My Observer Goes to 11
class Edge {
    Vertex a, b;
    Edge(a1, b1) {
        a = a1; b = b1;
        a1.insertEdge(this);
        b1.insertEdge(this);
    }
    void drop() {
        if (a.hasEdge(this)) a.dropEdge();
        if (b.hasEdge(this)) b.dropEdge();
        a = null; b = null;
    }
    …
}

class Vertex {
    Set<Edge> edges;
    …
    void addEdge(Edge e) { edges.add(e); }
    // no dangling edges!
    void dropEdge(Edge e) {
        edges.remove(e);
        e.drop();
    }
    …
}
What's wrong with this design?

class Edge {
    Vertex a, b;
    Edge(a1, b1) {
        a = a1; b = b1;
        a1.insertEdge(this);
        b1.insertEdge(this);
    }

    void drop() {
        if (a.hasEdge(this)) a.dropEdge();
        if (b.hasEdge(this)) b.dropEdge();
        a = null; b = null;
    }
    ...
}

class Vertex {
    Set<Edge> edges;
    ...
    void addEdge(Edge e) {
        edges.add(e);
    }
    // no dangling edges!
    void dropEdge(Edge e) {
        edges.remove(e);
        e.drop();
    }
    ...
}

The work (responsibility) of keeping the relationships correct is shared too much between the objects, resulting in a loop in the dependences and complex logic to avoid infinite recursion. This is the motivation for the Mediator Pattern.

A. It's too complicated (why?)
B. Violates SRP (how?)
C. There's a loop in the dependences (where & why bad?)
D. All of the above
Mediator Pattern

When in doubt add a level of indirection
myCity Story: Show Friends on Map

**Given** I am logged in  
**And** my location is known  
**When** my friends are online  
**And** my friends’ location is known  
**And** my friends are near me  
**Then** I see them on the map

- How get friends drawn on the map?  
- What **thing** does something to **itself** so friends get drawn on map?

```java
class Friend {
    Location location(); // null -> not-known  
    boolean online();  
    Bitmap image();  
}

class Friends implements Iterable<Friend> {  
    Friends online();  
    Friends locationKnown(); // -> online?  
}
```
Nearby Friends: Subclass Approach

- **Friends**
  - List<friend> friends
  - Friends online()
  - Friends locationKnown()

- **Friend**
  - boolean online()
  - Location location()
  - Bitmap image()

- **NearbyFriendsMap**
  - GoogleMap map
  - Friends friends
  - showNearbyFriends()

- **GoogleMap**
  - ...

- **Iterable**
  - ...

- **<interface>**
  - ...
Nearby Friends: Mediator Approach

- This is a mediator design
  - solves problem of complex inter-object relationship (friends <-> Map)
  - lets each focus on their single responsibility
Mediator Pattern

- **Name:** Mediator

- **Problem:** Object interactions or relationships are complex or interdependent

- **Solution:** Introduce an object whose responsibility is mediating the interactions or maintaining the relationships, removing those responsibilities from the mediated objects
Model-View Controller

Once again, the ten-cent version
How should NFT get called, by whom?

A. Rename `showNearbyFriends` to “update”
B. Put NFT in its own thread and have it poll friends for changes, call `sNF`
C. Observer pattern: Make `Friends` a “subject” and NFT an “observer” of it
D. Observer pattern: Make `Friend` a “subject” and NFT an “observer” of it
A: No. SNF is like the “display” method in ForecastDisplay. We need a separate “update” or “changed” method, which will call SNF.

B: You could do that, but it's massively inefficient, since Friend status changes rarely. Also, complex – hard to understand, debug, etc. Overkill.

C: Yes. Recall that Observer is called Listener in Android and Java in general.

D: No, that defeats the whole purpose of Friends as an aggregator class. See C.
How should NFT get called, by whom?

How does NearbyFriendsTracker get called when GoogleMap changes (e.g., you move)?

A. Observer pattern: have NFT observe your location through LocationManager

B. Observer pattern: Make Friends a “subject” and NFT an “observer” of it

C. Observer pattern: Make GoogleMap a “subject” and NFT an “observer” of it
A: No. That’s just one way the view port on the map changes.

B: No. That was for dealing with changes to Friends online status and location, not map’s changing view port.

