

*Desktop-Muster
und
Datenbindung für Swing*

JGoodies :: Karsten Lentzsch



Ziel

Ansätze kennenlernen
wie man die Präsentationslogik organisieren
und Fachdaten mit der GUI verbinden kann

Vorstellung

- Ich baue Swing-Anwendungen, die viele Leute elegant finden
- arbeite seit 1990 mit Objekten
- helfe Anderen über und unter der Haube
- biete Bibliotheken, die Swing ergänzen
- biete Swing-Beispiele zu Architekturen
- und schreibe über Desktop-Themen

Gliederung

- Einleitung
- Separated Presentation & Autonomous View
- MVP, MVC und Presentation Model
- Einzelwerte Synchronisieren
- Erfahrungsbericht

Swing-Bausteine



Swing-Bausteine



Fragen

- Wie hängen MVC und Swing zusammen?
- Wie gliedere ich meine Anwendung?
- Wie teile ich Modelle auf?
- Wie baue ich einen View?
- Wer sollte Ereignisse behandeln?
- Brauche ich einen Controller?

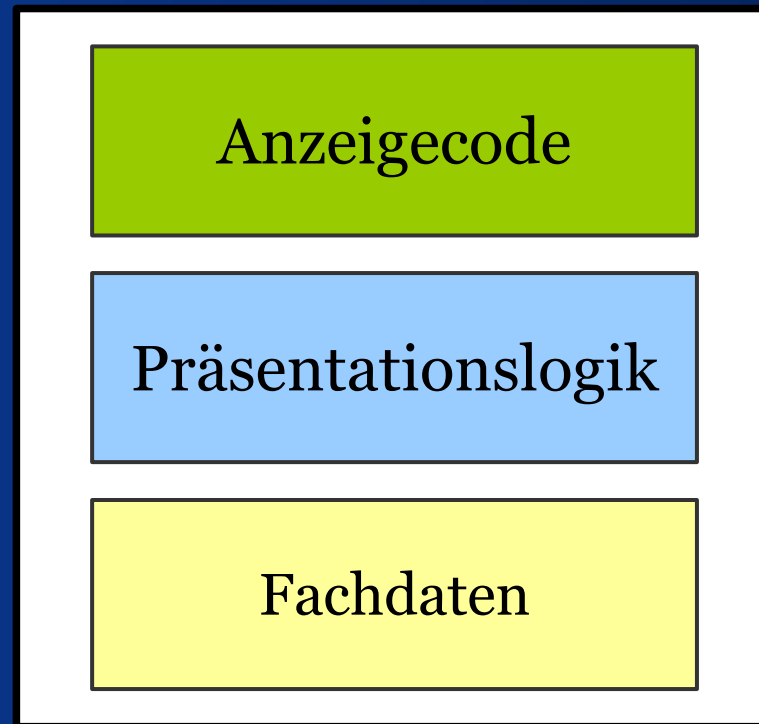
Dringend Empfohlen!

1. Folge **Separated Presentation!**
2. Lies “Organizing Presentation Logic” aus Fowlers “Further P of EAA”
3. Studiere **MVP** und **Presentation Model**
4. Kenne **Observer**
5. Wenn nötig zerlege **Autonomous View** mittels **MVP** bzw. **Presentation Model**

I - Grundlagen

Separated Presentation & Autonomous View

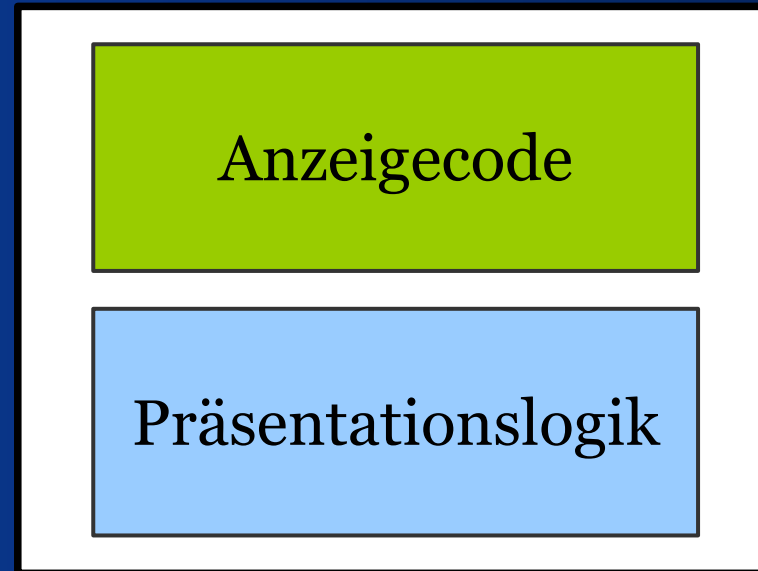
So Nicht!



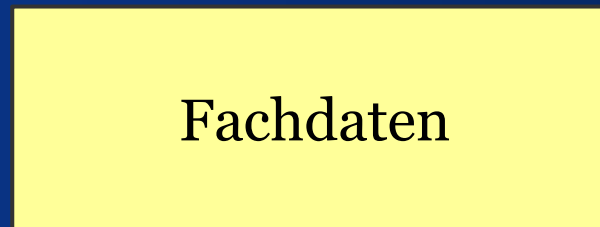
Fachschicht abtrennen

- Fachlogik enthält keinen GUI-Code
- Präsentation handhabt die Oberfläche
- Vorteile:
 - Die Teile sind leichter zu verstehen
 - Die Teile sind leichter zu ändern
- Daumenregel für Fachdaten:
Brauche ich den Code auch ohne GUI?

Separated Presentation



Präsentationsschicht

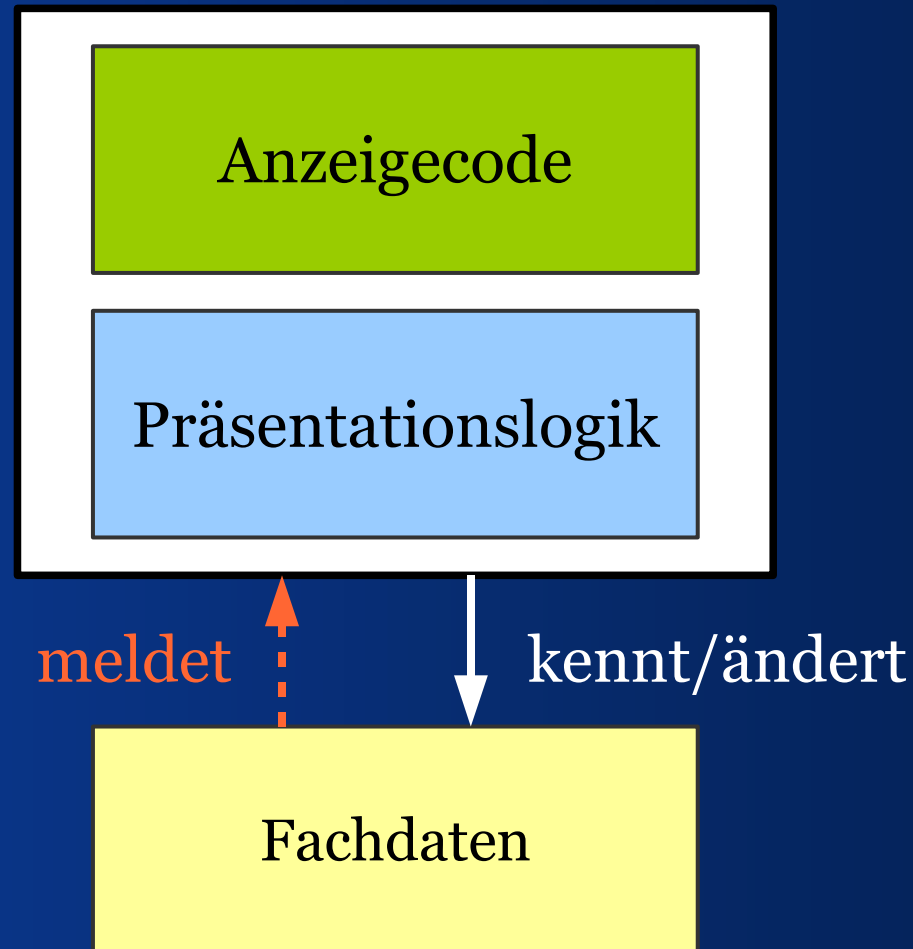


Fachschicht

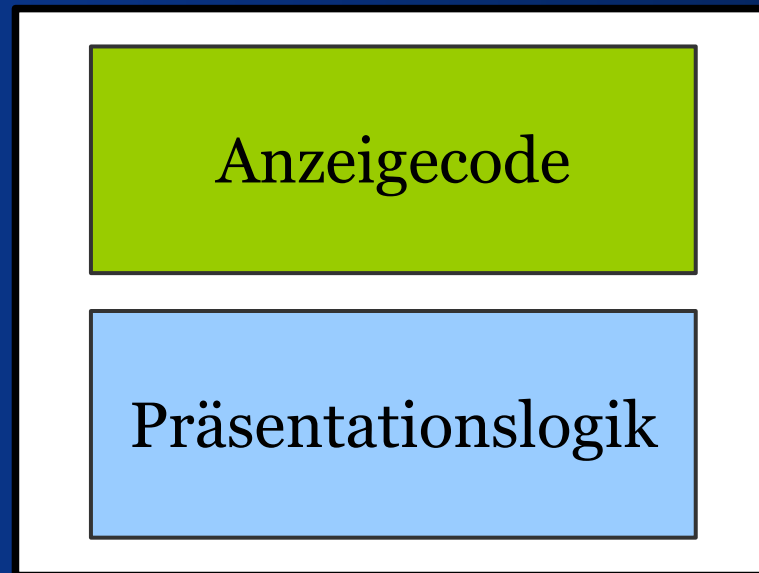
Präsentation entkoppeln

- Fachlogik referenziert keinen GUI-Code
- Präsentation kennt und ändert die Fachlogik
- Vorteile:
 - Reduziert Komplexität
 - Erleichtert **mehrere** Präsentationen **einer** Fachlogik

Sep.Presentation mittels Observer



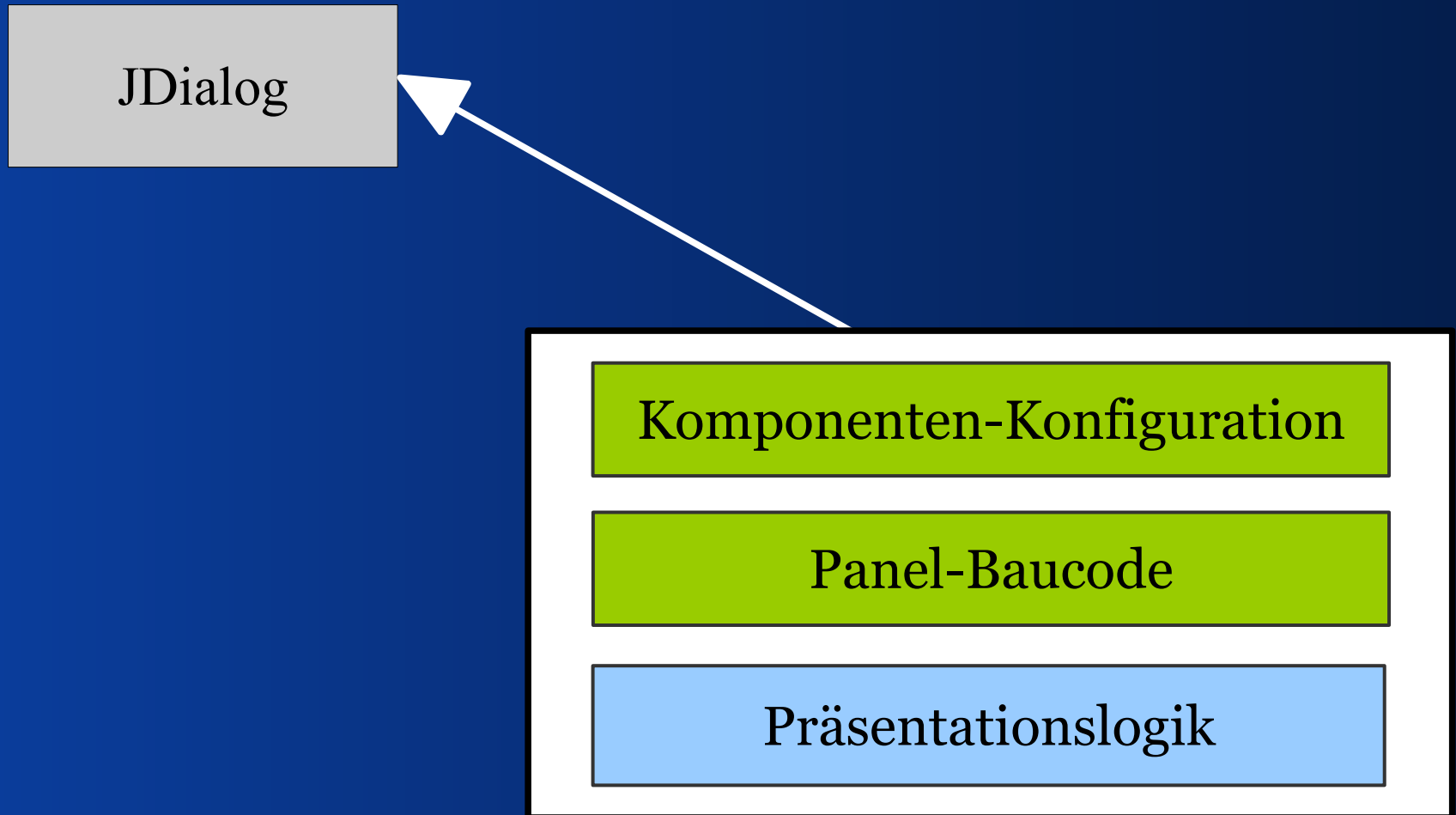
Autonomous View



Autonomous View

- Üblich: eine Klasse pro Fenster.
- Häufig einer Unterklasse von JDialog, JFrame, JPanel – ist aber meist unnötig.
- Muster ist geeignet für kleine Fenster.
- Bei komplexeren Aufgaben lohnt es, den Code aufzutrennen.

Autonomous View: Details



Tipps

- Baue Dialoge, Frames, Panel; erweitere sie nur, wenn nötig.
- Setze große Fenster aus Paneln zusammen.
 - in einfachen Fällen mittels Baumethoden `#buildMainPanel`, `#buildButtonBar`, etc.
 - ansonsten aus Paneln und Unterpaneln.
- Erwäge, die Präsentationslogik herauszutrennen.

Wann Autonomous View Teilen?

- Wenn Du die Präsentationslogik testest.
- Wenn Du den Code nicht mehr überblickst, etwa weil er nicht in die Outline passt.
- Wenn Du Code mit Kollegen tauscht.
- Wenn Du Teile wieder verwendest.

Präsentationslogik abgetrennt

Anzeigecode

Präsentationslogik

Fachdaten

Vorteile dieser Trennung I

- Erleichtert das Testen (nennt Fowler).
- GUI-Schicht ist einfach/dumm und deshalb einfach zu bauen, zu verstehen, zu warten.
- Mehr Team-Mitglieder trauen sich an GUI.
- GUI-Code kann Syntaxmustern folgen.
- GUI kann mit Werkzeug gebaut werden.

Vorteile dieser Trennung II

- Die schwierige Logik wird überschaubarer.
- Die Trennung sortiert das Denken.
- Erleichtert die Synchronisation im Team.
- Ermöglicht “verbotene Zonen”
 - Für Team-Mitglieder
 - Etwa vor einem neuen Release

Nachteile dieser Trennung

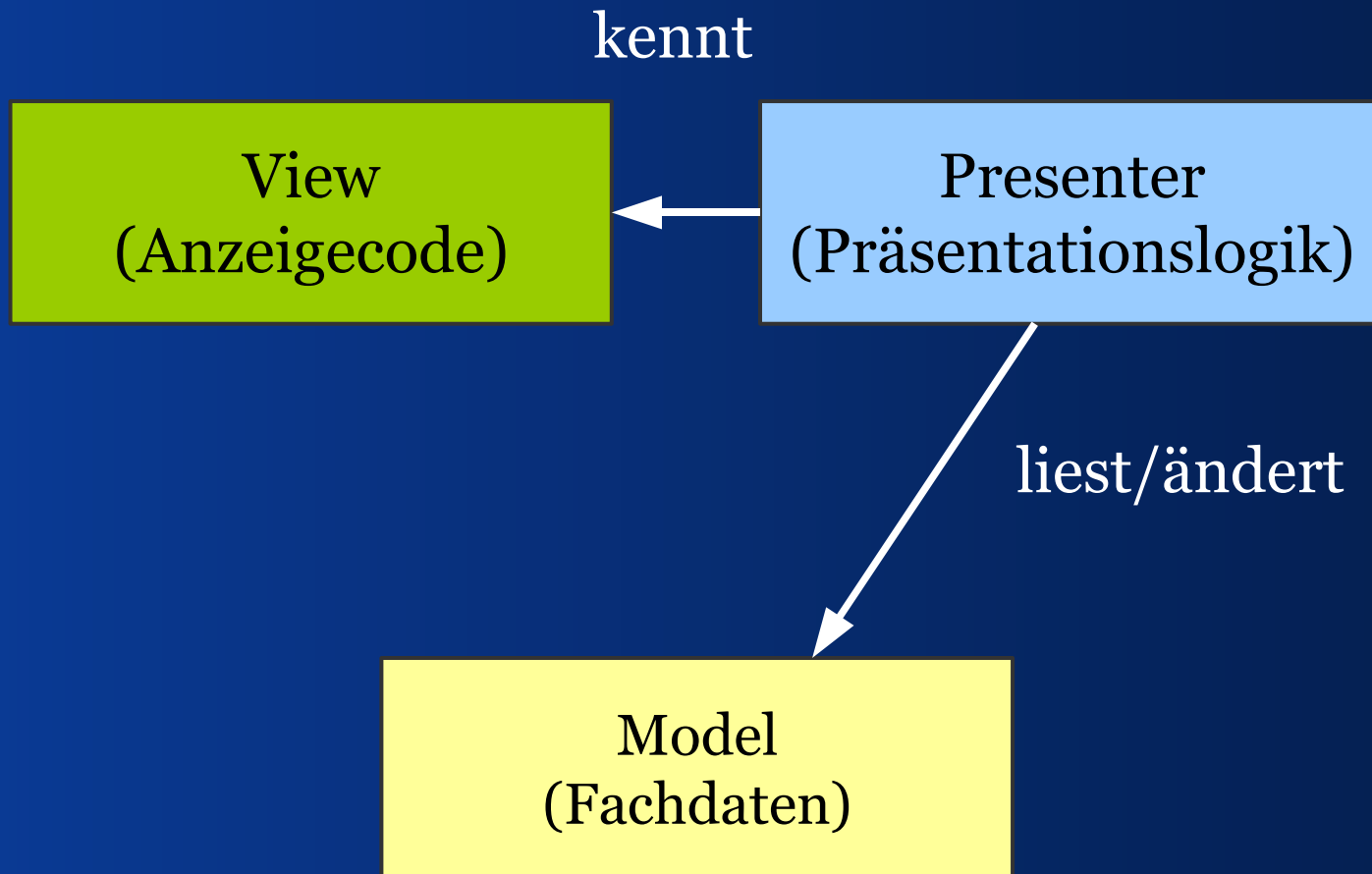
- Ist Mehrarbeit.
- Erfordert, mehrere Klassen im Zusammenhang zu betrachten.

