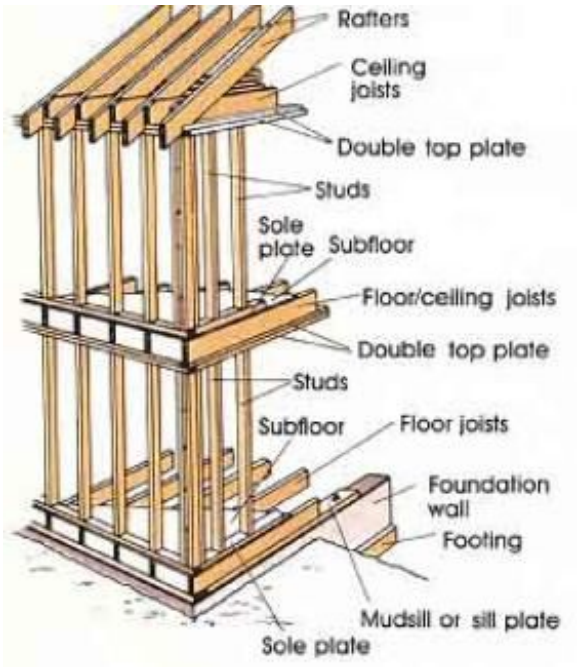
COM S 362
Object-Oriented Analysis & Design

