Patterns

Topics

1) How to best loop through some items?
2) How to best notify an object of a change?
3) How to best organize classes in an application?
4) How can design ideas be reused?

ood:
1.design patterns
2.design principles
3.design techniques:
ex: dependency injection - things we use to facilitize our design

Iterator

Accessing Items in a Collection

Java Iterator

```
List<String> data = // <snip>

Iterator<String> itr = data.iterator();
while (itr.hasNext()) {
  String word = itr.next();
  // <snip>

}
```

Direct Link List Code

```
List<String> data = // <snip>
LinkedList
Node n = data.head();
while (n != null) {
  String word = n.getData();
  // <snip>
  n = n.nextNode();
}
```

- What changes when switch to an ArrayList?
  - Using an iterator:..no change
  - Direct access:..change to index-iteration loop
- What changes when switch to an binary tree?
  - Using an iterator:..no change
  - Direct access:..change to recursive traversal

Iterator Idea

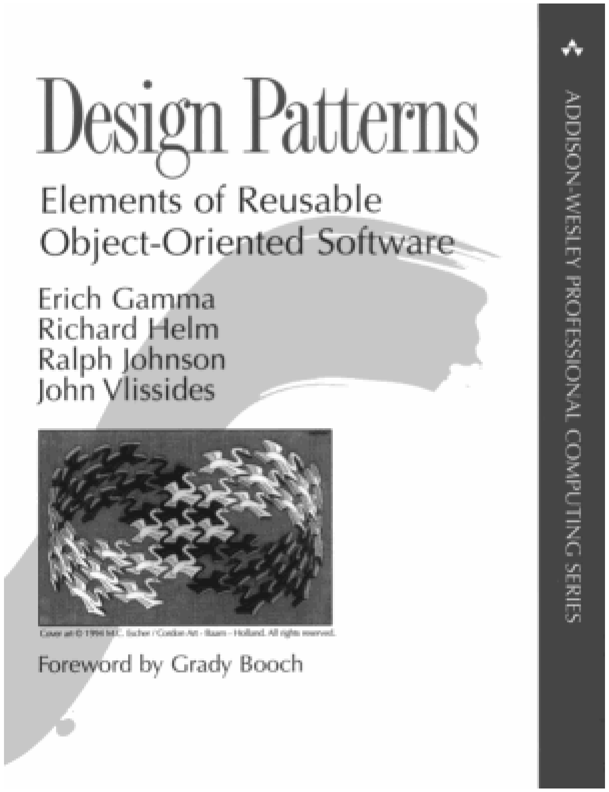
- Iterator Idea:
  - An object which allows iteration over items..

    without exposing implementation details
  - If details are hidden.. they can be changed without cost
  - Can have multiple iterators for a collection without them interfering.

```
int count = 0;
Iterator<String> itr1 = cars.iterator();
while (itr1.hasNext()) {
  String car1 = itr1.next();
  Iterator<String> itr2 = cars.iterator();
  while (itr2.hasNext()) {
    String car2 = itr2.next();
    if (car1.equals(car2)) {
      count++;
    }
  }
}
```

Pattern

- Software Design Pattern:
  - a description of a common software design problem
    and the essence of its solution
  - Allows discussion, implementation, and reuse of proven software designs.
- Gang of Four
  - A pioneering book on design patterns by 4 authors:
    Gamma, Helm, Johnson, Vlissides.


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

The Iterator Pattern

- Context
  - An aggregate object contains element objects
  - Clients need access to the element objects
  - The aggregate object should not expose its internal structure
  - Multiple clients may want independent access
- Solution
  - Iterator fetches one element at a time
  - Each iterator object.. tracks position of the next element
  - Iterators use a common interface.

Iterator UML

- Client only depends on..

  Iterator interface

  - It gets a concrete iterator, but knows only its ~~generic~~ type.

