Model-View Design Pattern

A Lecture for KMP's COMP 401 Class

Aaron Smith

November 14, 2019

Adapted from Ketan Mayer-Patel's Lecture Slides

COMP 401 Big Picture

First Half: Object Oriented Programming

- Interfaces
- Classes
- Polymorphism
- Inheritance

Second Half: Design Patterns

- Iterator
- Factory
- Observer
- Model-View

What are design patterns?

Design patterns are techniques

for **organizing your code**

that are often used in the real world

What are design patterns?

Writing code **without** design patterns

Writing code **with** design patterns

Design Patterns

- Iterator
- Factory
- Observer / Observable
- Model-View
- Model-View-Controller
- Decorator
- Singleton
- And more…

Review: AWT/Swing

What is AWT/Swing?

AWT/Swing are built-in Java libraries for making **user interfaces (UIs)**

AWT/Swing consists of about **200 classes (!)** which implement various UI components

Do I have to remember 200+ classes?

Swing: https://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html

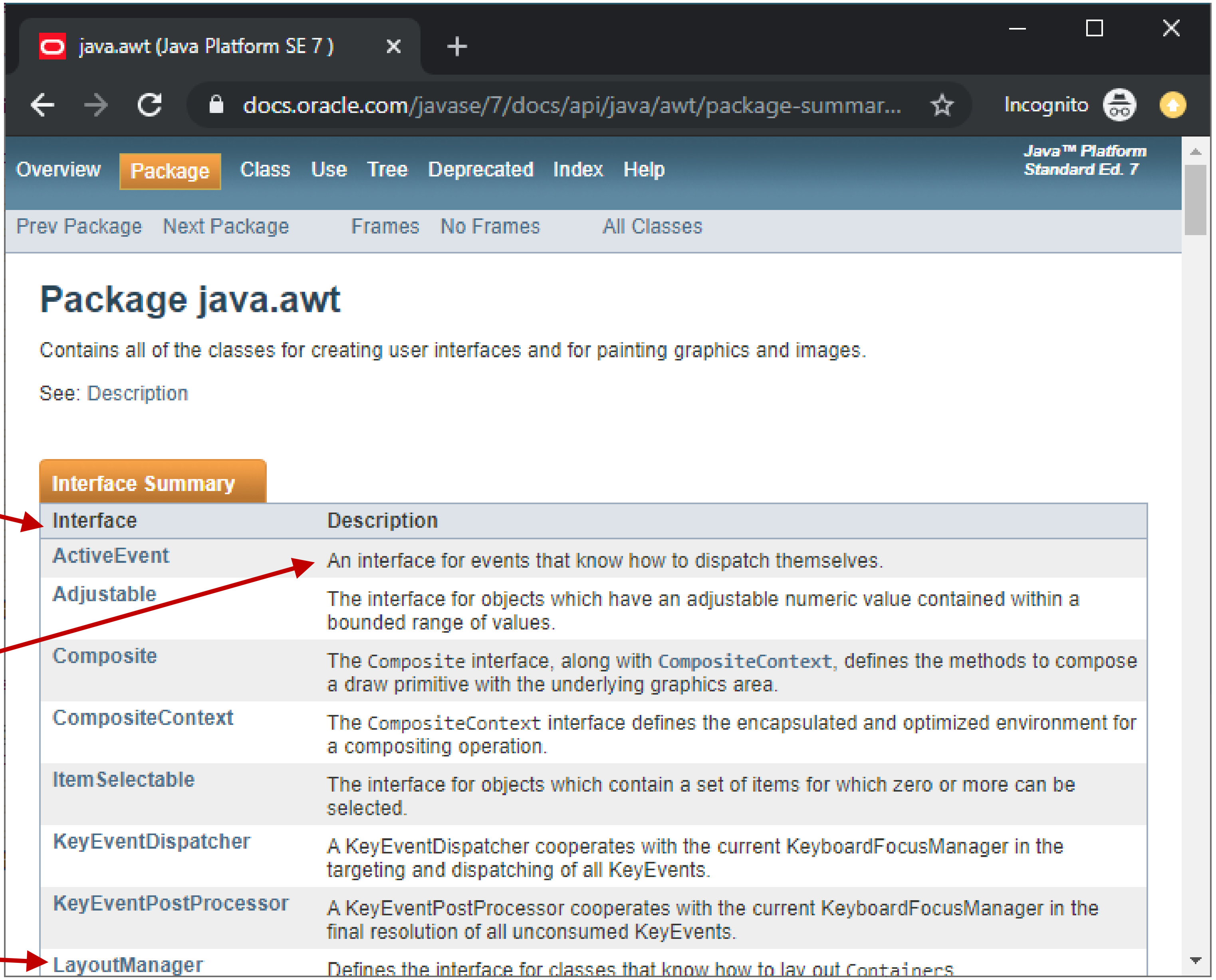
AWT: https://docs.oracle.com/javase/7/docs/api/java/awt/package-summary.html

AWT Documentation

Lists AWT's **interfaces** and **classes**

Provides descriptions of each

More details accessible by clicking the links


java.awt (Java Platform SE 7)
docs.oracle.com/javase/7/docs/api/java/awt/package-summar...
Incognito
Overview Package Class Use Tree Deprecated Index Help
Java™ Platform Standard Ed. 7
Prev Package Next Package Frames No Frames All Classes

Package java.awt

Contains all of the classes for creating user interfaces and for painting graphics and images.

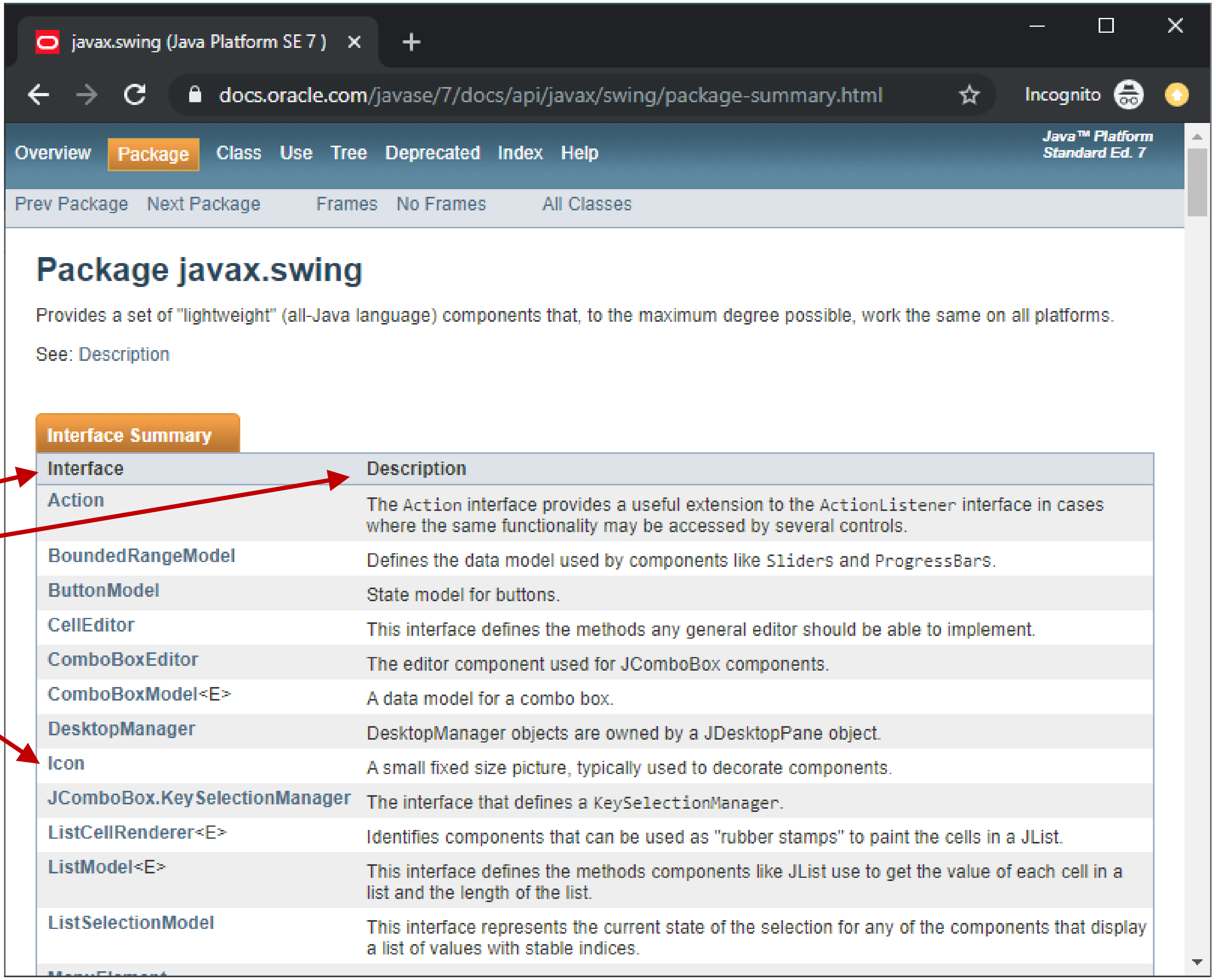
See: Description

Interface Summary

Interface	Description
ActiveEvent	An interface for events that know how to dispatch themselves.
Adjustable	The interface for objects which have an adjustable numeric value contained within a bounded range of values.
Composite	The `Composite` interface, along with `CompositeContext`, defines the methods to compose a draw primitive with the underlying graphics area.
CompositeContext	The `CompositeContext` interface defines the encapsulated and optimized environment for a compositing operation.
ItemSelectable	The interface for objects which contain a set of items for which zero or more can be selected.
KeyEventDispatcher	A KeyEventDispatcher cooperates with the current KeyboardFocusManager in the targeting and dispatching of all KeyEvents.
KeyEventPostProcessor	A KeyEventPostProcessor cooperates with the current KeyboardFocusManager in the final resolution of all unconsumed KeyEvents.
LayoutManager	Defines the interface for classes that know how to lay out `Containers`