Diese Nachteile werden meistens aufgewogen.

II - Autonomous View Teilen

MVP, MVC und Presentation Model

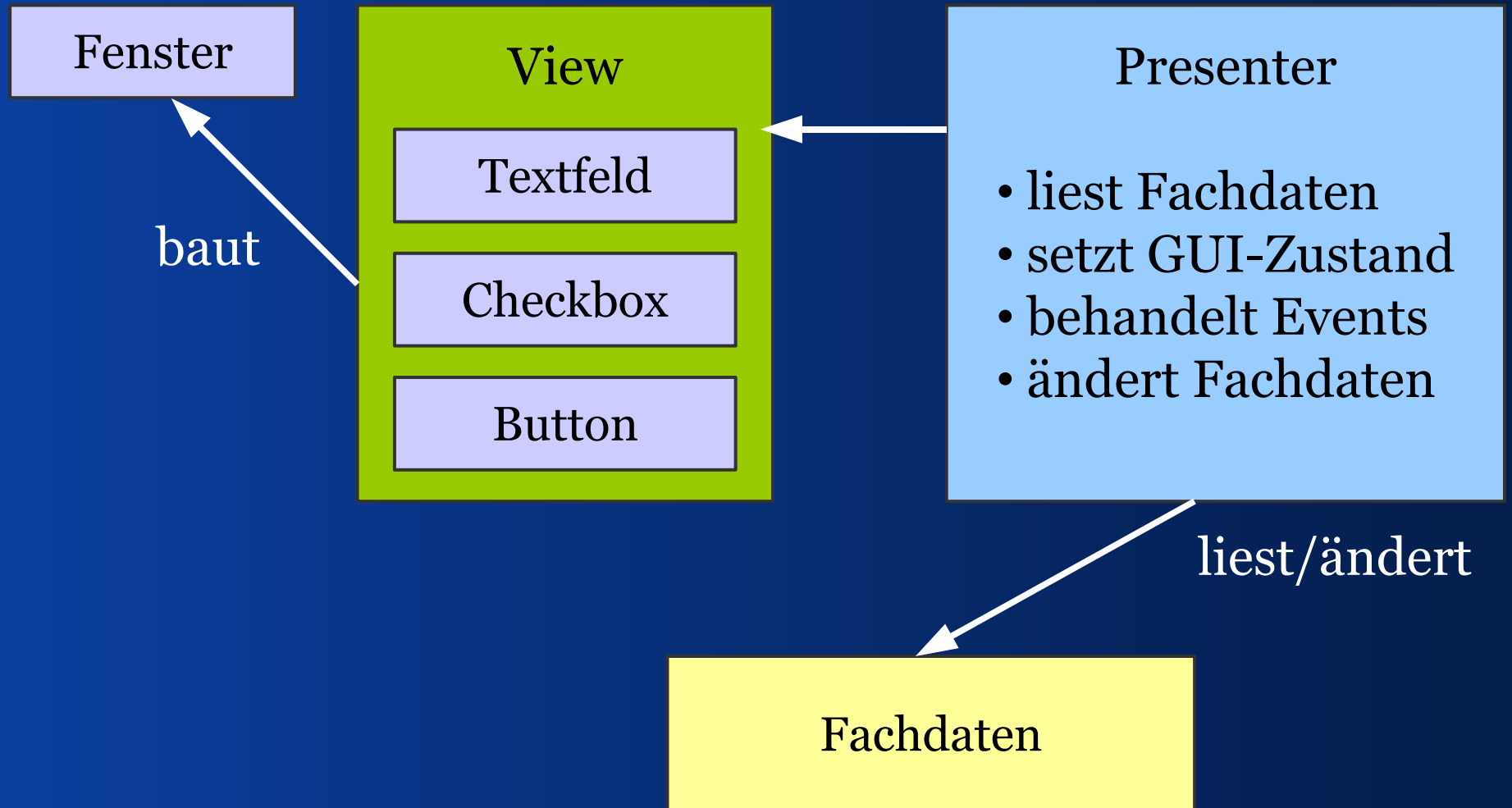
Model-View-Presenter (MVP)



MVP

- Der View
 - enthält den GUI-Zustand,
z. B. JTextField mit Text und Enablement
- Der Presenter
 - liest Fachdaten und schreibt sie in die
Komponenten des Views
 - reagiert auf GUI-Events und ändert
den GUI-Zustand im View
 - ändert Fachdaten anhand der GUI-Daten

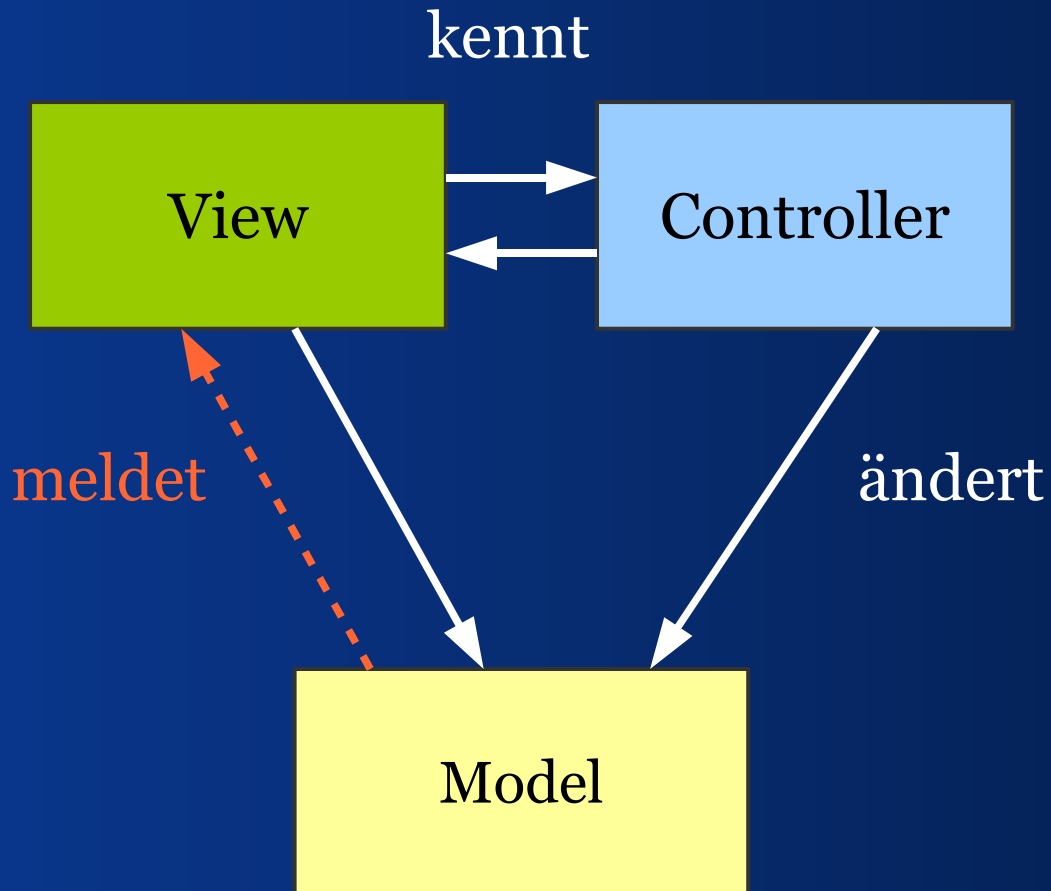
MVP



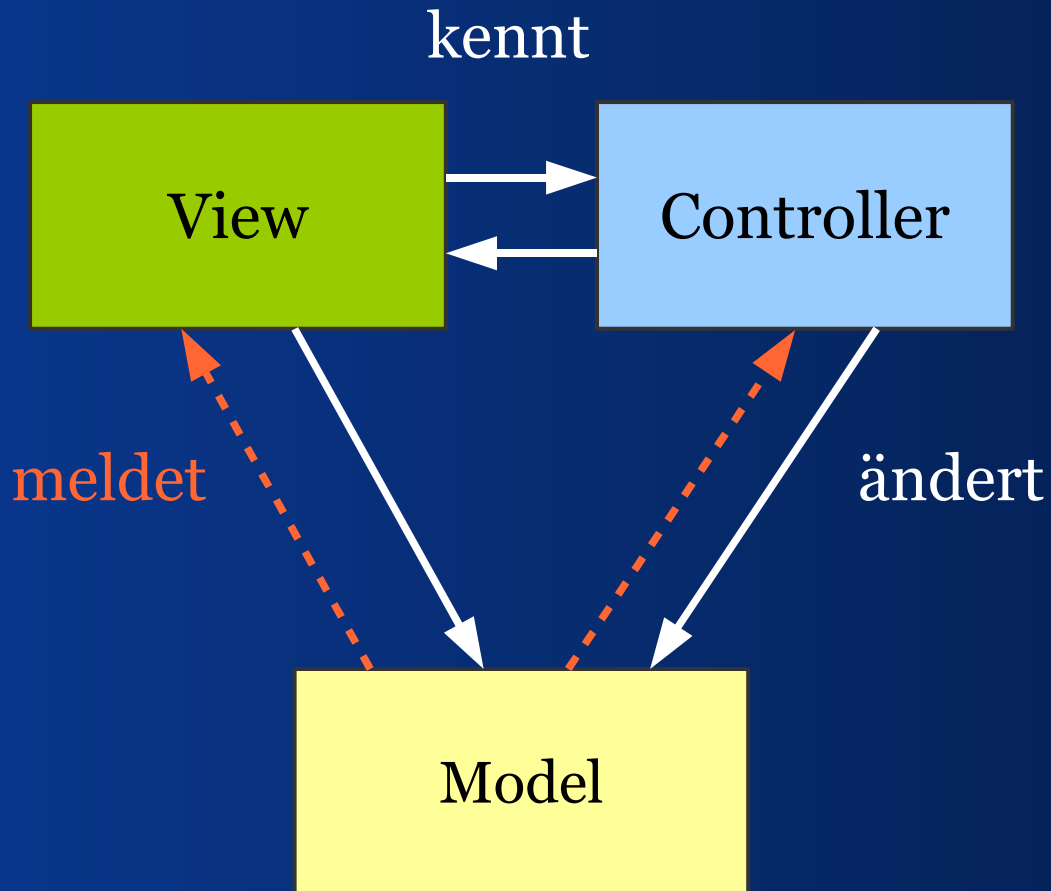
Einschub: MVP vs. MVC

Unterschiede und die Swing-MVC-Variante

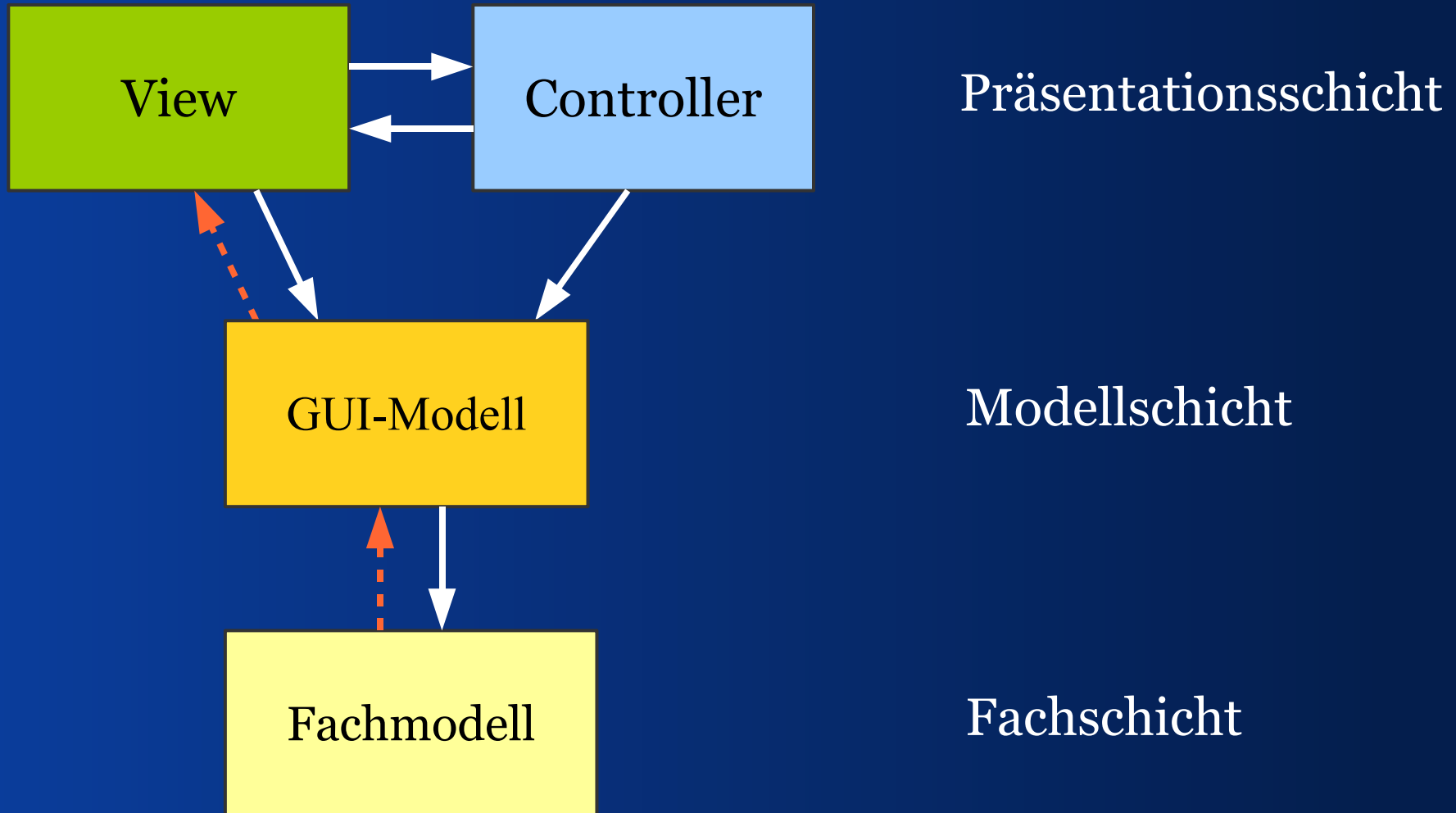
MVC



MVC



MVC mit Modellschicht

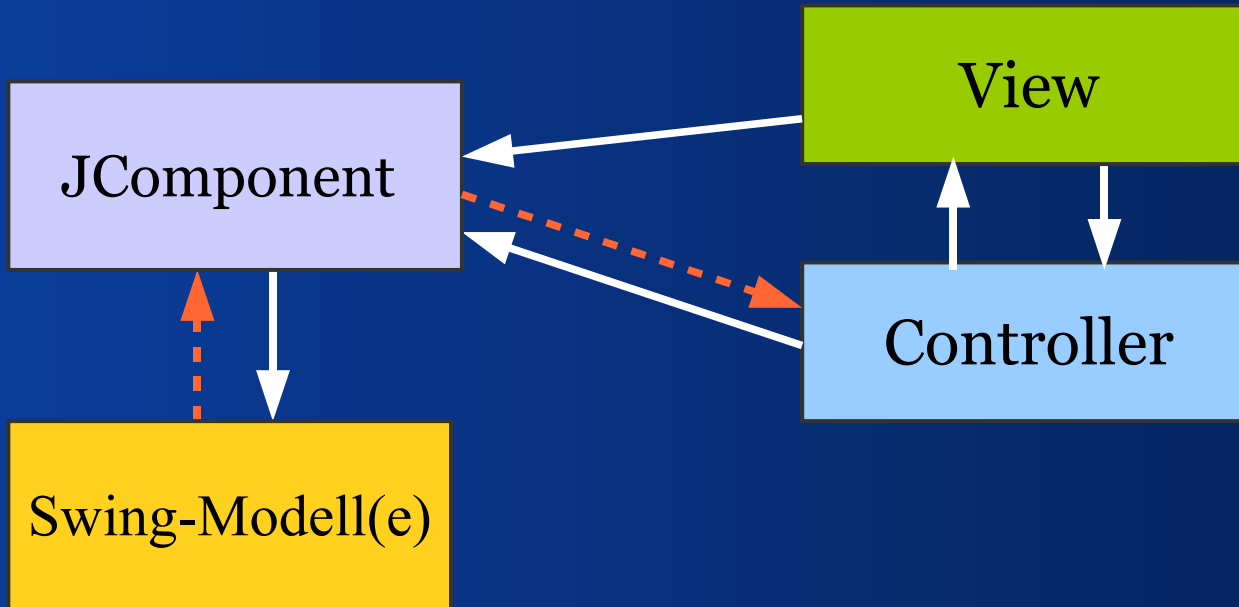


Look&Feel abtrennen

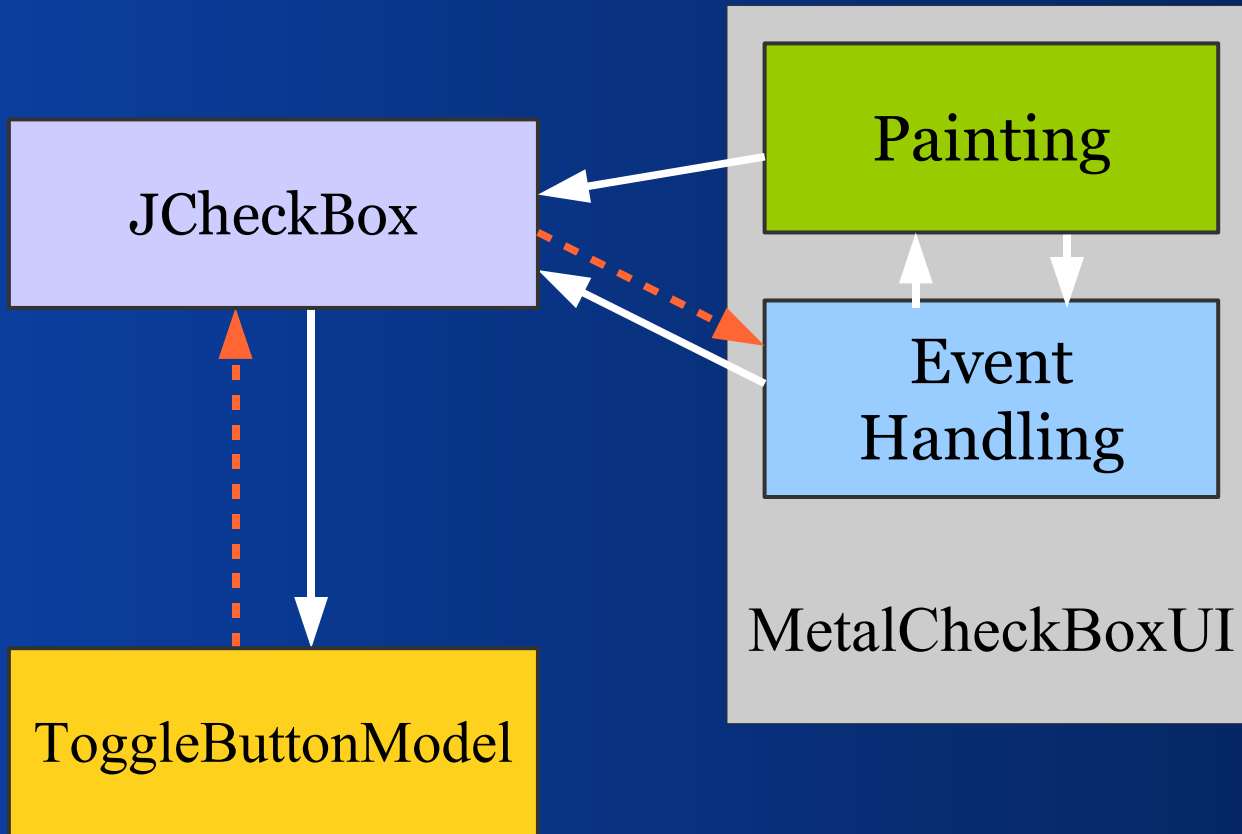
Swing kann Aussehen und Verhalten wechseln.