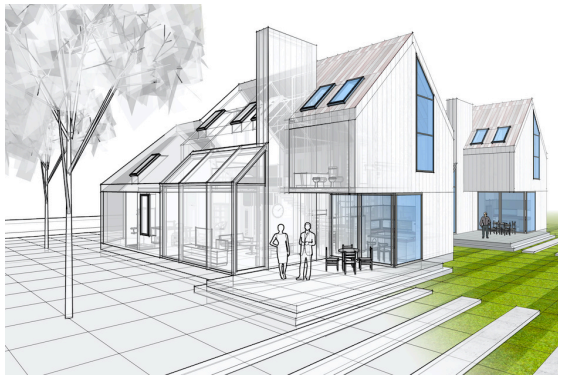
Design Case Study

Clean Code

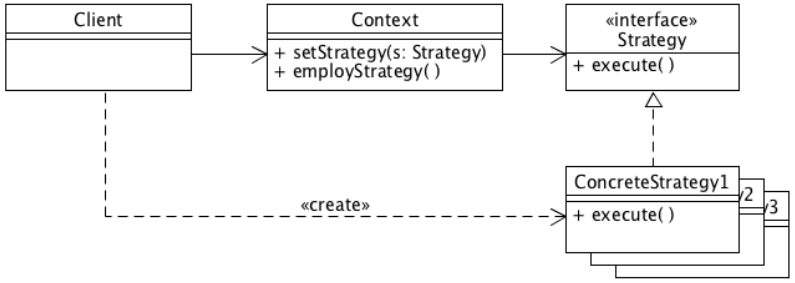
Design Patterns



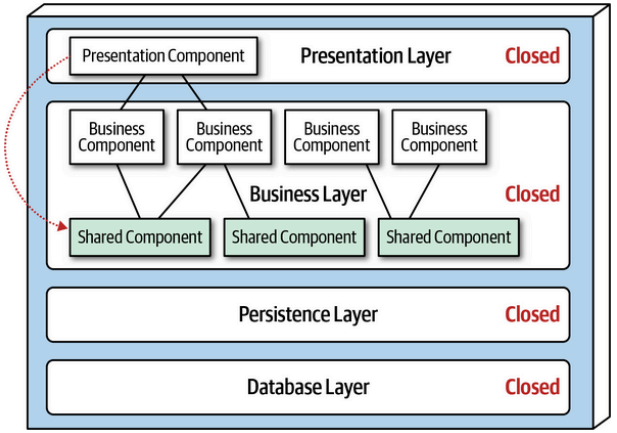
Architecture



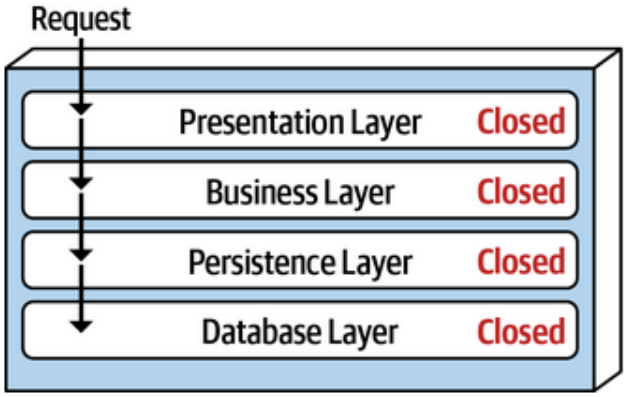
Design Patterns



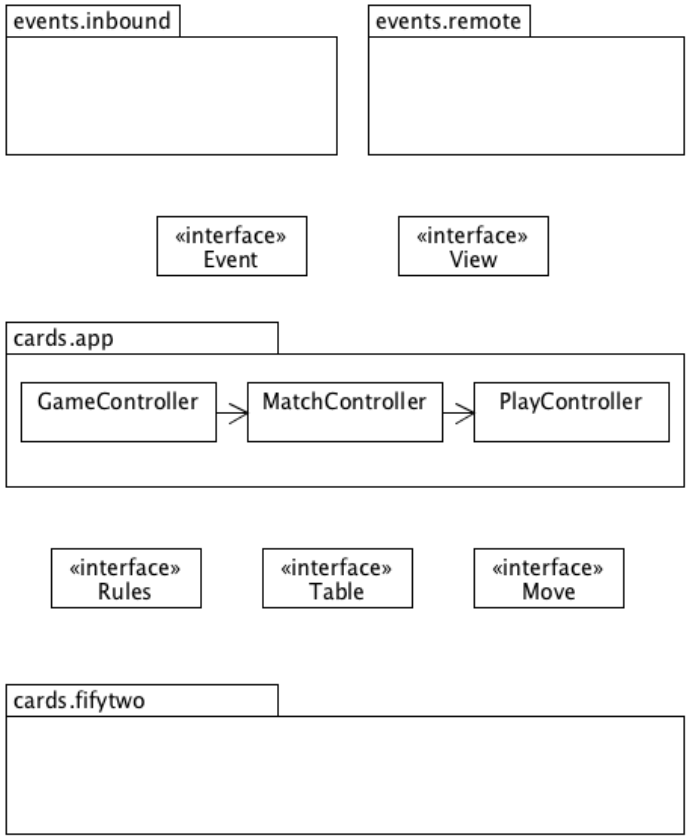
Architecture



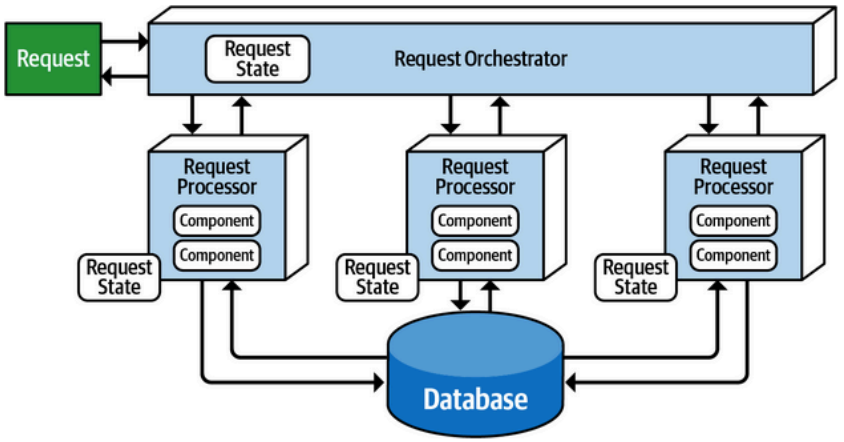
Layered Architecture in Cards Framework