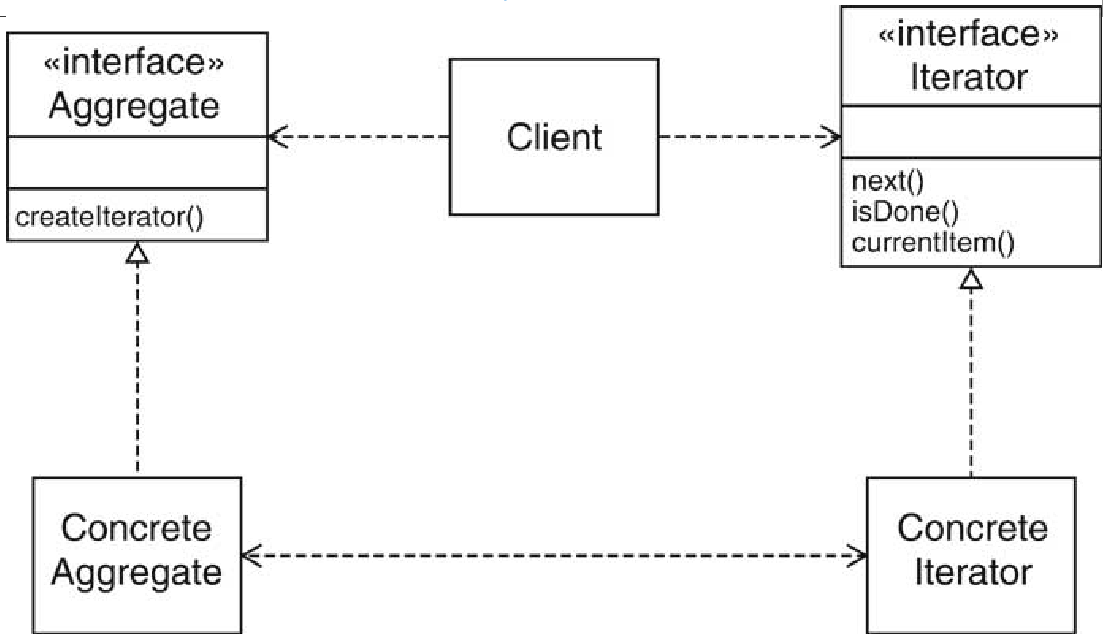
    base

- Mapping pattern to CarManager example:

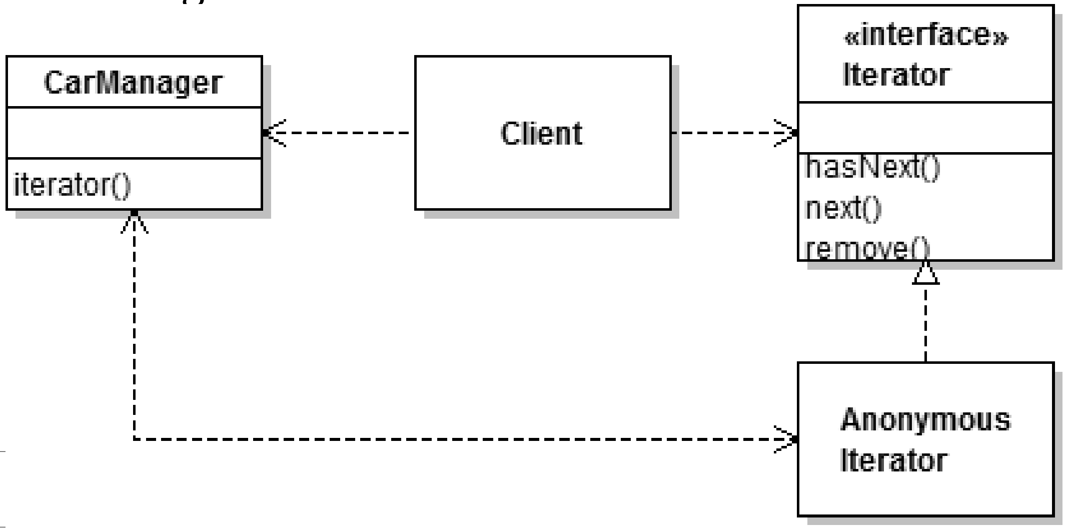
Design Pattern	**CarManager Ex.**
Concrete Iterator	Anon. Iterator
Concrete Aggregate	CarManager
Aggregate <<I>>	*nothing in this example.*
isDone()...	!hasNext()...

