

Der Event-Bus

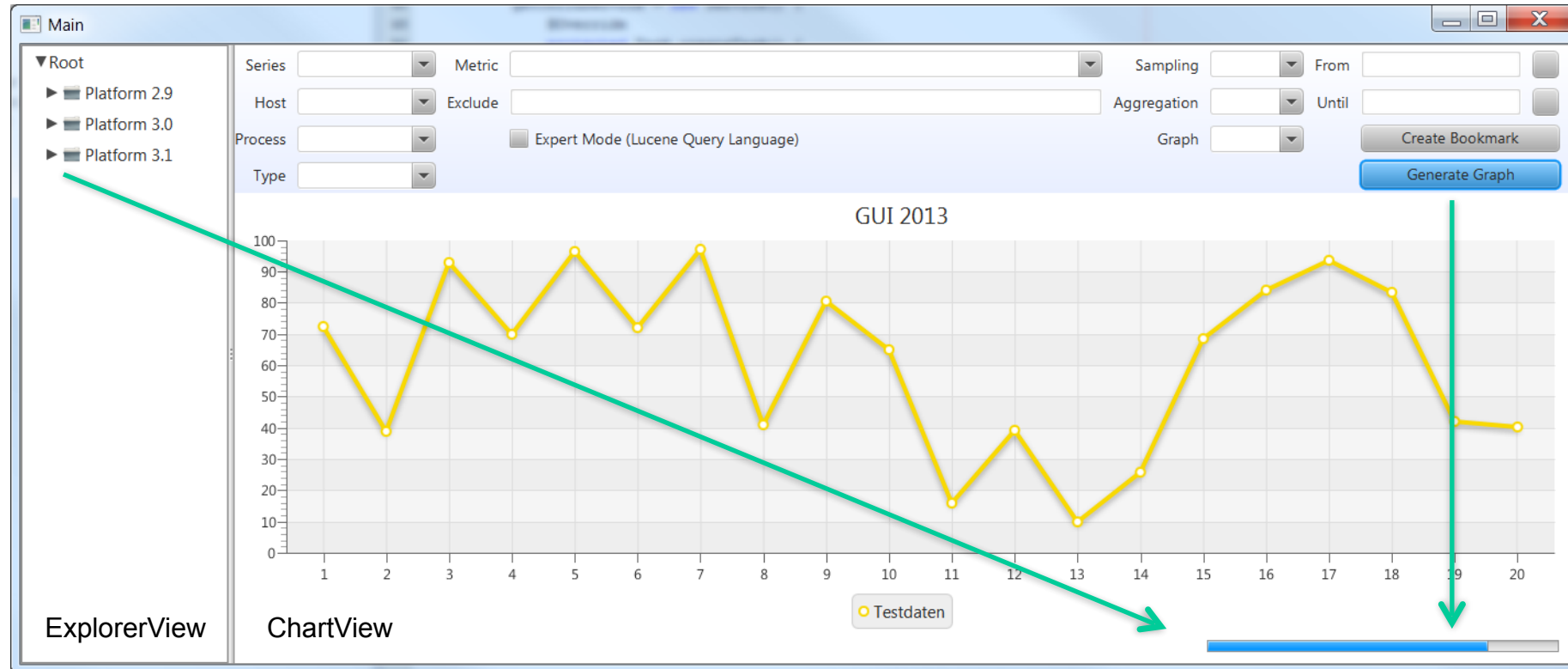
Lose Kopplung von wiederverwendbaren Bauteilen in der UI
Entwicklung



Das Problem

- Oft ist eine direkte Kopplung von Sender und Empfänger nicht möglich (oder sehr umständlich)
 - MVC ist für ein oder wenige relativ eng gekoppelte Komponenten geeignet (Referenzen auf Sender und Empfänger müssen bekannt sein)
- Anwendungsevents können durch unterschiedlichste UI-Komponenten ausgelöst werden
 - Beispiele:
 - Suche ...
 - Speichern ...
 - Beenden ...
 - Fortschritt

