Syntax

① enum Rank.values() → array arr.length
ACE.ordinal()

② throw new IllegalArgumentException();

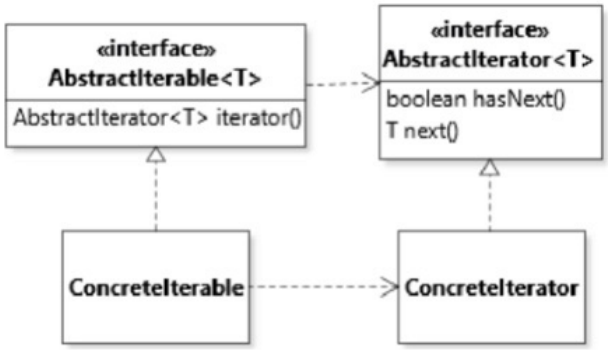
③ Optional
Optional<Rank> aRank

- Optional.empty()
- aRank.isPresent()
- Optional.of(value) / Optional.ofNullable(value)
- aRank.get()

④ HashMap

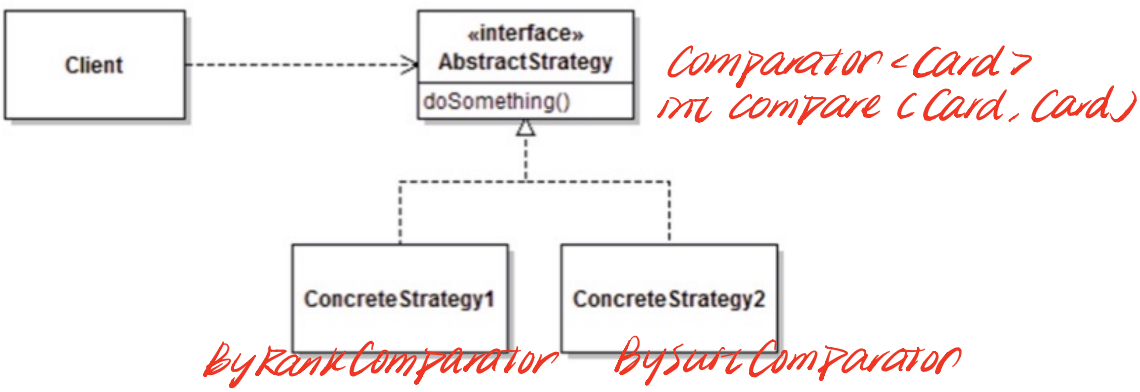
Iterator

Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.



Strategy

Define a family of algorithms, encapsulate each one, and make them interchangeable.



Null Object

```
public interface CardSource {

    public static CardSource NULL = new CardSource() {
        public boolean isEmpty() { return true; }
        public Card draw() { assert !isEmpty; return null; }
        public boolean isNull() { return true; }
    };

    boolean isEmpty();
    Card draw();
    default boolean isNull() { return false; }
}
```

Object Equality, Identity, Uniqueness

↗ Two distinct objects cannot be equal.

[==]

```
public boolean equals(Object o) {

  if (o == null) { return false; }
  if (o == this) { return true; }
  if (o.getClass() != getClass()) { return false; }
  o = (Card) o;
  return   o.aRank == aRank && o.aSuit == aSuit;
}

public int hashCode() {
    return this.toString().hashCode();
}
```

Flyweight (immutable) uniqueness

① private constructor

② a static flyweight store

③ a static access method

Method 1. Pre-initialized

```
public class Card
{
  private static final Card[][] CARDS =
    new Card[Suit.values().length][Rank.values().length];

  static
  {
    for( Suit suit : Suit.values() )
    {
      for( Rank rank : Rank.values() )
      {
        CARDS[suit.ordinal()][rank.ordinal()] =
          new Card(rank, suit);
      }
    }
  }
}
```

static initializer block

↑ private

```
public static Card get(Rank pRank, Suit pSuit)
{
  assert pRank != null && pSuit != null;
  return CARDS[pSuit.ordinal()][pRank.ordinal()];
}
```