- Views und Controller sind abgetrennt.

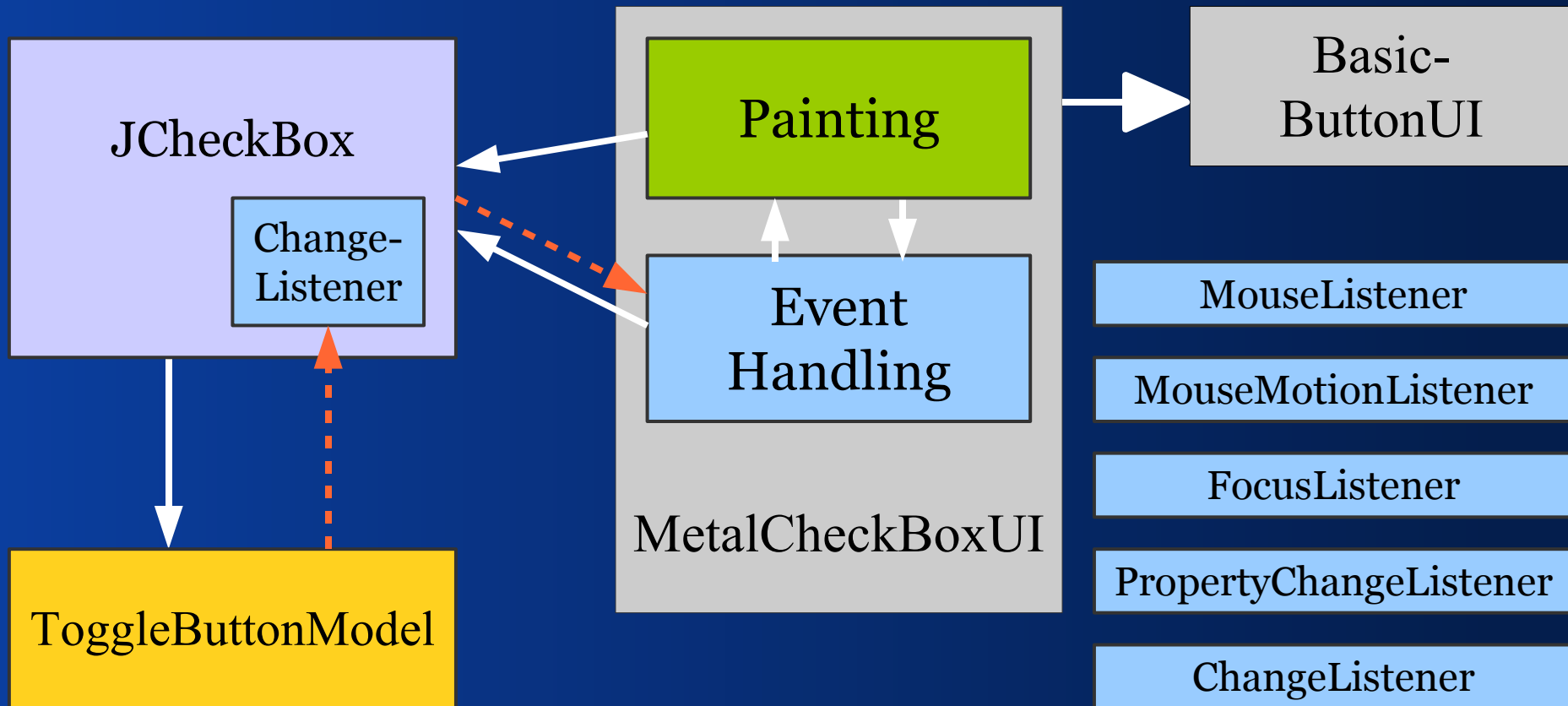
M-JComponent-VC



Beispiel: JCheckBox



JCheckBox: Einige Details

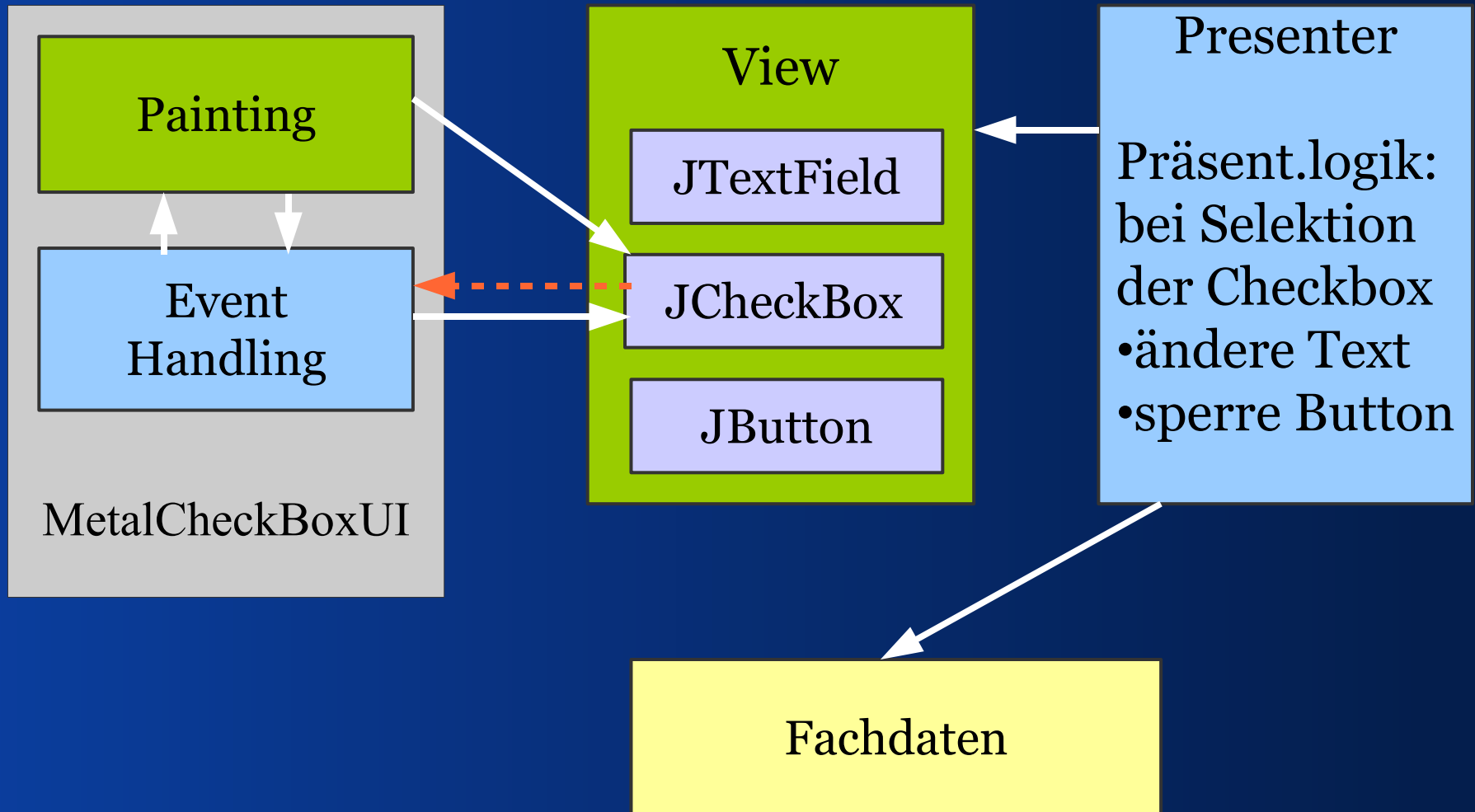


Fazit des Einschubs

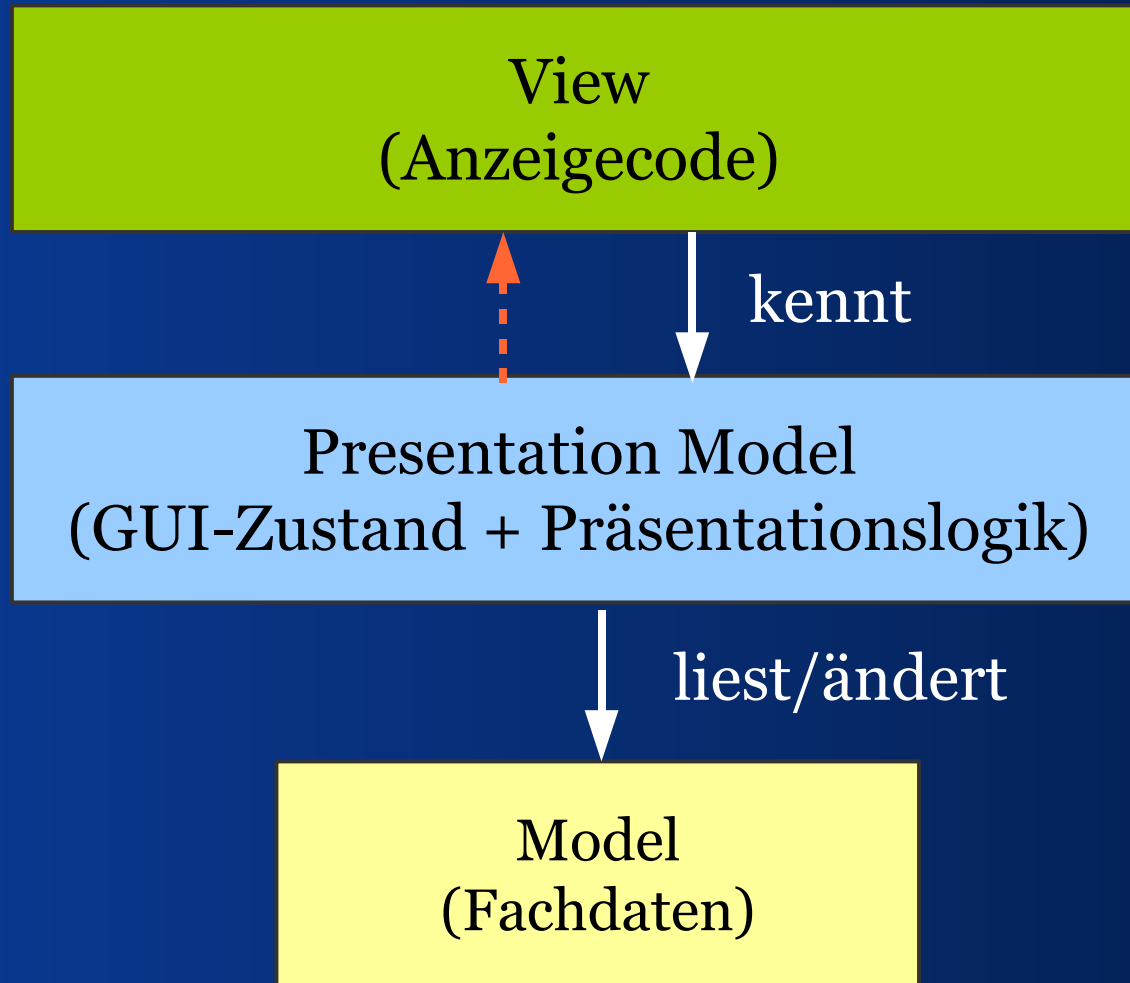
- Swing verwendet nicht das Original-MVC
- Swing nutzt eine erweiterte MVC-Form
- Swing teilt aber die Motivation hinter MVC
- Swing bietet etwas mehr als MVC
- Die UI-Delegates sind View und Controller

- MVC ist für Komponenten,
MVP für Anwendungen

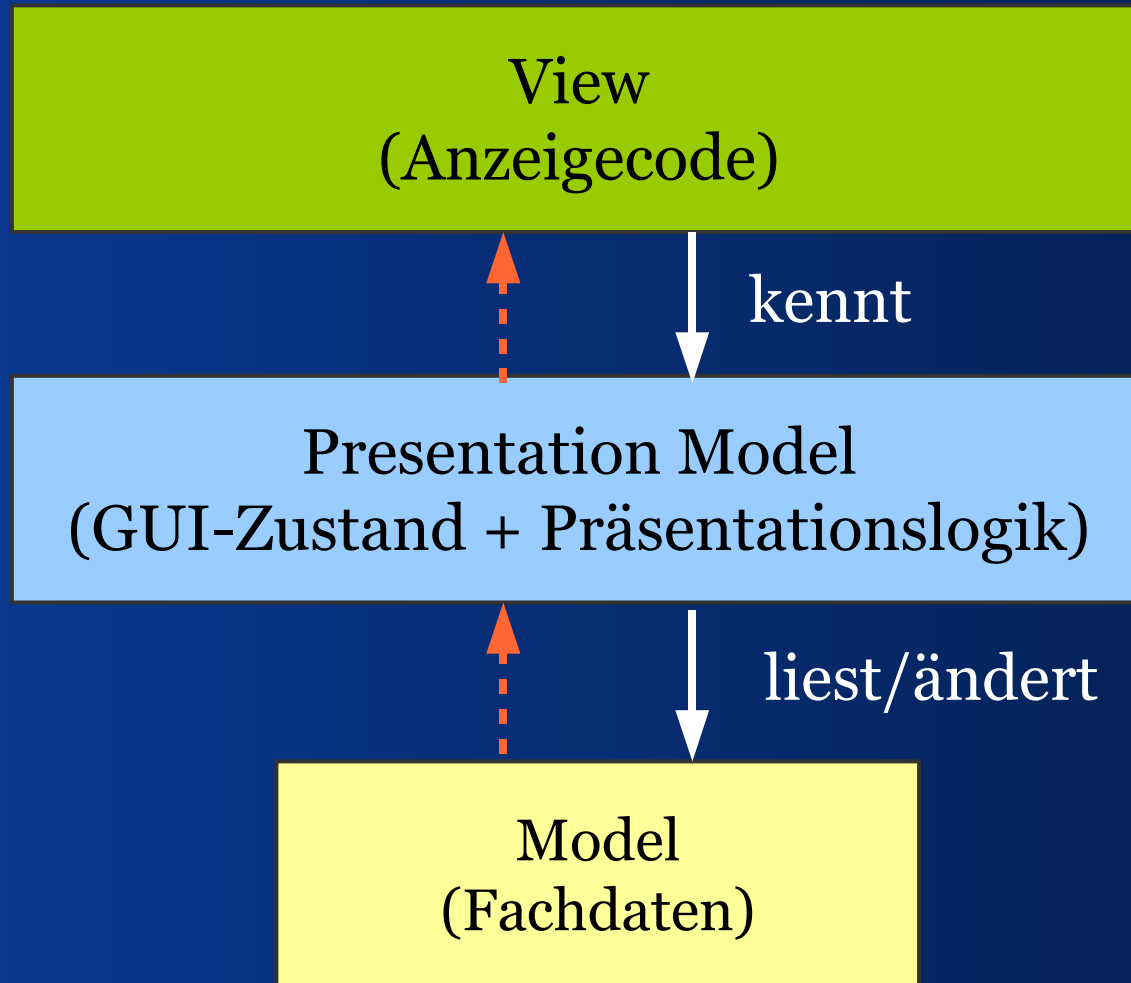
MVP in Swing



Presentation Model (PM)



