Grand Valley State University
www.gvsu.edu

Software Engineering
Design Patterns (3)
MVC & MVP

Erik Fredericks // frederer@gvsu.edu

Adapted from materials provided by Byron DeVries, Jagadeesh Nandigam

MVC // MVP

MVC (Model-View-Controller)

- Motivation
- MVC Class Structure
- Division of Labor within MVC
- Example

MVP (Model-View-Presenter)

- Motivation
- MVP Class Structure
- Division of Labor within MVP
- Example

MVC pattern

Pattern Category: Architectural

Intent: Decoupling major components in a user interface.

Problem addressed: Coupling of model and/or business logic within a GUI.

State/data of an application:

- should be agnostic of user interface
- should be logically independent of how it is displayed to the user

Solution: Divide the application into three components (Model, View, Controller) for modeling of the domain, the presentation, and the user inputs.

MVC pattern

Implementation:

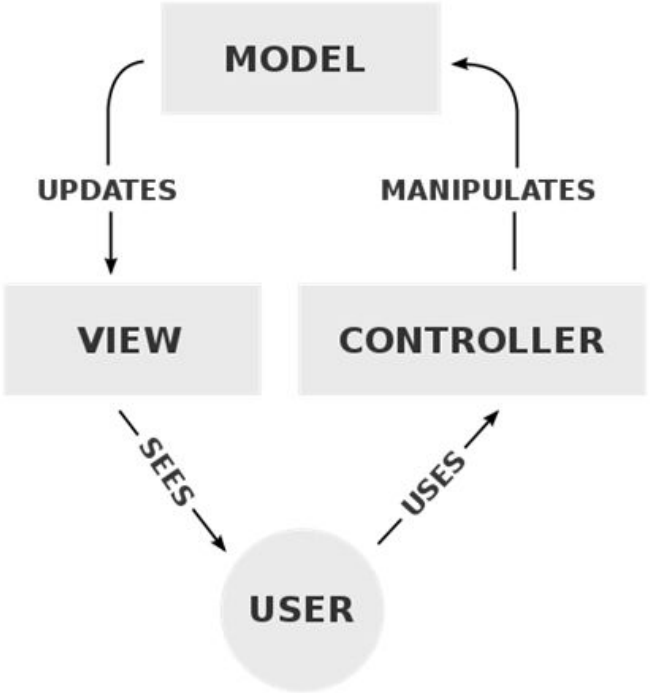
- Model expresses application behavior related to the problem domain, without consideration for the user interface.
- View is any output representation of the information within the model
- Controller converts input to be used within the problem domain (i.e., model)