Method 2. Instances are lazily created.

```
public class Card
{
  private static final Card[][] CARDS =
    new Card[Suit.values().length][Rank.values().length];

  public static Card get(Rank pRank, Suit pSuit)
  {
    if( CARDS[pSuit.ordinal()][pRank.ordinal()] == null )
    {
      CARDS[pSuit.ordinal()][pRank.ordinal()] =
        new Card(pRank, pSuit);
    }

    return CARDS[pSuit.ordinal()][pRank.ordinal()];
  }
```

Singleton (mutable) Only one instance

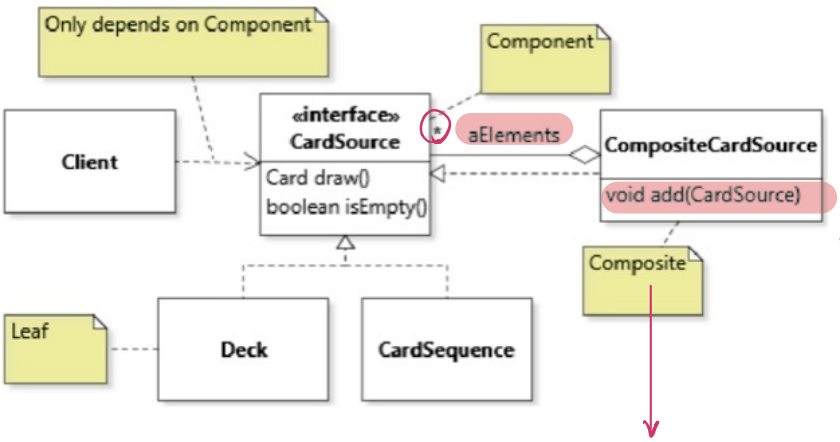
① private constructor

② a global variable holding the reference to the singleton object.

③ an accessor method (instance())

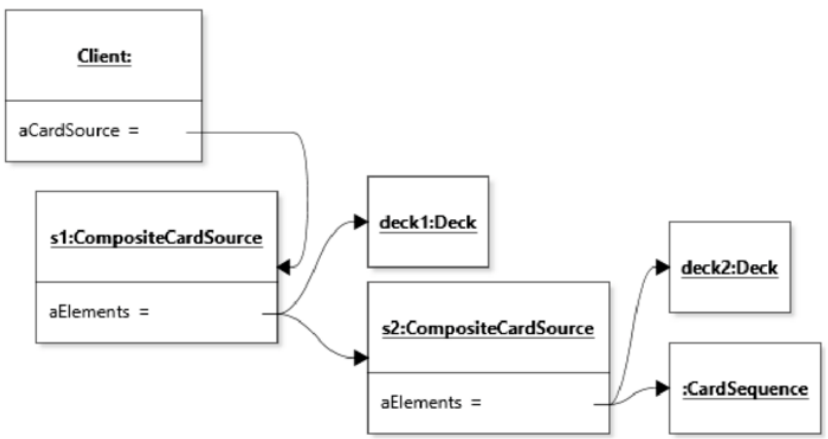
```
public class GameModel
{
  private static final GameModel INSTANCE = new GameModel();

  private GameModel() { ... }

  public static GameModel instance() { return INSTANCE; }
}
```

Composite



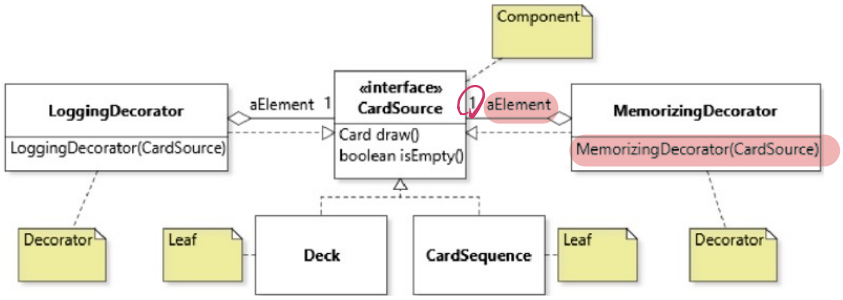
① add method
② constructor

① Aggregate a number of different objects of the component type.
② could be treated in exactly the same way as the leaf objects.



An iteration through all the aggregated elements and delegate the method call to all elements.