Iterator Pattern

client only knows 2 interfaces



CarManager Classes



Observer

Observer pattern motivation

For billionaires!

- Imagine you are writing an automatic day-planner:
  - It reads in the user's interests, plus information about the world, and suggest what they should do.
- Possible design idea:
  - You want to use different objects for cultural planning, sports planning, and sight-seeing.
  - Some objects bring in information about the world; your planning-objects use these info objects.
- Challenge:
  - All of these objects need to know the weather.
  - Your weather object gets updates now and then.
  - How do you tell...all the objects new data is available?

Possible Idea

principle OCP can be applied here -> this code is bad because it needs modification

- Have the weather object call each info. object:

```
class Weather
  void newDataUpdate() {
     String weatherData = ...;
     culturePlanner.update(weatherData);
     sportsPlanner.update(weatherData);
     sightseeingPlanner.update(weatherData);
     // Change here EVERY time you get a new planner.
  }
```

- Bad because:
  - Weather object is... tightly coupled to every planner!
  - Every new planner you get, you'll have to change the weather object's code, recompile, and re-run.

The observer pattern

difference with above solution is: it just notify all objects in its "list" but doesn't need to know which objects are they

- Observer Pattern:

  it allows objects to “register for updates” with another object at run-time

- Produces a one to many relationship:
  - one object observed (called the subject)
  - many objects observing (called the observers).
- Great because it loosely couples objects:
  - Object with something to report does not need a hard-coded list of who to tell; ...

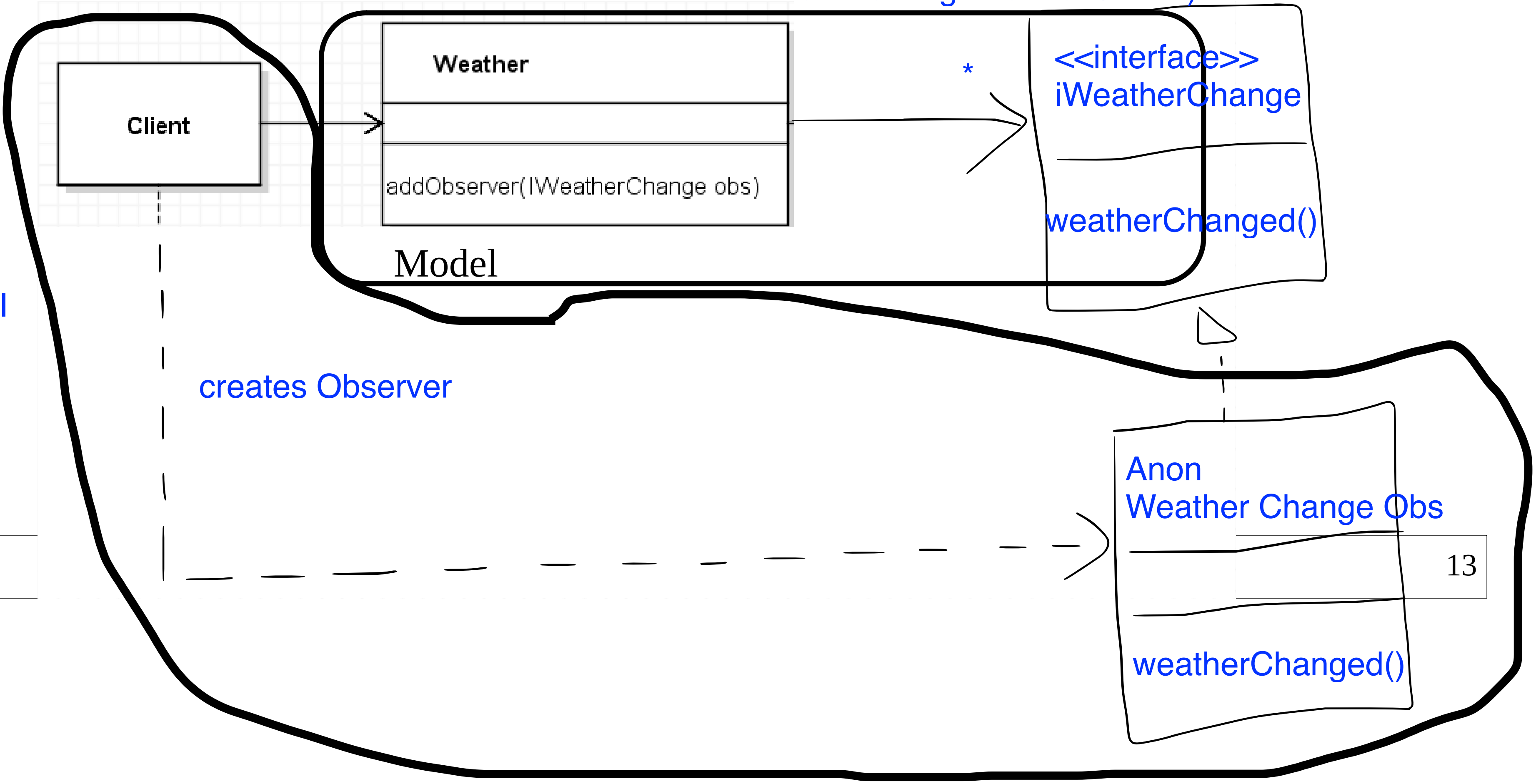
    it simply looks up its observer list

