



Der Event-Bus

Lose Kopplung von wiederverwendbaren Bauteilen in der GUI-Entwicklung

Sommersemester 2024

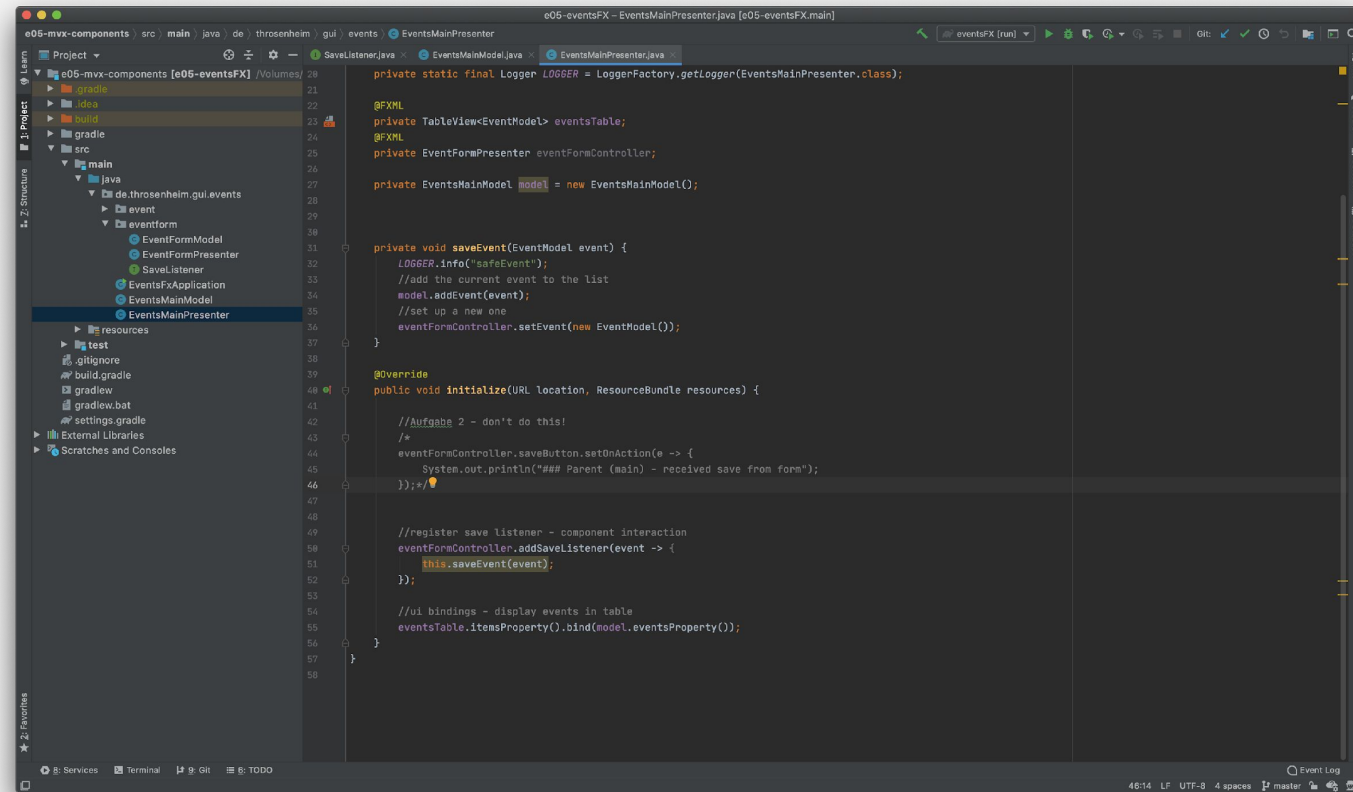
Stefan Langer

Veronika Dashuber

Demo 1: Musterlösung der Übung 5



QA|WARE



The screenshot shows an IDE window titled "e05-eventsFX - EventsMainPresenter.java [e05-eventsFX.main]". The left sidebar displays a project structure for "e05-mvx-components [e05-eventsFX]". The main editor area shows the following Java code:

```
20 private static final Logger LOGGER = LoggerFactory.getLogger(EventsMainPresenter.class);
21
22 @FXML
23 private TableView<EventModel> eventsTable;
24
25 @FXML
26 private EventFormPresenter eventFormController;
27
28 private EventsMainModel model = new EventsMainModel();
29
30
31 private void saveEvent(EventModel event) {
32     LOGGER.info("saveEvent");
33     //add the current event to the list
34     model.addEvent(event);
35     //set up a new one
36     eventFormController.setEvent(new EventModel());
37 }
38
39
40 @Override
41 public void initialize(URL location, ResourceBundle resources) {
42     //Aufgabe 2 - don't do this!
43     /*
44     eventFormController.saveButton.setOnAction(e -> {
45         System.out.println("### Parent (main) - received save from form");
46     });*/
47
48
49     //register save listener - component interaction
50     eventFormController.addSaveListener(event -> {
51         this.saveEvent(event);
52     });
53
54     //ui bindings - display events in table
55     eventsTable.itemsProperty().bind(model.eventsProperty());
56 }
57
58 }
```



Eigenschaften von GUI Komponenten

- Mehrfach instanzierbar
 - Jede Komponente benötigt eigene Daten (Model, Presenter, View)
- Eventbasiert
 - Komponenten benachrichtigen Ihre Nutzer mittels Event-Schnittstellen
- Wiederverwendbar
 - Eine Komponente kann in beliebigen Dialogen eingesetzt werden
- Kontextfrei
 - Eine Komponente darf nicht das Väterelement bzw. den umgebenden Kontext kennen.

