M6 - Composition

Jin L.C. Guo

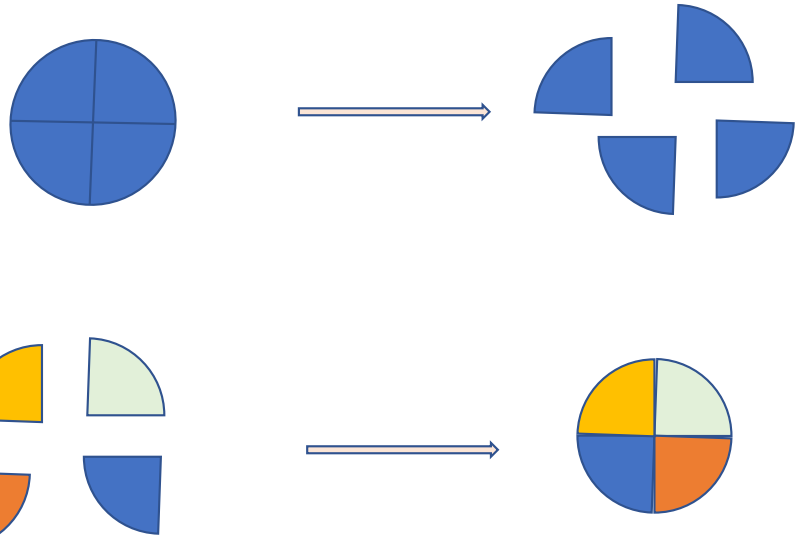
Image source: https://cdn.pixabay.com/photo/2017/11/05/21/21/container-2921882_960_720.jpg

Objective

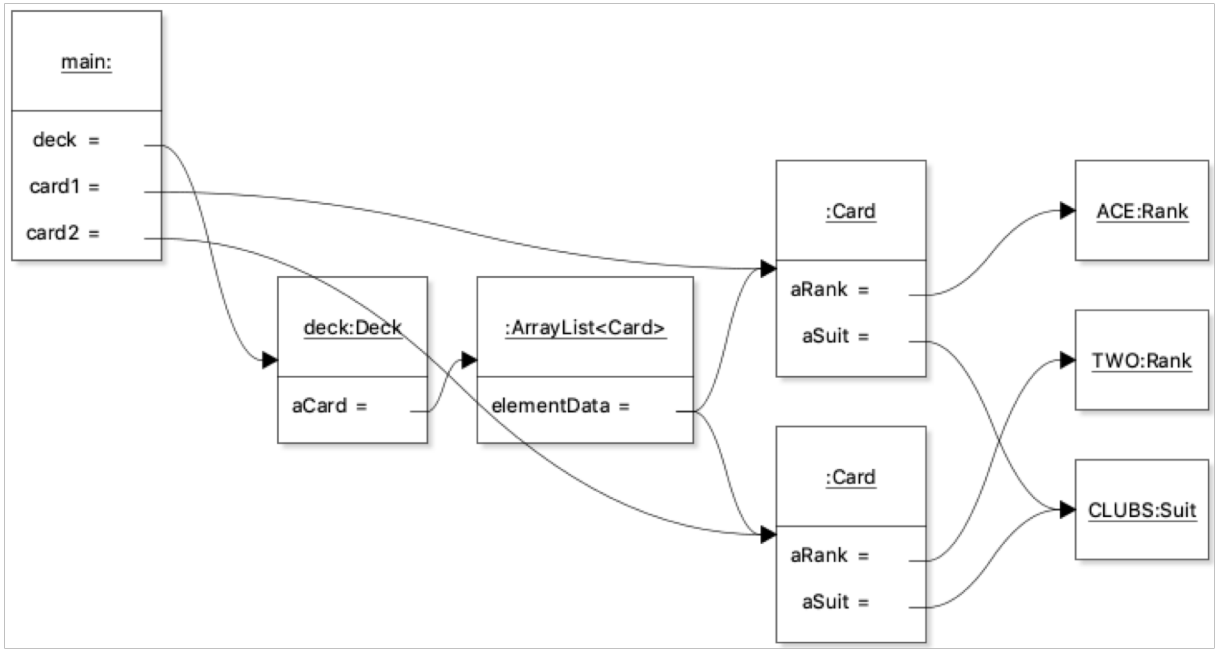
- Divide and Conquer Principle
- Aggregation and Delegation
- Sequence Diagram
- Composite Pattern
- Decorator Pattern
- Polymorphic Object Cloning

