Desktop Patterns and Data Binding for Swing

Karsten Lentzsch
www.JGoodies.com

Presentation Goals

Learn how to organize presentation logic and how to bind domain data to a Swing UI

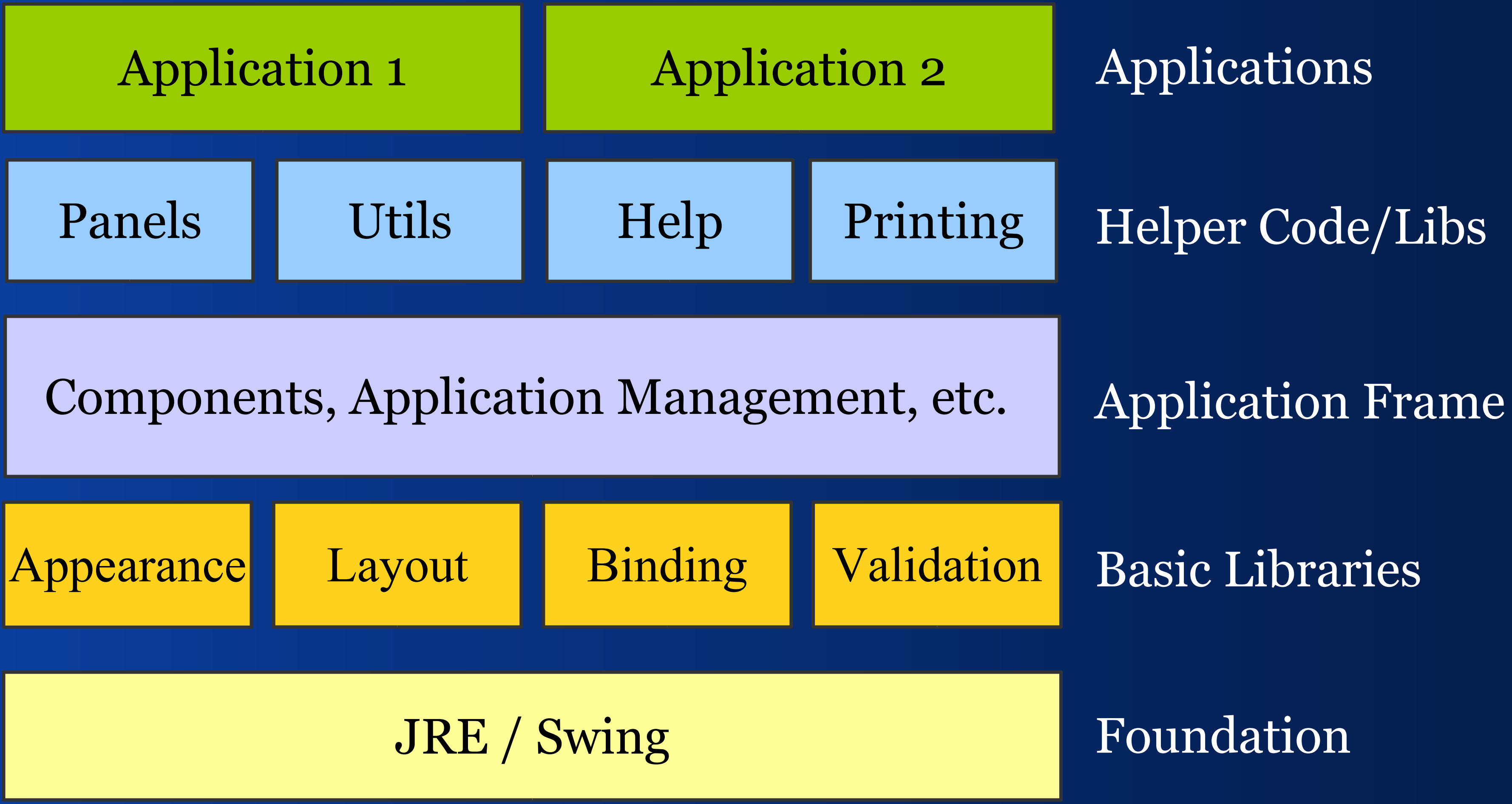
Speaker Qualifications

- Karsten builds elegant Swing apps
- works with Objects since 1990
- helps others with UI and architectures
- provides libraries that complement Swing
- provides examples for Swing architectures
- writes about Java desktop issues

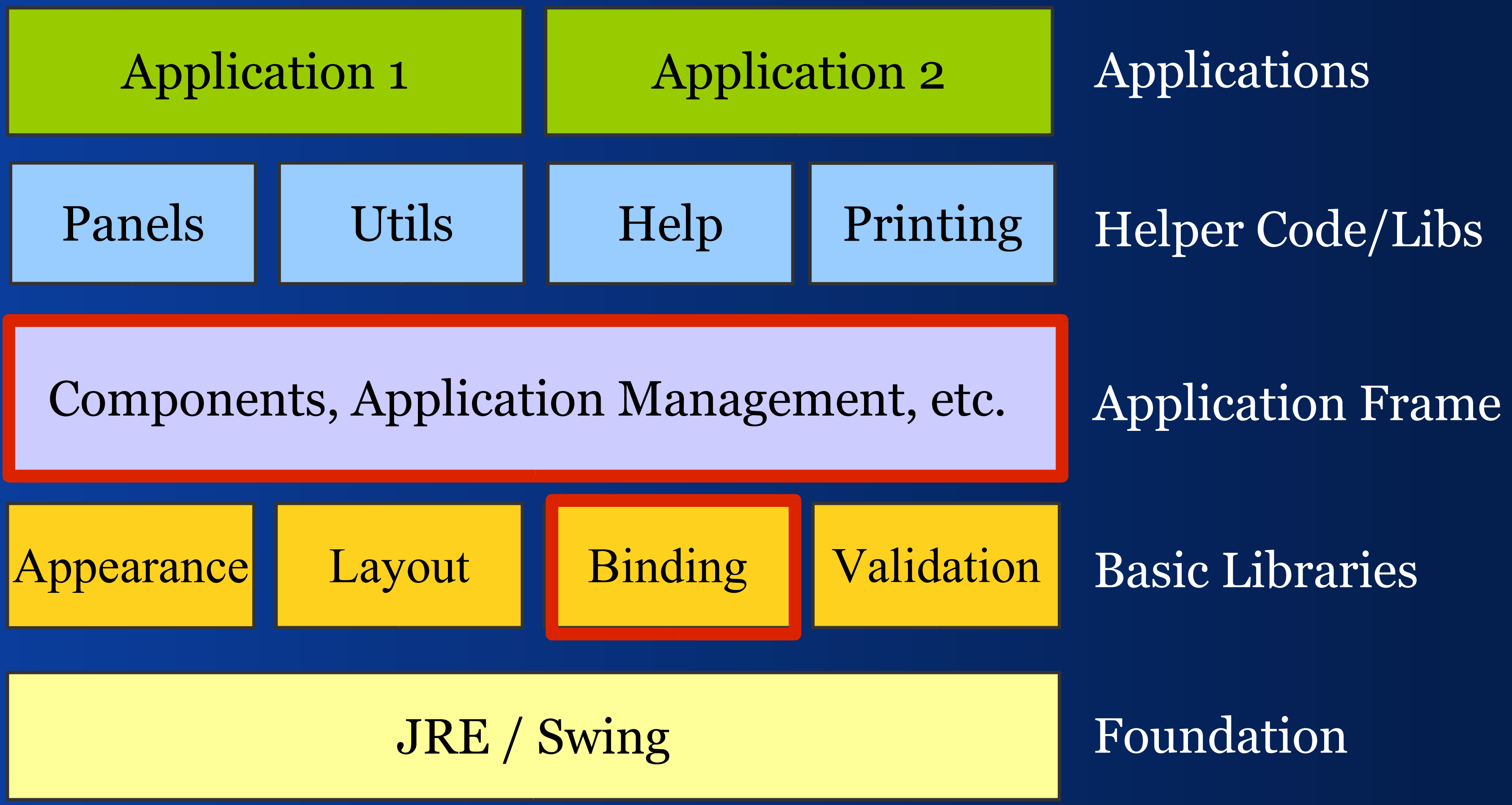
Agenda

- Introduction
- Separated Presentation & Autonomous View
- MVP, MVC and Presentation Model
- Synchronizing Single Values
- Field Report

Swing Building Blocks



Swing Building Blocks



Questions

- How and where is MVC used in Swing?
- How to structure my application?
- How to separate models?
- How to build a view?
- Who should handle events?
- Do I need a controller?

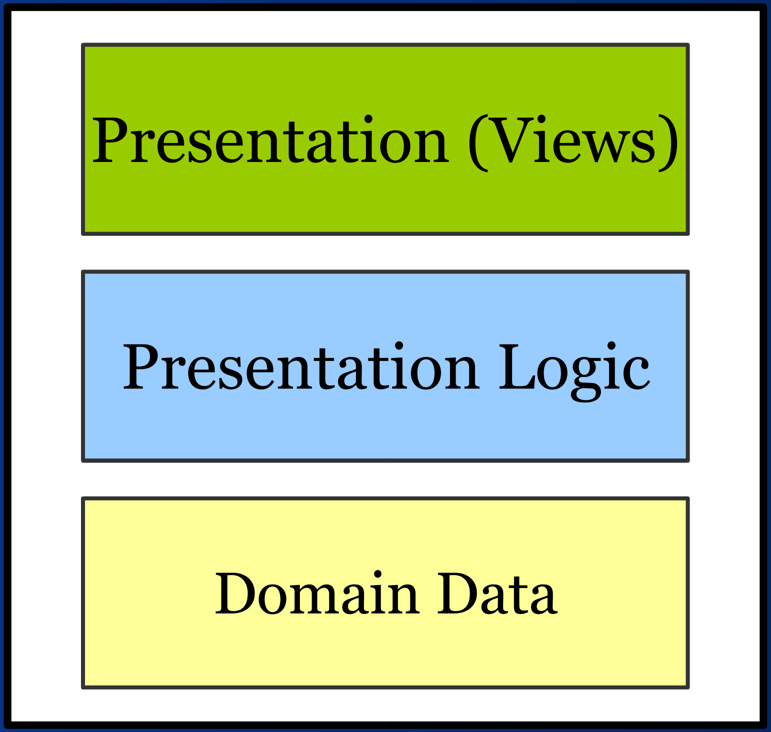
Strongly Recommended!

1. Use Separated Presentation!
2. Read “Organizing Presentation Logic” in Fowler's “Further P of EAA”
3. Study MVP and Presentation Model
4. Know Observer
5. If appropriate split Autonomous View using MVP or Presentation Model

I - Basics

Separated Presentation & Autonomous View

Not this way!

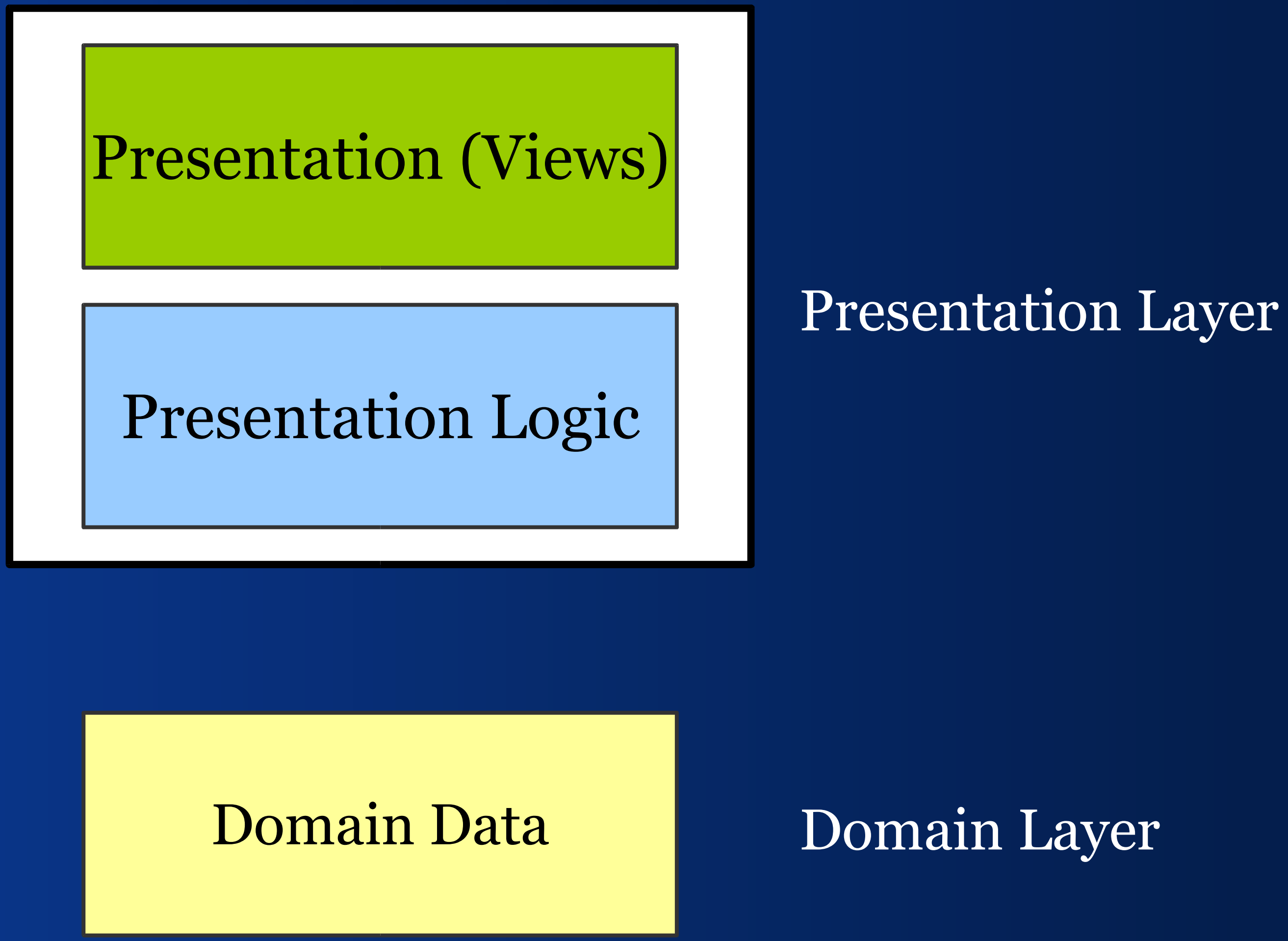


Separate Domain from Views

- Domain logic contains no GUI code
- Presentation handles all UI issues
- Advantages:
  - Each part is easier to understand
  - Each part is easier to change

- Rule of thumb for domain data:
  Do I need this class even without a GUI?

Separated Presentation

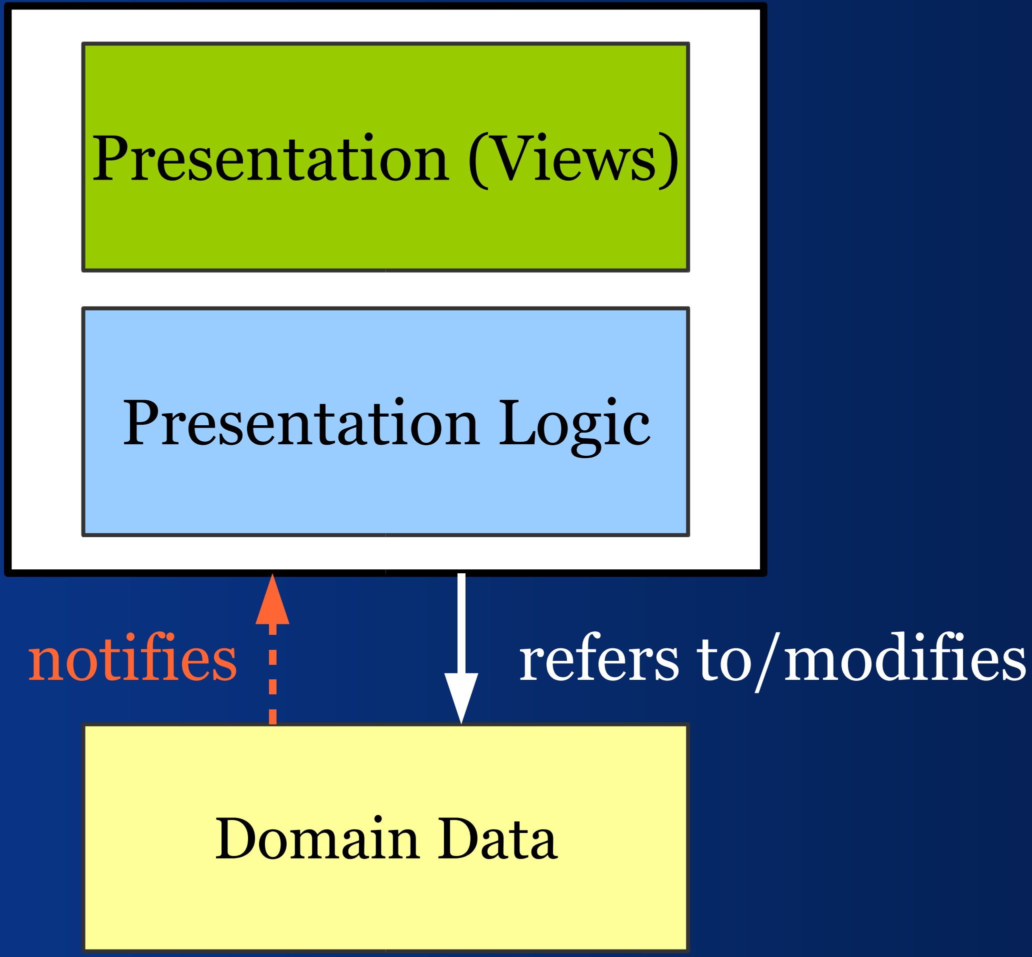


Loose Coupling

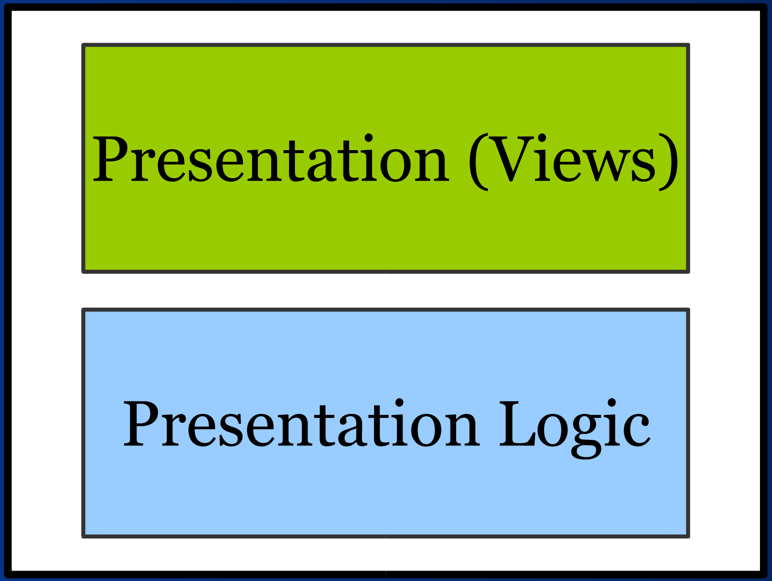
- The domain shall not reference the GUI
- Presentation refers to domain and can modify it

- Advantages:
  - Reduces complexity
  - Allows to build multiple presentations of a single domain object