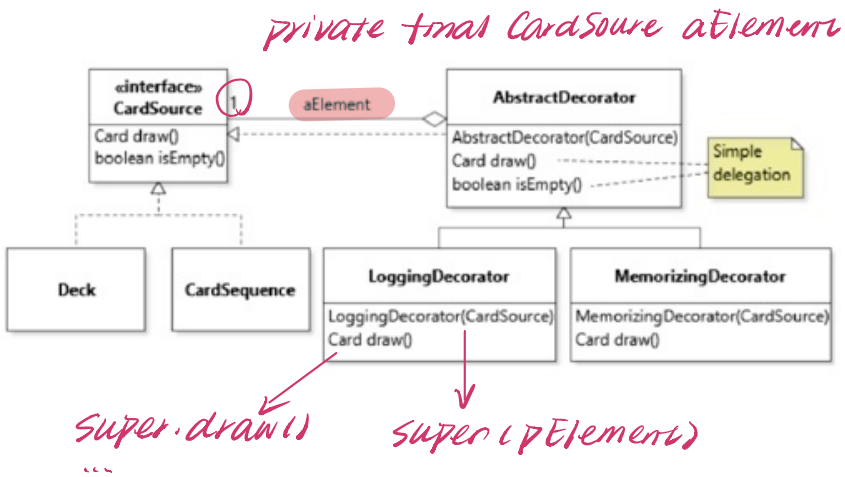
Decorator



① Delegate the original request to the decorated object

② Implement the decoration.

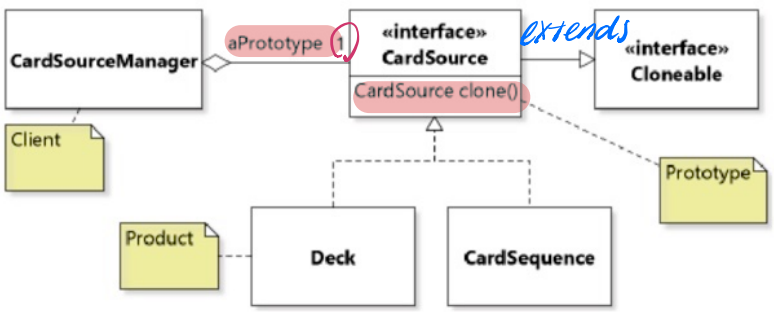
⚠ The decorated object lose its identity.



Prototype

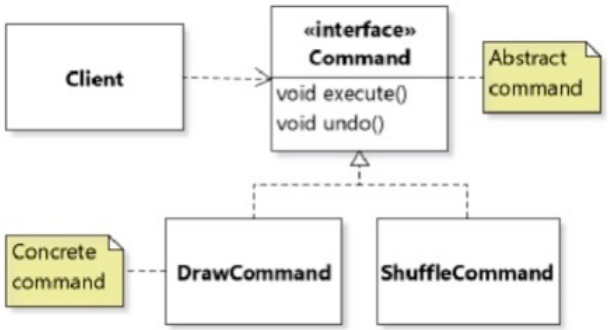
Create objects whose type may not be known at compile time.

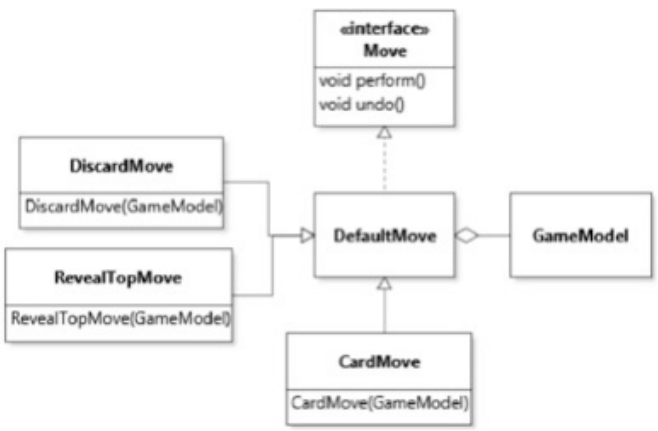
```
public class CardSourceManager {
    ① storing a reference to prototype
    private CardSource aPrototype = new Deck(); // default

    public void setPrototype(CardSource pPrototype){
        aPrototype = pPrototype;
    }

    public CardSource createCardSource(){
        return ② aPrototype.clone();
                  polymorphically copy
    }
}
```



Command

manageable units of functionality





The Law of Demeter

Objects do not return reference to their internal structure, but instead provide the complete service required by client in the delegation chain.

Template inheritance

Put all the common code in the superclass, and to define some "hooks" to allow subclasses to provide specialized functionality.

```
public abstract class AbstractMove implements Move {
    protected final GameModel aModel;
    protected AbstractMove(GameModel pModel){
        aModel = pModel;
    }
    public final void perform() {
        aModel.pushMove(this);
        execute();
        log();
    }
    protected abstract void execute();
    private void log() {
        System.out.println(getClass().getName());
    }
}
```

client

template method

step method

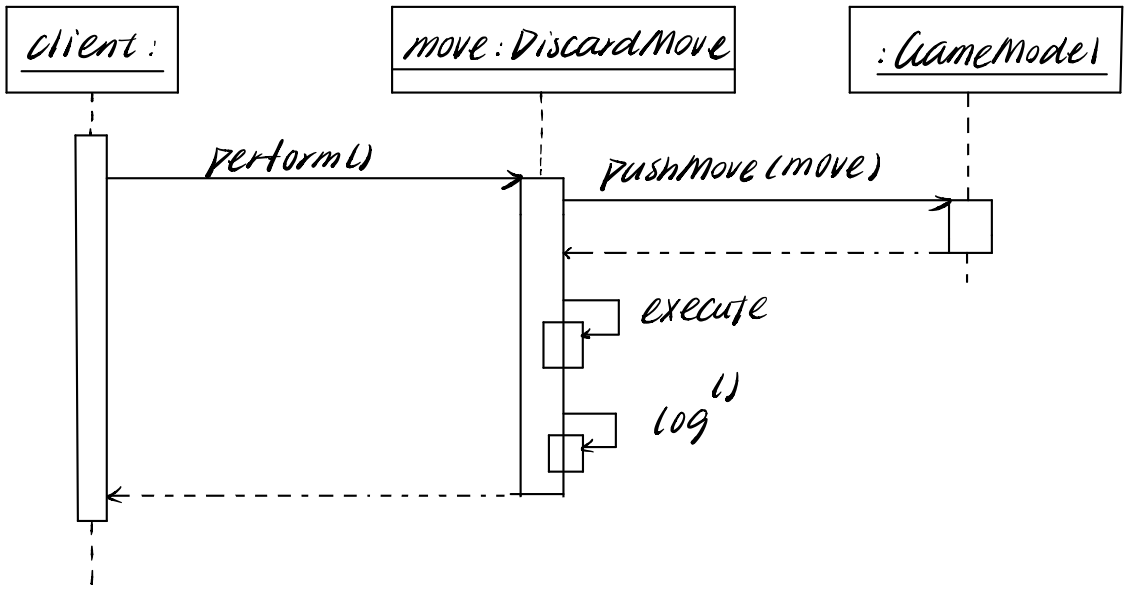
cannot be overridden by subclass

concrete method

abstract method

different name with template method

default method that can be overriden by subclass



Clone — polymorphic object copying

tagging interface

```
public class Deck ① implements Cloneable {

② public Deck clone() {
overriden △
        try {
            Deck clone = (Deck) ③ super.clone();
            clone.aCards = new CardStack(clone.aCards);
            return clone;
        ④ } catch (CloneNotSupportedException e) {
            assert false;
            return null;
        }
    }
}
```

⑤ Adding clone() to an interface. (optional)

```
CardSource extends Cloneable {
    CardSource clone();
}
```

MVC

model: keep a unique copy of the data

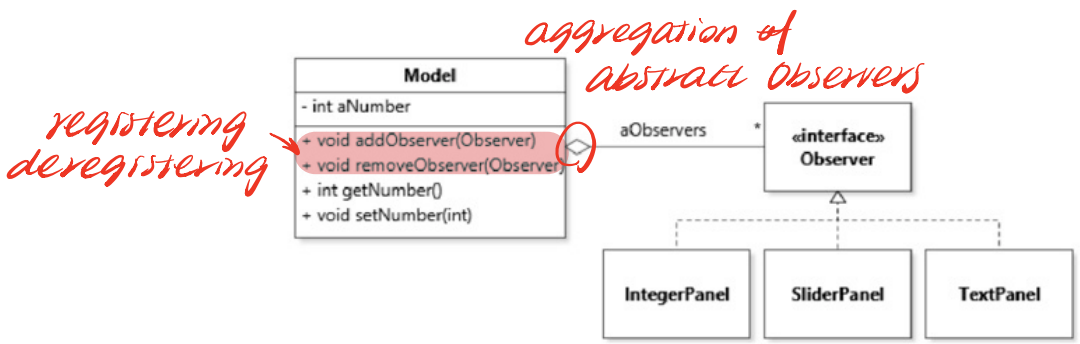
view: one view of the data

controller: change the data stored in the model.

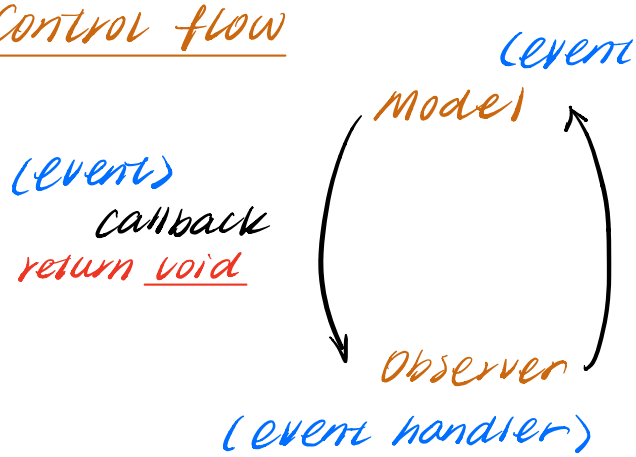
Observer

→ subject, model, observable

Store data of interest in a specialized object, and to allow other objects to observe this data.



Control flow



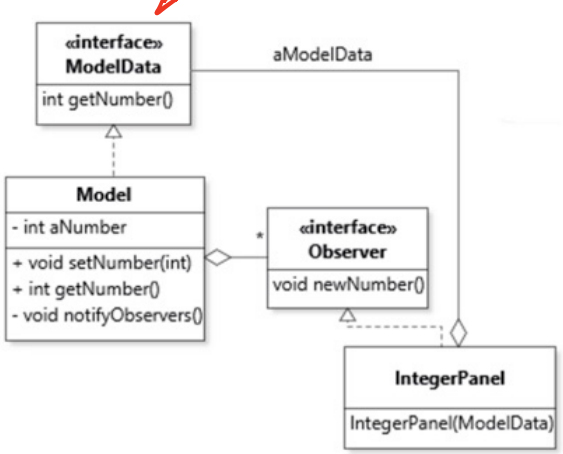
① push Assume type is known
make the info available through para of the callback.

② pull

reference to the model (*) { argument in callback / initialize Observerver with of (*) as a field.

```
private void notifyObservers()
{
  for(Observer observer : aObservers)
  { observer.newNumber(aNumber); }
}

public void setNumber(int pNumber)
{
  if( pNumber <= 0 ) { aNumber = 1; }
  else if( pNumber > 10 ) { aNumber = 10; }
  else { aNumber = pNumber; }
  notifyObservers();
}
}
```

ISP



Event-Based Programming

"do nothing" classes – adapter

```
public class ObserverAdapter implements Observer
{
  public void increased(int pNumber) {}
  public void decreased(int pNumber) {}
  public void changedToMax(int pNumber) {}
  public void changedToMin(int pNumber) {}
}


class IncreaseDetector extends ObserverAdapter
{
  public void increased(int pNumber)
  { System.out.println("Increased to " + pNumber); }
}
```

```
public interface ChangeObserver
{
  void increased(int pNumber) {}
  void decreased(int pNumber) {}
}

public interface BoundsReachedObserver
{
  void changedToMax(int pNumber) {}
  void changedToMin(int pNumber) {}
}
```

In cases where an observer doesn't need to react to an event the unused callbacks can be implemented as "do nothing" method

Visitor

Define an operation of interest in a separate class and "inject" it into the class hierarchy that needs to support it.