Manage Complexity -- Divide and conquer

- Modularization
  - Decomposable
  - Composable



Object Composition

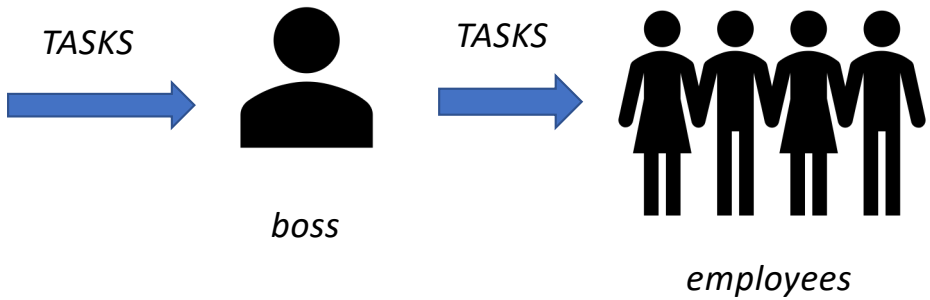


Purpose 1

- Aggregation: Representation of collections

Purpose 2

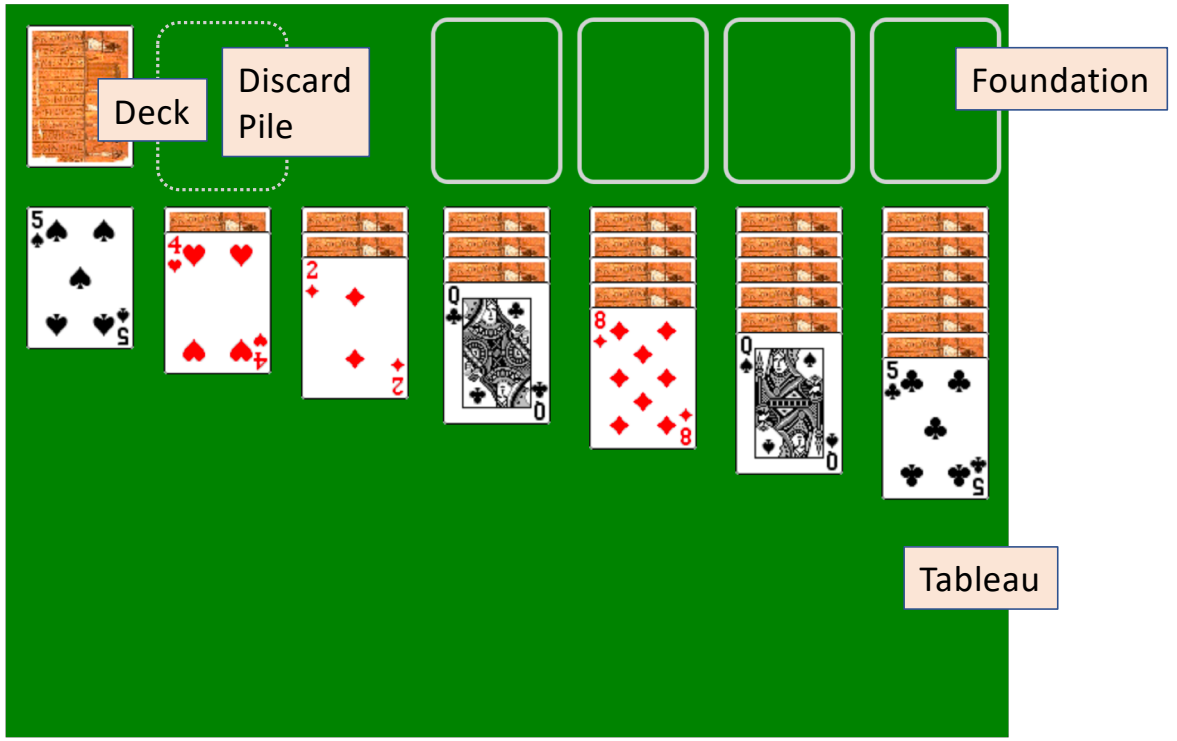
- Delegation: Redirect duties



GameModel in Solitaire

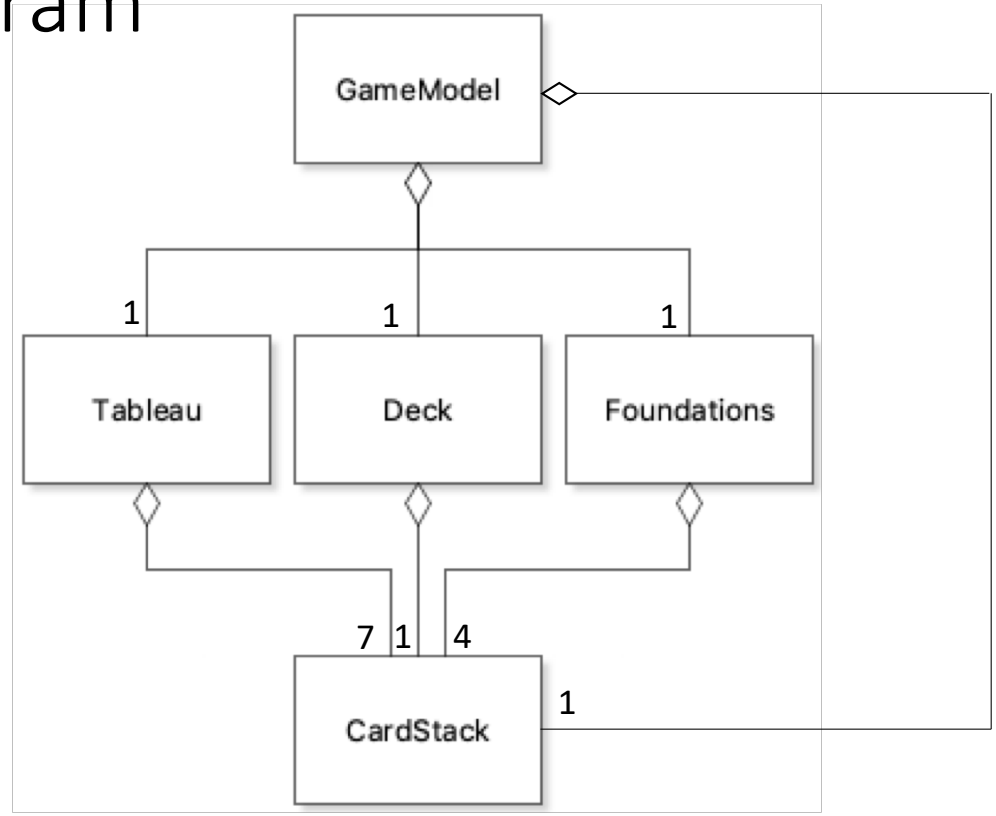
13 piles of cards?

God Class



The elements are both the component, and also entities providing services.

Class Diagram



Delegate Duties

```
public boolean isVisibleInTableau(Card pCard)
{
       return  aTableau.contains(pCard) &&
               aTableau.isVisible(pCard);
}
```

How to support creating a wide variety of card sources?

```
public interface CardSource {
    /*
    * Remove a card from the source and return it.
    *
    * @return The card that was removed from the source
    * @pre size() > 0
    */
    Card draw();
    /*
    * @return The number of cards in the source.
    */
    int size();
}
```

Combining multiple deck

Deck with only face cards

Four aces as a deck

```
public class MultiDeck implements CardSource {/* ... */}

public class FaceCards implements CardSource {/* ... */}

public class FourAces implements CardSource {/* ... */}
```

Large number of possible structures.