Sep.Presentation with Observer



Autonomous View

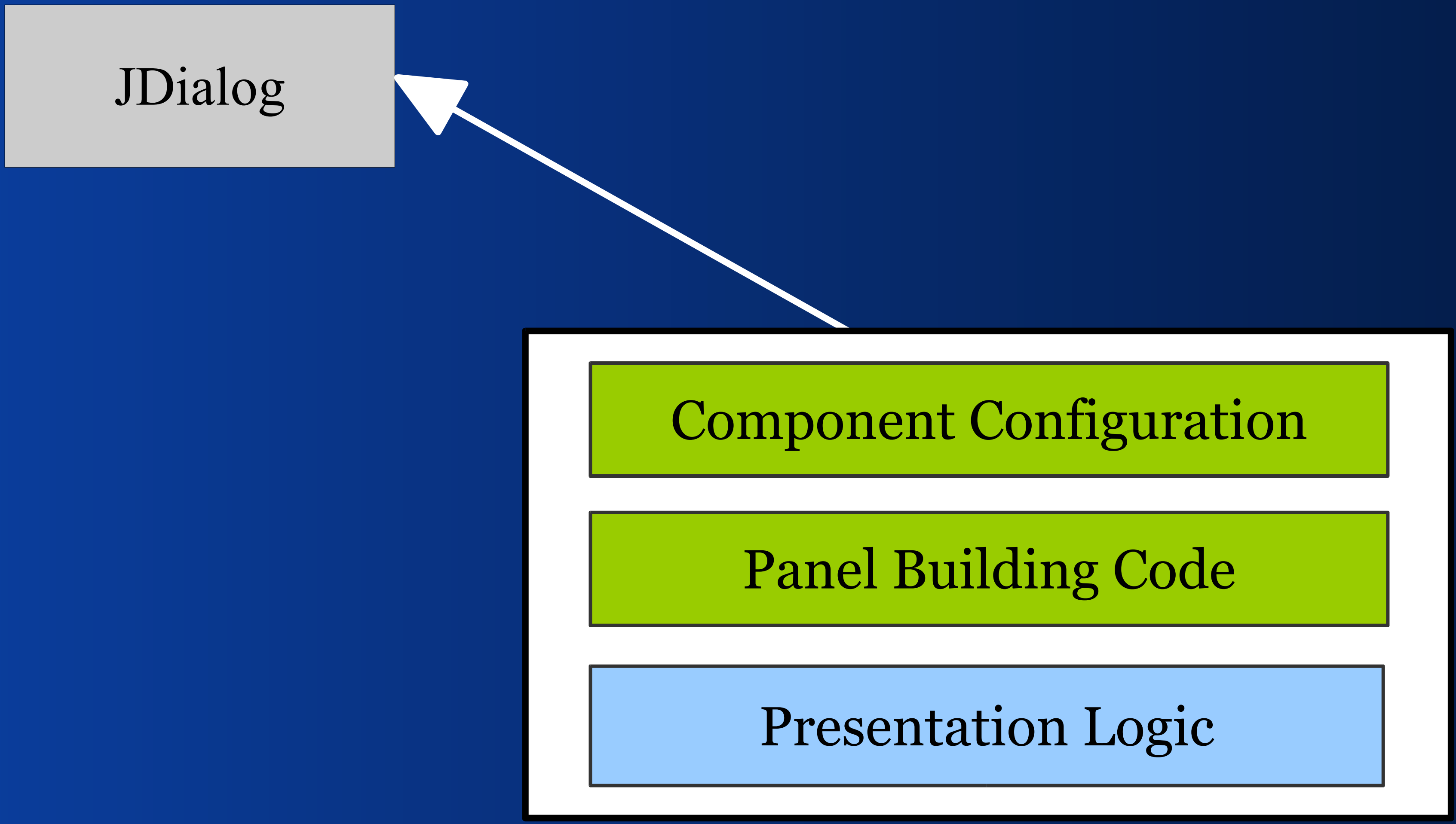


Autonomous View

- Often: one Java class per window/frame.
- Typically a subclass of JDialog, JFrame, JPanel – which isn't really necessary.

- Appropriate pattern for smaller views.
- If the views and logic get more complex, it's worth to separate concerns.

Autononous View: Details



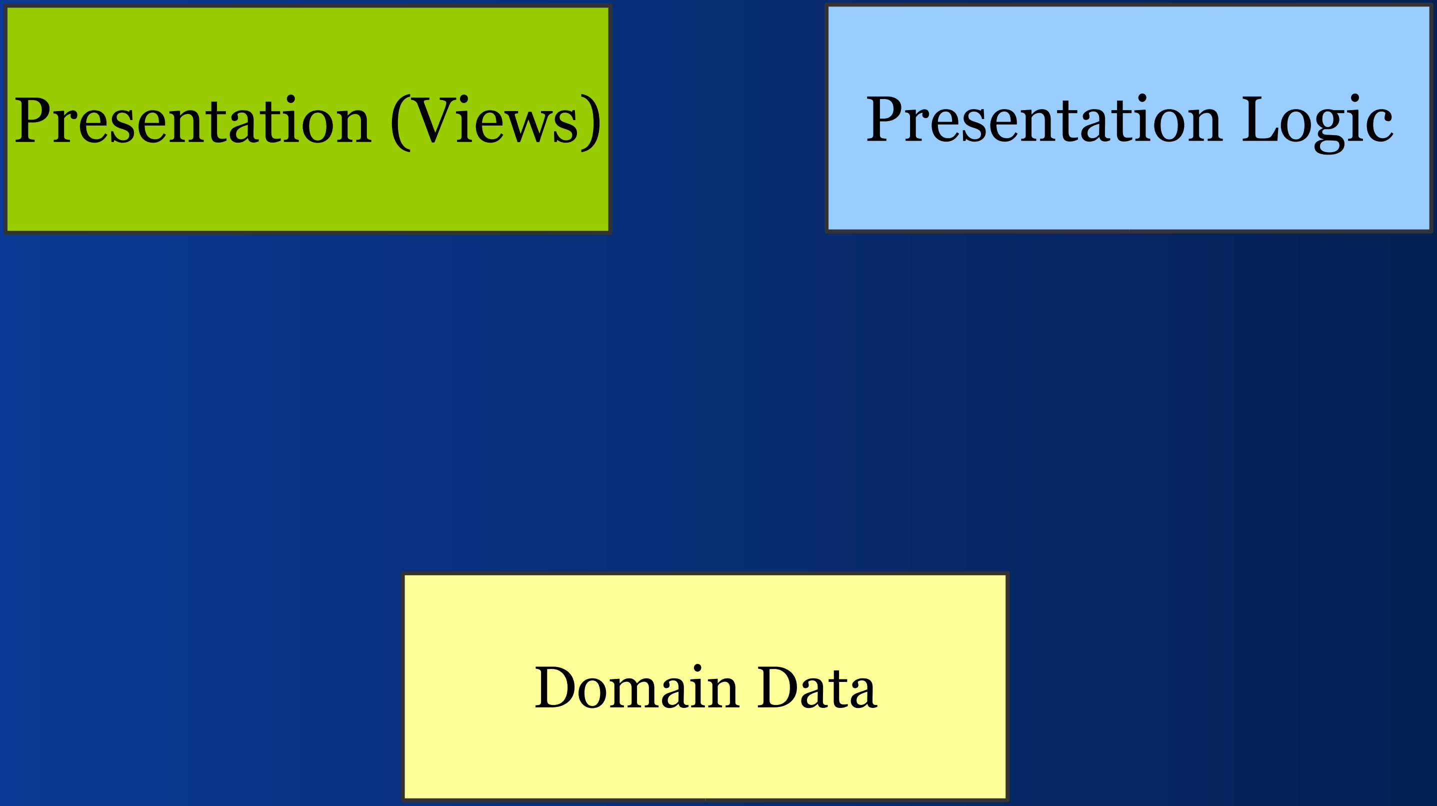
Tips

- Build dialogs, frames, panel; extend them only if necessary.
- Compose larger screens from small panels.
  - in simple cases use build methods like #buildMainPanel,#buildButtonBar, etc.
  - otherwise use panels and nest subpanels.
- Consider separating the presentation logic from the presentation.

When to Split Autonomous View?

- If you want to test the presentation logic.
- If you don't overview the source anymore, for example because it exceeds your outline.
- If you share code with colleagues.
- If you want to reuse the logic or views.

Presentation Logic Separated



Advantages of the Separation I

- Makes testing easier (Fowler).
- GUI layer becomes quite simple, and is easy to build, to understand, and to maintain.
- More team members can work on the GUI.
- GUI code can follow syntactical patterns.
- Makes it easier to work with visual builders.

Advantages of the Separation II

- The complex logic is easier to overview.
- The separation helps us structure our work.
- Simplifies team synchronisation.
- Allows to build “forbidden zones”
  - for team members
  - before you ship a new release

Disadvantages of the Separation

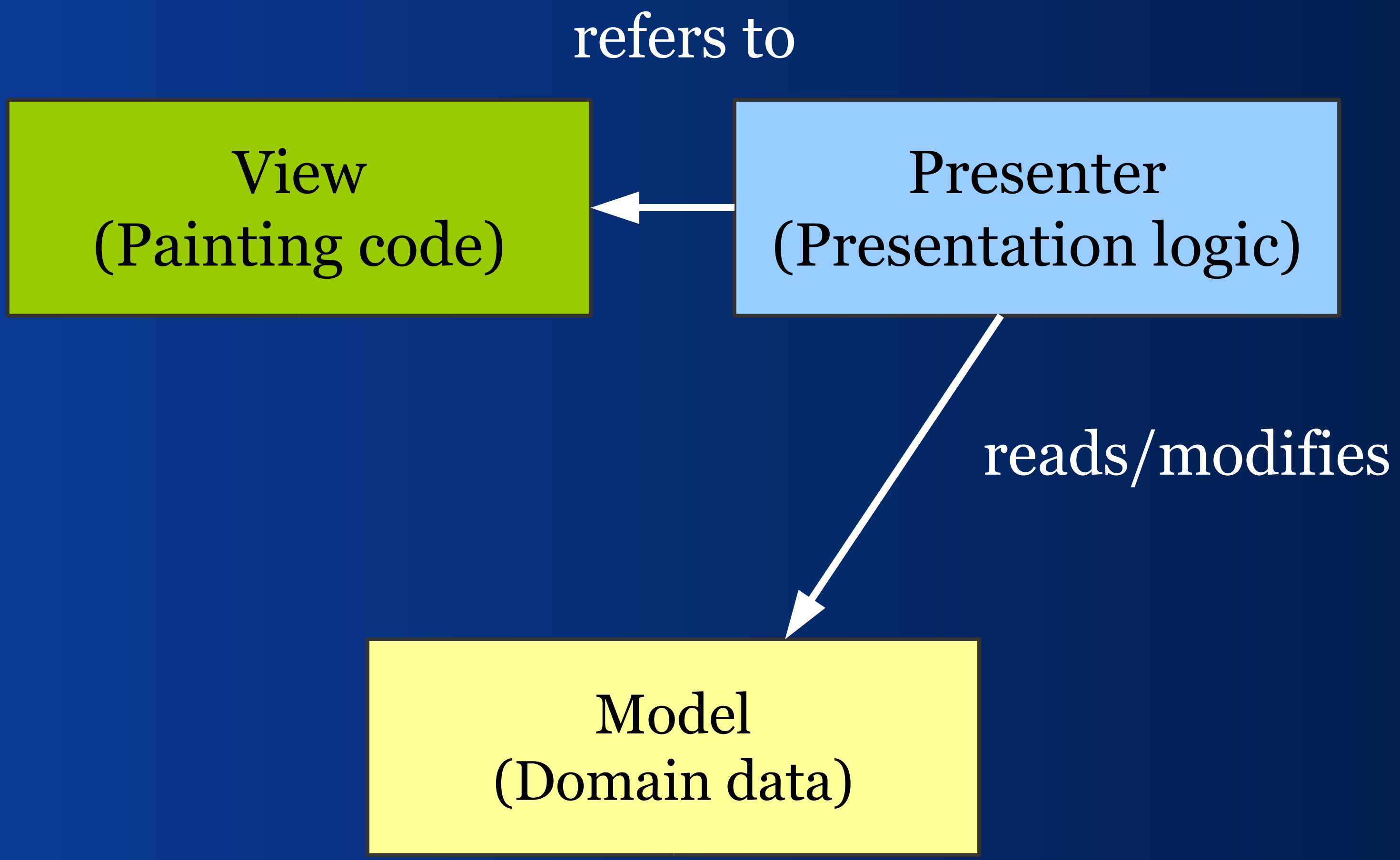
- More work.
- Requires to work with a set of related classes instead of a single class.

Typically you benefit from the separation.

II – Splitting Autonomous View

MVP, MVC and Presentation Model

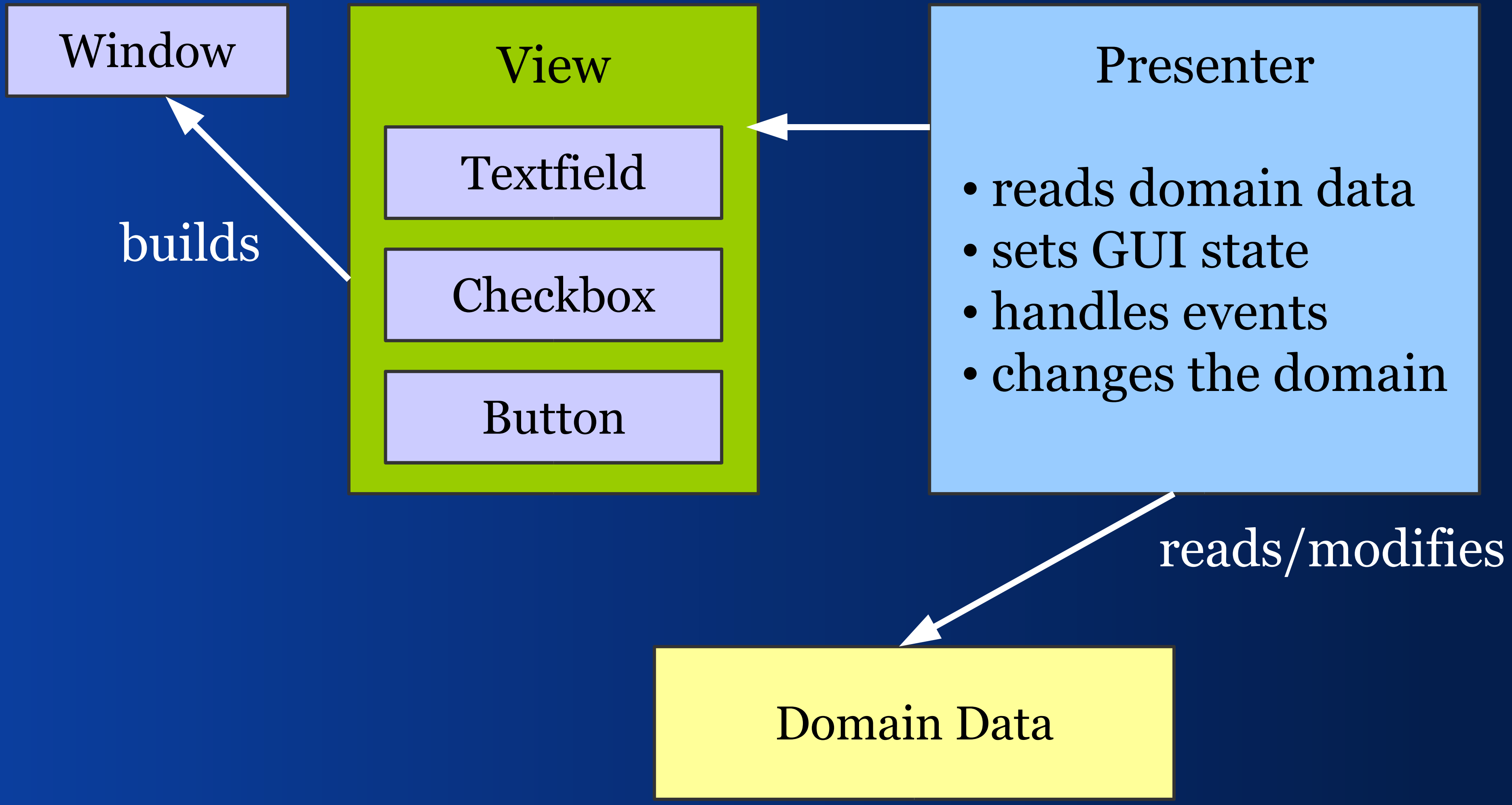
Model-View-Presenter (MVP)



MVP

- The View
  - holds the GUI state, for example a JTextField with Text and Enablement
- The Presenter
  - reads domain data and copies them to the components of the views
  - handles GUI events and modifies the GUI state in the view
  - modifies domain data using GUI data

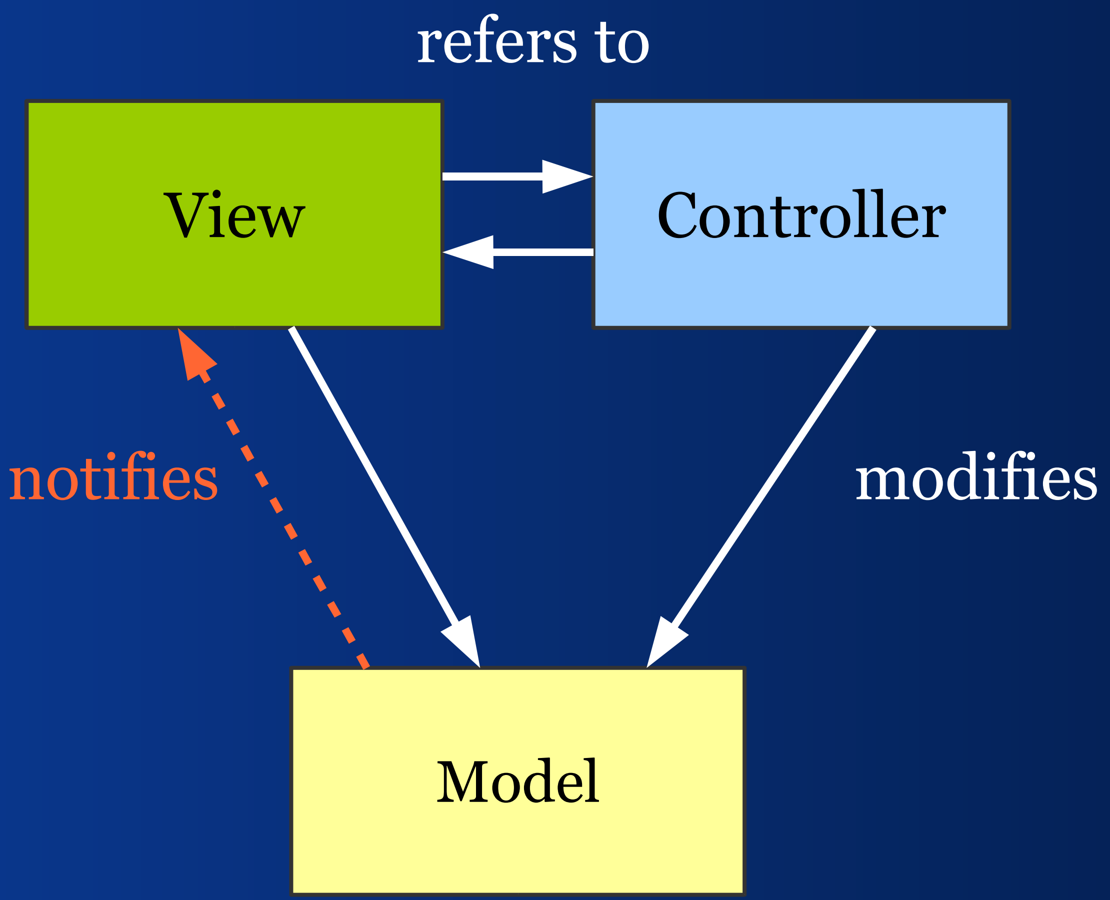
MVP


Window
builds
View
Textfield
Checkbox
Button
Presenter
• reads domain data
• sets GUI state
• handles events
• changes the domain
reads/modifies
Domain Data