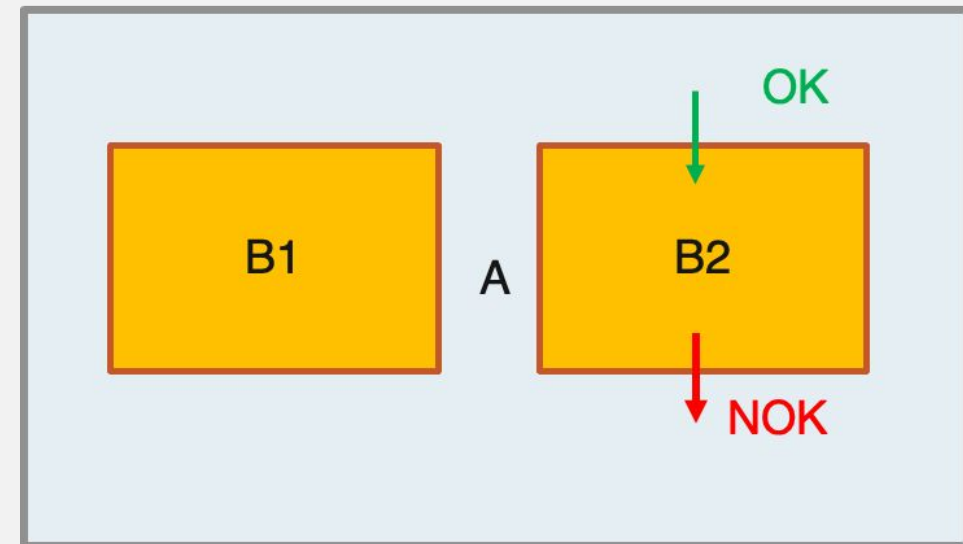
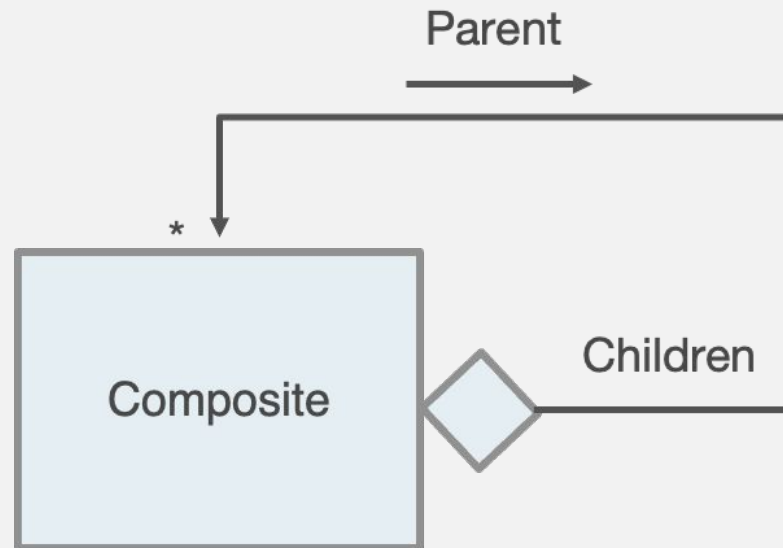
Kontextfreie Komponenten

Wiederholung



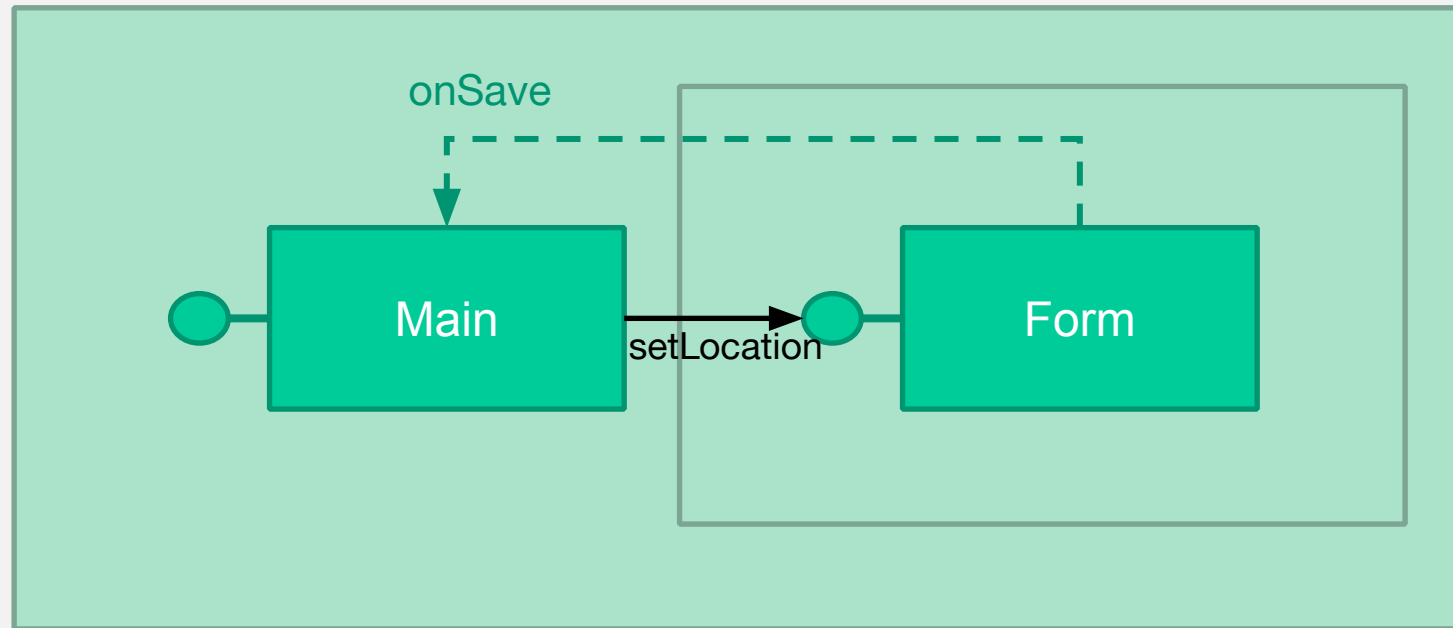
QA|WARE

Damit eine Komponente wiederverwendbar ist darf sie nicht direkt den Parent aufrufen.

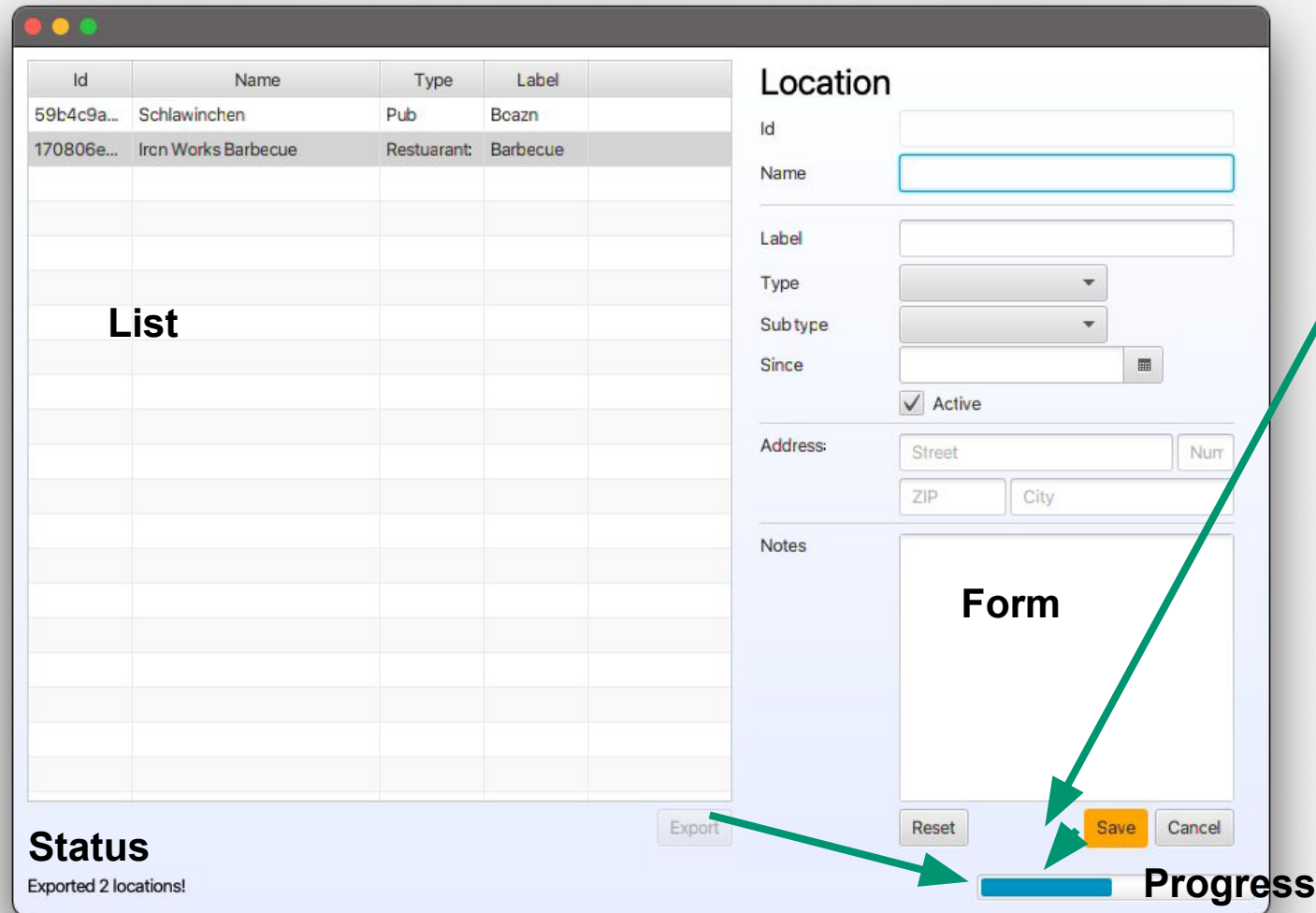


ABER: Die Kommunikation von B zu A ist zum Beispiel über Listener möglich.

Beispiel: Einfache Kommunikation durch direkte Integration und Events



Unser Problem: Wie aktualisiert man den Status oder Progressbar einer anderen Ansicht?



The screenshot shows a QA|WARE application interface with three main sections:

- List:** A table with columns: Id, Name, Type, Label. It contains two rows of data:

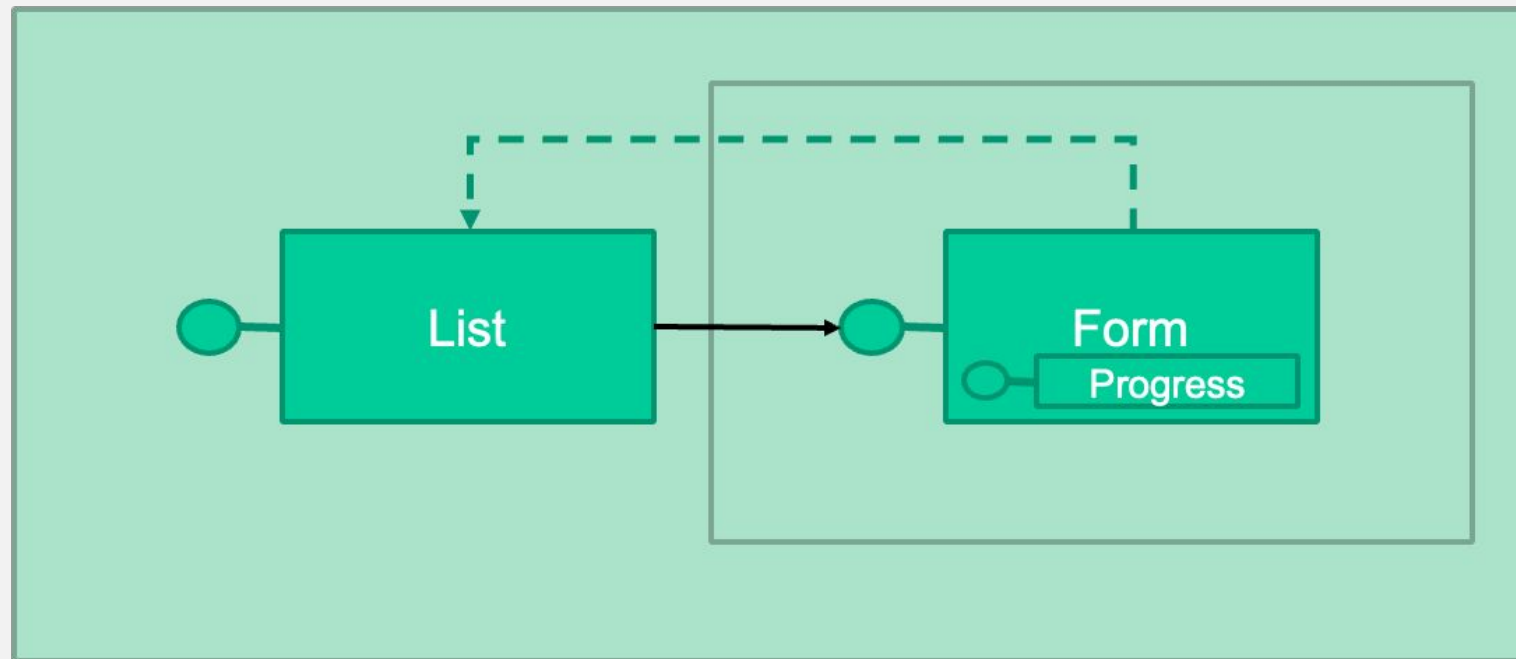
Id	Name	Type	Label
59b4c9a...	Schlawinchen	Pub	Boazn
170806e...	Ircn Works Barbecue	Restuarant:	Barbecue
- Form:** A section for editing a record, including fields for Id, Name, Label, Type, Subtype, Since, Active (checkbox), Address (Street, Nurr, ZIP, City), and Notes.
- Status:** A bar at the bottom left showing "Exported 2 locations!" and an "Export" button.

A green arrow points from a question mark to the "Save" button in the Form section. Another green arrow points from the "Export" button in the Status bar to the "Progress" bar at the bottom right.

Lösung A: List integriert Form-View

- View A (List) integriert View B (Form)

→ Verwendung des List-View ohne den Form-View ist nicht möglich



List-Controller hält eine Referenz auf den Form-Controller, führt das Binding durch und reagiert auf Events.