Each option requires a class definition, even rare cases

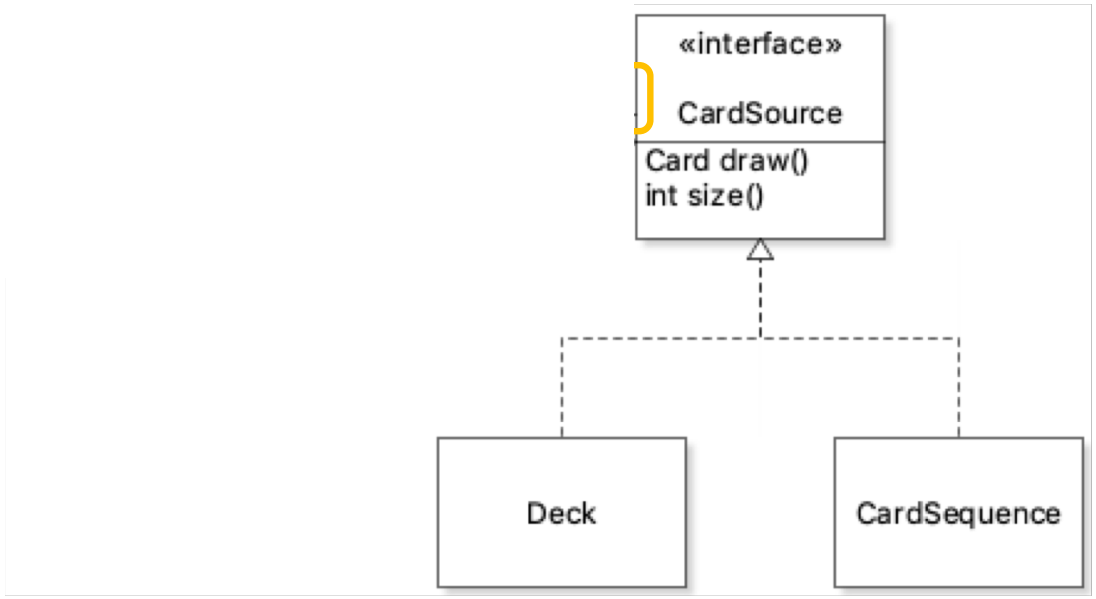
```
public class OneDeckAndFourAces implements CardSource {/* ... */}
```

Difficult to accommodate unexpected situations.

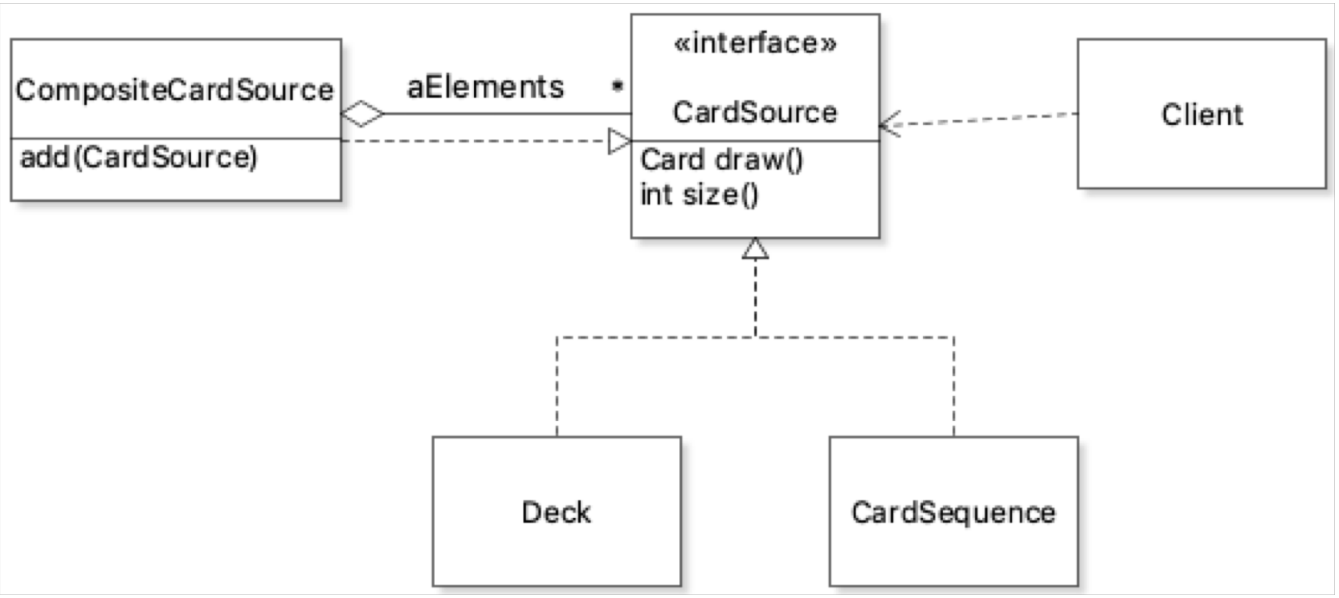
Better Solution

Image Source: https://upload.wikimedia.org/wikipedia/commons/6/61/Lego_blocks.jpg

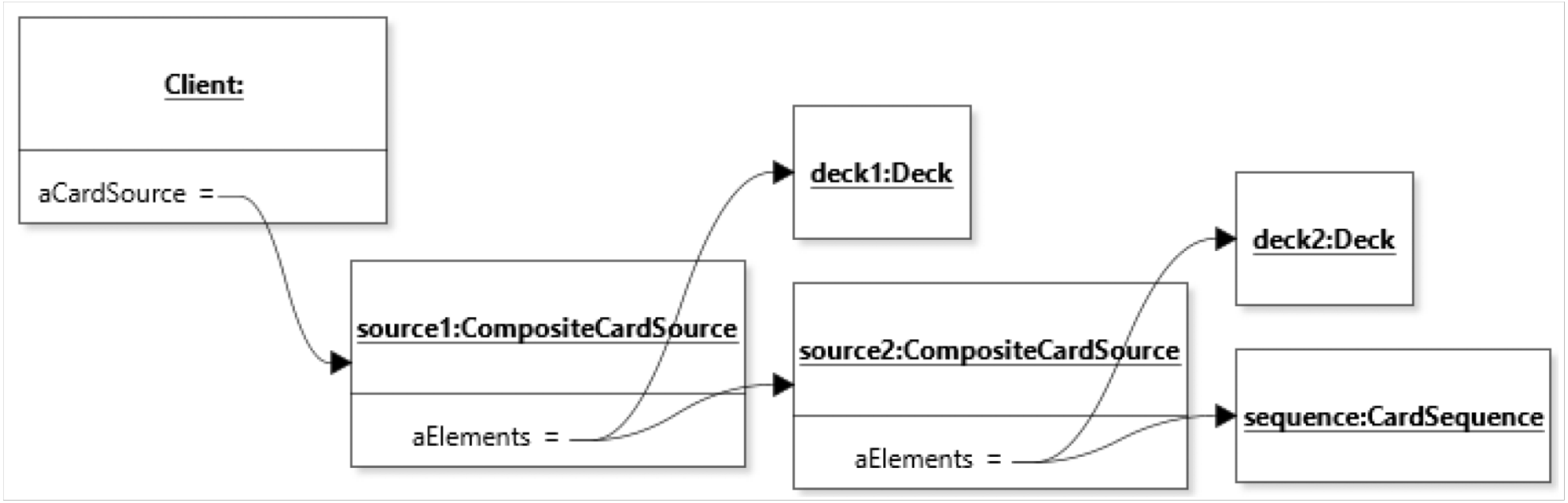
Better Solution



Better Solution



Object Diagram



Composite Pattern

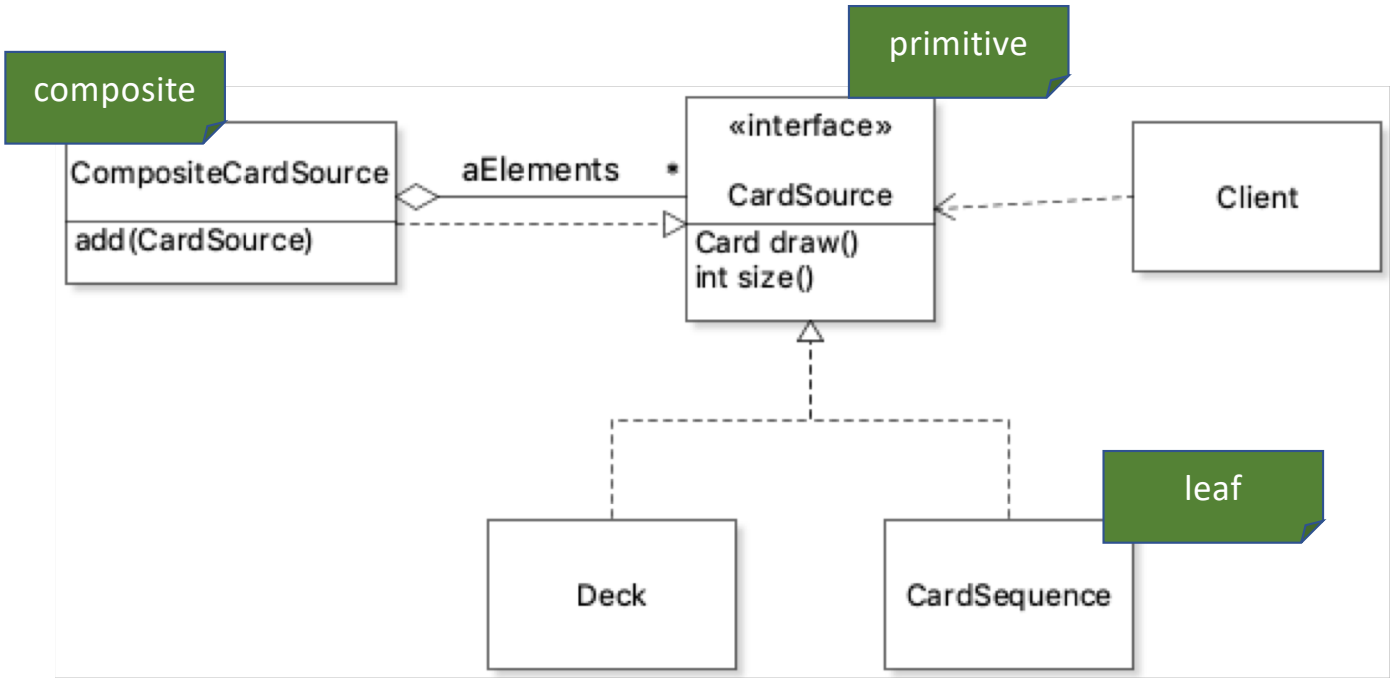
- Intent
  - Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly
- Participants:
  - Primitive (Component)

    Declares the interface for objects in the composition.
  - Leaf

    Defines behaviour for primitives
  - Composite

    Defines behaviour for primitives to have children

Composite Pattern



Implement Composite Pattern

```
public class CompositeCardSource implements CardSource
{
    private List<CardSource> aElements = new ArrayList<>();


    @Override
    public Card draw(){/* … */}


    @Override
    public int size() {/* … */}


}
```

Activity1: how to add Primitive instances to Composite?

```
public void add(CardSource pCardSource)
{
    aElements.add(pCardSource);
}
```

What are the tradeoffs between declaring add method in primitive and in composite?

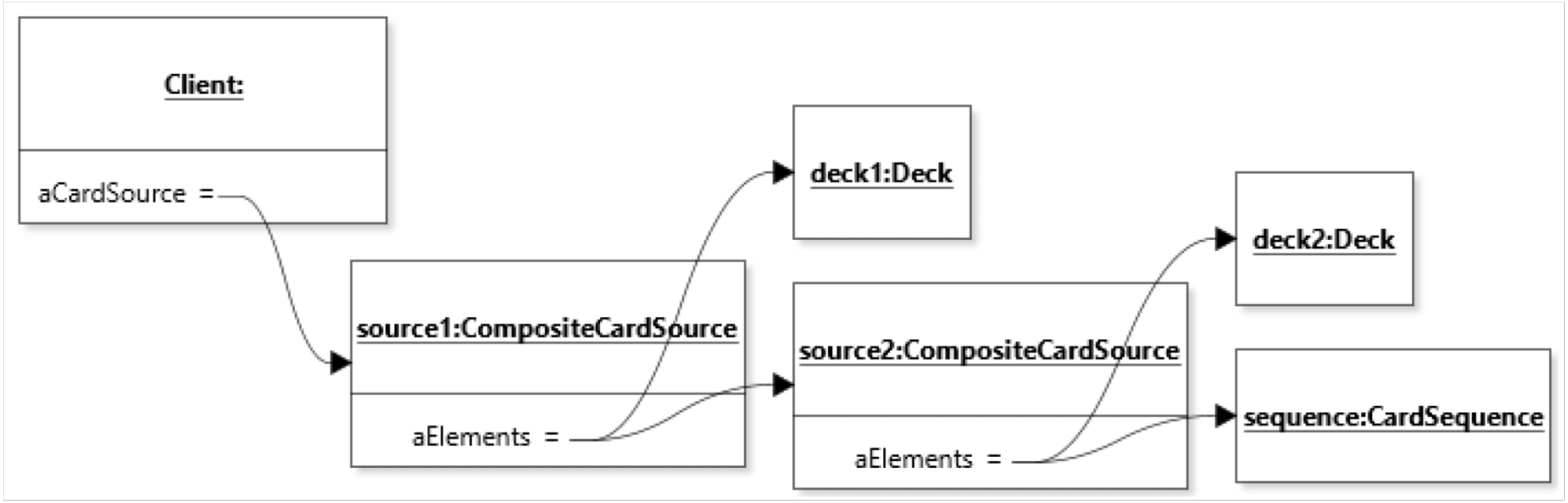
Other considerations

- Order to access the children?
- Enable leaves sharing?

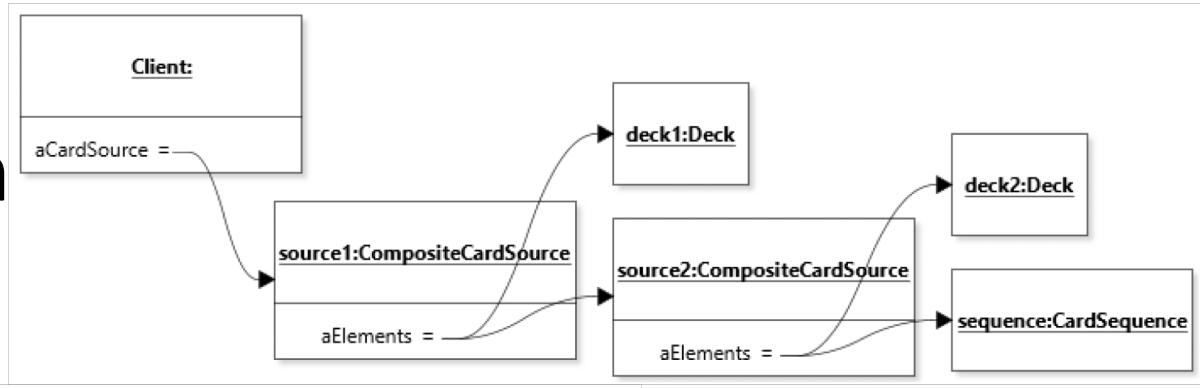
Object Collaboration

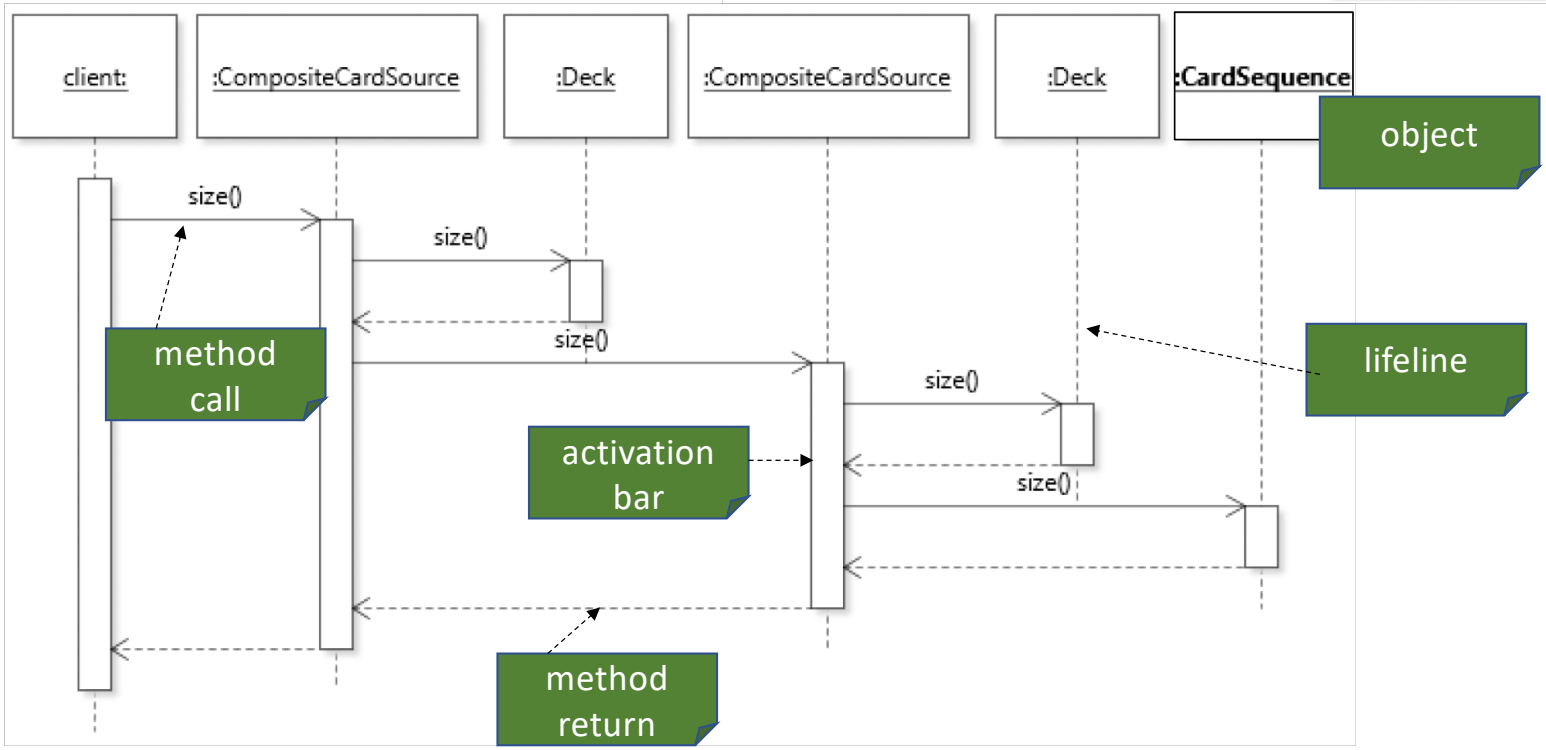
```
public class CompositeCardSource implements CardSource
{
    private List<CardSource> aElements = new ArrayList<>();


    @Override
    public int size()
    {
        int total = 0;
        for(CardSource source : aElements)
        {
            total += source.size();
        }
        return total;
    }
}
```

Modeling object call sequences?



Sequence Diagram





Attach additional responsibility to CardSource?

Memorize the card drawn?

```
public class MemoryDeck implements CardSource
{
   private List<Card> aCards = new ArrayList<>();
   private List<Card> aDrawnCards = new ArrayList<>();

   @Override
   public Card draw()
   {
      assert size()>0;
      Card cardDrawn = aCards.get(aCards.size()-1);
      aCards.remove(aCards.size()-1);
      aDrawnCards.add(cardDrawn);
      return cardDrawn;
   }
   @Override
   public int size()
   {
      return aCards.size();
   }
}
```

Specialized Class, hard to extend

Cannot turn responsibility on and off at runtime

Separate the essential and additional behavior

```
public class MemorizingDecorator implements CardSource
{
   private final CardSource aCardSource;
   private final List<Card> aDrawnCards = new ArrayList<>();


   public MemorizingDecorator(CardSource pCardSource)
   {
      aCardSource = pCardSource;
   }
   ……
}
```

Separate the essential and additional behavior

```
public class MemorizingDecorator implements CardSource
{
   ……
   @Override
   public Card draw()
   {
      Card cardDrawn = aCardSource.draw();
      aDrawnCards.add(cardDrawn);
      return cardDrawn;
   }

   ……
}
```

1. Delegate the original request

2. Implement additional feature

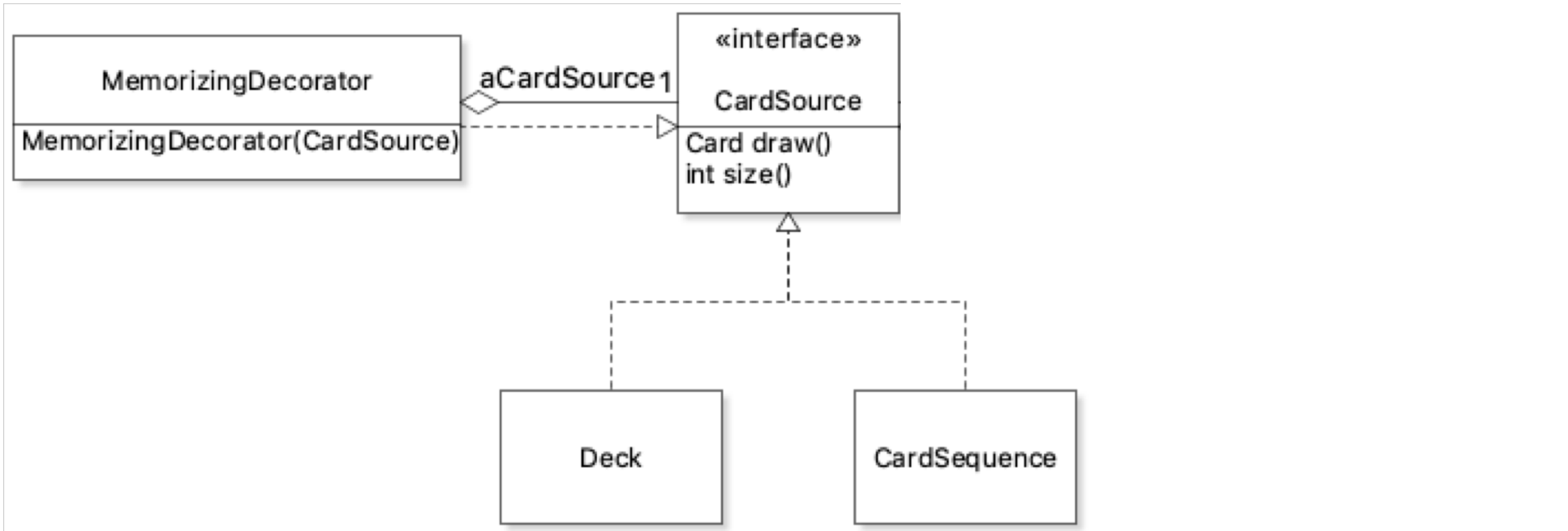
Decorator Pattern

- Intent:
  - Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
- Participants:
  - Primitive

    Declares the interface for objects that can have responsibilities added to them dynamically
  - Leaf

    Defines the class to which additional responsibilities can be attached.
  - Decorator

    Maintains a reference to the primitive and defines the interface that confirms the primitive's interface

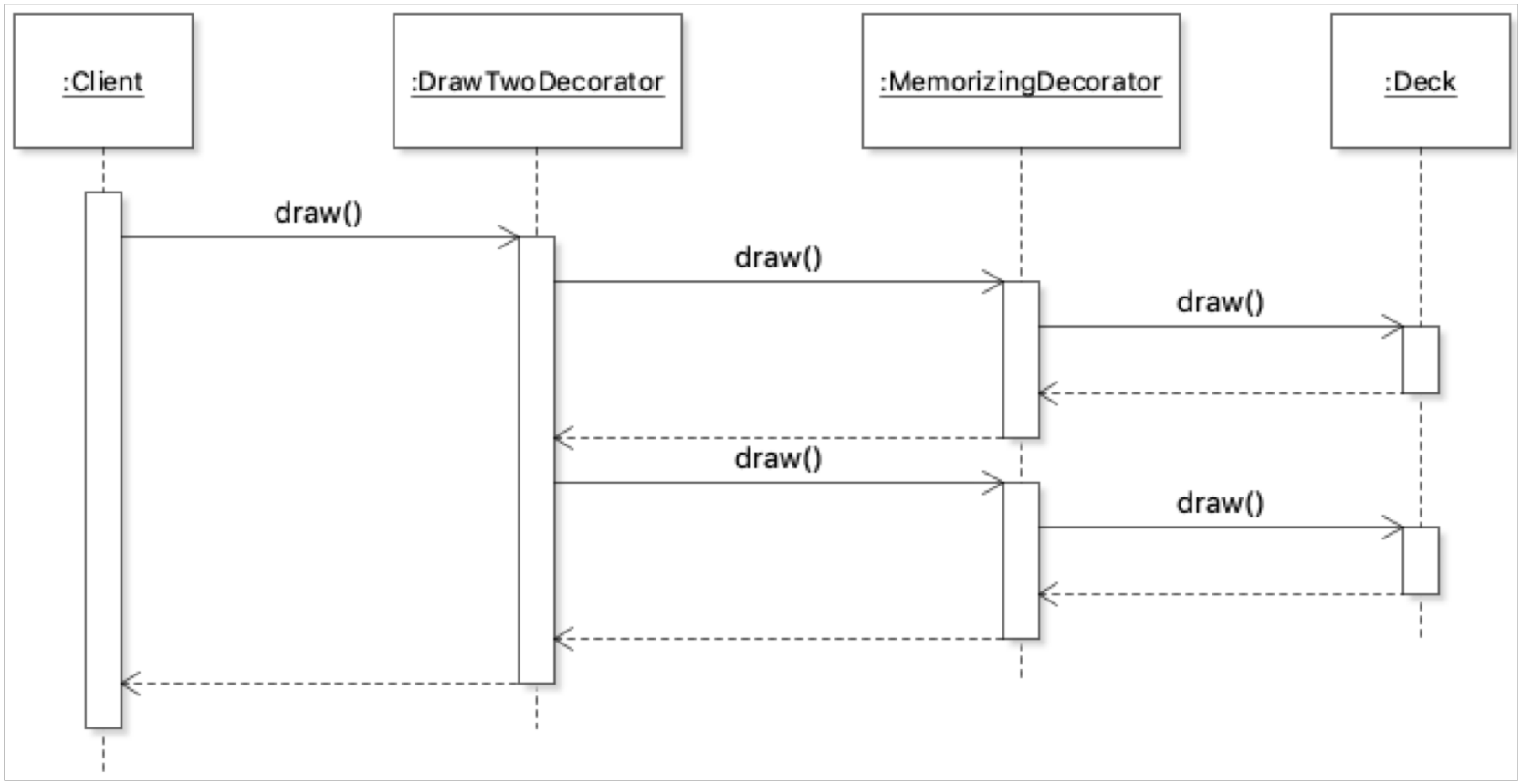

MemorizingDecorator
MemorizingDecorator(CardSource)
aCardSource
1
«interface»
CardSource
Card draw()
int size()
Deck
CardSequence

```
public class DrawTwoDecorator implements CardSource
{
   ……
   @Override
   public Card draw()
   {
      Card card1 = aCardSource.draw();
      if (aCardSource.size() > 0)
      {
         Card card2 = aCardSource.draw();
         if (card1.compareTo(card2)>0)
            return card1;
         return card2;
      } else {
         return card1;
      }
   }
}
```

Activity 2

- Draw the sequence diagram when the following client code is executed.

```
CardSource deck = new Deck();
CardSource memorizingDeck = new MemorizingDecorator(deck);
CardSource drawTwoMemorizingDeck =
   new DrawTwoDecorator(memorizingDeck);
Card card = drawTwoMemorizingDeck.draw();
```