C: Yes. That’s the idea. Of course, GM is final, but it already has a register method and all, like LocationManager.
Mediator + Observer (x 2)

It’s more than Observer x 2 because NFT directly queries and manipulates Friends and GMap to maintain the relationship between the two. Observer just "observes", it doesn’t manipulate.
MVC Object Interactions

a NearbyFriendsTracker

friends.registerObserver(this)

gm.addMarker(…), etc.
gm.setOnCameraChangeListener(this)

onFriendMove(Friend), etc.
onCameraChange(CameraPosition)

a Friends

a GoogleMap
Who “knows about” who?

A. Friends knows NFT, which knows Gmap
B. Friends and GMap know about NFT
C. NFT knows about Friends and Gmap
D. NFT, Friends, & GMap all know about each other

**Knows about:** Type A knows about Type B if A calls or uses B in any way.
**Discussion of who knows about who**

- **Best Answer - C**: NFT knows about Friends and Gmap
  - Technically NFT doesn’t even know about Friends, just the FriendsSubject interface. In theory this allows NFT to integrate other implementations of “friends”, as long as they implement the FriendsSubject.

- It’s the only one making direct calls on the other objects
  - Although Friends and GMap make callbacks on NFT, those are methods first declared in the interfaces
  - Friends and GMap know about those interfaces, but not the classes that implement the methods

- This “separation of concerns” allows NFT to glue together Friends and GoogleMap, without either having to know about the other
  - They do have to anticipate that something wants to observe them, but they don’t know who that is, just that they implement their observers
Model-View Controller

- Terminology
  - Model: the “data”; also known as Subject (Friends)
  - View: the visualization of the data, often a UI (Gmap)
  - Controller: mediator that manages the relationships

- MVC handles two limitations of Observer
  - View is not just a passive view (e.g., a visualization)
    - a UI where the View changes to an alternate view
    - View makes changes to the Model
  - Model and View pre-exist
    - Can’t use Observer pattern because View doesn’t implement Observer interface for Subject/Model
    - So, what do you do when you want to connect two pre-existing, incompatible components?
Ten-Cent MVC: Class Diagram

```
ModelSubject
 register, notify, ...

Model
 ...

Controller

ViewSubject
 register, notify, ...

View
 ...

* interface ModelSubject
  register, notify, ...

* interface ModelListener
  update(...)  

* interface ViewListener
  update(...)  

definitions
```

The diagram illustrates the relationships and methods involved in the Model-View-Controller (MVC) design pattern. The ModelSubject interface is responsible for registering and notifying listeners. Model objects have methods like `update(...)`. ViewSubject objects also register and notify listeners, and Views interact with Controllers, typically through a ViewListener that `update(...)` method.
Design Patterns

Now experience doesn’t have to be a cruel teacher
SRP & 10-cent OO Design aren’t Enough

- Slow, focus on “external” entities (e.g., friends, maps)
- Experience can help us take big, fast leaps
  - But experience takes years to acquire
- Design Patterns method is a shared vocabulary of software design experience
  - Concise way to talk about designs
  - Focused on making software change easier
  - Reduce coupling caused by object interactions
  - When one class changes, causes another to change
  - When a “big” class has changes that are mixed in with parts that don’t
Design Pattern

A template for how to solve a problem that can be used in many different situations (wikipedia)

- A pattern consists of (at least):
  1. A name
  2. A problem statement
  3. A solution: description, example, class diagram

- Best practice: name our code after the pattern
  - E.g., NearbyFriendsTracker would be FriendsMapMediator or FriendsMapController (it’s more than an observer, since it updates Model and View)

- Used only as-needed (if it ain’t broke, don’t fix it)
  - Use of design patterns adds complexity (and takes precious time!)
  - Cost vs. benefit
Take-Aways

- **When in doubt, add a level of indirection**
  - Mediator, Controller (in MVC)
  - Achieving loosely coupled designs is difficult
  - A lot of added complexity – *use only as needed*

- **Design patterns method is a concise way to share experience and talk about designs**

- **Composite design patterns**
  - If an object collaboration has multiple problems, it may require multiple patterns
  - Patterns can be blended together to address all the problems
  - Here: Observer and Mediator