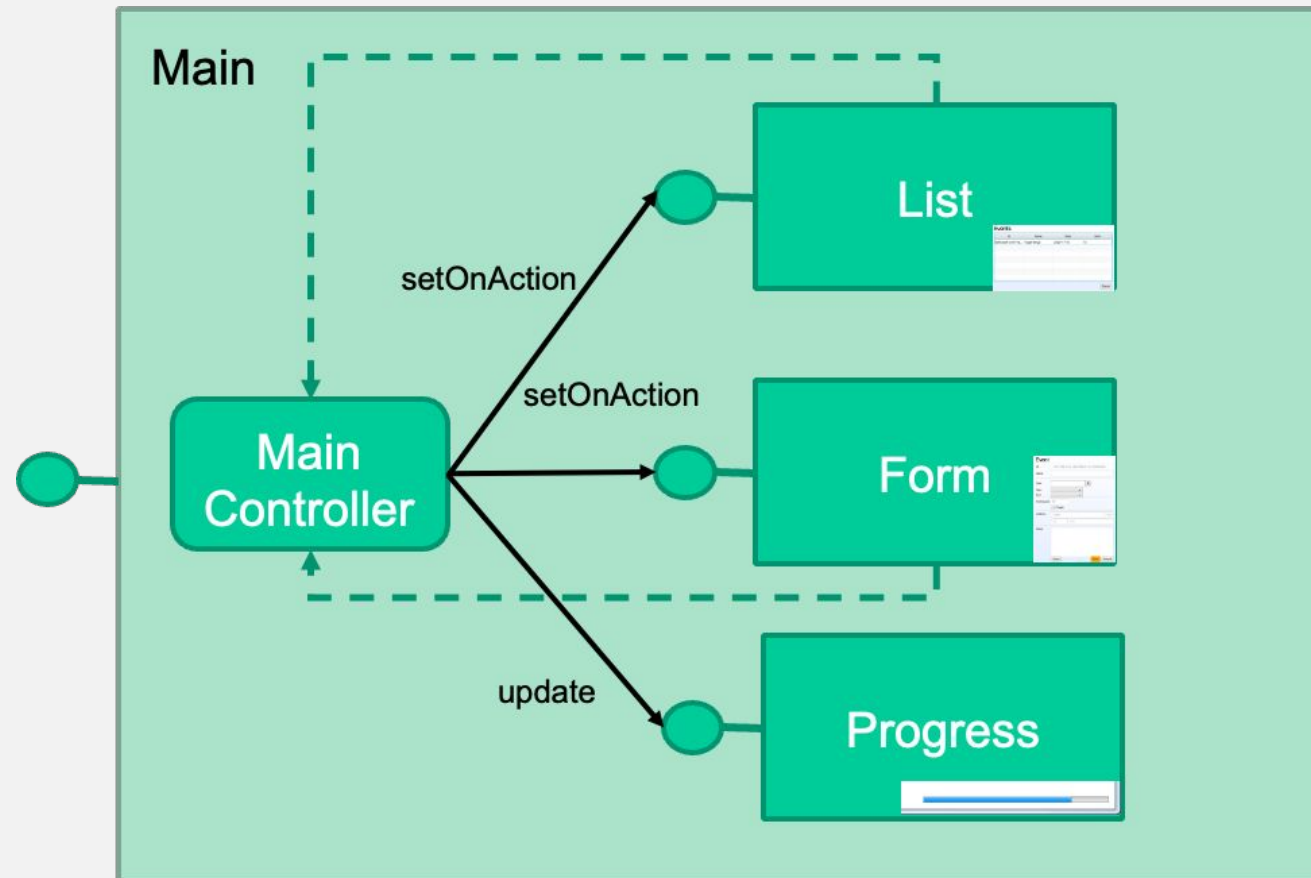
Lösung B: Übergeordneter „Main-View“



QA|WARE

- Es gibt eine übergeordnete Ansicht die per Events benachrichtigt wird und an den untergeordneten View delegiert

→ Codemenge und Aufwand steigt



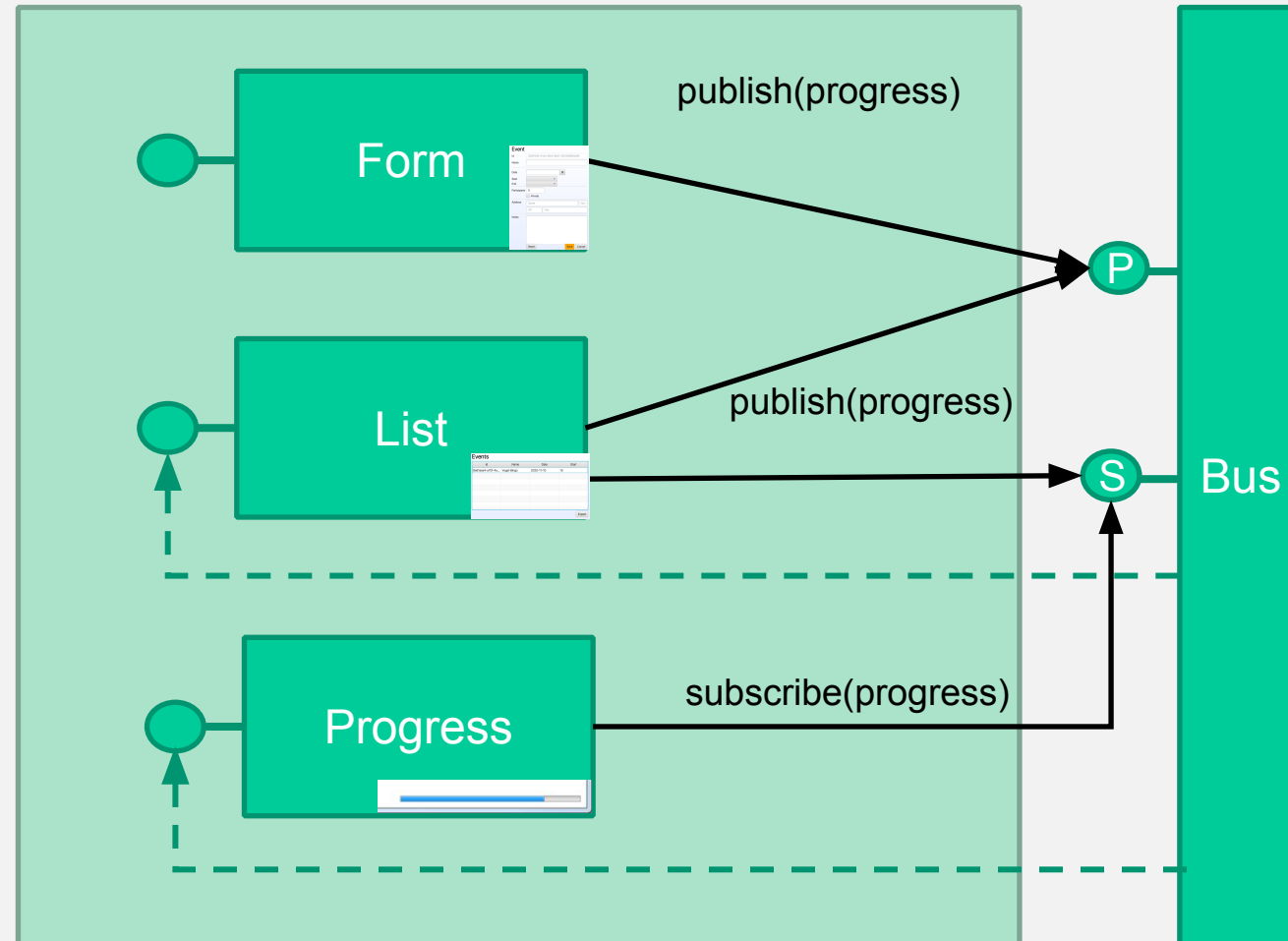
Der Main-Controller integriert die Komponenten.