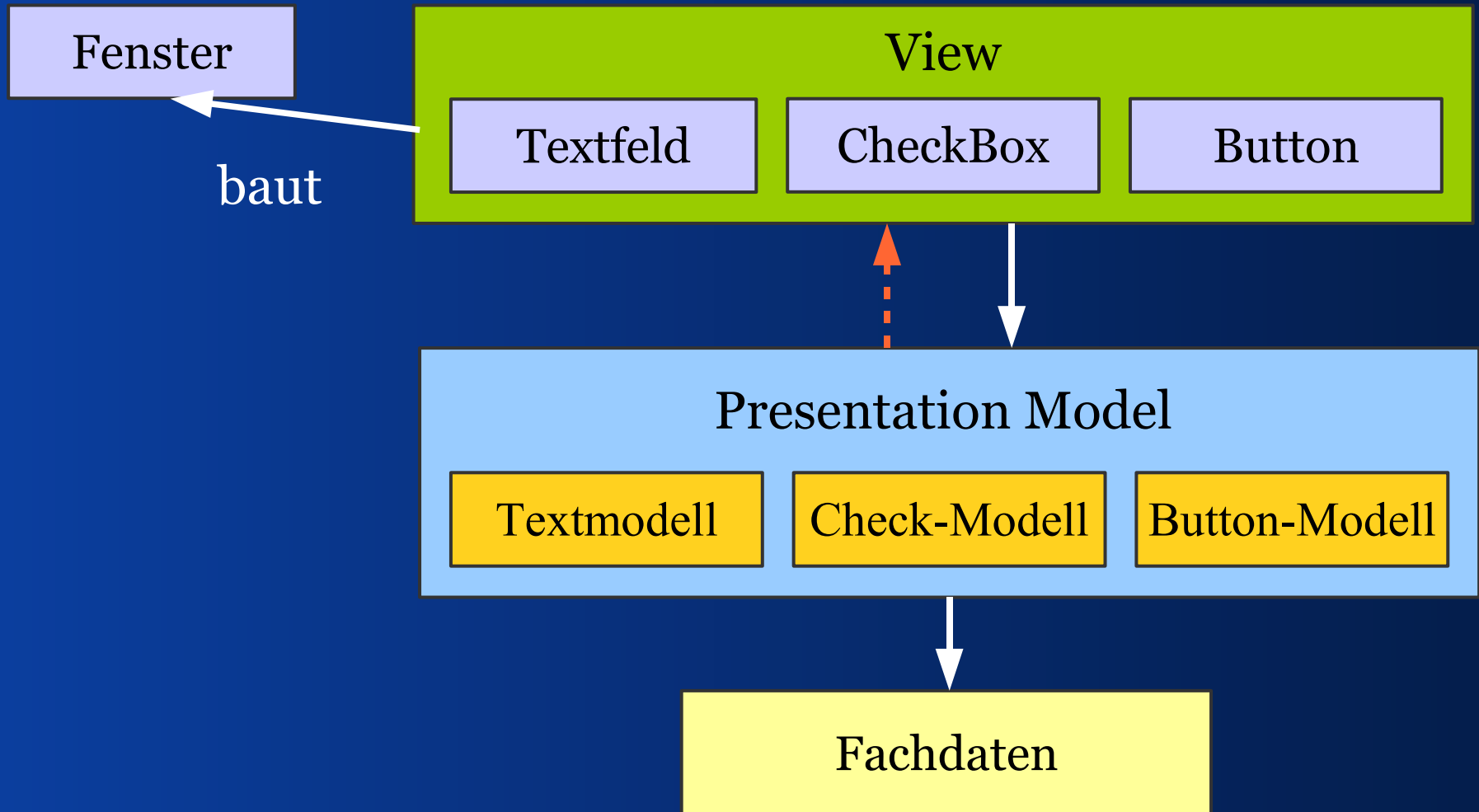
Presentation Model (PM)



Presentation Model

- Der View
 - enthält nur GUI-Komponenten.
 - beobachtet Änderungen im Presentation Model
- Das Presentation Model
 - enthält GUI-Zustand und Präsentationslogik
 - liest Fachdaten in seinen GUI-Zustand
 - reagiert auf GUI-Events, ändert daraufhin seinen GUI-Zustand und meldet Änderungen
 - ändert Fachdaten anhand seines GUI-Zustands

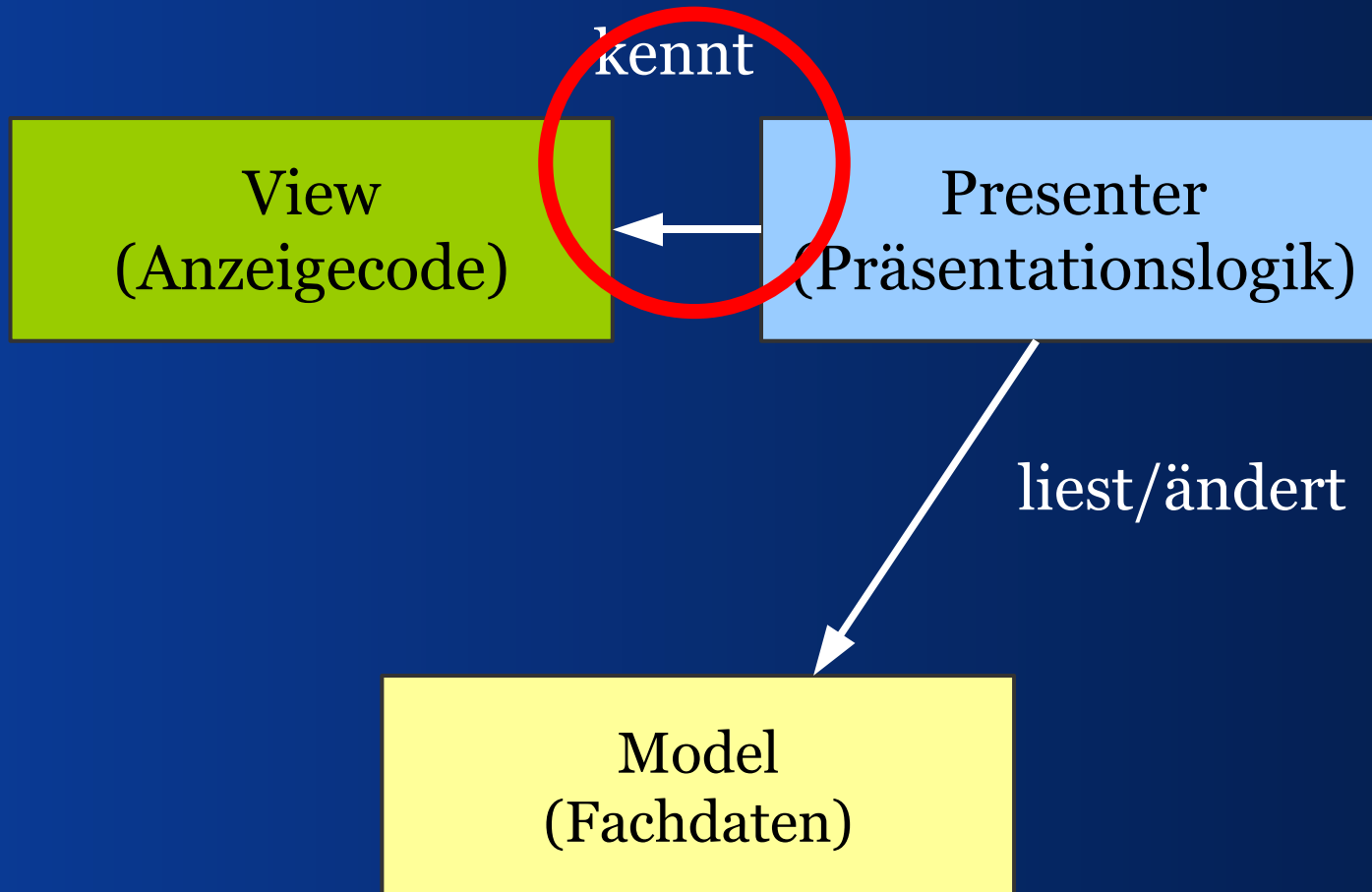
Presentation Model



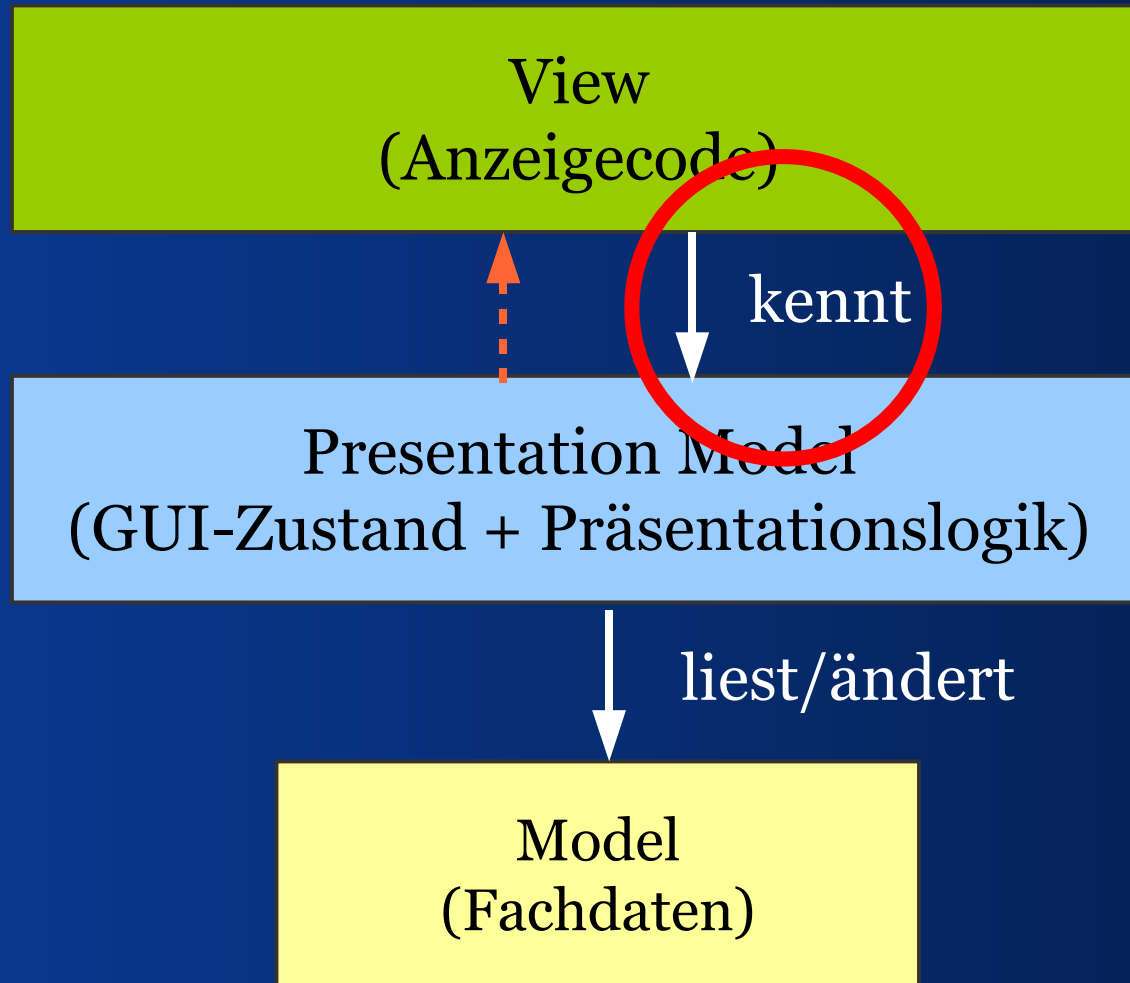
MVP vs. Presentation Model

- Presenter referenziert den View, PM nicht.
- In MVP hält der View den GUI-Zustand.
- Das PM hält den GUI-Zustand selbst.
- Der Presenter ändert GUI-Zustand im View.
- Das PM ändert den GUI-Zustand bei sich und meldet die Änderung an den View.