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

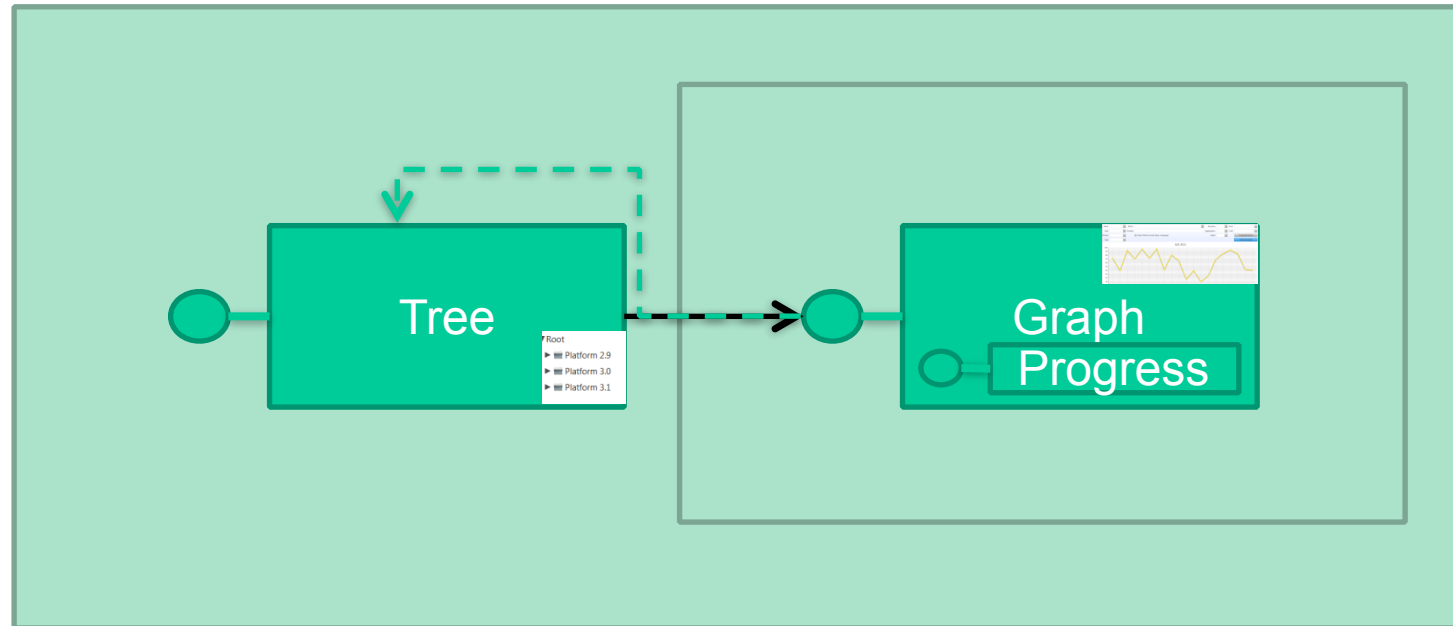


Drei Lösungen

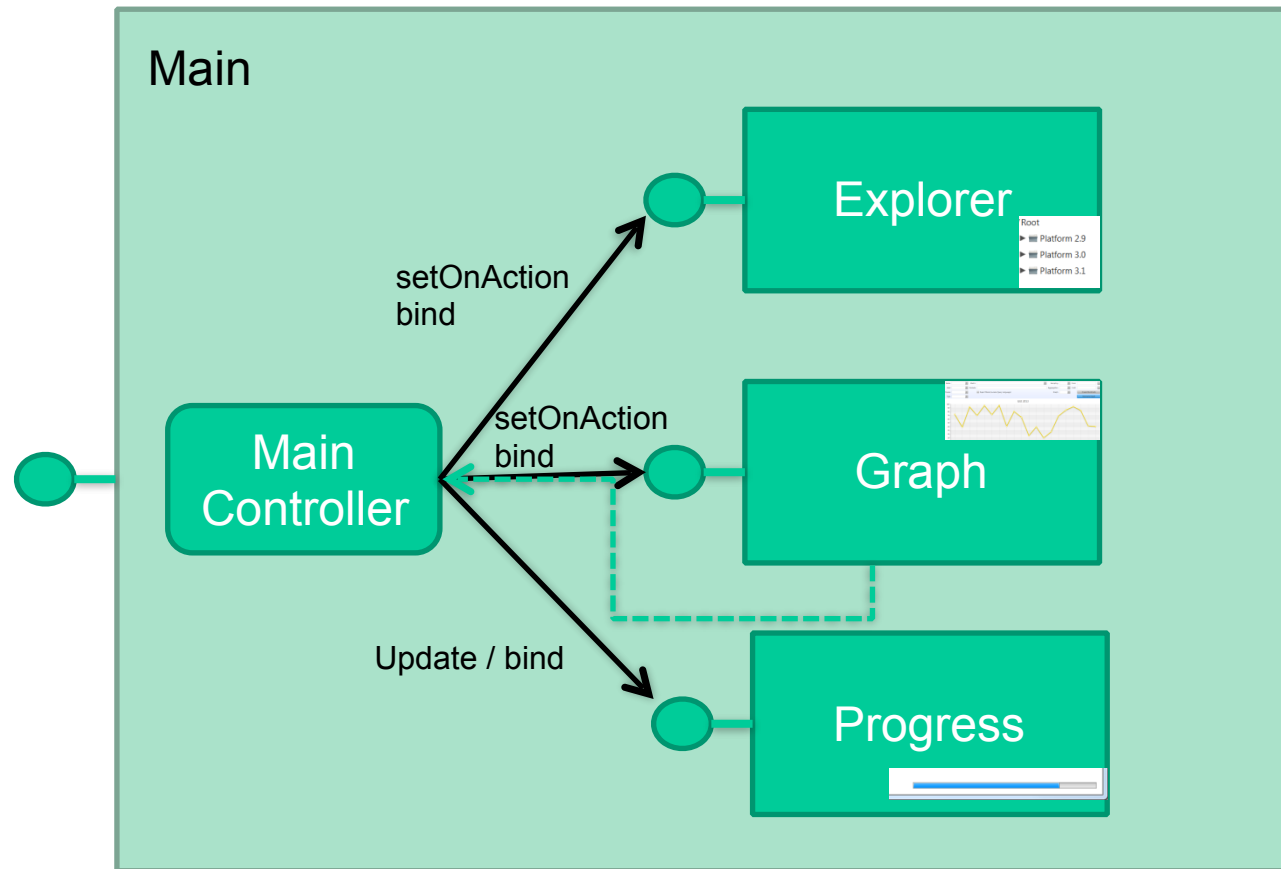
- A) View A (Tree) integriert View B (Graph)
 - Verwendung des Tree-View ohne den Graph-View ist nicht möglich
- B) Es gibt eine übergeordnete Ansicht die per Events benachrichtigt wird und an den untergeordneten View delegiert
 - Codemenge und Aufwand steigt
- C) Es gibt eine zentrale Stelle die Nachrichten empfängt und weiterleitet

