

A Problem

- when moves are made in Klondike, several things may need updating
 - the table display
 - the status line ("Stock: 12 cards")
 - a move counter, a timer, a stats window, ...

```
public boolean moveWasteTo(int column) {
    // ... make the move, including checking that it is legal ...
    tableView.refresh();
    statusLine.refresh();
    moveCounter.refresh();
    return true;
}
```

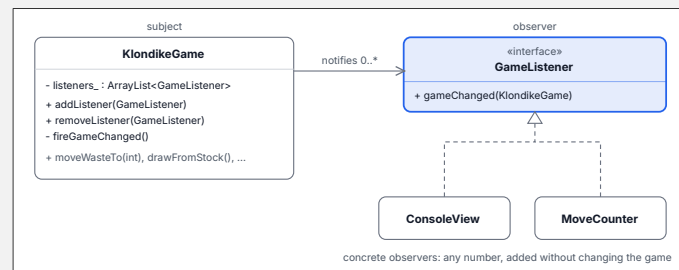
- the game depends on every view – it can't be tested, reused, or compiled without them
- adding a view means editing many places in the game

Observer

one change, many interested parties

- intent: when one object changes, notify any number of other objects without the changing object depending on them
- roles
 - Subject (Observable) – keeps a list of observers and notifies them
 - Observer – interface with the notification method(s)
 - concrete observers – any classes that implement the Observer interface
- three operations on the subject – register, unregister, notify
- also known as *publish-subscribe* and *listener*
 - "listener" is the Java/JavaFX name

Observer



- Java's terminology
 - observer → listener
 - register → addListener
 - unregister → removeListener
 - notify → fire...

Subject – Code

```
public interface GameListener {
    void gameChanged(KlondikeGame game);
}

public class KlondikeGame {
    private ArrayList<GameListener> listeners_ = new ArrayList<GameListener>();

    public void addListener(GameListener listener) {
        listeners_.add(listener);
    }

    public void removeListener(GameListener listener) {
        listeners_.remove(listener);
    }

    private void fireGameChanged() {
        for (GameListener listener : listeners_) {
            listener.gameChanged(this);
        }
    }

    public boolean moveWasteTo(int column) {
        // ... check the rules, handle invalid move ...
        columns_.get(column).add(waste_.remove(waste_.size() - 1));
        moves++;
        fireGameChanged();           // after the state is fully updated
        return true;
    }
}
```

KlondikeGame has no idea who is listening and contains no JavaFX

Observers – Code

```
public class ConsoleView implements GameListener {
    @Override
    public void gameChanged(KlondikeGame game) {
        System.out.println("Stock " + game.getStockSize()
            + " Waste " + game.getWasteTop());
    }
}

public class MoveCounter implements GameListener {
    @Override
    public void gameChanged(KlondikeGame game) {
        ...
    }
}

KlondikeGame game = new KlondikeGame(42);
game.addListener(new ConsoleView());
game.addListener(new MoveCounter());
game.drawFromStock(); // both observers react
```

any class can observe a game
adding a new observer doesn't require any changes to KlondikeGame

Push or Pull?

- the observers (likely) want to know more than just that there was a change – what changed?
- *pull* – the notification just says “I changed”, observers ask the subject for what they need
 - `gameChanged(KlondikeGame game)`, then `game.getStockSize()`, ...
 - simple
 - observers always see a consistent, complete state
 - an observer may redraw more than it needs to
- *push* – the notification carries the details of what changed
 - e.g. `cardMoved(Card card, int fromColumn, int toColumn)`
 - efficient and specific – observers can animate or log exactly what happened
 - the listener interface grows with every kind of change, and observers depend on those details

Granularity

- specific notification methods force every observer to implement all of them
- options
 - one general method – `gameChanged(game)`
 - every observer gets every notification and ignores the ones it doesn't care about
 - observer may do some wasted work, or have to figure out what changed
 - several specific methods with empty defaults
 - observer overrides only the ones it needs
 - e.g. AWT adapter classes (`MouseListener`, ...)
 - keeps specific methods, makes empty implementations someone else's problem
 - split interfaces
 - e.g. AWT/Swing `MouseListener`, `MouseMotionListener`, `MouseWheelListener`
 - keeps observers small, but subject needs add/remove and a listener list for each interfaces
 - one method, registered by event type
 - subject filters notifications sent by type, so observer only gets the ones it cares about
 - e.g. JavaFX `addEventHandler(MouseEvent.MOUSE_CLICKED, handler)`
 - more work for subject