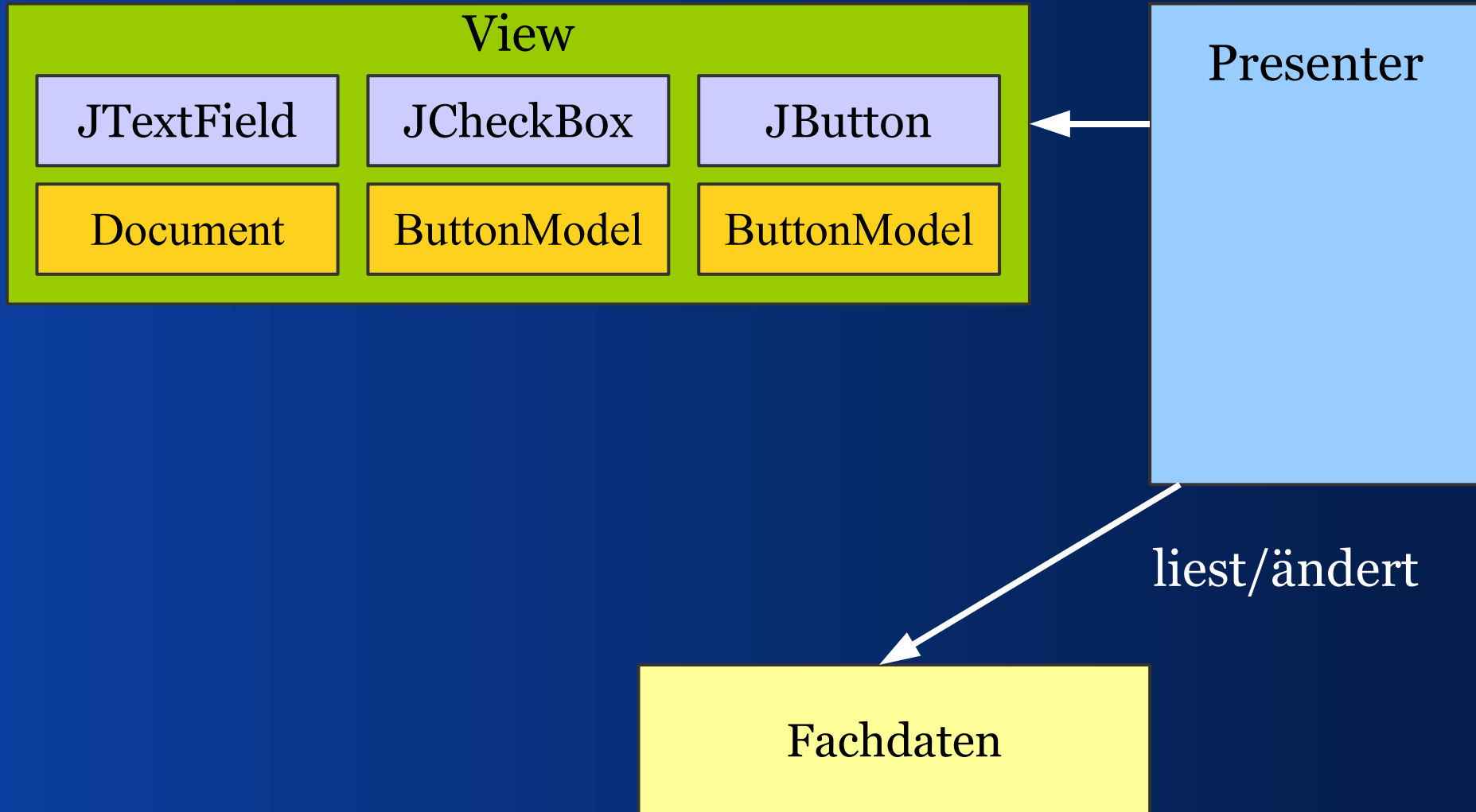
MVP



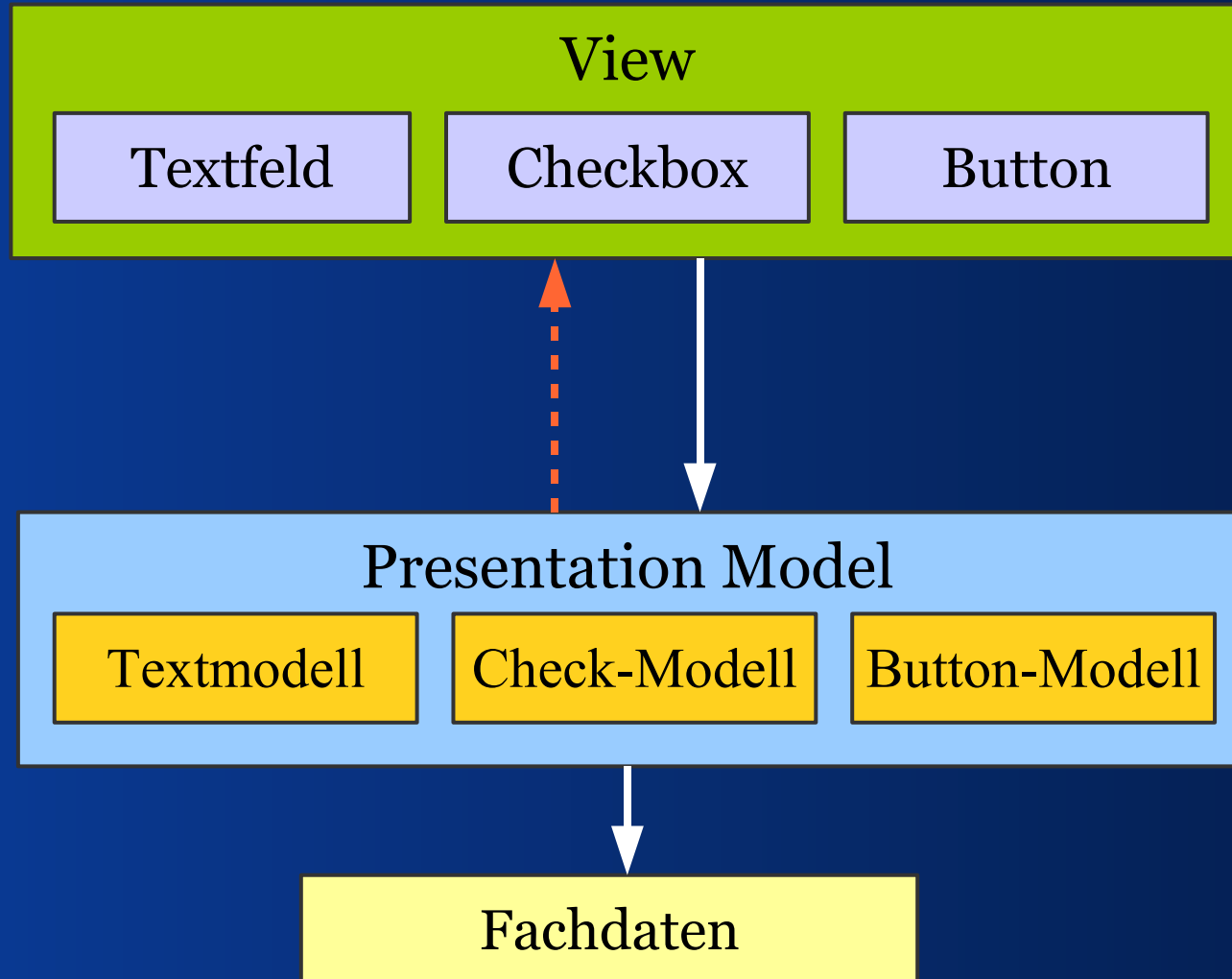
Presentation Model



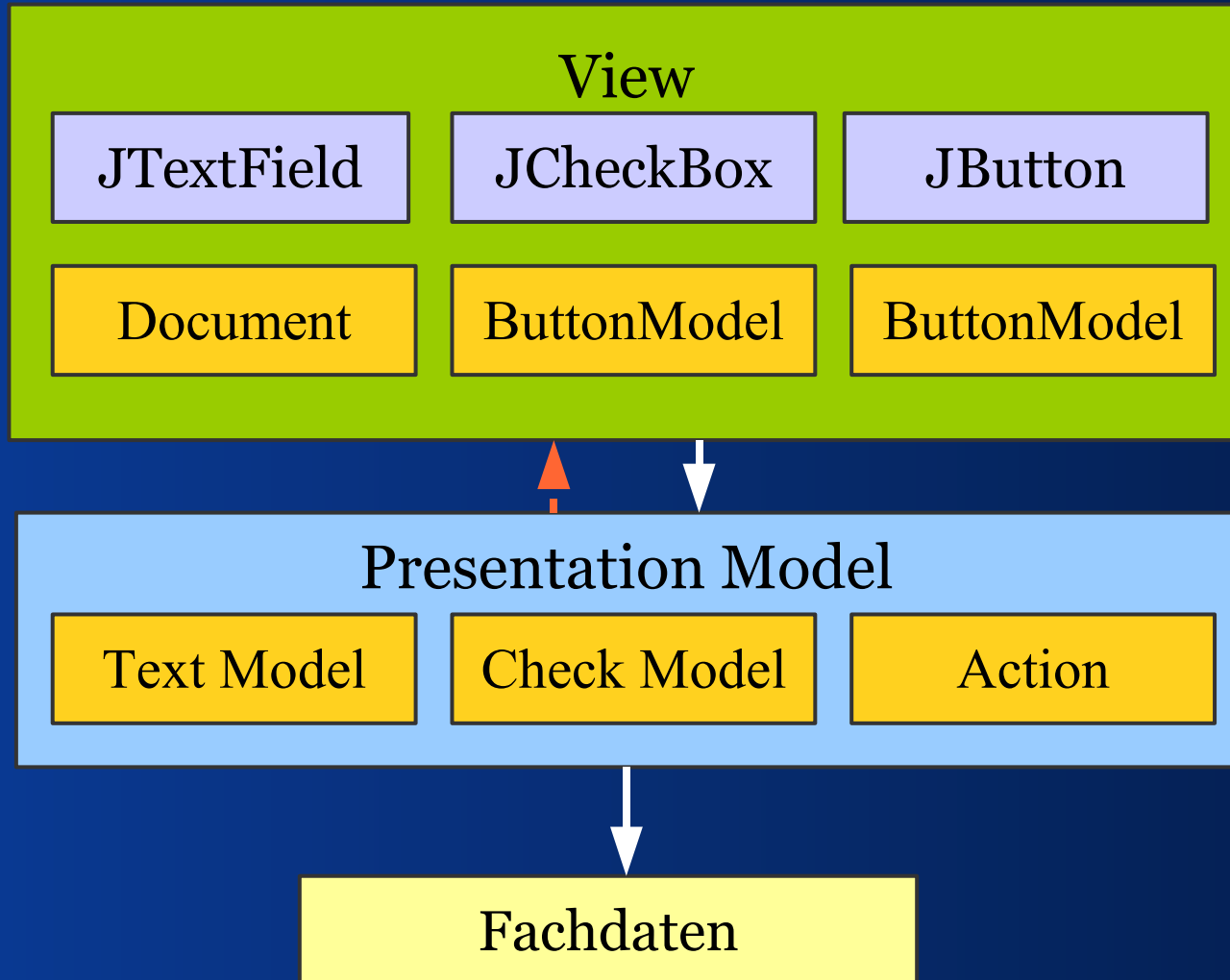
MVP in Swing: GUI-Zustand



PM: GUI-Zustand



PM in Swing: GUI-Zustand

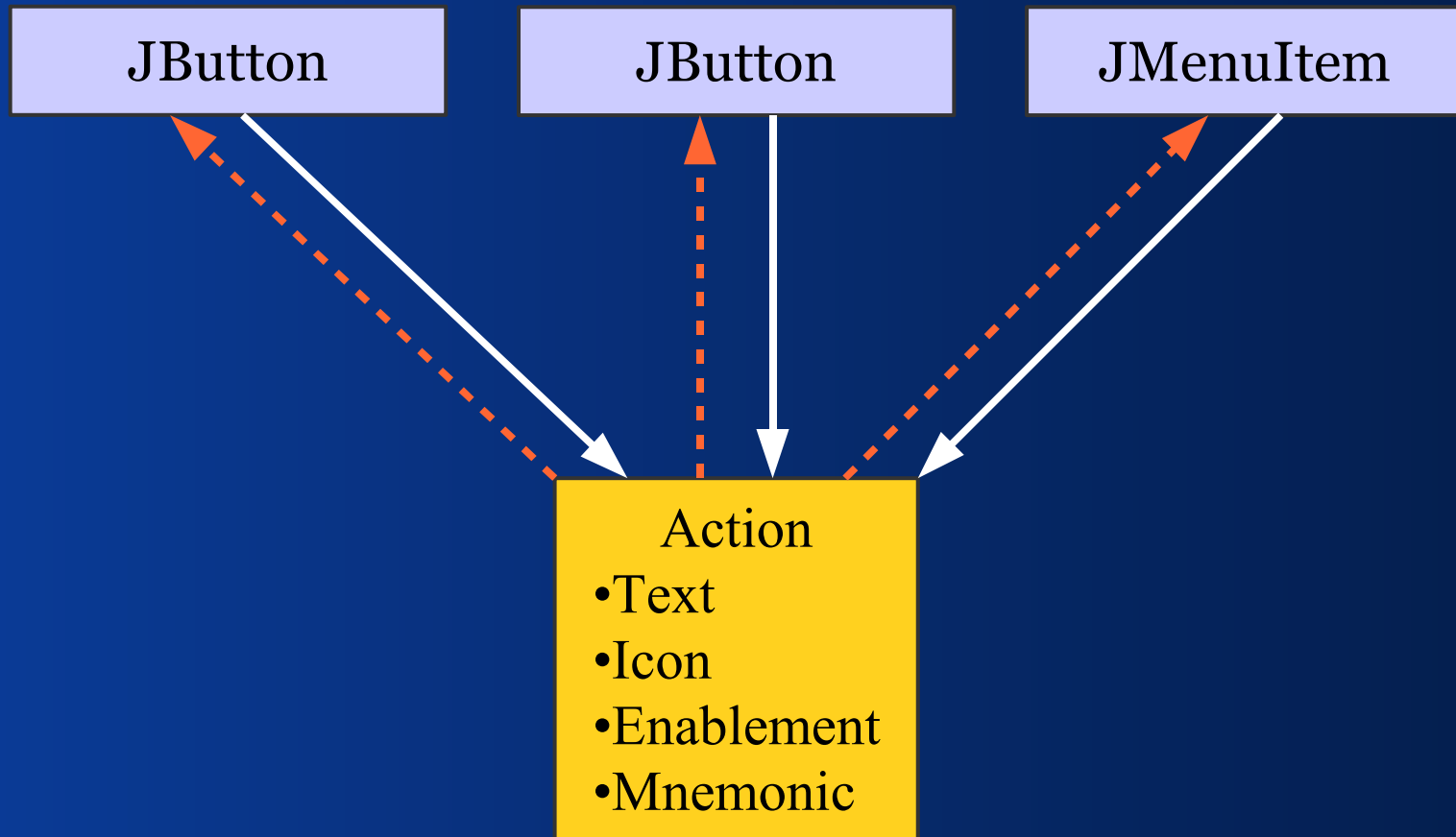


MVP vs. Presentation Model

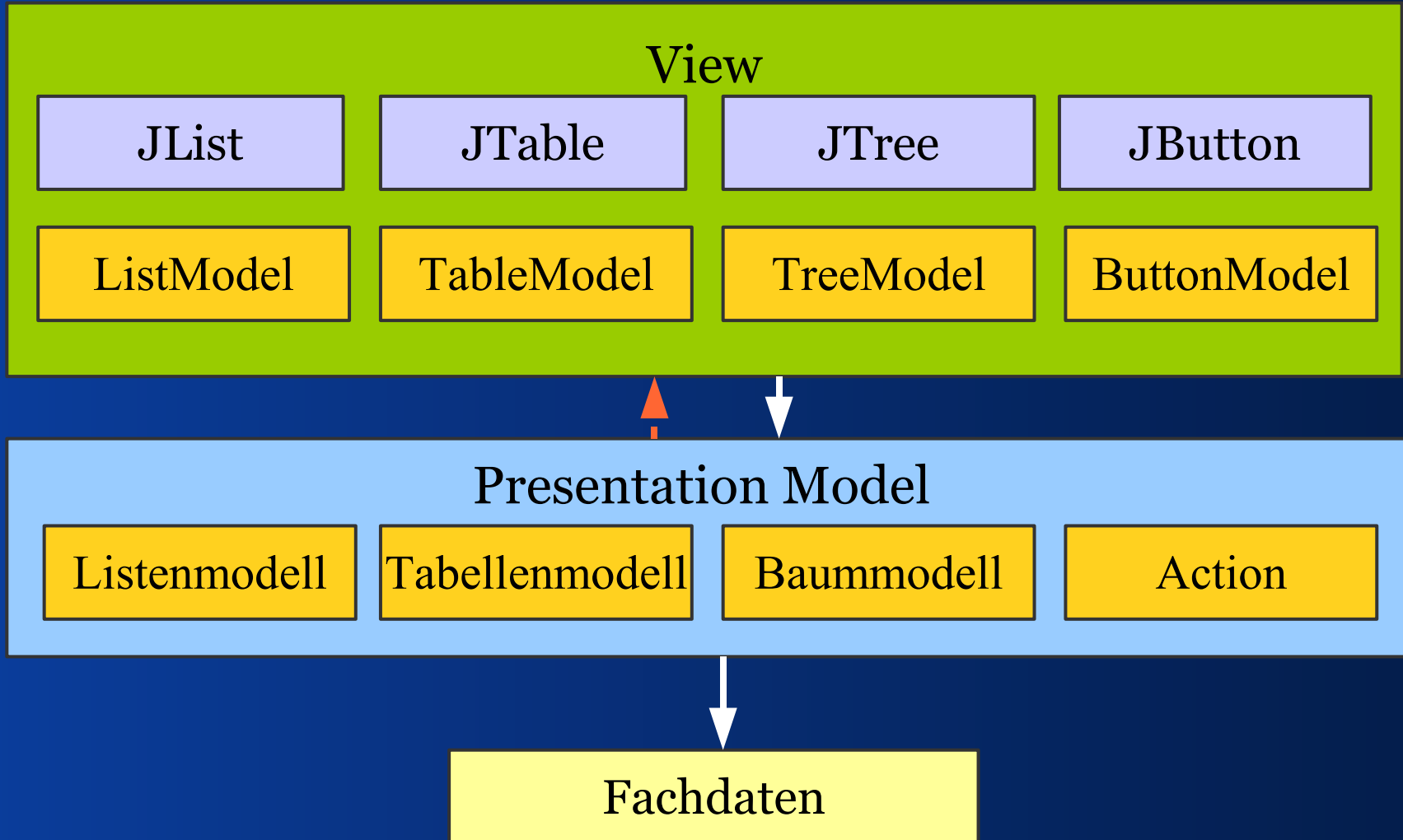
- MVP hält den GUI-Zustand nur **einmal**.
- PM hält ihn **doppelt**, im View und PM.
- PM braucht eine Synchronisation zwischen dem PM-Zustand und dem View-Zustand.

- Keine Angst vor der Synchronisation; sie ist in Swing relativ einfach.