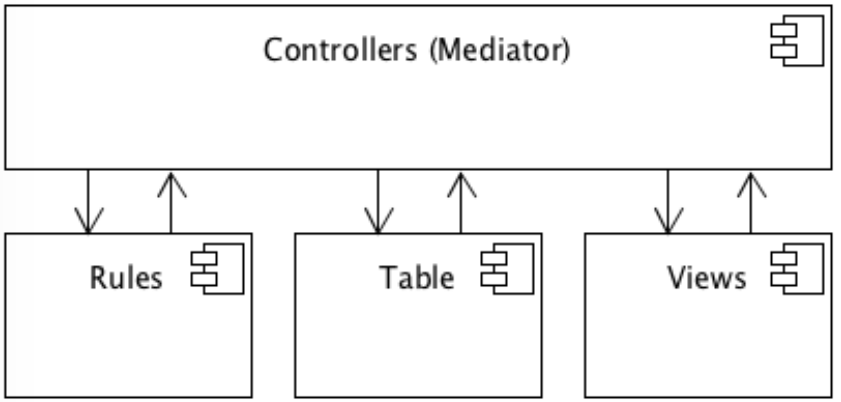


- Simple and widely used architecture
- Common layers: presentation, business, persistence, database
- Follows design principle of Separation of Concerns
- Cards framework follows Model View Controller (MVC) pattern



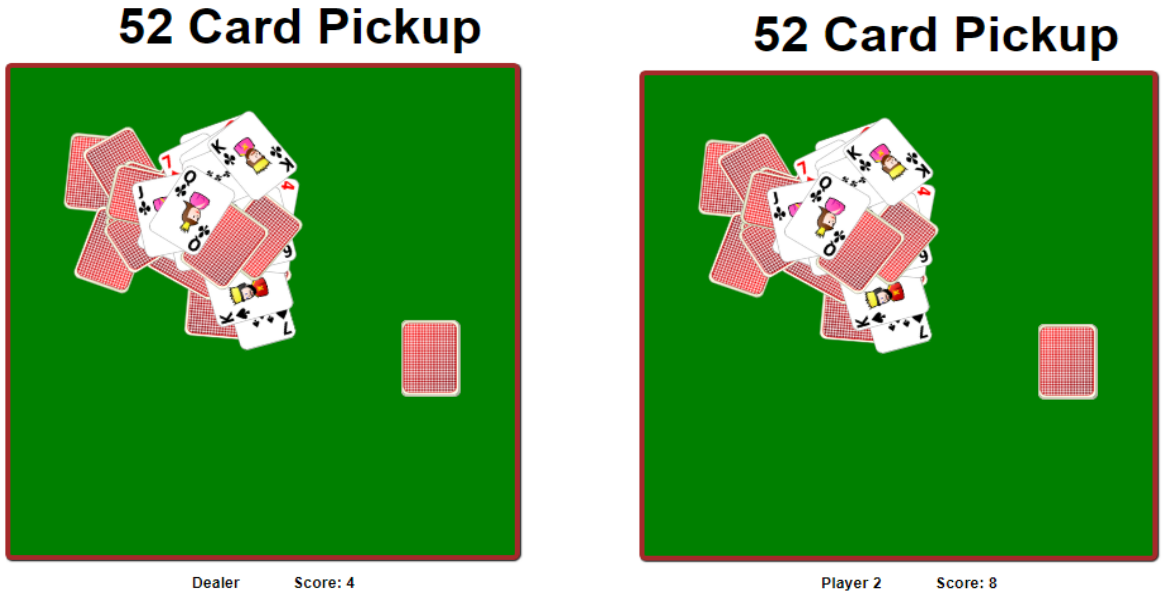
Event-Driven Architecture in Cards Framework