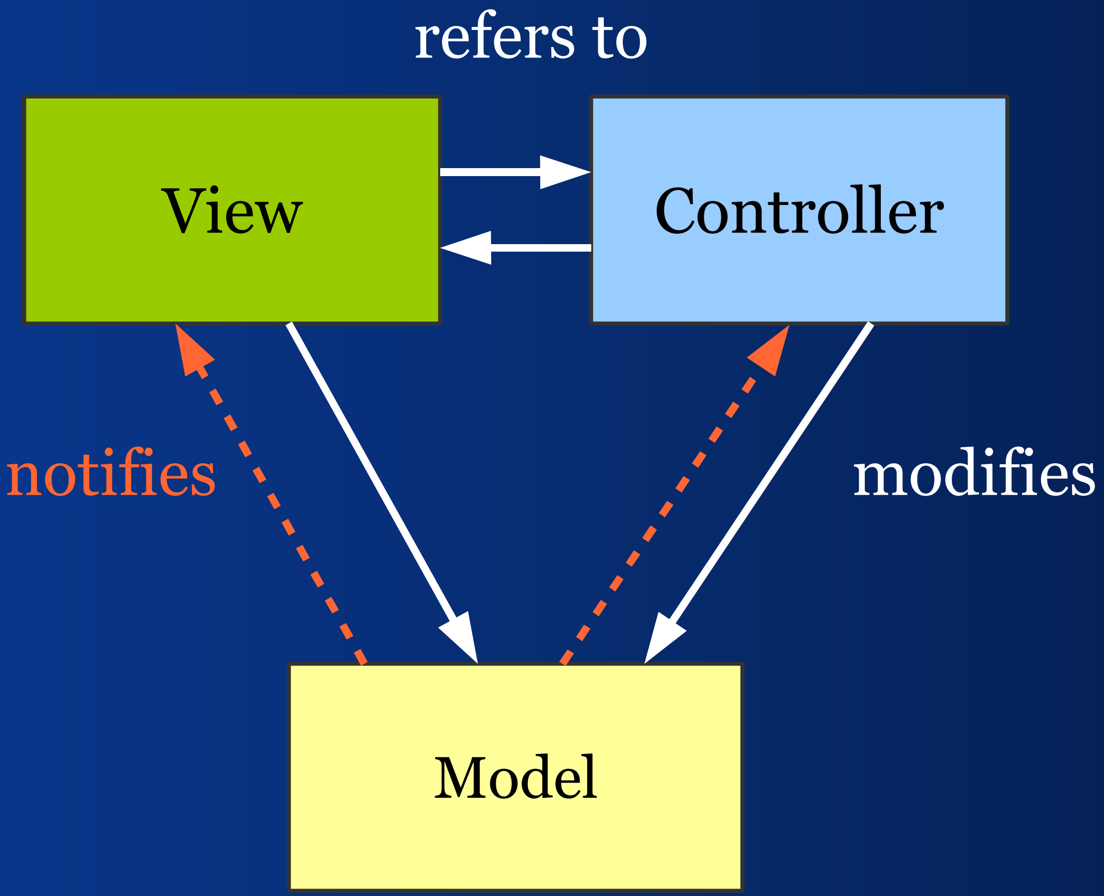
MVP vs. MVC

Differences and the Swing-MVC-Variant

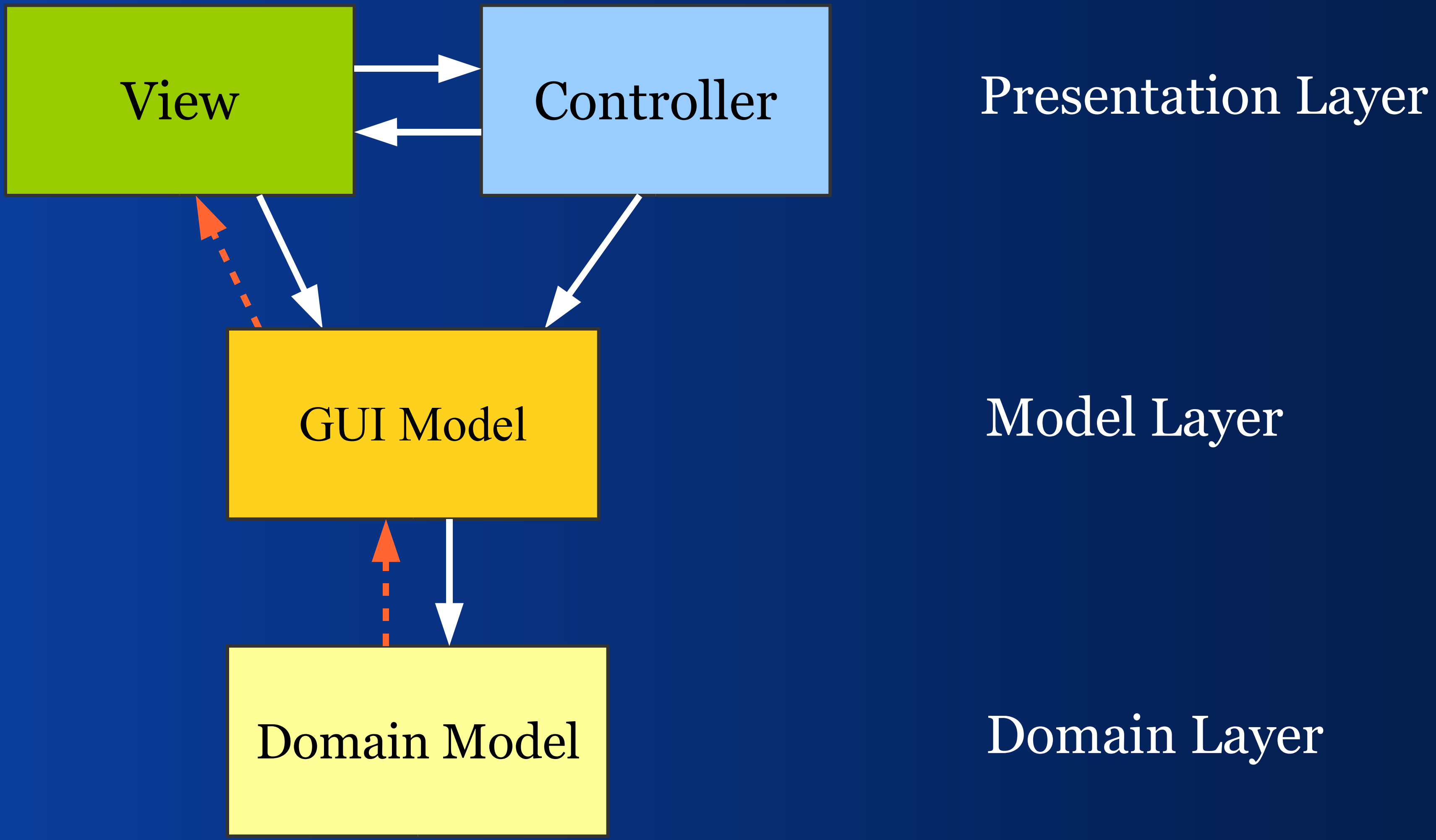
MVC



MVC



MVC with Model Layer

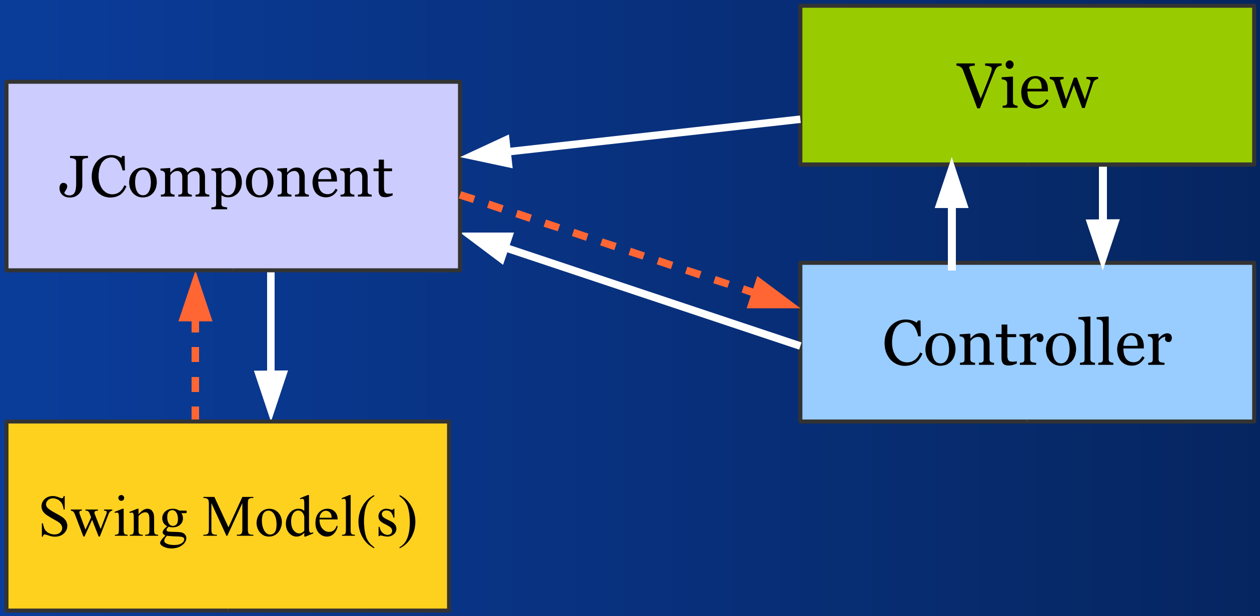


Factoring out the Look&Feel

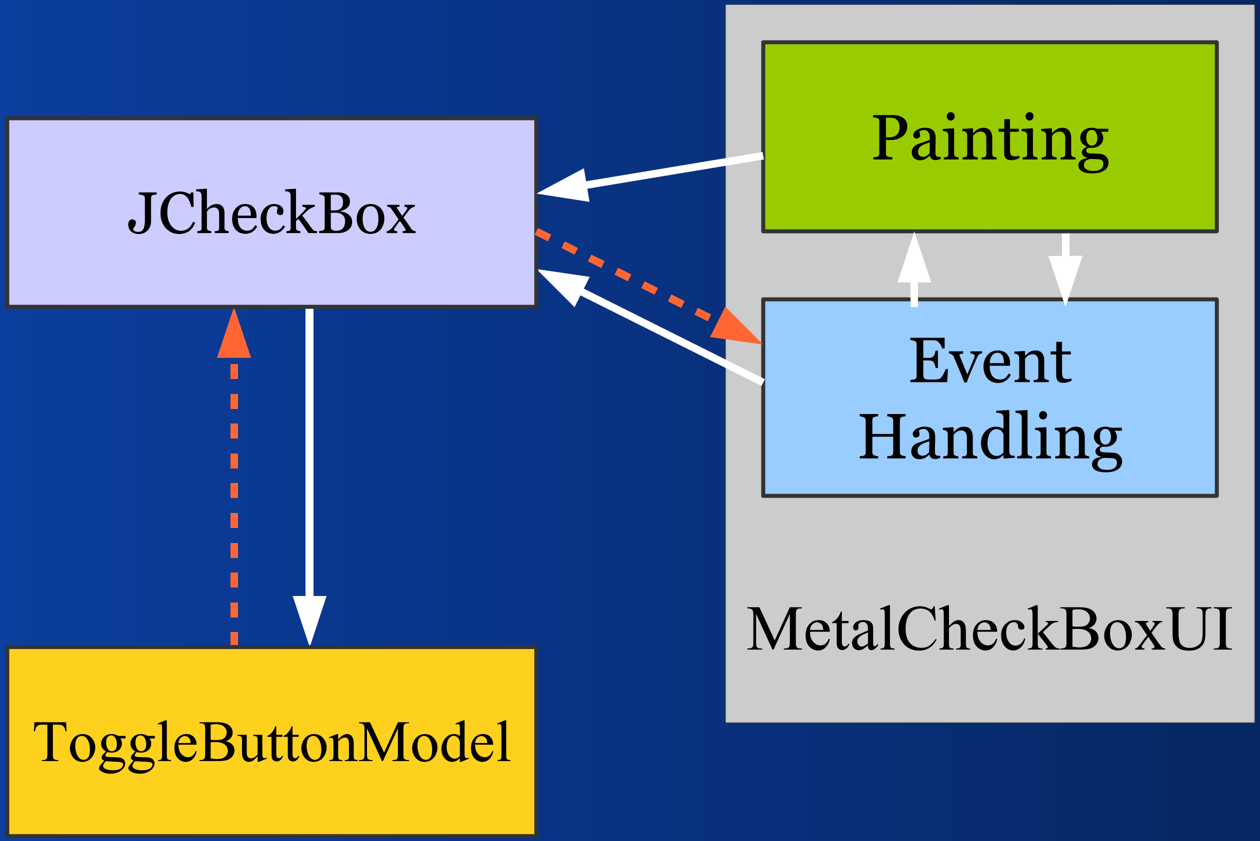
Swing can change the application's appearance and behavior (look & feel).

- Views and Controller are separated from the UI components and are put together as a UI Delegate.

