Jin L.C. Guo

M2 (b) - Types and Polymorphism

Image Source: https://upload.wikimedia.org/wikipedia/commons/2/2b/Cepaea_nemoralis_active_pair_on_tree_trunk.jpg

Recall of last class

- Programming mechanism:

Java Interface type, Subtype polymorphism

- Concepts and Principles:

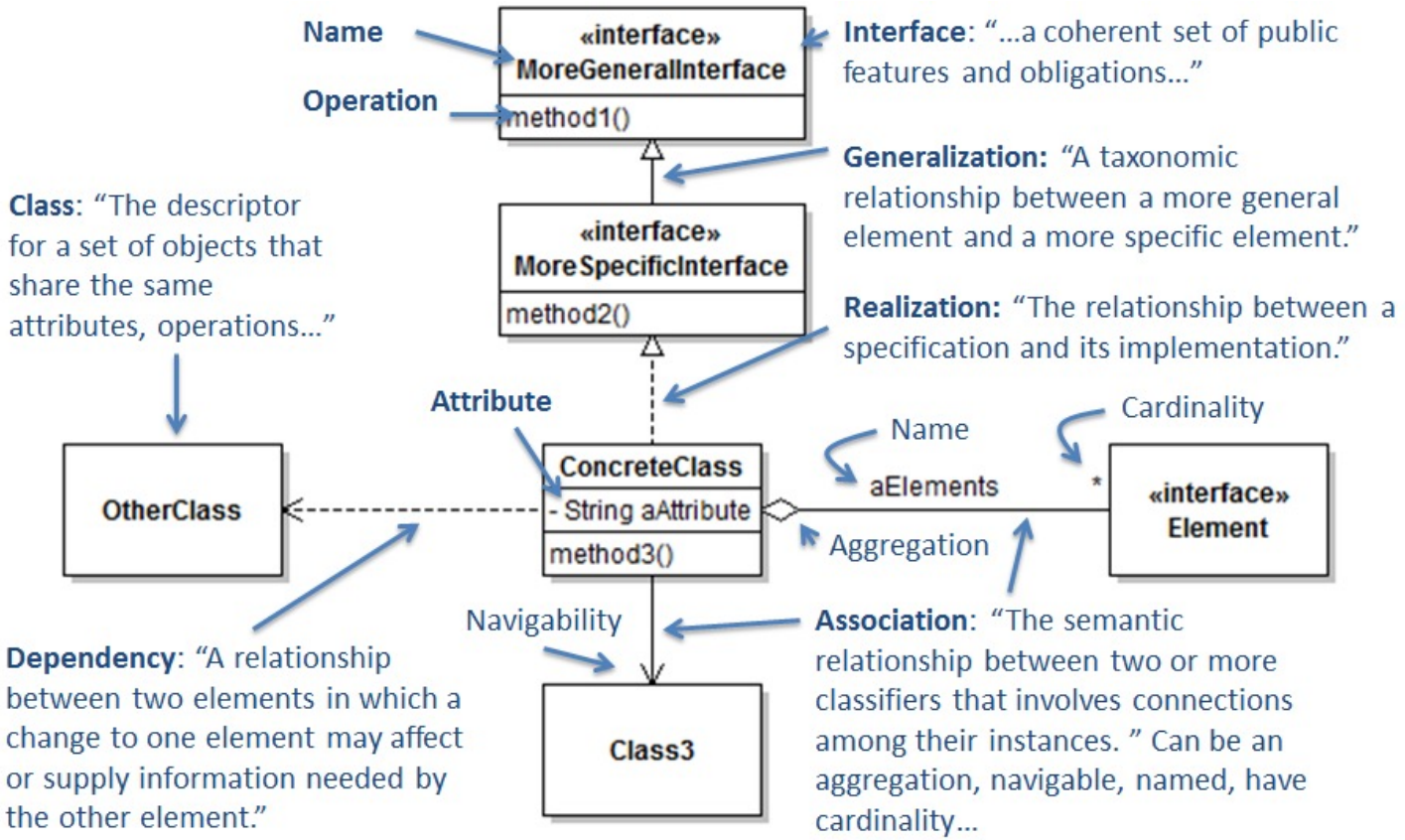
class's interface, Separation of concerns