Weather Observer

we want to announce clients if anything change,
but we don’t want to depend on clients
—> use observer

- Weather has forecast and updates it periodically; Client needs to know when new forecast is ready
- Client creates anonymous IWeatherChange obj
  - Client registers it with Weather as a listener for ..call-back on forecast change
- Benefit is. decoupling: model knows nothing of UI

(things being observed by objects know nothing about observer)



Observer Pattern

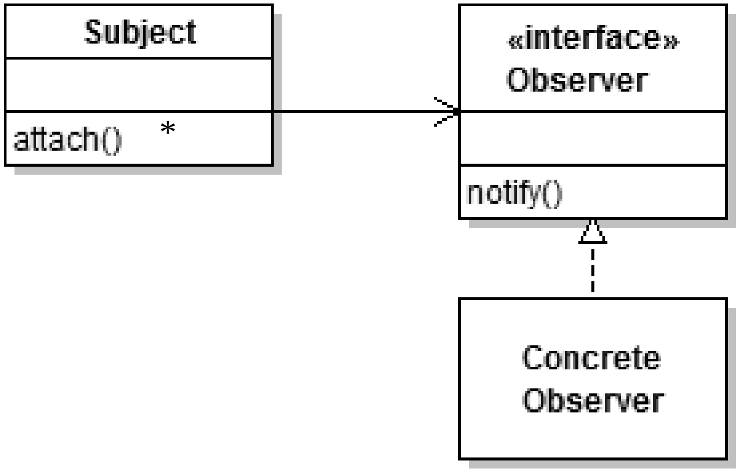
- Context
  - An object, called the subject, is source of events
  - One or more observer objects want to be notified when such an event occurs.
- Solution
  - Define an observer interface type. All..concrete observers implement it
  - Subject maintains a collection of observers.
  - Subject supplies methods for attaching and detaching observers.
  - Whenever an event occurs, the subject..notifies all observer

!! simple NumberList example !!