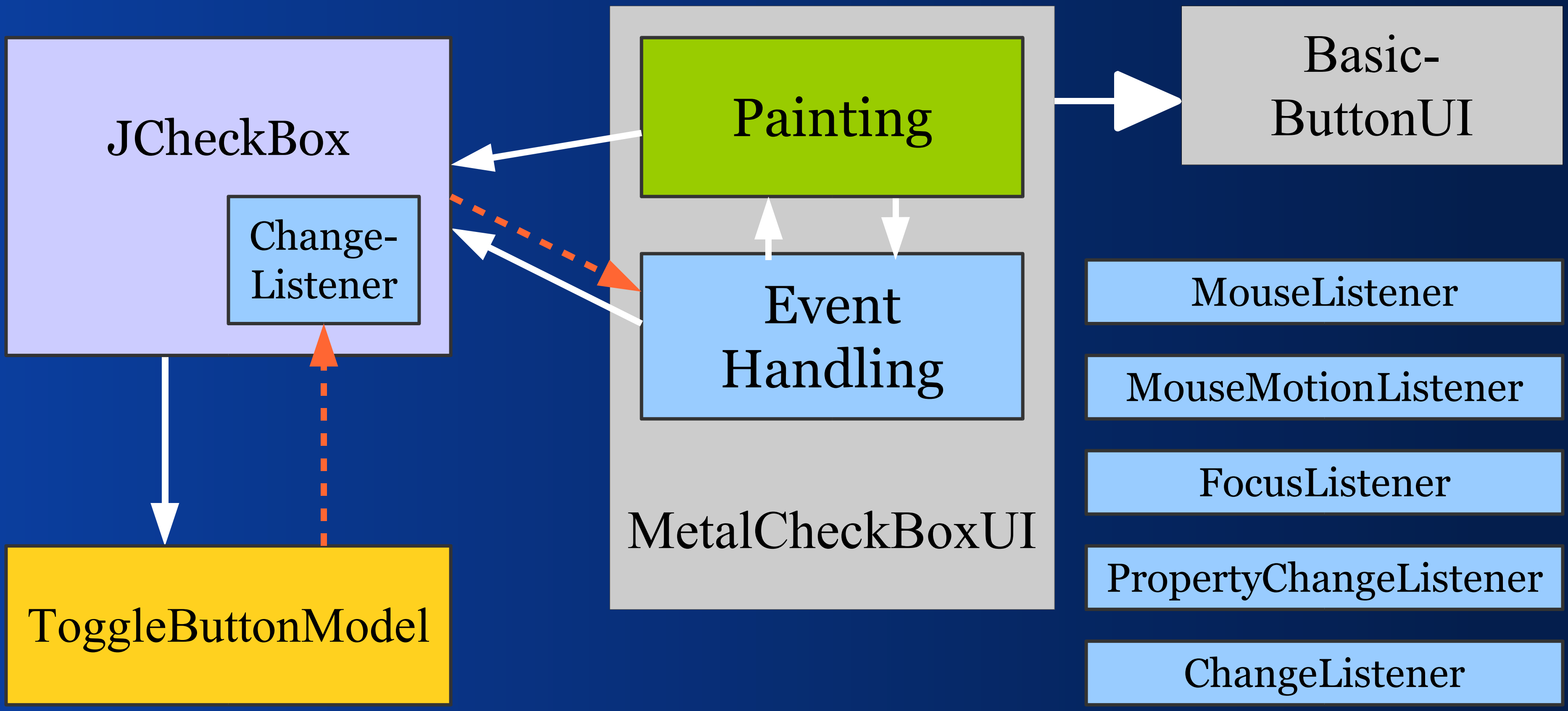
M-JComponent-VC



Example: JCheckBox



JCheckBox: Some Details

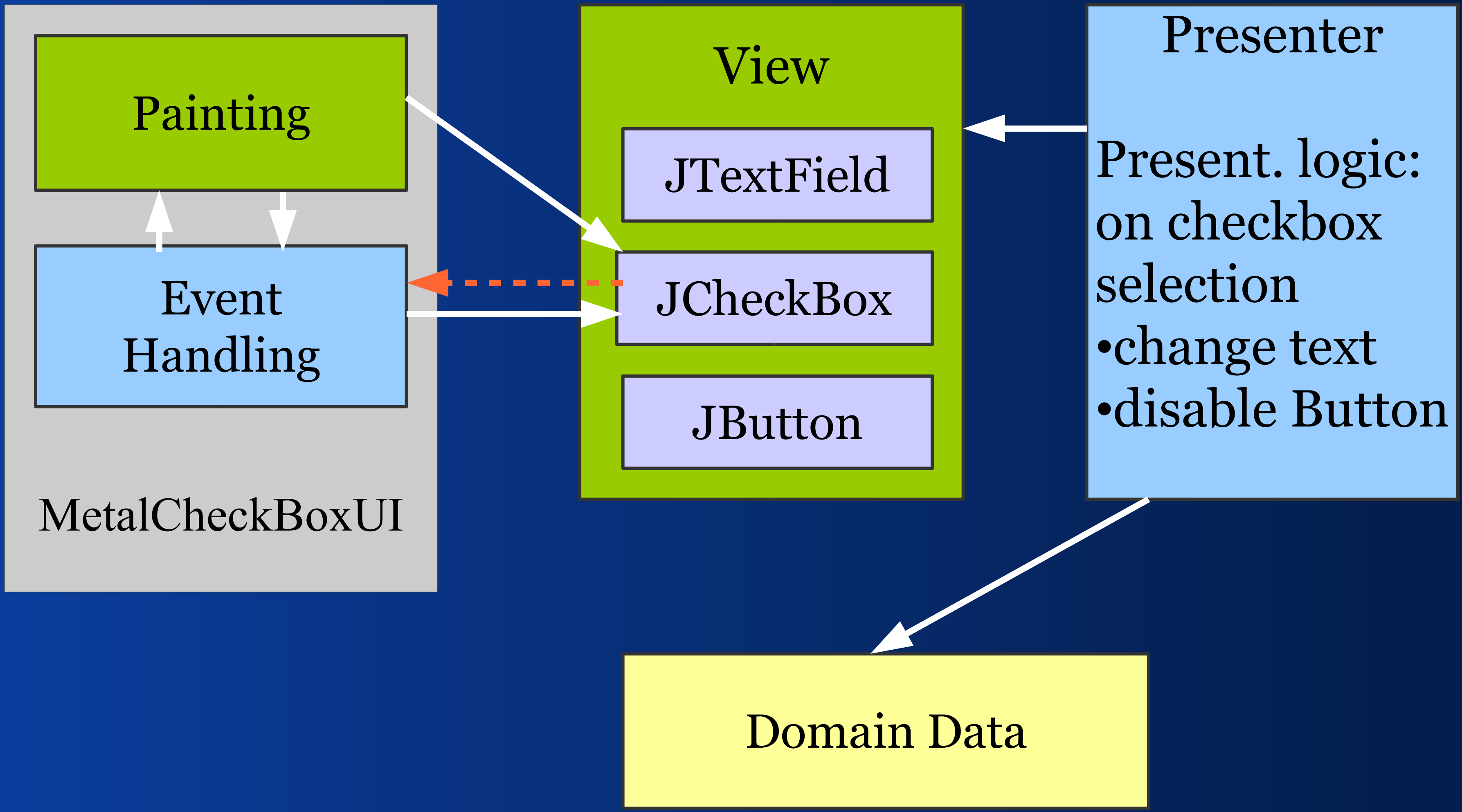


Summary

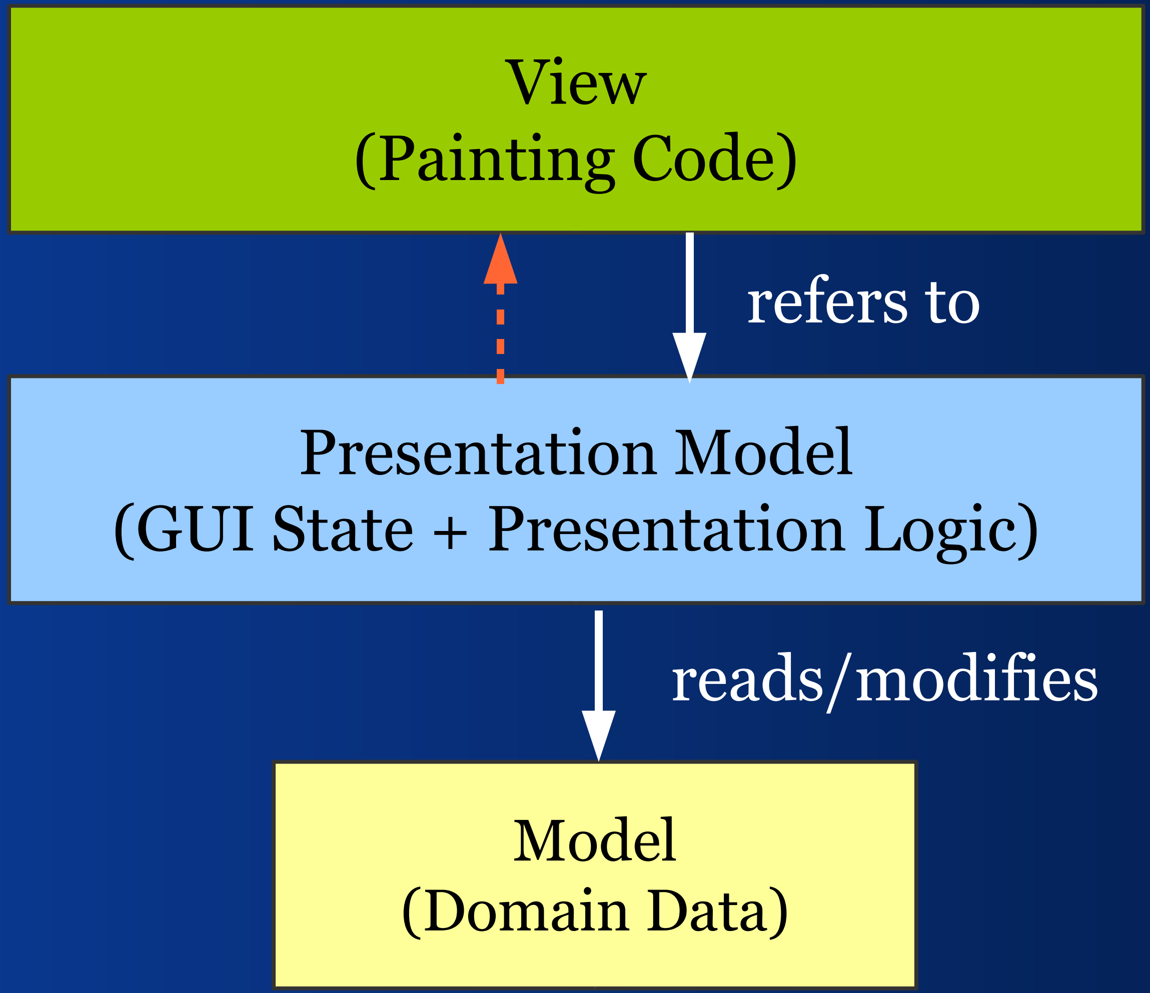
- Swing doesn't use the original MVC
- Swing uses an extended form of MVC
- Swing shares the motivation behind MVC
- Swing adds features to the original MVC
- UI delegates are both view and controller

- MVC is for components,
  MVP for applications

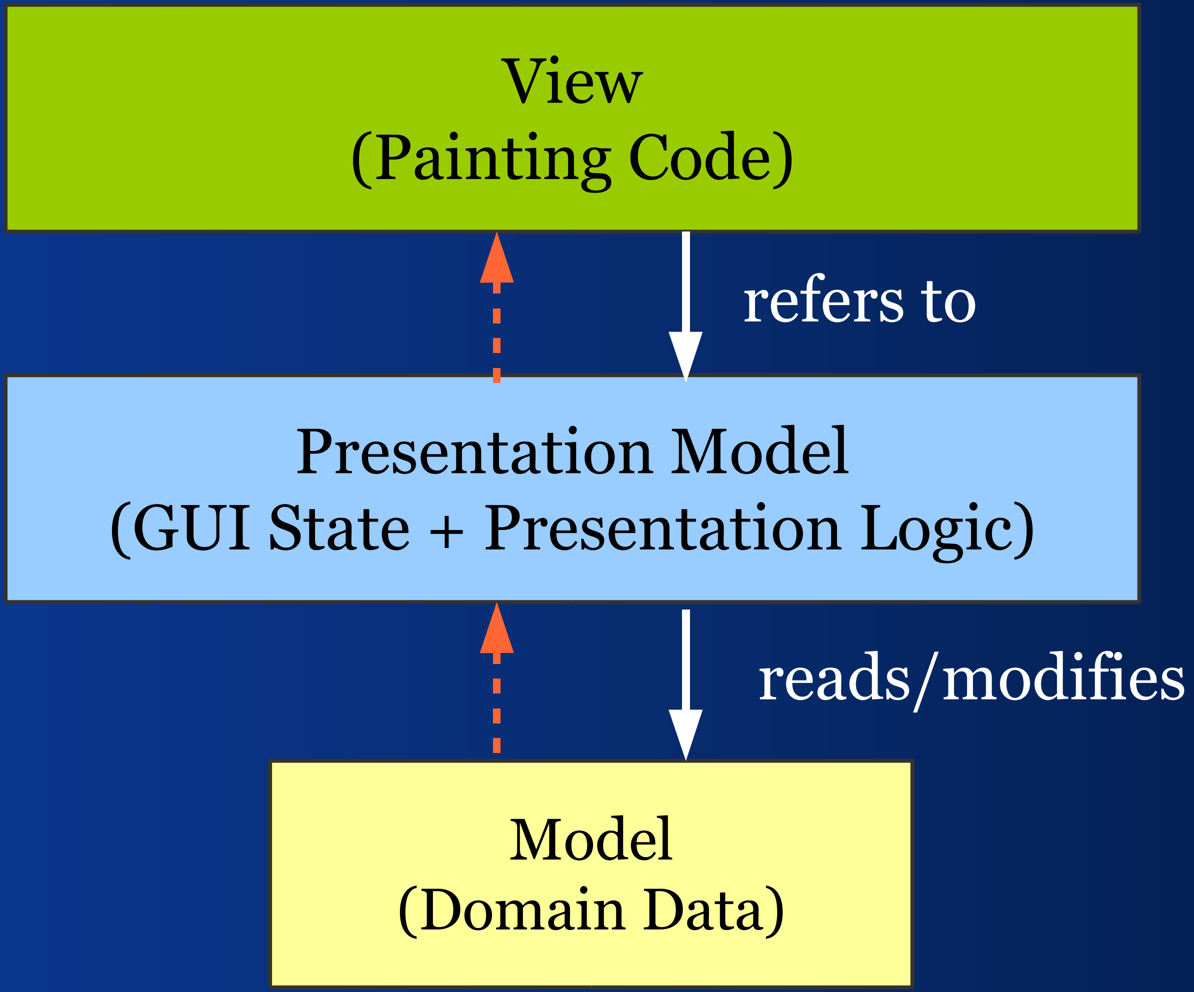
MVP in Swing


Painting
Event Handling
MetalCheckBoxUI
View
JTextField
JCheckBox
JButton
Presenter
Present. logic: on checkbox selection
•change text
•disable Button
Domain Data

Presentation Model (PM)



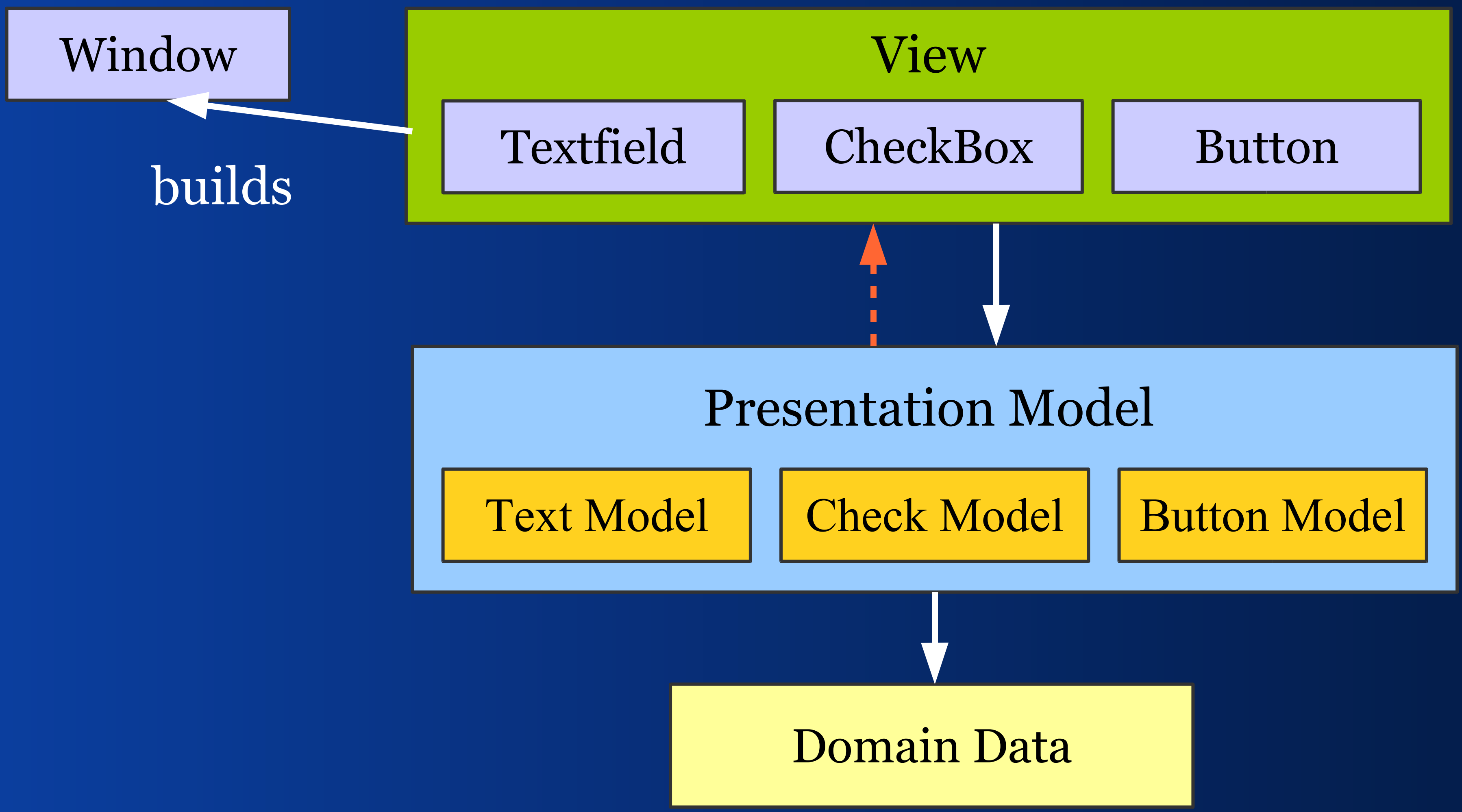
Presentation Model (PM)



Presentation Model

- The View
  - – consists only of GUI components
  - – observes changes in the Presentation Model
- The Presentation Model
  - – contains GUI state and presentation logic
  - – reads domain to update its GUI state
  - – handles GUI events by changing its GUI state; then reports changes
  - – modifies domain data using its GUI state

Presentation Model

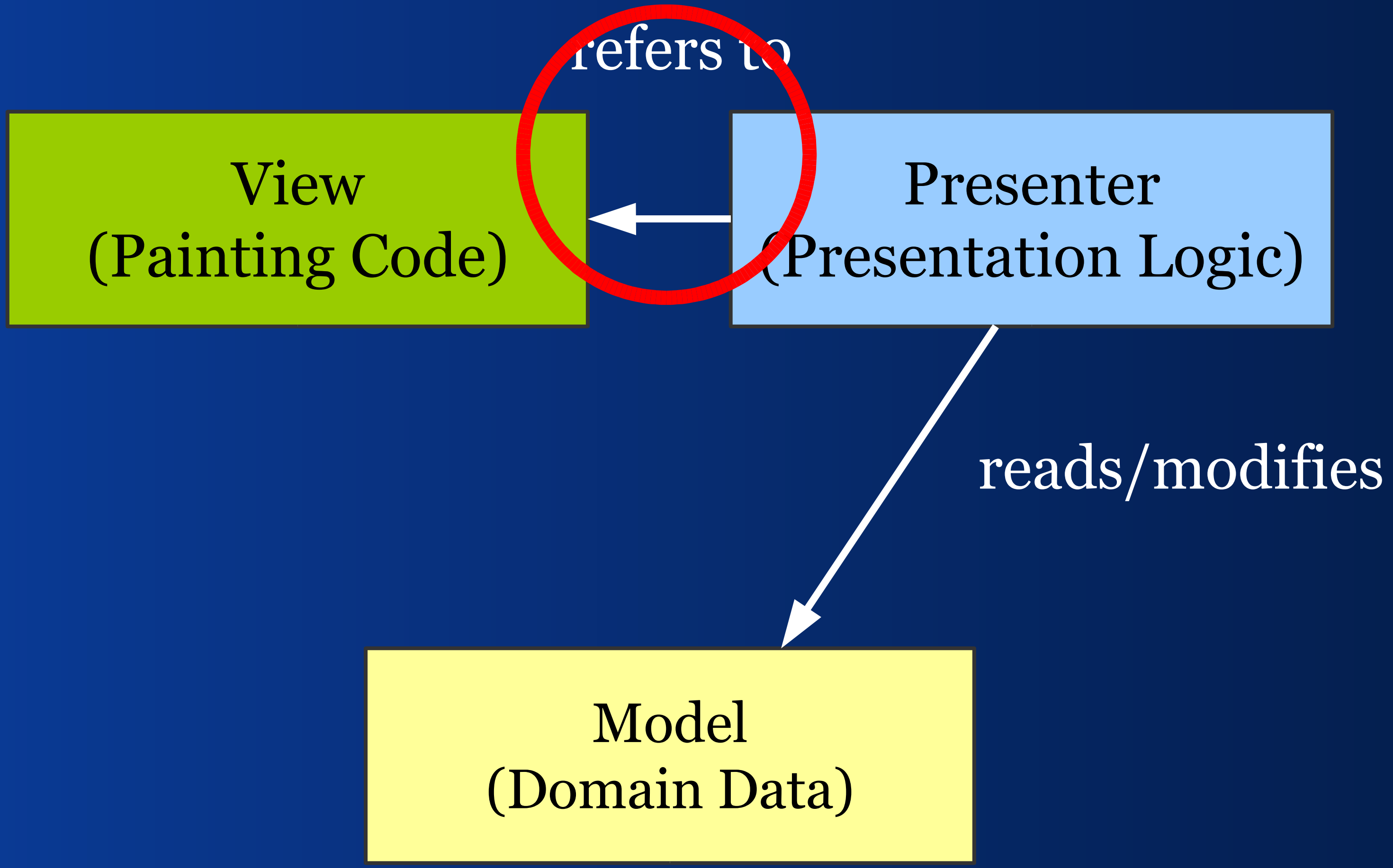


MVP vs. Presentation Model

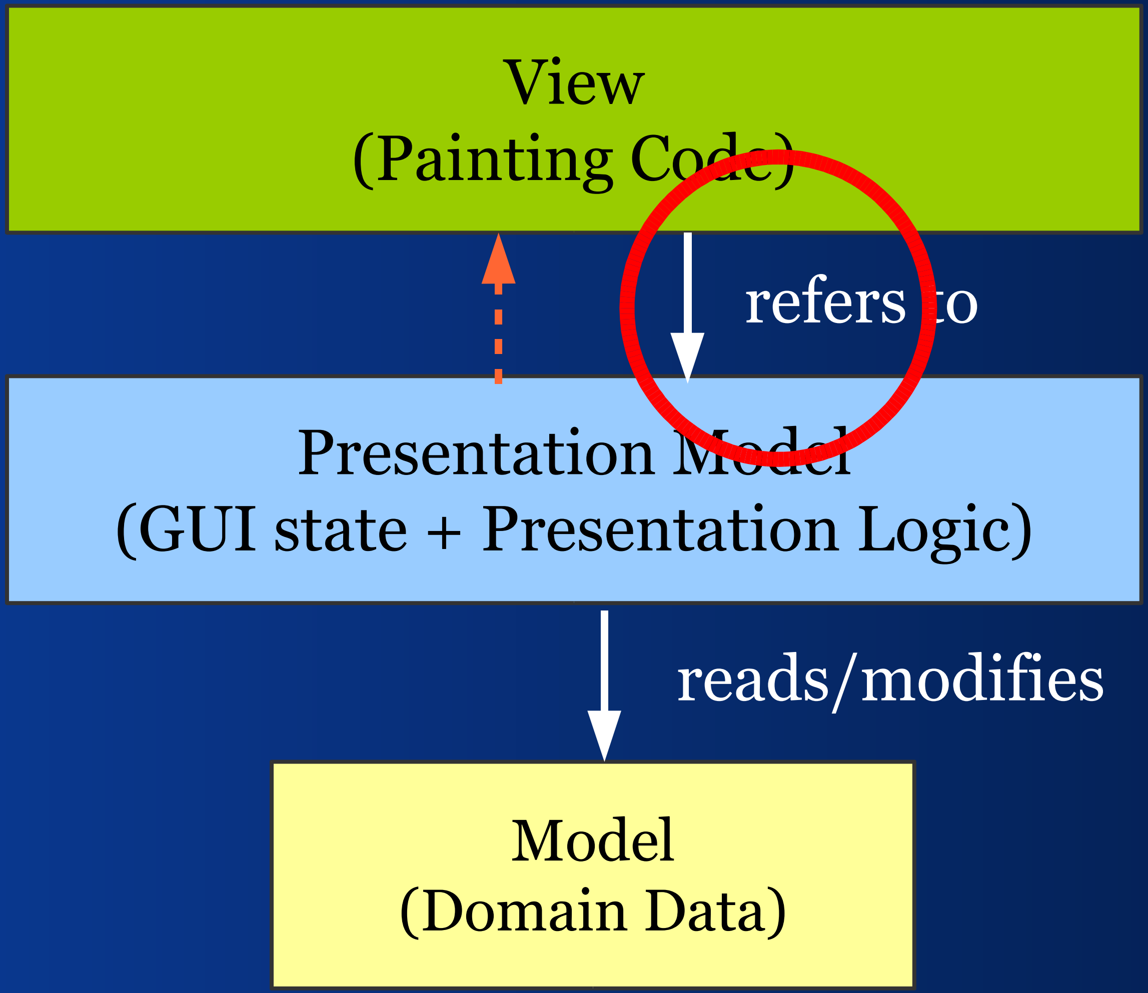
- Presenter refers to the View, PM does not refer (directly) to the View.
- In MVP the View holds the GUI state.
- The PM holds the GUI state itself.