Swing Documentation

Java's **Swing** documentation page is similar


javax.swing (Java Platform SE 7)
docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html
Incognito
Overview Package Class Use Tree Deprecated Index Help
Java™ Platform Standard Ed. 7
Prev Package Next Package Frames No Frames All Classes

Package javax.swing

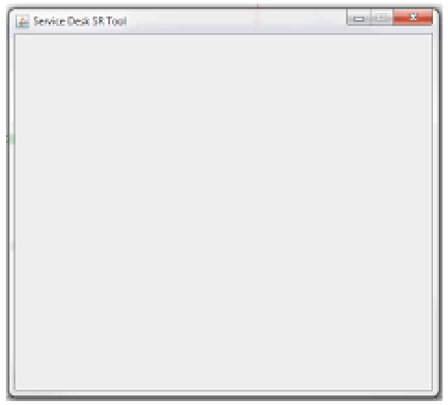
Provides a set of "lightweight" (all-Java language) components that, to the maximum degree possible, work the same on all platforms.

See: Description

Interface Summary

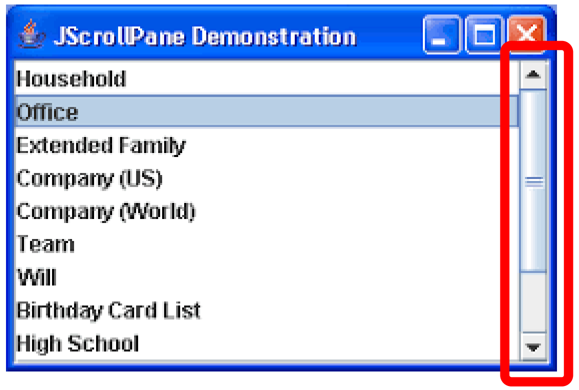
Interface	Description
Action	The `Action` interface provides a useful extension to the `ActionListener` interface in cases where the same functionality may be accessed by several controls.
BoundedRangeModel	Defines the data model used by components like `Sliders` and `ProgressBars`.
ButtonModel	State model for buttons.
CellEditor	This interface defines the methods any general editor should be able to implement.
ComboBoxEditor	The editor component used for JComboBox components.
ComboBoxModel<E>	A data model for a combo box.
DesktopManager	DesktopManager objects are owned by a JDesktopPane object.
Icon	A small fixed size picture, typically used to decorate components.
JComboBox.KeySelectionManager	The interface that defines a `KeySelectionManager`.
ListCellRenderer<E>	Identifies components that can be used as "rubber stamps" to paint the cells in a JList.
ListModel<E>	This interface defines the methods components like JList use to get the value of each cell in a list and the length of the list.
ListSelectionModel	This interface represents the current state of the selection for any of the components that display a list of values with stable indices.

Review: Java AWT/Swing concepts

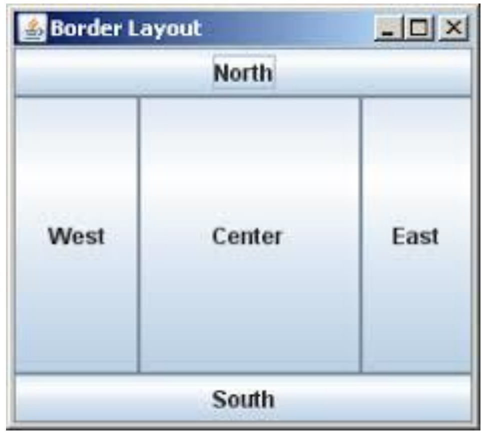


JPanel

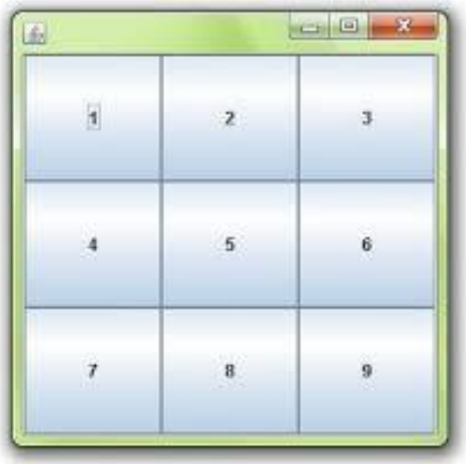
add()
setLayout()
revalidate()
removeAll()



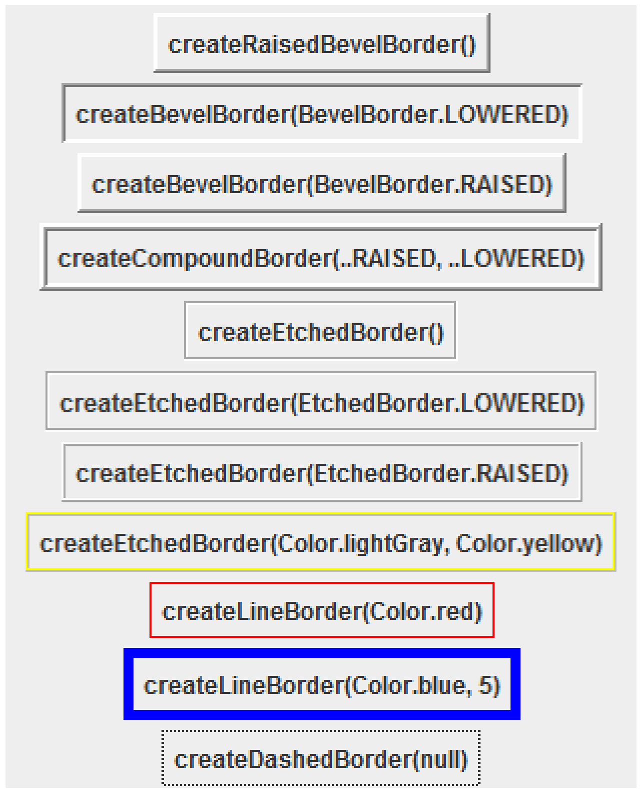
JScrollPane



BorderLayout

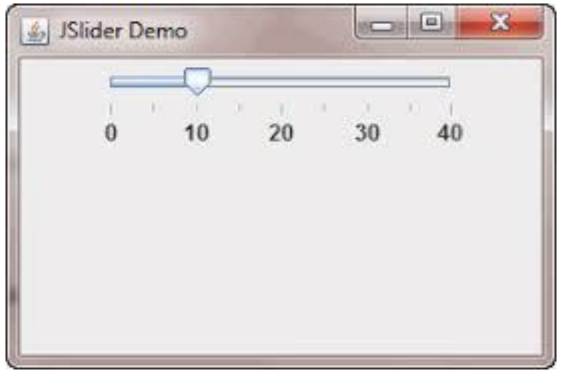


GridLayout


createRaisedBevelBorder()
createBevelBorder(BevelBorder.LOWERED)
createBevelBorder(BevelBorder.RAISED)
createCompoundBorder(..RAISED, ..LOWERED)
createEtchedBorder()
createEtchedBorder(EtchedBorder.LOWERED)
createEtchedBorder(EtchedBorder.RAISED)
createEtchedBorder(Color.lightGray, Color.yellow)
createLineBorder(Color.red)
createLineBorder(Color.blue, 5)
createDashedBorder(null)

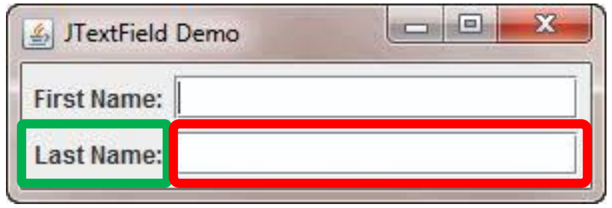
BorderFactory

Review: Java AWT/Swing concepts



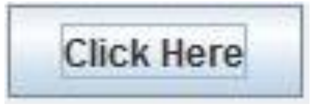
JSlider

getValue()



JLabel **JTextField**

getText()



JButton

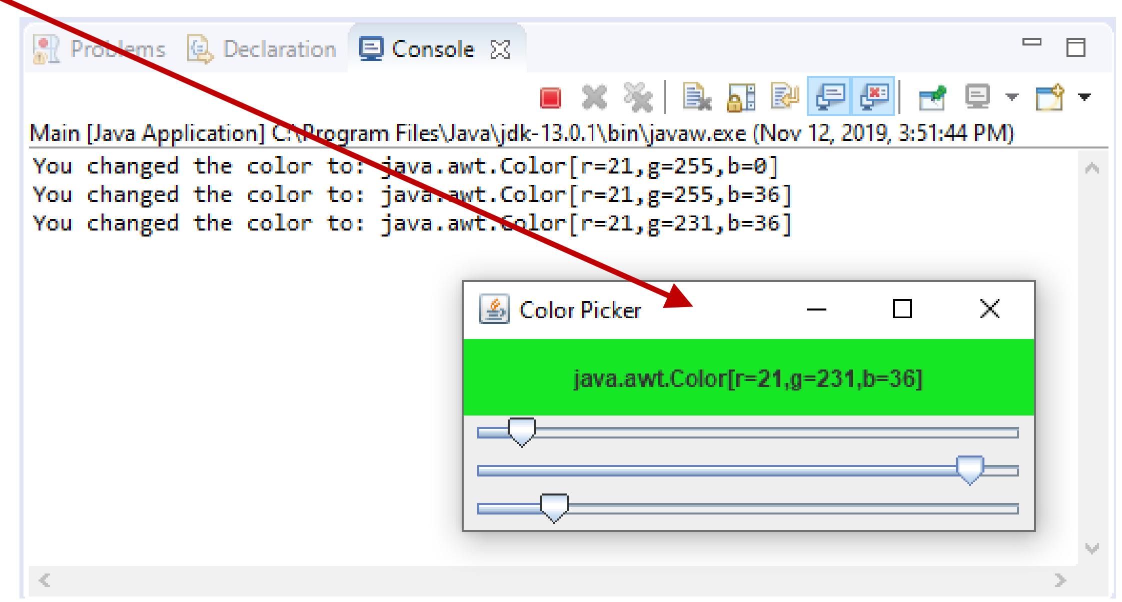
addActionListener()

setActionCommand()

Review: ColorChooser Widget

Featuring:

- **Java's AWT/Swing API**



```
Main [Java Application] C:\Program Files\Java\jdk-13.0.1\bin\javaw.exe (Nov 12, 2019, 3:51:44 PM)
You changed the color to: java.awt.Color[r=21,g=255,b=0]
You changed the color to: java.awt.Color[r=21,g=255,b=36]
You changed the color to: java.awt.Color[r=21,g=231,b=36]
```

java.awt.Color[r=21,g=231,b=36]

Review: ColorChooser Widget

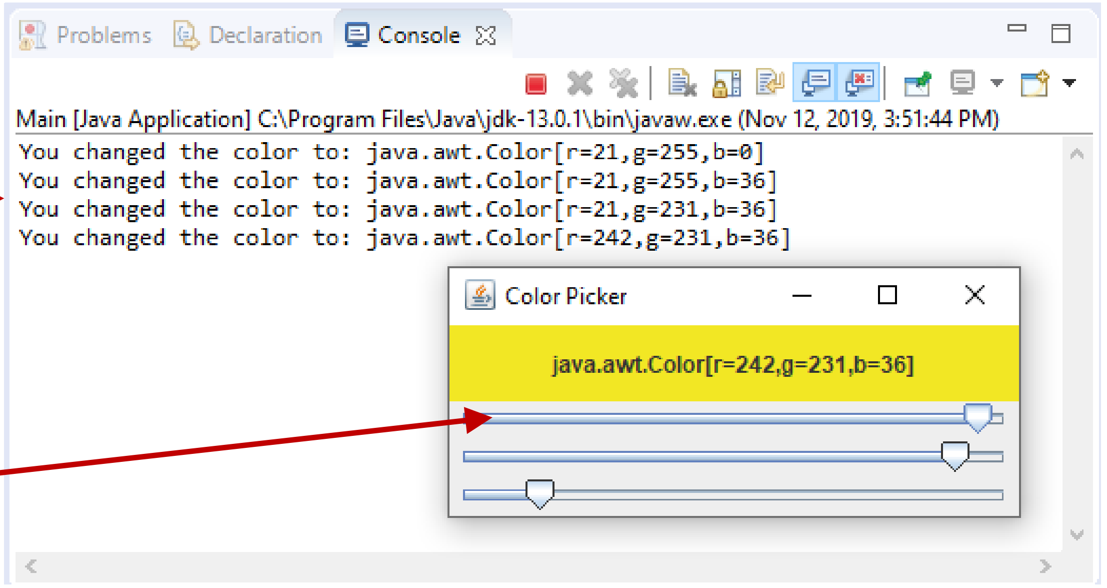
Featuring:

- Java's AWT/Swing API
- **Observer design pattern**

The **observer** design pattern is very common in user interface development

2. A method in your code is executed

1. User changes slider



```
Main [Java Application] C:\Program Files\Java\jdk-13.0.1\bin\javaw.exe (Nov 12, 2019, 3:51:44 PM)
You changed the color to: java.awt.Color[r=21,g=255,b=0]
You changed the color to: java.awt.Color[r=21,g=255,b=36]
You changed the color to: java.awt.Color[r=21,g=231,b=36]
You changed the color to: java.awt.Color[r=242,g=231,b=36]
```

java.awt.Color[r=242,g=231,b=36]

This class is a **JPanel** UI Component

It listens for **keyboard** events and **slider changes**

```
public class ColorChooser extends JPanel implements ChangeListener, KeyListener {

    Color color;
    JSlider red_slider;
    JSlider green_slider;
    JSlider blue_slider;
    JLabel color_label;
    List<ChangeListener> change_listeners;

    public ColorChooser(Color init_color) {
        // ...
    }


    // ...


}
```

Instance Variables

Job: Store object state

Constructor

Job: Set initial values for instance variables

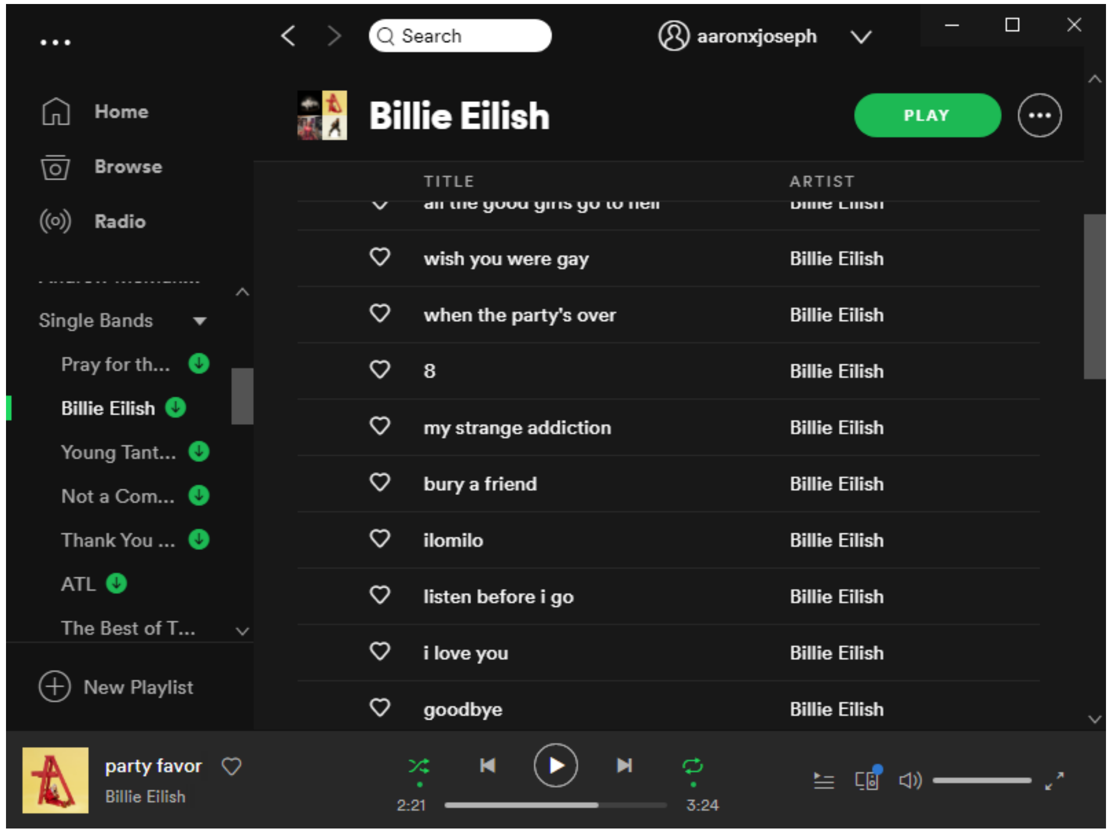
- Current color (**Color**)
- UI components (**JSlider**, **JLabel**)
- Observers (**List<ChangeListeners>**)

Q: *How can we improve this design?*

Imagine a more complicated widget…

…like a **song playlist** (think of Spotify)

Separation of concerns:
state vs **presentation**

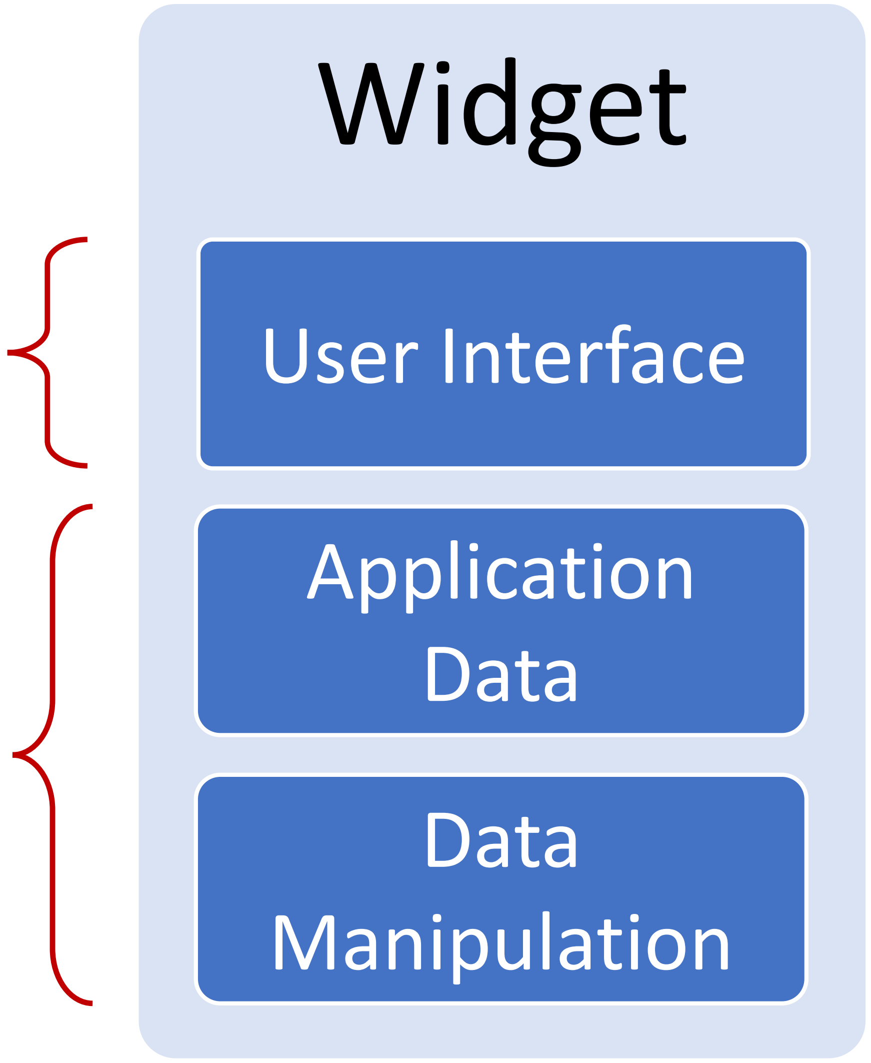


TITLE	ARTIST
all the good girls go to hell	Billie Eilish
wish you were gay	Billie Eilish
when the party's over	Billie Eilish
8	Billie Eilish
my strange addiction	Billie Eilish
bury a friend	Billie Eilish
ilomilo	Billie Eilish
listen before i go	Billie Eilish
i love you	Billie Eilish
goodbye	Billie Eilish

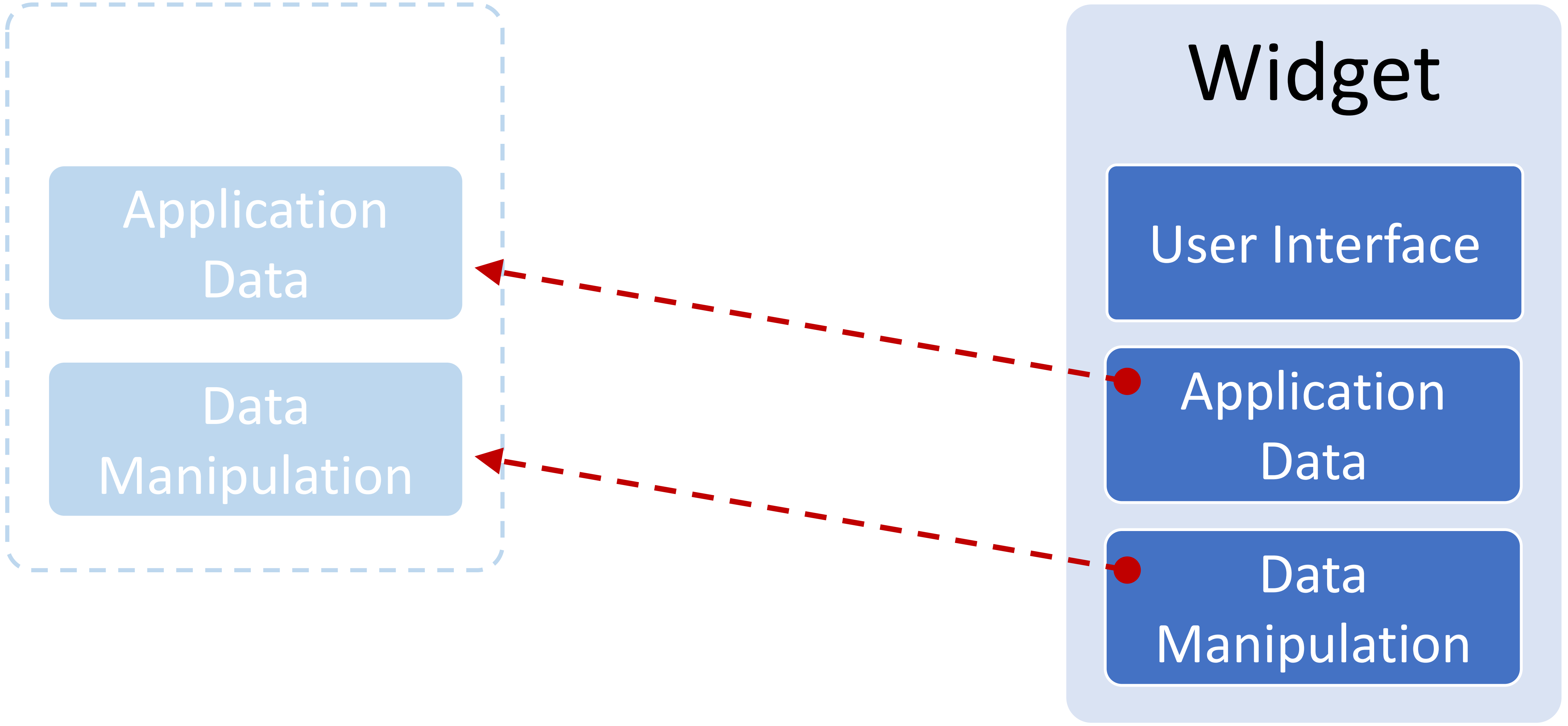
Model-View Design Pattern

...and this represents
how the state is **presented**

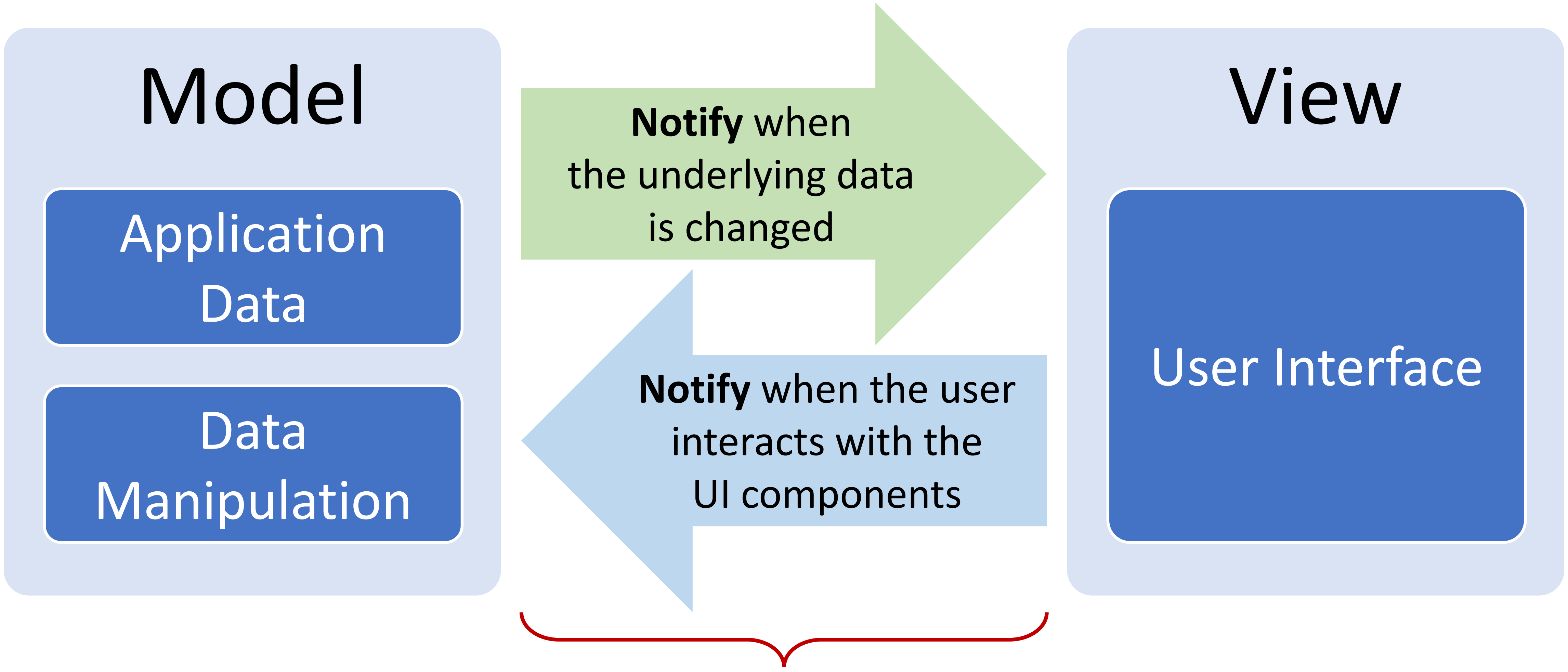
This represents
the **state** of the system...



Model-View Design Pattern

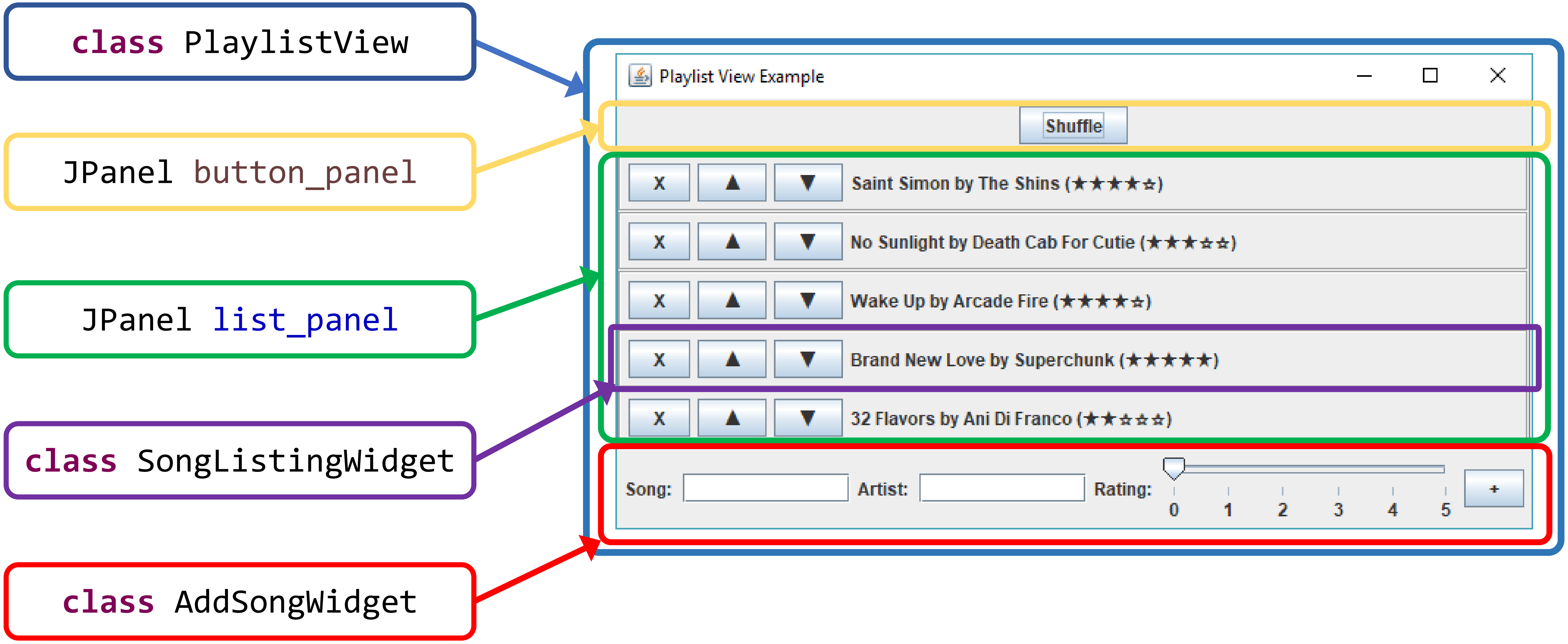


Model-View Design Pattern


Model
Application Data
Data Manipulation
Notify when the underlying data is changed
Notify when the user interacts with the UI components
View
User Interface

This is the **observer** design pattern

Today's MV Example: Song Playlist



Saint Simon by The Shins (★★★★☆)
No Sunlight by Death Cab For Cutie (★★★☆☆)
Wake Up by Arcade Fire (★★★★☆)
Brand New Love by Superchunk (★★★★★)
32 Flavors by Ani Di Franco (★★☆☆☆)

Q: What should the model be for this app?

In other words,

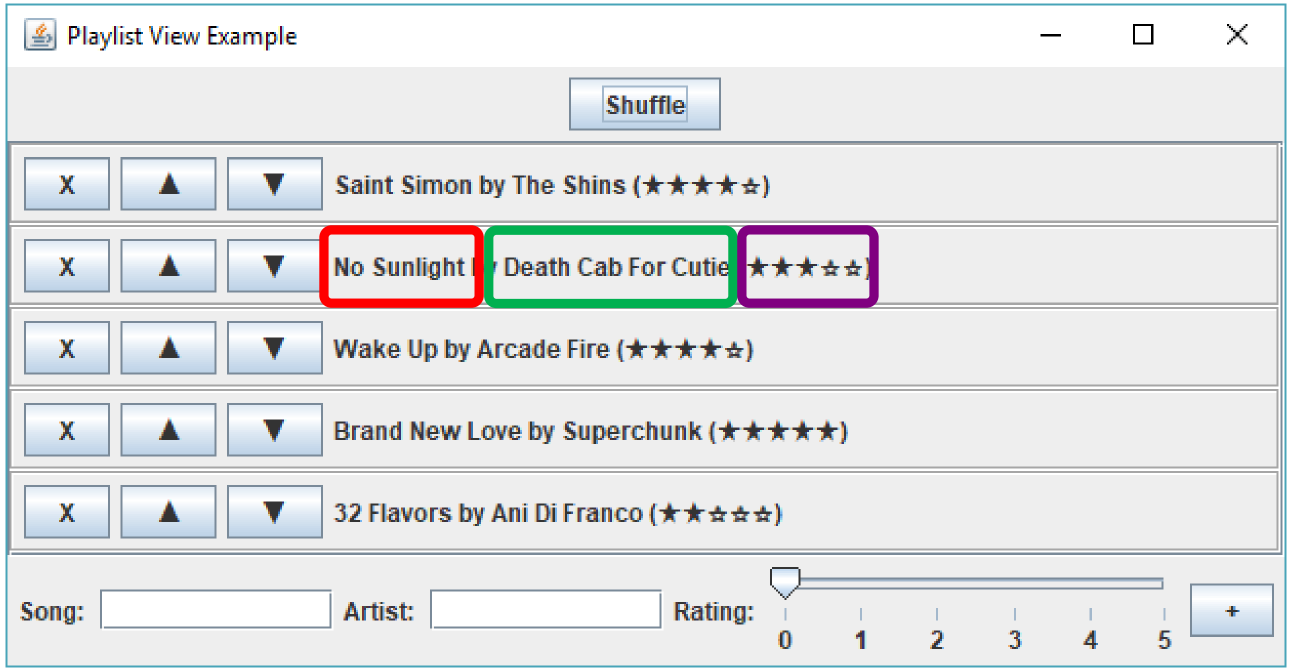
what **data state** is presented by this **view**?

A: A list of songs

(`List<Song>`)

Each song contains:

- **A title (`String`)**
- **An artist (`String`)**
- **A rating (`int`)**



Playlist View Example

Saint Simon by The Shins (★★★★☆)

No Sunlight by Death Cab For Cutie (★★★☆☆)

Wake Up by Arcade Fire (★★★★☆)

Brand New Love by Superchunk (★★★★★)

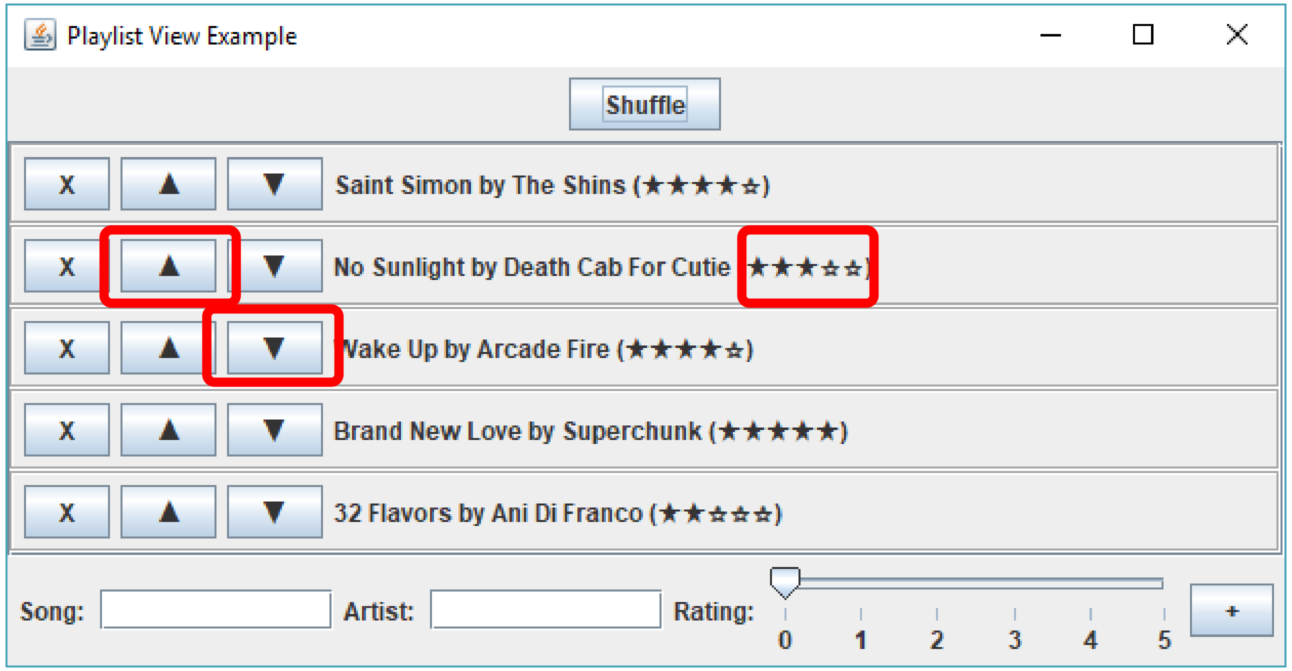
32 Flavors by Ani Di Franco (★★☆☆☆)

Unicode Characters

★	Unicode character U+2605 (`"\u2605"`)
☆	Unicode character U+2606 (`"\u2606"`)
▲	Unicode character U+25b2 (`"\u25b2"`)
▼	Unicode character U+25bc (`"\u25bc"`)

Q: How do we display the **rating stars** and the **arrows**?

A: With **Unicode** glyphs!


Playlist View Example

Shuffle

X	▲	▼	Saint Simon by The Shins (★★★★☆)
X	▲	▼	No Sunlight by Death Cab For Cutie (★★★☆☆)
X	▲	▼	Wake Up by Arcade Fire (★★★★☆)
X	▲	▼	Brand New Love by Superchunk (★★★★★)
X	▲	▼	32 Flavors by Ani Di Franco (★★☆☆☆)

Song: Artist: Rating: 0 1 2 3 4 5 +

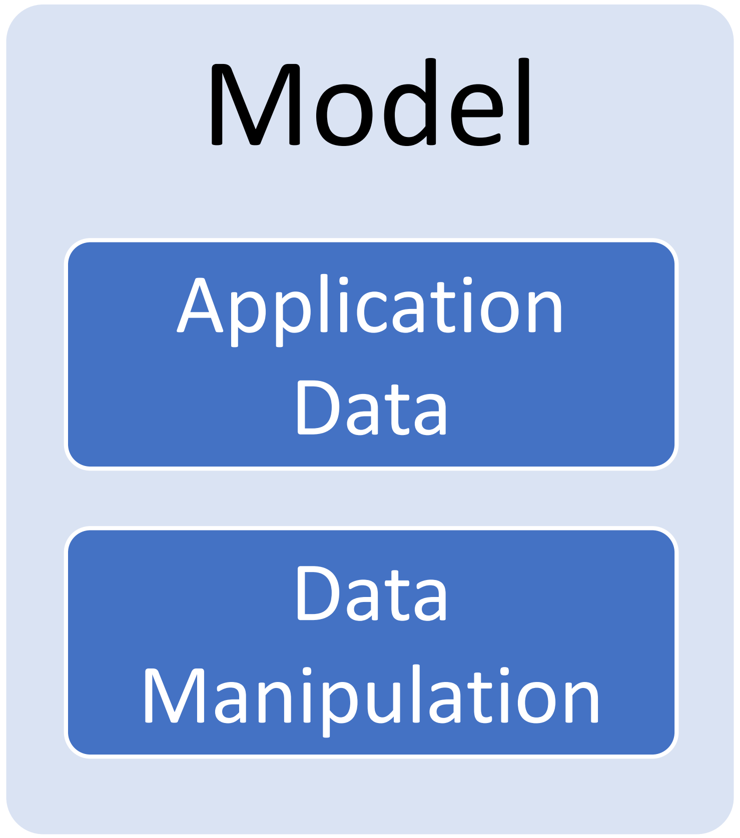
Version 1:

The Model

Playlist.java

Song.java

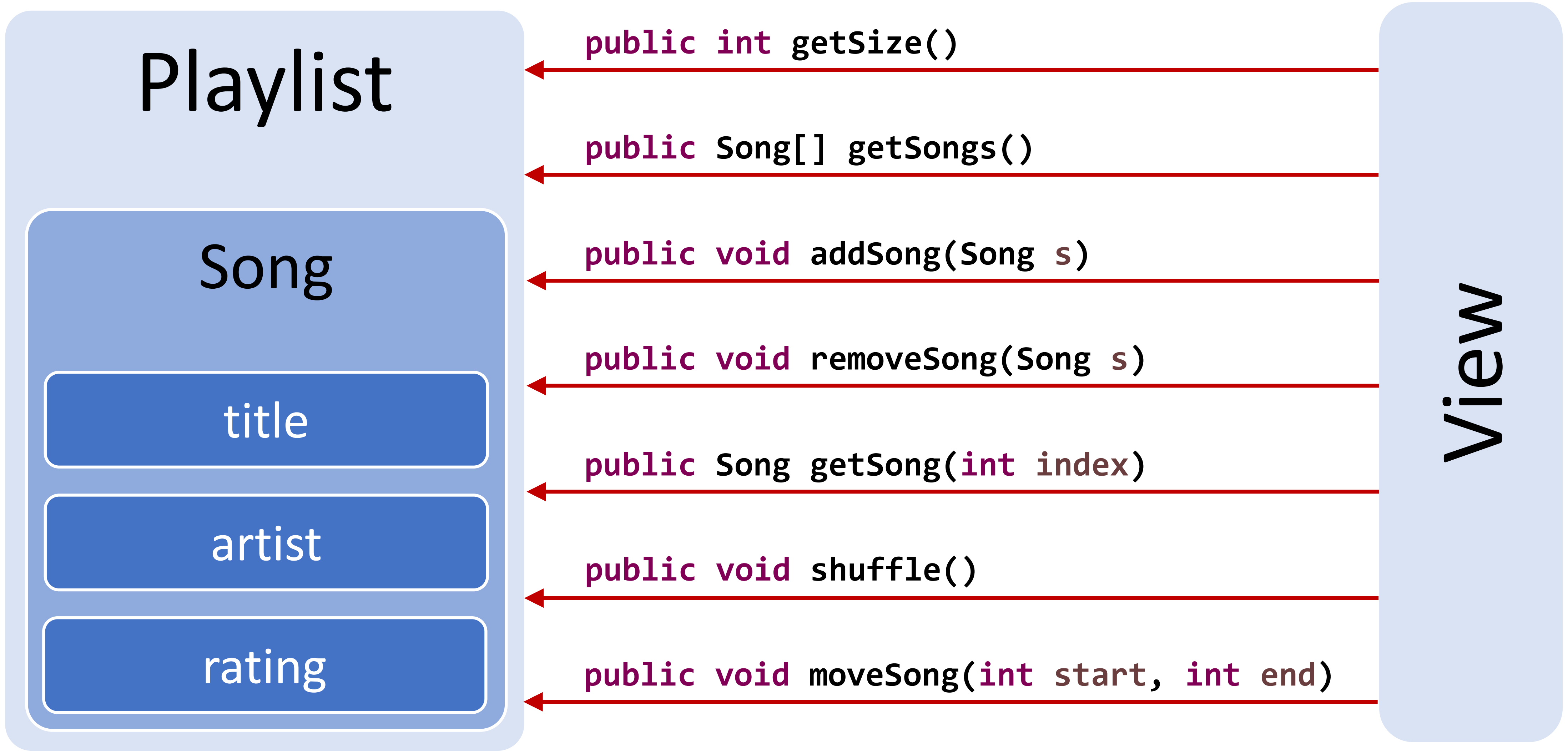
The model **exposes methods** to the data



The purpose of the model is to…

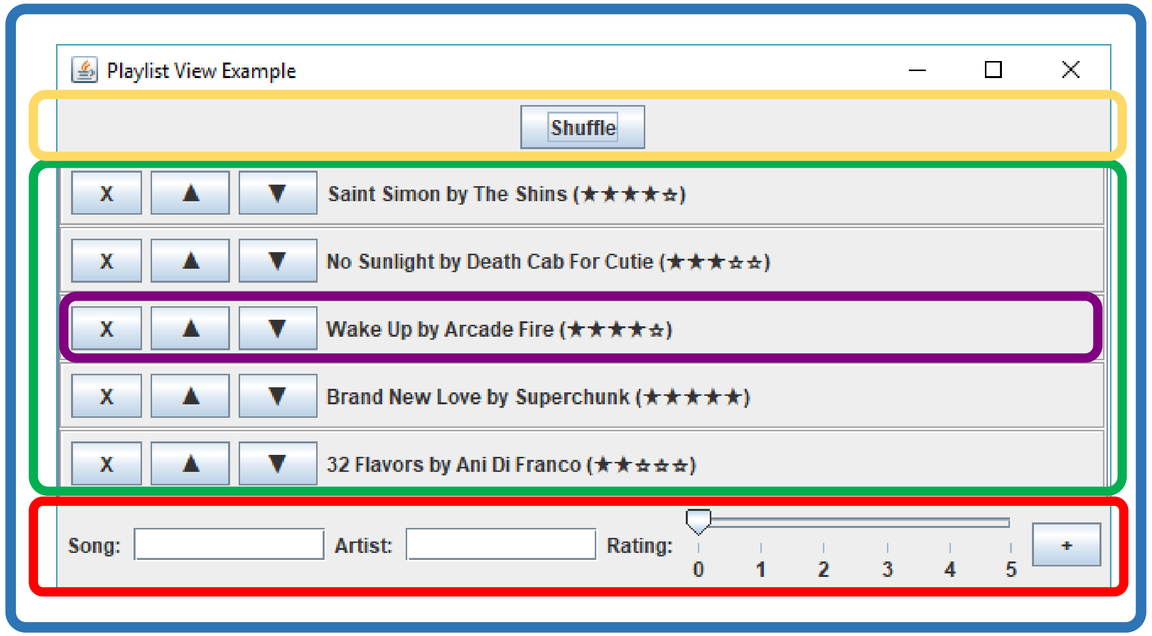
1. Store the data
2. Provide **methods** for data **access** and **manipulation**