- Eine direkte Kopplung von Sender und Empfänger ist nicht immer gewünscht.
 - Für wenige relativ eng gekoppelte Komponenten geeignet (Referenzen auf Sender und Empfänger müssen bekannt sein)
 - Die Komponenten müssen immer bekannt sein
- Anwendungsevents können durch unterschiedlichste UI-Komponenten ausgelöst werden und von unterschiedlichsten UI-Komponenten verarbeitet werden (n:m).
 - Zum Beispiel: Suche, Speichern, ...
 - Empfänger könnten nicht immer aktiv sein
- Aufwand und Code-Menge werden in größeren Hierarchien sehr schnell sehr umfangreich.

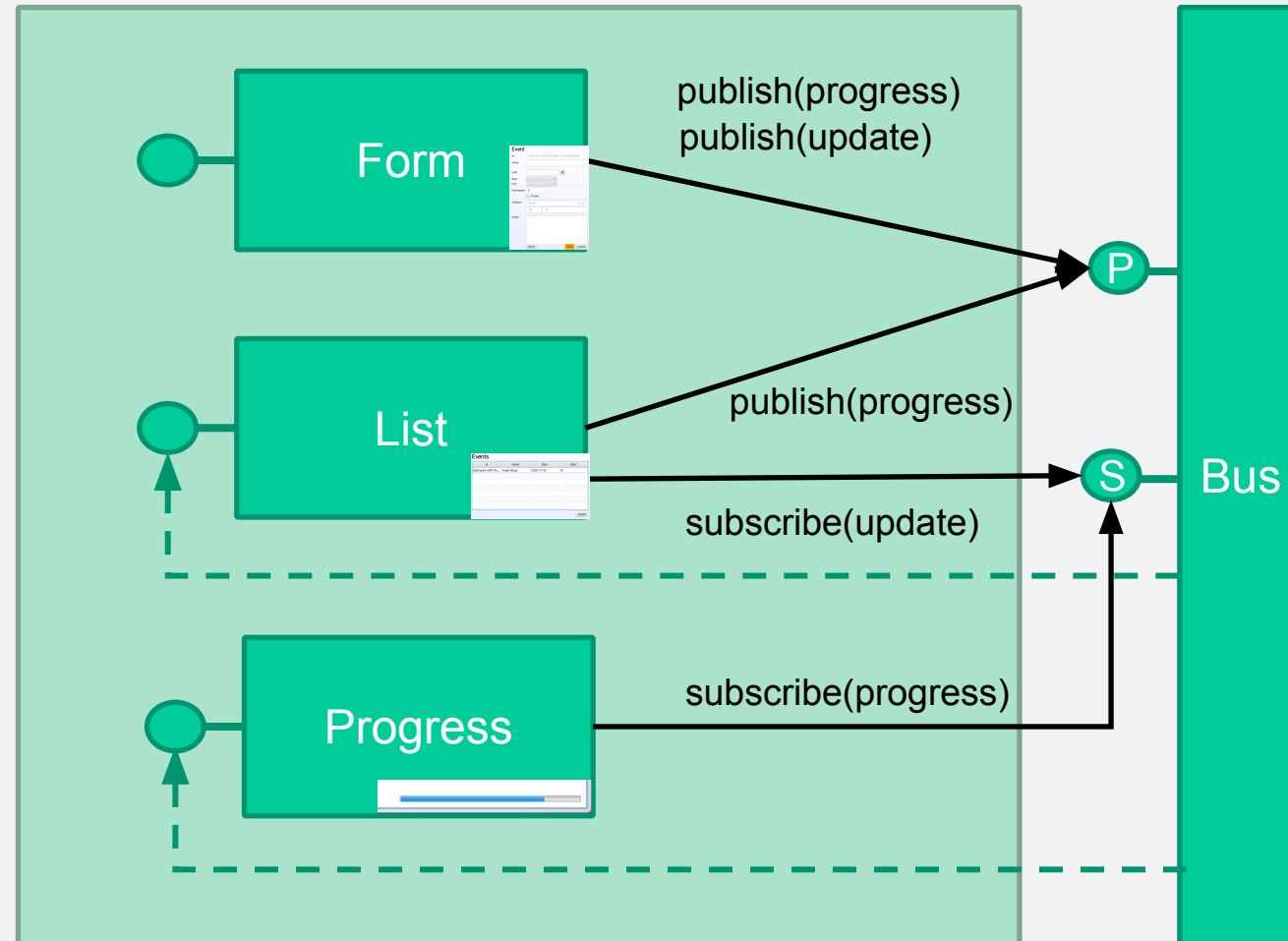


- A) View A (List) integriert View B (Form)
- B) Es gibt eine übergeordnete Ansicht die per Events benachrichtigt wird und an den untergeordneten View delegiert
- **C) Es gibt eine zentrale Stelle die Nachrichten empfängt und weiterleitet (EventBus)**

Lösung C: Ein zentraler Bus verbindet die Komponenten lose.



Lösung C: Ein zentraler Bus verbindet die Komponenten lose.