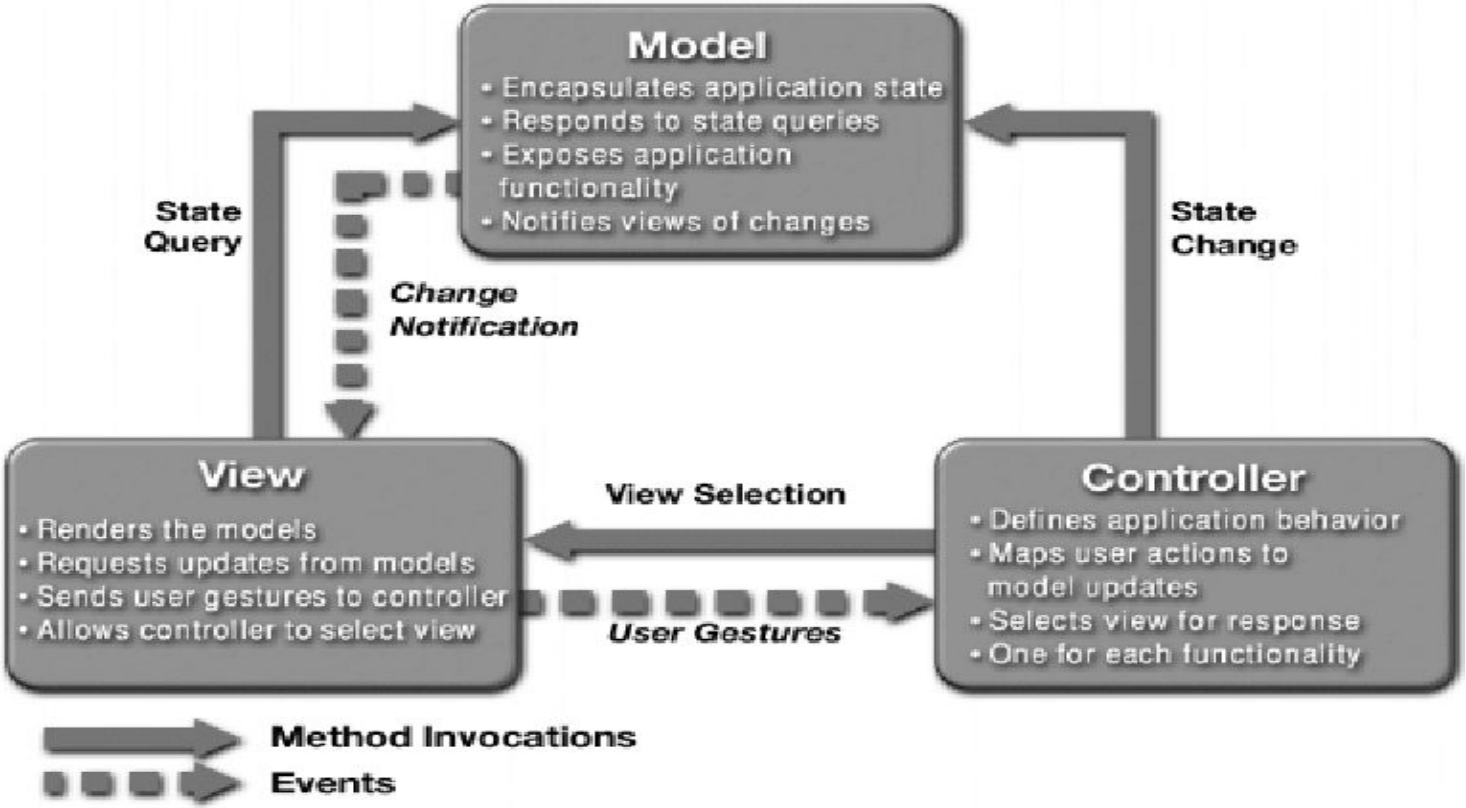
MVC pattern

Implementation:
Model expresses application behavior related to the problem domain, without consideration for the user interface.

View is any output representation of the information within the model

Controller converts input to be used within the problem domain (i.e., model)




Model
• Encapsulates application state
• Responds to state queries
• Exposes application functionality
• Notifies views of changes
State Query
Change Notification
State Change
View
• Renders the models
• Requests updates from models
• Sends user gestures to controller
• Allows controller to select view
View Selection
User Gestures
Controller
• Defines application behavior
• Maps user actions to model updates
• Selects view for response
• One for each functionality
Method Invocations
Events

MVC pattern

Model:

- Manages application data/state
- Expresses application behavior/functionality
- Responds to instructions to change (usually from the controller)
- Responds to requests for information about its state (usually from the view)
- Notifies views (and controllers) of changes (if model is not passive).

View:

- Renders the contents of the model for the user
- Sends user gestures/actions to controller
- Requests information from model
- Allows controller to select view

MVC pattern

Controller:

- Translates user actions into operations on the model
- Manages application behavior
- Commands the view to change as appropriate

MVC pattern

- The model **knows only about itself**. That is, the source code of the model has **no references to either the view or controller**.
- An object can act as a model for **more than one MVC triad at a time**.
- The model may be totally ”unaware” of the existence of either the view or the controller and of its participation in an MVC triad.
- The model object may see a **view or controller** object merely as **an observer** object if state change notifications are required.
- Both view and controller know about the model.
- The **view and controller** are specifically designed to **work together**. Each view is associated with a unique controller and vice versa.