The model exposes methods to the data

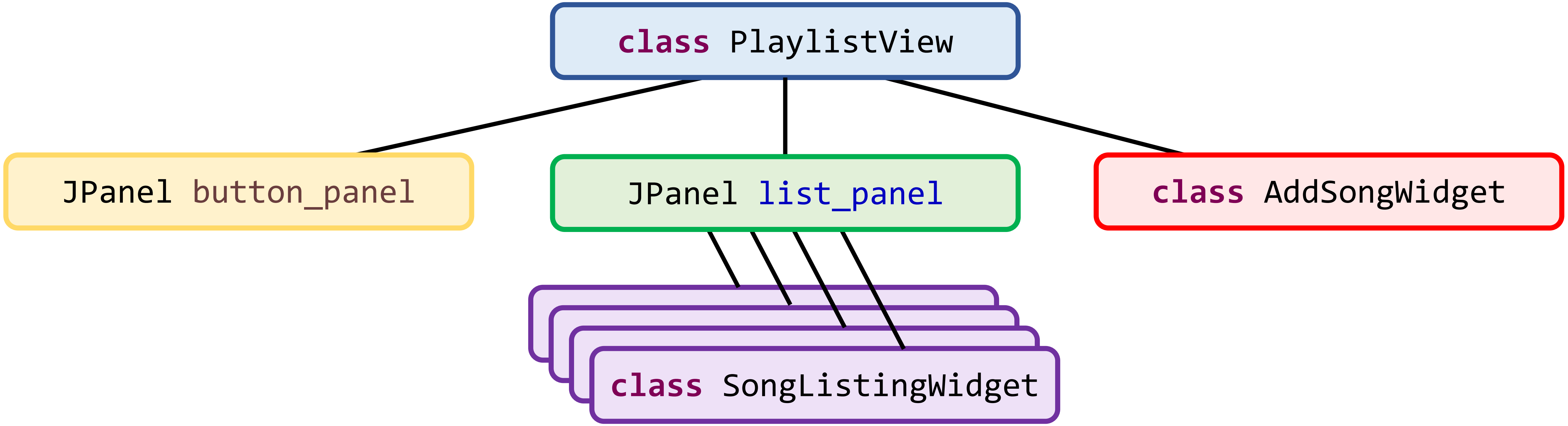


```
public int getSize()
public Song[] getSongs()
public void addSong(Song s)
public void removeSong(Song s)
public Song getSong(int index)
public void shuffle()
public void moveSong(int start, int end)
```

PlaylistView
Structure



Playlist View Example
Shuffle
X ▲ ▼ Saint Simon by The Shins (★★★★☆)
X ▲ ▼ No Sunlight by Death Cab For Cutie (★★★☆☆)
X ▲ ▼ Wake Up by Arcade Fire (★★★★☆)
X ▲ ▼ Brand New Love by Superchunk (★★★★★)
X ▲ ▼ 32 Flavors by Ani Di Franco (★★☆☆☆)
Song:
Artist:
Rating:
0 1 2 3 4 5
+



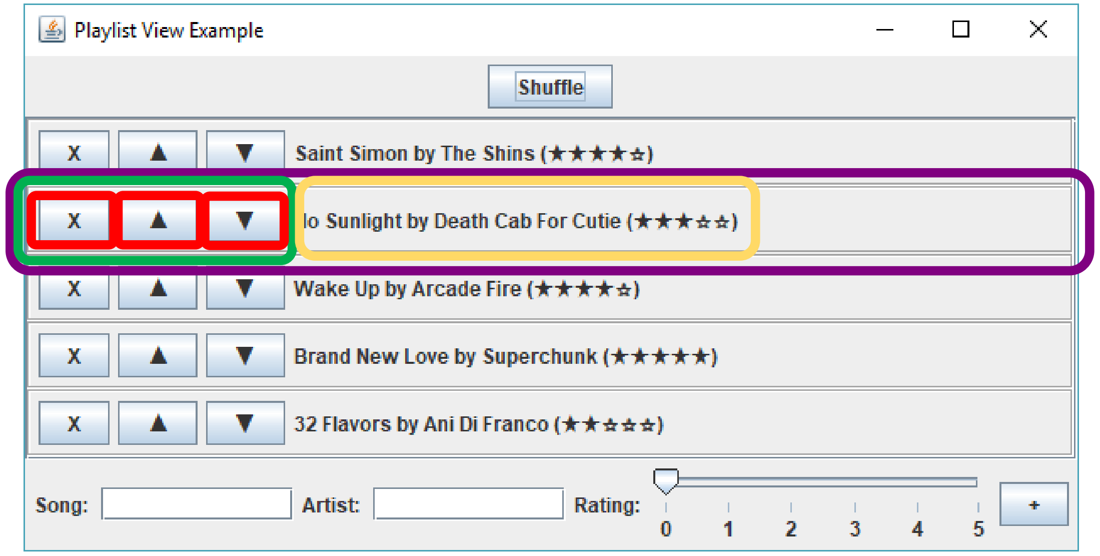
Version 2:

The View

PlaylistView.java

Main.java

SongListingWidget Structure



Playlist View Example

Shuffle

X ▲ ▼ Saint Simon by The Shins (★★★★☆)

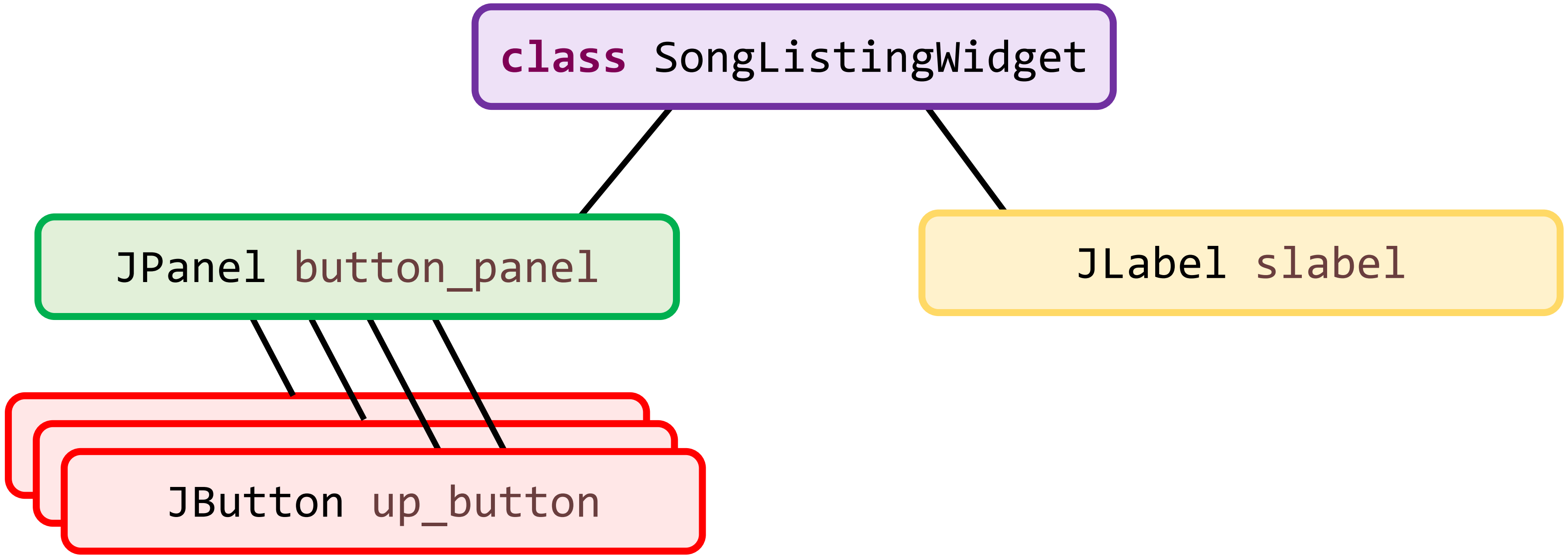
X ▲ ▼ Io Sunlight by Death Cab For Cutie (★★★☆☆)

X ▲ ▼ Wake Up by Arcade Fire (★★★★☆)