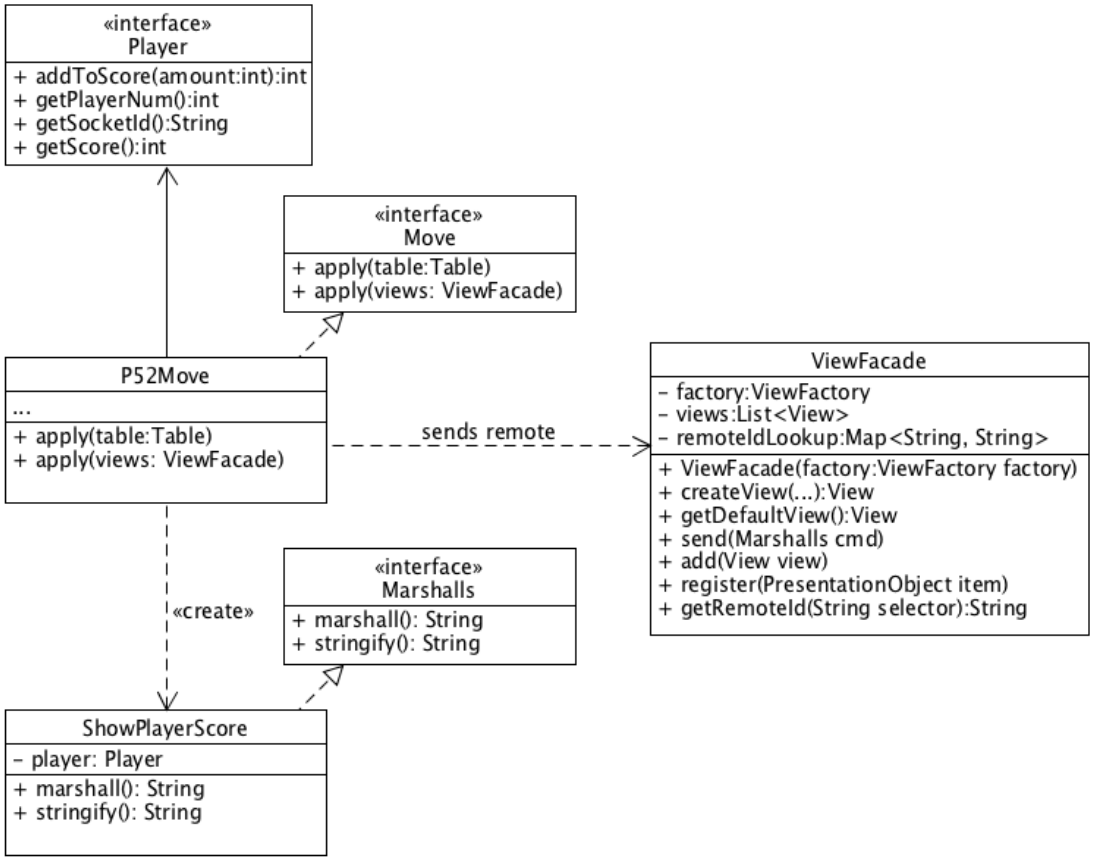
- **Even-Driven Architecture** - Event processing components that asynchronously receive and process events
- Two common models:
  - **Request-based** – User makes request and expects response (e.g., web server)
  - **Event-based** – incoming events generate actions in the system (e.g., security system)
- Common topologies:
  - **Broker** – event processors are piped together by sending and receiving on particular channels
  - **Mediator** – where event are sent is decided by a controller component, the mediator

Player Specific Views



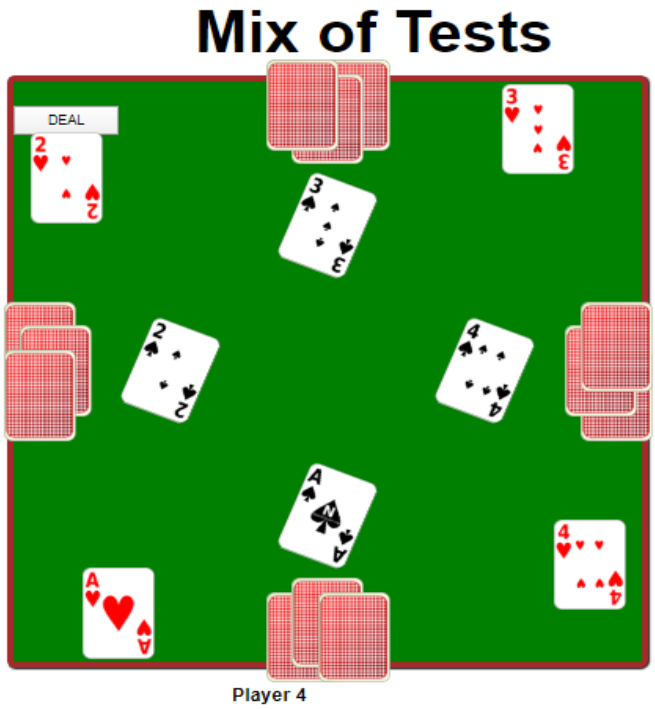
- In the current games all players see the table the same way
- Only player name and score are different