- The Presenter changes GUI state in the View
- The PM changes its own GUI state and reports changes to all its View.

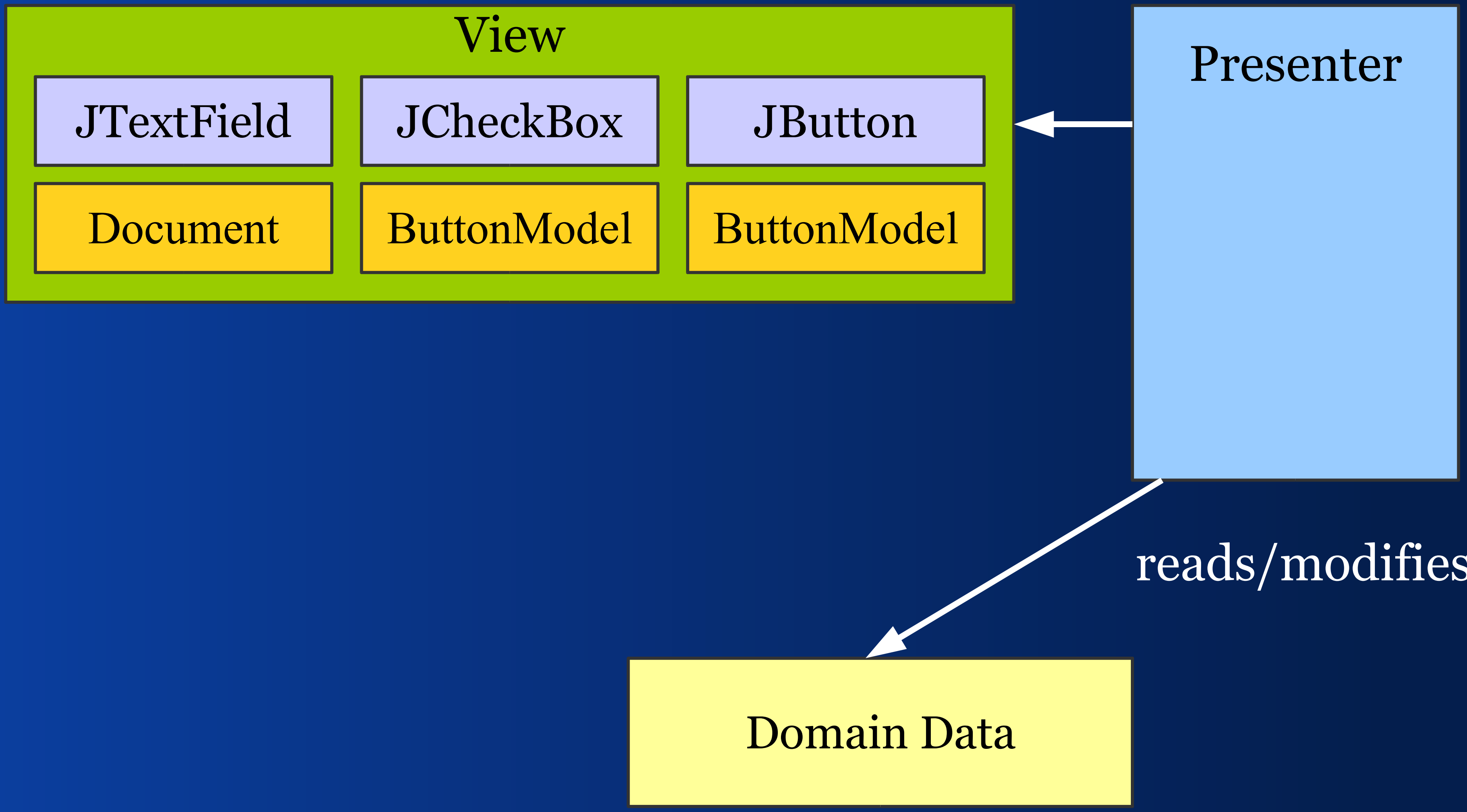
MVP


refers to
View
(Painting Code)
Presenter
(Presentation Logic)
reads/modifies
Model
(Domain Data)

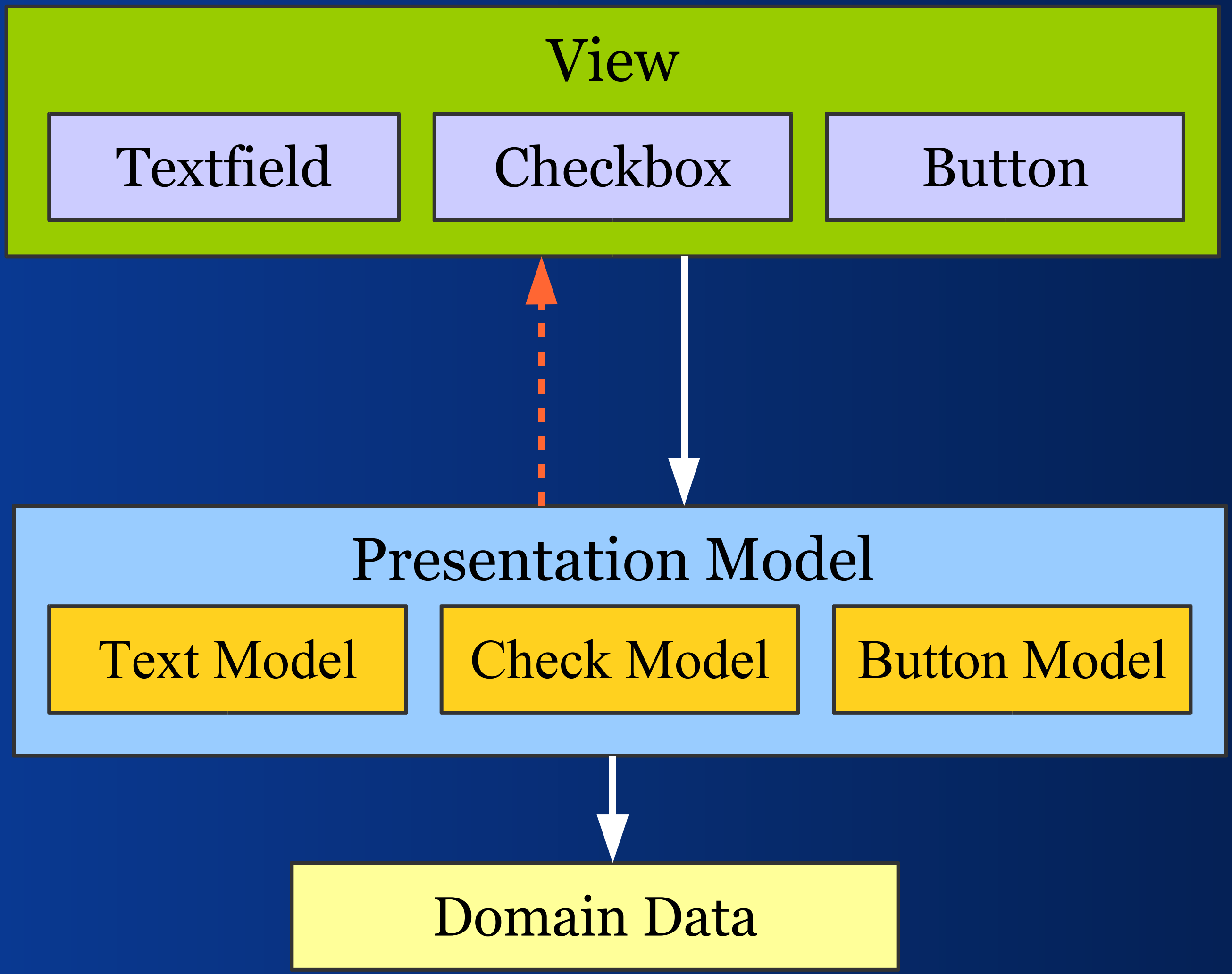
Presentation Model



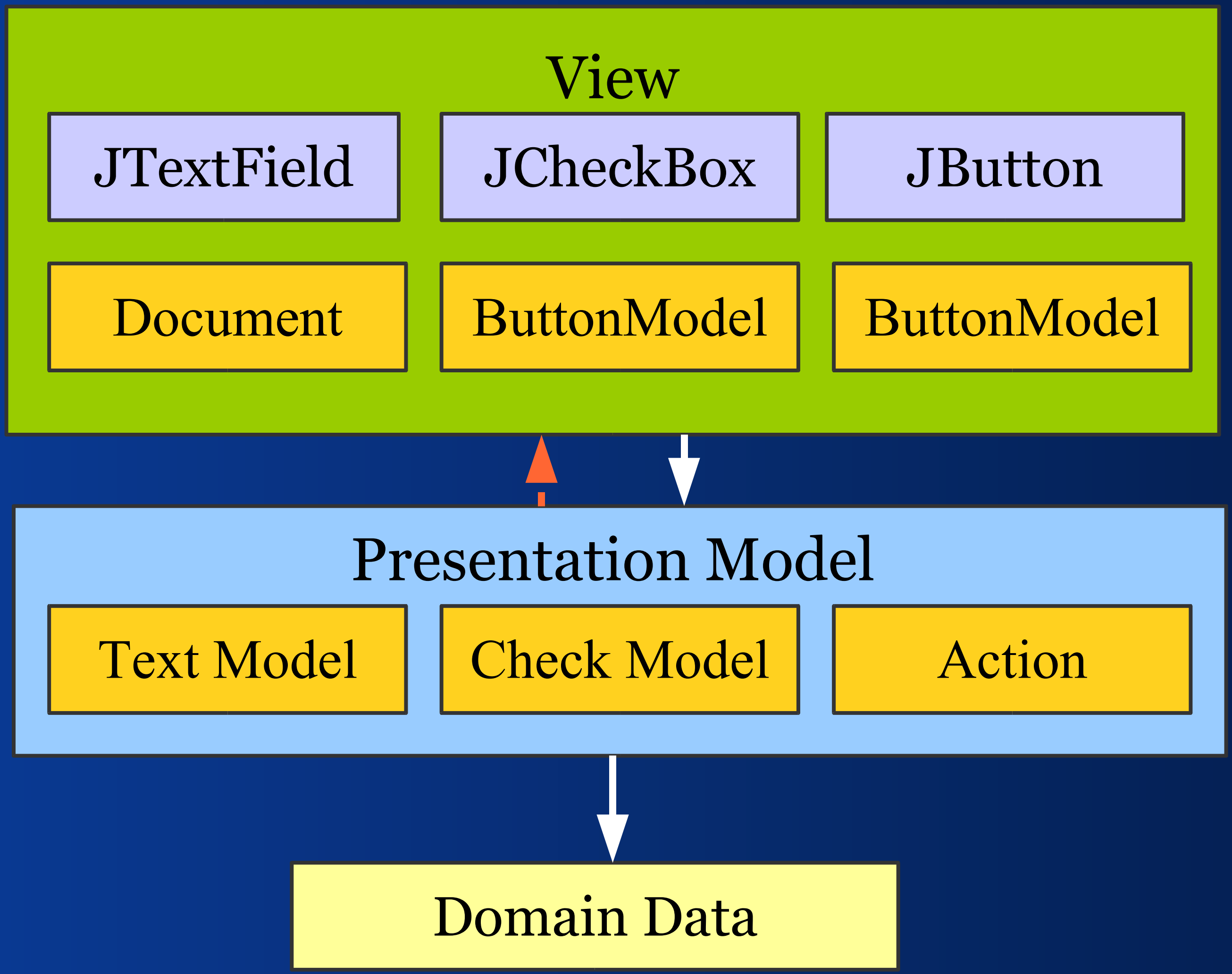
MVP in Swing: GUI State



PM: GUI State


View
Textfield
Checkbox
Button
Presentation Model
Text Model
Check Model
Button Model
Domain Data

PM in Swing: GUI State

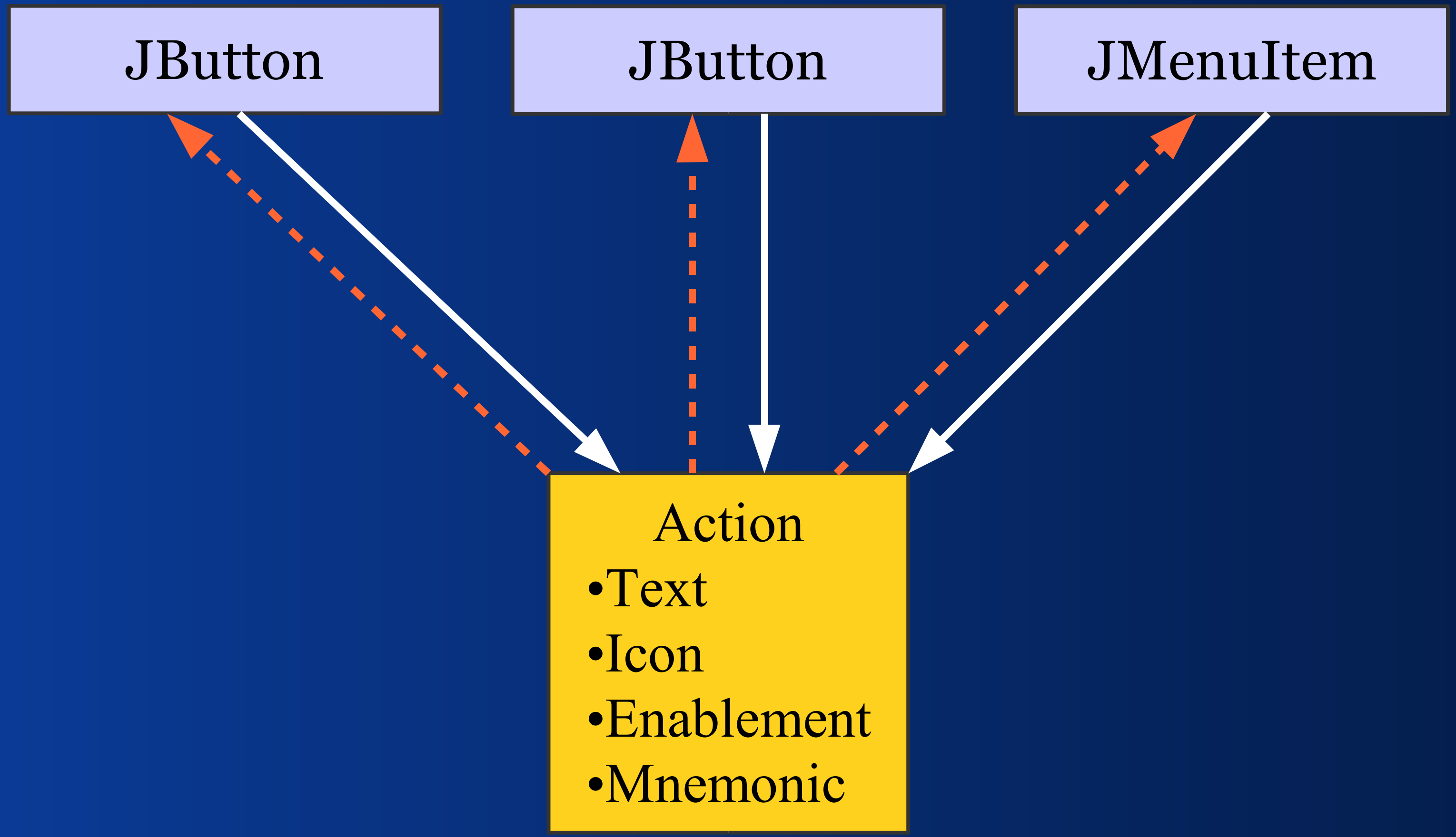


MVP vs. Presentation Model

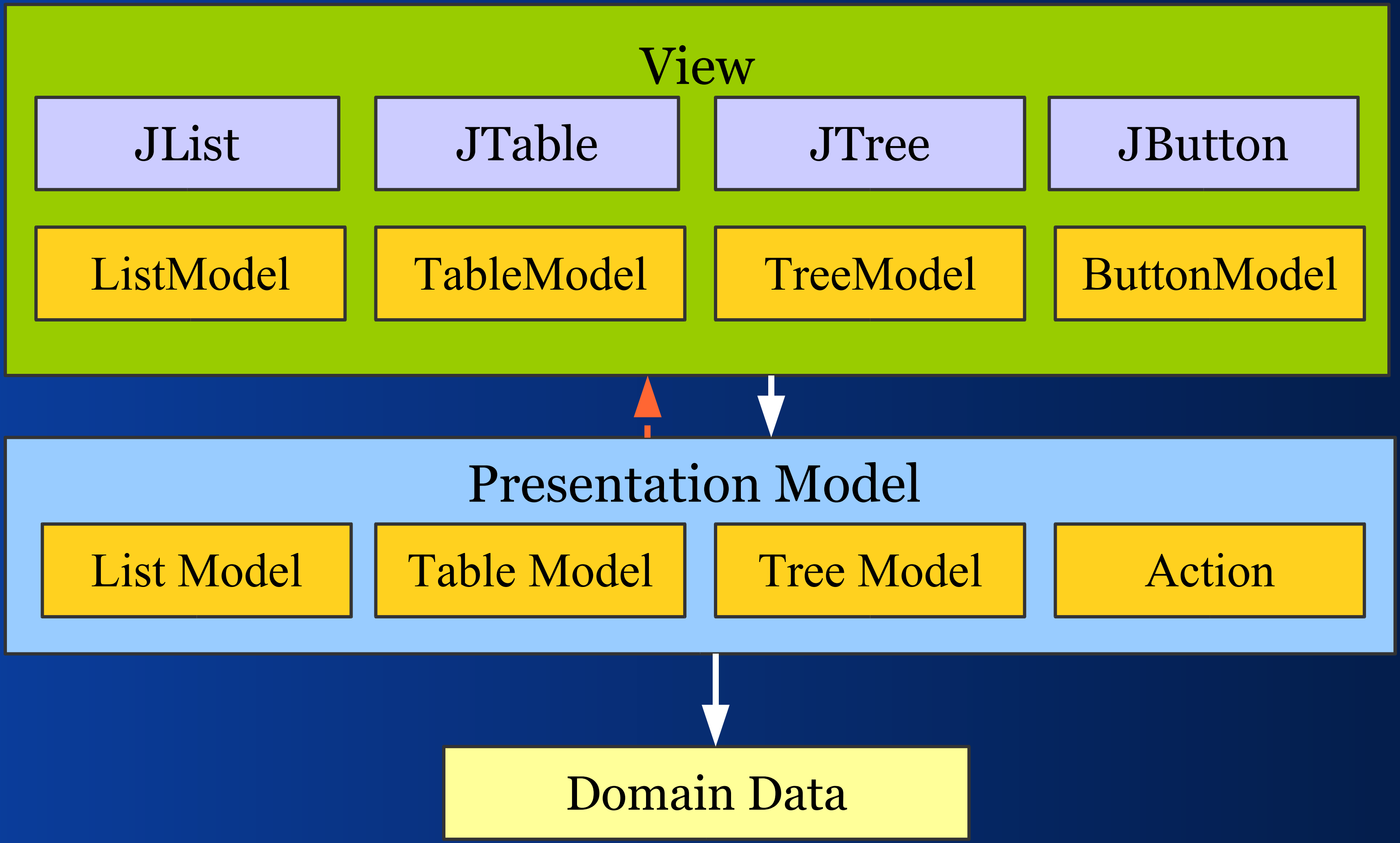
- MVP holds the GUI state once.
- PM holds it twice, in the View and the PM.
- PM requires a synchronisation between the PM state and the View state.