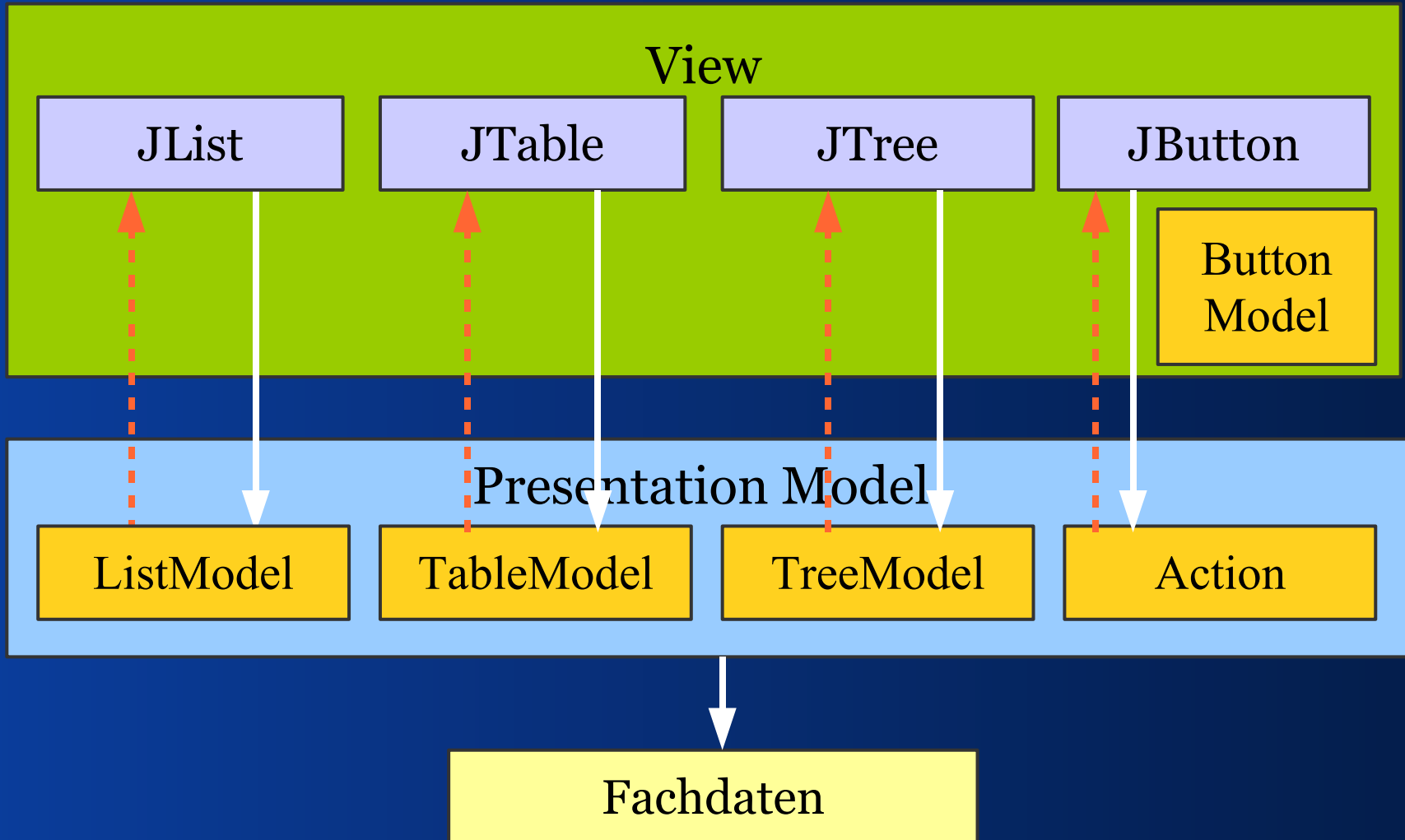
Erinnerung: Swing-Actions



PM: Listen, Tabellen, Bäume



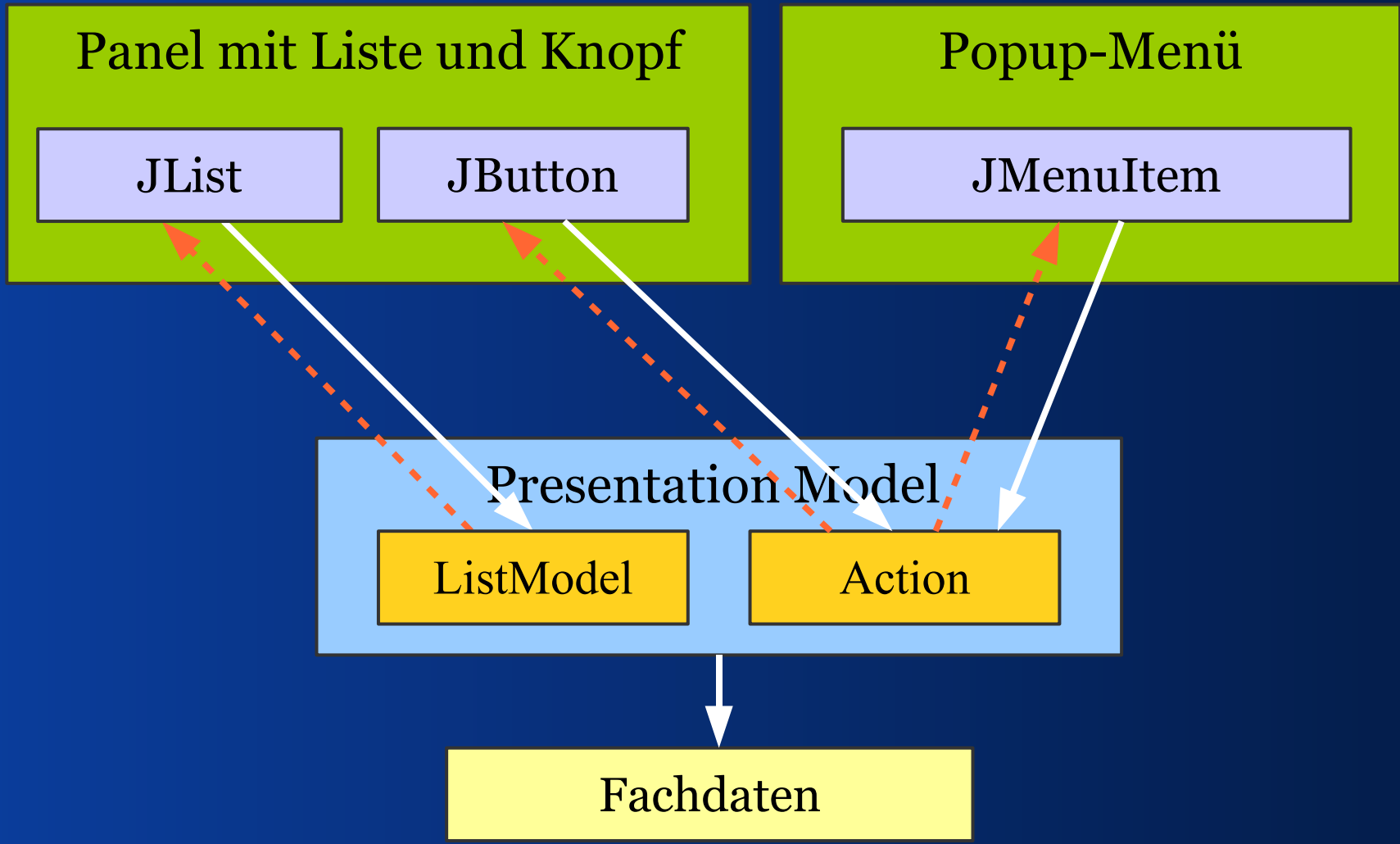
PM: Listen, Tabellen, Bäume



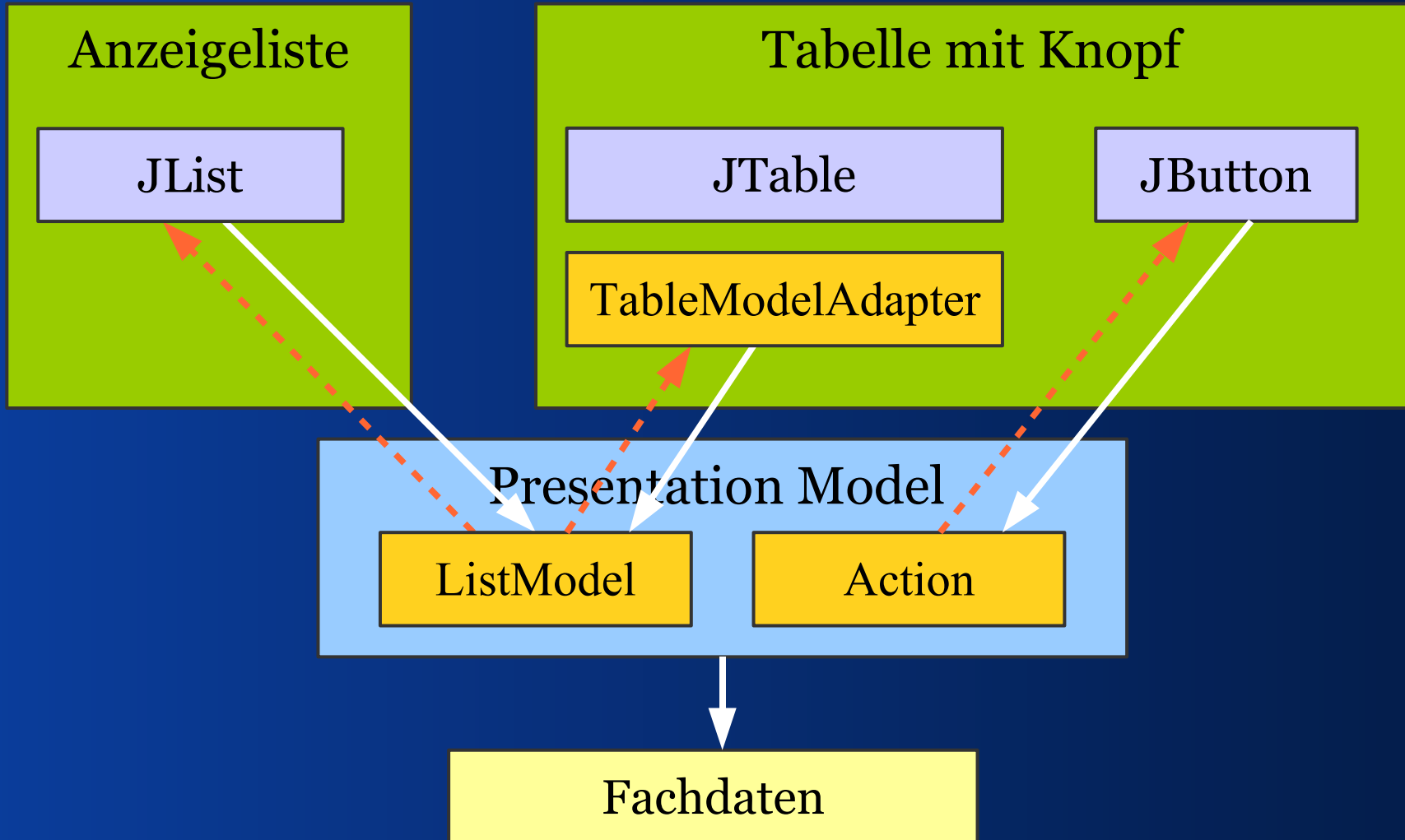
Synchronisationsbeispiel

```
private void initComponents() {  
  
    okButton = new JButton(  
        presentationModel.getOKAction());  
  
    albumList = new JList(  
        presentationModel.getAlbumListModel());  
  
    ...  
}
```

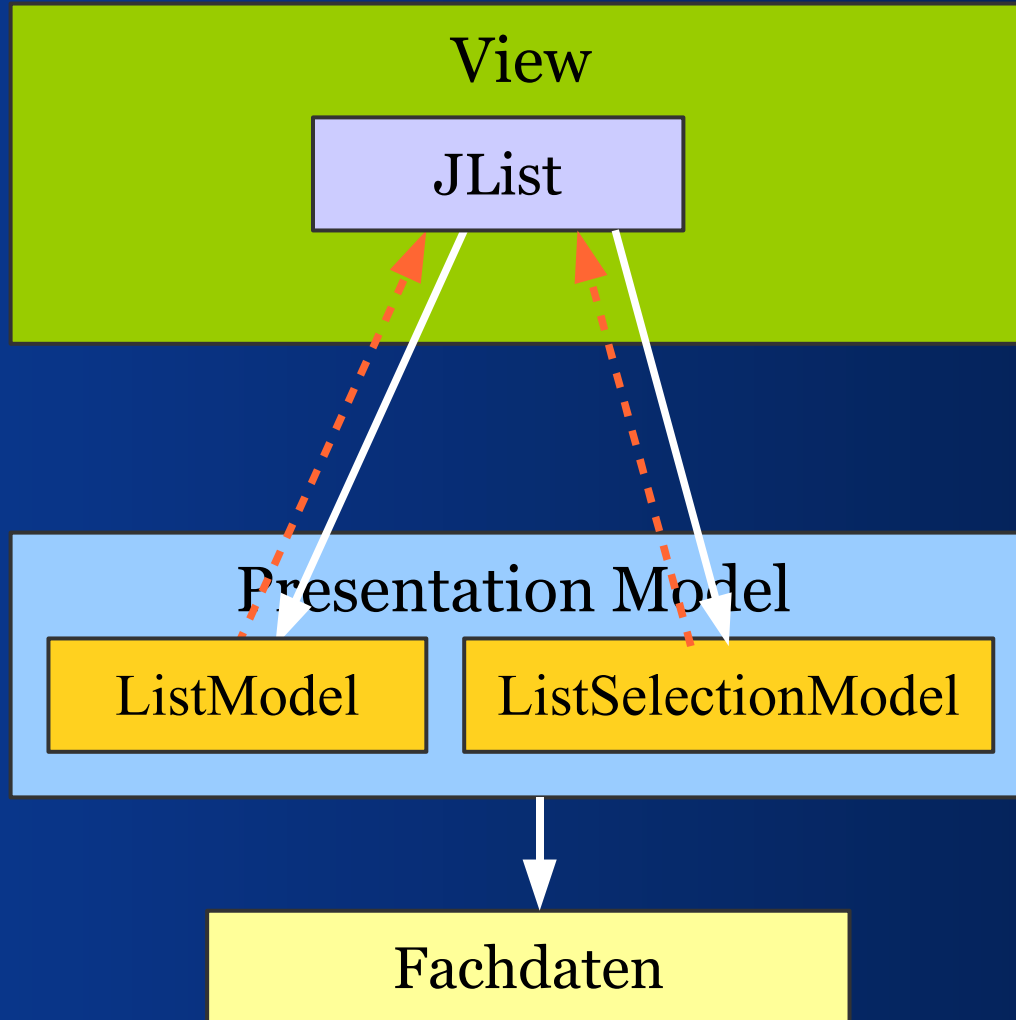
PM: Mehrere Views I



PM: Mehrere Views II



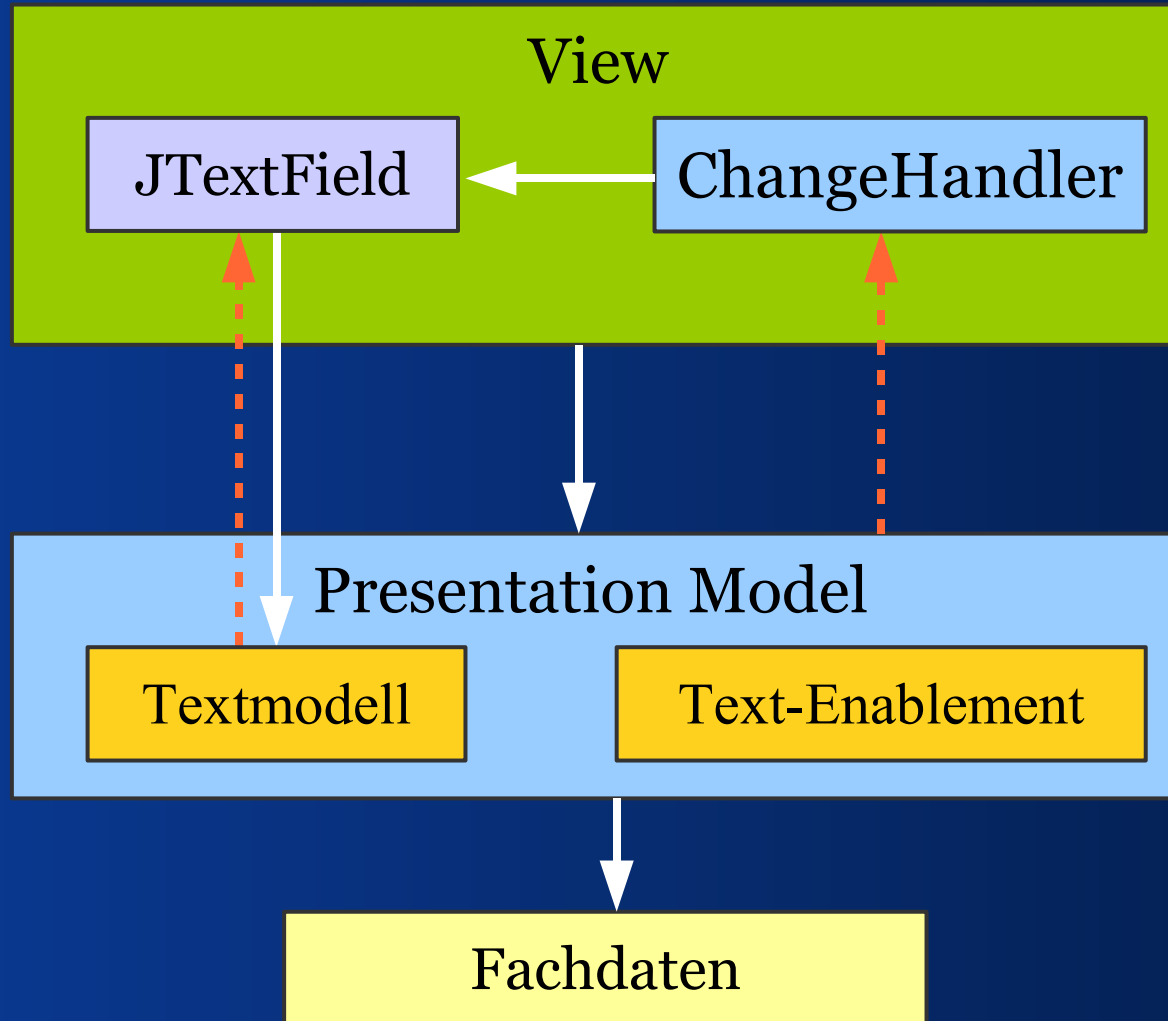
PM: Liste mit Selektion



Drei Dinge fehlen noch

- Wie modellieren wir nicht-Daten-GUI-Zustand, etwa Enablement?
- Wie synchronisieren wir Einzelwerte für JTextField, JFormattedTextField, JLabel?
- Wie synchronisieren wir Einzelwerte der Fachschicht mit dem PM?

PM-Beispiel: Enablement



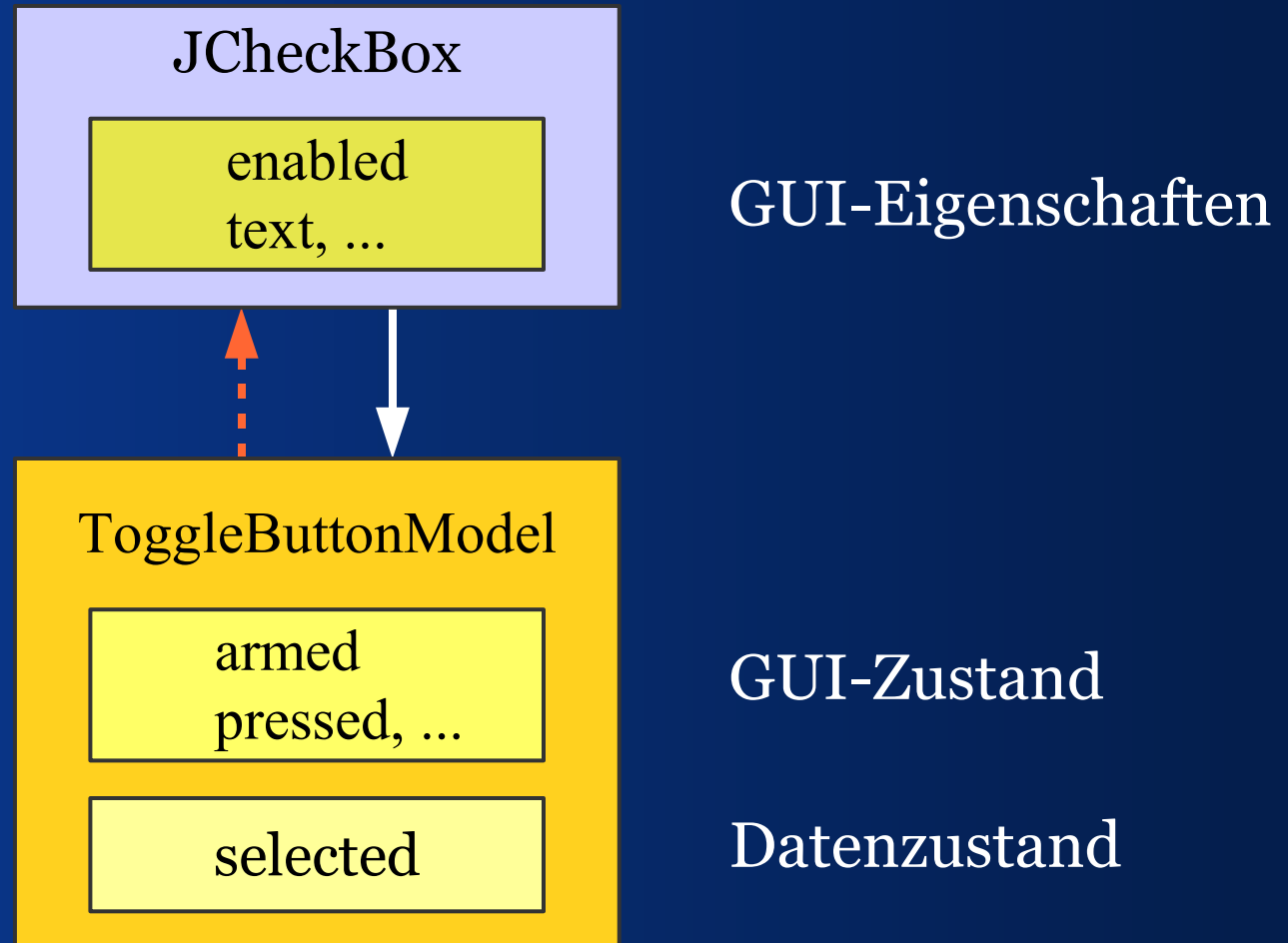
IV - Einzelwerte synchronisieren

*Wie verbinde ich
Facheigenschaften mit Komponenten?*

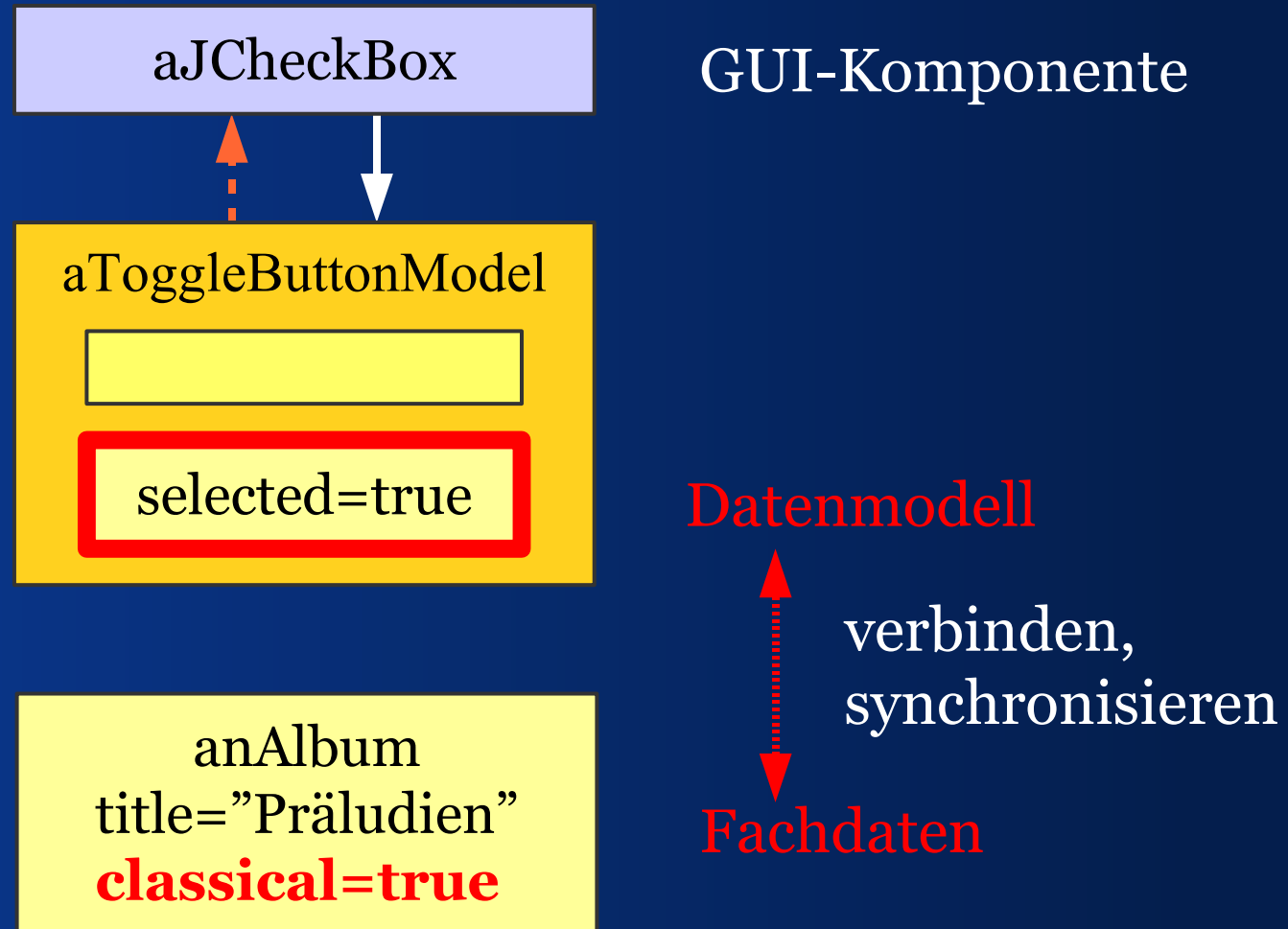
Bindeaufgaben

- Fachdaten lesen und schreiben
- GUI-Modellwerte lesen und setzen
- Fachdatenänderungen melden/behandeln
- Werte puffern – bis OK zurück halten
- Änderungsmanagement – OK nötig?
- Indirektion wie in Übersicht-Detail-Views
- Typen konvertieren, z. B. Datum nach Text

