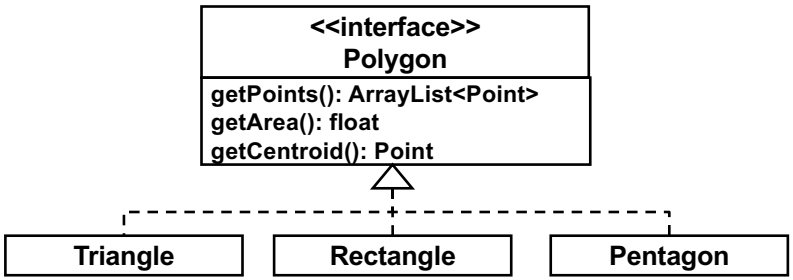
# Strategy Design Pattern

## *Strategy* Design Pattern

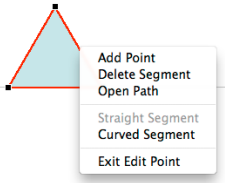
- Intent
  - Define a family of algorithms
  - Encapsulate each algorithm in a class
  - Separate one algorithm from another
    - Make those algorithms interchangeable



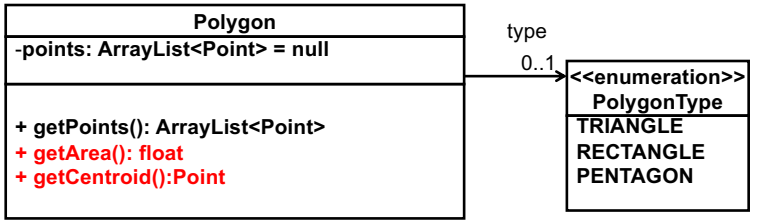
## Recap




- Can a triangle become a rectangle dynamically?
- If we allow that, eliminate class inheritance





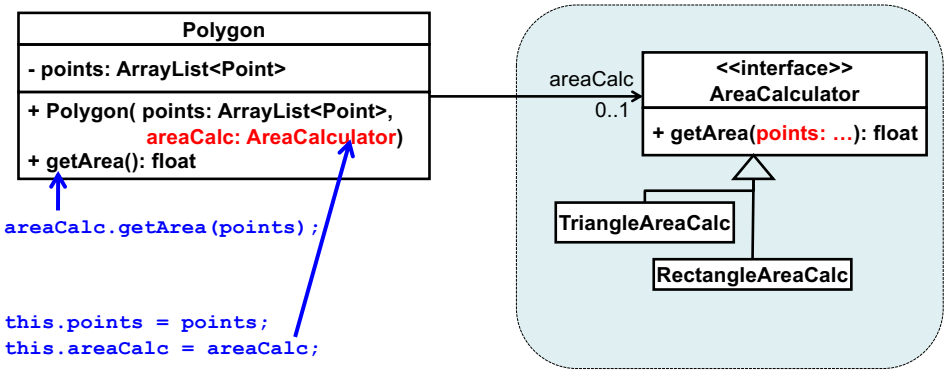

- Need to expect conditionals

# Suggested Read

- Replace Type Code with Class (incl. enumeration)
  - http://sourcemaking.com/refactoring/replace-type-code-with-class
- Replace Type Code with *Strategy*
  - http://sourcemaking.com/refactoring/replace-type-code-with-state-strategy
- Replace Type Code with Subclasses
  - http://sourcemaking.com/refactoring/replace-type-code-with-subclasses
- Replace Conditional with Polymorphism
  - http://sourcemaking.com/refactoring/replace-conditional-with-polymorphism



# An Example of *Strategy*



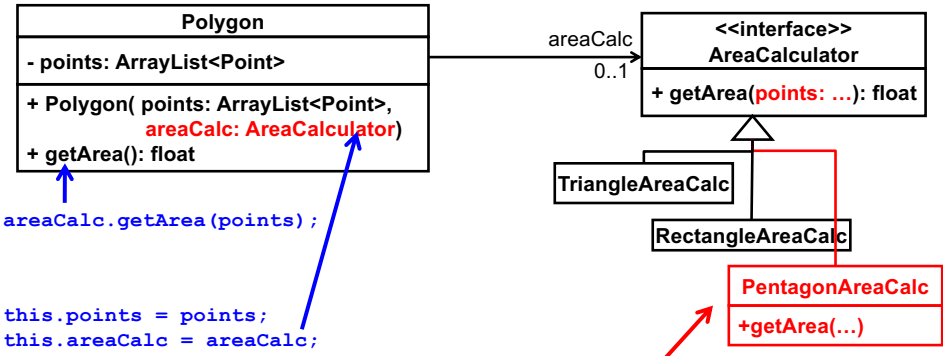

**Strategy Pattern:**
Area calculation is "strategized."

**User/client of Polygon:**

```
ArrayList<Point> al = new ArrayList<Point>();
al.add( new Point(…) ); al.add( new Point(…) ); al.add( new Point(…) );

Polygon p = new Polygon( al, new TriangleAreaCalc () );
p.getArea();
```






***A new area calculator NEVER alter existing code (i.e., Polygon, area calcurators, and clients of Polygon).***

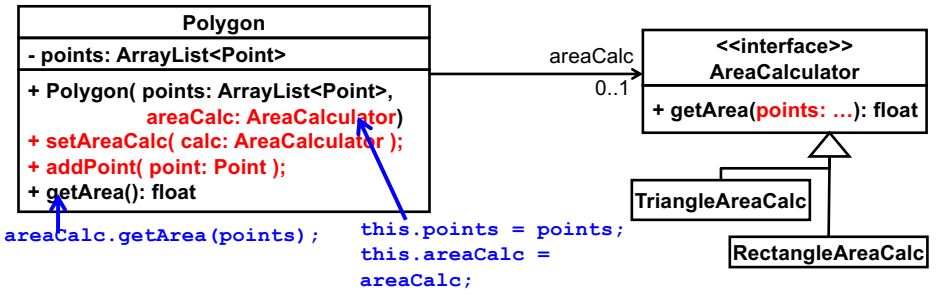
**User/client of Polygon:**

```
ArrayList<Point> a2 = new ArrayList<Point>();
al.add( new Point(…) ); a1.add(...); a1.add(...); a1.add(...);

Polygon p = new Polygon( a2, new PentagonAreaCalc() );
p.getArea();
```



# Polygon Transformation




**User/client of Polygon:**

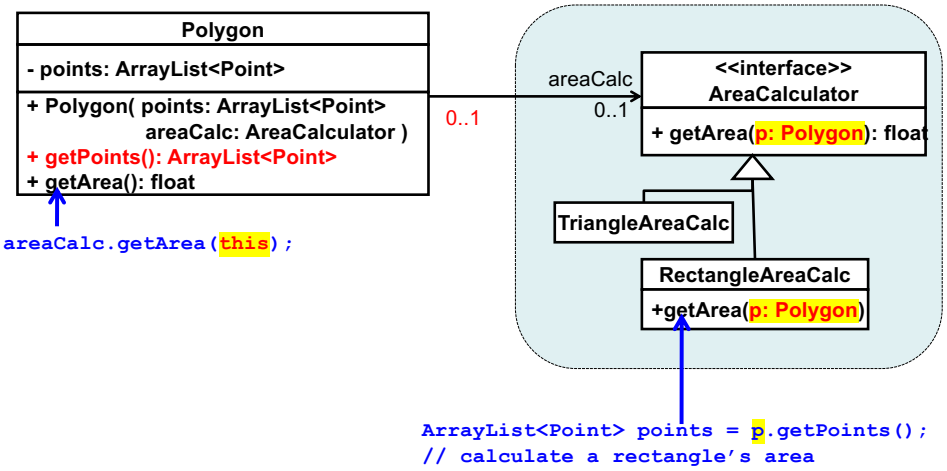
```
ArrayList<Point> al = new ArrayList<Point>();
al.add( new Point(…) ); al.add(...); al.add( ... );

Polygon p = new Polygon( al, new TriangleAreaCalc() );
p.getArea();    // triangle's area

p.addPoint( new Point(…) );
p.setAreaCalc( new RectangleAreaCalc() );
p.getArea();    // rectangle's area
```

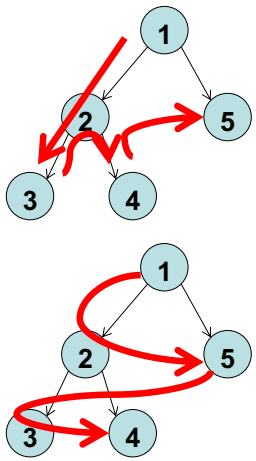
***No changes in existing code. Dynamic polygon transformation. Dynamic replacement of area calculators***

# An Alternative



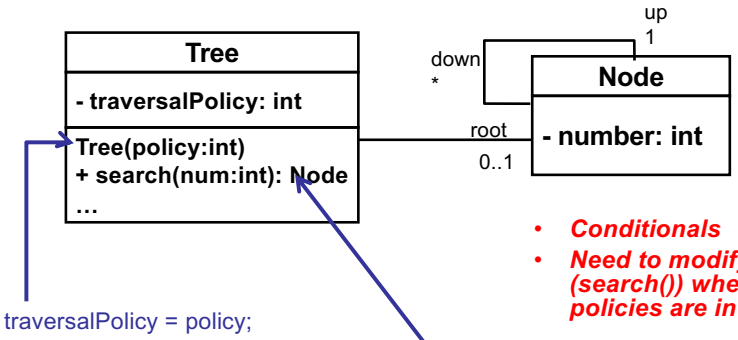



# Tree Traversal with *Strategy*




- Tree traversal
  - Visiting all nodes in a tree one by one
  - Many applications:
    - AI engine for strategy games (e.g. chess)
    - Maze solving
- Two major (well-known) algorithms
  - Depth-first
  - Width-first
- Assume you need to dynamically change one traversal algorithm to another.

# Not Good

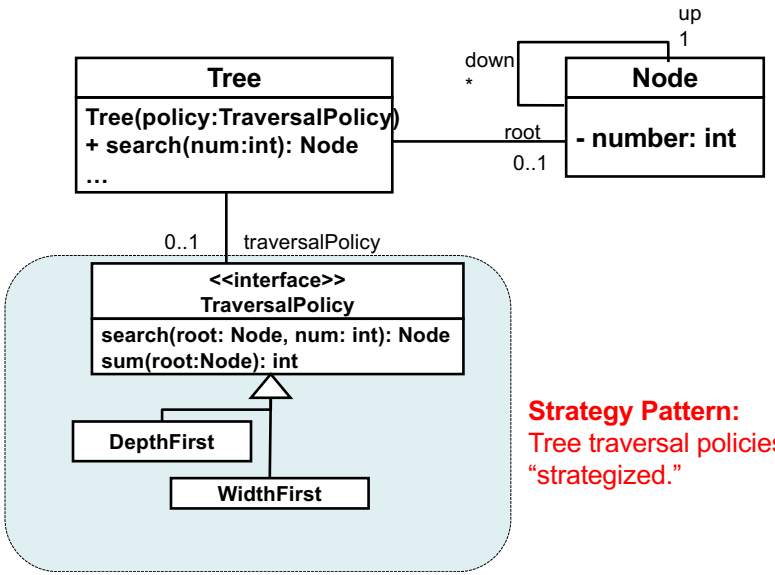



- ***Conditionals***
- ***Need to modify existing code (search()) when new traversal policies are introduced.***

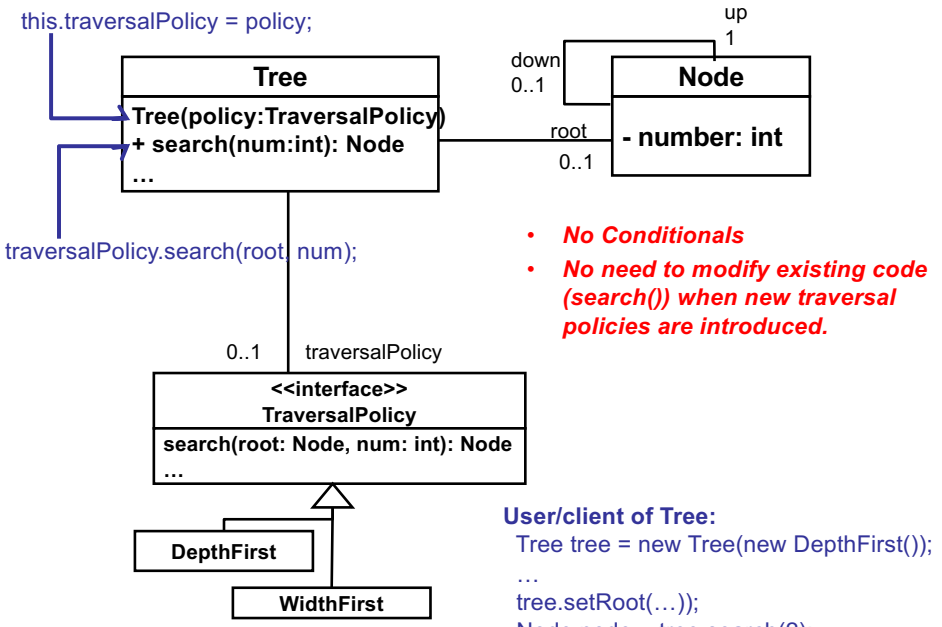
```
if(traversalPolicy==1){
  // do depth-first search
}
else if(traversalPolicy==0){
  // do width-first search
}
```



# With Strategy Classes…




**Strategy Pattern:**
Tree traversal policies are "strategized."




- ***No Conditionals***
- ***No need to modify existing code (search()) when new traversal policies are introduced.***

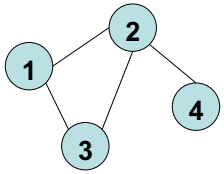
**User/client of Tree:**

```
Tree tree = new Tree(new DepthFirst());
…
tree.setRoot(…));
Node node = tree.search(2);
```



# Graph Traversal with *Strategy*

- A graph consists of nodes and links.
- Requirement: Find the shortest path between given two nodes.
  - 1 -> 2 -> 4: 2 hops between Node 1 and Node 4
  - 2 -> 3: 1 hop between Node 2 and Node 3



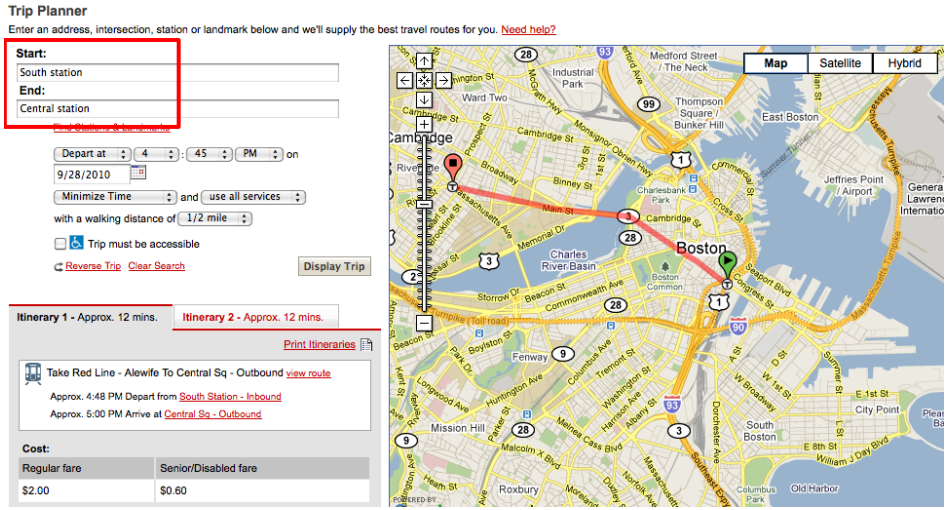

| Graph |
| --- |
| - graph: ArrayList<ArrayList<Boolean>> |
| + Tree(graph:ArrayList<ArrayList<Boolean>>)<br>+ getShortestPath(n1:int, n2:int): ArrayList<Integer> |

- 1 1 1 0<br>1 1 1 1<br>1 1 1 0<br>0 1 0 1

Connectivity matrix



# Trip Planner at mbta.org

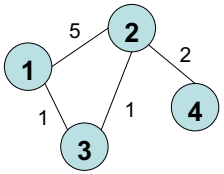



- Directions from one place to another via T (e.g. South Station to Central Sq.)
- This is a shortest path search problem.



# Weighted Graphs

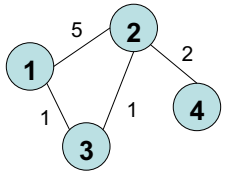
- What if you need to consider the *weighed* shortest path between two nodes?
  - 1 -> 3-> 2-> 4: total weight = 4, between Node 1 and Node 4
  - 1 -> 3 -> 2: total weight = 2, between Node 1 and Node 2




| Graph |
| --- |
| - graph: ArrayList<ArrayList<Integer>> |
| Tree(graph:ArrayList<ArrayList<Integer>>)<br>+ getShortestPath(n1:int, n2:int): ArrayList<Integer> |

- 0 5 1 -1<br>5 0 1 2<br>1 1 0 -1<br>-1 2 -1 0

Weighted connectivity matrix




| Graph |
| --- |
| - graph: ArrayList<ArrayList<Boolean>> |
| Tree(graph:ArrayList<ArrayList<Boolean>>)<br>+ getShortestPath(n1:int, n2:int): ArrayList<Integer> |

- ***Add conditional statements in getShortestPath()***
  - ***Conditional statements***
  - ***Need to modify existing code when new algorithms are introduced to compute the shortest path.***
- ***Add getWeightedShortestPath(…)***
  - ***No conditional statements***
  - ***Still need to modify existing code when new algorithms are introduced to compute the shortest path.***



| Graph |
| --- |
| - graph: ArrayList<ArrayList<Integer>> |
| Tree(graph:ArrayList<ArrayList<Integer>>)<br>+ getShortestPath(n1:int, n2:int): ArrayList<Integer> |

graph 1 — 0..1 shortestPathFinder

| <<interface>><br>ShortestPathFinder |
| --- |
| getShortestPath(graph:Graph, n1:int, n2:int): ArrayList<Integer> |

UnweigedShortestPathFinder

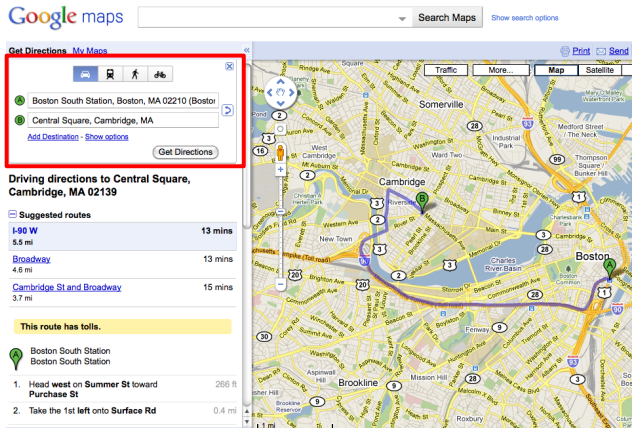
WeightedShortestPathFinder

**Strategy Pattern:**
Shortest path algorithms are “strategized.”

- ***No Conditional statements***
- ***No need to modify existing code when new algorithms are introduced.***
  - **Dijkstra’s (w>=0)**
  - **Bellman-Ford**
  - **Directed graphs**



# Google Maps




- Directions from one place to another (e.g. South Station to Central Sq.)
  - By car
  - By T
  - By walk
  - By bicycle
  - With a shared ride (e.g., Uber and Lyft).



GMaps — graph 1 — pathFinder 0..1

| <<interface>><br>PathFinder |
| --- |
| getPath(graph:Graph, n1:int, n2:int, …): … |

DrivePathFinder

WalkPathFinder

**Strategy Pattern:**
Path finding algorithms are “strategized.”

# Comparators in Java API

- Sorting array elements:
  - `int years[] = {2010, 2000, 1997, 2006};`

```
Arrays.sort(years);
for(int y: years)
    System.out.println(y);
```

  - `java.util.Arrays`: a utility class (a collection of static methods) to process arrays and array elements
  - `sort()` sorts array elements in an ascending order.
    - 1997 -> 2000 -> 2006 -> 2010



- Sorting collection elements:
  - `ArrayList<Integer> years2 = new ArrayList<Integer>();`

```
years2.add( new Integer.valueOf(2010) );
years2.add( new Integer.valueOf(2000) );
years2.add( new Integer.valueOf(1997) );
years2.add( new Integer.valueOf(2006) );

Collections.sort(years2);
for(Integer y: years2)
    System.out.println(y);
```

  - `java.util.Collections`: a utility class (a collection of static methods) to process collections and collection elements
  - `sort()` sorts array elements in an ascending order.
    - 1997 -> 2000 -> 2006 -> 2010



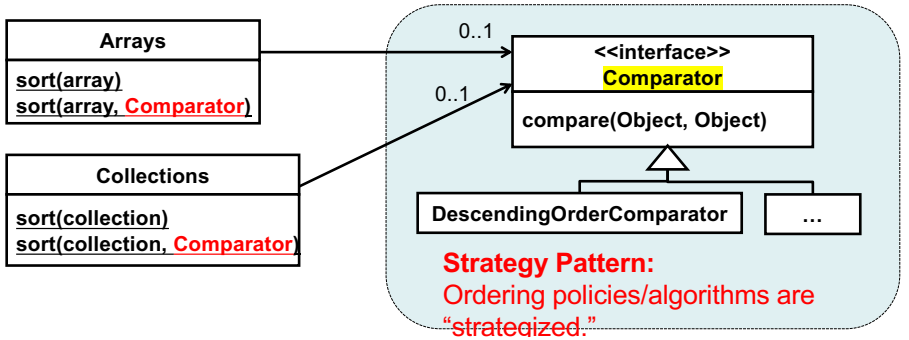
# Comparison/Ordering Policies

- What if you want to sort array/collection elements in a descending order or any specialized (user-defined) order?
  - `Arrays.sort()` and `Collections.sort()` implement ascending ordering only.
    - They do not implement any other policies.



# Comparison/Ordering Policies

- Java API allows you to define a custom comparator (i.e., your own comparator) by implementing `java.util.Comparator`.



- **`Arrays.sort()`** and **`Collections.sort()`** are defined to sort array/collection elements from "smaller" to "bigger" elements.
  - By default, "smaller" elements mean the elements that have *lower* numbers.
- A descending ordering can be implemented by treating "smaller" elements as the elements that have *higher* numbers.
- **`compare()`** in comparator classes can define what "small" means and what's "big" means.
  - Returns a negative integer, zero, or a positive integer as the first argument is "smaller" than, "equal to," or "bigger" than the second.
- 
```
public class DescendingOrderComparator implements Comparator{
  public int compare(Object o1, Object o2){
    return ((Integer)o2).intValue()-((Integer) o1).intValue(); }}
```



# Sorting Collection Elements with a Custom Comparator

```
ArrayList<Integer> years = new ArrayList<Integer>();
years.add(new Integer(2010)); years.add(new Integer(2000));
years.add(new Integer(1997)); years.add(new Integer(2006));

Collections.sort(years);
for(Integer y: years)
    System.out.println(y);

Collections.sort(years, new DescendingOrderComparator());
for(Integer y: years)
    System.out.println(y);
```

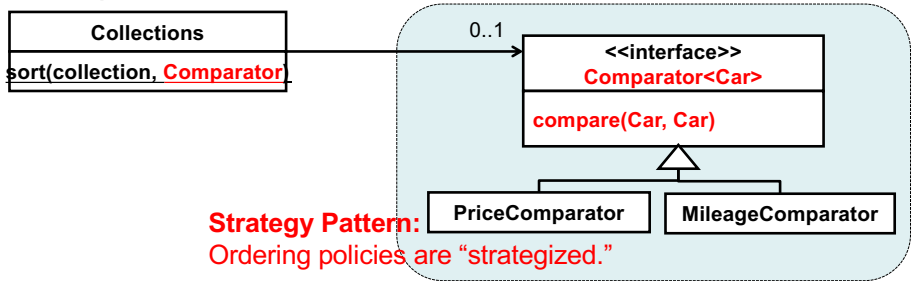
- 1997 -> 2000 -> 2006 -> 2010
- 2010 -> 2006 -> 2000 -> 1997



- 
```
public class DescendingOrderComparator implements Comparator{
  public int compare(Object o1, Object o2){
    return ((Integer)o2).intValue()-((Integer) o1).intValue();
  }
}
```
- A more type-safe option is recommended:
- 
```
public class DescendingOrderComparator{
  implements Comparator<Integer>{
    public int compare(Integer o1, Integer o2){
      return o2.intValue()-o1.intValue();
    }
}
```



- What if you want to sort a collection of your own (i.e., user-defined) objects?
  - 
```
public class Car {
  private String model, make;
  private int mileage, year;
  private float price;  }
```
    - c.f. HW 4
  - 
```
ArrayList<Car> usedCars= new ArrayList<Car>();
usedCars.add(new Car(…)); usedCars.add(…); ...
Collections.sort(usedCars, ...);
```
- Can define a car-ordering policy as a custom comparator class.




**Strategy Pattern:**
Ordering policies are "strategized."

- