    {roles.includes("admin") && <p> This is a ADMIN part of the container </p>}
    {roles.includes("editor") && <p>This is an Editor part of the container </p>}
  </div>

  return <PageLayout userBox = {userBox} content={content}/>
}

const PageLayout = ({userBox, content}:any) =>
<div>
  <div style={{ border: "1px solid" }}>This is the header {userBox()} </div>
  <div> {content()} </div>
</div>
```



React.Context - A side channel

```
interface ContextInterface { username: string | null; roles:string[] }

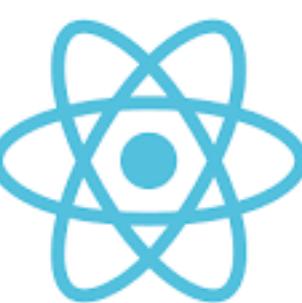
const UserContext = React.createContext<ContextInterface>({username:null, roles:[]})

const App = () =>
  <UserContext.Provider value={({username:"mary", roles:["admin","editor"]})}>
    <div>
      <Header/>
      <Container/>
    </div>
  </UserContext.Provider>

const Header = () => <div style={{border: "1px solid"}}>This is the header <UserBox/></div>

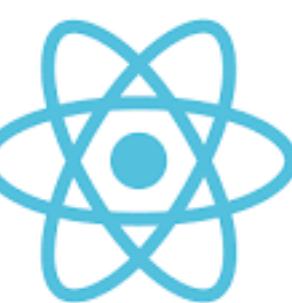
const UserBox = () =>
  <UserContext.Consumer>
    {value => <div><p>The username is {value.username}</p></div> }
  </UserContext.Consumer>

const Container = () =>
  <UserContext.Consumer>
    { value => <div>
        { value.roles.includes("admin") && <p> This is a ADMIN part of the container </p> }
        { value.roles.includes("editor") && <p>This is an Editor part of the container </p> }
      </div>
    }
  </UserContext.Consumer>
```



Sketch for remote connections (with jquery :-))

```
componentDidMount() {
    $.getJSON(this.TASKS_URL)
        .then(doneFilter: (data:Task[])) =>
            this.setState(
                f: (state:StateSig) =>
                    this.setState( state: {...state, tasks:data})))
    $.getJSON(this.SPRINTS_URL)
        .then(doneFilter: (data:Sprint[])) =>
            this.setState(
                f: (state:StateSig) =>
                    this.setState( state: {...state, sprints:data})))
    $.getJSON(this.USERS_URL)
        .then(doneFilter: (data:User[])) =>
            this.setState(
                f: (state:StateSig) =>
                    this.setState( state: {...state, users:data})))
}
```

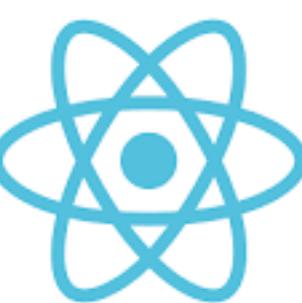


Sketch

```
addSprint = (sprint:Sprint) =>
  $.post(this.SPRINTS_URL, JSON.stringify(sprint))
    .then( doneFilter: ()=> this.setState(f: (state:any)=>({...state, sprints:[sprint,...state.sprints]})))
```



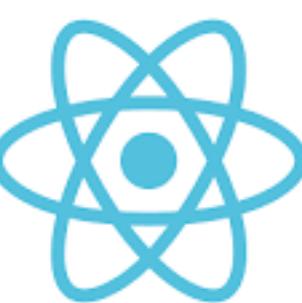
```
addTask = (task:Task) =>
  $.post(this.SPRINTS_URL, JSON.stringify(task))
    .then( doneFilter: ()=>(this.setState(f: (state:any)=>({...state, tasks:[task,...state.tasks]}))))
```



Solution: complete a (kind of) client-side MVC

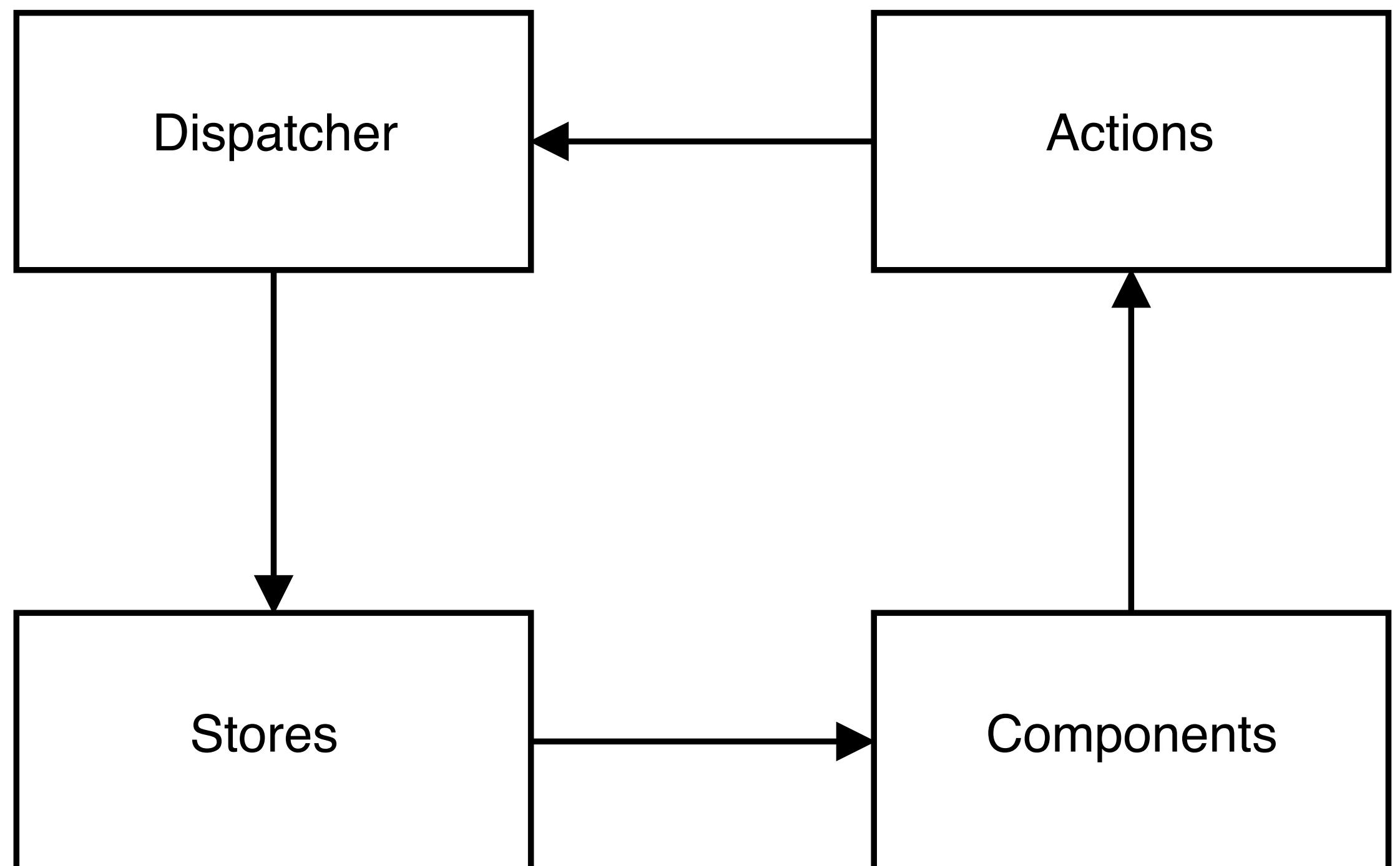
- Flux pattern extends the React with a store and dispatcher components.
- Implements a uni-directional flow of data
(controller does not intermediate the views)
It's not really MVC or MVP.
- React components (views) bind to stored elements.
- Actions (store functions) are triggered by the view
- Actions get dispatched to have effect on a store





Flux pattern

- diagram



A Store is an EventEmitter (“Subject”)

```
import { EventEmitter } from "events"
import {Task} from "../Tasks";

class TasksStore extends EventEmitter {
  tasks:Task[]

  constructor() {
    super()
    this.tasks = [
      {id: 1, name:"Go shopping", dueDate:"2017-10-23"},
      {id: 2, name:"Do the dishes", dueDate:"2017-10-22"},
      {id: 3, name:"Do homework", dueDate:"2017-10-21"},
    ] // Later will have to be managed from the server
  }

  getAllTasks() {
    return this.tasks;
  }
}

const tasksStore = new TasksStore()

export default tasksStore
```

A Store is an EventEmitter (“Subject”)

```
class TasksStore extends EventEmitter {  
  tasks:Task[]  
  count:number  
  
  constructor() {...}  
  
  createTask(name:string, dueDate:string) {  
    this.tasks.push({id: this.count++, name, dueDate})  
    this.emit(event: "change")  
  }  
  
  getAllTasks() {...}  
}  
  
componentWillMount() {  
  tasksStore.on(event: "change",  
    listener: () => this.setState(  
      f: (state:StateSig) => ({...state, tasks:tasksStore.getAllTasks()}))  
  )  
}
```

App:

Flux - connect with server

```
class TasksStore extends EventEmitter {  
    private TASKS_URL: "/tasks"  
  
    tasks:Task[]  
    count:number  
  
    constructor() {...}  
  
    createTask(task:Task) {  
        this.tasks.push({...task, id: this.count++})  
        $.post(this.TASKS_URL, JSON.stringify(task))  
            .then( doneFilter: ()=>this.emit( event: "change") )  
            .fail( failCallback: () => this.emit( event: "TASK_ERROR") )  
    }  
  
    getAllTasks() {...}  
}
```

Redux

Redux - Learn with the authors...

FREE

Fundamentals of Redux Course from Dan Abramov



Instructor
Dan Abramov

react redux 2h 1m

★ ★ ★ ★ ★ 4.7 9769 people completed

Published 7 years ago | Updated 2 years ago

Bookmark

Download

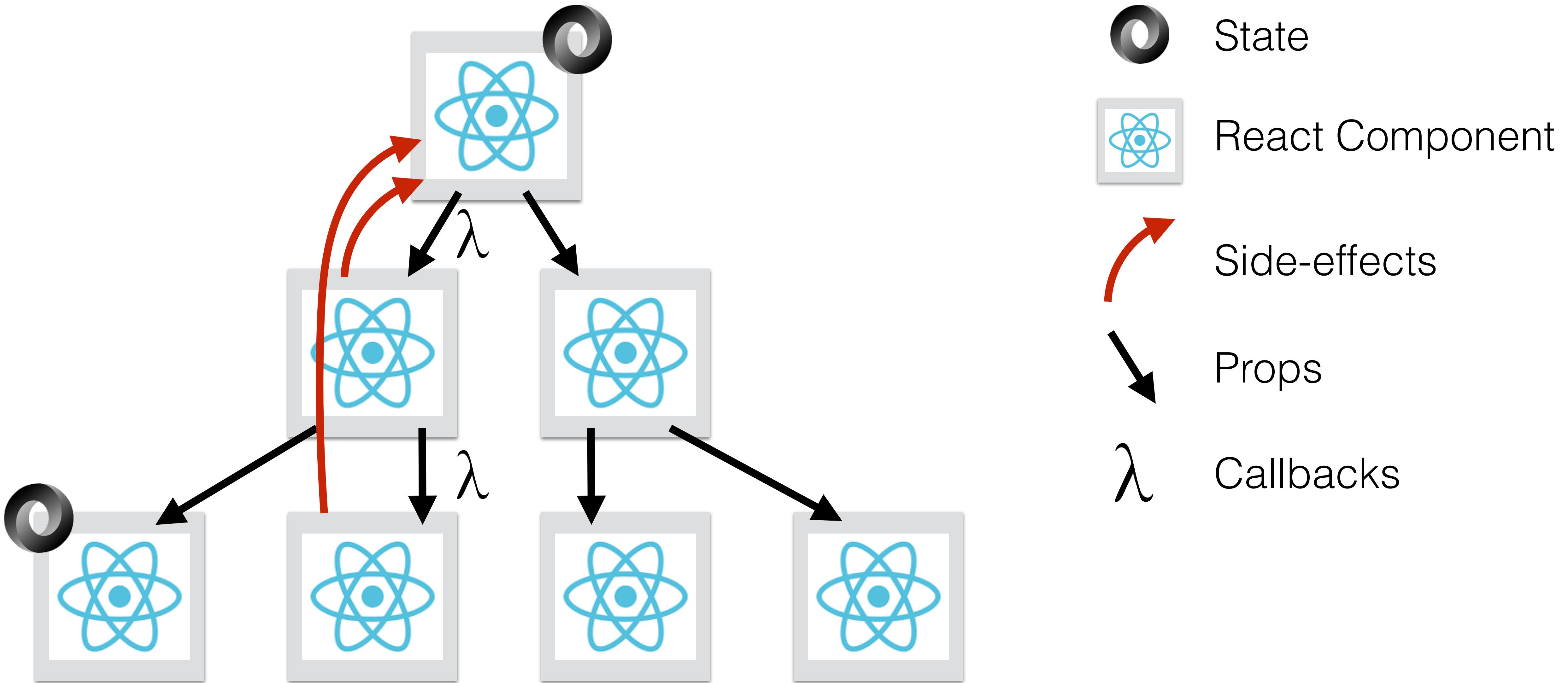
RSS



In this comprehensive tutorial, Dan Abramov – the creator of Redux – will teach you how to manage state in your [React](#) application with [Redux](#).

▶ Start Watching

[State management](#) is absolutely critical in providing users with a well-crafted experience with minimal bugs.





Redux

- Implements a variant of a flux pattern.
- Based on functional state transformations. The state is read-only, can only be changed by action dispatching.
- A store that implements the observer pattern.
- Ingredients are Actions, Store, Reducers.

<https://redux.js.org/introduction/getting-started>

```
import {Action, createStore} from 'redux'

function counter(state = 0, action:Action) {
  switch (action.type) {
    case 'INCREMENT':
      return state + 1;
    case 'DECREMENT':
      return state - 1;
    default:
      return state
  }
}

let store = createStore(counter);

store.subscribe(() => console.log(store.getState()));

store.dispatch({ type: 'INCREMENT'});
store.dispatch({ type: 'INCREMENT'});
store.dispatch({ type: 'DECREMENT'});
```



Redux - reducers

```
store.dispatch({ type: 'INCREMENT'});
```

undefined

1

```
store.dispatch({ type: 'DECREMENT'});
```

2

1

```
store.dispatch({ type: 'INCREMENT'});
```

```
counter(undefined, {type:'INCREMENT'}) = 1  
counter(1, {type:'INCREMENT'}) = 2  
counter(2, {type:'DECREMENT'}) = 1
```



Redux - reducers

- The state can be compositionally built from smaller pieces.
- Reducers can be combined to form a more elaborate state.

```
class ToDo { text:string; completed:boolean;
  constructor(text:string, completed: boolean) {
    this.text = text; this.completed = completed;
  }
}
interface AddTodoAction extends Action { text:string }
interface CompleteTodoAction extends Action { index:number }

function todos(state = [] as ToDo[], action:Action): ToDo[] {
  switch (action.type) {
    case 'ADD_TODO':
      let a1 = action as AddTodoAction;
      return [...state, new ToDo(a1.text, false)]
    case 'COMPLETE_TODO':
      let a2 = action as CompleteTodoAction;
      return state.map((todo:ToDo, index) => {
        return index == a2.index ? {...todo, completed:true}: todo
      });
    default:
      return state
  }
}
```



Redux - reducers

- The state can be compositionally built from smaller pieces.
- Reducers can be combined to form a more elaborate state.

```
interface SetFilterAction extends Action { filter: string }

function filter(state = 'SHOW_ALL', action:Action): string {
  switch (action.type) {
    case 'SET_FILTER':
      return (action as SetFilterAction).filter;
    default:
      return state;
  }
}

const reducer = combineReducers({ todos, filter });
const todoStore = createStore(reducer);

todoStore.subscribe(() => console.log(todoStore.getState()));

todoStore.dispatch({type:'ADD_TODO', text:'Complete the project'});
todoStore.dispatch({type:'ADD_TODO', text:'Start the project'});
todoStore.dispatch({type:'ADD_TODO', text:'Submit the project'});
todoStore.dispatch({type:'COMPLETE_TODO', index:0});
todoStore.dispatch({type:'COMPLETE_TODO', index:1});
todoStore.dispatch({type:'COMPLETE_TODO', index:2});
```



Redux - reducers

- The state can be compositionally built from smaller pieces.
- Reducers can be combined to form a more elaborate state.

```
[ ToDo { text: 'Complete the project', completed: false } ]
```

```
[ ToDo { text: 'Complete the project', completed: false },
  ToDo { text: 'Start the project', completed: false } ]
```

```
[ ToDo { text: 'Complete the project', completed: false },
  ToDo { text: 'Start the project', completed: false },
  ToDo { text: 'Submit the project', completed: false } ]
```

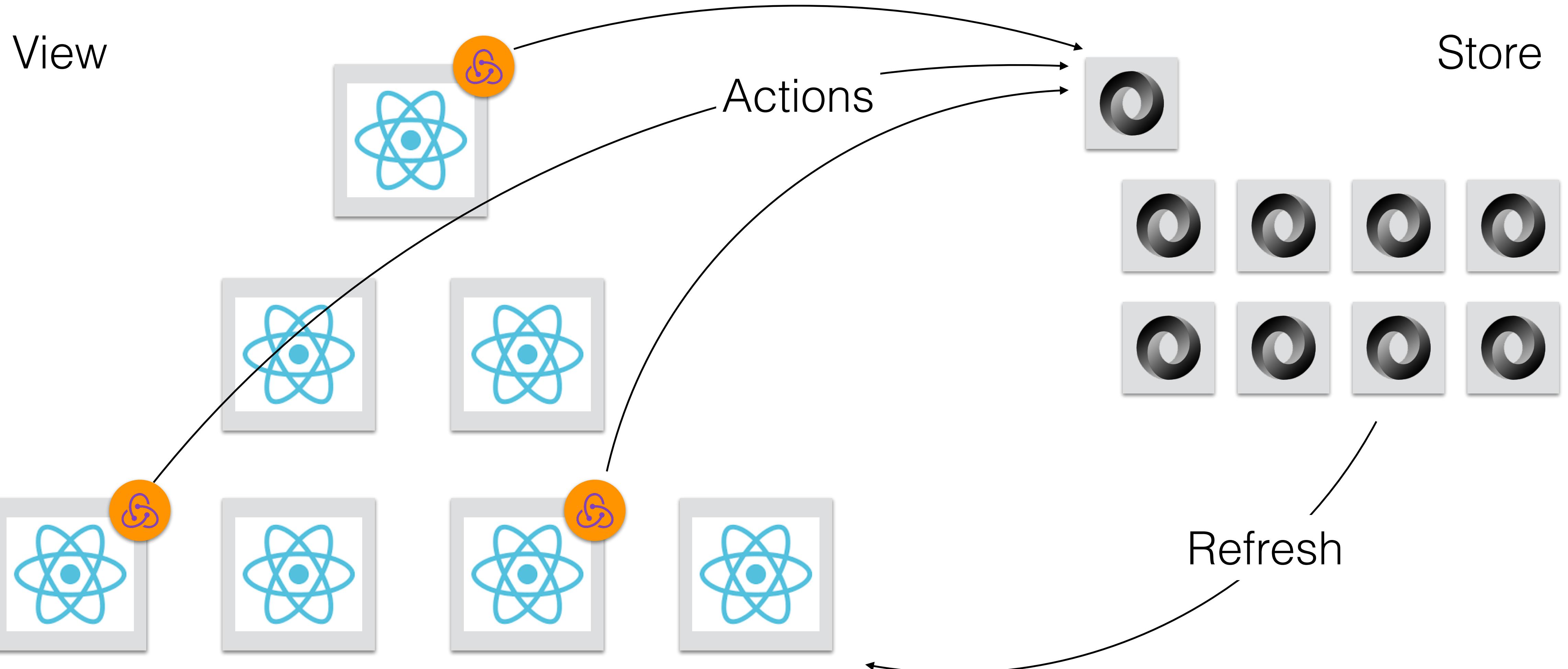
```
[ { text: 'Complete the project', completed: true },
  ToDo { text: 'Start the project', completed: false },
  ToDo { text: 'Submit the project', completed: false } ]
```

```
[ { text: 'Complete the project', completed: true },
  { text: 'Start the project', completed: true },
  ToDo { text: 'Submit the project', completed: false } ]
```

```
[ { text: 'Complete the project', completed: true },
  { text: 'Start the project', completed: true },
  { text: 'Submit the project', completed: true } ]
```



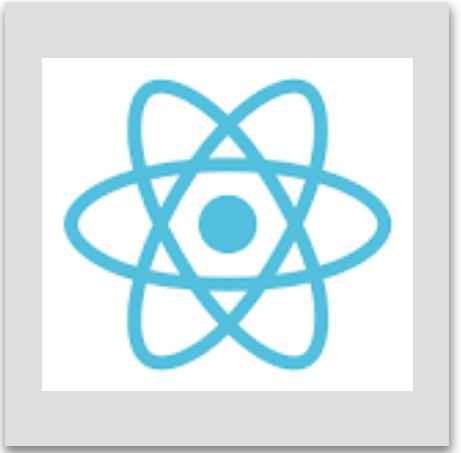
Redux-react





Presentational Components

- React and redux can be combined by having a presentational component, extended with connections to the state generated automatically.



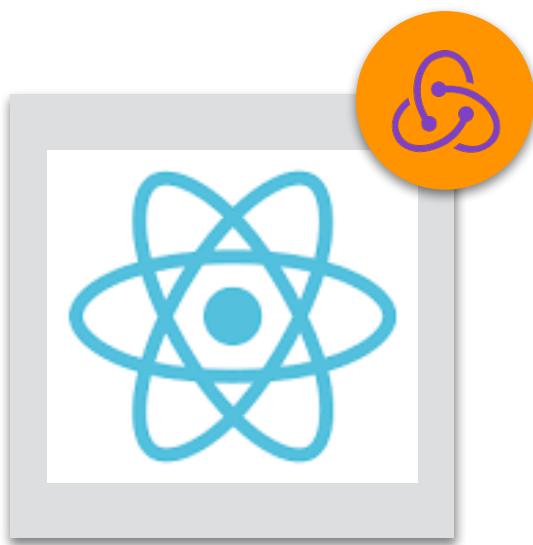
```
interface ToDoItemInterface {text:string, completed:boolean, onClick:() => void}
const ToDoItem = ({text, completed, onClick}:ToDoItemInterface) =>
  <li
    onClick={onClick}
    style={{textDecoration: completed ? 'line-through': 'none'}}
  >
    {text}
  </li>;

interface ToDoListInterface { todos: ToDo[], onToDoClick: (index:number) => void}
const ToDoList = ({todos, onToDoClick}:ToDoListInterface) =>
  <ul>
    { todos.map( (todo, index) => {
      console.log(index);
      return <ToDoItem key={index} {...todo} onClick={() => onToDoClick(index)} />
    })}
  </ul>;
```

Presentational Components



- React and redux can be combined by having a presentational component, extended with connections to the state generated automatically.



```
const mapStateToProps =  
  (state:ToDoAppStateInterface) => ({todos: filterTodos(state.todos, state.filter)});  
  
const completeToDo = (index:number) => ({ type:'COMPLETE_TODO', index});  
const mapDispatchToProps = (dispatch:(action:Action)=>void) =>  
  ({ onToDoClick: (id:number) => { dispatch(completeToDo(id)) }});  
  
const VisibleToDoList = connect(mapStateToProps, mapDispatchToProps)(ToDoList);  
  
const ToDoListWithStore = () =>  
  <Provider store={todoStore}>  
    <VisibleToDoList/>  
  </Provider>;
```

Async actions

- All changes to the state are done via actions
- Integration with asynchronous code

```
class Pet { name:string; constructor(name:string) { this.name = name; }}
```

```
interface PetState { pets: Pet[] }
```

```
const ADD_PET = 'ADD_PET';
const REQUEST_PETS = 'REQUEST_PETS';
const RECEIVE_PETS = 'RECEIVE_PETS';
```

```
const addPet = (name:string) => ({type: ADD_PET, name:string});
```

```
interface AddPetAction extends Action { name:string }
```

```
function petsReducer(pets = [] as Pet[], action:Action):Pet[] {
  switch (action.type) {
    case ADD_PET:
      let addPet = action as AddPetAction;
      return [...pets, new Pet(addPet.name)];
    default:
      return pets
  }
}
```



Async actions

- All changes
- Integration v

```
const requestPets = () => ({type: REQUEST_PETS});  
const receivePets = (data:Pet[]) => ({type: RECEIVE_PETS, data: data});  
  
function fetchPets() {  
    return (dispatch) => {  
        dispatch(requestPets());  
        return fetch("/pets")  
            .then( response => response.json())  
            .then( data => dispatch(receivePets(data)))  
    }  
}  
  
const loggerMiddleware = createLogger();  
  
const store =  
    createStore( combineReducers({pets: petsReducer}),  
                applyMiddleware( thunkMiddleware, loggerMiddleware ) );  
store.dispatch(fetchPets()).then(() => console.log(store.getState()));
```



Using Redux Toolkit (Slices)

```
import { createSlice, configureStore } from '@reduxjs/toolkit'

const counterSlice = createSlice({
  name: 'counter',
  initialState: {
    value: 0
  },
  reducers: {
    incremented: state => {
      state.value += 1
    },
    decremented: state => {
      state.value -= 1
    }
  }
})

export const { incremented, decremented } = counterSlice.actions

const store = configureStore({ reducer: counterSlice.reducer })

// Can still subscribe to the store
store.subscribe(() => console.log(store.getState()))

// Still pass action objects to `dispatch`, but they're created for us
store.dispatch(incremented()) // {value: 1}
store.dispatch(incremented()) // {value: 2}
store.dispatch(decremented()) // {value: 1}
```

<https://redux.js.org/introduction/getting-started>



Using React Redux (Hooks)

```
import React from 'react'
import { useSelector, useDispatch } from 'react-redux'
import { decrement, increment } from './counterSlice'

export function Counter() {
  const count = useSelector((state) => state.value)
  const dispatch = useDispatch()

  return (
    <div>
      <button
        className={styles.button}
        aria-label="Increment value"
        onClick={() => dispatch(increment())}>
        +
      </button>
      <span className={styles.value}>{count}</span>
      <button
        className={styles.button}
        aria-label="Decrement value"
        onClick={() => dispatch(decrement())}>
        -
      </button>
    </div>
  )
}
```



Further readings

- <https://redux.js.org/introduction/getting-started>
- <https://react-redux.js.org/>
- <https://egghead.io/courses/getting-started-with-redux>
- <https://react-redux.js.org/next/api/hooks>
- <https://egghead.io/lessons/react-redux-the-single-immutable-state-tree>
- <https://redux.js.org/advanced/async-actions>

Internet Applications Design and Implementation

(Lecture 9 - Part 2 - From IFML to React)

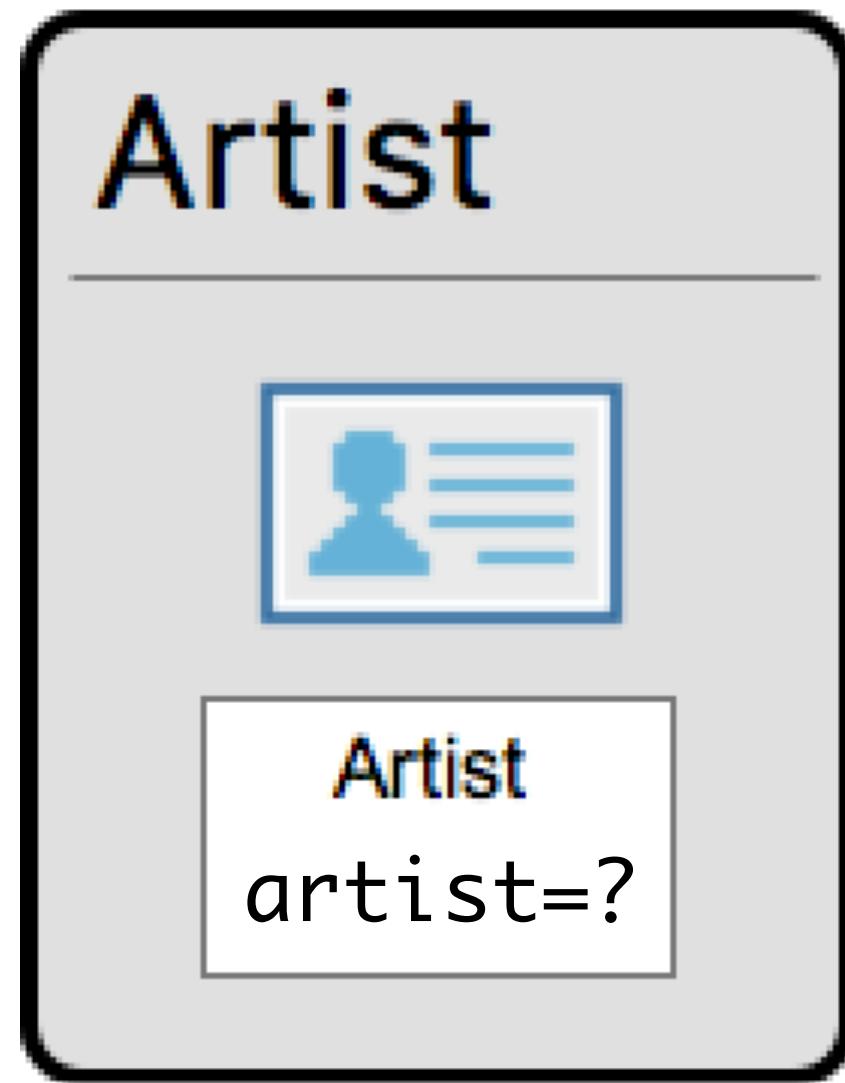
**MIEI - Integrated Master in Computer Science and Informatics
Specialization block**

João Costa Seco (joao.seco@fct.unl.pt)

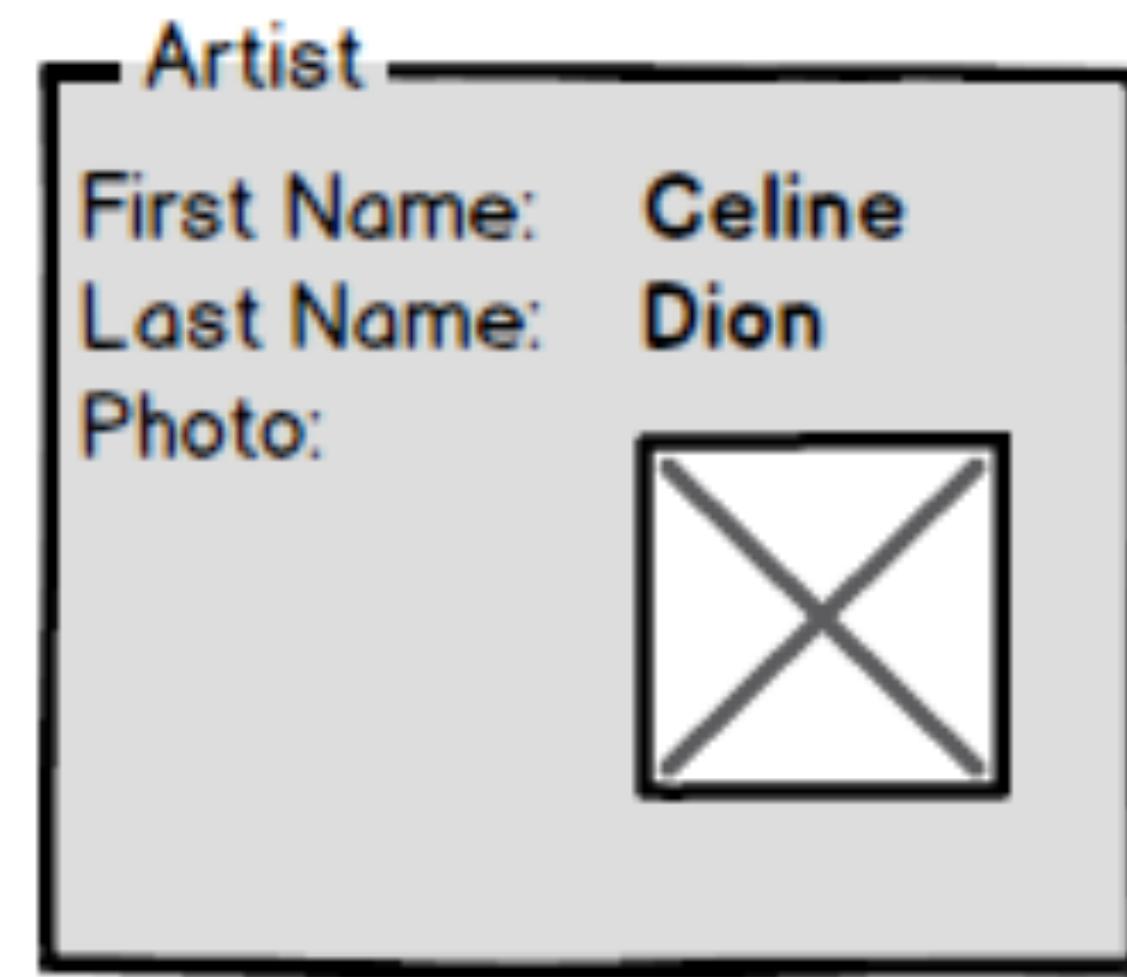
Jácome Cunha (jacome@fct.unl.pt)

João Leitão (jc.leitao@fct.unl.pt)

IFML Syntax – View Components: Details

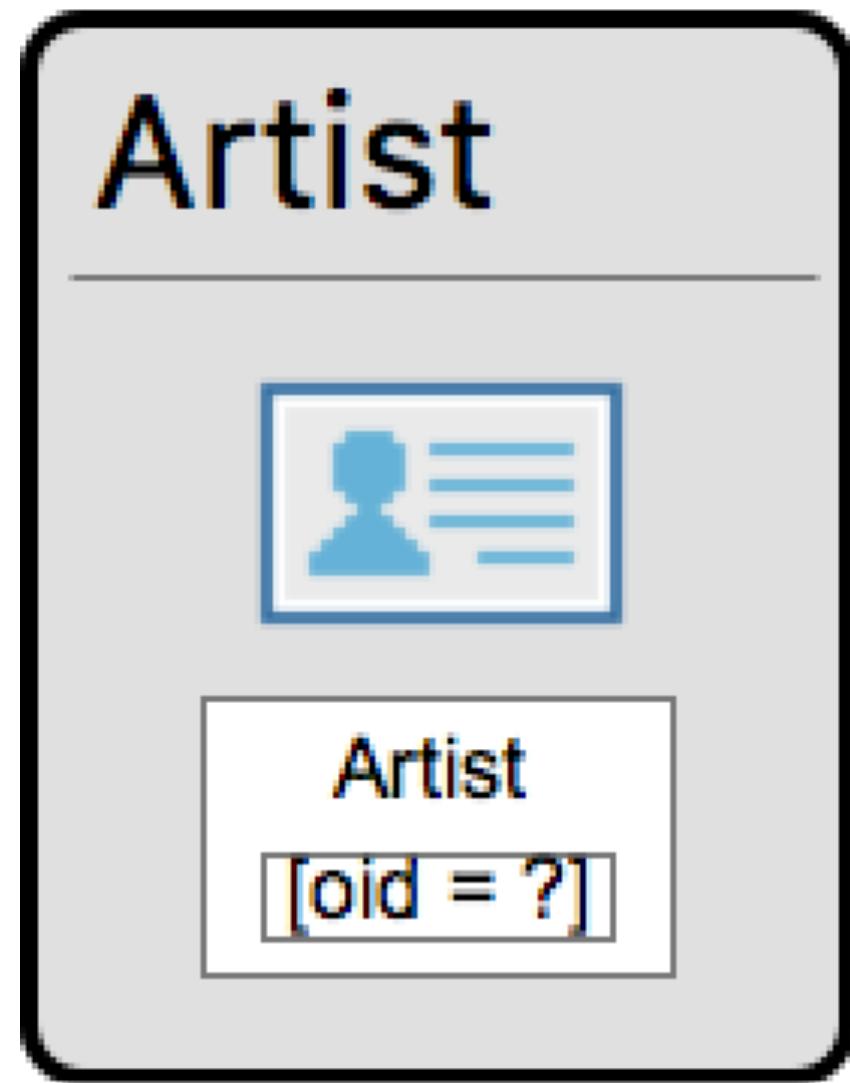


```
const ArtistDetails = (props:{artist:Artist}) =>
  <div>
    <p>First Name: {props.artist.firstName}</p>
    <p>Last Name: {props.artist.lastName}</p>
    <img src={props.artist.photo} />
  </div>
```



- User defines:
 - Type/Entity/Interface (Artist in this case)
 - Shown attributes (e.g. id, name, photo, etc.)
 - Value to be shown (“`artist = ?`”)
- Also available *Multiple Details* and *Scroller*

IFML Syntax – View Components: Details



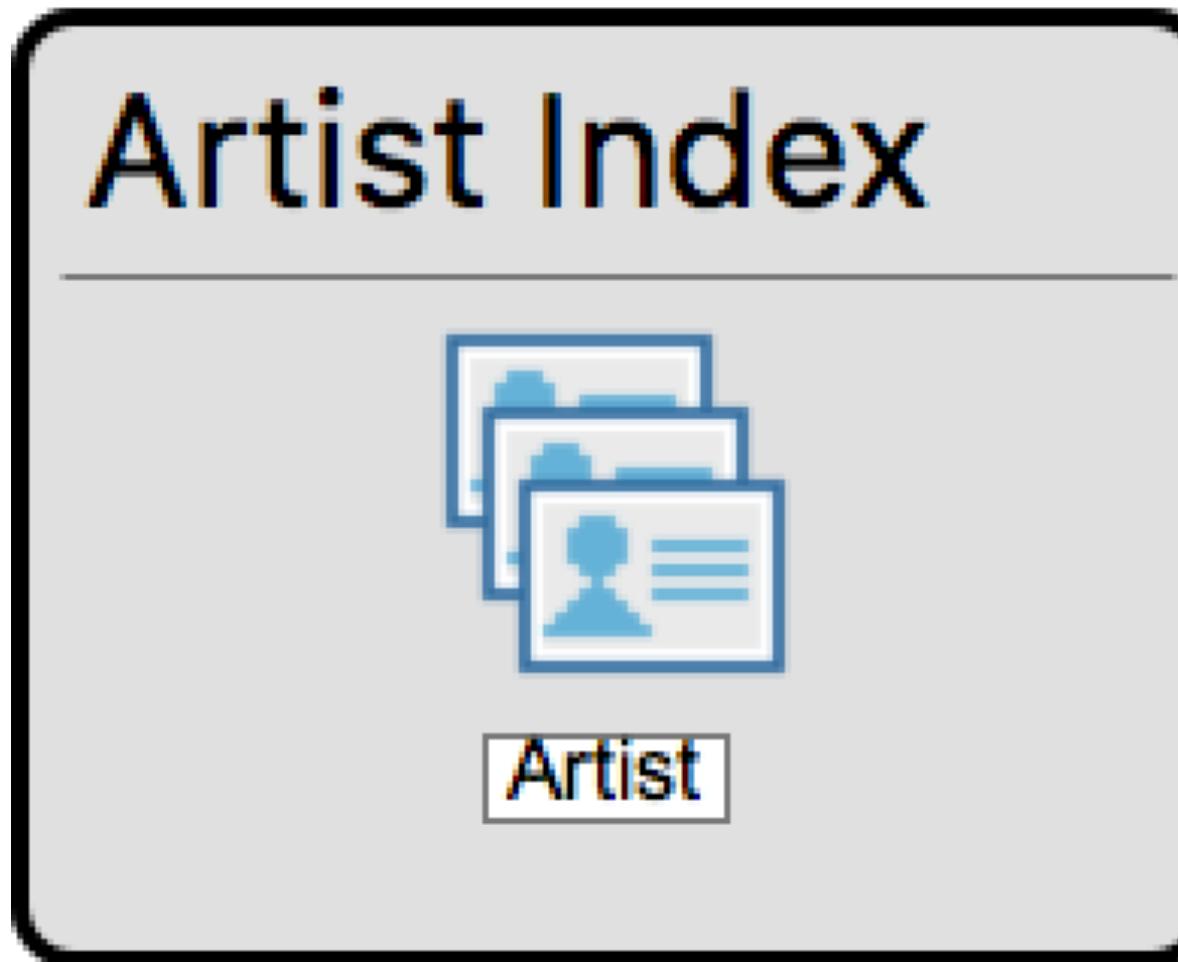
```
const ArtistDetails = (props:{artistId:number}) => {
  const [ artist, setArtist ] = useState<Artist | null>(null);
  getData<Artist>(`/artists/${props.artistId}`)
    .then( data => data && setArtist(data));

  return <>
    {artist &&
      <div>
        <p>First Name: {artist.firstName}</p>
        <p>Last Name: {artist.lastName}</p>
        <img src={artist.photo}/>
      </div>}
    { !artist && <div>No artist</div>}
  </>;
};
```



- Value to be shown can be indirect
if “oid = ?” is the parameter, then search the entry with the given oid

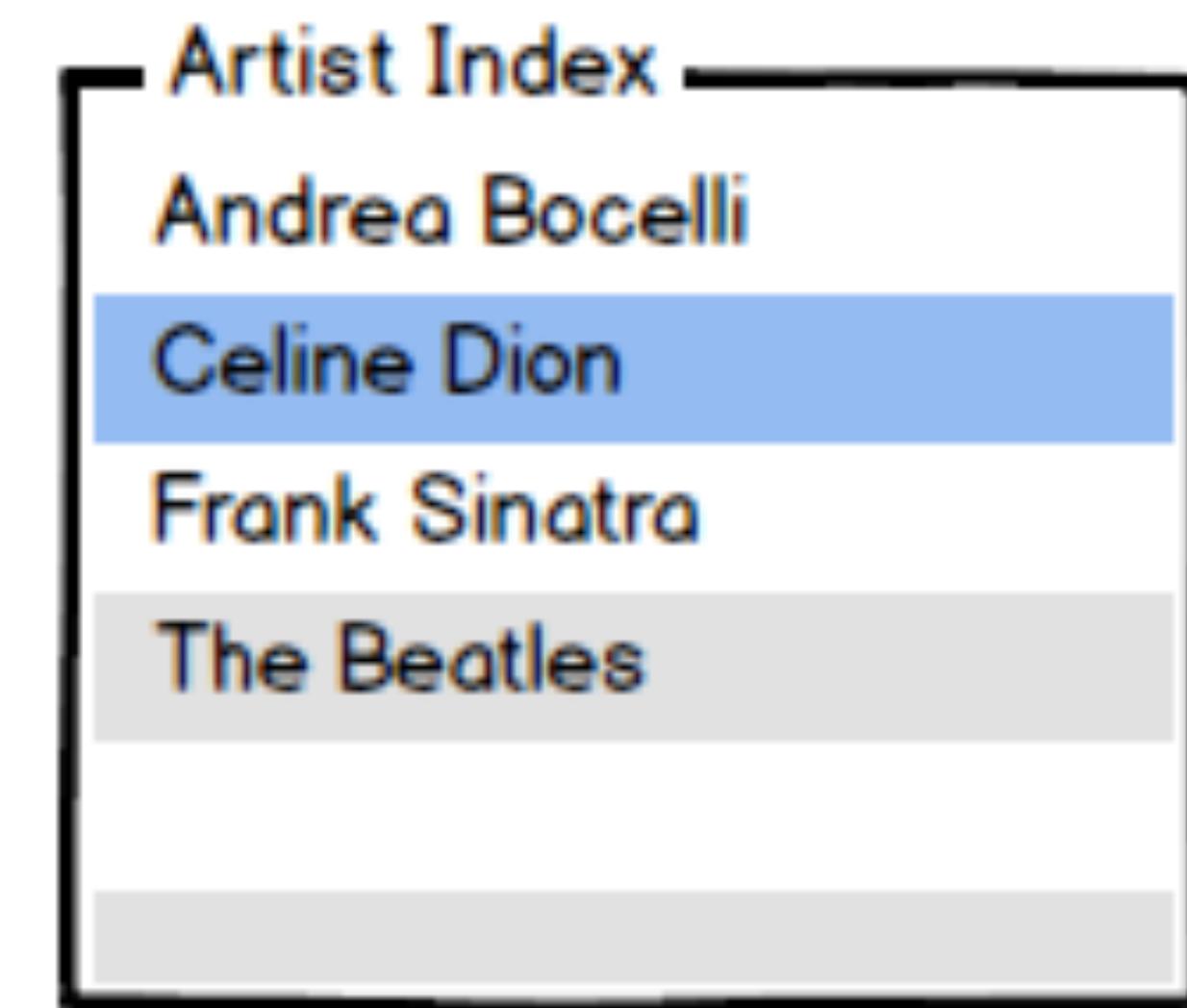
IFML Syntax – View Components: List



```
const ArtistListItem = (props:{artist:Artist}) =>
  <li key={props.artist.id}> {props.artist.firstName} {props.artist.lastName}</li>

const ArtistIndex = (props:{artists:Artist[]}) =>
  <ul>
    { props.artists.map((artist:Artist) => <ArtistListItem artist={artist}/>) }
  </ul>;
```

- User defines:
 - Type/Entity/Interface (Artist in this case)
 - Shown attributes (e.g. id, name, etc.)
 - Selection criteria (... where X = Y)
- Also available *Checkable List*, *Hierarchical List*, etc.



IFML Syntax – View Components: Form

Album Search



Album

```
const AlbumSearch = (props:{loadAlbums:(title:string, year:string) => void}) => {
  const [ searchTitle, setSearchTitle ] = useState("");
  const [ searchYear, setSearchYear ] = useState("");

  useEffect(() => props.loadAlbums(searchTitle, searchYear), [searchTitle, searchYear]);

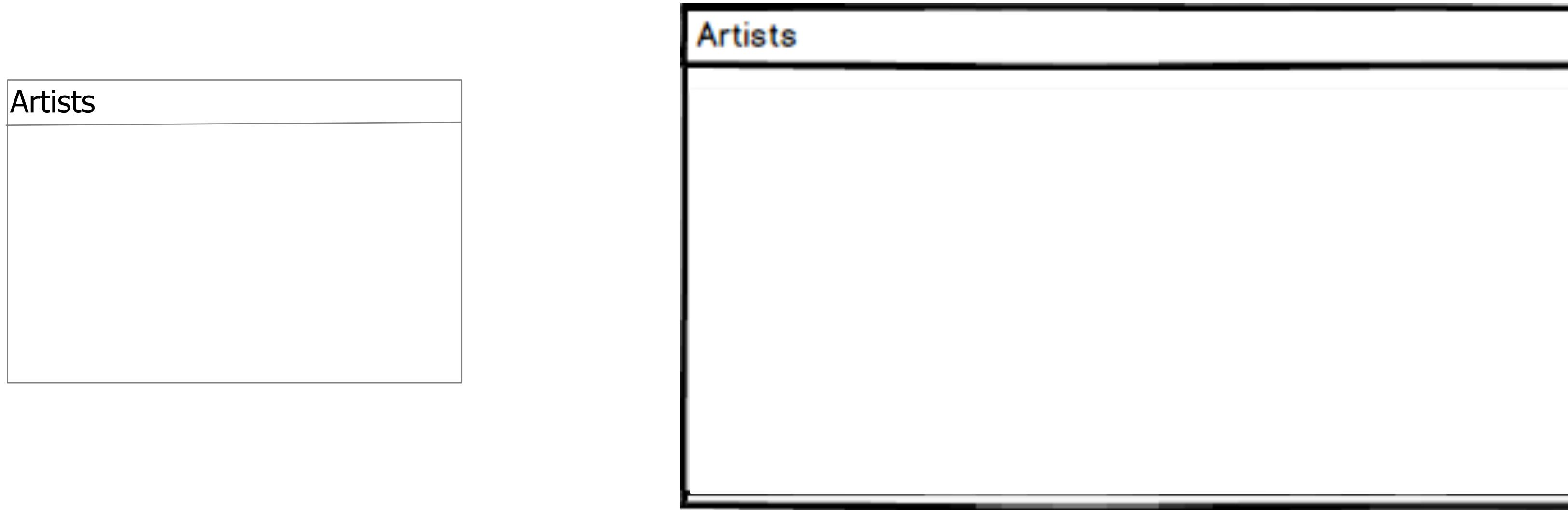
  let handleTitle = (e:ChangeEvent<HTMLInputElement>) => setSearchTitle(e.target.value);
  let handleYear = (e:ChangeEvent<HTMLInputElement>) => setSearchYear(e.target.value);

  return <form>
    <label> Title: <input type="text" onChange={handleTitle}></label>
    <label> Year: <input type="text" onChange={handleYear}></label>
  </form>;
};
```

- User defines:
 - Entity, optionally (Album in this case)
 - Fields (e.g. title, year, etc.)
 - Also available *Multiple Form*

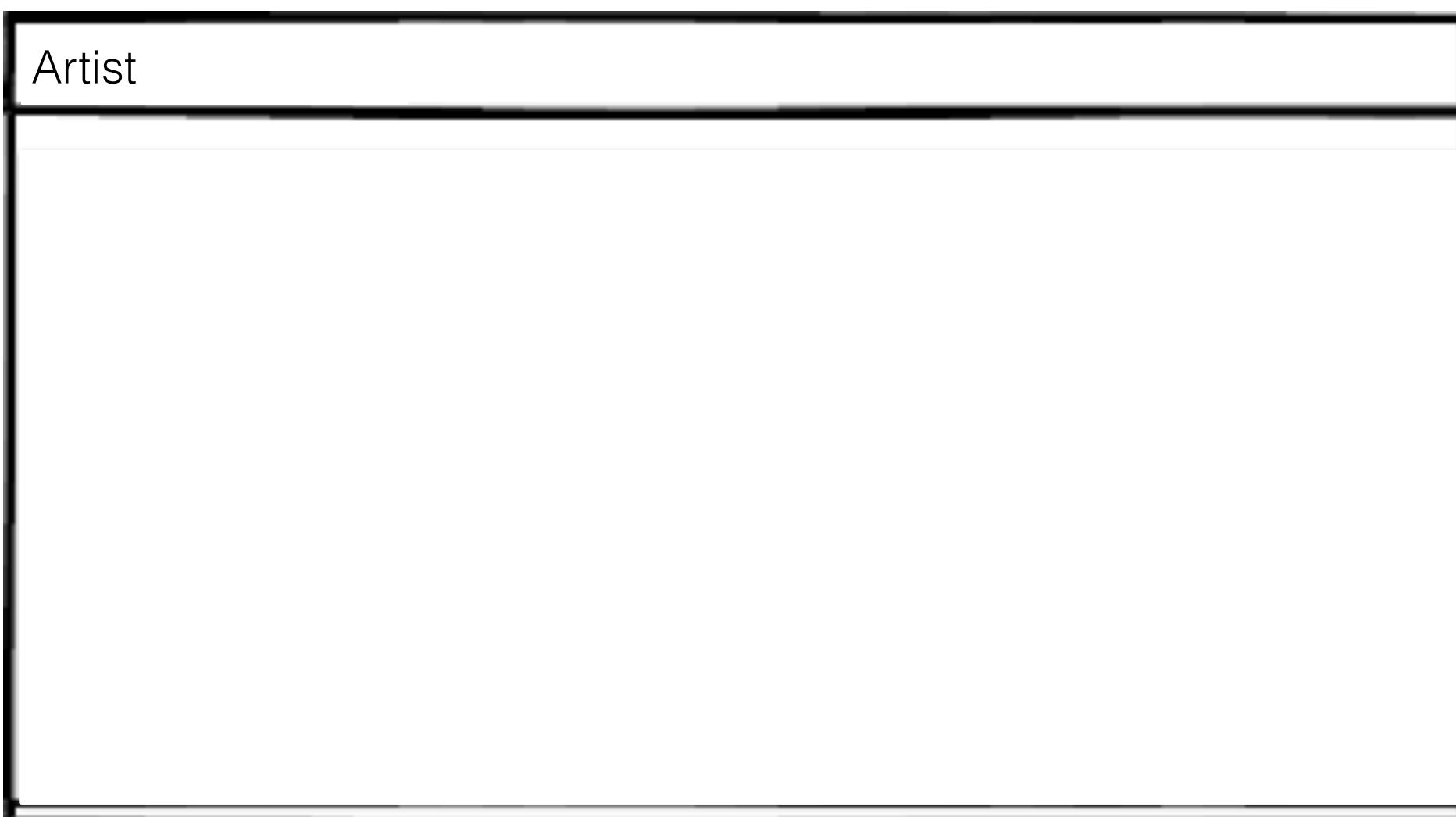
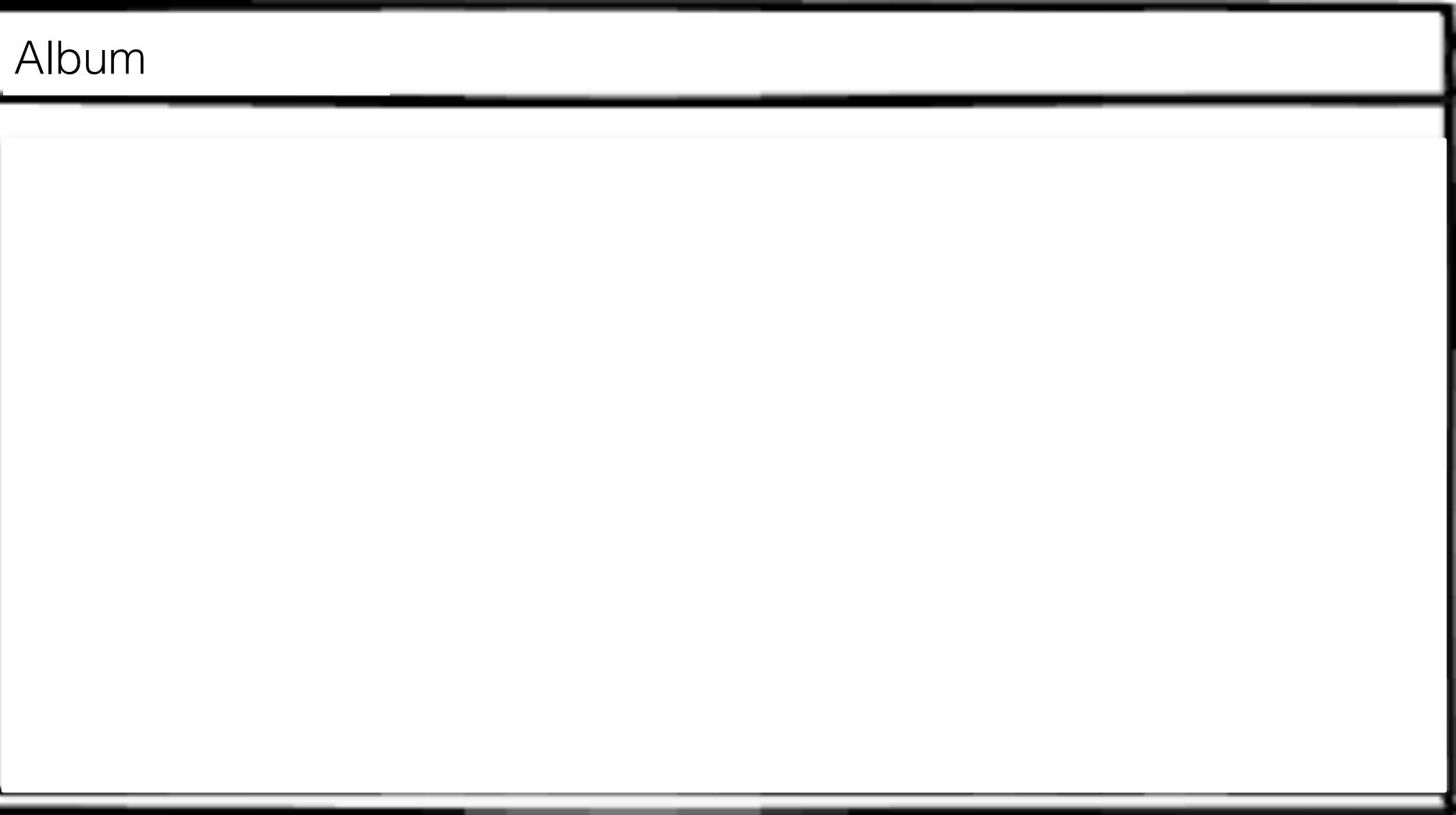
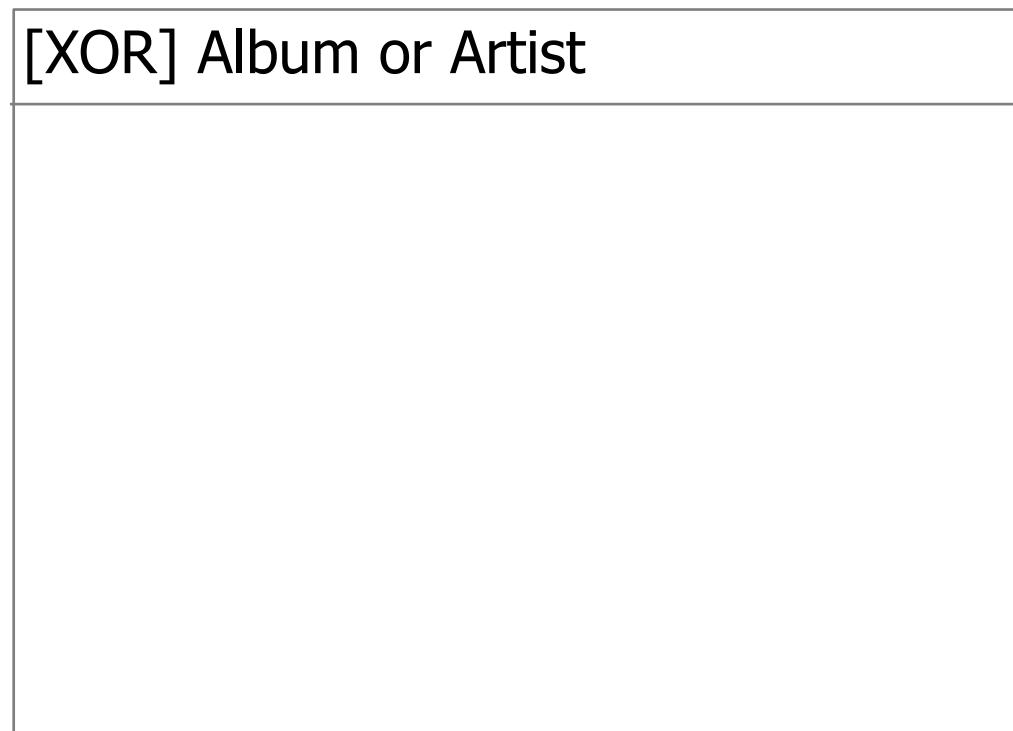
A screenshot of a web application window titled "AlbumSearch". Inside, there is a heading "Album Search" and two form fields. The first field is labeled "Title:" and the second is labeled "Year:". The "Year" field contains the value "1999".

IFML Syntax – Containers: Page



- Containers to hold other constructors (including other pages - nesting)
- Not necessarily mapped into web pages

IFML Syntax – Containers: Alternative Page



- Container to hold other pages in XOR mode
- It either contains Album or Artist, not both at the same time (depends on navigation)

IFML Syntax – (User) Navigation Flow



```
<Link to={`${this.props.id}`}>{"More details about "+this.props.firstName+" "+this.props.lastName}</Link>
```

- User defines:
 - Source (from where to navigate)
 - Target (to where to navigate)
 - Binding parameters/values
- Also available *OK Flow*, and *KO Flow*

IFML Syntax – Data Flow



```
<ArtistDetails artist={artist} />
```

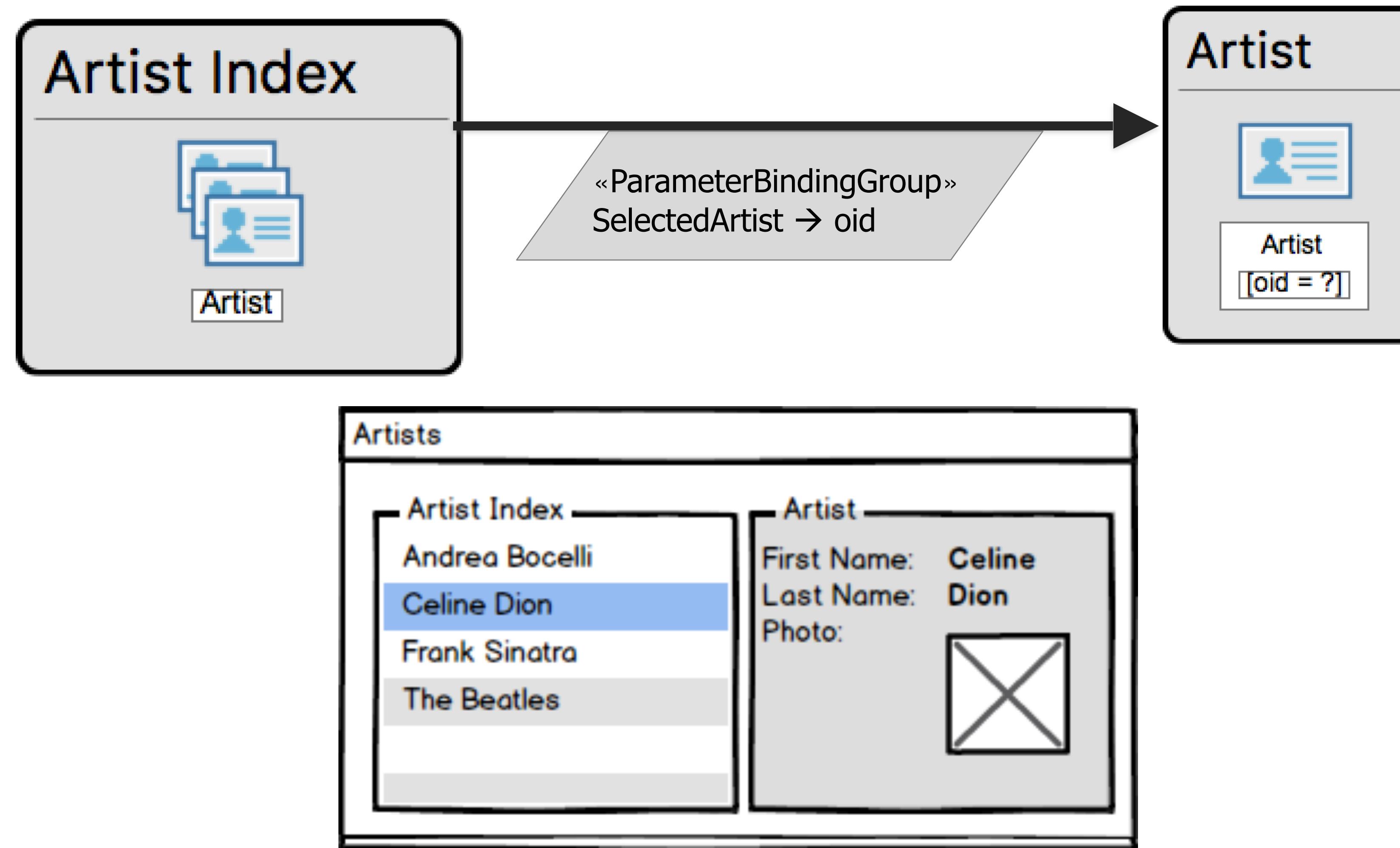
- User defines:
 - Source (from where to navigate)
 - Target (to where to navigate)
 - Binding parameters/values

IFML Syntax – Parameters Binding

```
«ParameterBindingGroup»  
SelectedArtist → AnArtist
```

- User defines:
 - Source parameter/value (SelectedArtist in this case)
 - Target parameter/value (AnArtist in this case)
 - Data flows from source to target
 - Usually associated with flows (user navigation flow or data flow)

IFML Syntax – Parameters Binding



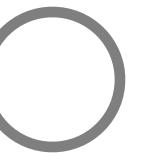
IFML Syntax – Parameters Binding

```
const ArtistIndex = (props:{artists:Artist[], selectArtist:(idx:number)=>void}) =>
  <ul>
    { props.artists.map((artist:Artist) => <ArtistListItem artist={artist}/>) }
  </ul>;
```



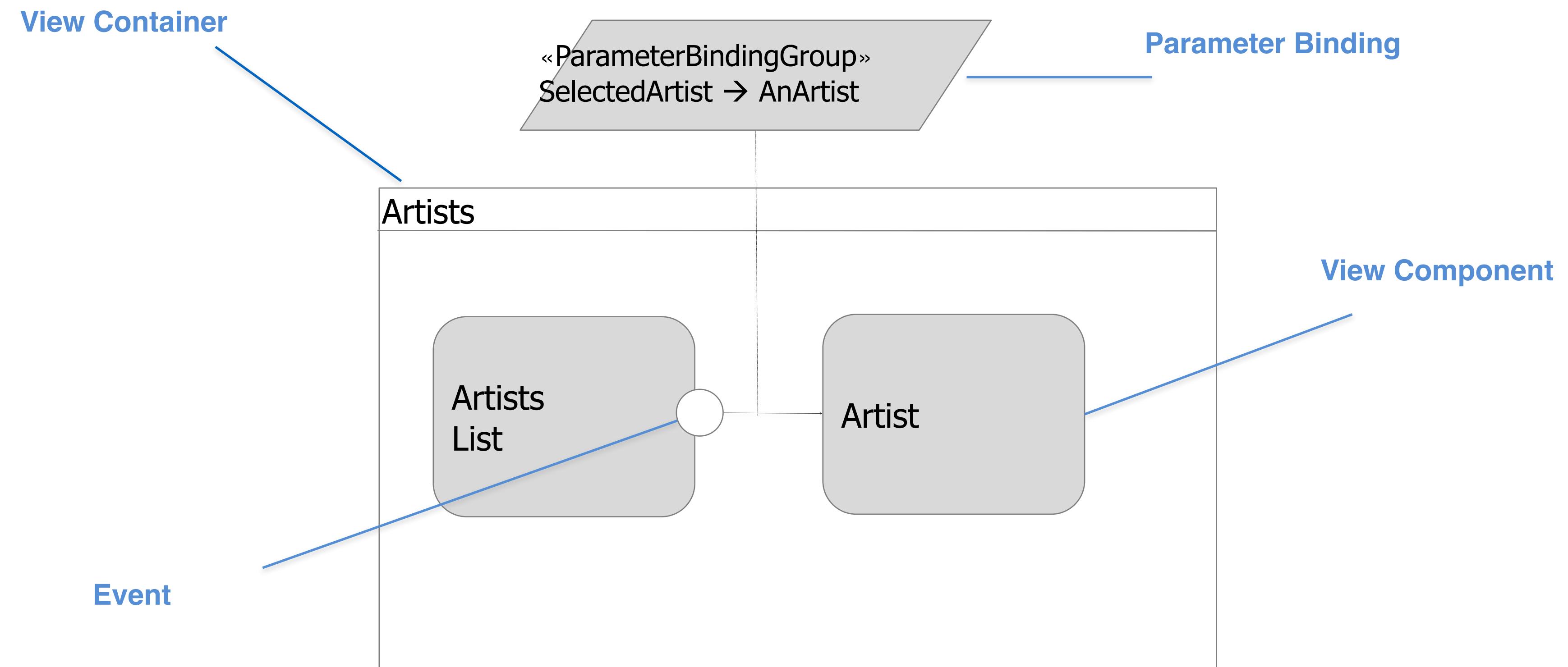
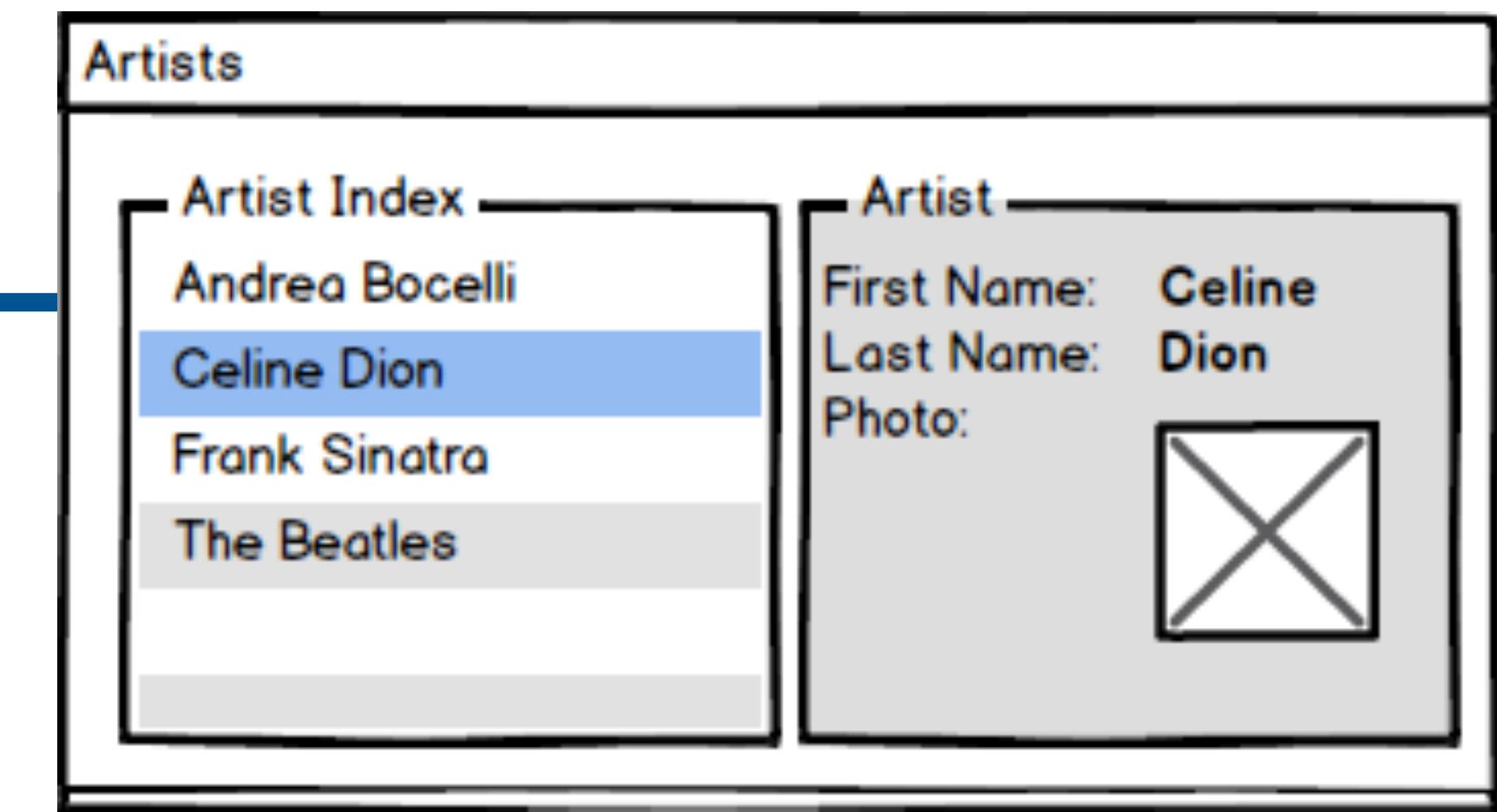
```
const ArtistIndexWithDetails = (props:{artists:Artist[]}) => {
  const [selectedArtist, setSelectedArtist] = useState<number | null>(null);
  return <>
    <ArtistIndex artists={props.artists} selectArtist={setSelectedArtist}>
      { selectedArtist && <ArtistDetails artistId={artists[selectedArtist]}> }
    </>;
};
```

IFML Syntax – Events

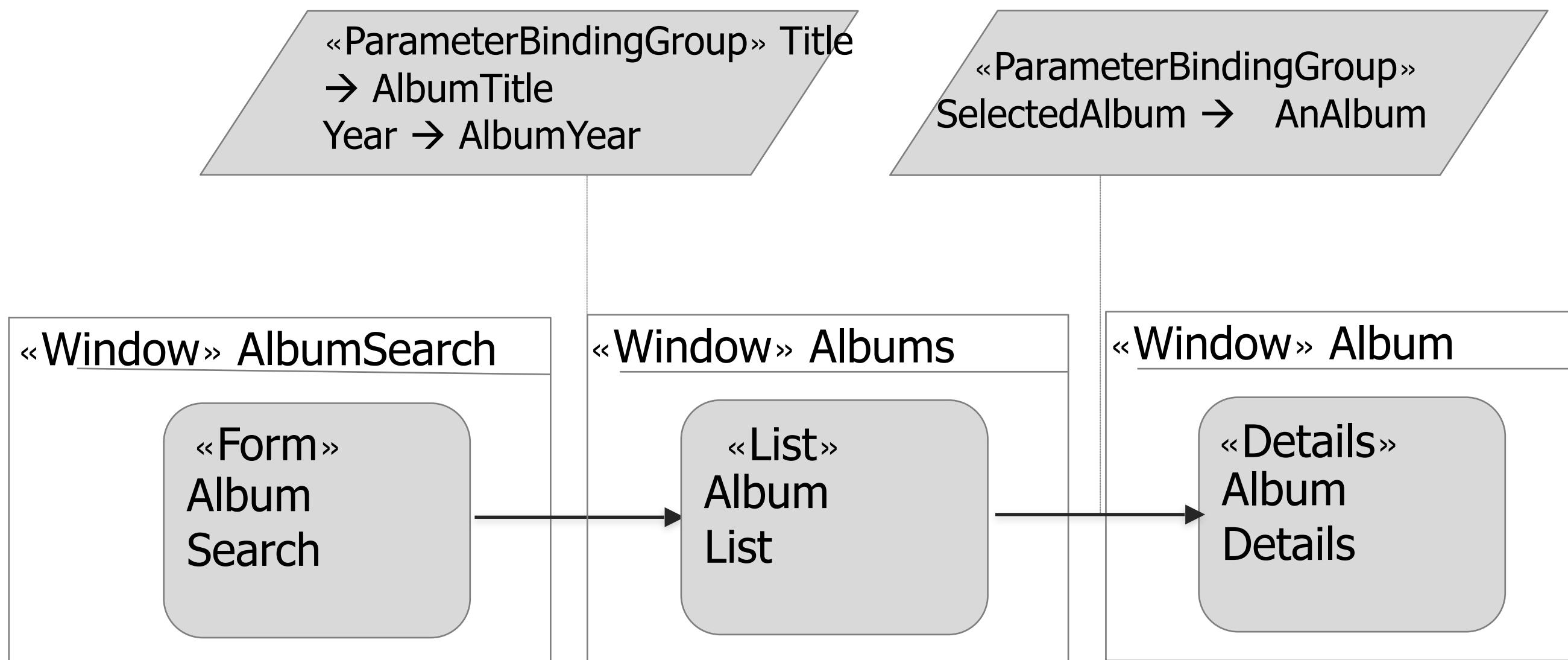
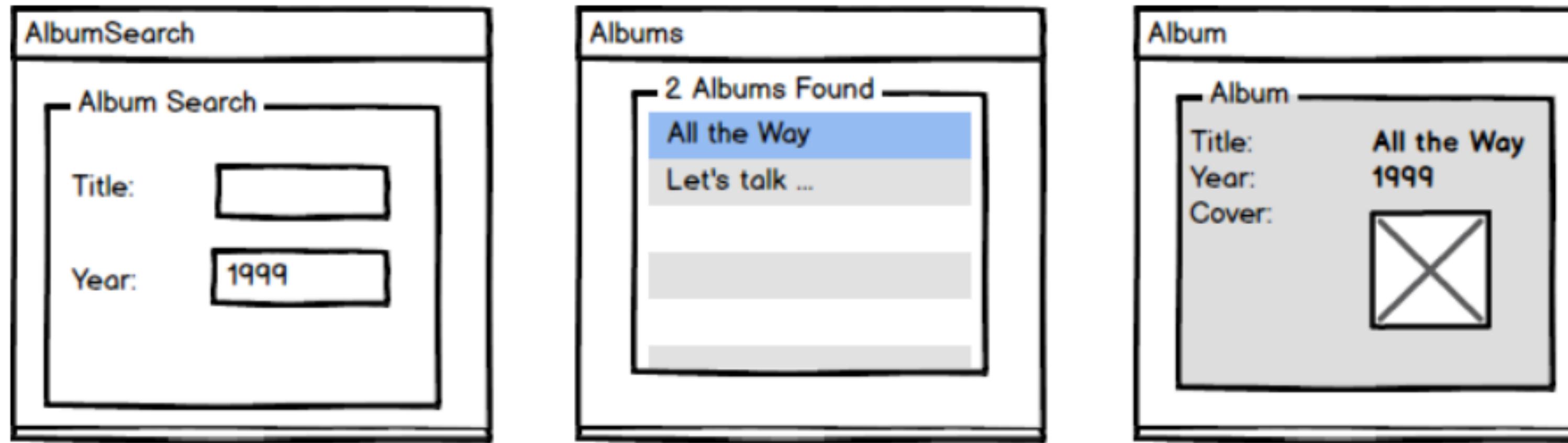


- Created when user clicks on something
 - To select an element from a list
 - To navigate
 - Etc.

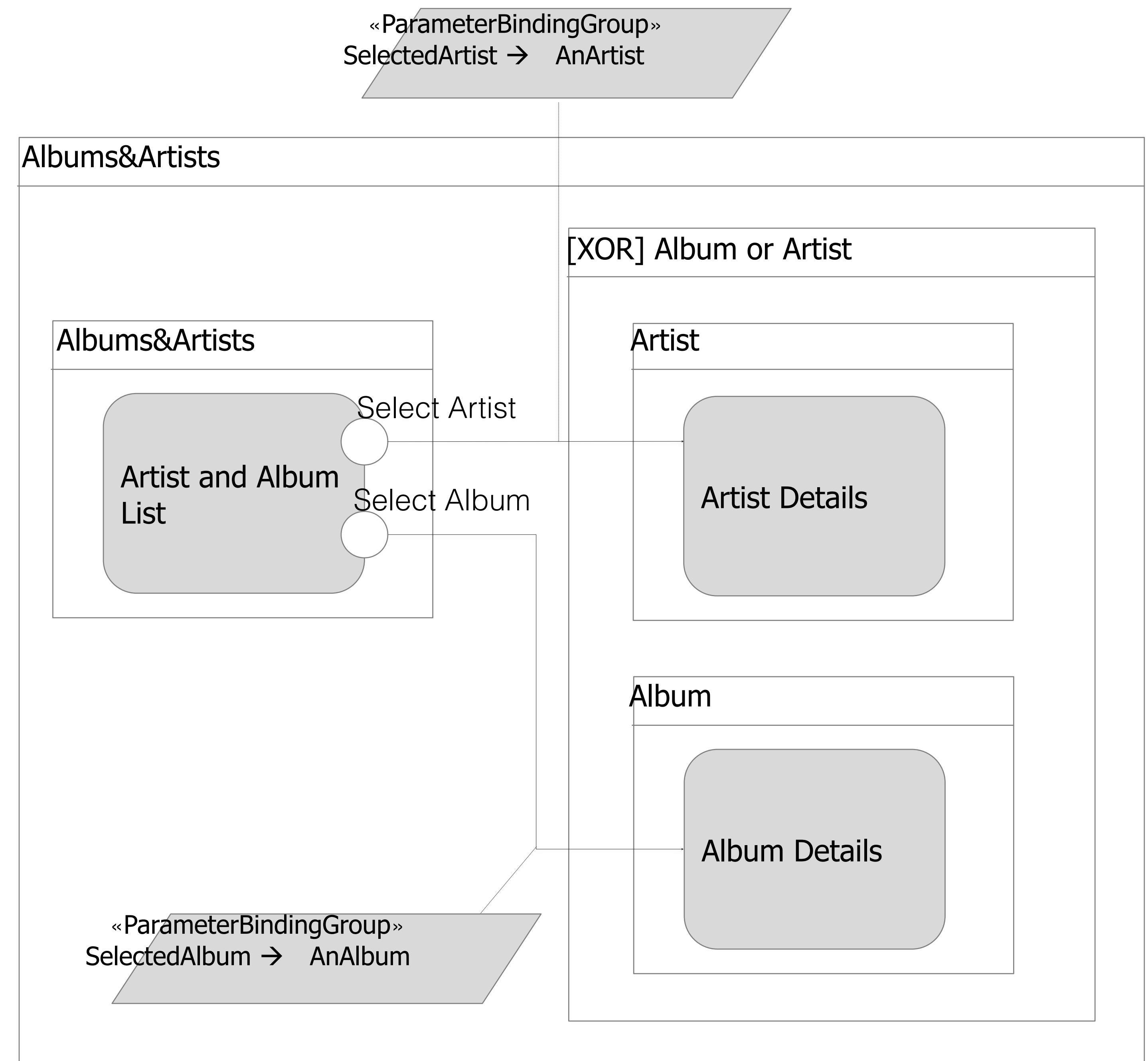
♪ All Together Now, All Together Now ♪



IFML by example



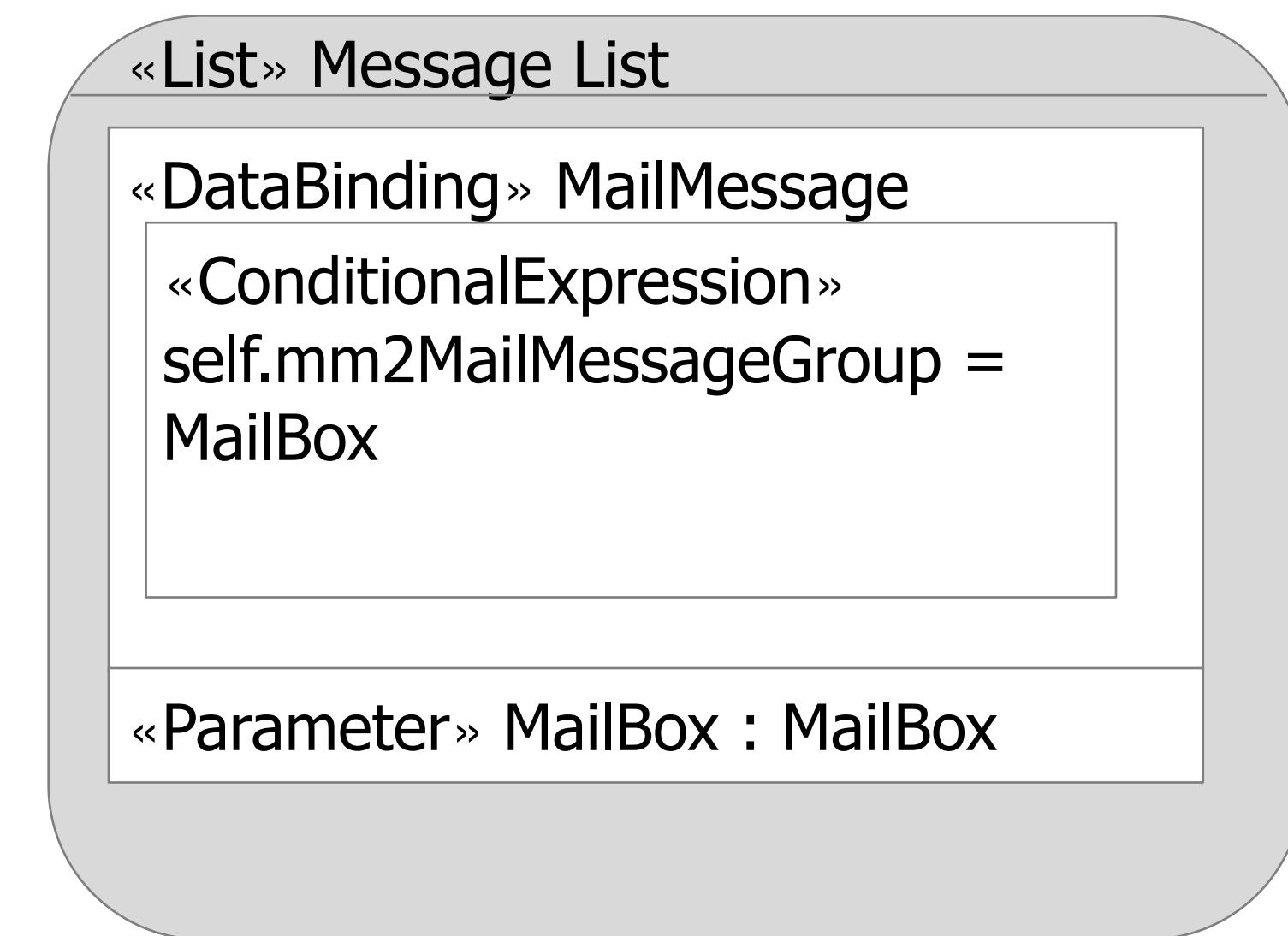
IFML by example



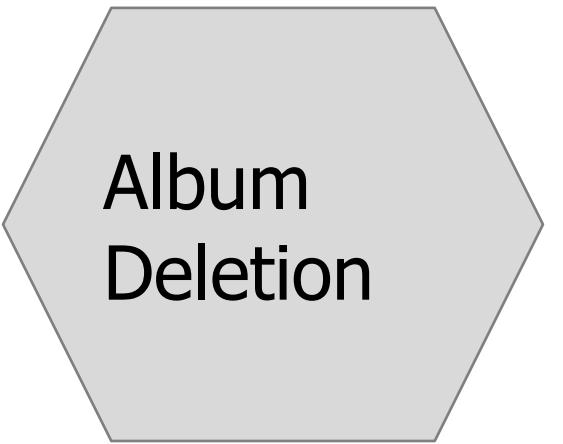
IFML – Adding Details to View Components

View Component Parts:

- Data binding
- Parameters

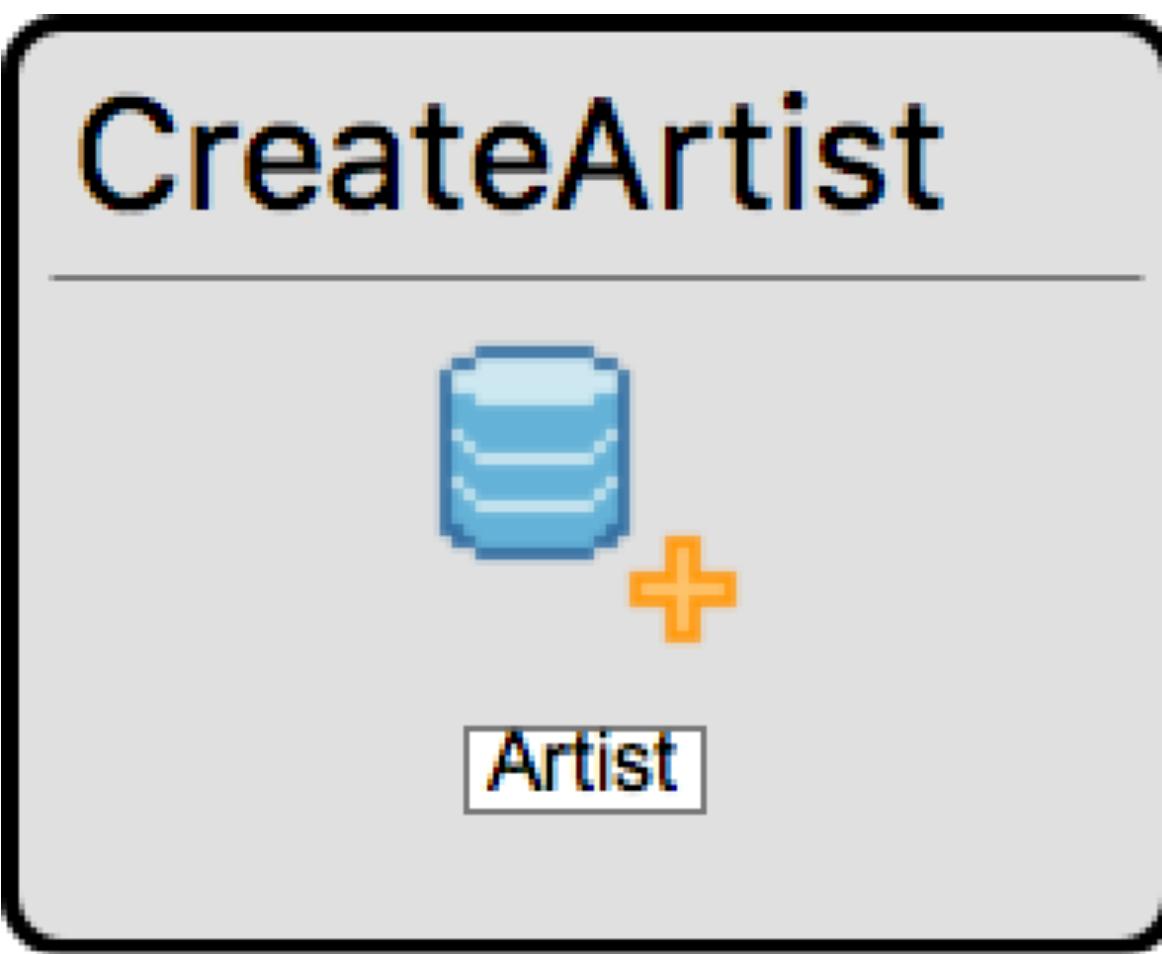


More IFML Syntax – Actions



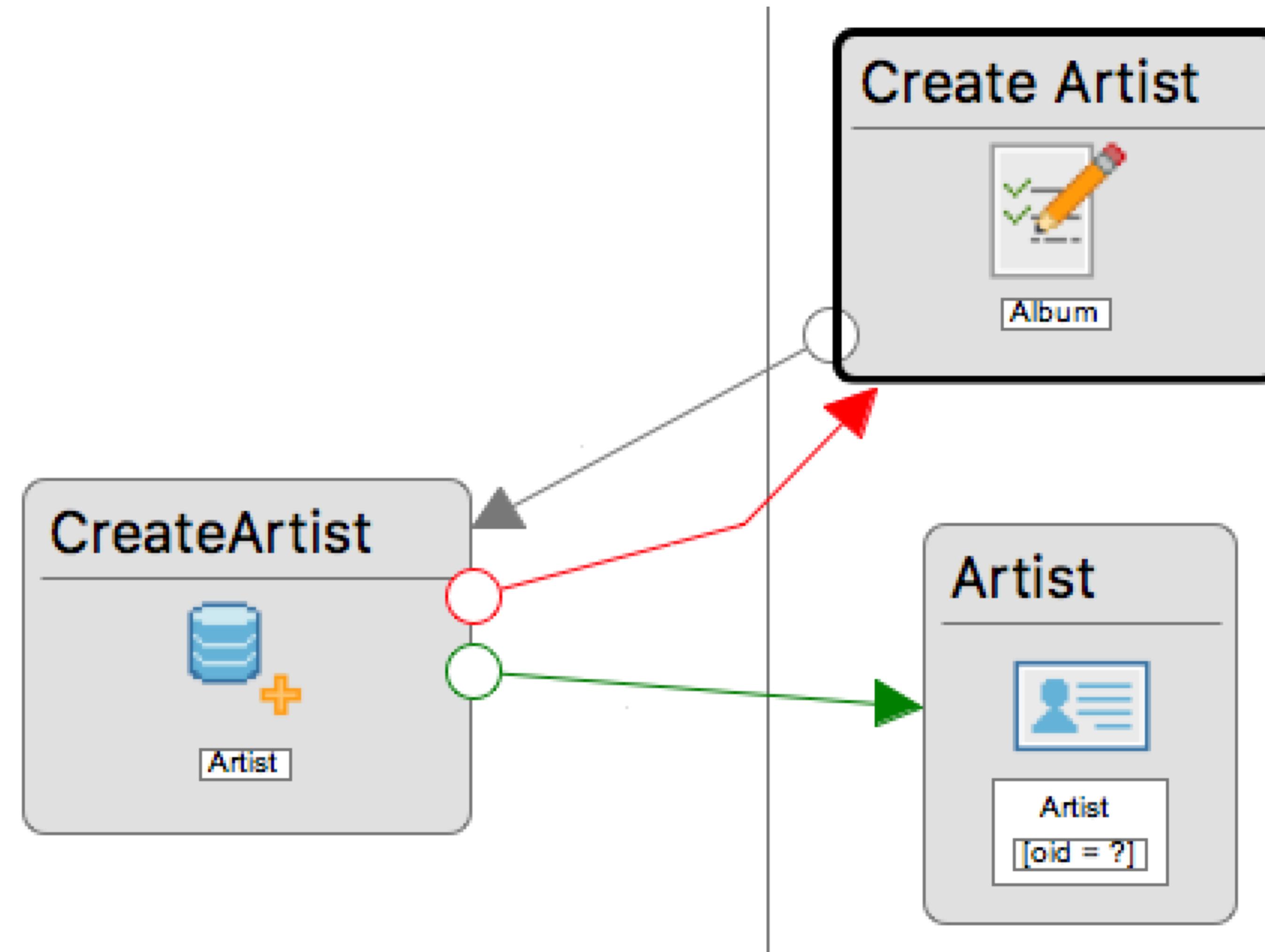
- Generic actions can be defined by the user
- Tool support adds a few very useful (next slide)

More IFML Syntax – Actions: Create

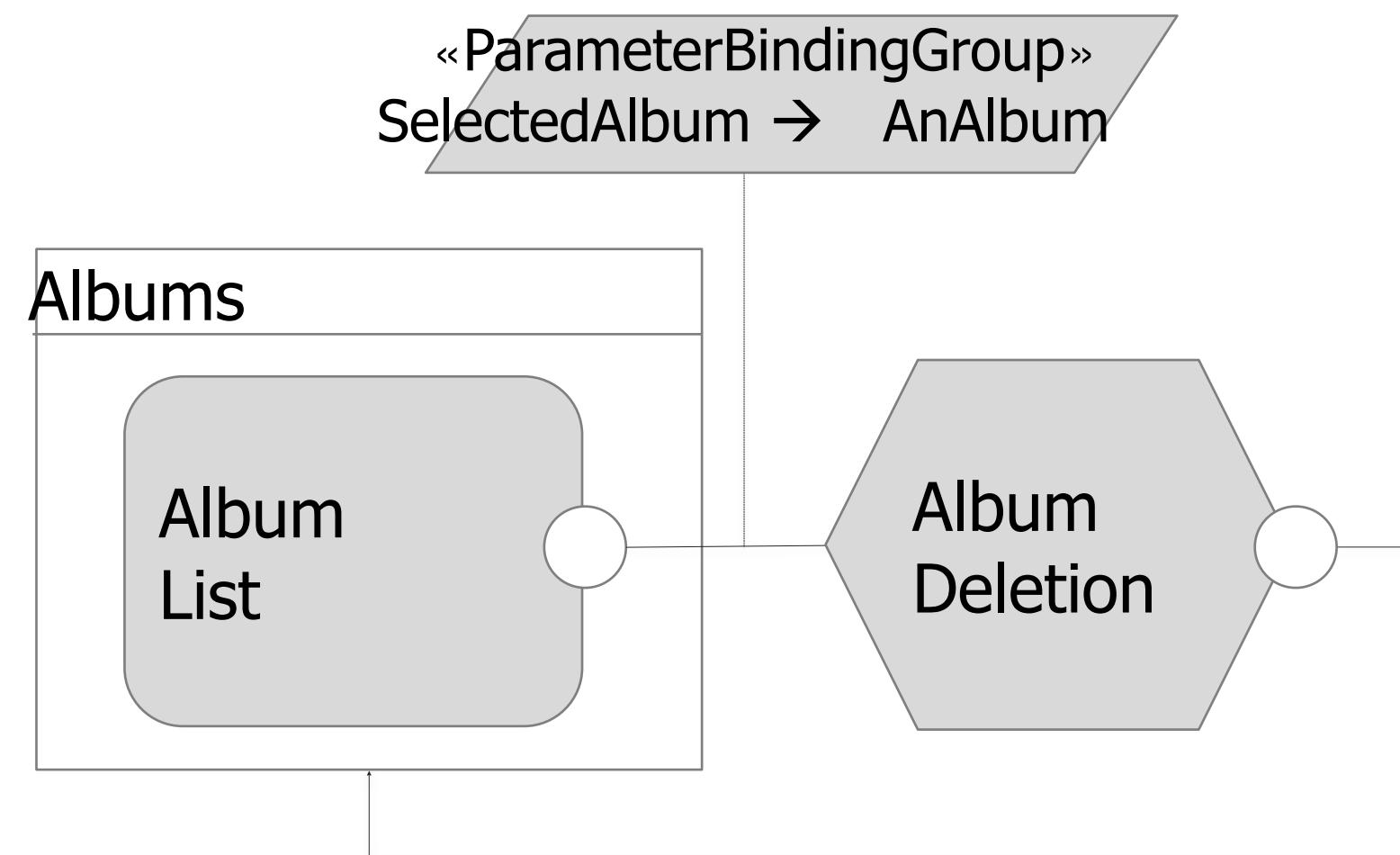


- Create
 - Creates an Entity entry (user selects entity)
 - Receives necessary data through flow and parameter binding

More IFML Syntax – Actions: Create



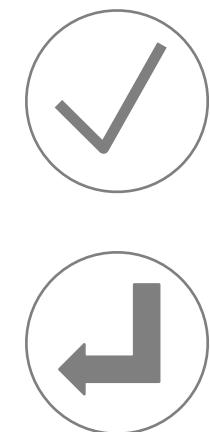
IFML by example



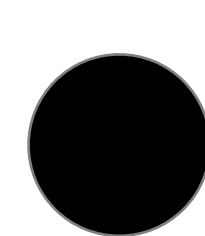
IFML – Subtyping Events

OnSelect event

OnSubmit event



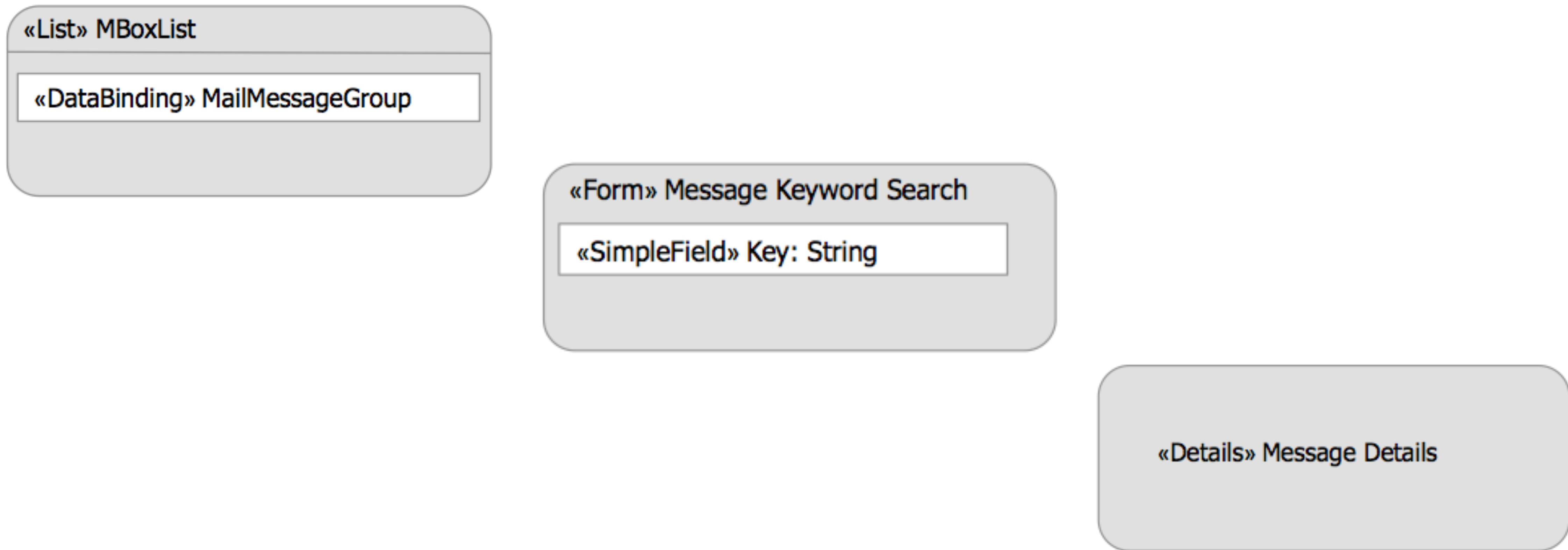
} Catching
events



Throwing
events

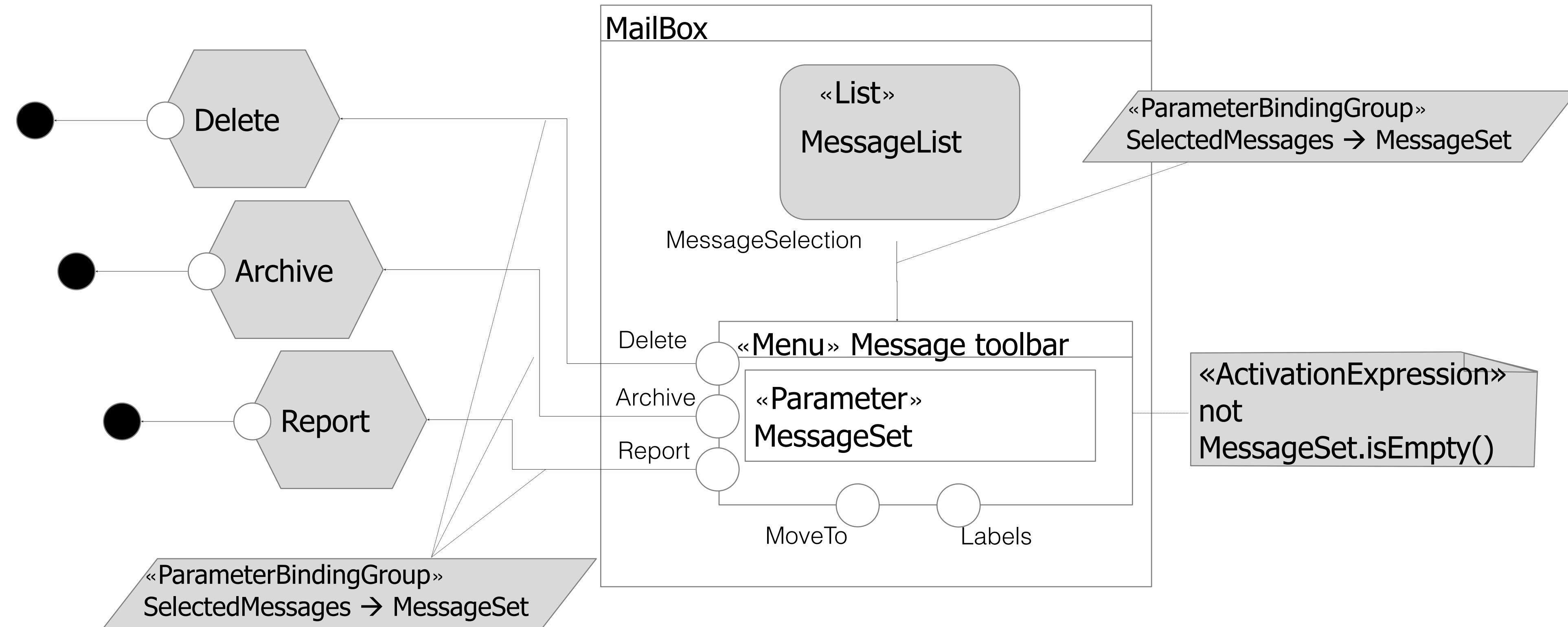
IFML – Subtyping Components

And as many others as you want!

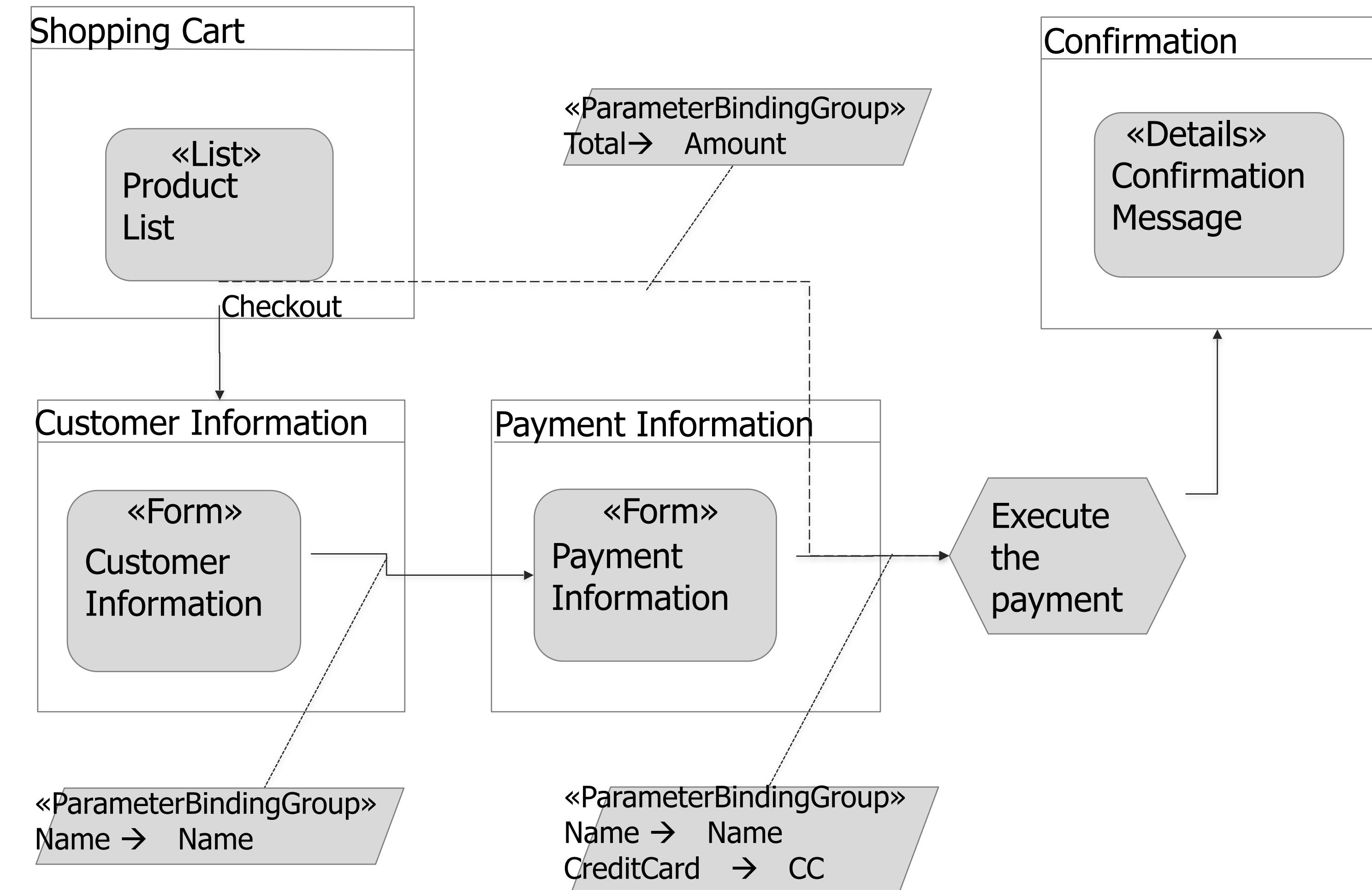


IFML by example – mailbox

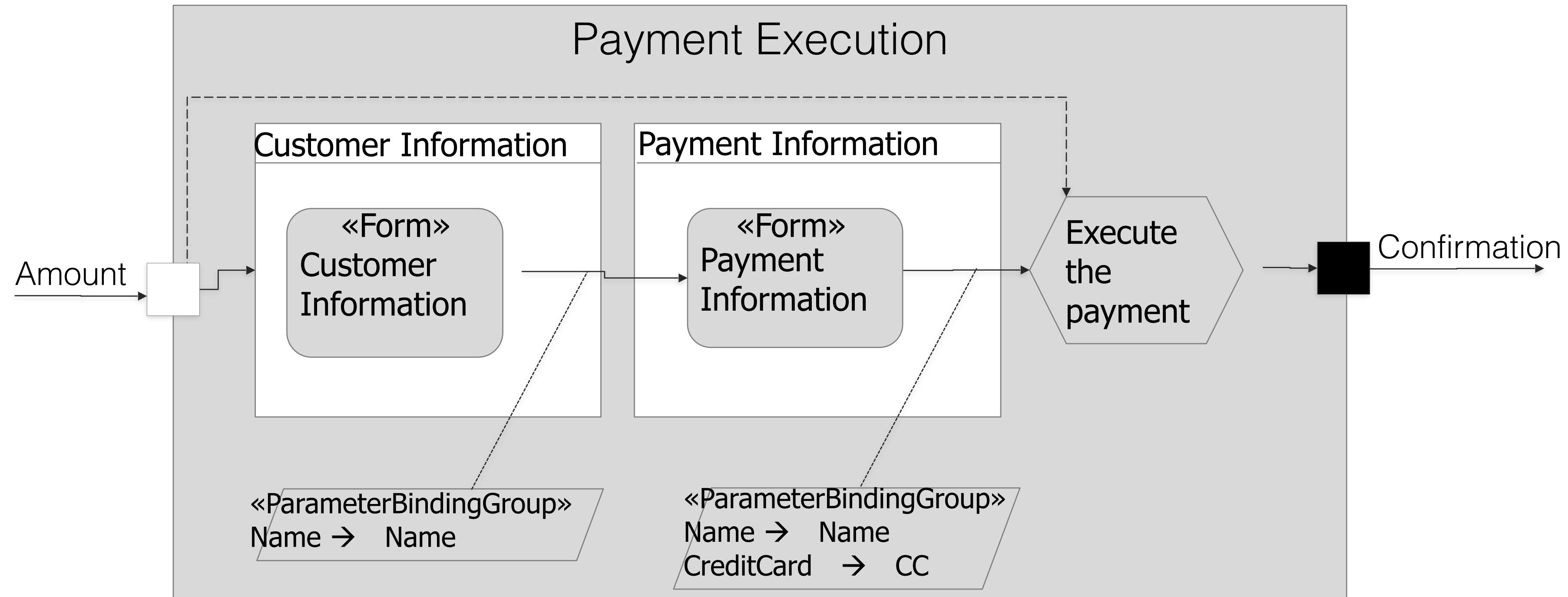
ActivationExpression, SubmitEvent, Event generation



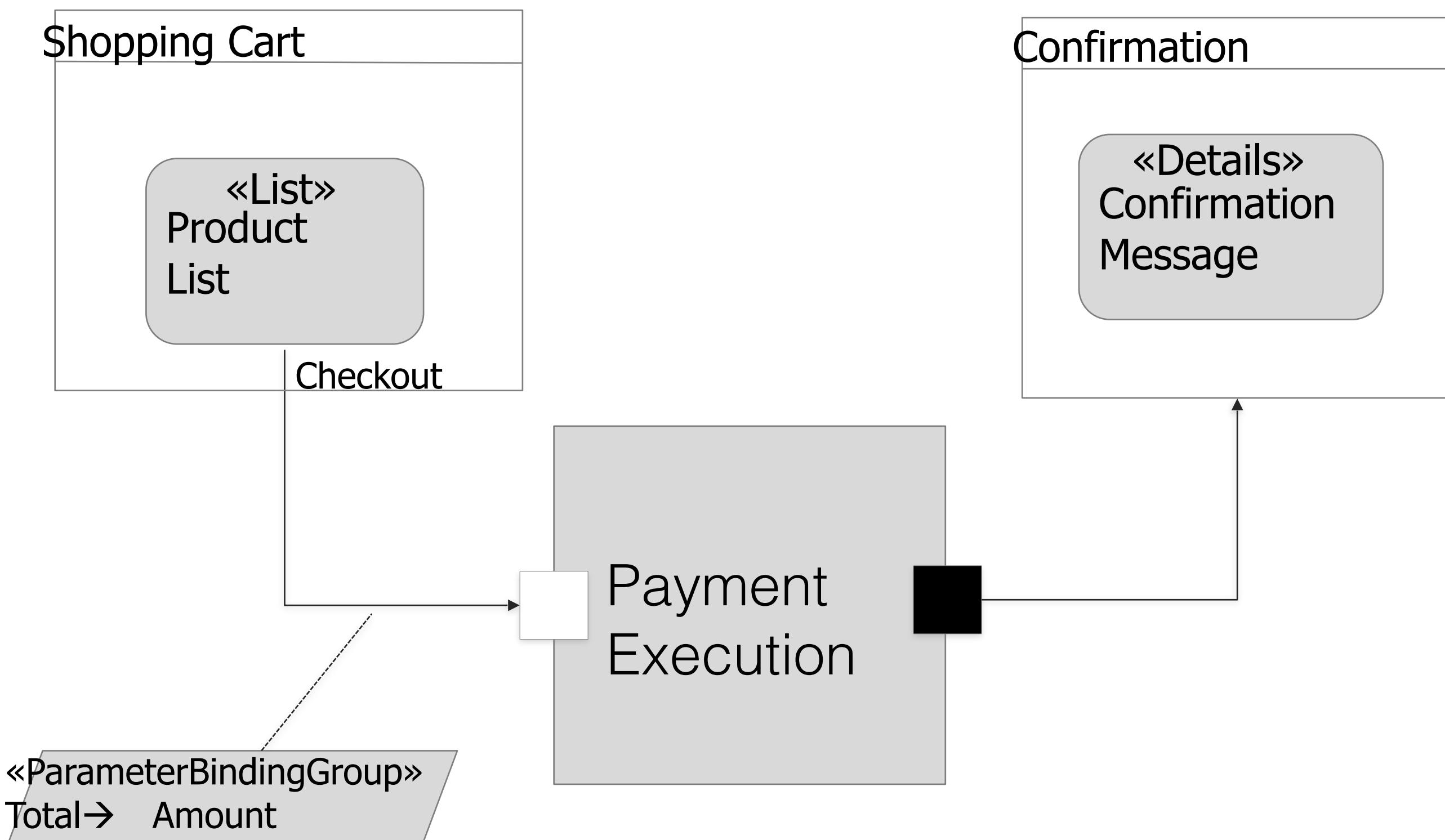
IFML example – online payment



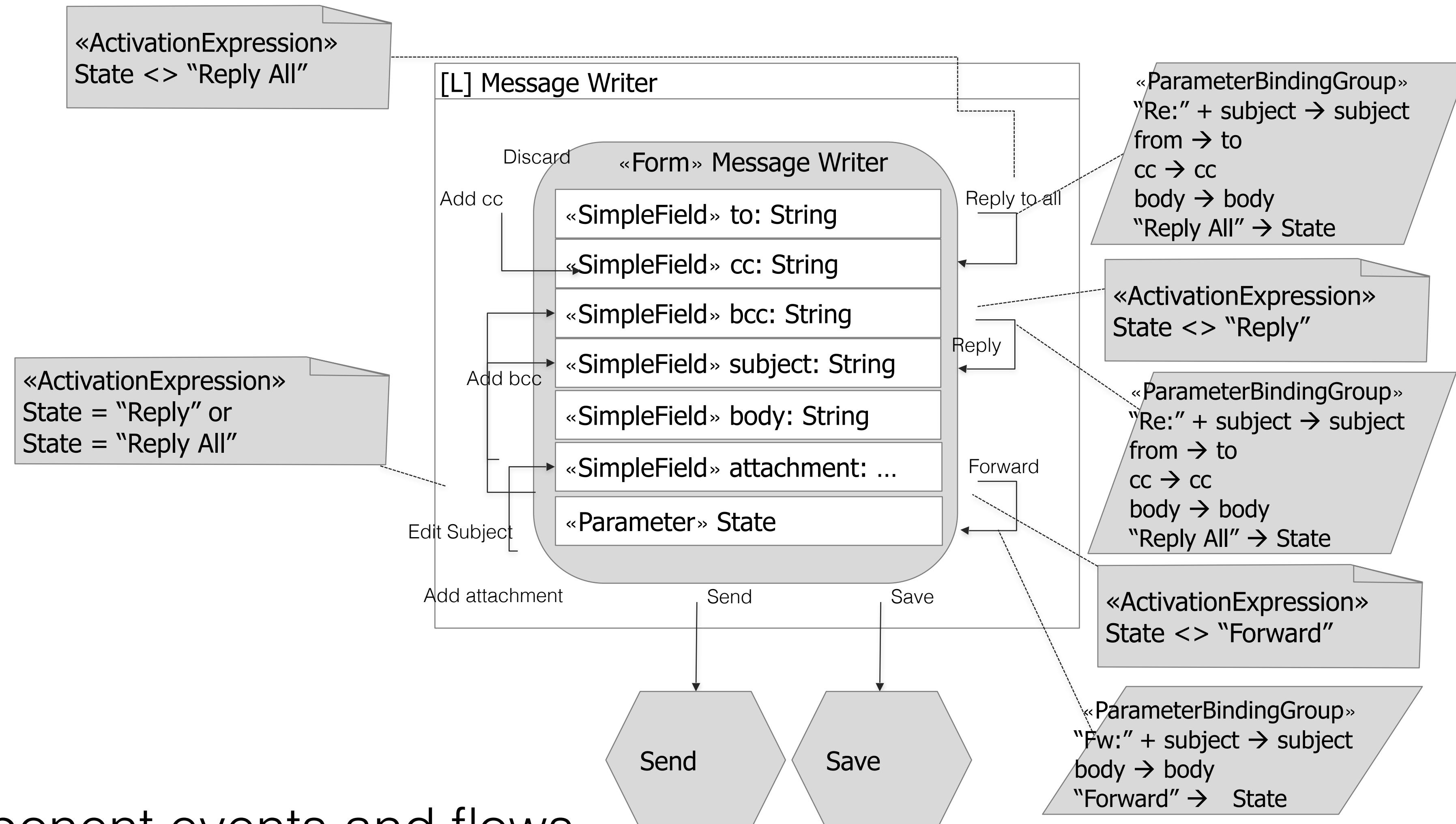
IFML – Modules



IFML – (Re)using Modules



IFML by example – email



intra-component events and flows

Tools & docs

- IFML page: ifml.org
- Eclipse Modelling Tools : <http://www.eclipse.org/downloads/packages/>
- IFML Eclipse editor: <http://ifml.github.io/>

Internet Applications Design and Implementation

(Lecture 9 - Part 3 - An App end-to-end)

**MIEI - Integrated Master in Computer Science and Informatics
Specialization block**

João Costa Seco (joao.seco@fct.unl.pt)
Jácome Cunha (jacome@fct.unl.pt)
João Leitão (jc.leitao@fct.unl.pt)

Example for today: Shared Task List

- A project management app where users have a backlog, can create tasks, divide them into sprints, assign them to users, mark them as completed, and see a variety of statistics about tasks and sprints.

Roadmap

1. User stories define a user centred application
2. IFML specifications to specify the (user stories) interactions and components.
3. React components implement each one of the IFML components and views.
4. Interface Mockups define the style and final composition of the UI.

User stories (I)

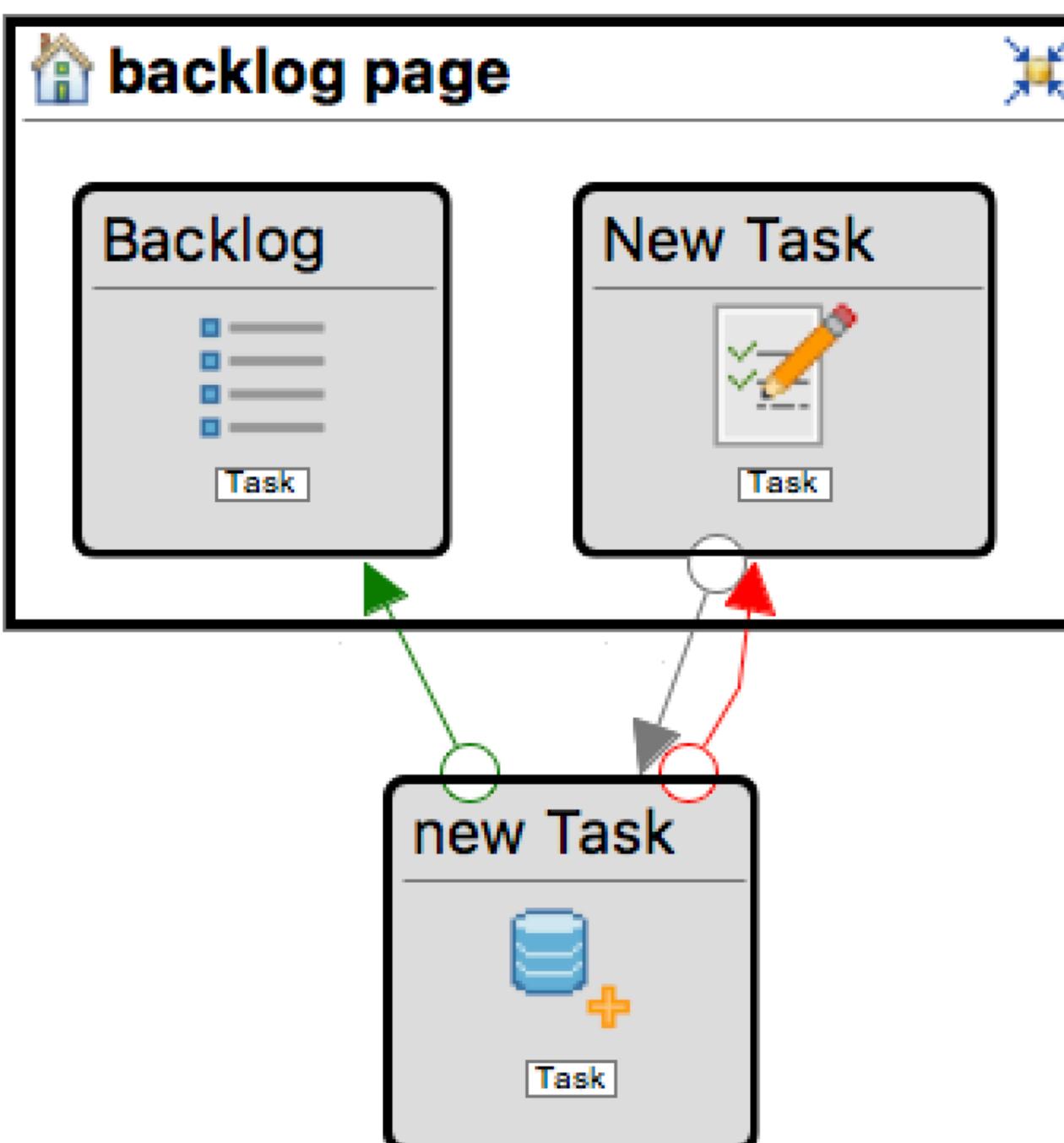
- 1.A user creates a new **task** given a name, a description, and due date, and sees the result in the **backlog**.
- 2.A user creates a new **sprint** given a name, a description, and a due date, and sees the result in the list of sprints.
- 3.A user selects a task (from the backlog or a sprint), assigns it to a (another) sprint and sees its name in the sprint task list.
- 4.A user selects a task (from the backlog or a sprint), and sees its name, description, and dates.
- 5.A user selects a task (from the backlog or a sprint), assigns it to a (another) user and sees the result in the details of the task and on the original list.
- 6.A user selects a task (from the backlog or a sprint), completes it (in a given date) and sees the end date in original list of tasks.

User Stories (II)

- 7.A user opens its home page, selects a sprint, and sees the sprint details, statistics and task list.
- 8.A user opens its home page, selects a user, and sees the user details, statistics and tasks list.
- 9.A user opens its home page, selects the backlog and sees the task list.
- 10.A user opens its home page, searches the backlog using a text and task properties, and sees the task list filtered by the given criteria.

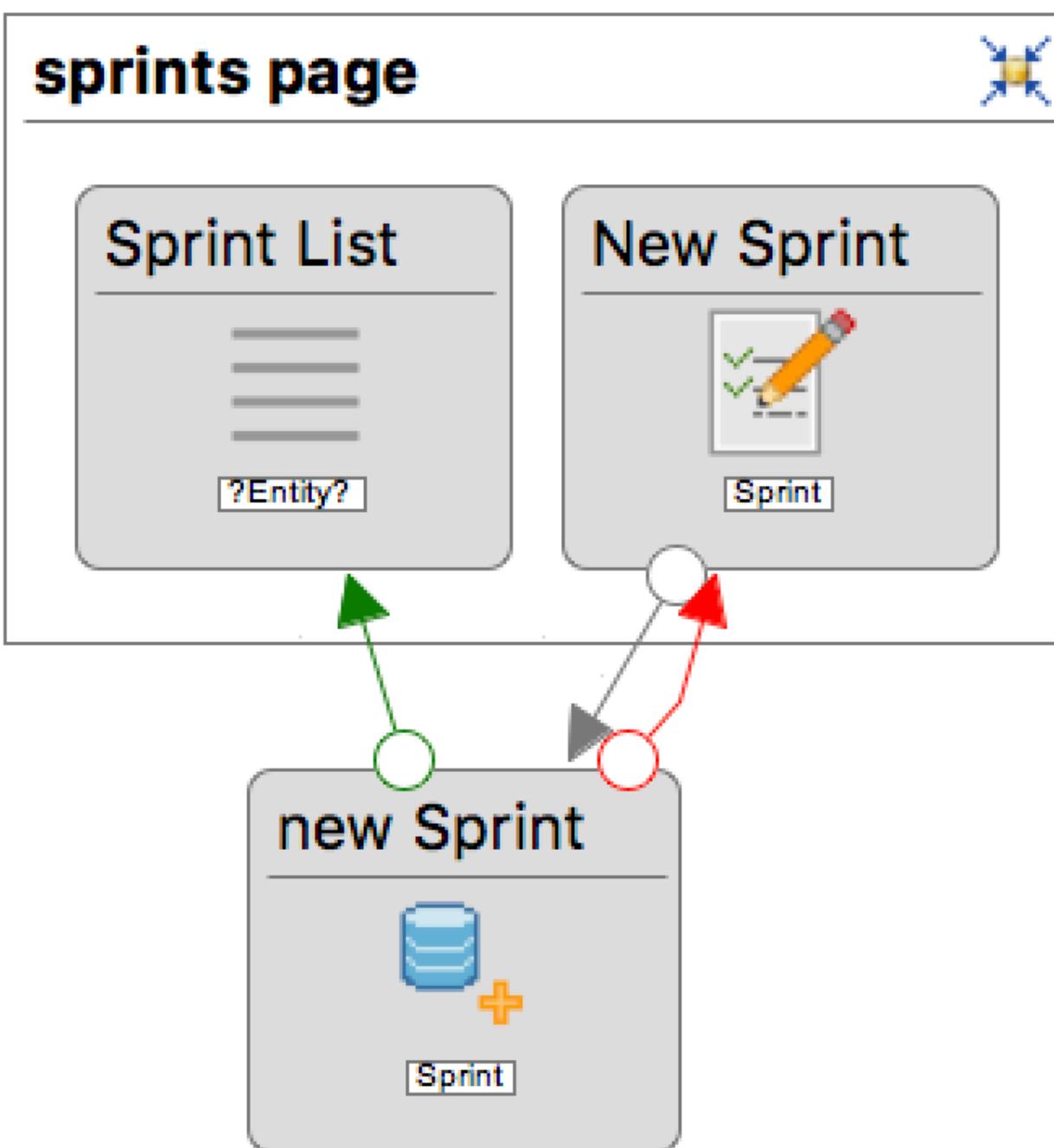
IFML Specification

- A user creates a new **task** given a name, a description, and due date, and sees the result in the **backlog**.



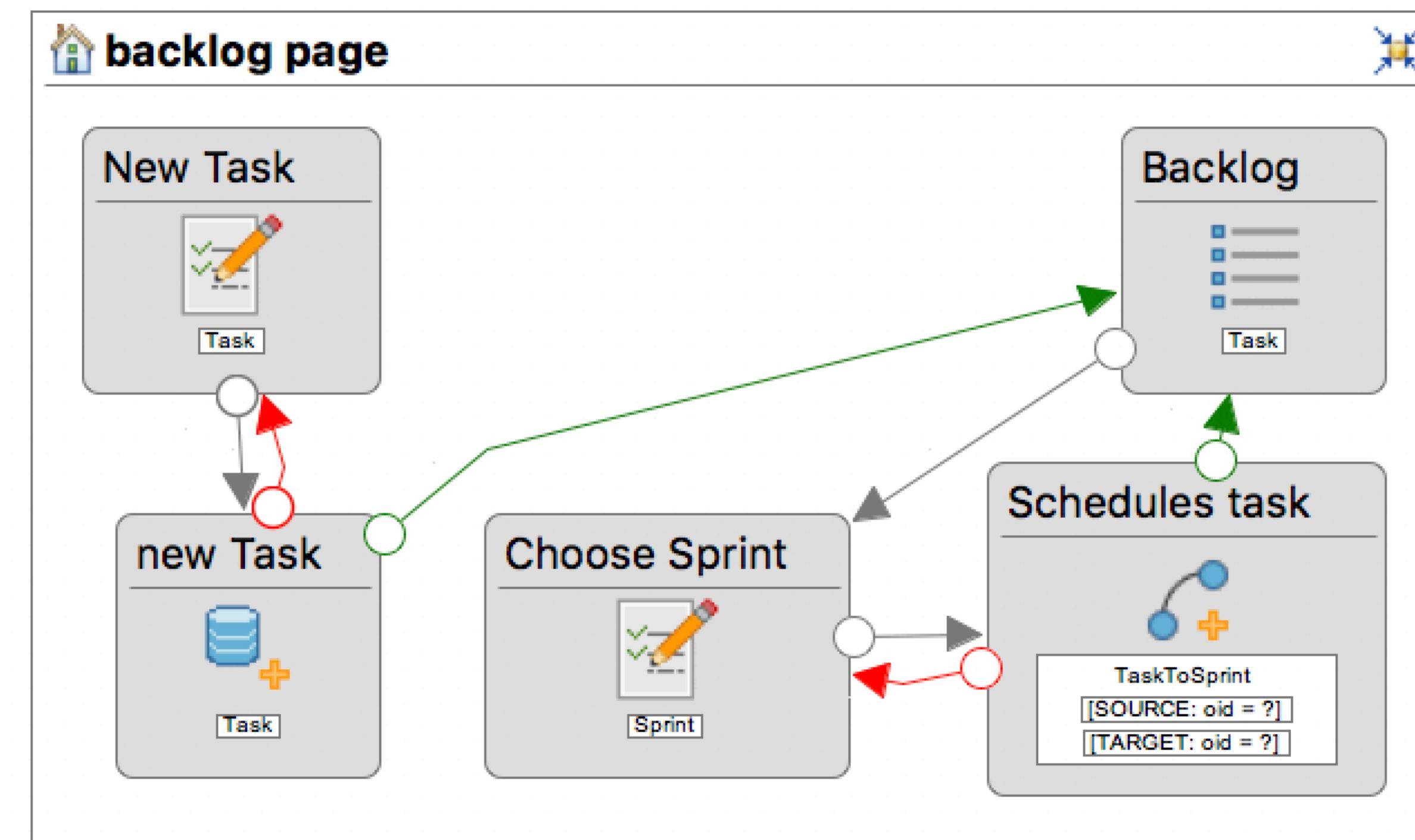
IFML Specification

- A user creates a new **sprint** given a name, a description, and a due date, and sees the result in the list of sprints.



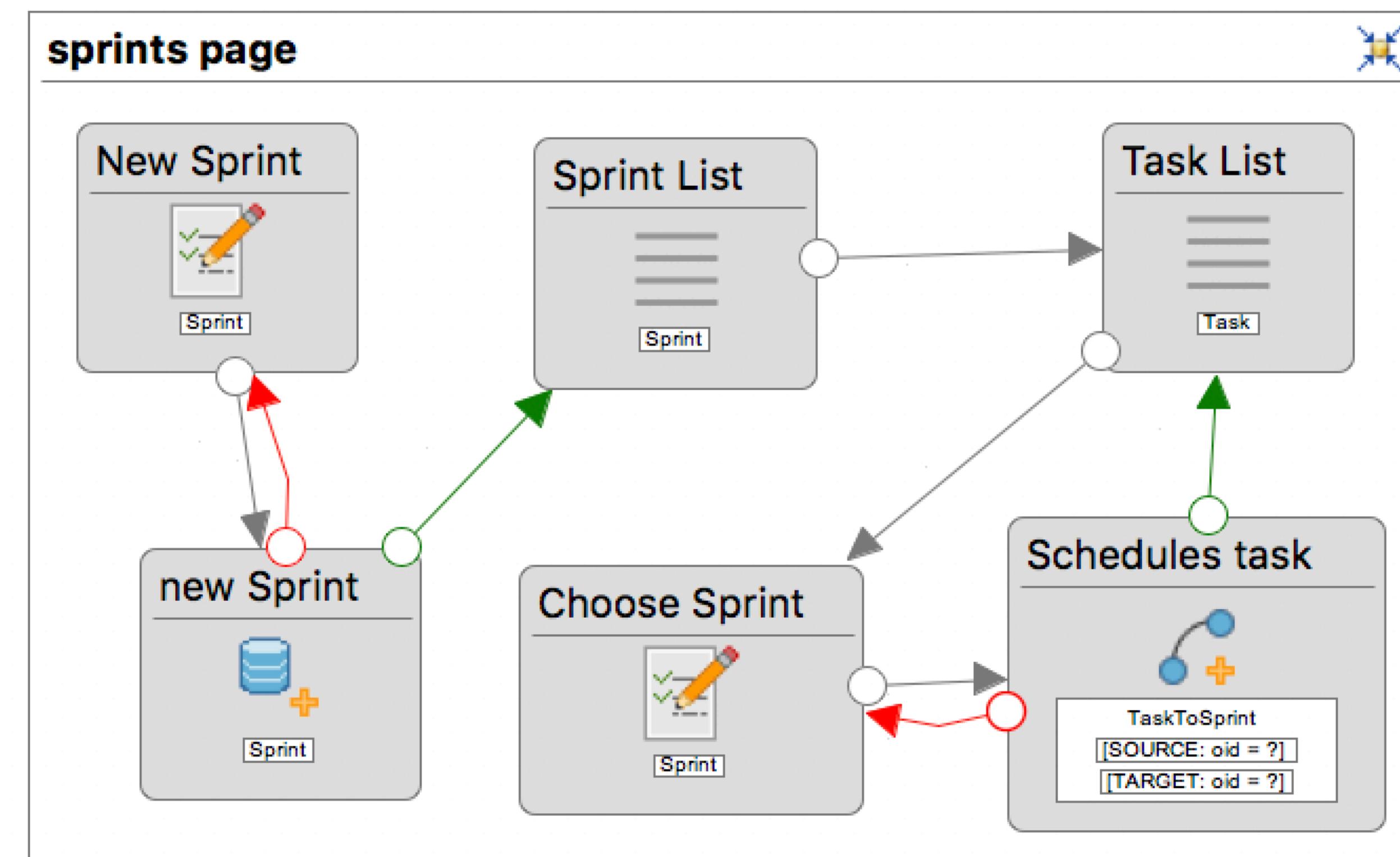
IFML Specification

- A user selects a task from the backlog, assigns it to a (another) sprint and sees its name in the sprint task list.



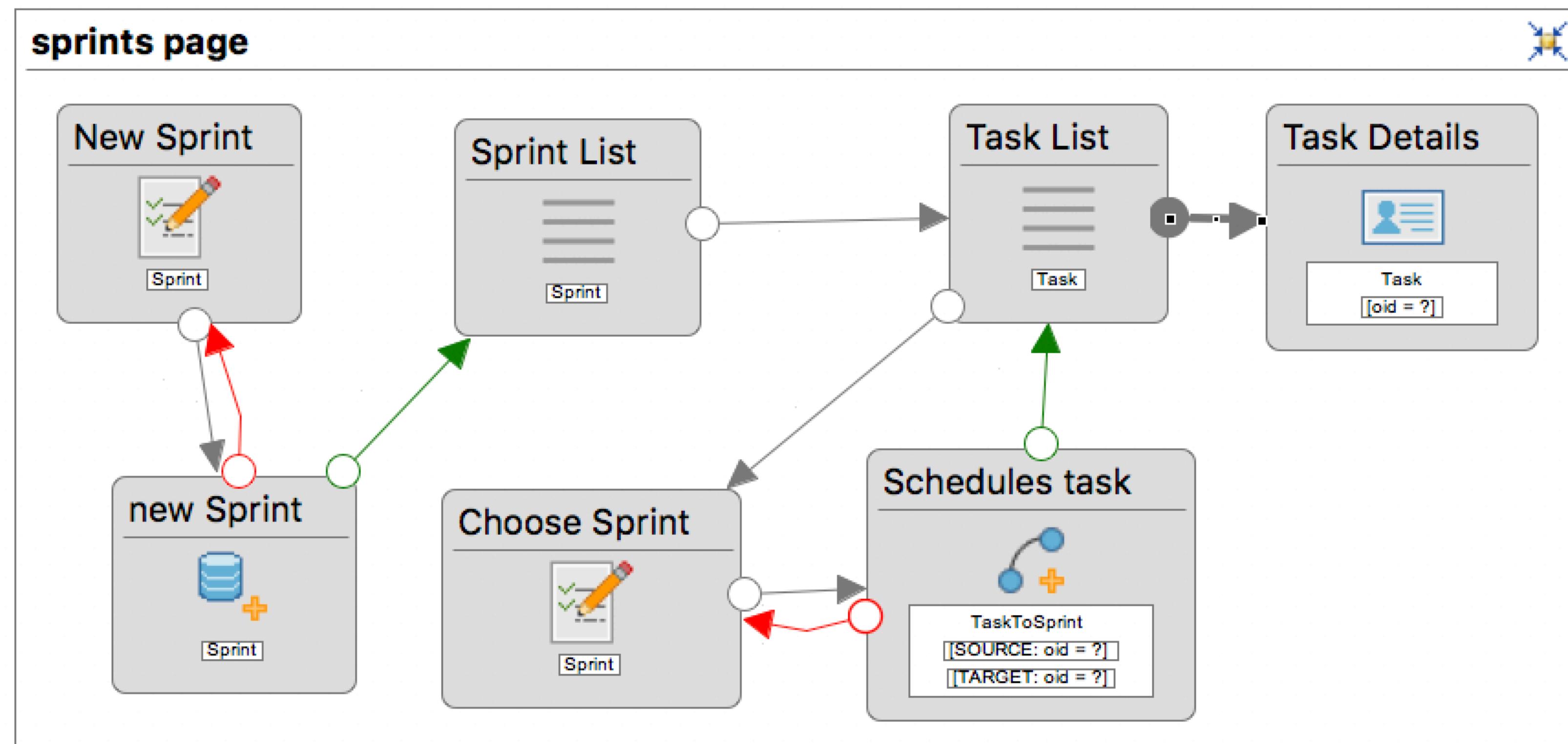
IFML Specification

- A user selects a task from a sprint, assigns it to a (another) sprint and sees its name in the sprint task list.



IFML Specification

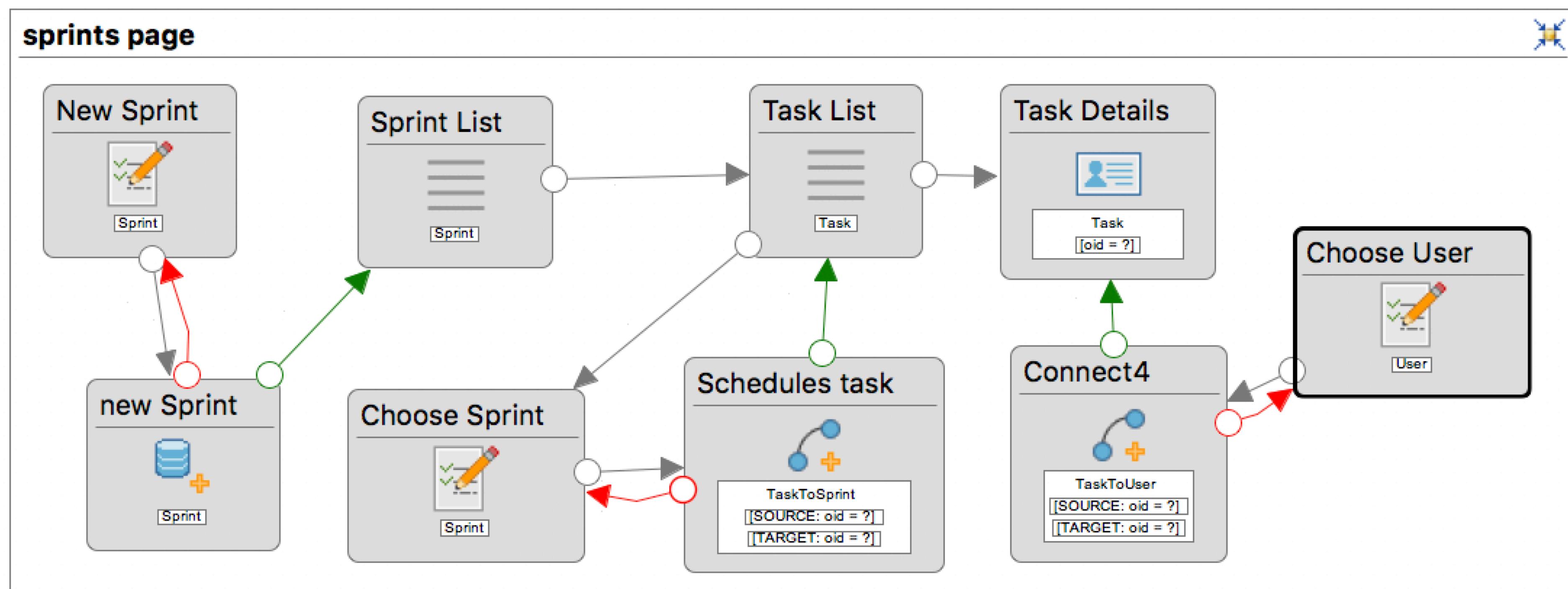
- A user selects a task from a sprint, and sees its name, description, and dates.



IFML Specification

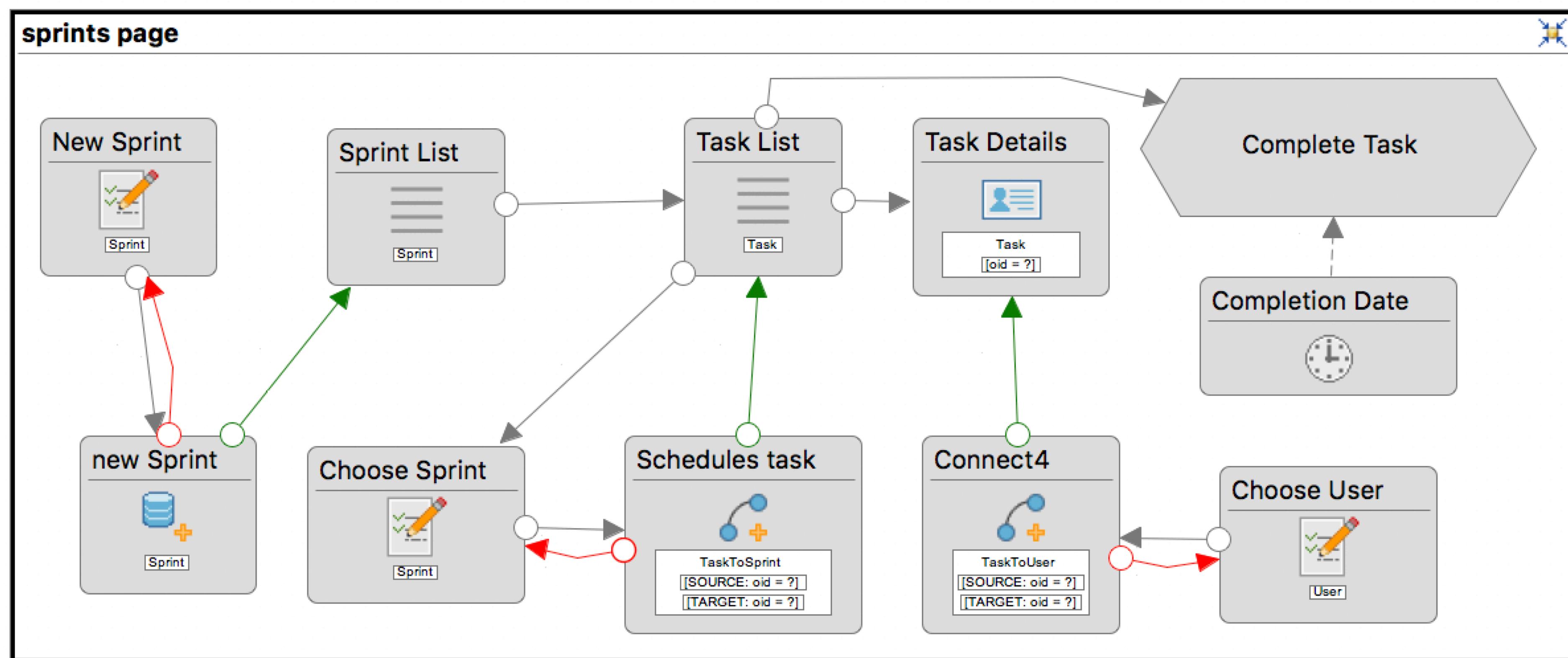
- A user selects a task from a sprint, assigns it to a (another) user and sees the result in the details of the task and on the original list.

-



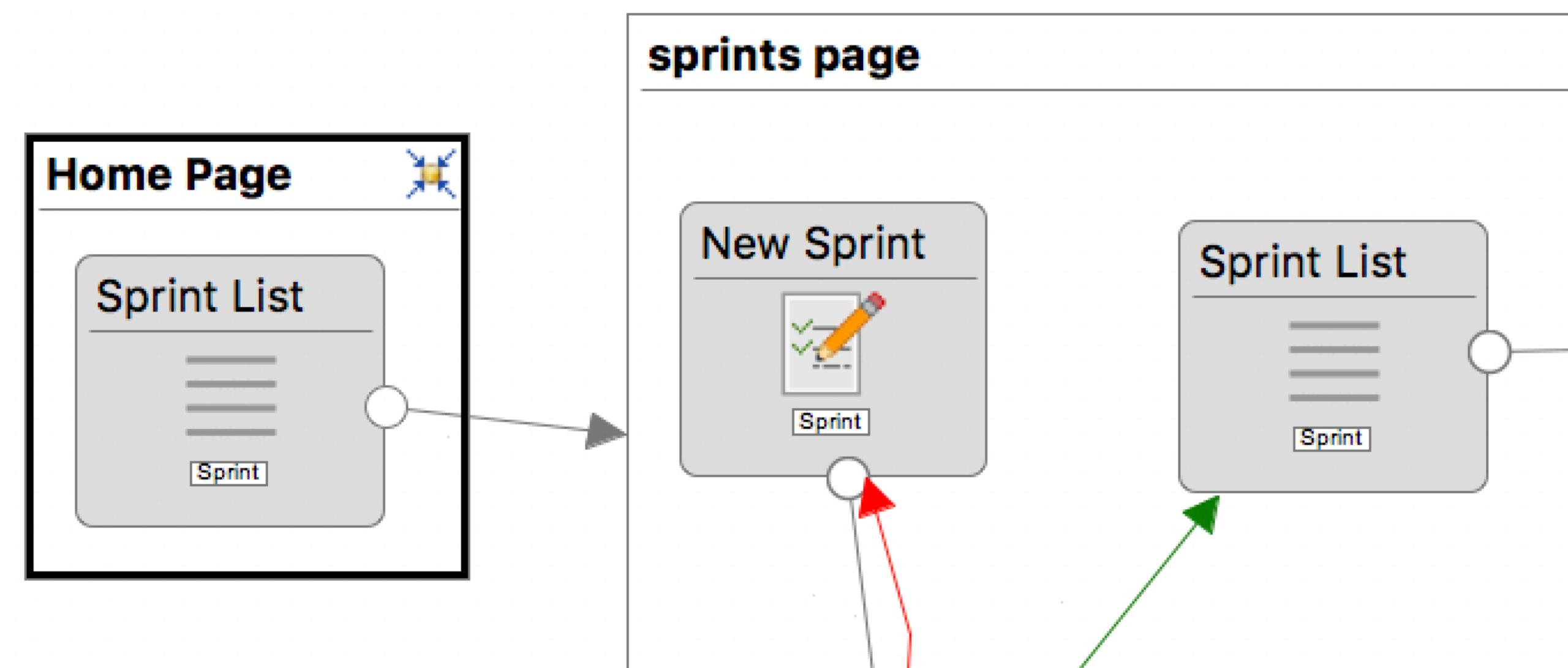
IFML Specification

- A user selects a task from a sprint, completes it (in a given date) and sees the end date in original list of tasks.



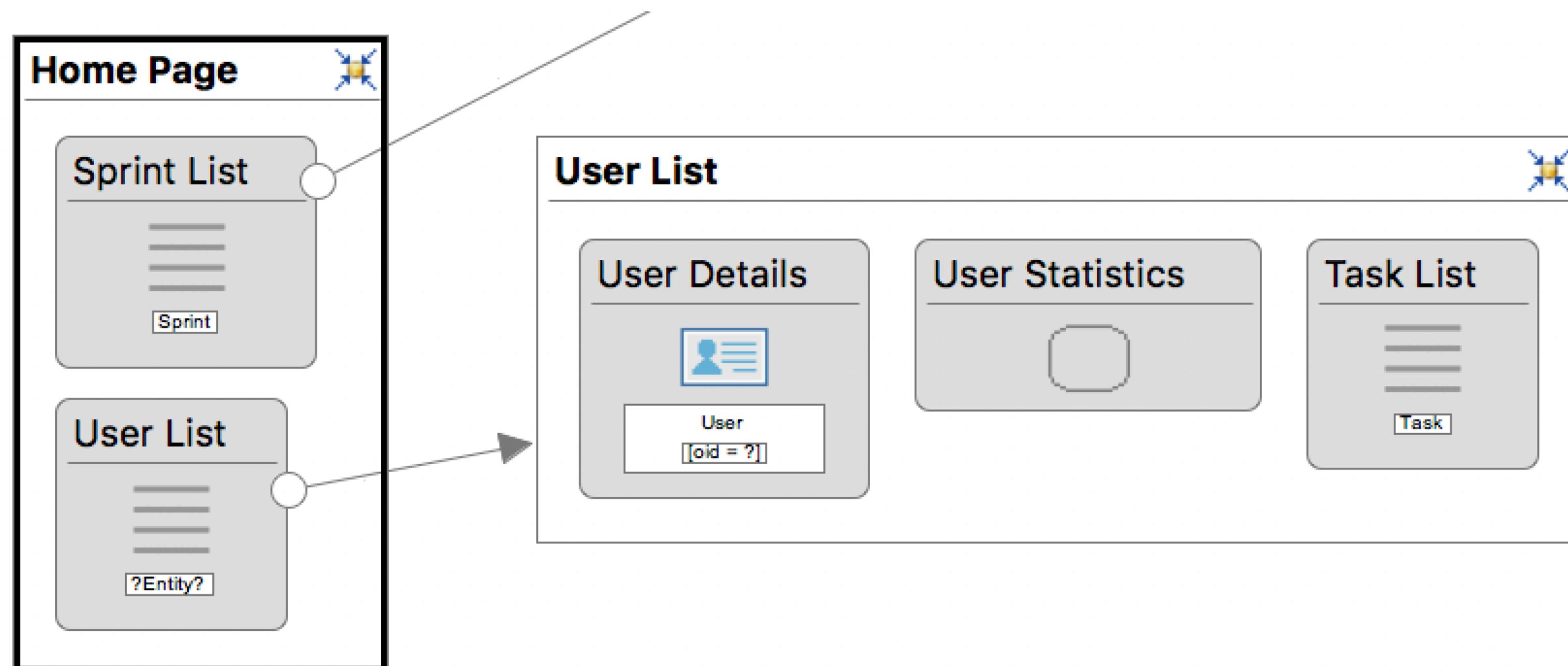
IFML Specification

- A user opens its home page, selects a sprint, and sees the sprint details, statistics and task list.



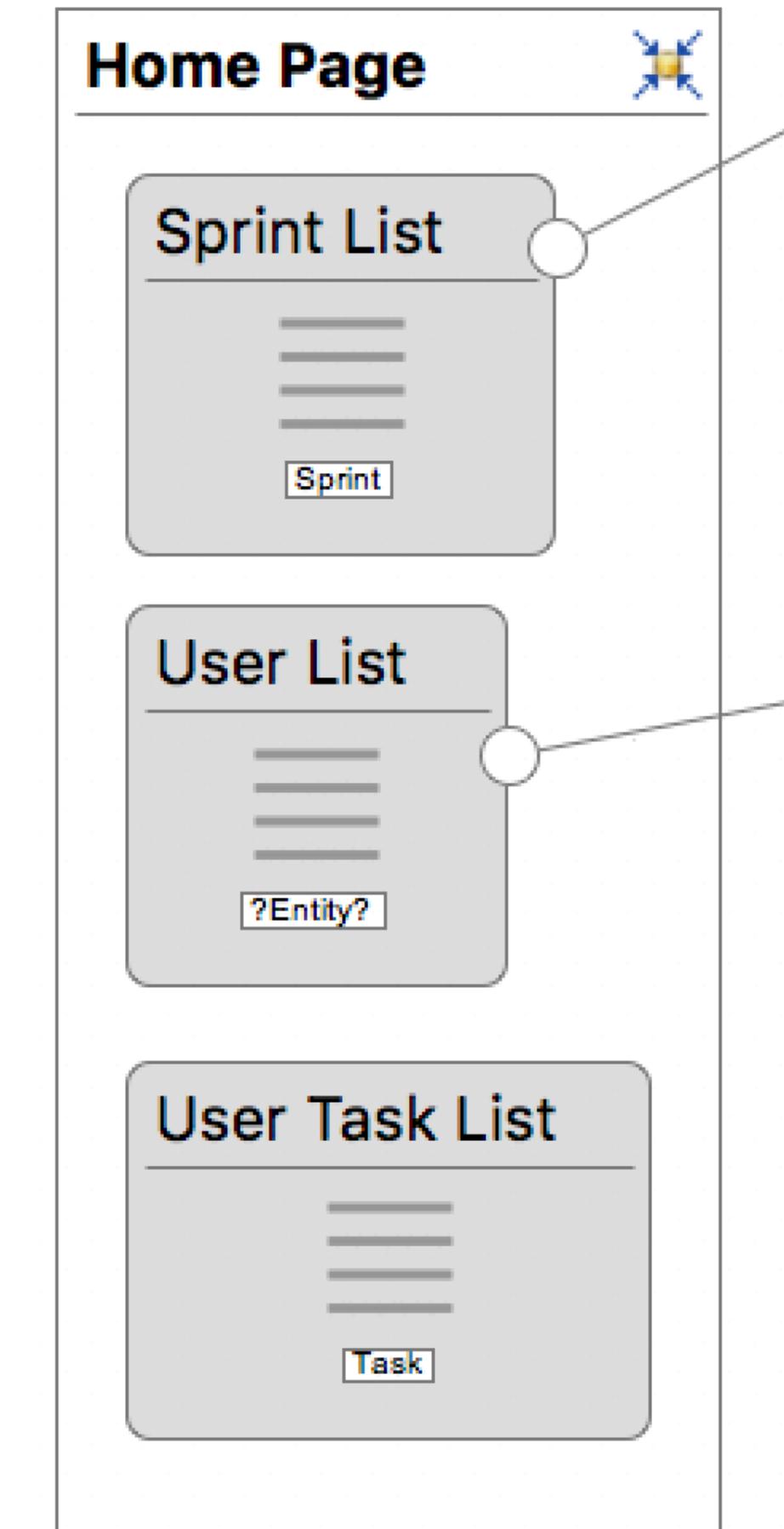
IFML Specification

- A user opens its home page, selects a user, and sees the user details, statistics and tasks list.



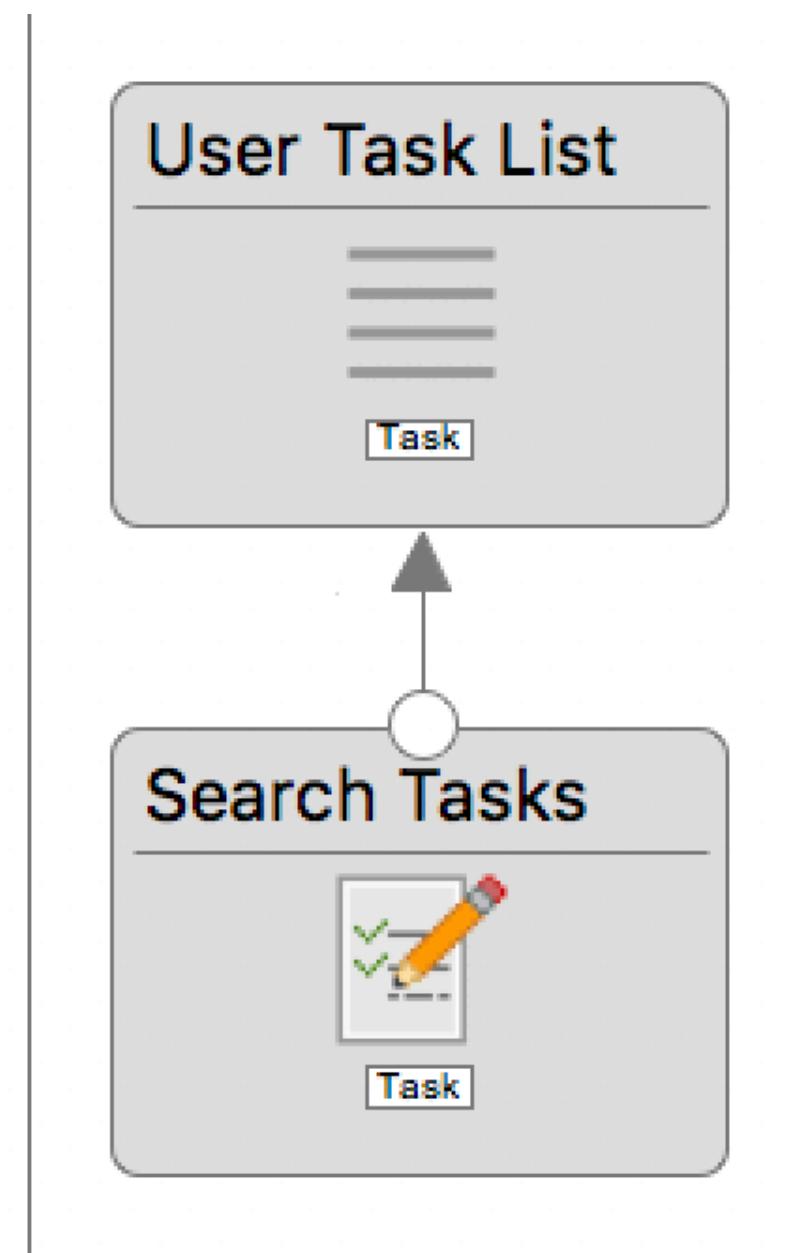
IFML Specification

- A user opens its home page, selects the backlog and sees the task list.

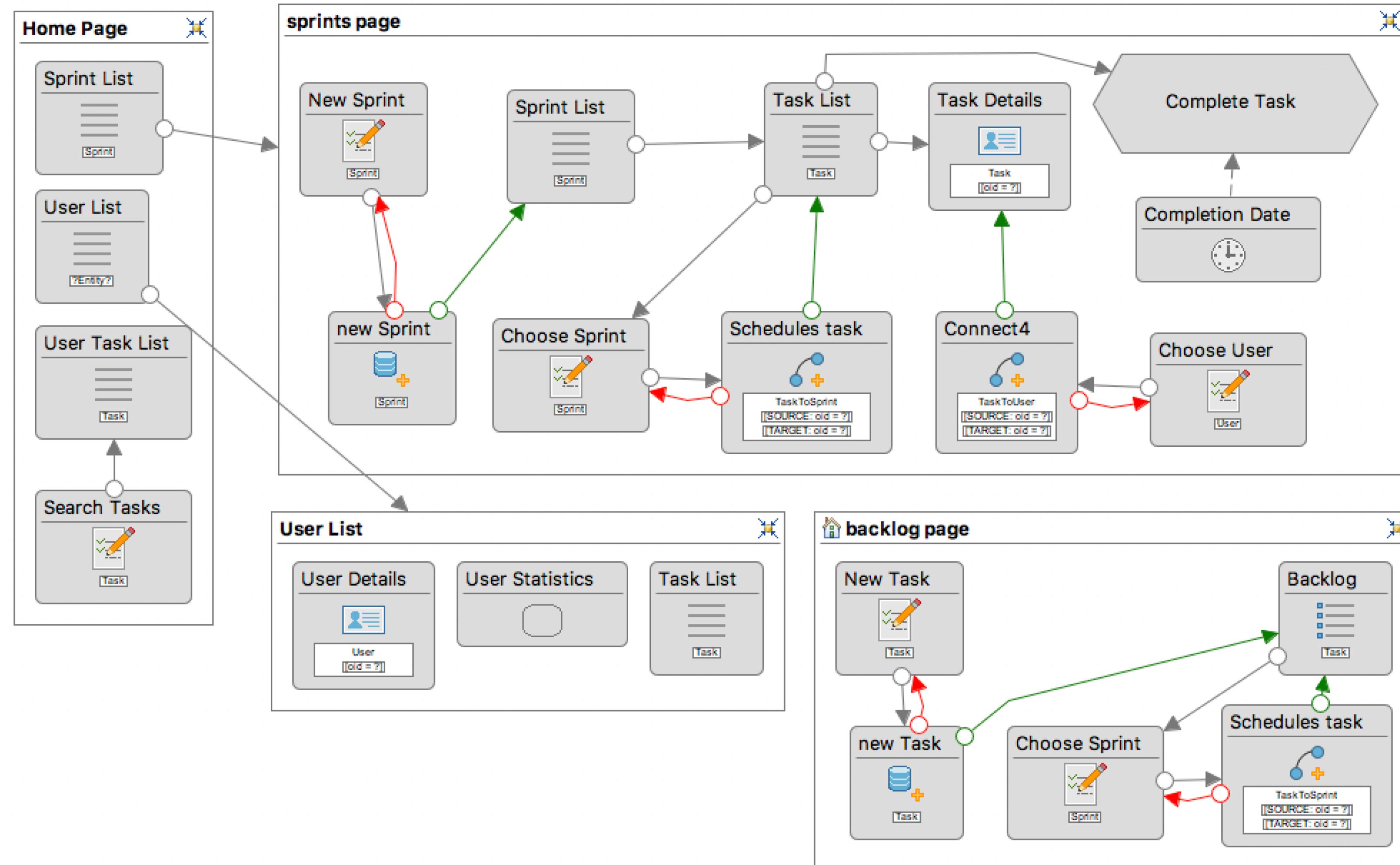


IFML Specification

- A user opens its home page, searches the backlog using a text and task properties, and sees the task list filtered by the given criteria.



IFML Specification: The full picture



React structure

```
import * as React from "react";
import { BrowserRouter as Router, Route, Link } from "react-router-dom";

export interface HelloProps { compiler: string; framework: string; }

export const Dashboard = () => <div>Dashboard</div>

export const BackLog = () => <div>BackLog</div>

export const Sprints = () => <div>Sprints</div>

export const Users = () => <div>Users</div>

export const App = () =>
  <Router>
    <div className="container">
      <h1>Task List</h1>
      <div>
        <ul>
          <li><Link to="/"> Home </Link></li>
          <li><Link to="/backlog"> BackLog </Link></li>
          <li><Link to="/sprints"> Sprints </Link></li>
          <li><Link to="/users"> Users </Link></li>
        </ul>
      </div>
      <div>
        <div>
          <Route exact path="/" component={Dashboard}/>
          <Route path="/backlog" component={BackLog}/>
          <Route path="/sprints" component={Sprints}/>
          <Route path="/users" component={Users}/>
        </div>
      </div>
    </div>
  </Router>
```

React structure



```
const SearchCriteria = () =>
  <form>
    <input type="text" name="search" />
    <input type="submit" value="Search" />
  </form>

const SeacheableUserTaskList = () =>
  <div>
    <TaskList/>
    <SearchCriteria/>
  </div>

const Dashboard = () =>
  <div>
    <h1>Dashboard</h1>
    <SprintList/>
    <UserList/>
    <SeacheableUserTaskList/>
  </div>
```

```
const UserList = () =>
  <div>
    <h2>Users</h2>
    <ul>
      <li>User 1</li>
      <li>User 2</li>
      <li>User 3</li>
      <li>User 4</li>
    </ul>
  </div>

const SprintList = () =>
  <div>
    <ul>
      <li>Sprint 1</li>
      <li>Sprint 2</li>
      <li>Sprint 3</li>
      <li>Sprint 4</li>
    </ul>
  </div>

const TaskList = () =>
  <div>
    <h2>Tasks</h2>
    <ul>
      <li>Task 1</li>
      <li>Task 2</li>
      <li>Task 3</li>
      <li>Task 4</li>
    </ul>
  </div>
```

React structure

```
export class App extends Component<any,any> {

    constructor(props: any) {
        super(props);
        this.state = {
            sprints: [],
            tasks: []
        }
    }

    render() {
        return (
            <Router>
                <div className="container">
                    <h1>Task List</h1>
                    <div>
                        <ul>
                            <li><Link to="/"> Home </Link></li>
                            <li><Link to="/sprints"> Sprints </Link></li>
                            <li><Link to="/backlog"> BackLog </Link></li>
                            <li><Link to="/users"> Users </Link></li>
                        </ul>
                    </div>

                    <div>
                        <div>
                            <Route exact path="/" component={() => <DashboardPage sprints={this.state.sprints}/>} />
                            <Route path="/sprints" component={Sprints} />
                            <Route path="/backlog" component={BackLogPage} />
                            <Route path="/users" component={UsersPage} />
                        </div>
                    </div>
                </div>
            </Router>
        )
    }
}
```

Summary

Pros and Cons of using specs and frameworks

- Formal Design and Method
- Communication between stakeholders
- Documentation for future reference
 - It's important to use tools that sync specs and code
- Maintenance and evolution
 - Allows the analysis of the impact of code evolutions