JCheckBox: Zustandsarten



JCheckBox: Bindeaufgabe



Kopieren: Vor- und Nachteile

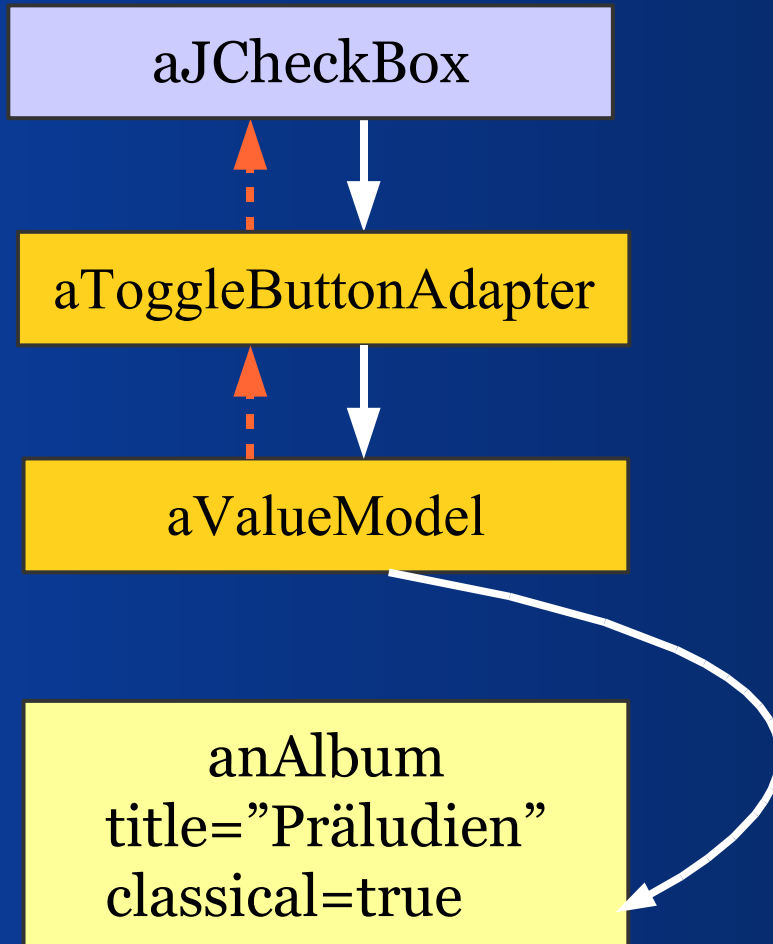
- Einfach zu verstehen und zu erklären
- Funktioniert in fast allen Situationen
- Einfach zu debuggen

- Relativ viel Schreibarbeit
- Synchronisation aufwendig
- Reaktion auf Fachdatenänderung schwierig

Konzept zur Synchronisation

- Nutze ein Universalmodell (ValueModel)
- Konvertiere Facheigenschaften nach ValueModel
- Baue Konverter von ValueModel zu den Swing-Modellen.

ValueModel und Adapter



ValueModel: Anforderungen

- Wir wollen einen Wert lesen
- Wir wollen einen Wert setzen
- Wir wollen Änderungen bemerken

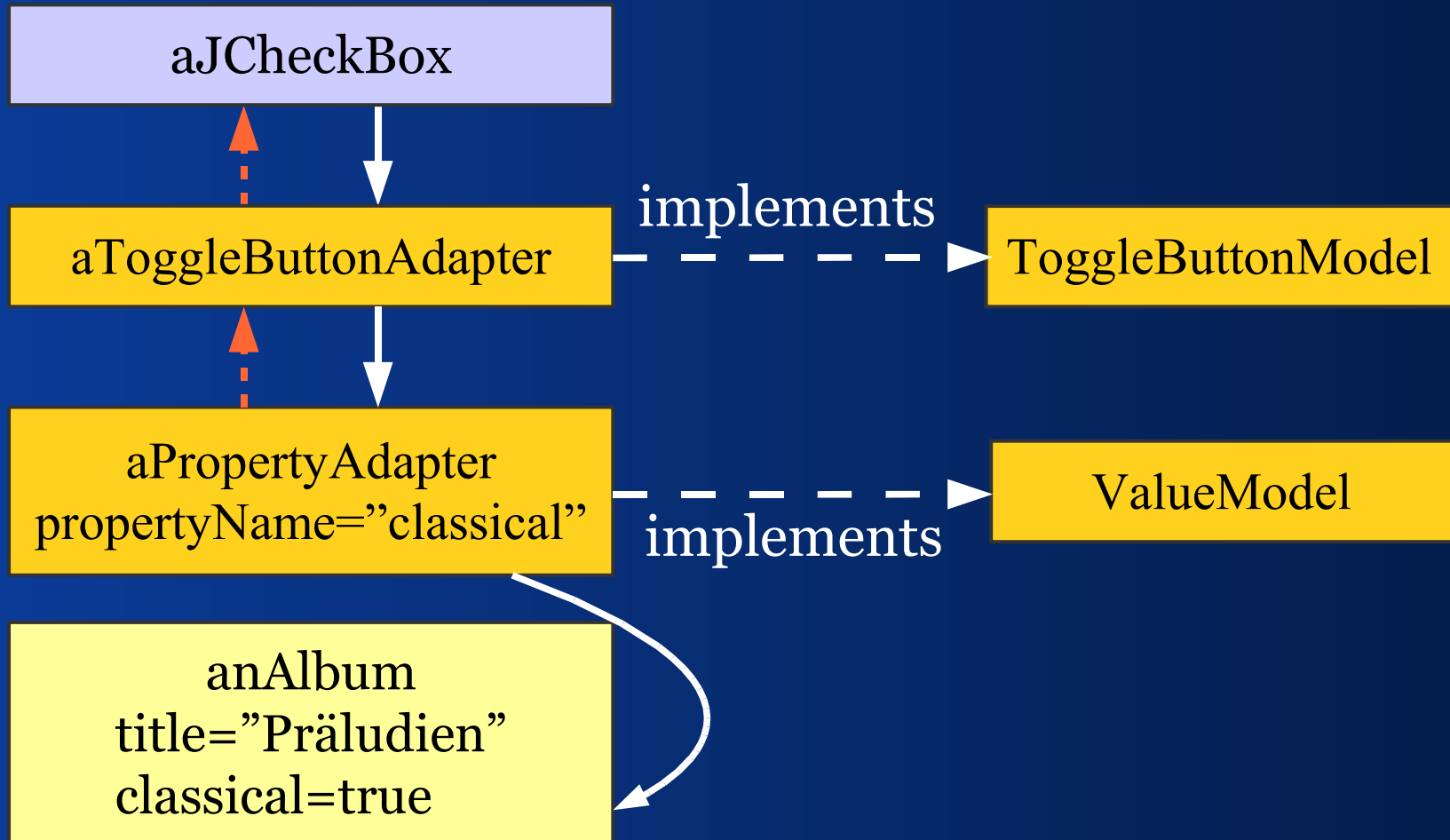
Die Schnittstelle ValueModel

```
public interface ValueModel {  
  
    Object getValue();  
  
    void setValue(Object newValue);  
  
    void addChangeListener(ChangeListener l);  
  
    void removeChangeListener(ChangeListener l);  
}
```

Welcher Event-Typ?

- **ChangeEvent** meldet keinen Wert; den muss man bei Bedarf holen.
- **PropertyChangeEvent** liefert den alten und neuen Wert; beide können aber **null** sein.

ValueModel & PropertyAdapter



Anforderungen an Fachobjekte

- Wir wollen Eigenschaften lesen und setzen
- Wir wollen das einheitlich tun
- Änderungen sollen gemeldet werden

Das bieten Java Beans.

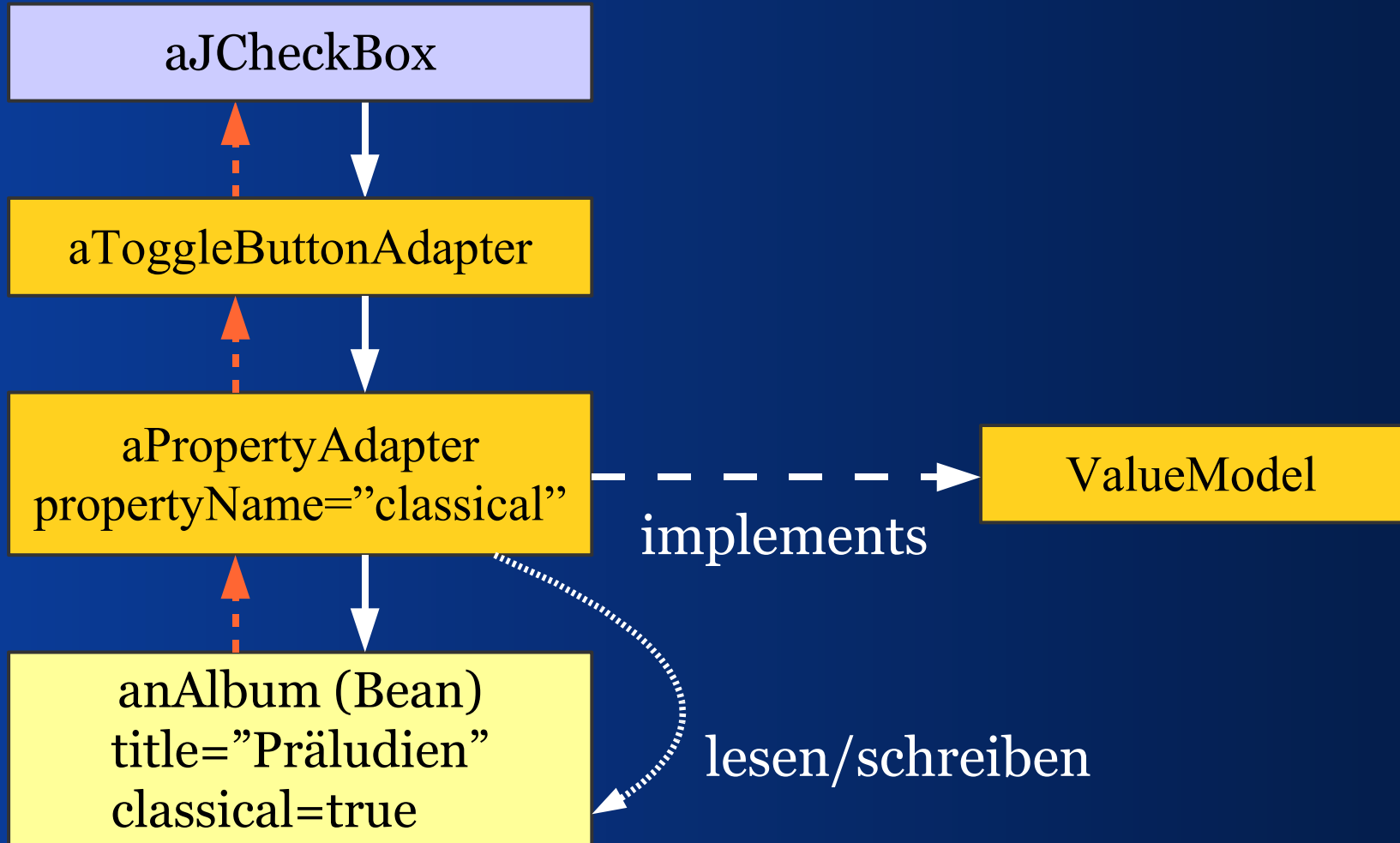
(Bound) Bean Properties

- Java Beans haben Eigenschaften, die man uniform lesen und schreiben kann.
- Bean-Eigenschaften heißen **gebunden** (bound), wenn man Änderungen mittels **PropertyChangeListener** beobachten kann.

PropertyAdapter

- **BeanAdapter** und **PropertyAdapter** konvertieren Bean-Eigenschaften nach ValueModel
- Beobachten Bound properties
- Nutzen Bean-Introspection, das wiederum Reflection nutzt, um Eigenschaften zu lesen und zu setzen

ValueModel & PropertyAdapter



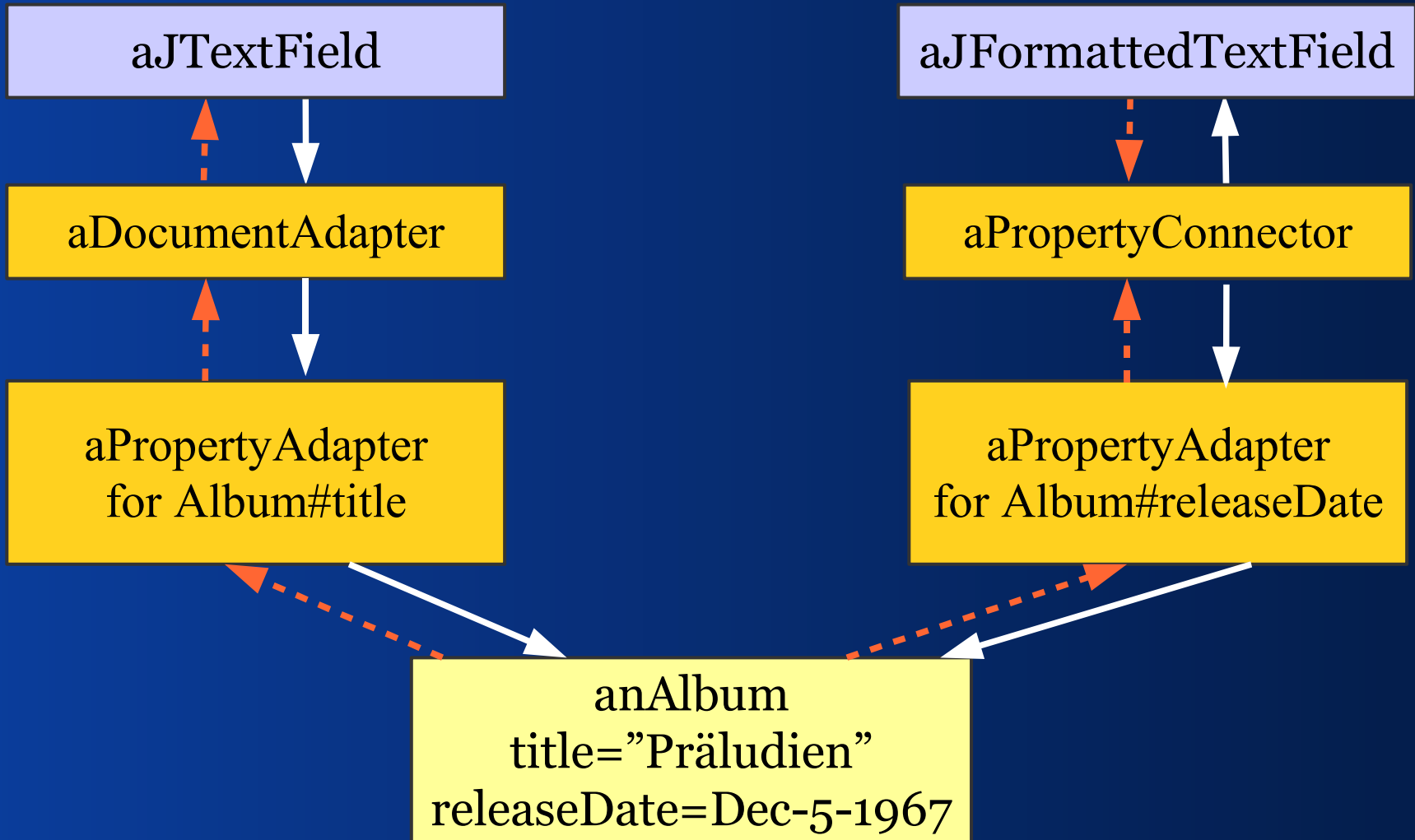
Adapterkette bauen

```
private void initComponents() {  
  
    Album album = getEditedAlbum()  
  
    ValueModel aValueModel =  
        new PropertyAdapter(album, "classical");  
  
    JCheckBox classicalBox = new JCheckBox();  
    classicalBox.setModel(  
        new ToggleButtonAdapter(aValueModel));  
}
```