- Design techniques:

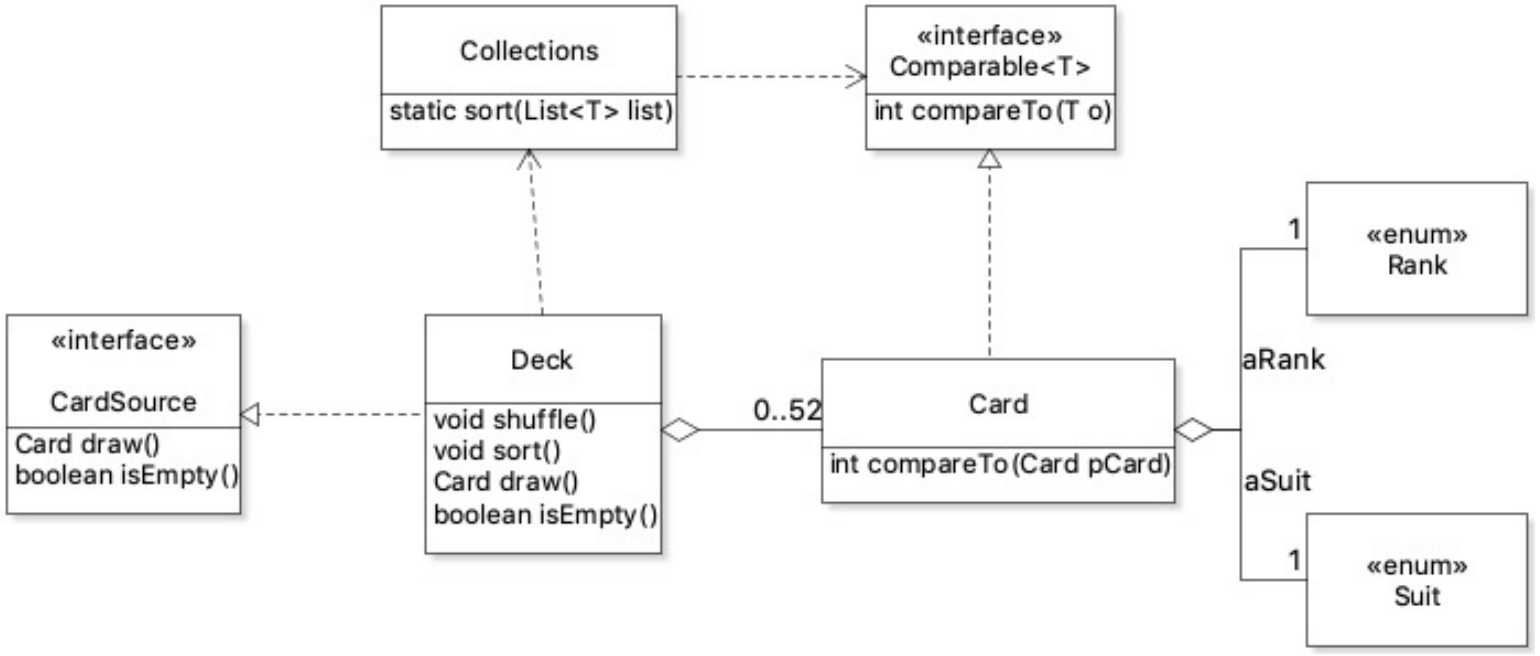
Interface-based behavior specification, UML Class Diagrams

UML Class Diagram

- Represent Type (mainly classes and interfaces) definitions and relations
- *Static* view (cannot show *run-time* properties)
- Tool: JetUML


Name
«interface»
MoreGeneralInterface
Operation
method1()
Interface: "...a coherent set of public features and obligations..."
Generalization: "A taxonomic relationship between a more general element and a more specific element."
«interface»
MoreSpecificInterface
method2()
Realization: "The relationship between a specification and its implementation."
Class: "The descriptor for a set of objects that share the same attributes, operations..."
Attribute
Name
Cardinality
ConcreteClass
aElements
*
«interface»
Element
OtherClass
- String aAttribute
method3()
Aggregation
Navigability
Association: "The semantic relationship between two or more classifiers that involves connections among their instances. " Can be an aggregation, navigable, named, have cardinality...
Dependency: "A relationship between two elements in which a change to one element may affect or supply information needed by the other element."
Class3

Current Design of Deck



Separation of Concern

- Concern: anything that matters in providing a solution to a problem
- Prevent information Leakage
- To achieve "orthogonality": changes in one does not affect any of the others.

Implements `Comparable<T>`

```
Collections.sort(aCards);// aCards is a List<Card> instance
```

```
public class Card implements Comparable<Card>
{

 … …


   @Override
    public int compareTo(Card pCard)
    {
    … …   return aRank.compareTo(pCard.aRank);
    }
}
```

compreTo in Enum

abstract enum

cant override

compareTo

```
public final int compareTo(E o)
```

Compares this enum with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object. Enum constants are only comparable to other enum constants of the same enum type. The natural order implemented by this method is the order in which the constants are declared.

Specified by:

`compareTo` in interface `Comparable<E extends Enum<E>>`

Parameters:

`o` - the object to be compared.

Returns:

a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

Objective

- Programming mechanism:

Java Generics, Java Nested Classes

- Concepts and Principles:

Separation of concerns;

- Patterns and Antipatterns:

STRATEGY, SWITCH Statement 

- Design techniques:

Function objects

Java Generics

variables one different but rest is the same

```
public interface ListOfCard {
    boolean add(Card pElement);
    Card get(int index);
}
```

```
public interface ListOfNumbers {
    boolean add(Number pElement);
    Number get(int index);
}
```

```
public interface ListOfIntegers {
    boolean add(Integer pElement);
    Integer get(int index);
}
```

… …

Java Generics

- Purpose: make the code reusable for many different types

```
boolean add(Number pElement);
Number get(int index);
```

```
public interface List<E> {
    boolean add(E pElement);
    E get(int index);
}
```

template

Java Generics

```
List<Card>  cards;
```

Type Argument

Generic type invocation(Parameterized Type)

- Generic Types
  - A class or interface whose declaration has one or more type parameter

: <E, T>

Convention:

E for Element
K for Key
V for Value
T for Type

good practice

Raw Type

Type Parameter/Variable

```
public interface List<E> {
    boolean add(E pElement);
    E get(int index);
}
```

Recall Java `Comparable<T>` Interface

- This interface imposes a total ordering on the objects of each class that implements it.

```
public interface Comparable<T>
{
        int compareTo(T o);
}


public class Card implements Comparable<Card>
{
        @Override
        public int compareTo(Card pCard)
        {
                …
        }

}
```

pCard.

Activity 1: Design a generic class that represents a pair of objects with the same type.

```
public interface pair<T>{
    final private T first;
    final private T second;

    public pair(T one, T two){
        first = one;
        second = two;
    }
    public T getFirst(){return first;}
    public T getSecond(){return second;}
}
```

```java
public class Pair<T>
{
   final private T aFirst;
   final private T aSecond;

   public Pair(T pFirst, T pSecond)
   {
      aFirst = pFirst;
      aSecond = pSecond;
   }

   public T getFirst() { return aFirst; }
   public T getSecond() { return aSecond; }

}
```

```java
Pair<Card> pair =
    new Pair<>(new Card(Rank.FIVE, Suit.CLUBS),
               new Card(Rank.FOUR, Suit.CLUBS));
Card card1 = pair.getFirst();
```

Type Inferred by Compiler

a = attribute
p = parameter

Pair<E,T> for two different types

Java Generics

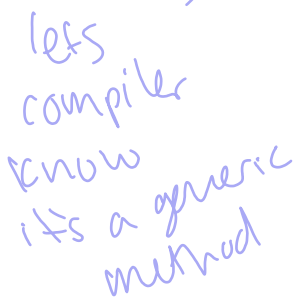
- Generic Method
  - A method that takes type parameters

emptySet method in java.util.Collections:

```
public static <T> Set<T> emptySet()
```