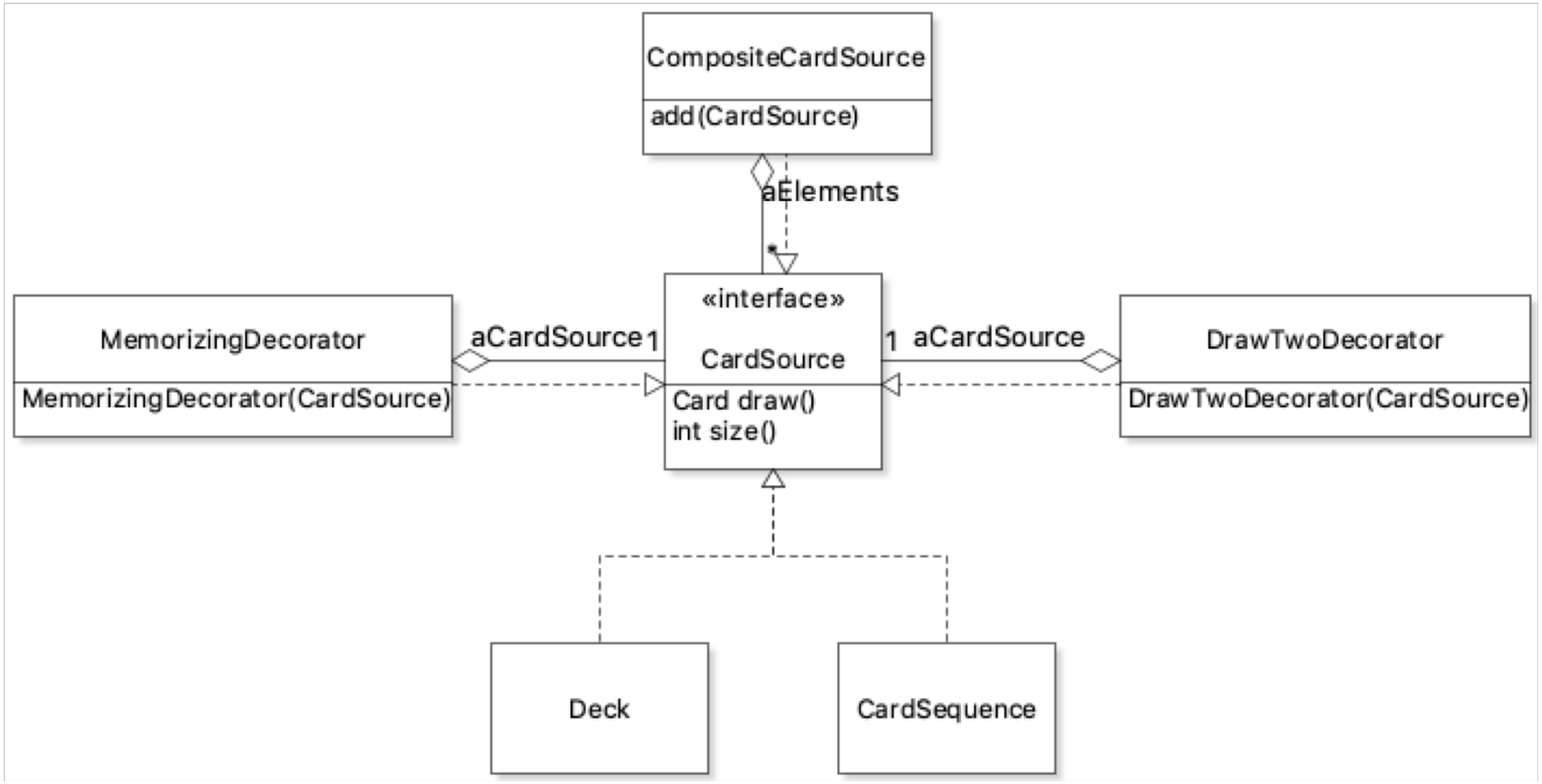

:Client
:DrawTwoDecorator
:MemorizingDecorator
:Deck
draw()
draw()
draw()
draw()
draw()

Identity of decorator and decorated object

```
CardSource deck = new Deck();
CardSource memorizingDeck = new MemorizingDecorator(deck);
CardSource drawTwoMemorizingDeck =
   new DrawTwoDecorator(memorizingDeck);
Card card = drawTwoMemorizingDeck.draw();

System.out.println(deck == memorizingDeck); // true or false?
```

Decorated object identity lost

Combining Decorator and Composite



```
Deck deck = new Deck();

List<Card> fourAces = new ArrayList<Card>();
fourAces.add(new Card(Rank.ACE, Suit.CLUBS));
fourAces.add(new Card(Rank.ACE, Suit.DIAMONDS));
fourAces.add(new Card(Rank.ACE, Suit.HEARTS));
fourAces.add(new Card(Rank.ACE, Suit.SPADES));
CardSource fourAcesDeck = new CardSequence(fourAces);

CompositeCardSource compositeDeck = new CompositeCardSource();
compositeDeck.add(fourAcesDeck);
compositeDeck.add(deck);

CardSource memorizingDeck = new MemorizingDecorator(compositeDeck);
```

Polymorphic Object Copying

```
public class CardSourceManager
{
    List<CardSource> aCardSources = new ArrayList<>();


    public List<CardSource> getCardSources() {
        // return a copy of aCardSources;
    }
}
```

Use copy constructor?

```
new Deck(source);
```

Static factory method?

Implements Cloneable

- java.lang.Cloneable

this interface does *not* contain the clone method.

implement this interface should override Object.clone with a public method.

A class implements the Cloneable interface to indicate to the Object.clone() method that it is legal for that method to make a field-for-field copy of instances of that class.

Invoking Object's clone method on an instance that does not implement the Cloneable interface results in the exception CloneNotSupportedException being thrown.

Override Object.clone()

protected Object clone()
throws CloneNotSupportedException

Creates and returns a copy of this object.

x.clone() != x

x.clone().getClass() == x.getClass()

x.clone().equals(x)

object should be obtained by calling super.clone

the object returned by this method should be independent of this object

Polymorphic Object Copying Demo

```java
public class CompositeCardSource implements CardSource
{
    private List<CardSource> aElements = new ArrayList<>();

    @Override
    public CardSource clone()
    {
        try
        {
            CompositeCardSource clone = (CompositeCardSource) super.clone();
            clone.aElements = new ArrayList<CardSource>();
            for (CardSource cs:aElements)
            {
                clone.aElements.add(cs.clone());
            }
                return clone;
        }
        catch (CloneNotSupportedException e)
        {
            assert false;
            return null;
        }
    }
}
```