- No worries about this synchronisation! The Swing architecture supports this well.

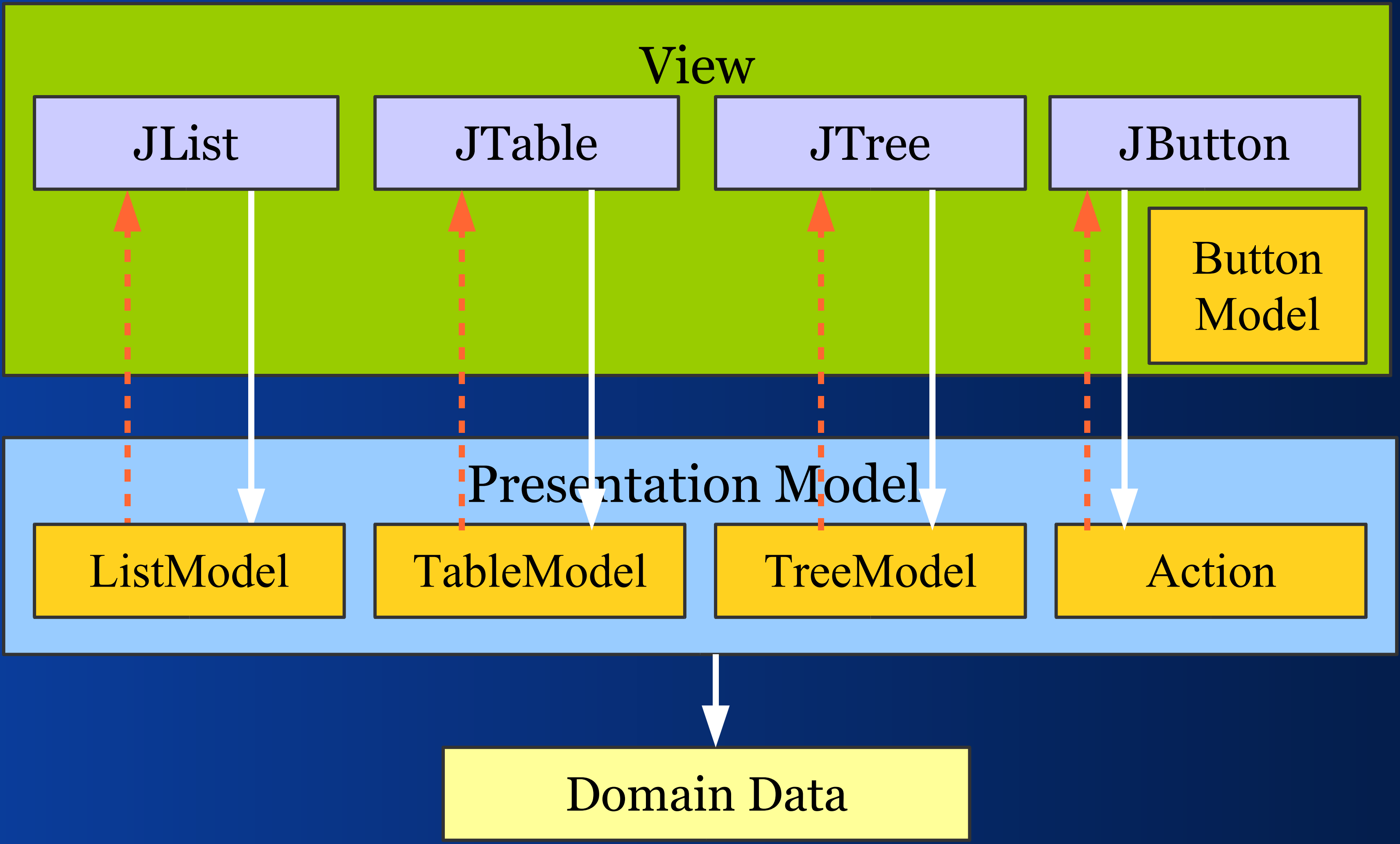
Reminder: Swing Actions



PM: Lists, Tables, Trees



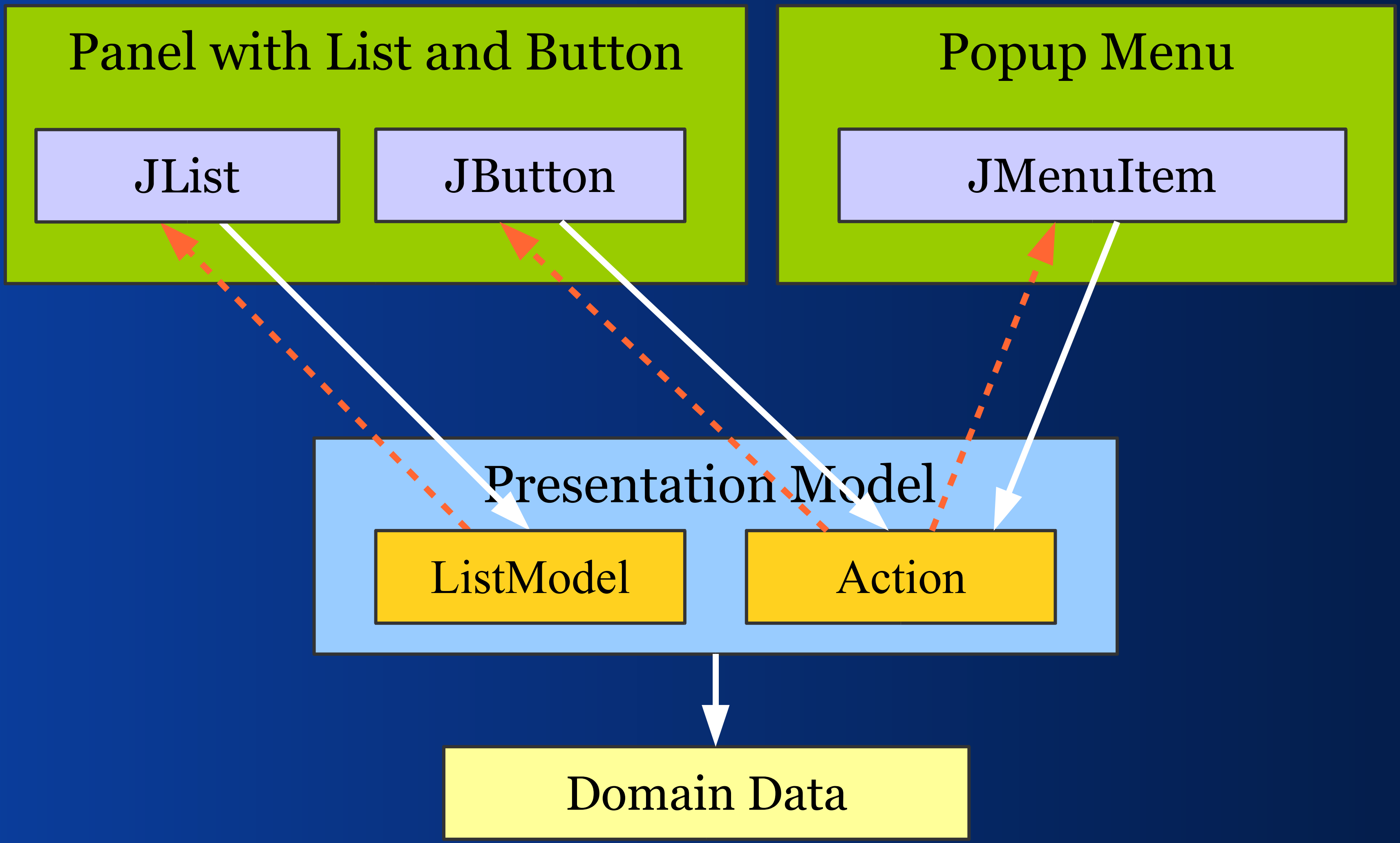
PM: Lists, Tables, Trees



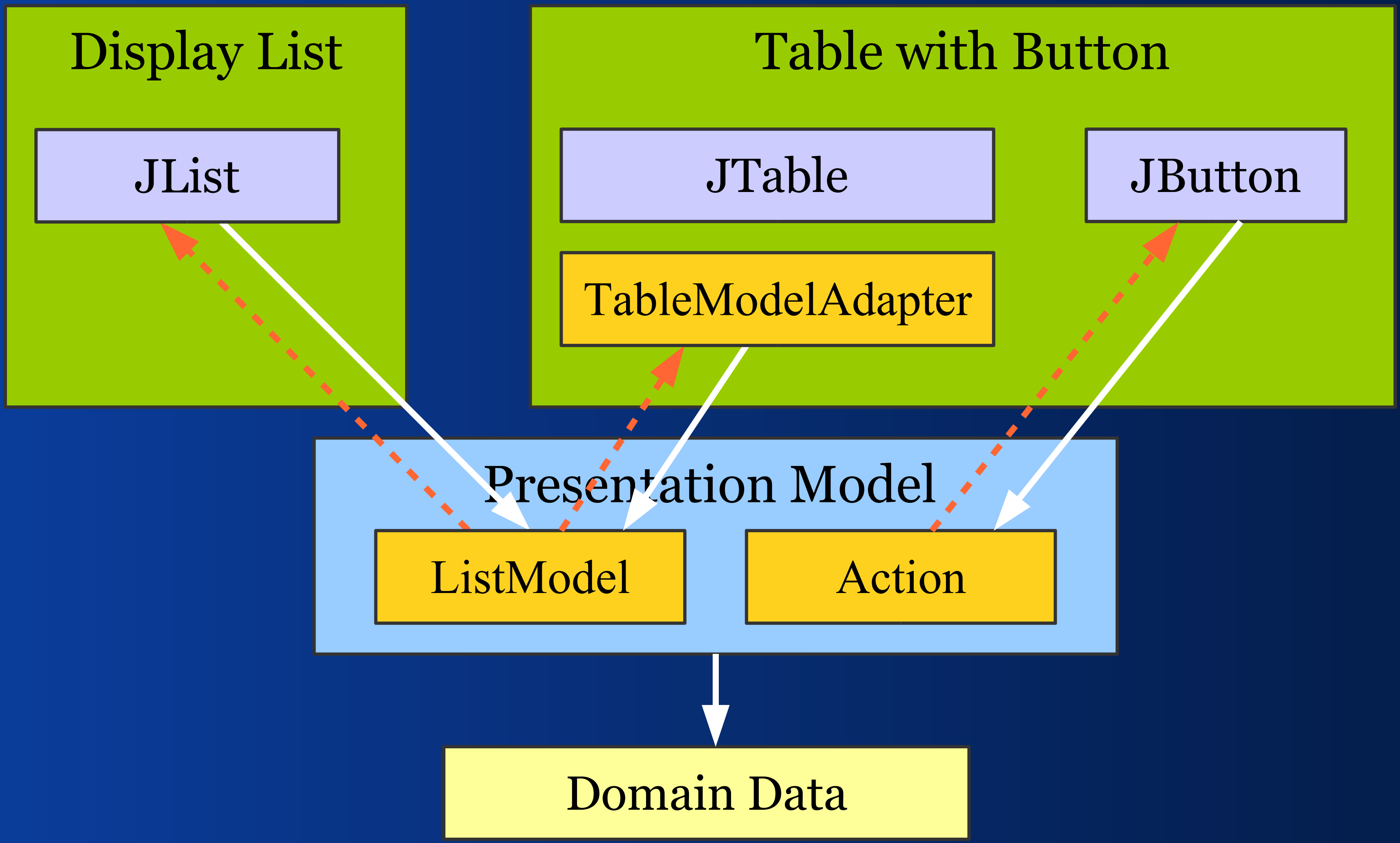
Synchronization Example

```
private void initComponents() {

   okButton = new JButton(
           presentationModel.getOKAction());


   albumList = new JList(
           presentationModel.getAlbumListModel());

     ...
}
```

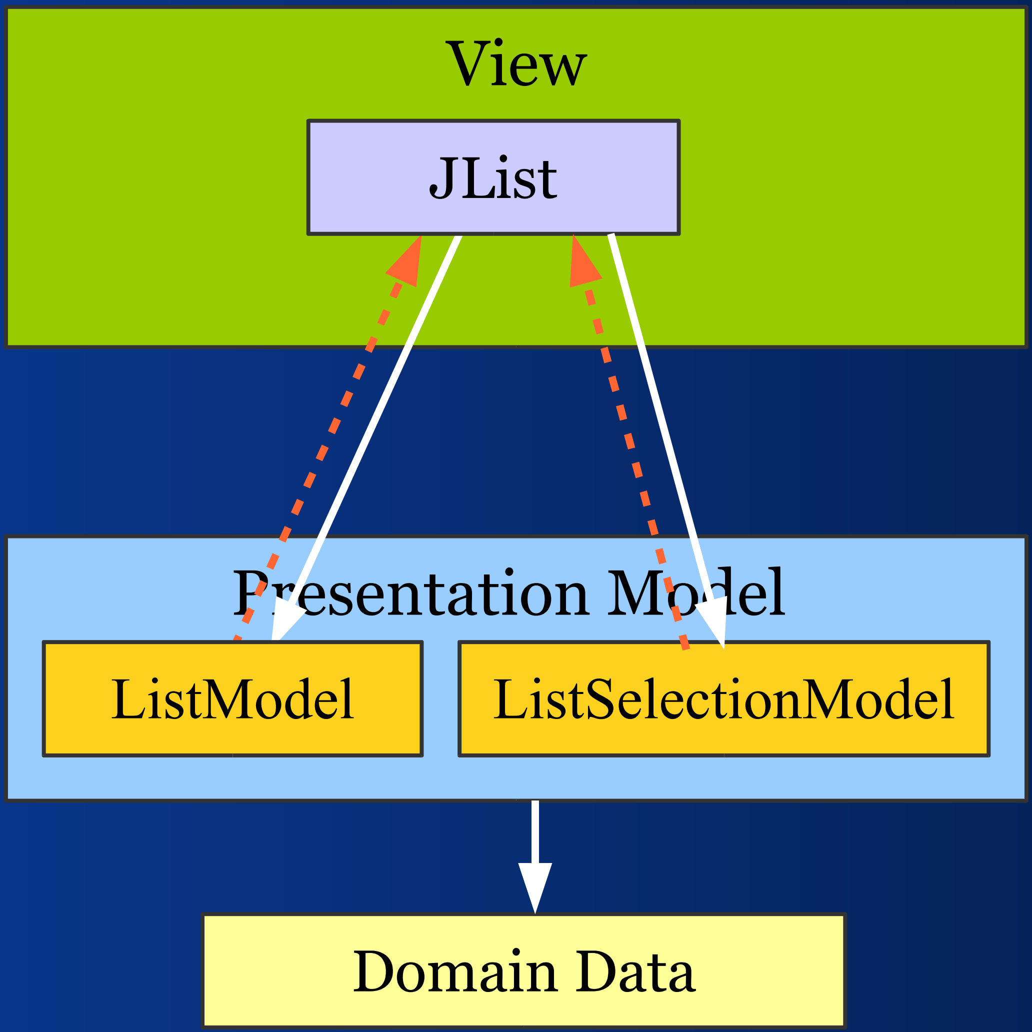
PM: Multiple Views I



PM: Multiple Views II



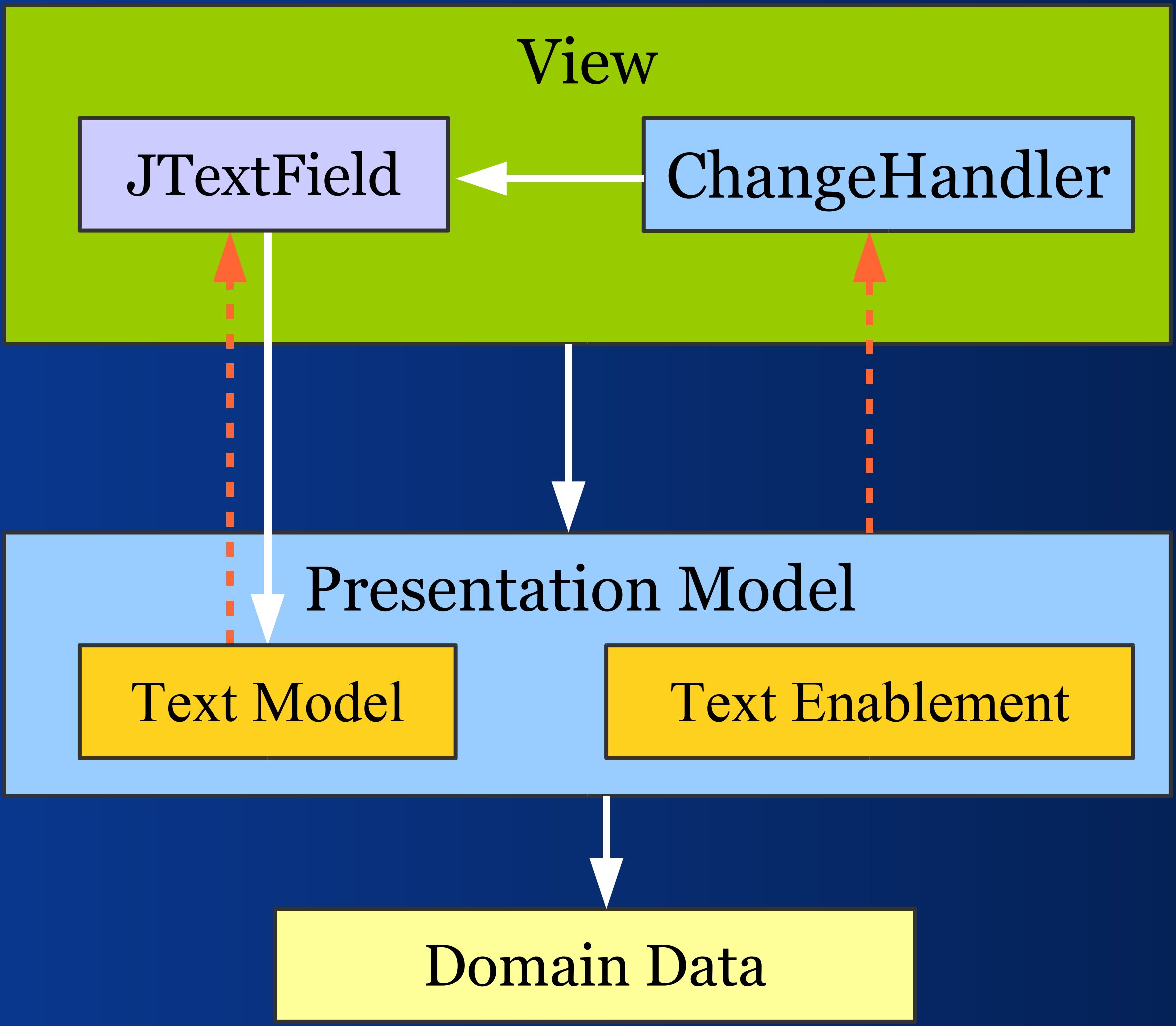
PM: List with Selektion



Three Things Missing

- How do we model non-data GUI state, for example Enablement?
- How do we synchronize single values for JTextField, JFormattedTextField, JLabel?
- How do we synchronize single values between the domain layer and the PMs?