abstract visitor

```
public interface CardSourceVisitor
{
  void visitCompositeCardSource( CompositeCardSource pSource );
  void visitDeck( Deck pDeck );
  void visitCardSequence( CardSequence pCardSequence );
}
```

visitElementX(ElementX pElementX)

↓ concrete class

concrete visitor

```
public class PrintingVisitor implements CardSourceVisitor
{
 public void visitCompositeCardSource(CompositeCardSource pSource)
 {}

 public void visitDeck(Deck pDeck)
 {
   for( Card card : pDeck)
   { System.out.println(card); }
 }

 public void visitCardSequence(CardSequence pCardSequence)
 {
   for( int i = 0; i < pCardSequence.size(); i++ )
   { System.out.println(pCardSequence.get(i)); }
 }
}
```

⇒ Organize code in terms of functionality as opposed to data.

Integrating Operations into a Class Hierarchy

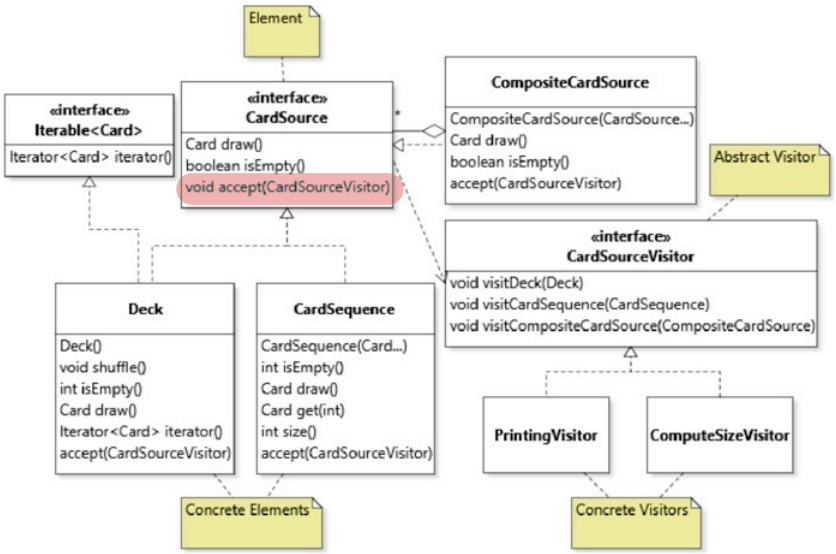
common super type

```
public interface CardSource
{
  Card draw();
  boolean isEmpty();
  void accept(CardSourceVisitor pVisitor);
}
```

abstract visitor

in Deck

```
public void accept(CardSourceVisitor pVisitor)
{ pVisitor.visitDeck(this); }
```



Traversing the Object Graph

① in the accept method of aggregate type.

② in the visit method that serve as call back for aggregate type.

Functional Design

① Function object

Functional interface — a single abstract method

② Lambda function

③ Model reference — C::m (no arguments)

- Comparator<T> — compare(T, T): int
- Predicate<T> — test(T): boolean

```
ArrayList<Card> cards = ...
cards.removeIf(Predicate<T>)

Predicate<Card> redFilter = blackFilter.negate()
```

- Comparator.comparing(...) — create a comparator by substracting a comparable from input argument.

thenComparing(...)
reversed()

- Supplier<T> — get(): T
- Function<T, R> — apply(T): R
- Consumer<T> — accept(T): ()

Stream (a flow of data)

```
Stream<Card> aStream = aCards.stream();
```

- obatain a stream : call the stream() method on an instance of a collection class.

Intermediate operations

Input : a stream as its implicit parameter
Output : a different stream.

① sorted() implements Comparable()

② Stream.concat(new Deck().stream(), new Deck.stream())

③ distinct() : remove duplicates

④ filter(Predicate<T>) → apply(T) : R

⑤ map(Function<T, R>)

mapToInt(T -> int).sum() → IntStream

Terminal operations

count()

limit(int max) : return up to max elements from the stream

Higher-order Function

① forEach(Consumer<T>) → accept(T) : ()

forEachOrdered

② allMatch / anyMatch / noneMatch(Predicate<T>) → test(T) : boolean

↓ boolean

reducing { accumulator, collector

```
int sum =
    numbers.reduce(0, (a,b) -> a+b);

collect(Collectors.toList());
```