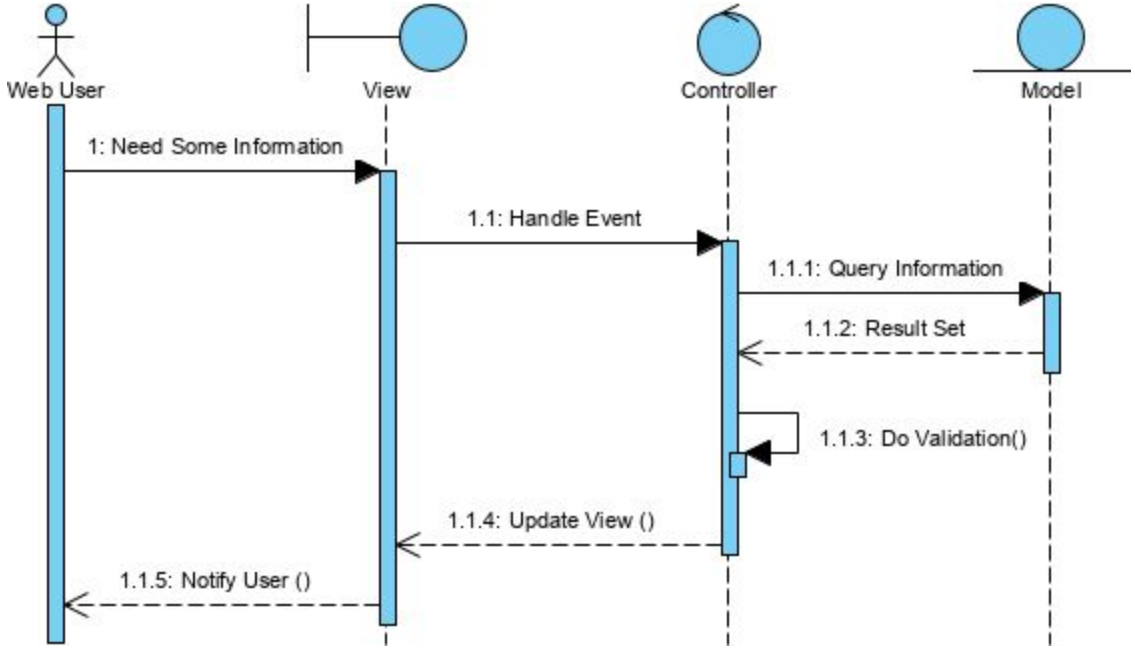
MVC pattern

Consequences (Advantages):

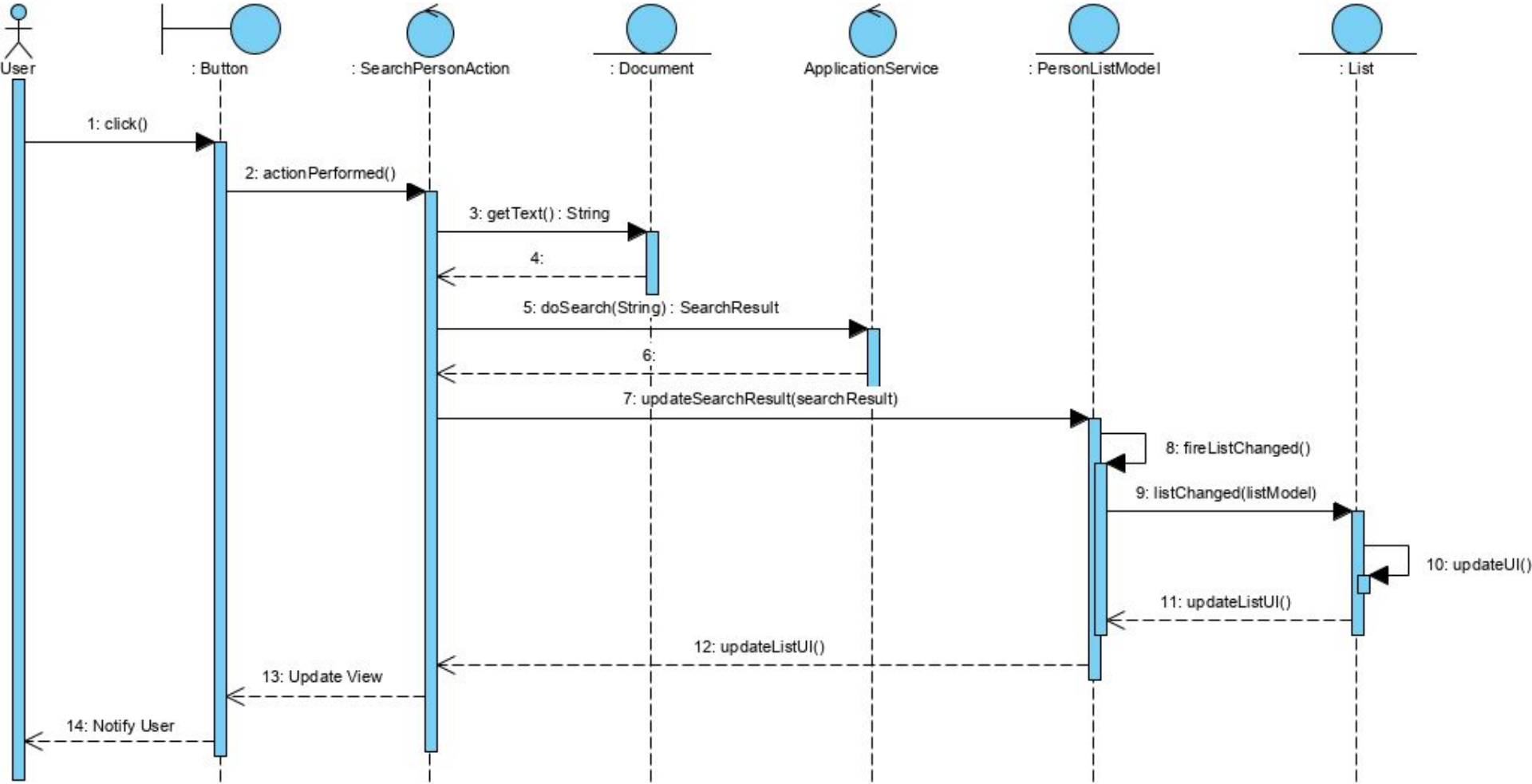
- Multiple developers can work on the application simultaneously due to the decoupling
- Higher cohesion (i.e., similar things together) and lower coupling (i.e., less forced connections between dissimilar things)
- Models can have multiple views