Type Parameter

Between Modifier and Return Type



Activity 2:

Write a static generic method that add elements of Pair in any type to a collection of the same type.

```
public class Pair<T>
{
   final private T aFirst;
   final private T aSecond;

   public Pair(T pFirst, T pSecond)
   {
      aFirst = pFirst;
      aSecond = pSecond;
   }

   public T getFirst() { return aFirst; }
   public T getSecond() { return aSecond; }
}
```

Interface Collection<E>

```
boolean add(E e)
```

```
static <T> void toCollection(Pair<T> pPair, Collection<T> pCollection) {
    pCollection.add(pPair.getFirst());
    pCollection.add(pPair.getSecond());
}
```

Activity 2

Write a generic method that add elements of Pair in any type to a collection of the same type.

```
/*
* Add the elements of type T stored in Pair to a Collection of Type T
* @pre pair !=null && collection != null
* @pre pair.getFirst()!=null && pair.getSecond()!=null
* @post collection.contains(pair.getFirst()) && collection.contains(pair.getSecond())
*
* @see Pair
*/
static <T> void fromPairToCollection(Pair<T> pair, Collection<T> collection) {
    /* assertion on pre conditions*/
    collection.add(pair.getFirst());
    collection.add(pair.getSecond());
    /* assertion on post conditions*/
}
```

all must be same type

Adding Restriction on Type Variables

```
public class Pair<T>
{
   final private T aFirst;
   final private T aSecond;

   public Pair(T pFirst, T pSecond)
   {
      aFirst = pFirst;
      aSecond = pSecond;
   }

   public T getFirst() { return aFirst; }
   public T getSecond() { return aSecond; }

}
```

Adding Restriction on Type Variables

```
public class Pair<T extends Deck>
{
   final private T aFirst;
   final private T aSecond;

   public Pair(T pFirst, T pSecond)
   {
      aFirst = pFirst;
      aSecond = pSecond;
   }

   public T getFirst() { return aFirst; }
   public T getSecond() { return aSecond; }

   public boolean isTopCardSame()
   {
       Card topCardInFirst = aFirst.draw();
       Card topCardInSecond = aSecond.draw();
       return topCardInFirst.equals(topCardInSecond);
   }
}
```