Lösung A: Tree-View integriert die Graph-View

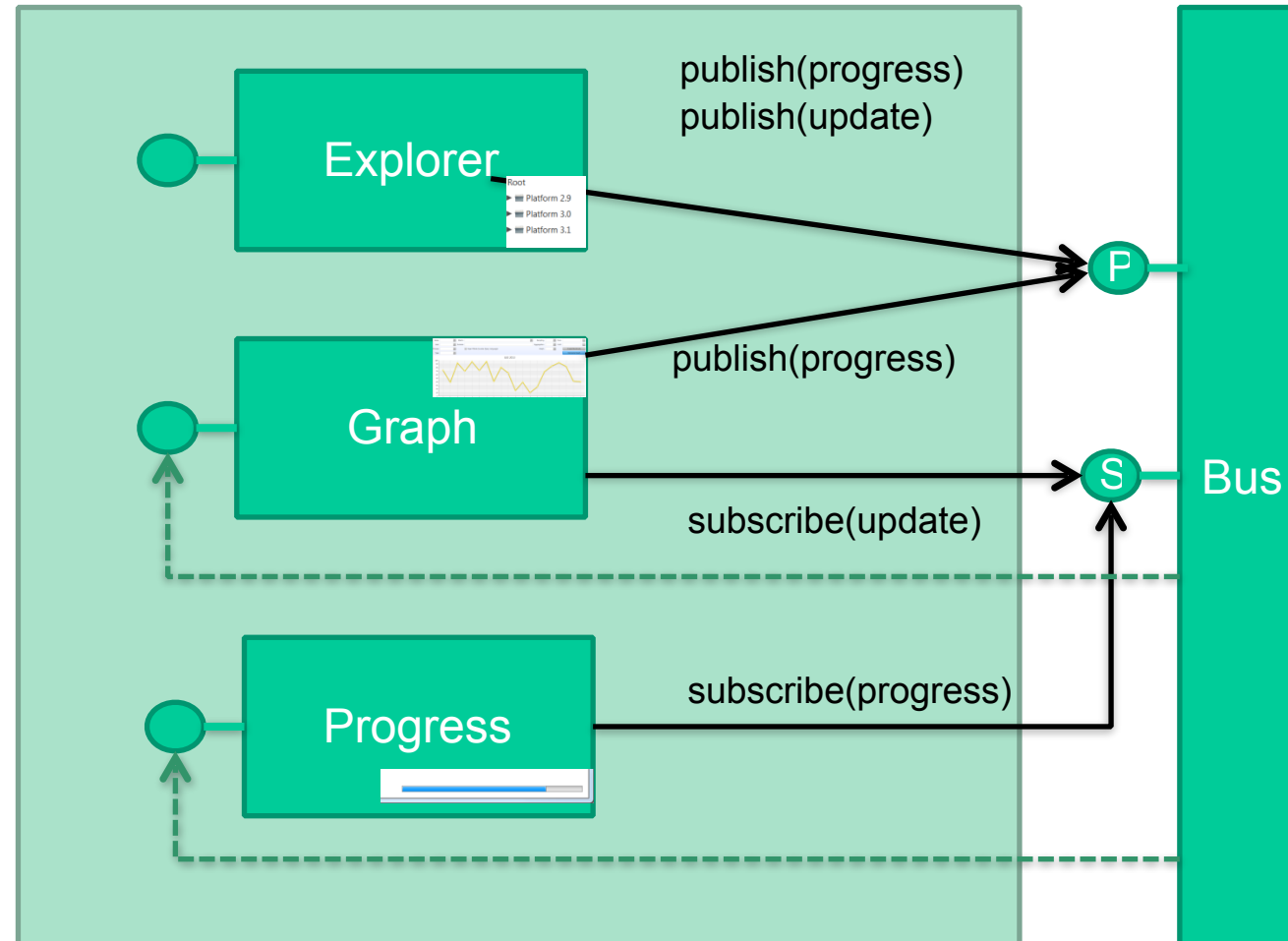
Tree-Controller hält eine Referenz auf den Graph-Controller



Lösung B: Übergeordneter „Main-View“



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: Explorer / Editor in Netbeans/Eclipse)
- 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

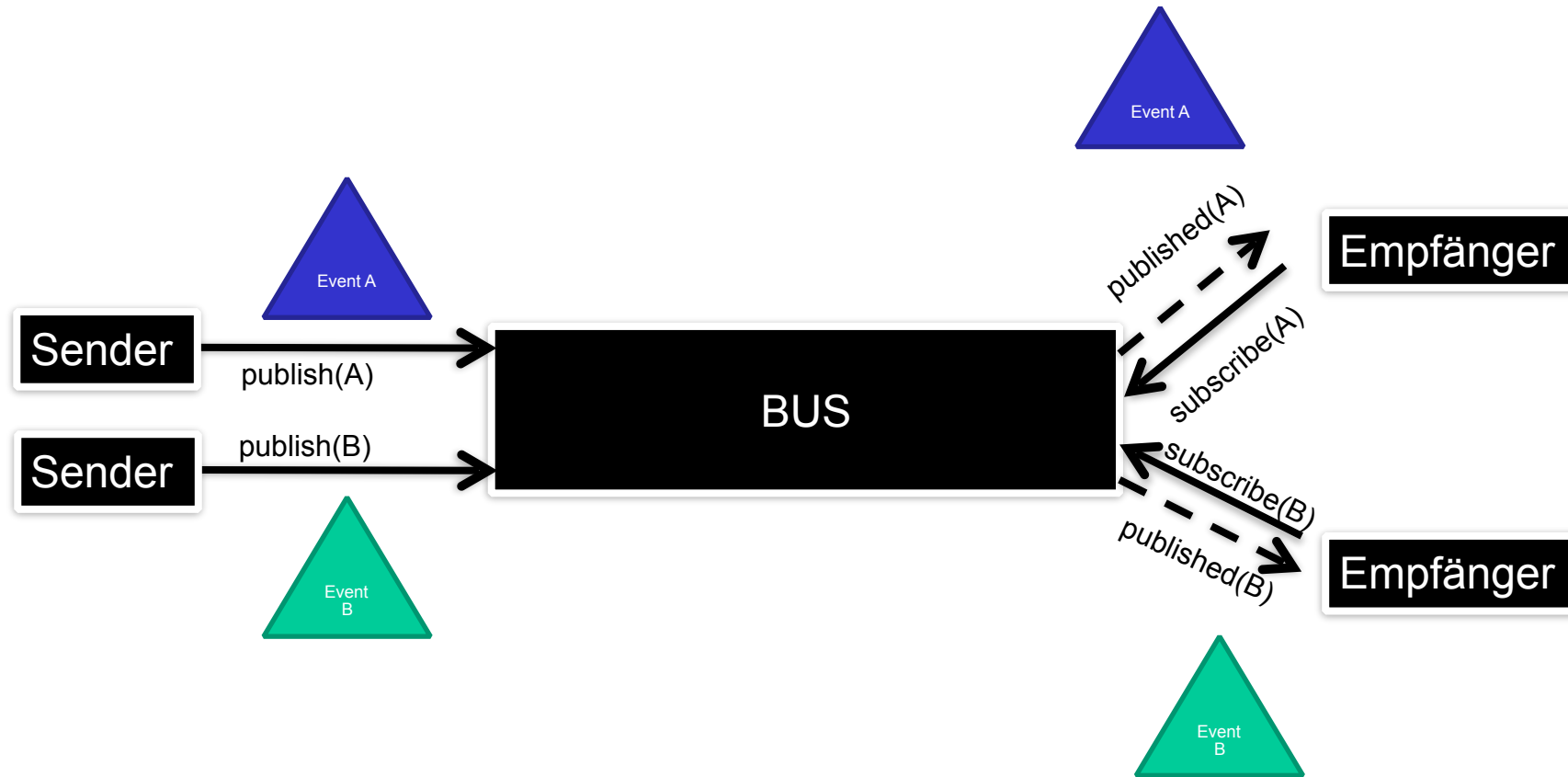
Kopplung

stark /
eng

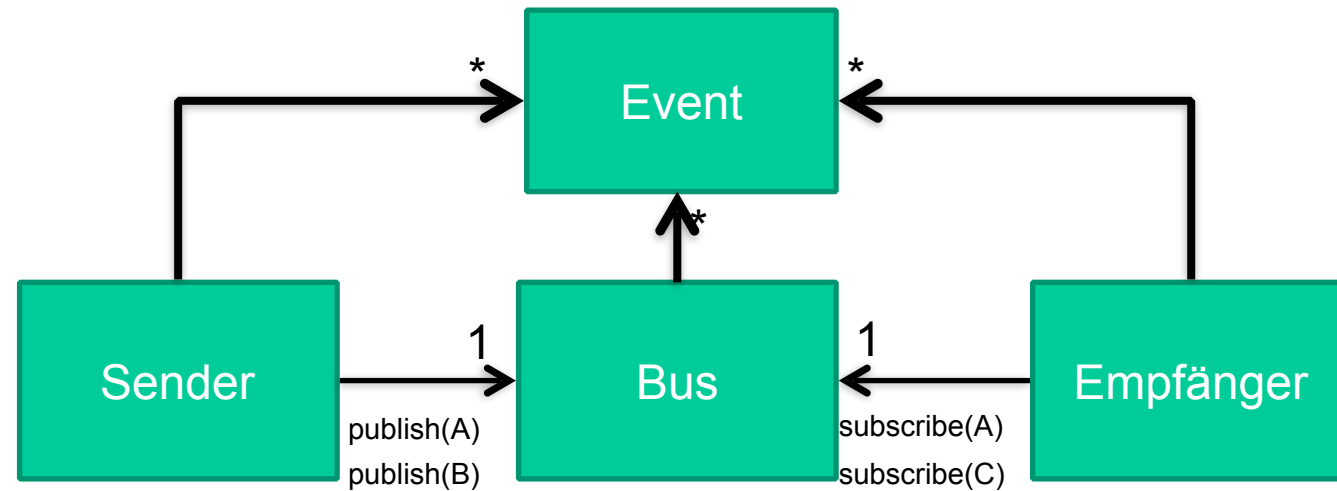


schwach /
lose

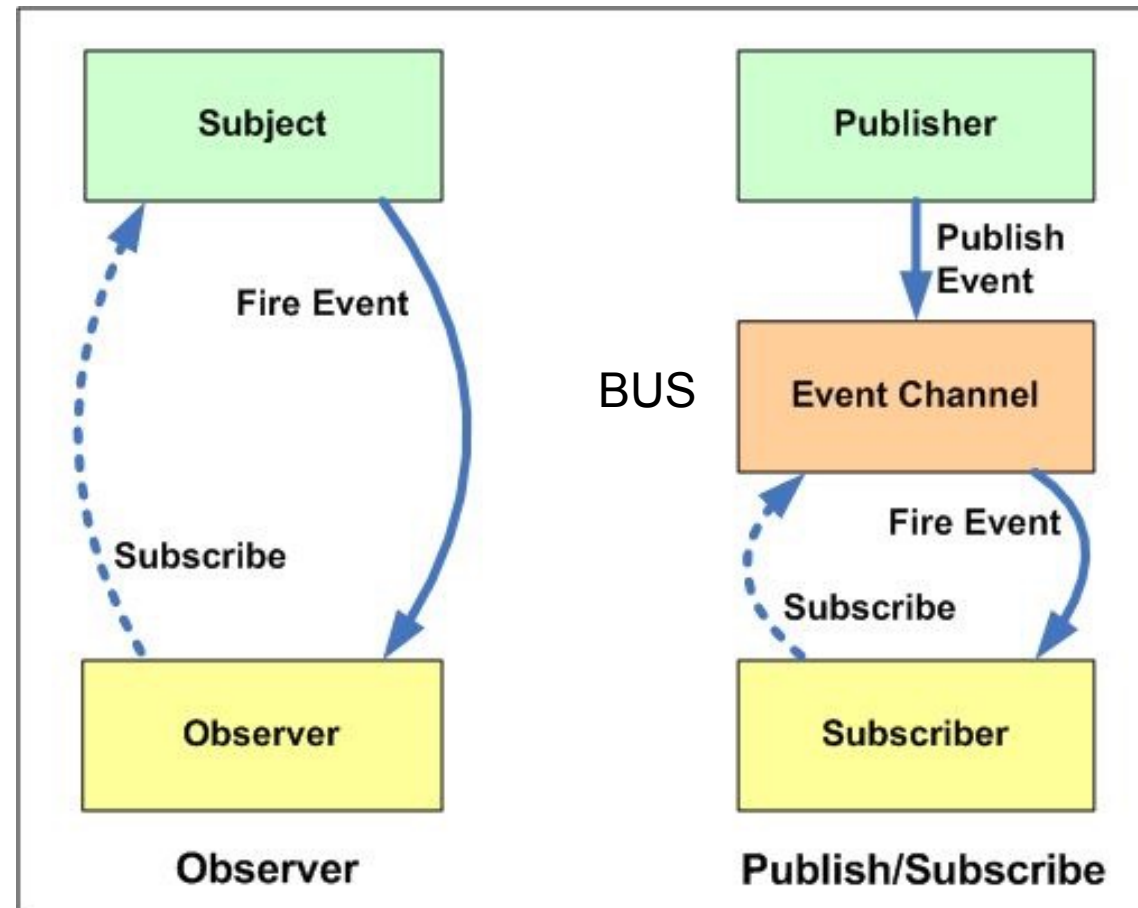
Publish() – Subscribe() - Muster



Der Event verbindet Sender und Empfänger.
Der Bus entkoppelt den Sender vom Empfänger.



Publish() – Subscribe() vs. Observer



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

```
public interface IEventBus {  
  
    /**  
     * Publish a event to all registered listeners.  
     * The event is dispatched by using event.getClass().  
     * @param event the event.  
     */  
    void publish(EventObject 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.

(Singleton, CDI, Spring-Bean, java.util.ServiceLoader, OSGi)

```
// 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()

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

Subscribe()

```
@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

```
// 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);