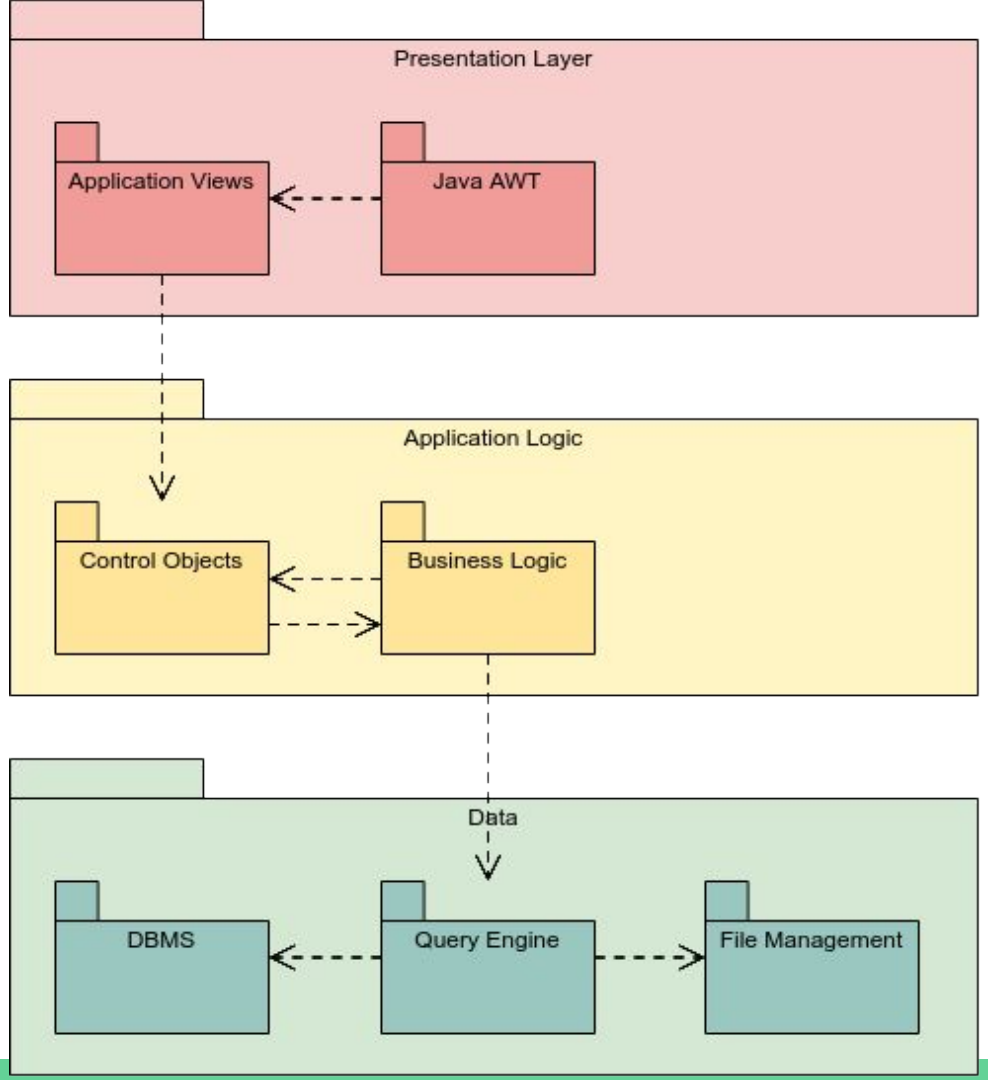
Consequences (Disadvantages):