Type can only be Deck or its subtype

call methods of Deck

because it extends Deck

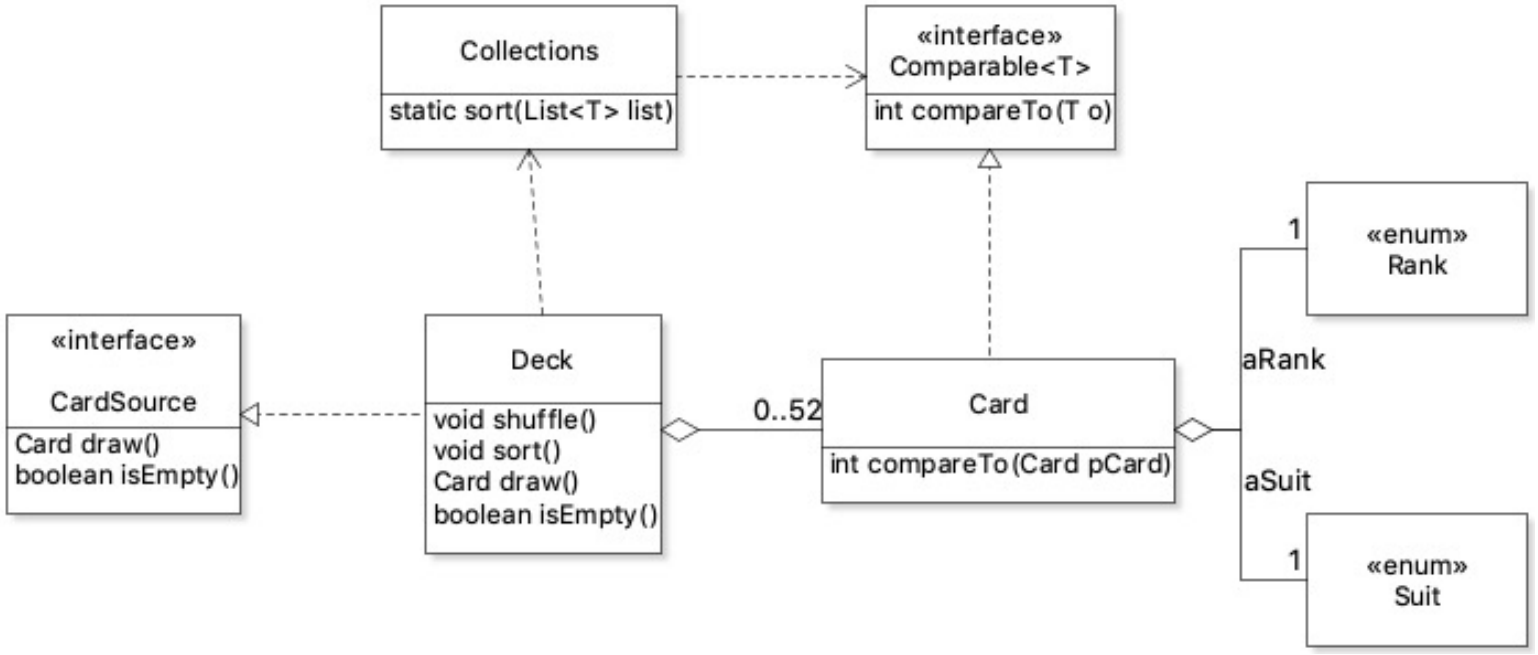
Generic Method With Type Bound

for generic method

```
static <T extends Deck>
   void fromPairToCollection(Pair<T> pair, Collection<T> collection) {}
```

only accepts decks or deck subtypes

Back to the sort method for comparable types



Back to the sort method for comparable types

- In java.util.collections

wild card

```
public static <T extends Comparable<? super T>>  void  sort(List<T> list)
```

```
class Card implements Comparable<Card> {…}
class FancyCard extends Card {…}
```

```
List<FancyCard> fancyCardList = new ArrayList<>();
Collections.sort(fancyCardList);
```

Objective

- Programming mechanism:

Java Generics, Java Nested Classes

- Concepts and Principles:

Separation of concerns;

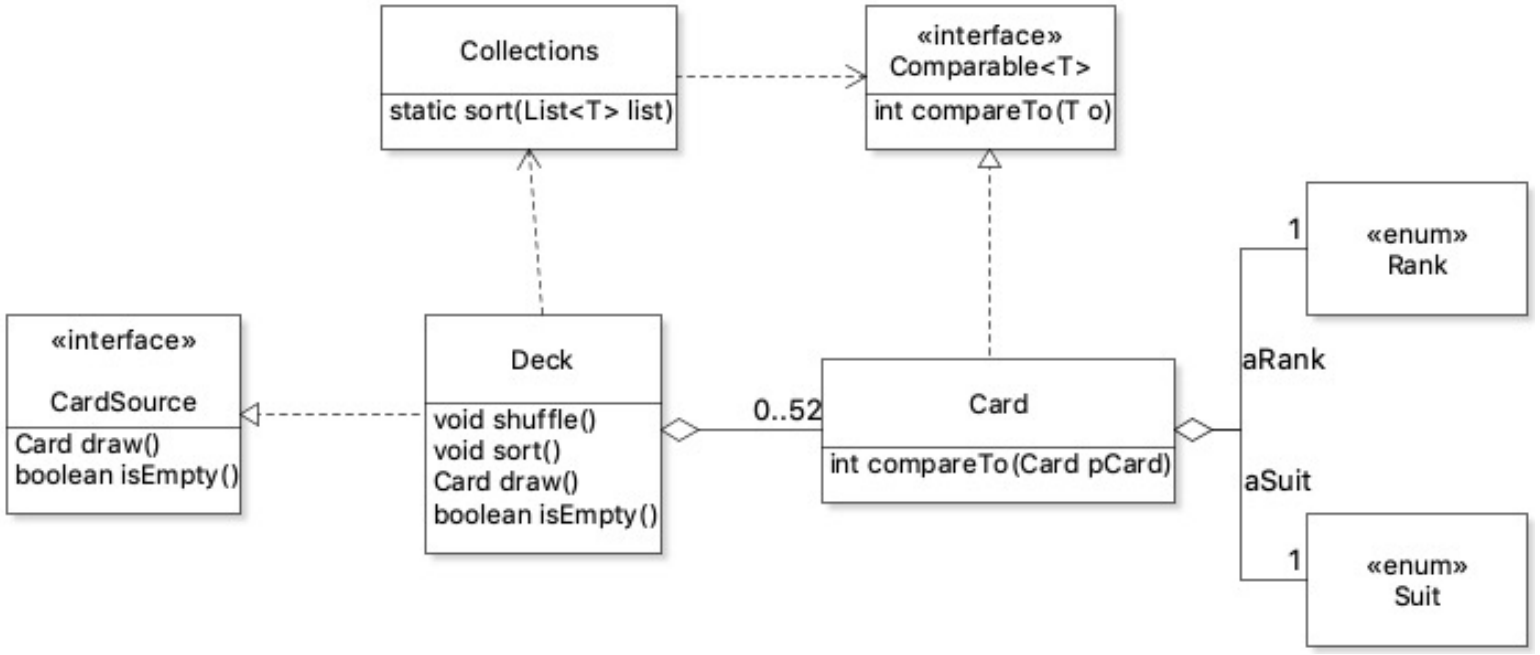
- Patterns and Antipatterns:

STRATEGY, SWITCH Statement 

- Design techniques:

Function objects

Current Design of Deck



How to support more than one strategy to compare cards?

Activity 3

Design a UniversalComarator that can compare two cards with more than one strategies including by rank, suit, reversed rank, suit first then rank.

```
public class UniversalComparator {
  public enum strat { byRank, bySuit, reversedRank, suitRank }
  final private strat astrategy;
  public UniversalComparator(strat pstrategy) {
    astrategy = pstrategy;
  }
  public int compar(Card c1, Card c2) {
    switch :
```

```
public class UniversalComparator {
    public enum ComparisonStrategy {ByRank, BySuit, ByRankThenSuit}

    ComparisonStrategy aStrategy;
    public UniversalComparator(ComparisonStrategy pStrategy) {
        aStrategy = pStrategy;
    }
    public int compare(Card c1, Card c2) {
        switch (aStrategy) {
            case ByRank:
                return compareByRank(c1, c2);
            case BySuit:
                return compareBySuit(c1, c2);
            case ByRankThenSuit:
                return compareByRankThenSuit(c1, c2);
            default:
                throw new AssertionError(this);
        }
    }

    private int compareBySuit(Card c1, Card c2) {
              …
    }
…}
```

not easy to accommodate additions

Recall Polymorphism

```
public class Undergrad implements Student

public class Graduate implements Student

public class NonDegreeStudent implements Student

public class VisitingStudent implements Student
```

Polymorphic **Student**

Program to the interface

```
public boolean attendSeminar(Student pStudent)
{
     if(registeredStudents.size()<=cap) {
          registeredStudents.add(pStudent.getID());
          return true;

     }
     return false;
}
```

Can we do the same thing for the compare strategy?

make it more expandable

Recall Polymorphism

```
public class ComparatorBySuit implements Comparator

public class ComparatorByRank implements Comparator

public class ComparatorBySuitThenRank implements Comparator

public class ComparatorByRankReverse implements Comparator
```

Polymorphic **Comparator**

Program to the interface

Client

```
public void sort(Comparator pComparator)
{
    …
        if pComparator.compare(card1, card2))

    …
}
```

Java Comparator Interface

- **Interface Comparator<T>**

```
public int compare(T o1, T o2)
```

Compares its two arguments for order. Returns a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second.

ByRank Comparator

parameterize the type

```
public class ByRankComparator implements Comparator<Card> {

    @Override
    public int compare(Card pCard1, Card pCard2) {
        return pCard1.getRank().compareTo(pCard2.getRank());
    }
}
```

enum compareTo

BySuit Comparator

```
public class BySuitComparator implements Comparator<Card>
{
    @Override
    public int compare(Card pCard1, Card pCard2) {
        return pCard1.getSuit().compareTo(pCard2.getSuit());
    }
}
```

enum compare to

Another sort method provided by Java Collections

- In java.util.collections

```
public static <T> void sort(List<T> list, Comparator<? super T> c)
```

comparator that knows how to compare T

wild card

```
Collections.sort(aCards, new ByRankComparator());
```

List<Card>

Objective

- Programming mechanism:

Java Generics, Java Nested Classes

- Concepts and Principles:

Separation of concerns;

- Patterns and Antipatterns:

STRATEGY, SWITCH Statement 👎

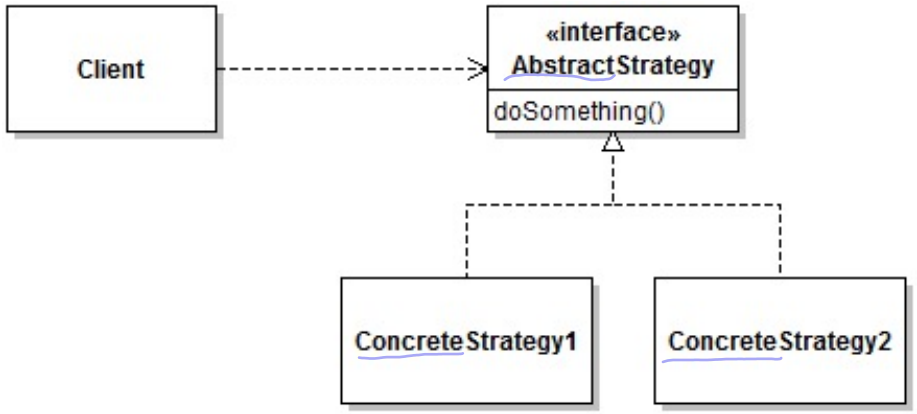
not easy to change or add

- Design techniques:

Function objects

Strategy Design Pattern

- Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.



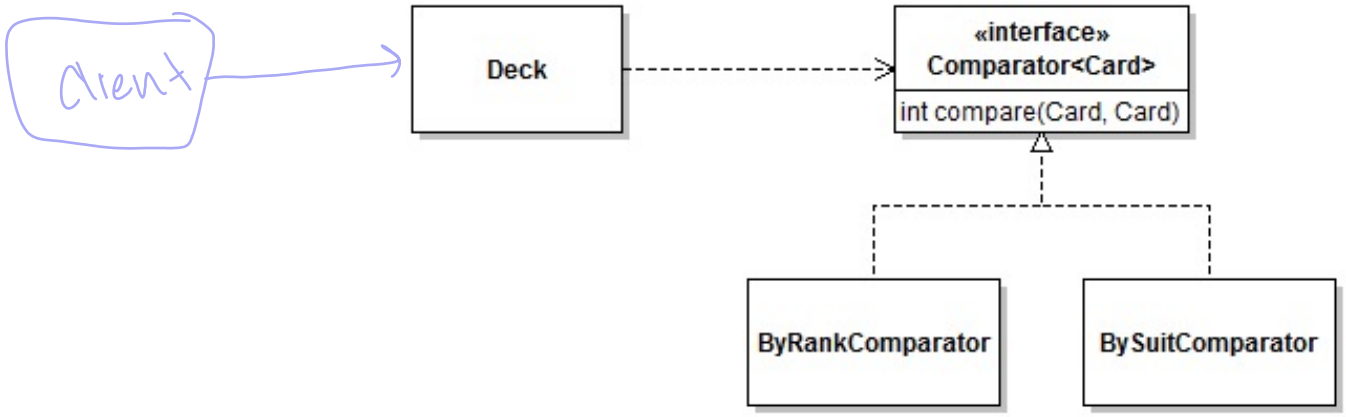
everything separate so they can be changed

Algorithms are appropriate at different times

New Algorithms need to be introduced when necessary

Strategy Design Pattern

- Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.



Objective

- Programming mechanism:

Java Generics, Java Nested Classes

- Concepts and Principles:

Separation of concerns;

- Patterns and Antipatterns:

STRATEGY, SWITCH Statement 

- Design techniques:

Function objects

Function Object

- An interface with single abstract method
- The actual function is achieved by the object of a class which implements that interface

Function Object

```
Collections.sort(aCards, new ByRankComparator());
```

- An interface with single abstract method
- The actual function is achieved by the object of a class which implements that interface

Function Object

```
Collections.sort(aCards, new ByRankComparator());
```

- An interface with single abstract method
- The actual function is achieved by the object of a class which implements that interface suitable choice?

Is the function is only used once?

Should the function have state?

Does the function need to access the private field?

Anonymous Class

- An inner class that is declared and instantiated at the same time.

```
class OuterClass
{
    public void method()
    {
        SuperType instance = new SuperType() {
                ⋯ ⋯
        };
    }

}
```

created on the fly

good for concision

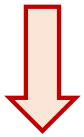
inner class

Anonymous Class for Function Object

```
public class ByRankComparator implements Comparator<Card> {
    @Override
    public int compare(Card pCard1, Card pCard2) {
        return pCard1.getRank().compareTo(pCard2.getRank());
    }

}
```

```
Collections.sort(aCards, new ByRankComparator());
```



Interface to implement or class to extend

```
Collections.sort(aCards, new Comparator<Card>() {
    public int compare(Card pCard1, Card pCard2) {
        return pCard1.getRank().compareTo(pCard2.getRank());
    }
});
```

Enable access to the private field

```
public class Card
{
    public static Comparator<Card> createByRankComparator()
    {
        return new Comparator<Card>()
            {
                @Override
                public int compare(Card pCard1, Card pCard2) {
                        return pCard1.aRank.compareTo(pCard2.aRank);
            }
        };
    }
}
```

private field

Objective

- Programming mechanism:

Java Generics, Java Nested Classes

- Concepts and Principles:

Separation of concerns;

- Patterns and Antipatterns:

STRATEGY, SWITCH Statement 

- Design techniques:

Function objects

Java Nested Classes

- Classes defined within another class
  - Static member class
  - Non-static member class
  - Local class
  - Anonymous class

Inner class

Static Member Class

Static

```
class OuterClass {
    ...
    static class StaticMemberClass {
        ...
    }
}
```

Stand alone but attatched to outerclass

```
OuterClass.StaticMemberClass nestedObject
        = new OuterClass.StaticMemberClass();
```

Non-Static Member Class

```
class OuterClass {
    ... private afield;
    class InnerClass {
        ... function() {
            ... afield ...
        }
    }
}
```

no static key word

attatched to an object created from outer class - has access to outer fields

```
OuterClass.InnerClass innerObject =
        outerObject.new InnerClass();
```

need this to access afield/outer object state

need an instance of outer object to use inner constructer

Local Class

local class can access final local variables

- An inner class that is defined in a block

inside method

```
class OuterClass
{
    public void method()
    {   int localInt = 2;   <- final
        class LocalClass implements Supertype {
            ……
        }
        Supertype instance = new LocalClass();
    }

}
```

class declaration

instance of class

can access local variables if its final

use when you want multiple instances within the method;
if you want one outside of the method, use a different class type

Anonymous Class

- An inner class that is declared and instantiated at the same time.

```
class OuterClass
{
    public void method()
    {
        SuperType instance = new SuperType() {
                ⋯ ⋯
        };
    }

}
```

creating an instance while declaring the class

Used when you only want 1 instance of a class inside the method

Summary

- Programming mechanism:

Java Generics, Java Nested Classes

- Concepts and Principles:

Separation of concerns;

- Patterns and Antipatterns:

STRATEGY, SWITCH Statement 

- Design techniques:

Function objects