PM Example: Enablement



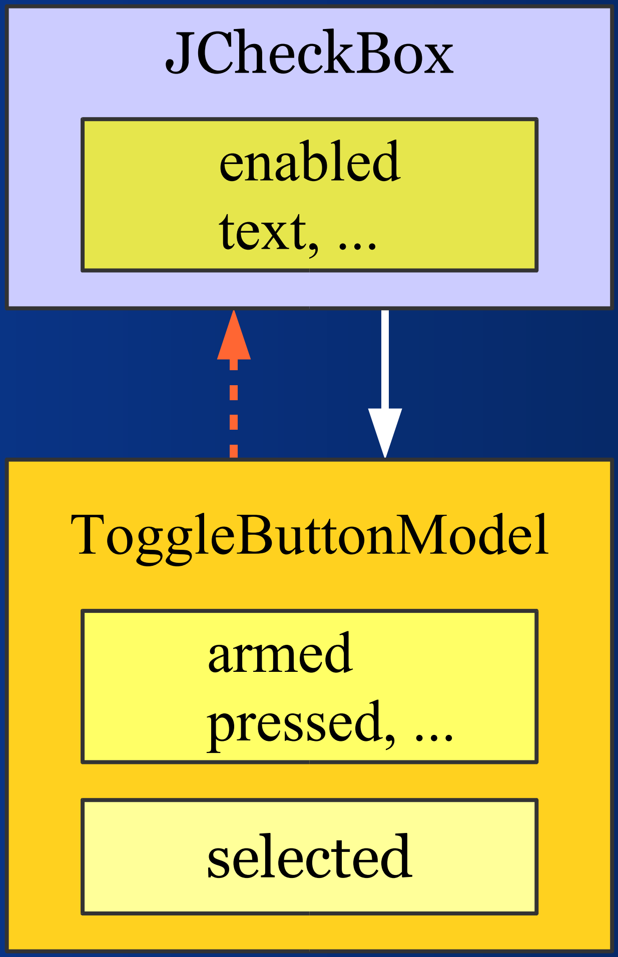
IV – Synchronizing Single Values

How to bind
domain data to UI components?

Binding Tasks

- Read and write domain object properties
- Get and set GUI model state
- Report and handle changes in the domain
- Buffer values – delay until OK pressed
- Change management – commit required?
- Indirection as in an *Master-Detail* view
- Convert types, e. g. Date to String

JCheckBox: Types of State

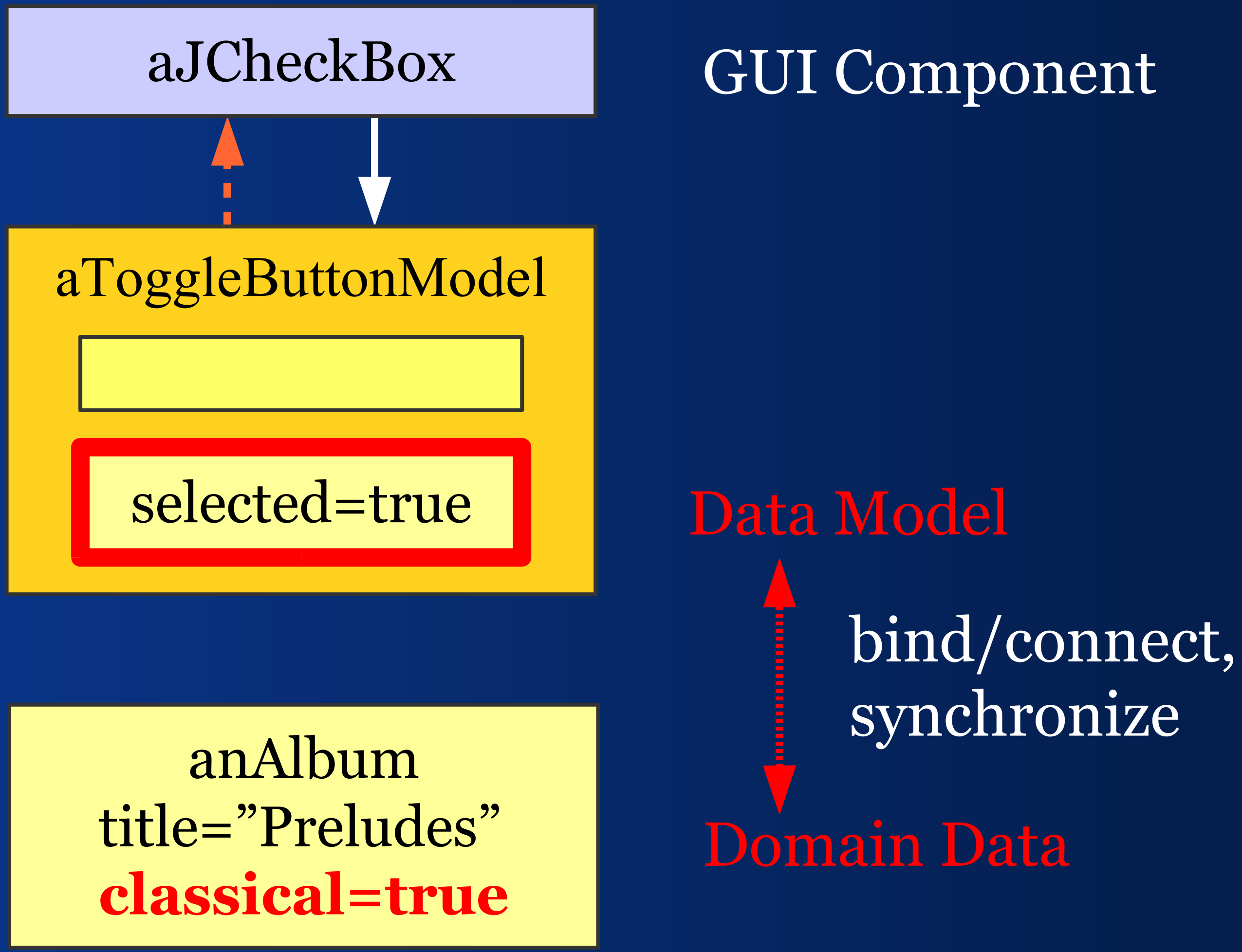


GUI Property

GUI State

Data State

JCheckBox: Binding Task



Copying: Pros and Cons

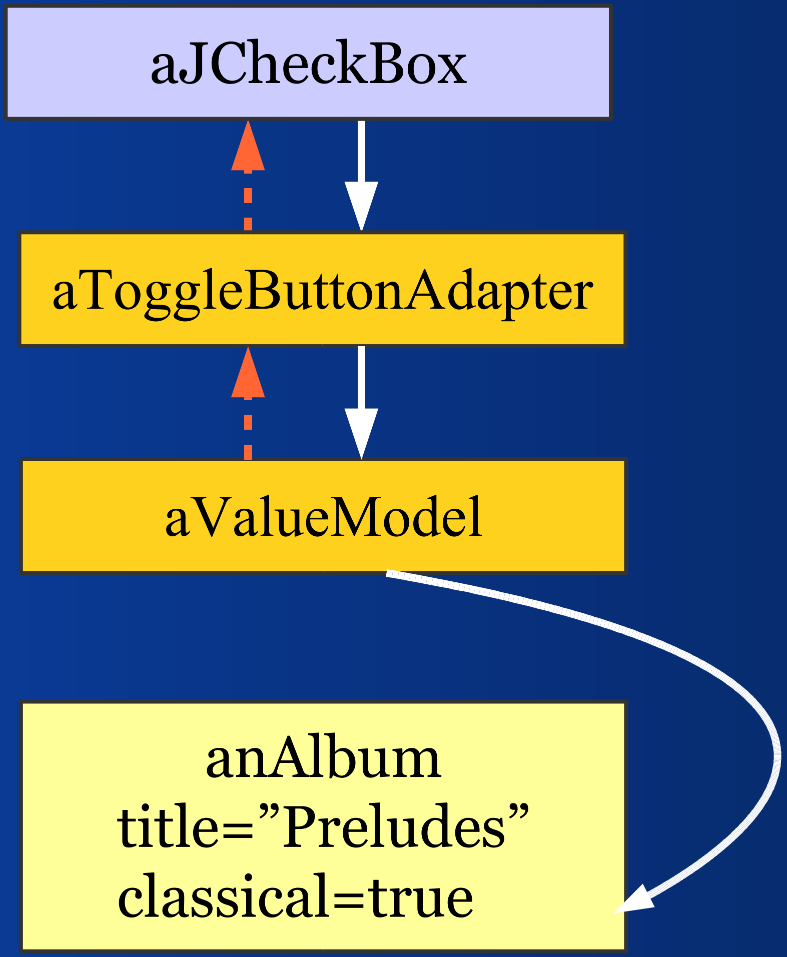
- Easy to understand, easy to explain
- Works in almost all situations
- Easy to debug – explicit data operations

- Blows up the view code
- It's difficult to synchronize views
- Handles domain changes poorly

Concept

- Use a universal model (ValueModel)
- Convert domain properties to ValueModel
- Build converters from ValueModel
  to Swing models: ToggleButtonModel, etc.

ValueModel and Adapter



ValueModel: Requirements

- We want to get its value
- We want to set its value
- We want to observe changes

The ValueModel Interface

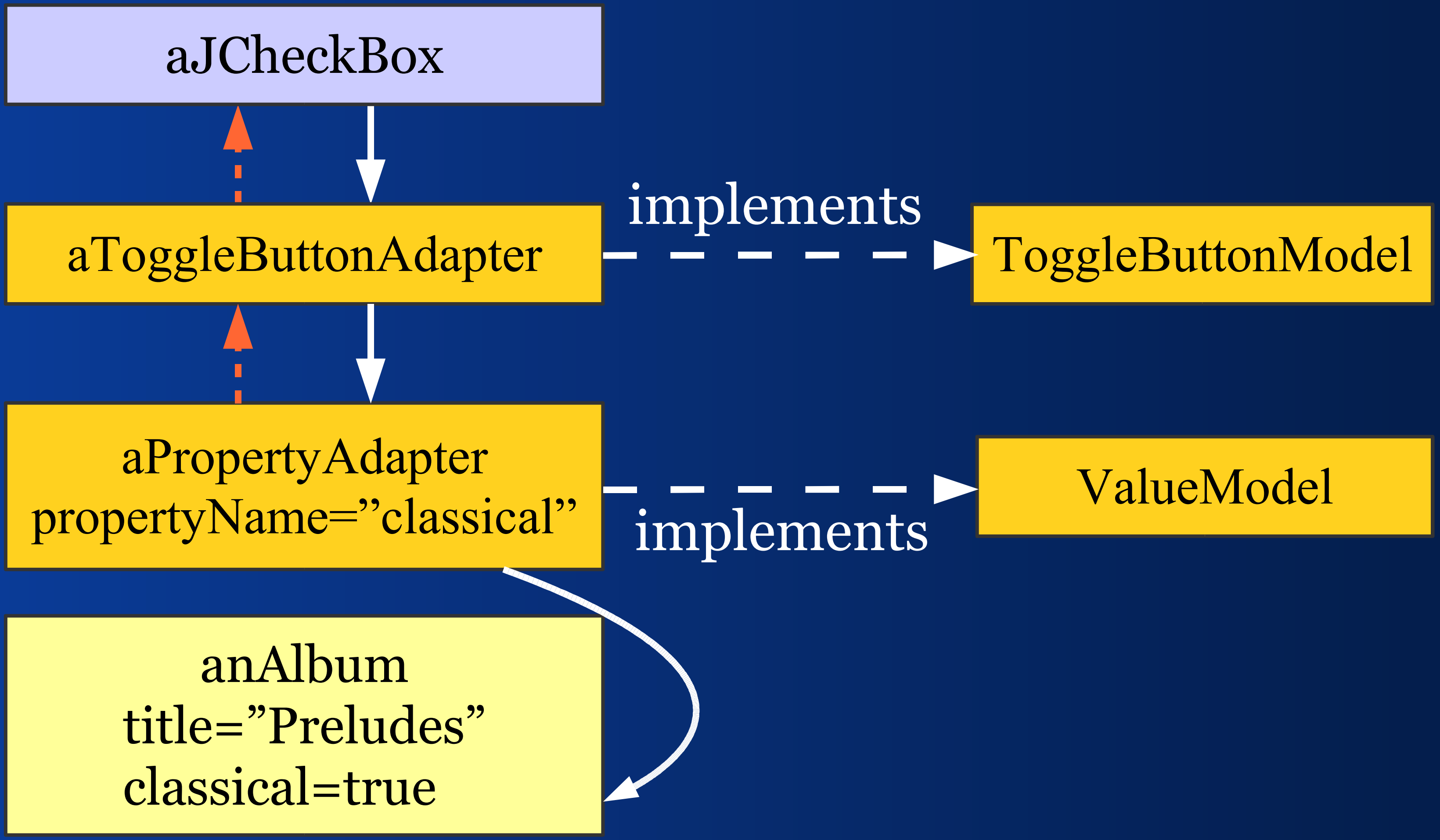
```
public interface ValueModel {

  Object getValue();

  void setValue(Object newValue);

  void addChangeListener(ChangeListener l);

  void removeChangeListener(ChangeListener l);
}
```

Which Event Type?

- ChangeEvent reports no new value;
  must be read from the model – if necessary

- PropertyChangeEvent
  provides the old and new value;
  both can be **`null`**

ValueModel & PropertyAdapter



Domain Object Requirements

- We want to get and set values
- We want to do so in a uniform way
- Changes shall be observable

That's what Java Beans provide.

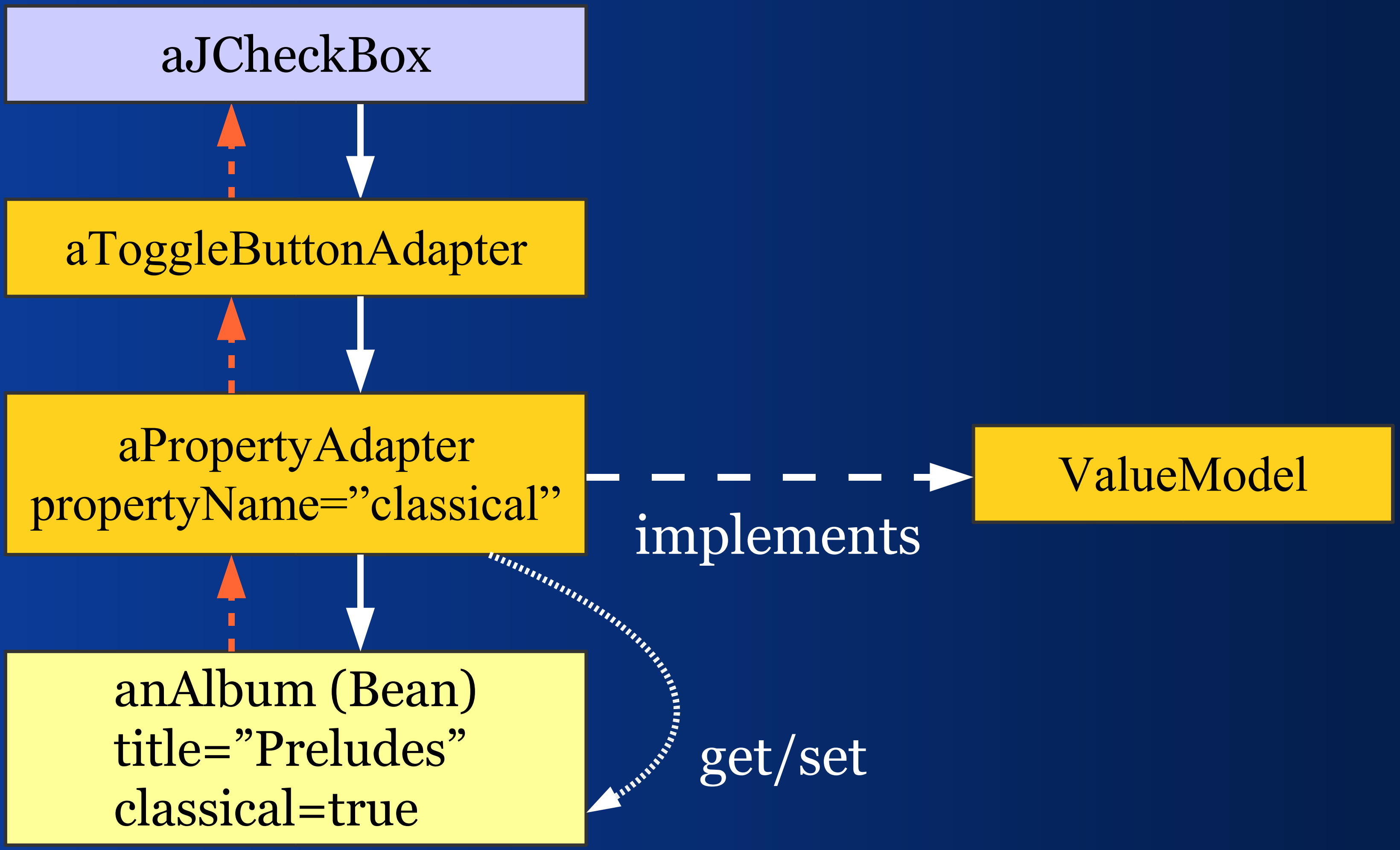
(Bound) Bean Properties

- Java Beans have properties,
  that we can get and set in a uniform way.
- Bean properties are bound,
  if we can observe property changes
  by means of PropertyChangeListeners.