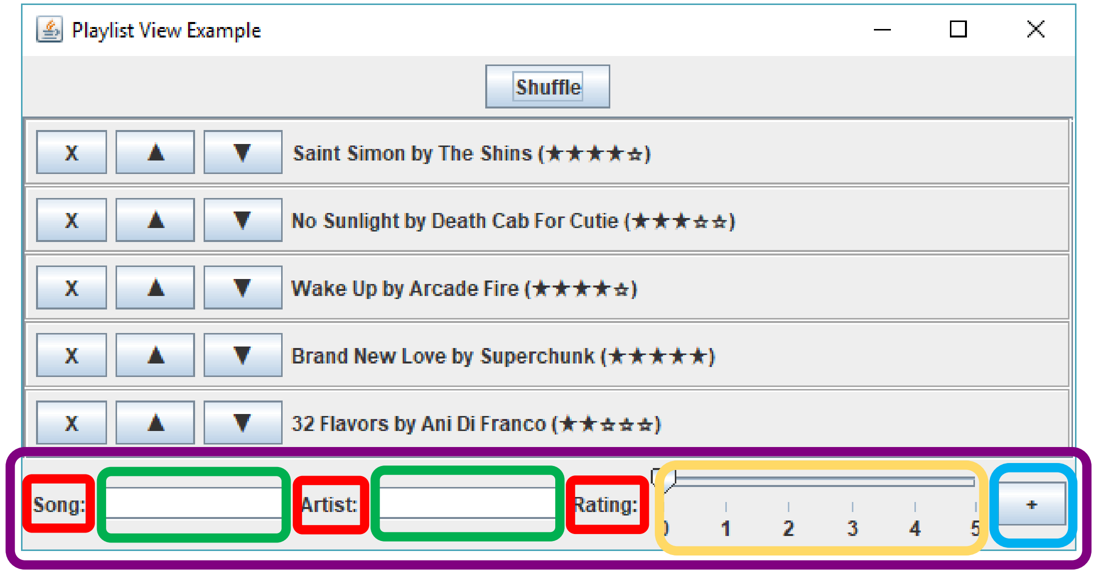
X ▲ ▼ Brand New Love by Superchunk (★★★★★)

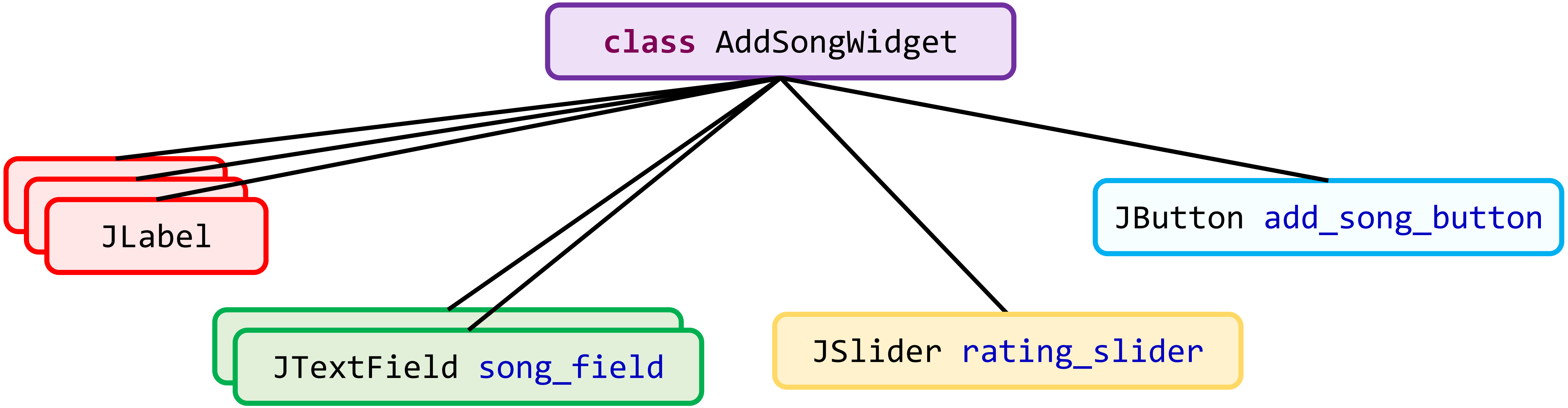
X ▲ ▼ 32 Flavors by Ani Di Franco (★★☆☆☆)

Song: Artist: Rating: 0 1 2 3 4 5 +



AddSongWidget Structure


Playlist View Example
Shuffle
X ▲ ▼ Saint Simon by The Shins (★★★★☆)
X ▲ ▼ No Sunlight by Death Cab For Cutie (★★★☆☆)
X ▲ ▼ Wake Up by Arcade Fire (★★★★☆)
X ▲ ▼ Brand New Love by Superchunk (★★★★★)
X ▲ ▼ 32 Flavors by Ani Di Franco (★★☆☆☆)
Song:
Artist:
Rating:
0 1 2 3 4 5
+



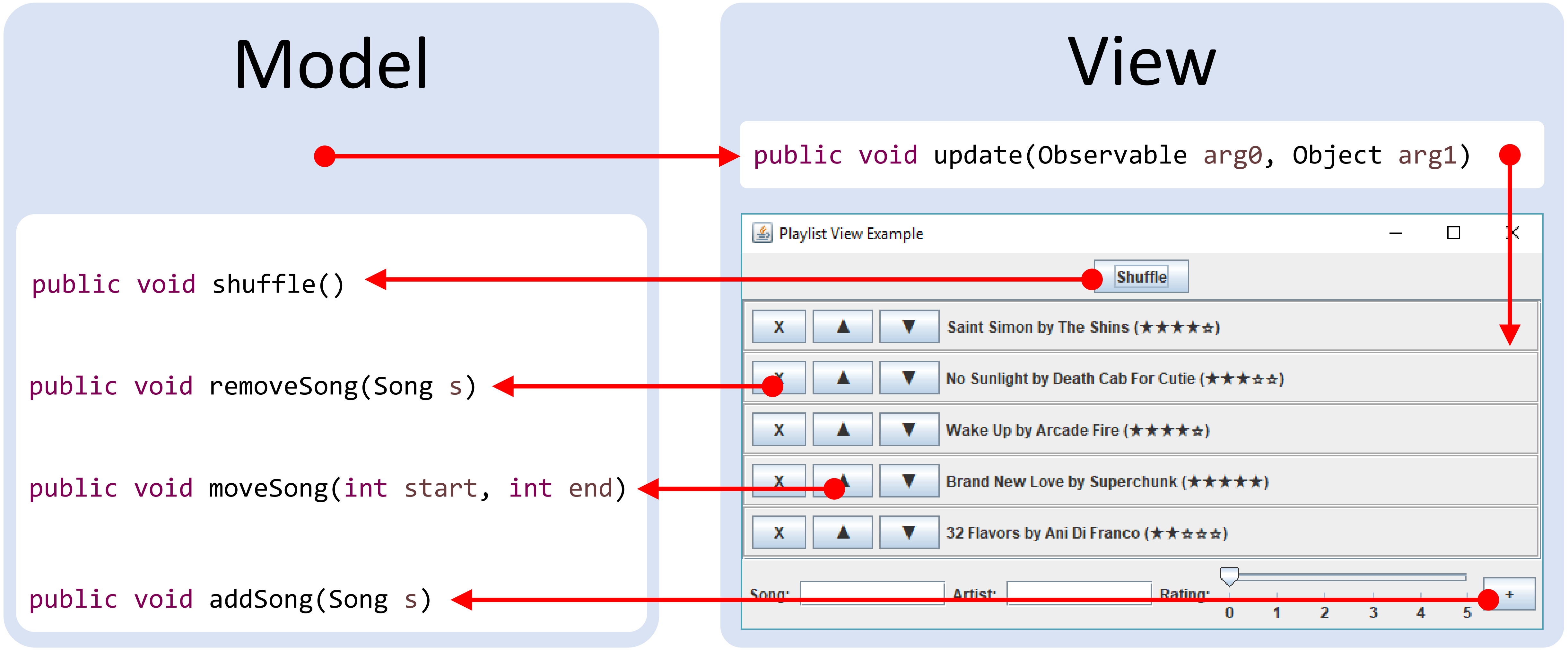
Version 3:

More View Pieces

AddSongWidget.java

SongListingWidget.java

Observer Data Flow



```
public void update(Observable arg0, Object arg1)
```

```
public void shuffle()

public void removeSong(Song s)

public void moveSong(int start, int end)

public void addSong(Song s)
```

Playlist View Example

Shuffle

X ▲ ▼ Saint Simon by The Shins (★★★★☆)

X ▲ ▼ No Sunlight by Death Cab For Cutie (★★★☆☆)

X ▲ ▼ Wake Up by Arcade Fire (★★★★☆)

X ▲ ▼ Brand New Love by Superchunk (★★★★★)

X ▲ ▼ 32 Flavors by Ani Di Franco (★★☆☆☆)

Song: Artist: Rating: 0 1 2 3 4 5 +

Version 4:

Connecting View and Model

Playlist.java

PlaylistView.java

Limitations of Model-View Design

Problem: Sometimes, user events require a more sophisticated response than updating raw Model data

Example: How would a song title and artist name **autocomplete** feature be implemented?

Solution: Make a **Controller** class to mediate between the View and Model