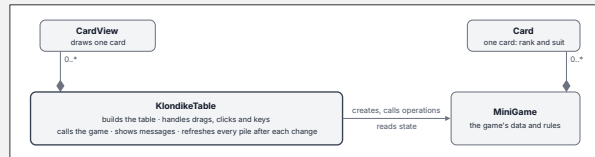


The Story So Far

- game state and logic (legal moves, carrying out moves) is contained in MiniGame but KlondikeTable handles GUI setup, events, repainting, ...

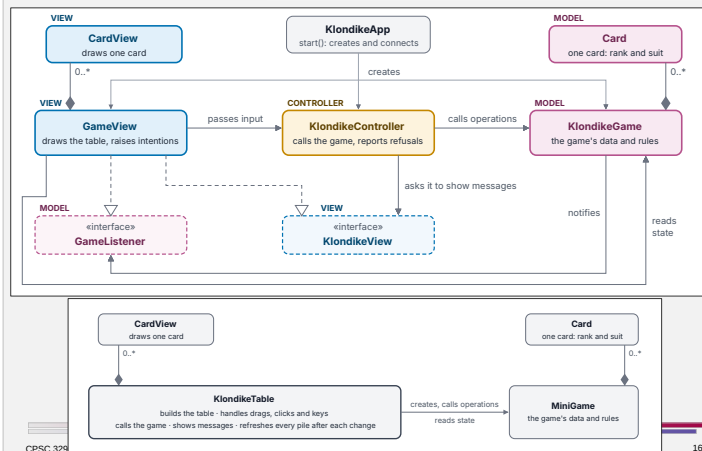


Implementing MVC in JavaFX

class	role	contains JavaFX?
KlondikeGame, Card	model (MiniGame renamed to KlondikeGame, but otherwise unchanged)	no
GameListener	the model's Observer interface	no
GameView	the table: listens to the game, draws it, detects gestures, raises intentions	yes
CardView	displays the card (unchanged)	yes
KlondikeView	what the controller may ask of a view: showMessage(text)	no
KlondikeController	handles intentions: calls the game, reports refusals	no
KlondikeApp	start(): creates and connects	yes

MVC in JavaFX

concrete listeners are part of the view, but GameListener is model – model must be independent of the view and needs GameListener to compile
KlondikeView defines capabilities of the view, so is part of view – if the capabilities change, the controller needs to change too



start() is Only Wiring

KlondikeApp
start(): creates and connects

```
@Override
public void start(Stage stage) {
    KlondikeGame game = new KlondikeGame(42);
    GameView view = new GameView();
    KlondikeController controller = new KlondikeController(game, view);

    game.addListener(view);           // model -> view
    view.setController(controller);    // view -> controller

    view.refresh(game);                // first drawing
    stage.setScene(new Scene(view));
    stage.show();
}
```

- start()'s job
 - create the model, the view, the controller
 - connect them
 - show the view
- no game rules, no drawing, no statements about cards
- every connection is visible in one place
 - reading this method tells you how the program fits together

Model

```
public class KlondikeGame {
    // state: stock, waste, columns, move count, last error, listeners
    private ArrayList<GameListener> listeners_ = new ArrayList<GameListener>();

    // Observer: registration and notification
    public void addListener(GameListener listener) { ... }
    public void removeListener(GameListener listener) { ... }
    private void fireGameChanged() { ... } // calls gameChanged on every listener

    // operations: check the rules, change the state, then notify
    public void drawFromStock() { ... }
    public boolean moveWasteTo(int column) { ... } // false, with a reason, if refused

    // queries: views read the state through these
    public Card getWasteTop() { ... }
    public Card getColumnTop(int column) { ... }
    public int getStockSize() { ... }
    public String getLastError() { ... }
}
```

- model was already separate – make it a subject and add getters for state

Controller

```
public class KlondikeController {
    private KlondikeGame game_;
    private KlondikeView view_;

    public KlondikeController(KlondikeGame game, KlondikeView view) {
        game_ = game;
        view_ = view;
    }

    public void drawFromStock() {
        view_.showMessage("");
        game_.drawFromStock();
    }

    public boolean playWasteTo(int column) {
        boolean moved = game_.moveWasteTo(column);
        if (moved) {
            view_.showMessage("");
        } else {
            view_.showMessage(game_.getLastError());
        }
        return moved;
    }
}
```

unchanged from the monolithic version, except for setting the message label text directly becomes showMessage and refreshTable goes away, replaced by the model notifying the view

- one public method per intention – each calls the game (model) to carry out the intention and decides what the user is told (via messages)
 - after a legal move it does nothing more – the game notifies its listener(s) (the view, which redraws)
 - a refused move changes nothing in the game, so showMessage to the view gives the reason

View

```
public class GameView extends StackPane implements KlondikeView, GameListener {
    public GameView() { /* build stock, waste, columns, background, message */ }

    @Override
    public void gameChanged(KlondikeGame game) { // the game changed: redraw
        refresh(game);
    }

    public void refresh(KlondikeGame game) { // redraw every pile from game state
        Card top = game.getWasteTop();
        if (top == null) {
            waste_.setVisible(false);
        } else {
            waste_.setCard(top.toString(), top.isRed());
            waste_.setVisible(true);
        }
        // ... stock opacity, the seven column tops
    }

    @Override
    public void showMessage(String text) { // what the controller may ask for
        message_.setText(text);
    }
}
```

- this is start() and refreshTable() now in a class of their own
- scene graph is built once (in the constructor), appearance is refreshed from game state (model) – gameChanged calls refresh to redraw everything
 - reads the model but never changes it
- keeps only the view state – keyboard cursor, drag highlights, focus
- showMessage is the one thing the controller can ask

View

handlers are set up in the constructor

```
public GameView() {
    ...
    stock_.setOnMouseClicked(e -> controller_.drawFromStock()); // click stock: "draw card"
    setUpDragSource(); // drag start and end: dim the waste card
    setUpDropTargets(); // drop on a column: "play the waste card there"
    setUpKeyboard(); // Enter: the same intention; D: "draw a card"
    ...
}

// inside setUpDropTargets(), which loops over the seven column targets
// v0: the drop handler does the work itself
target.setOnDragDropped(e -> {
    boolean moved = playWasteTo(col); // calls the game, shows the message, refreshes
    e.setDropCompleted(moved);
    e.consume();
});

// v1: the drop handler passes the intention to the controller
target.setOnDragDropped(e -> {
    controller_.playWasteTo(col); // "play the waste card to this column"
    e.setDropCompleted(true);
    e.consume();
});
}
```

- event handlers either change view's own state or pass an intention to the controller – they don't call the model directly
 - raise intention = translates user mouse/keyboard interaction into the game action it represents
 - controller is independent of how the user interface gets user input

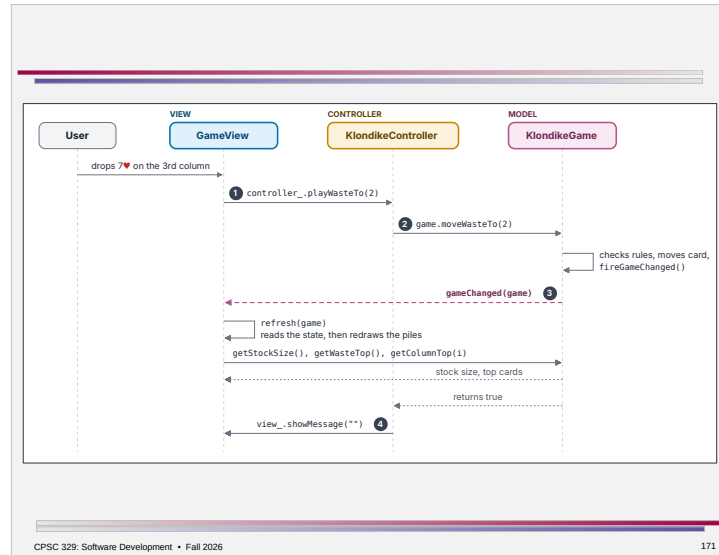
View

```
// GameView
private KlondikeController controller_; // set by start()

public void setController(KlondikeController controller) {
    controller_ = controller;
}

// start()
GameView view = new GameView();
KlondikeController controller = new KlondikeController(game, view);
view.setController(controller);
```

- the controller needs the view in its constructor, so the view gets the controller afterwards through a setter
 - until then `controller_` is null – `start()` sets it before the window appears
- the view calls the controller's public methods, one per intention
 - the same way the monolithic version calls its own private methods



Handling Refresh

- full or incremental refresh?
- full – `refresh(game)` redraws every pile after every change
 - simple
 - screen can't become out of sync with the game
 - cheap enough when there's only a few dozen nodes in the scene graph
- incremental – redraw only what changed
 - needs to know what changed – the model pushes the details or the view keeps a copy of what it last drew and compares
 - view keeps copy of what it last drew
 - worth it for large views (speed), necessary for animation
 - animation needs to know the "before" to animate from – a full refresh only knows the new state and can only snap there

Pitfalls

- feedback loops – a control that both shows model state and changes it
 - e.g. a draw 1 / draw 3 ComboBox
 - `refresh` sets its value → its handler fires → the controller changes the game → the game notifies → `refresh` sets the value again → ...
 - fixes
 - set the control only if the value differs, or
 - ignore events while refreshing (boolean `refreshing_` flag)
- the wrong thread – a listener runs on whatever thread changed the model
 - currently every change comes from a handler, on the JavaFX Application Thread
 - but if it comes from a timer or network thread, special handling is needed – UI updates need to be done on the JavaFX Application Thread

Properties and Bindings

- most settings of a JavaFX node are *properties* – a value plus a list of listeners
- every property has three methods – `getX()`, `setX(...)`, `xProperty()`

```
label.getText()           // read the value
label.setText("Hello")    // change it; the listeners are told
label.textProperty()      // the property object itself, to listen to or bind
```

- using properties
 - a *listener* reacts to the value changing, no matter who changed it
 - a *binding* keeps the target equal to the source

Properties and Bindings

- using properties, example from GameView –

```
// listen to it: redraw the keyboard cursor ring whenever the board gains or
// loses focus
tableauLayer.focusedProperty().addListener((obs, wasFocused, isFocused) ->
showCursor());
```

- a *listener* reacts to the value changing, no matter who changed it (e.g. tab, click, `requestFocus()`)
 - an event handler like `setOnAction` reacts only to a specific user action
- listeners are only really necessary for focus – there is no event handler for focus

Properties and Bindings

- using properties, example from GameView –

```
// bind to it: the hint band is visible exactly when the check box is ticked
hint_.visibleProperty().bind(showHint.selectedProperty());
```

- a *binding* keeps the target equal to the source
 - a bound value can't be set by hand
- alternative: initialization + event handler

```
// start in sync with the control – checkbox is not checked to start
hint_.setVisible(false)
// event handler to update when checkbox is changed
showHint.setOnAction(e -> hint_.setVisible(showHint.isSelected()));
```

Where Properties Belong

- use properties for UI-to-UI wiring inside a view



- when one control drives another
 - e.g. show hint check box displays the hint band (or not)
 - e.g. a message label only shows when it has text

```
// message visibility tied to its content
message_.visibleProperty().bind(message_.textProperty().isNotEmpty());
// 'managed' determines whether or not the component is part of the layout
// only include the message in the layout when it is visible
message_.managedProperty().bind(message_.visibleProperty());
```

- for view state – things only the display knows about
 - focus – e.g. show the keyboard cursor only while the tableau has focus
 - hover – e.g. highlight the column under the mouse
 - selection – e.g. enable Start button only once a deal is picked from a list

```
// focus
tableauLayer_.focusedProperty().addListener((obs, was, is) -> showCursor());
// hover
hoverRing.visibleProperty().bind(column.hoverProperty());
// selection
start.disableProperty().bind(deals.getSelectionModel().selectedItemProperty().isNull());
```

Where Properties Belong

- do not use properties in the model
 - model should stay JavaFX-free so it can be tested without JavaFX and be reusable in a console version or a server
 - properties bound to the screen must change on the JavaFX Application Thread
- to have properties bound to game data, use MVVM
 - have a *view model* with properties that listens to the model via `GameListener`