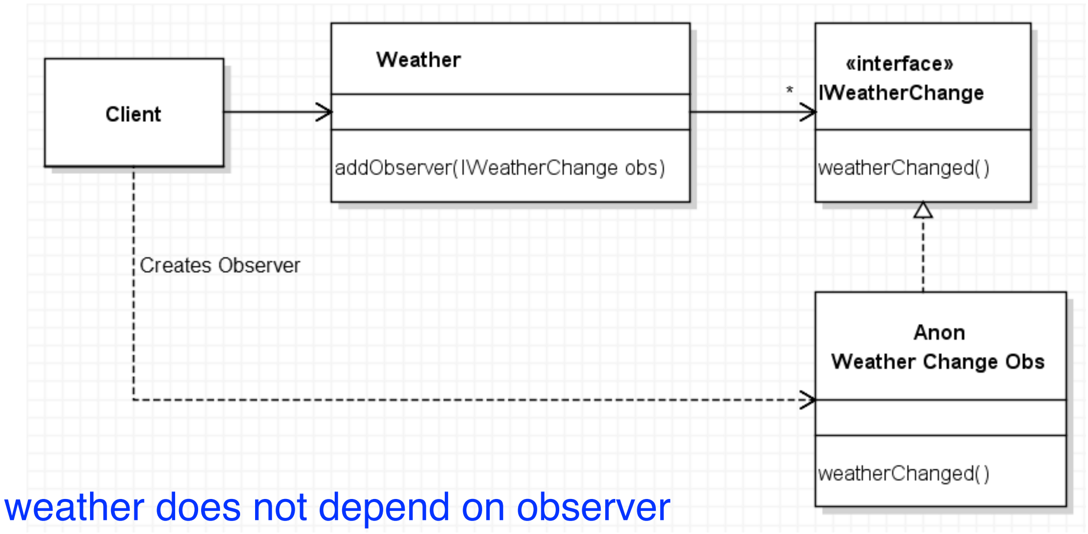
Observer UML

- Subject object knows nothing about class observing it.
  - ..decoupled

Observer Pattern



Design Pattern	**Weather Ex.**
Subject	Weather
attach()	addObserver()
Observer <<I>>	IWeatherWatcher<<I>>
notify()	weatherChanged()
Concrete Observer	Anon. Weather Change Obs



weather does not depend on observer

note: need to know how to convert code not using observer to used one

everything time model changes, observer needs to notify the appropriate ui?

Model View Controller Pattern
and
Facade Pattern

Briefly!

Terminology

- Model:

  stores data and application logic

  – Not like a "model airplane": it's the brains of your system.

- View:

  displays information to the user

  – Numerous views (parts of UI) may register as observers to a model.