PropertyAdapter

- `BeanAdapter` and `PropertyAdapter` convert Bean properties to ValueModel
- Observe bound properties
- Use Bean Introspection that in turn uses Reflection to get and set bean properties

ValueModel & PropertyAdapter



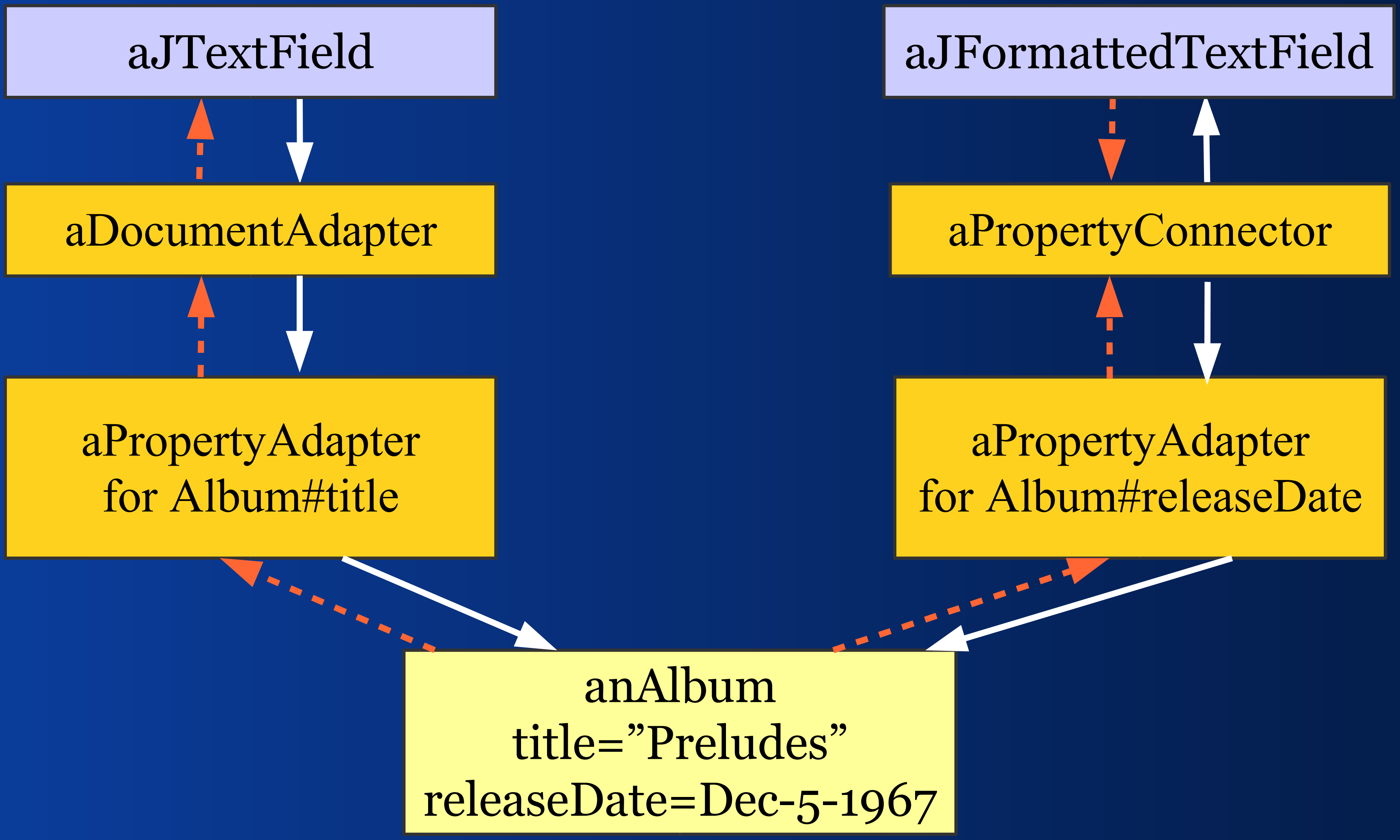
Build a Chain of Adapters

```
private void initComponents() {

   Album album = getEditedAlbum();

   ValueModel aValueModel =
         new PropertyAdapter(album, “classical”);

   JCheckBox classicalBox = new JCheckBox();
   classicalBox.setModel(
         new ToggleButtonAdapter(aValueModel));
}
```

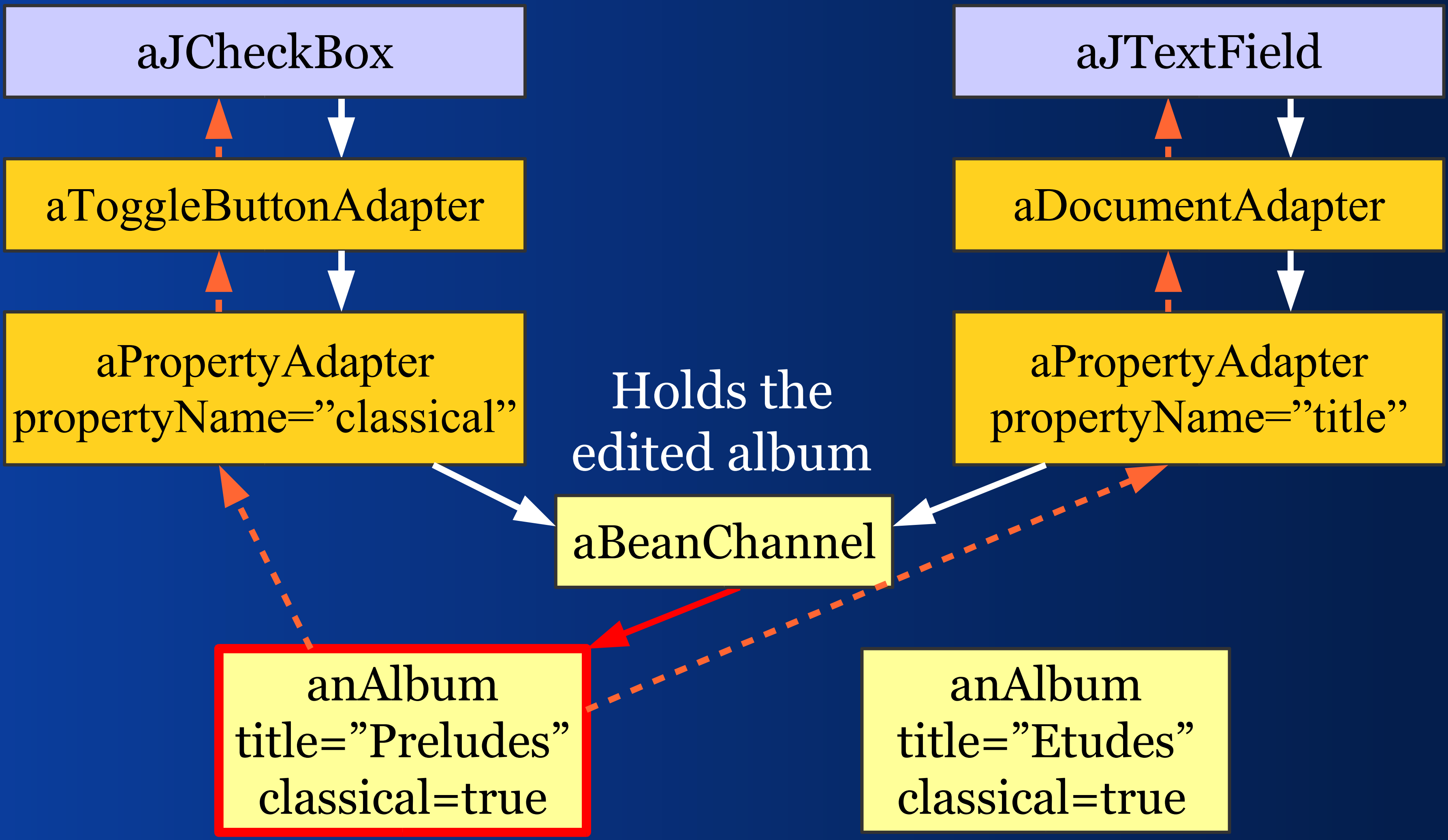
ComponentFactory

```
private void initComponents() {

   Album album = getEditedAlbum();

   JCheckBox classicalBox =
       ComponentFactory.createCheckBox(
           album,
           Album.PROPERTYNAME_CLASSICAL);
}
```

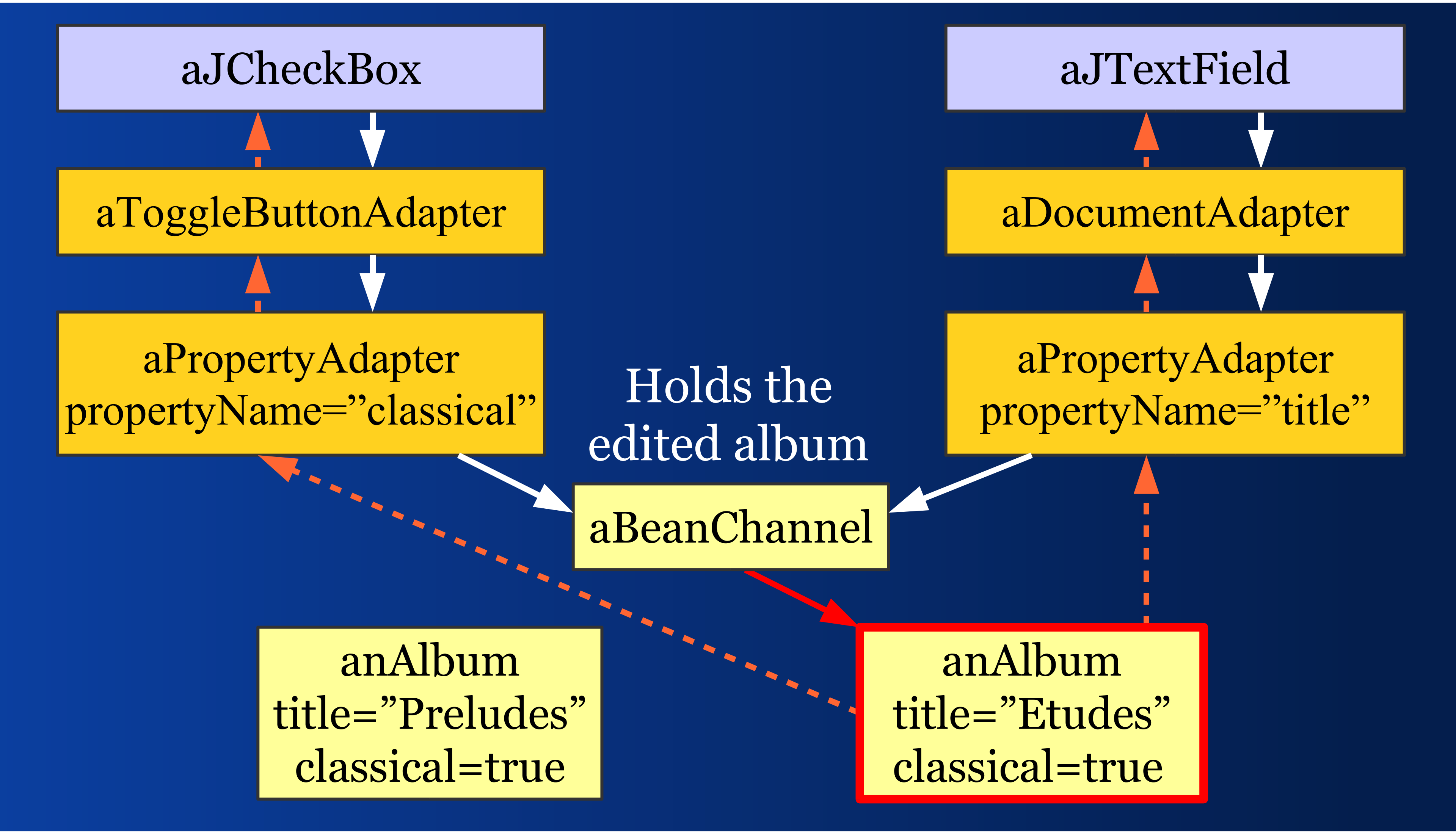
Adapter vs. Connector



Indirection


aJCheckBox
aJTextField
aToggleButtonAdapter
aDocumentAdapter
aPropertyAdapter propertyName=”classical”
Holds the edited album
aPropertyAdapter propertyName=”title”
aBeanChannel
anAlbum title=”Preludes” classical=true
anAlbum title=”Etudes” classical=true

Indirection



Example View Source Code

1) Variables for UI components
2) Constructors
3) Create, bind, configure UI components
4) Register GUI state handlers with the model
5) Build and return panel
6) Handlers that update GUI state

Example View 1/7

```
public final class AlbumView {

    // Refers to the model provider
    private AlbumPresentationModel model;

    // UI components
    private JTextField titleField;
    private JCheckBox  classicalBox;
    private JButton    buyNowButton;
    private JList      referencesList;
    ...
```

Example View 2/7

```
public AlbumView(AlbumPresentationModel m) {

    // Store a ref to the presentation model
    this.model = m;

    // Do some custom setup.
    ...
}
```

Example 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());
```

Example View 4/7

```
private initEventHandling(){
    // Observe the model to update GUI state
    model.addPropertyChangeListener(
        “composerEnabled”,
        new ComposerEnablementHandler());
}
```

Example View 5/7

```
public JPanel buildPanel() {
    // Create, bind and configure components
    initComponents();

    // Register handlers that change UI state
    initEventHandling();

    FormLayout layout = new FormLayout(
        "right:pref, 3dlu, pref", // 3 columns
        "p, 3dlu, p");            // 3 rows

    ...
```

Example 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();
}
```

Example View 7/7

```
/* Listens to #composerEnabled,
   changes #enabled of the composerField.  */
private class ComposerEnablementHandler
    implements PropertyChangeListener {

    public void propertyChange(
        PropertyChangeEvent evt) {

        composerField.setEnabled(
            model.isComposerEnabled());
    }
}
```

Simpler Event Handling

```
private initEventHandling(){
    // Synchronize model with GUI state
    PropertyConnector.connect(
        model,           "composerEnabled",
        composerField, "enabled");
}
```

V - Field Report

How does PM and Adapter Binding work?

Design Goals

- Works with standard Swing components
- Works with custom Swing components

- Requires no special components
- Requires no special panels

- Integrates well with validation
- Works with different validation styles

Costs

- Adapter Binding:
  - increases learning costs
  - decreases production costs a little
  - can significantly reduce change costs

Use a ComponentFactory!

- Encapsulate the creation of adapters from ValueModel to Swing components.
- Some components have no appropriate model, e. g. JFormattedTextField
- Vends components for ValueModels

Tip

- Observer/Observable works well between different layers.
- Use Observer judiscously in a layer.

Warnings

- Using Observer in the domain layer makes it more difficult to understand what's going on if a domain property changes.

- Be aware of memory leaks, if you observe domain data with listeners that are registered permanently. In this case, the domain data references the GUI.

Performance

- Adapter chains fire many change events
- That seems to be no performance problem

- ListModel can improve the performance compared to copying list contents

Debugging

- Copying approach is easy to debug; you can see when where what happens.
- Adapter chains “move“ values implicitly; it's harder to understand updates.
- Reflection and Introspection hide who reads and writes values.
- Favor named over anonymous listeners.

Renaming Methods

- Reflection and Introspection make it more difficult to rename bean properties and their getter and setters.
- Use constants for bean property names!
- Obfuscators fail to detect the call graph.

When is Binding Useful?

- I guess that adapter binding can be applied to about 80% of all Swing projects.
- However, you need at least one expert who masters the binding classes.

Why MVP has been Created

- MVP has been created, because many Smalltalk developers have implemented Presentation Models that referred directly to the (single) view.

- Swing and Binding help avoid this problem.

State of the JGoodies Binding?

- Approach is 10 years old and stable.
- Architecture of the Java port is stable.
- Tests cover 90% of the classes.
- Little documentation.
- Tutorial is quite small.

End

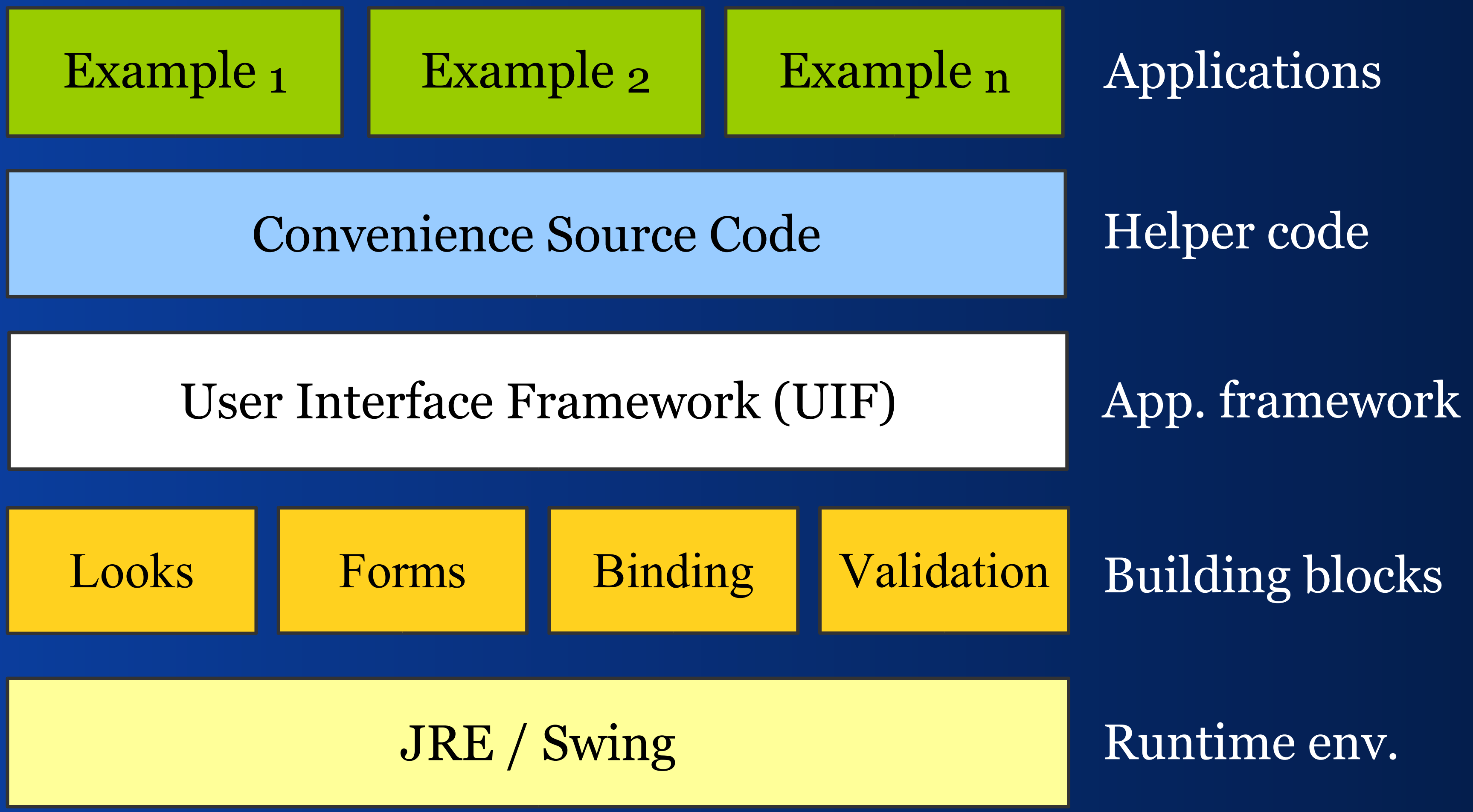
Summary and References

Summary

- Separate the domain from the presentation! That is Separated Presentation.
- Separate Autonomous View if appropriate
- Choose MVP or Presentation Model

- Swing makes Presentation Model easy
- PM requires a binding solution

JGoodies Swing Suite



References I

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

References II

- Suns JDNC
  jdnc.dev.java.net
- Oracles JClient and ADF
  otn.oracle.com/, search 'JClient'
- Spring Rich Client Project
  www.springframework.org/spring-rcp.html

References III

- VisualWorks Application Architecture
  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/

Tiny Examples/Tutorial:

JGoodies Binding Tutorial

Data binding problems and solutions

Ships with the JGoodies Binding

Questions & Answers

End

Hope that helps!

Good luck!