- Harder to navigate code due to additional layers of abstraction
- Requires maintenance of multiple artifacts
- Steeper learning curve, requires knowledge in multiple areas

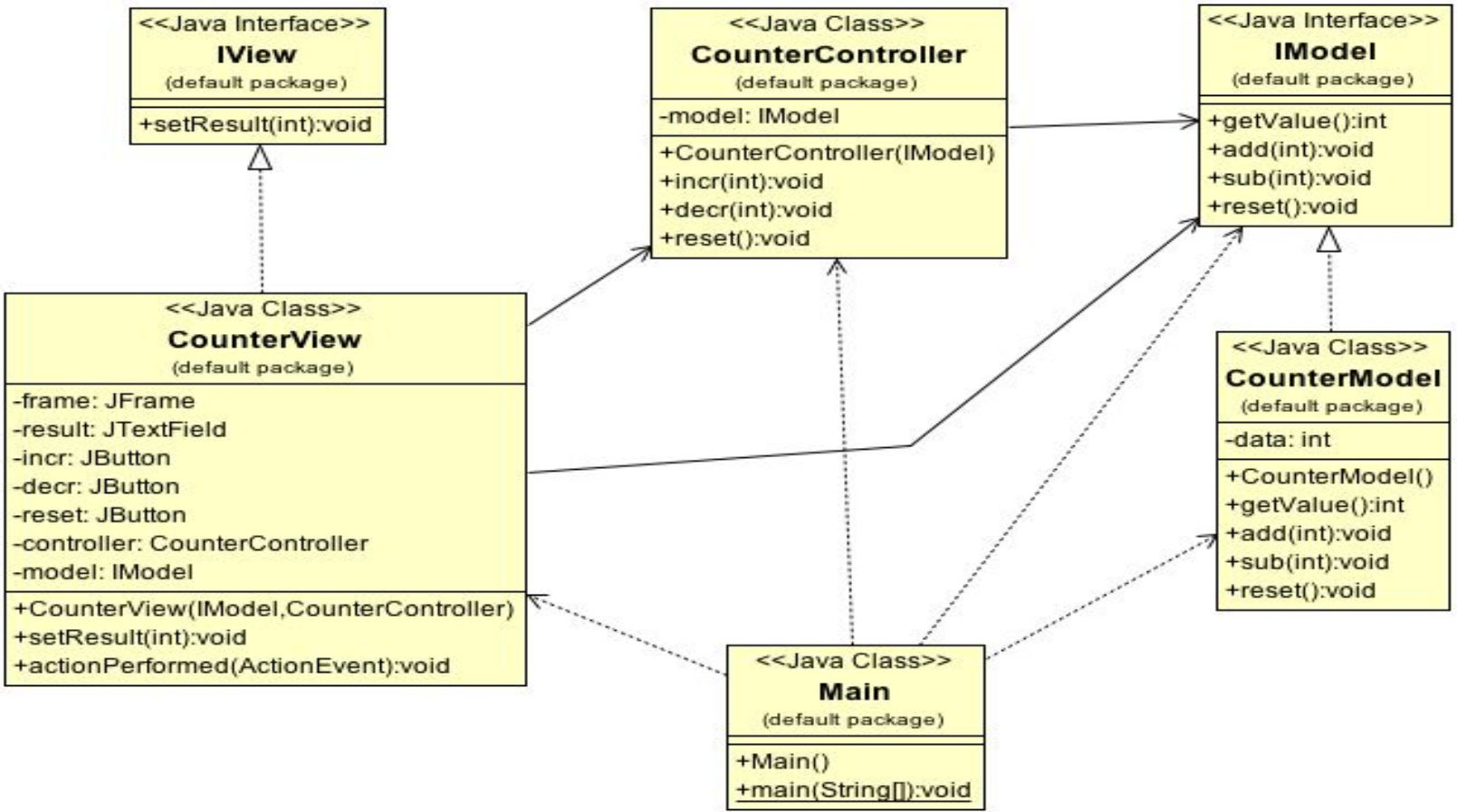


% visual paradigm


User
: Button
: SearchPersonAction
: Document
ApplicationService
: PersonListModel
: List
1: click()
2: actionPerformed()
3: getText() : String
4:
5: doSearch(String) : SearchResult
6:
7: updateSearchResult(searchResult)
8: fireListChanged()
9: listChanged(listModel)
10: updateUI()
11: updateListUI()
12: updateListUI()
13: Update View
14: Notify User