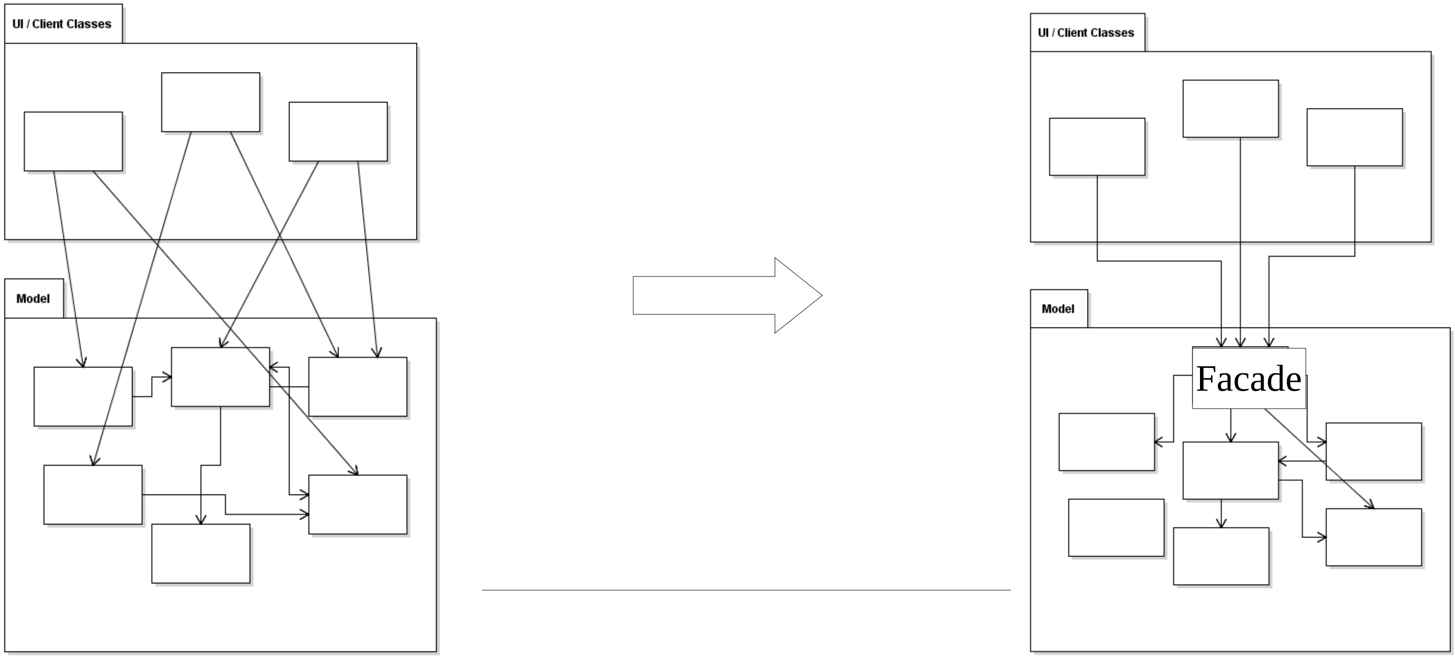
MVC

- Clean design
  Split business logic into.. separate class from UI
- Model View Controller Pattern
  MVC splits off 3 things:
  - Model: .. hold data and logic
    - Ex: HistogramData
  - View: .. present information to user
    - Ex: HistogramIcon, UI components
  - Controller: .. handles user interaction
    - Ex: ActionListeners for buttons.

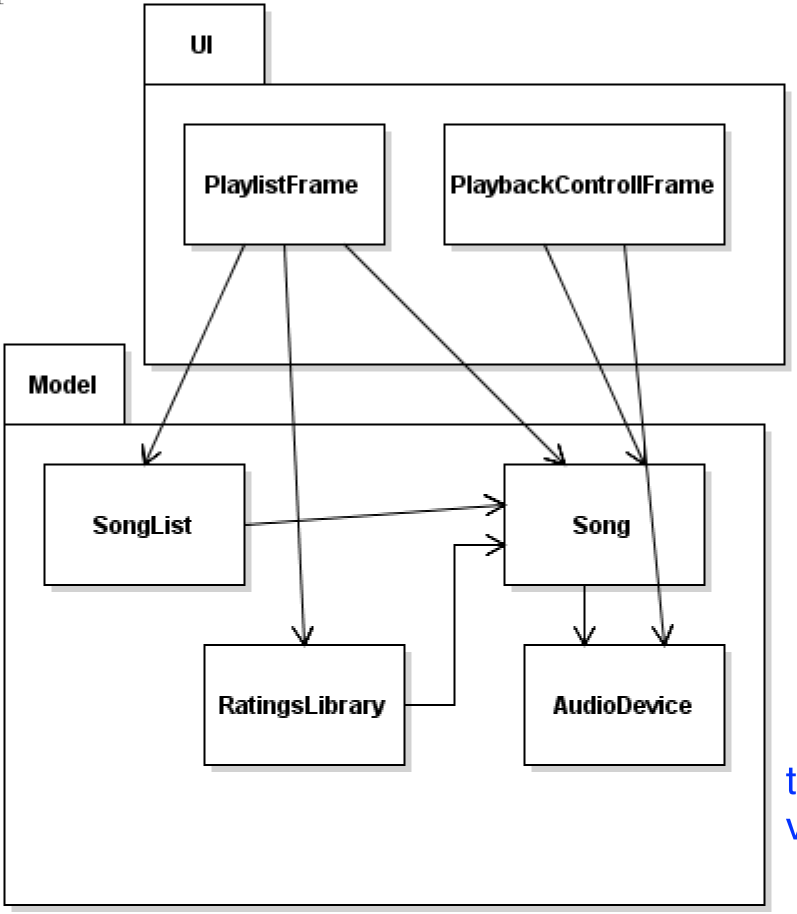
Model is likely to be most unchanged
View usually changes
> thats why we want to decouple Model and View

Facade Pattern

- Separate your model from your UI!
  - What if the model is complicated?
    UI gets..coupled to many classes in the model.
- Facade Pattern
  - Introduce a new class to the model to..
    hide complexities of model structure from client code