How does each player see a different score?

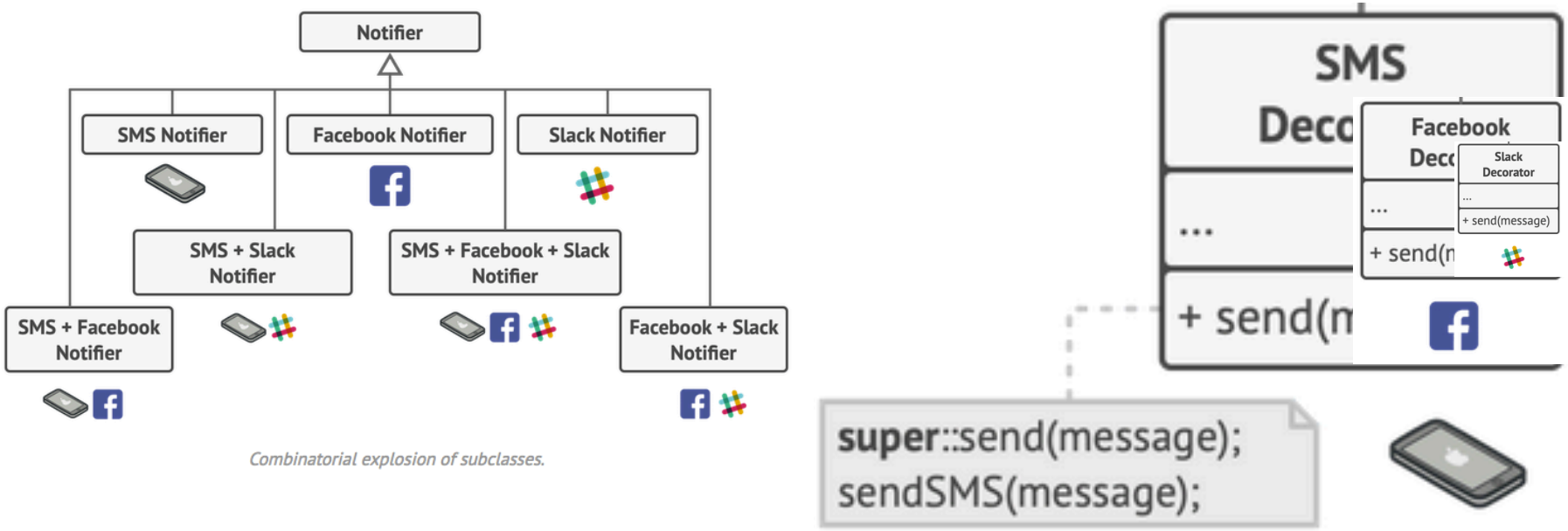


How to show different views of table?

- First approach: In apply(views:ViewFacade) need some way to determine if card should be face up or not
- if (card.owner == player) {
  // does card show face up for owner?
  } else {
  // does card show face down for others?
  }
- How many variations do we need to support
  - Card face up or down
  - Card visible or not
  - Deal button visible or not
  - Card rotation
  - Pile rotation
  - Hand face up or down
  - Hand visible or not
  - Hand rotation
  - …



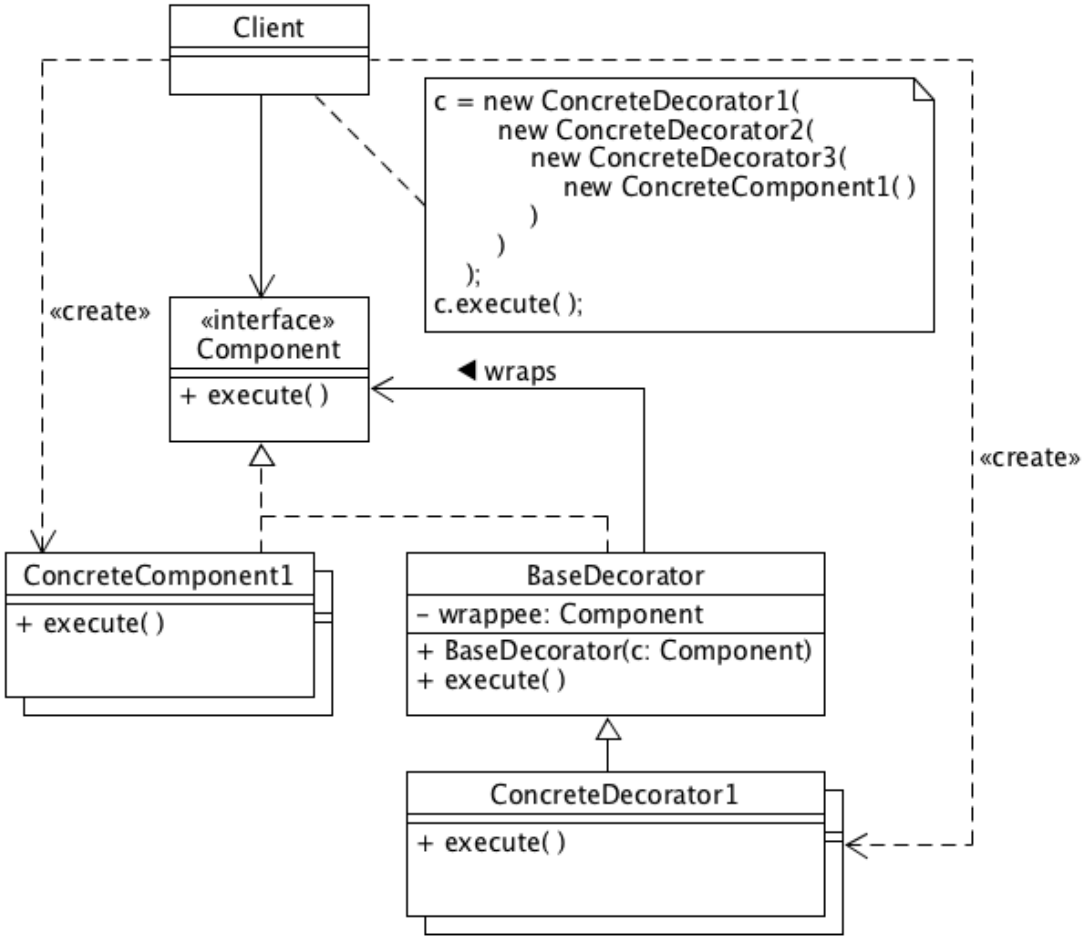
Decorator Pattern: Problem/Solution



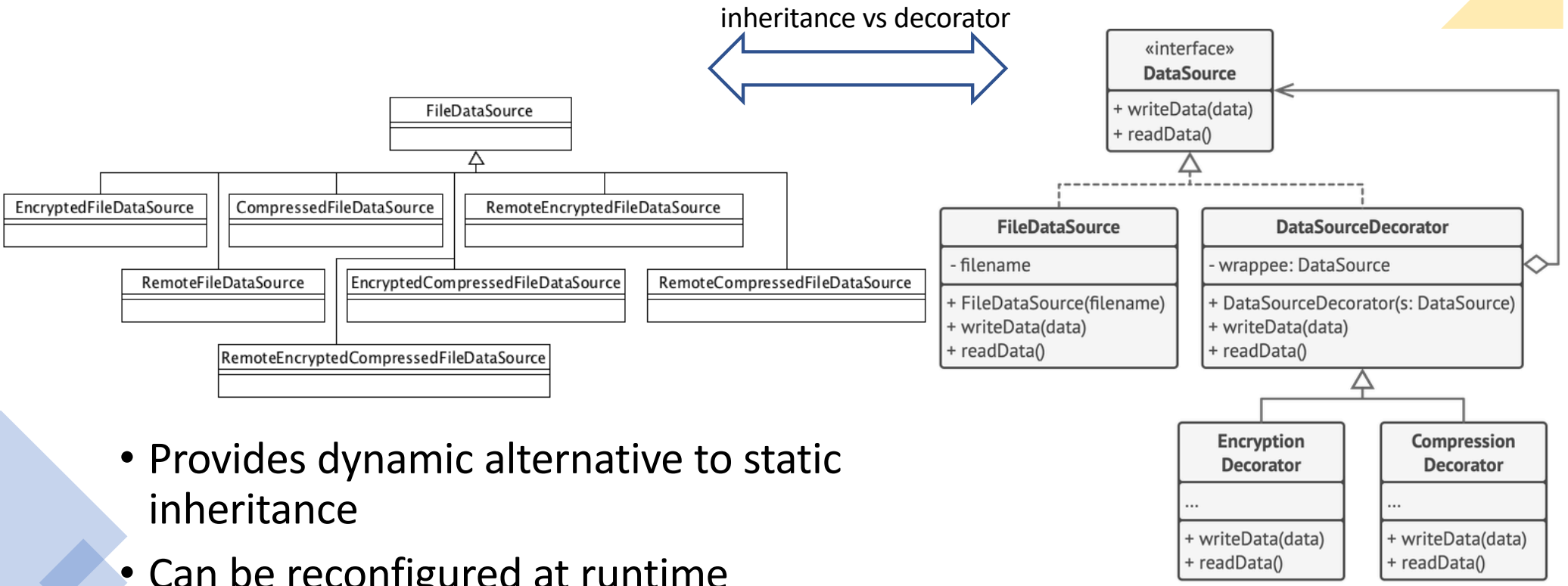
- Problem: You want to create subclasses that take on different combinations of behaviors
- Solution: Put each variation (decorator) into a separate class that wrappers an object of a common superclass, an overridden method call the super class version before performing its own behavior