Presentation Layer
Application Views
Java AWT
Application Logic
Control Objects
Business Logic
Data
DBMS
Query Engine
File Management

MVC Demo


<<Java Interface>>
IView
(default package)
+setResult(int):void
<<Java Class>>
CounterController
(default package)
-model: IModel
+CounterController(IModel)
+incr(int):void
+decr(int):void
+reset():void
<<Java Interface>>
IModel
(default package)
+getValue():int
+add(int):void
+sub(int):void
+reset():void
<<Java Class>>
CounterView
(default package)
-frame: JFrame
-result: JTextField
-incr: JButton
-decr: JButton
-reset: JButton
-controller: CounterController
-model: IModel
+CounterView(IModel,CounterController)
+setResult(int):void
+actionPerformed(ActionEvent):void
<<Java Class>>
CounterModel
(default package)
-data: int
+CounterModel()
+getValue():int
+add(int):void
+sub(int):void
+reset():void
<<Java Class>>
Main
(default package)
+Main()
+main(String[]):void


MVP

MVP pattern

Pattern Category: Architectural

Intent: Decoupling major components in a user interface & make sure view is completely passive.

Problem addressed: Coupling of model and/or business logic within a GUI.
State/data of an application:
should be agnostic of user interface
should be logically independent of how it is displayed to the user

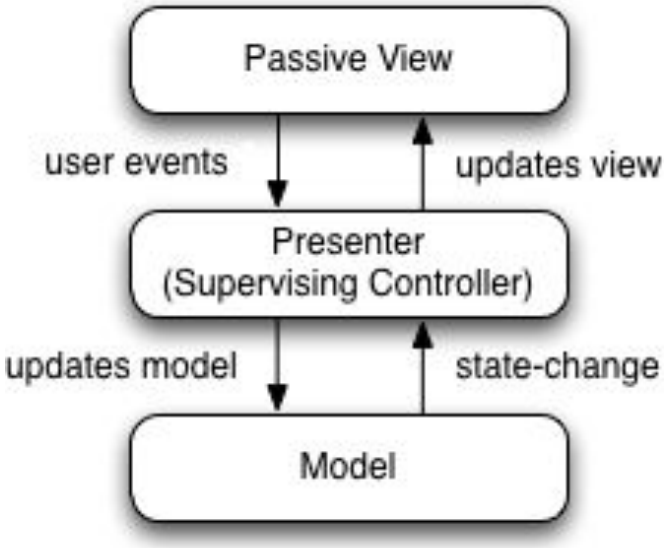
Solution: View and Model are fully decoupled and unaware of each other.

Implementation:

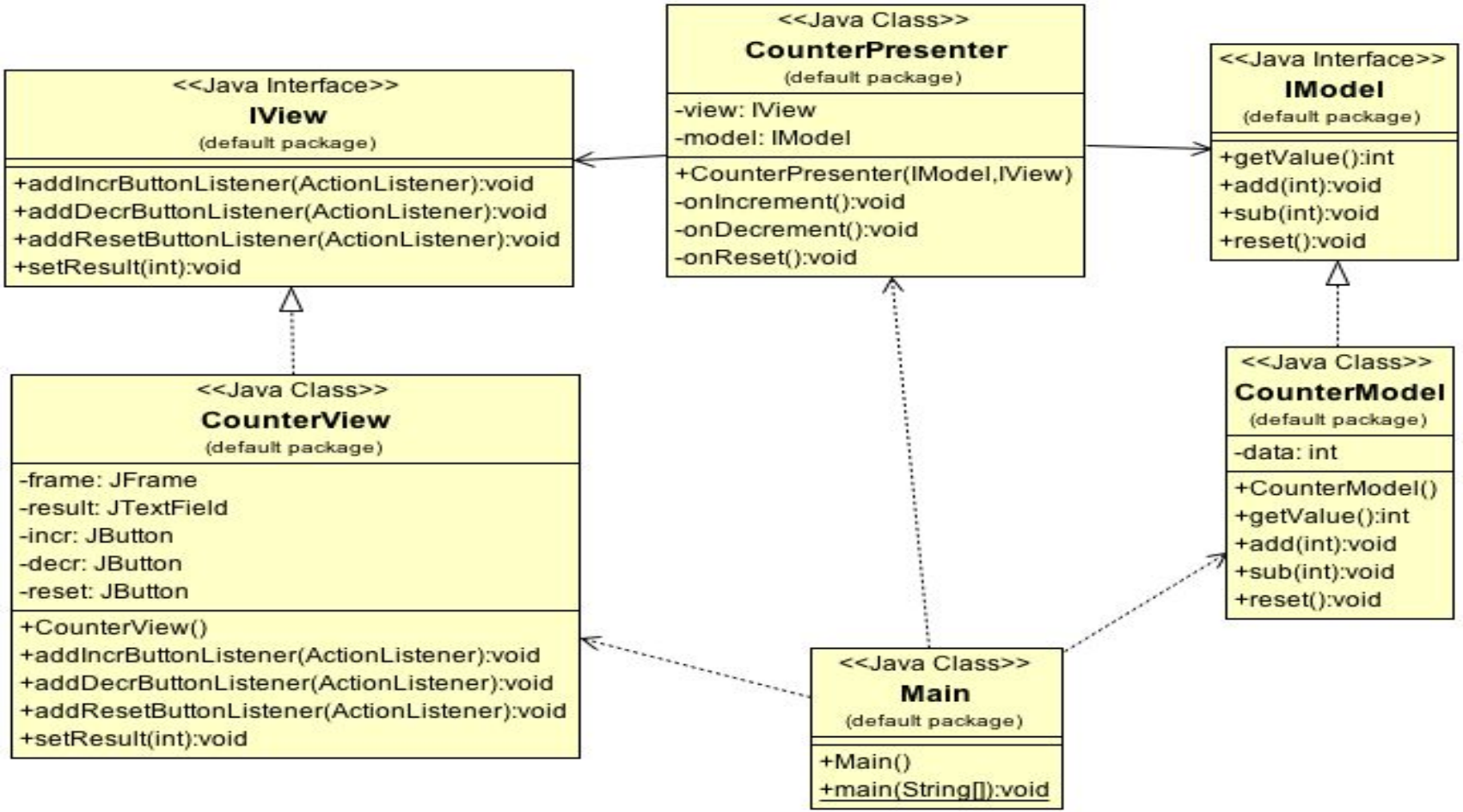
- *Model* manages data/state and expresses application behavior/functionality.

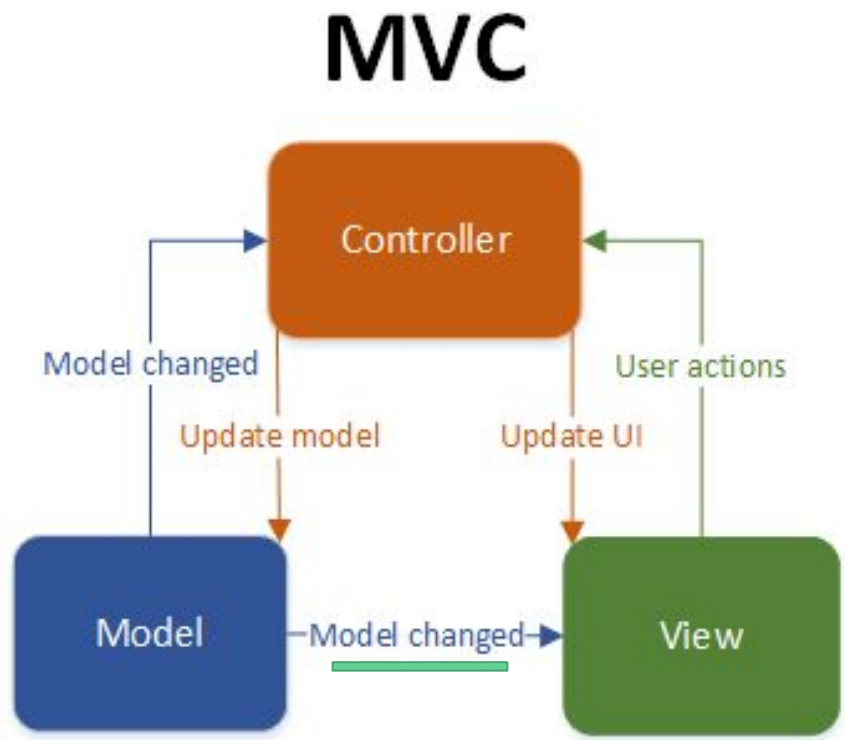
- *View* is very passive, thin, and shallow.
  - Provides data to presenter
  - Routes events to presenter
  - Obtains data from the presenter