Facade Pattern Example: Music Player

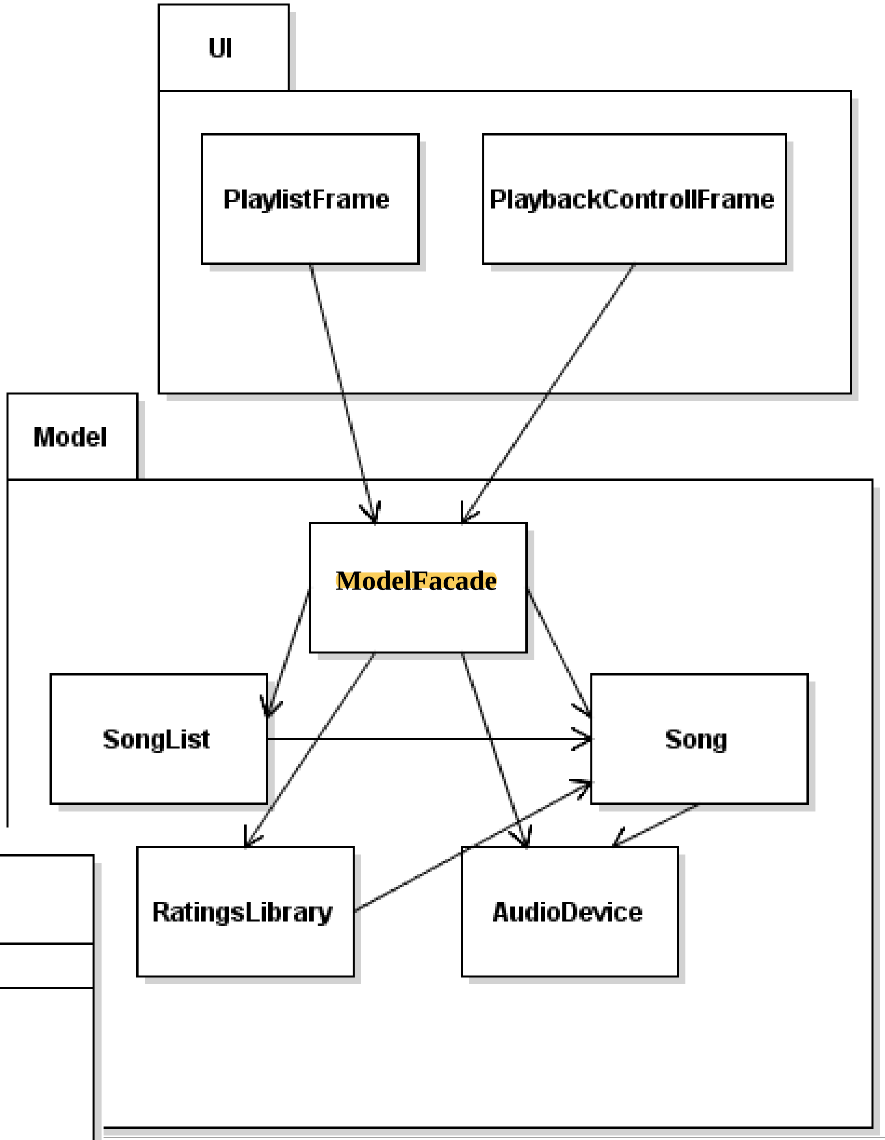


but it makes convenient for users

this pattern violates cohesion?

ModelFacade

openPlaylist(file)
getPlaylistIterator()
playSong(song)
getSongRatingInfo(song)



Recognizing Patterns

Briefly!

Applying Patterns

- Recognize a pattern by.. its intention
  - Iterator: cycle through a collection
  - Observer: register for events
  - Strategy: wrap part of an algorithm into a class
- Helps to remember examples
  - Pattern name a hint, but it's not always applicable.
- Ex: What strategy applies to.. looping through button events?
  - Strategy?
  - Observer? “events” -> use this?
  - Iterator? “looping” -> use this?

Summary

- Design patterns allow reuse of design ideas.
- Iterator: An object which abstracts iteration through items in a collection.
    - Decoupled: change collection without changing client code.
- Observer: Notify observing objects of a change without being coupled to those objects.
- MVC: Separate the model from the view.
    - Consider Facade Pattern to decouple UI from model complexity.
- Apply patterns based on patterns intention
(not name or UML diagram).