Designing Notifications

- guidelines
 - start with pull
 - push only what an observer can't find out by asking afterwards
 - e.g. which card just moved, for an animation
 - start with one general method
 - add a specific one only if an observer really needs to tell that event apart from others
 - e.g. `gameWon`

Pitfalls

- notify only after the state is complete, never halfway through a change
 - observers may read subject state right away
- don't change the subject from inside a notification
 - change triggers another notification...
- unregister observers you no longer need – e.g. a closed window
 - otherwise the subject keeps them alive and keeps updating them
- don't depend on notification order
 - observers are independent of each other
- one observer's exception stops the notification loop for the rest

Observer is Not New

- JavaFX is full of Observer
 - `button.setOnAction(handler)`
 - the button is the subject
 - the handler is the observer
 - `setOnAction` registers it, a click notifies it
 - the button doesn't know what your handler does
 - `addEventHandler(...)` allows registering several handlers for the same event – many observers
 - `setOnAction` sets a specific property (single observer), so subsequent calls replace previous settings – use `addEventHandler` to register multiple handlers for `ActionEvent`
 - the handler gets an event object – `handle(ActionEvent e)`
 - push, sending a single object
 - `e.getSource()` supports pull
- event-driven programming is Observer applied to user input

From Pattern to Architecture

- Observer solved one problem – how the game tells the things that display it to update
- KlondikeTable still mixes three jobs in one class
 - drawing the cards and the table
 - interpreting drags, clicks, and keys
 - deciding what to do with those user actions – do game operations? show errors?
- this is not separation of concerns!

Architectural Style vs Design Pattern

- an *architectural style* extends code organizations to a whole application
 - patterns live inside an architecture

	architectural style	design pattern
scale	the whole system	a few classes
answers...	How is the system divided into big parts, and how do they communicate?	How do I solve this recurring design problem here?
examples	layered, client-server, pipe-and-filter, event-driven	Strategy, Template Method, Factory, Observer
changing it later	expensive: touches everything	local: a few classes

Model-View-Controller (MVC)

- splits an interactive program into three parts
 - the *model* – the program's data and rules
 - the *view* – what the user sees
 - the *controller* – turns user input into operations on the model
- architectural style or design pattern? yes
 - it organizes a whole application, like an architectural style
 - it is built from design patterns, Observer in particular (also Composite and Strategy)
- often called an *architectural pattern*

Model-View-Controller (MVC)

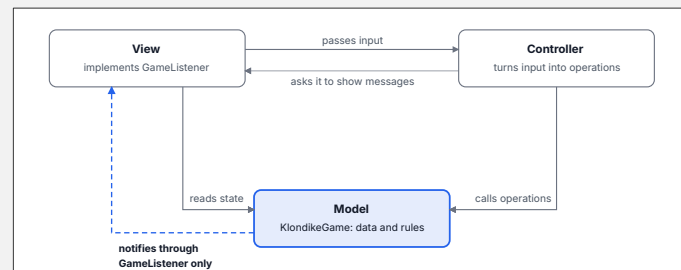
role	job	in Klondike
model	the data and the rules knows nothing about screens	KlondikeGame: piles, legal moves, move count
view	shows the model to the user	the table, the status line, a console printout
controller	turns user input into model operations	"waste card dropped on column 3" → <code>game.moveWasteTo(2)</code>

- the model is the part you keep if you throw the interface away
- the view and controller are the parts you rewrite for a new interface
 - console, desktop, web, phone, etc