    }
});
```



Geht das syntaktisch noch einfacher?

Subscriber per Annotation

```
@EventSubscriber(eventClass = EventObject.class)
public void x(EventObject e) {
    // called if a event happens
}
```

Die Annotation

```
@Target(value= ElementType.METHOD)
@Retention(value= RetentionPolicy.RUNTIME)
@interface EventSubscriber {
    Class eventClass();
}
```

```


class AnnotationProcessor {

    public static void process(final Object object) {
        Class clazz = object.getClass();
        Method[] methods = clazz.getMethods();
        for (int i = 0; i < methods.length; i++) {
            final Method method = methods[i];
            if (method.isAnnotationPresent(EventSubscriber.class)) {
                EventSubscriber s = method.getAnnotation(EventSubscriber.class);
                SimpleEventBus.getBus().subscribe(s.eventClass(), new IEventBusListener()
                    @Override
                    public void eventPublished(Object event) {
                        try {
                            method.invoke(object, event);
                        } catch (Exception e) {
                            throw new IllegalStateException(e);
                        }
                    }
                });
            }
        }
    }
}

```

Nutzung der Annotation

```
initialize() {  
    IEventBus bus = SimpleEventBus.getBus();  
    AnnotationProcessor.process(this); // ab hier werden Events empfangen  
}  
  
@EventSubscriber(eventClass = EventObject.class)  
public void x(EventObject e) {  
    System.out.println("x: " + e);  
}
```



Welche Vorteile bietet die Annotation?

- Mehrere Methoden sind für unterschiedliche Eventtypen in einer einzigen Klasse möglich
 - Vermeidet switch()-Statement wenn eine einzige eventPublished() Methode implementiert wird
 - Vermeidet Hilfsklassen / anonyme Klassen wenn unterschiedliche Methoden verwendet werden sollen