Decorator Pattern: Structure

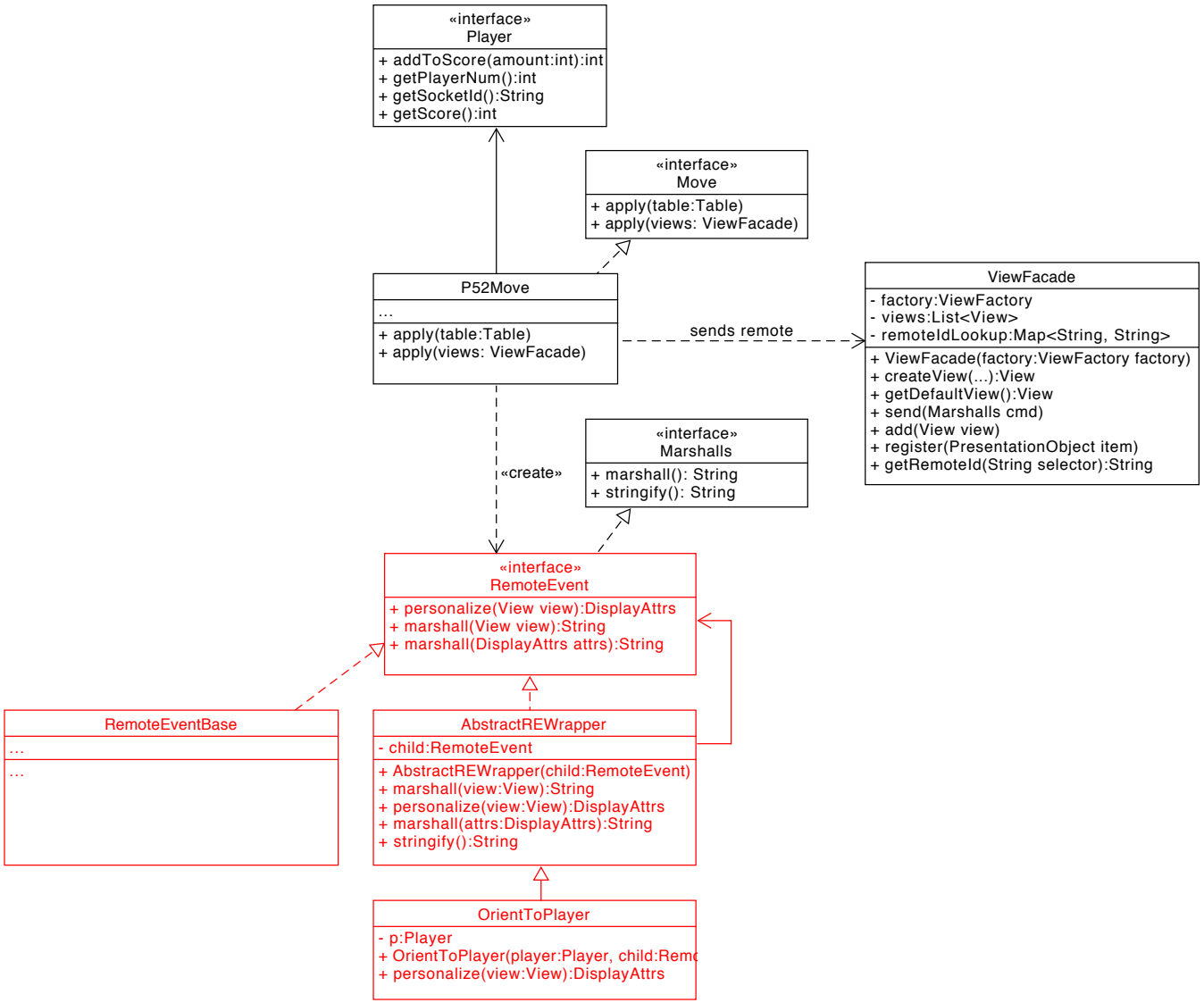
- ConcreteComponent: inner most object being wrapped
- Component: the interface seen by the client
- BaseDecorator: an abstract class that defines the common features of all decorators
- ConcreteDecorator: the decorators that can wrap other decorators and a ConcreteComponent



Alternative to Inheritance



- Provides dynamic alternative to static inheritance
- Can be reconfigured at runtime
- Can take on multiple combinations


«interface»
Player
+ addToScore(amount:int):int
+ getPlayerNum():int
+ getSocketId():String
+ getScore():int
«interface»
Move
+ apply(table:Table)
+ apply(views: ViewFacade)
P52Move
...
+ apply(table:Table)
+ apply(views: ViewFacade)
sends remote
ViewFacade
- factory:ViewFactory
- views:List<View>
- remoteIdLookup:Map<String, String>
+ ViewFacade(factory:ViewFactory factory)
+ createView(...):View
+ getDefaultView():View
+ send(Marshalls cmd)
+ add(View view)
+ register(PresentationObject item)
+ getRemoteId(String selector):String
«create»
«interface»
Marshalls
+ marshall(): String
+ stringify(): String
«interface»
RemoteEvent
+ personalize(View view):DisplayAttrs
+ marshall(View view):String
+ marshall(DisplayAttrs attrs):String
RemoteEventBase
...
...
AbstractREWrapper
- child:RemoteEvent
+ AbstractREWrapper(child:RemoteEvent)
+ marshall(view:View):String
+ personalize(view:View):DisplayAttrs
+ marshall(attrs:DisplayAttrs):String
+ stringify():String
OrientToPlayer
- p:Player
+ OrientToPlayer(player:Player, child:Reme
+ personalize(view:View):DisplayAttrs