MVC Interactions

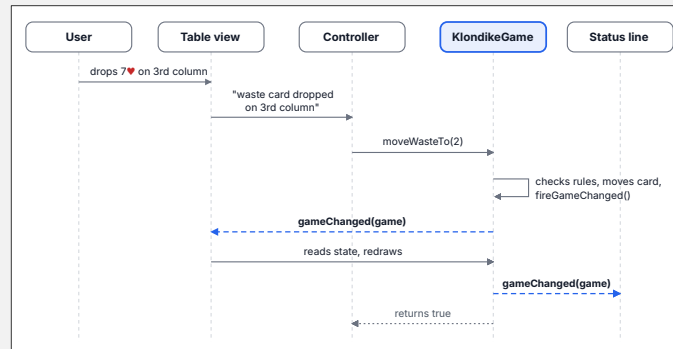
- controller calls model operations
 - e.g. `moveWasteTo`, `drawFromStock`
- view reads model state in order to display it
 - e.g. `getWasteTop`, `getStockSize`
- model tells the view to update – only through the listener interface
 - e.g. `fireGameChanged`
- view passes user input to the controller
 - e.g. "card dropped on column 3"
- controller asks view to show messages (things that aren't model state)
 - e.g. "show this error message"
- the model depends on nothing else – it compiles, runs, and can be tested on its own

MVC Interactions



- solid arrows are direct calls, dashed is through listener interface notifications only
- selection and cursor position (e.g. keyboard cursor) are view state, not game state
 - in view only, not model

Tracing a Move



- Table view and Status line are both view
- KlondikeGame is model
- no more `refreshTable()` or `refreshStatus()` calls

CP

15

Exercise – Model, View, or Controller?

1. Decide whether 7♥ may go on the 8♦
2. Draw the green felt
3. Keep track of which cards are in the stock
4. Show "12 cards left"
5. Turn the → key into "move the cursor one column right"
6. Turn "Enter pressed on a column" into a move
7. Count the moves
8. Highlight a column while a card is dragged over it
9. Shuffle the deck from a seed
10. Show a dialog when the game is won

CPSC 329: Software Development • Fall 2026

156

#	responsibility	role	reason
1	decide whether 7♥ may go on the 8♦	model	a game rule – true whatever the interface
2	draw the green felt	view	pure appearance
3	track which cards are in the stock	model	game state
4	show "12 cards left"	view, reading the model	the count is model state the words and where they appear are view
5	→ moves the keyboard cursor	view	the cursor is interface state, not game state – a console version wouldn't have it
6	"enter on a column" becomes a move	controller	turns input into a model operation
7	count the moves	model	part of the game's state (related to the score)
8	highlight a column during a drag	view	feedback about a gesture in progress
9	shuffle from a seed	model	determines the game – needed for reproducible deals
10	dialog when the game is won	model knows, view shows	the model reports "won" (a method or a gameWon notification) the view decides to show a dialog

CPSC 329: Software Development • Fall 2026

157

Why MVC?

- testable model
 - JUnit tests for the rules, no window or JavaFX required
- supports several views of one model
 - e.g. a table and a stats window
 - e.g. same game with a text-based console and a GUI
- changes stay contained
 - e.g. new card art only touches the view
 - e.g. a rule change only touches the model
- allows for parallel work (and specialization)
 - one person works on the model, another on the view
- but there is a cost
 - more classes and more indirection – can be overkill for small applications

CPSC 329: Software Development • Fall 2026

158

MVC Variations

all share the core idea – keep the model independent of the UI

- **MVC (model-view-controller)**
 - the controller handles user actions and calls the model
 - the view passes user input to the controller
 - the view listens to the model and reads the model's state
 - the controller knows about the view only to pass messages (e.g. errors)
 - the view knows about the model
- **MVP (model-view-presenter)**
 - the presenter handles user actions and calls the model
 - the view passes user input to the presenter
 - the presenter listens to the model, reads the model's state, tells the view what to display
 - the presenter knows about the view (through an interface)
 - the view does not know about the model
 - presenter can be tested with a fake view that records what it was told to show – allows JUnit tests for screen-update logic too
- **MVVM (model-view-viewmodel)**
 - the view model handles user actions and calls the model
 - the view passes user input to the view model
 - the view model holds display-ready observable values – it listens to the model and reads the model's state to update them
 - the view binds to the observable values – when a value changes, the view updates automatically
 - the view does not know about the model
 - the view model does not know about the view
 - the view model is testable without a view; the cost is that it needs a framework that supports bindings