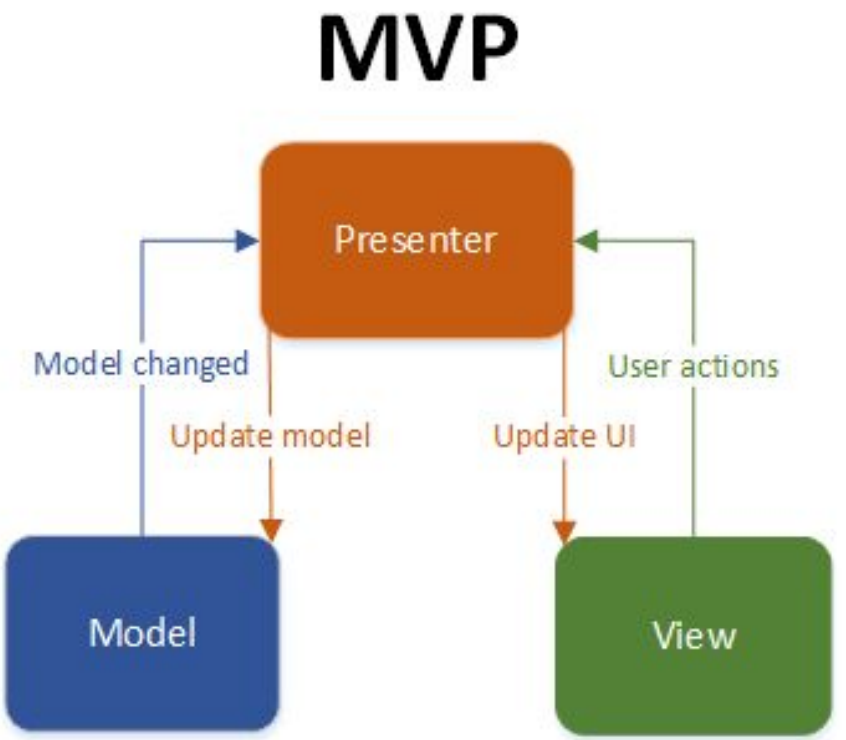
- *Presenter* acts as both view and model.
  - Bulk of application logic resides here
  - Fully decouples view & model



MVP Demo


<<Java Interface>>
IView
(default package)
+addIncrButtonListener(ActionListener):void
+addDecrButtonListener(ActionListener):void
+addResetButtonListener(ActionListener):void
+setResult(int):void
<<Java Class>>
CounterPresenter
(default package)
-view: IView
-model: IModel
+CounterPresenter(IModel,IView)
-onIncrement():void
-onDecrement():void
-onReset():void
<<Java Interface>>
IModel
(default package)
+getValue():int
+add(int):void
+sub(int):void
+reset():void
<<Java Class>>
CounterView
(default package)
-frame: JFrame
-result: JTextField
-incr: JButton
-decr: JButton
-reset: JButton
+CounterView()
+addIncrButtonListener(ActionListener):void
+addDecrButtonListener(ActionListener):void
+addResetButtonListener(ActionListener):void
+setResult(int):void
<<Java Class>>
CounterModel
(default package)
-data: int
+CounterModel()
+getValue():int
+add(int):void
+sub(int):void
+reset():void
<<Java Class>>
Main
(default package)
+Main()
+main(String[]):void


MVC
Controller
Model changed
Update model
Update UI
User actions
Model
Model changed
View


MVP
Presenter
Model changed
Update model
Update UI
User actions
Model
View

(Python)

https://www.giacomodebidda.com/mvc-pattern-in-python-introduction-and-basicmodel/

https://www.tutorialspoint.com/python_design_patterns/python_design_patterns_model_view_controller.htm

MVC Demo ➜ Flask!
MVP Demo ➜ self-guided!
https://medium.com/cr8resume/make-you-hand-dirty-with-mvp-model-view-presenter-eab5b5c16e42

MVC vs MVP

Which is better?