Bewertung der Varianten

- A – Direkte Integration
 - **Vorteil:** Gut Geeignet um größere Bauteile aus Einzelteilen zusammenzubauen.
 - **Nachteil:** Enge Kopplung, wenig Freiheitsgrade, für Teilfensterbereiche (Editor, Navigation) ungeeignet
- B – Übergeordnete UI-Komponente
 - **Vorteil:** Ansichten können unabhängig entwickelt und getestet werden
 - **Nachteil:** Aufwand – Kompliziert wenn sich Ansichten dynamisch ändern können (Beispiel: Dateienansicht in IntelliJ)
- C – Lose Kopplung über einen Event-Bus
 - **Vorteil:** Die übergeordnete UI-Komponente muss Ihre Komponenten nicht konkret kennen (Notwendig für unabhängige Fenster)
 - **Nachteil:** Eventfluss kann komplex werden und ist nicht mehr im Code eindeutig ersichtlich

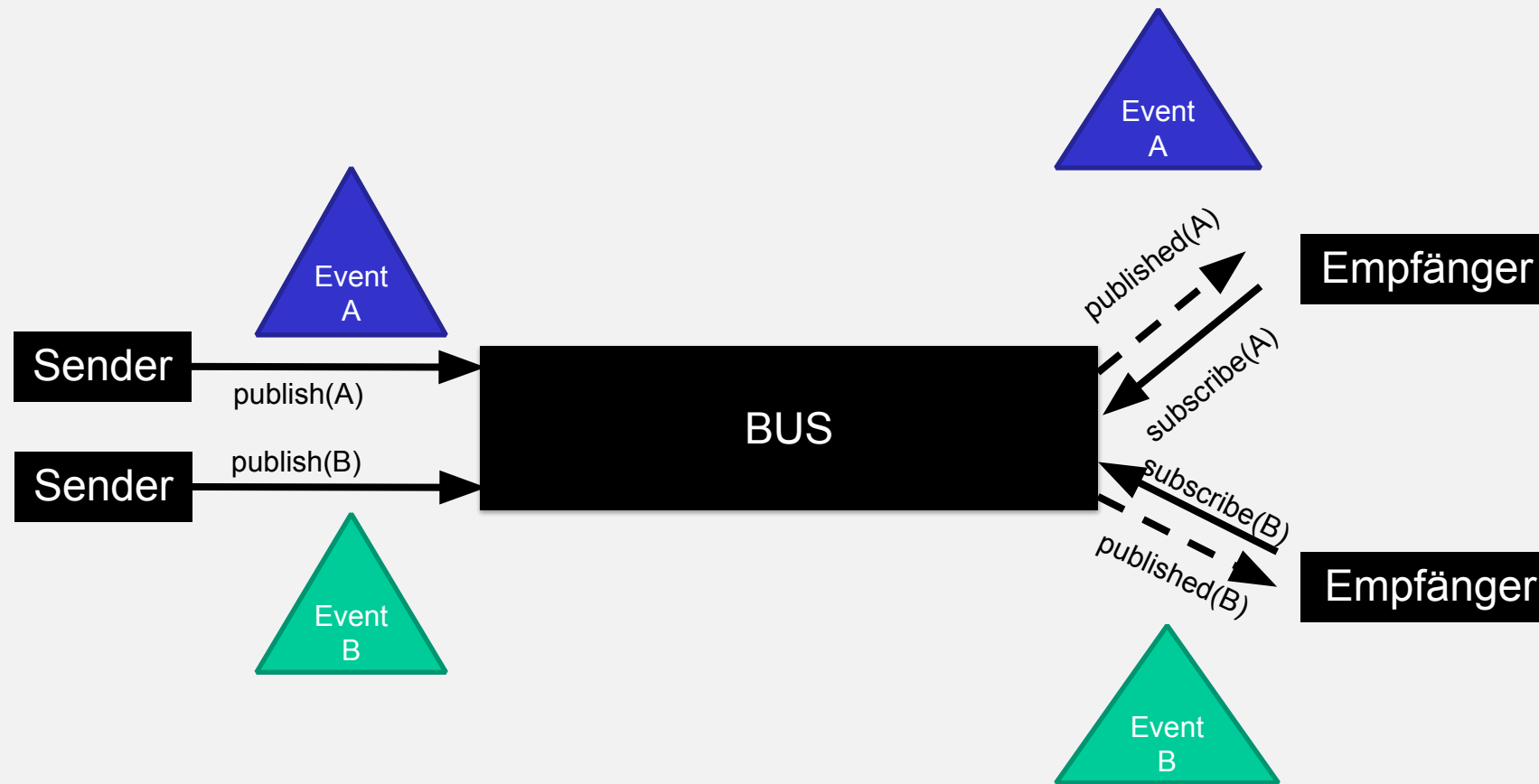
Kopplung

stark /
eng



schwach /
lose

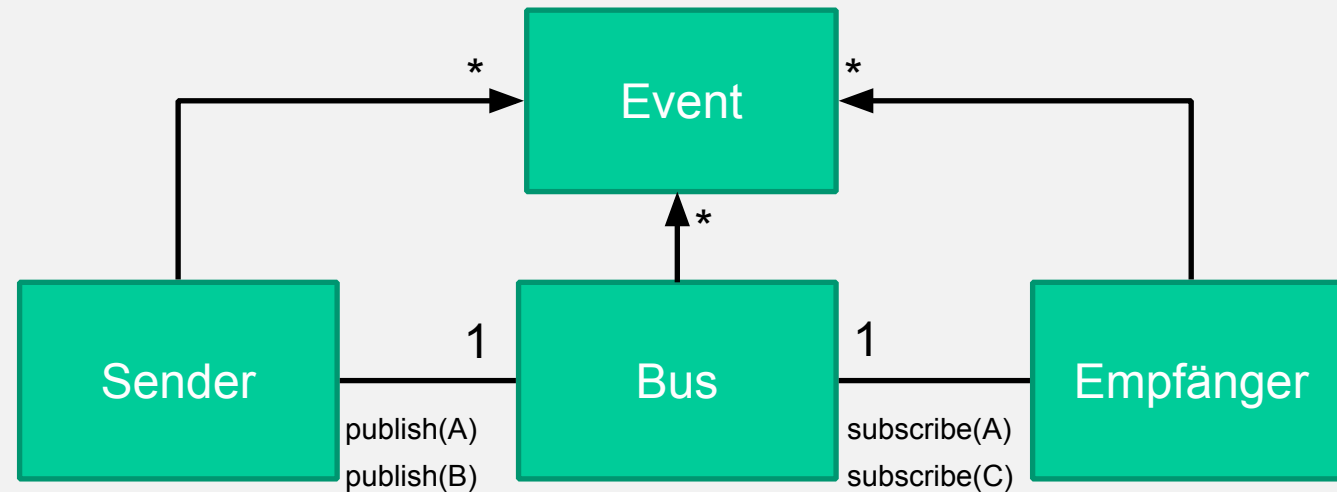
Publish() – Subscribe() - Muster



In welcher Reihenfolge erfolgen hier die Aufrufe? (5 Min)

Das Event verbindet Sender und Empfänger.

Der Bus entkoppelt den Sender vom Empfänger.



Der Bus

- Der Bus entkoppelt Sender und Empfänger
- Sender und Empfänger benötigen keine direkten Referenzen aufeinander
- Die Information steckt ausschließlich in den Event-Objekten – nicht in den Listener-Methoden
 - Das kann u.U. unübersichtlich werden

Die minimale Bus Schnittstelle



QA|WARE

```
public interface IEventBus {  
  
    /**  
     * Publish an event to all registered listeners.  
     * The event is dispatched by using event.getClass().  
     * @param event the event.  
     */  
    void publish(Object event);  
  
    /**  
     * Subscribe for a given type of event. The type is not polymorphic.  
     * You have to make a separate subscription for every concrete type.  
     * @param type the event type to subscribe.  
     * @param listener your listener, which will be called if a event happens.  
     */  
    void subscribe(Class type, IEventBusListener listener);  
}
```

Der Subscriber ist Bus-Listener

```
/**
 * The callback interface.
 */
public interface IEventBusListener {

    /**
     * Method will be called by the bus if the event type is
     * subscribed by this listener.
     * @param event the event.
     */
    void eventPublished(Object event);
}
```

Der Bus ist zentral erreichbar. (z.B. ein Singleton)



QA|WARE

```
// Singleton
public class SimpleEventBus implements IEventBus {

    private SimpleEventBus() {
        // singleton
    }

    public static IEventBus getBus() {
        return bus;
    }

    private static SimpleEventBus bus = new SimpleEventBus();
    ...
}
```

Der Bus verwaltet n-Subscriptions in einer Hash-Map

```
class SimpleEventBus implements IEventBus {  
  
    private Map<Class, List<IEventBusListener>> subscriptions =  
  
        new HashMap<Class, List<IEventBusListener>>();  
    ...  
}
```

Publish()



QA|WARE

```
@Override
public void publish(Object event) {
    List<IEventBusListener> subscriptionsForType = subscriptions.get(event.getClass());
    if (subscriptionsForType != null) {
        for (IEventBusListener l : subscriptionsForType) {
            l.eventPublished(event);
        }
    }
}
```

Subscribe()



QA|WARE

@Override

```
public void subscribe(Class type, IEventBusListener listener) {  
    List<IEventBusListener> subscriptionsForType = subscriptions.get(type);  
    if (subscriptionsForType == null) {  
        subscriptionsForType = new ArrayList<IEventBusListener>();  
        subscriptions.put(type, subscriptionsForType);  
    }  
    subscriptionsForType.add(listener);  
}
```

Client Code



QA|WARE

```
// publisher code
IEventBus bus = SimpleEventBus.getBus();
bus.publish(new EventObject(„Hello World"));
```

```
// subscriber code
IEventBus bus = SimpleEventBus.getBus();
bus.subscribe(EventObject.class, new IEventBus.IEventBusListener() {
    @Override
    public void eventPublished(Object e) {
        System.out.println(e);
    }
});
```

Demo 2: Event-Bus



QA|WARE

A screenshot of an IDE window showing a Java project. The left sidebar displays the project structure with folders like 'src', 'main', and 'resources'. The main editor area shows the code for 'EventsMainPresenter.java'. The code includes imports for 'Logger', 'LoggerFactory', 'FXML', 'TableView', 'EventFormPresenter', and 'EventsMainModel'. It defines a static logger, initializes an event table and controller, and implements a 'saveEvent' method that logs and adds events to a model. The 'initialize' method sets up the save button's action and registers a listener. The code is as follows:

```
private static final Logger LOGGER = LoggerFactory.getLogger(EventsMainPresenter.class);

@FXML
private TableView<EventModel> eventsTable;
@FXML
private EventFormPresenter eventFormController;

private EventsMainModel model = new EventsMainModel();

private void saveEvent(EventModel event) {
    LOGGER.info("saveEvent");
    //add the current event to the list
    model.addEvent(event);
    //set up a new one
    eventFormController.setEvent(new EventModel());
}

@Override
public void initialize(URL location, ResourceBundle resources) {
    //Aufgabe 2 - don't do this!
    /*
    eventFormController.saveButton.setOnAction(e -> {
        System.out.println("### Parent (main) - received save from form");
    });*/

    //register save listener - component interaction
    eventFormController.addSaveListener(event -> {
        this.saveEvent(event);
    });

    //ui bindings - display events in table
    eventsTable.itemsProperty().bind(model.eventsProperty());
}
```




QAWARE

Event-Bus - Fragen

Fragen



QA|WARE

- Nennen Sie die zwei Schnittstellen des Event-Bus.
- Beschreiben Sie die Schnittstelle des EventBusListener in Java.
- Wie werden beim Event-Bus Daten von A nach B übertragen?
- Malen Sie das Publish-Subscribe-Muster.