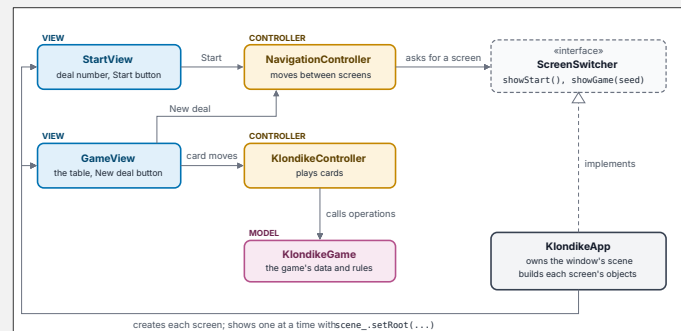
Multiple Screens

- e.g. a start screen (choose game parameters) and the game screen
 - switch start screen → game screen when start button is clicked
 - switch game screen → start screen when new game button is clicked
- one stage, one scene – the window stays the same, only what is inside changes
 - each screen is a view class (like `GameView`)
 - switch screens by changing the scene's root – `scene.setRoot(...)`
 - using `setRoot()` keeps the window size and position – only the contents are changed
- use a second stage for a separate window, not a new screen within the same window

Implementation

- starting a game and setting up a new game are user intentions, so the `setRoot(...)` calls go into a controller
- organize controllers by responsibility
 - a *game controller* exists for the duration of one game and handles playing it
 - `drawFromStock()`, `playWasteTo()`
 - an *app* (or *navigation*) *controller* exists for the duration of the application's run and handling moving between screens
 - `start()`, `newGame()`
- a single controller for everything risks becoming a god class
- an alternative is one controller per screen
 - convenient: easy to find code, each view knows about exactly one controller, adding a new screen means adding a new controller, facilitates splitting team work by screen
 - drawbacks: controllers have mixed jobs and grow large for big screens, shared behavior on multiple screens gets duplicated, navigation is spread out among multiple classes

Organization



Implementation

- controllers have no JavaFX
- KlondikeApp owns the scene
- controllers only depend on an interface

```
public interface ScreenSwitcher {  
    void showStart();  
    void showGame(int seed);  
}
```

Implementation

```
public class NavigationController {  
    private ScreenSwitcher screens_;  
  
    public NavigationController(ScreenSwitcher screens) {  
        screens_ = screens;  
    }  
  
    public void startGame(int seed) { // from the start screen  
        screens_.showGame(seed);  
    }  
  
    public void newDeal() { // from the game screen  
        screens_.showStart();  
    }  
}
```

Implementation

```
public class KlondikeApp extends Application implements ScreenSwitcher {  
    private Scene scene_;  
    private StartView startView_;  
  
    @Override  
    public void start(Stage stage) {  
        startView_ = new StartView();  
        startView_.setController(new StartController(this)); // this = the ScreenSwitcher  
        scene_ = new Scene(startView_, 600, 380);  
        stage.setScene(scene_);  
        stage.show();  
    }  
  
    @Override  
    public void showStart() {  
        scene_.setRoot(startView_);  
    }  
  
    @Override  
    public void showGame(int seed) {  
        KlondikeGame game = new KlondikeGame(seed);  
        GameView view = new GameView();  
        KlondikeController controller = new KlondikeController(game, view, this);  
  
        game.addListener(view); // model -> view  
        view.setController(controller); // view -> controller  
        view.refresh(game); // first drawing  
        scene_.setRoot(view);  
  
        // starting a new game means a new game controller - avoids state controller problems  
    }  
}
```

MVC in JavaFX – Summary

- `start()` creates and connects, nothing else
- view classes –
 - build their scene graph once – except possibly for leaf nodes
 - listen to the model for notifications of game state updates
 - refresh pulls from the state
 - pass user intentions to their controller
- controller turns intentions into game operations and report refusals (errors)
 - contains no JavaFX, knows about view only through interfaces
 - one controller per responsibility
- `scene.setRoot` switches screens
- properties and bindings are JavaFX's Observer
 - for UI-to-UI wiring – necessary for focus, handy for keeping UI elements in sync otherwise
- controller is testable with a fake view
- a fake listener tests the model's notifications