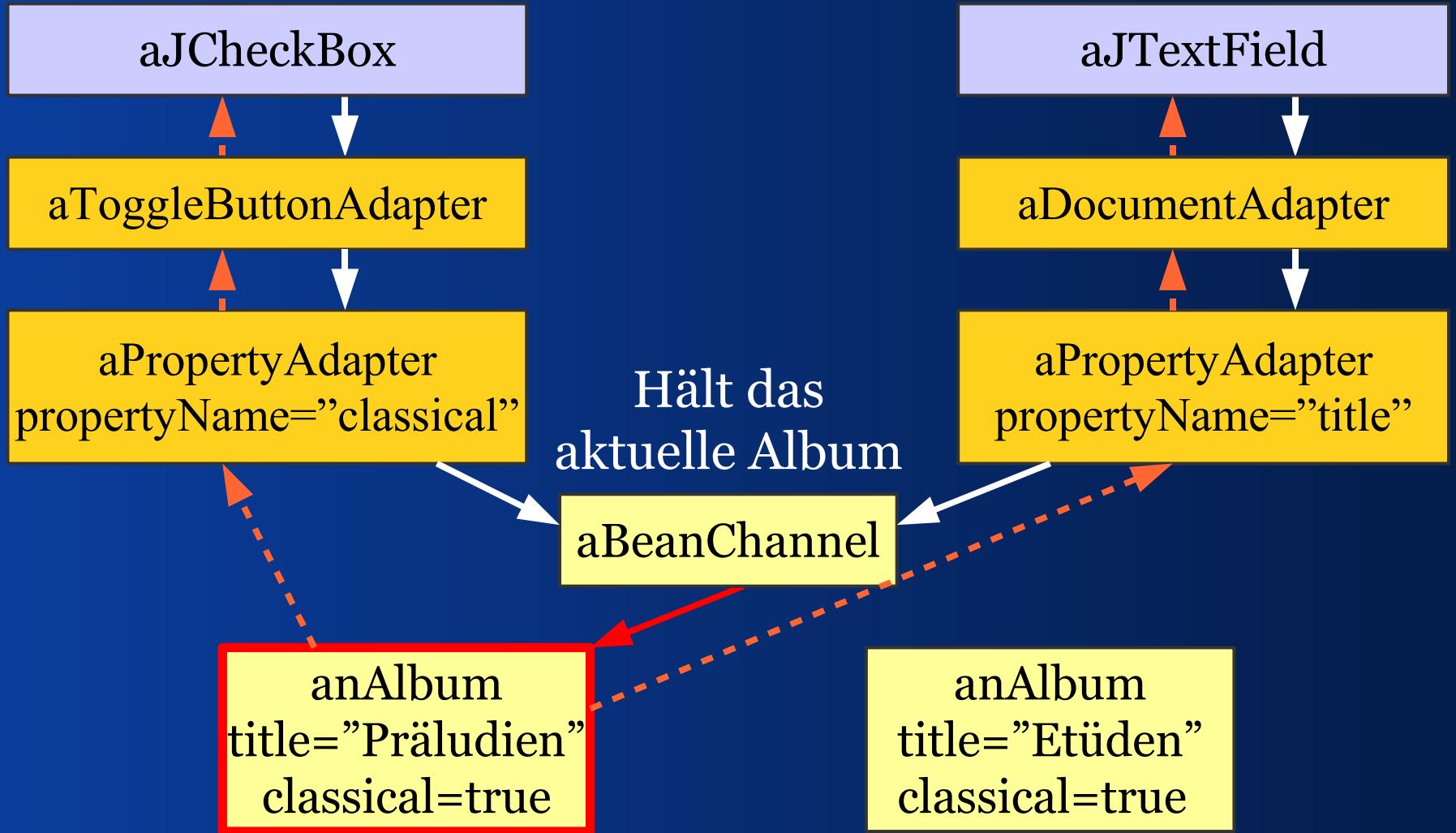
ComponentFactory

```
private void initComponents() {  
  
    Album album = getEditedAlbum();  
  
    JCheckBox classicalBox =  
        ComponentFactory.createCheckBox(  
            album,  
            Album.PROPERTYNAME_CLASSICAL);  
}
```

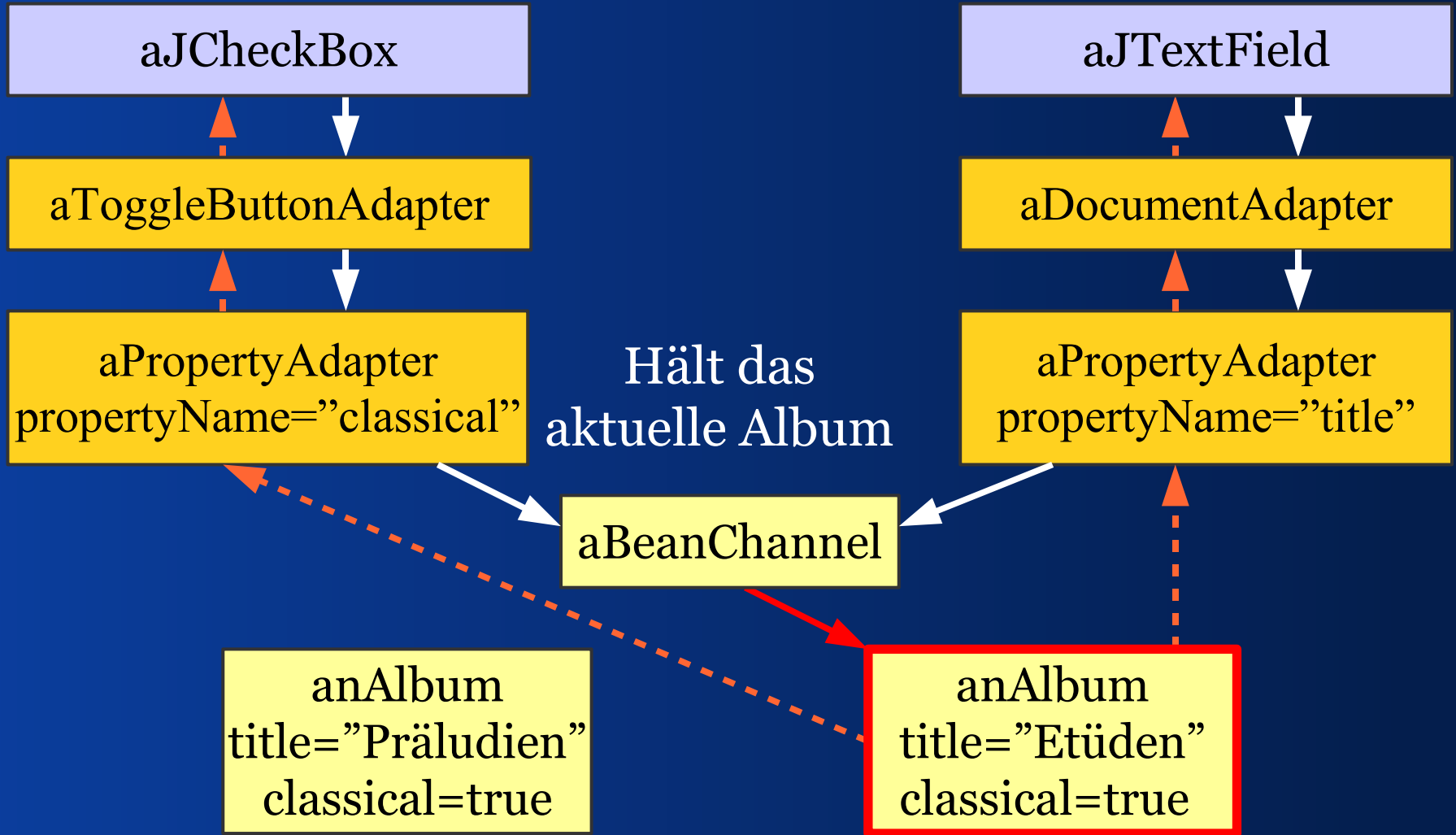
Adapter vs. Connector



Indirektion



Indirektion



View-Quelltextbeispiel

- 1) Variablen für UI-Komponenten
- 2) Konstruktoren
- 3) Erzeuge,binde,konfiguriere UI-Komponenten
- 4) Registriere GUI-Zustands-Handler im Model
- 5) Baue und liefere ein Panel
- 6) Handler, die den GUI-Zustand ändern

Beispiel-View 1/7

```
public final class AlbumView {  
  
    // Referenziert den Modelllieferanten  
    private AlbumPresentationModel model;  
  
    // UI-Komponenten  
    private JTextField titleField;  
    private JCheckBox  classicalBox;  
    private JButton    buyNowButton;  
    private JList      referencesList;  
    ...  
}
```

Beispiel-View 2/7

```
public AlbumView(AlbumPresentationModel m) {  
    // Merke den Modelllieferanten  
    this.model = m;  
  
    // Konfiguration des Views.  
    ...  
}
```

Beispiel-View 3/7

```
private void initComponents() {  
    titleField = ComponentFactory.createField(  
        model.getTitleModel());  
    titleField.setEditable(false);  
  
    buyNowButton = new JButton(  
        model.getBuyNowAction());  
  
    referenceList = new JList(  
        model.getReferenceListModel());  
    referenceList.setSelectionModel(  
        model.getReferenceSelectionModel());  
}
```

Beispiel-View 4/7

```
private initEventHandling() {  
    // Registriere einen GUI-Zustands-Handler  
    model.addPropertyChangeListener(  
        "composerEnabled",  
        new ComposerEnablementHandler());  
}
```

Beispiel-View 5/7

```
public JPanel buildPanel() {  
    // Erzeuge,binde,konfiguriere Komponenten  
    initComponents();  
  
    // Registriere GUI-Zustands-Handler  
    initEventHandling();  
  
    FormLayout layout = new FormLayout(  
        "right:pref, 3dlu, pref", // 3 columns  
        "p, 3dlu, p");           // 3 rows  
  
    ...  
}
```

Beispiel-View 6/7

```
PanelBuilder builder =  
    new PanelBuilder(layout);  
CellConstraints cc = new CellConstraints();  
  
builder.addLabel("Title", cc.xy(1, 1));  
builder.add(titleField, cc.xy(3, 1));  
builder.add(availableBox, cc.xy(3, 3));  
builder.add(buyNowButton, cc.xy(3, 5));  
builder.add(referenceList, cc.xy(3, 7));  
  
return builder.getPanel();  
}
```

Beispiel-View 7/7

```
/* Lauscht auf #composerEnabled,  
   schaltet #enabled im composerField.   */  
private class ComposerEnablementHandler  
    implements PropertyChangeListener {  
  
    public void propertyChange(  
        PropertyChangeEvent evt) {  
  
        composerField.setEnabled(  
            model.isComposerEnabled());  
    }  
}
```

Einfacheres Event Handling

```
private initEventHandling() {  
    // Synchronisiere Model- mit GUI-Zustand  
    PropertyConnector.connect(  
        model,          "composerEnabled",  
        composerField, "enabled");  
}
```

V - Erfahrungsbericht

Wie funktioniert PM und Bindung im Alltag?

Anspruch an ein Data Binding

- Arbeitet mit Standard-Swing-Komponenten
- Arbeitet mit eigenen Swing-Komponenten
- Braucht keine besonderen Komponenten
- Braucht keine besonderen JPanel
- Passt zu unterschiedlichen Prüfstilen

Kosten

- Adapterbinding:
 - Erhöht die **Lernkosten**
 - Senkt die **Produktionskosten** geringfügig
 - Kann **Änderungskosten** stark senken

Verbinde mittels Fabrik.

- Kapsel den Aufbau der Verbindung vom ValueModel zur Swing-Komponente.
- Einige Swing-Komponenten haben kein geeignetes Modell z. B. JFormattedTextField
- Liefere z. B. Komponenten zu ValueModels

Tipp

- Observer/Observable funktioniert gut zwischen verschiedenen Schichten.
- Aber meide Observer innerhalb einer Schicht.

Warnungen

- Observer in der Fachschicht verhindert, dass man auf einen Blick begreift, was bei einer Fachdatenänderung geschieht.
- Achte auf Memory-Leaks, wenn Du Fachdaten beobachtest durch Listener, die permanent registriert werden. Dann referenzieren Fachdaten die GUI.

Kritik am PM-Muster

- MVP ist entworfen worden, weil in Smalltalk-Presentation Modellen etliche Entwickler doch direkt auf View-Eigenschaften zugegriffen hatten.
- Swing und Binding fördern die Disziplin.

Performanz

- In den Adapterketten werden viele PropertyChangeEvents gefeuert.
- Das ist nicht als Problem zu merken.
- ListModel spart, Listeninhalte zu kopieren.

Debugging

- Was die Bindeklassen an Arbeit abnehmen lasten sie einem beim Debuggen auf.
- Reflection und Introspection erschweren das Verständnis, wer Werte liest und setzt.
- Darum gib allen Listenern einen Namen.
- Nutze bei Bedarf loggende Listener.

Umbenennen

- Reflection und Introspection erschweren das Umbenennen von Eigenschaften und deren Gettern/Settern.
- Verwende Konstanten für die Eigenschaftsnamen.
- Obfuscator verliert Zusammenhänge.

Für wen passt PM und Binding?

- Ich schätze, PM und Adapterbindung passen in etwa 80% aller Fälle.
- Allerdings braucht man einen Experten im Team, der die Bindeklassen **beherrscht**; es reicht nicht, sie zu kennen.

Ganz oder gar nicht

- Eine 3-Schichtenarchitektur mit PM lohnt, wenn die Anwendung überwiegend danach aufgebaut ist.
- Refaktoriert man Teile zu diesem Muster, sind die Umbaukosten hoch und der Nutzen zweifelhaft.

Wo steht JGoodies Binding?

- Ansatz ist 10 Jahre alt und stabil.
- Architektur des Java-Ports ist stabil.
- Tests decken etwa 90% der Klassen ab.
- Wenig Dokumentation.
- Tutorial ist noch recht klein.

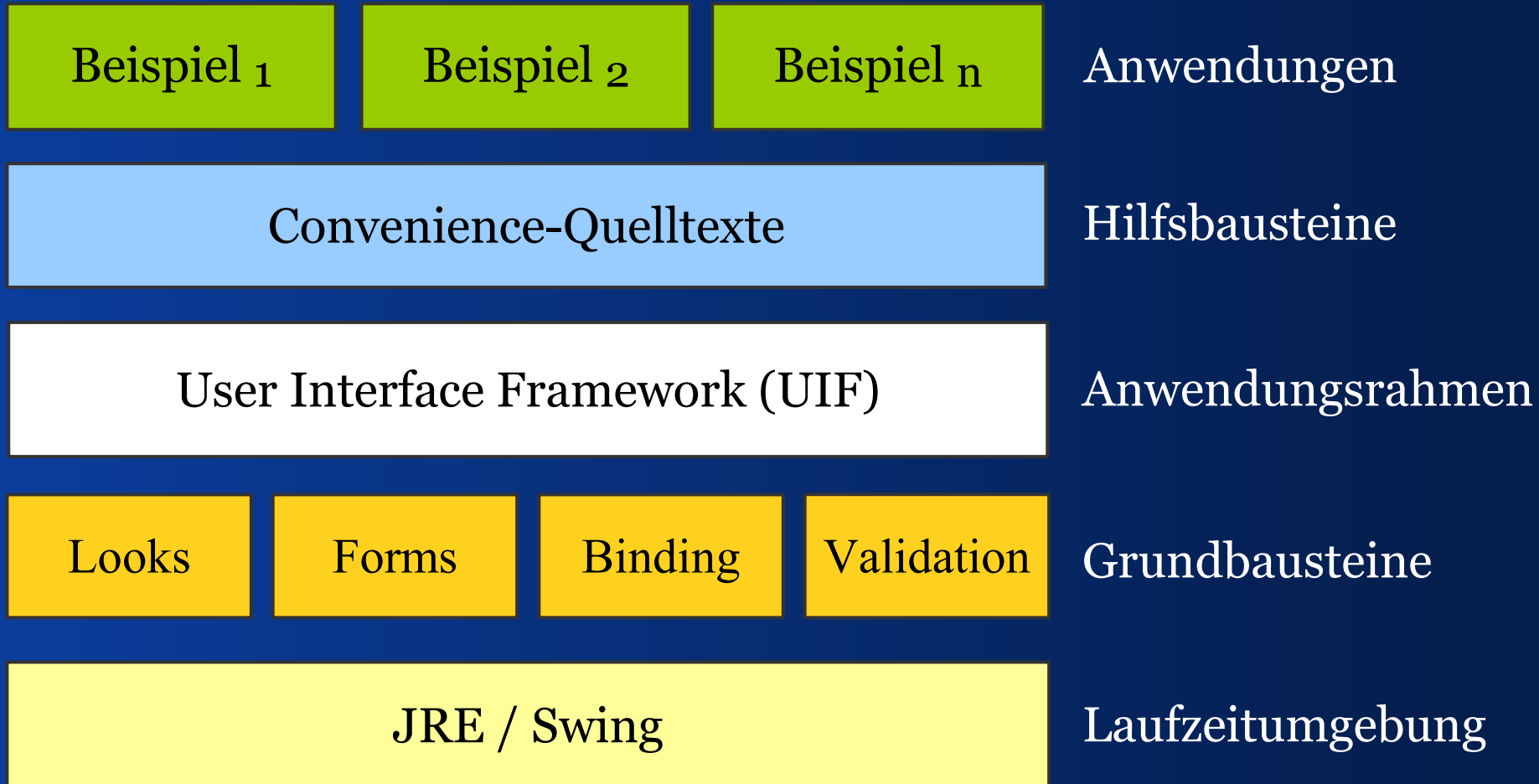
Schluss

Zusammenfassung und Referenzen

Zusammenfassung

- Trenne Fachdaten von der Präsentation!
Das ist **Separated Presentation**.
- Zerlege **Autonomous View** bei Bedarf
- Wähle **MVP** oder **Presentation Model**
- Swing erleichtert **Presentation Model**
- PM braucht eine Bindelösung

JGoodies Swing Suite



Referenzen I

- Fowlers Enterprise Patterns
martinfowler.com/eaDev/
- JGoodies Binding
binding.dev.java.net
- JGoodies-Artikel
www.JGoodies.com/articles/
- JGoodies-Demos
www.JGoodies.com/freeware/

Referenzen II

- Suns JDNC
jdnc.dev.java.net
- Oracles JClient und ADF
otn.oracle.com/, nach 'JClient' suchen
- Spring Rich Client Project
www.springframework.org/spring-rcp.html
- “Desktop Java Live” von Scott Delap

Referenzen III

- VisualWorks-Anwendungsarchitektur
tinyurl.com/yulru
- Understanding and Using ValueModels
c2.com/ppr/vmodels.html
- Model-View-Presenter (MVP)
tinyurl.com/33snk
- HMVC / Scope
tinyurl.com/39q9u, scope.sourceforge.net/

Kleinbeispiele/Tutorial:

JGoodies Binding Tutorial

Aufgaben und Lösungen zum Binden

Kommt mit dem JGoodies Binding

Fragen und Antworten

Ende

Hoffentlich hilft's!

Viel Erfolg!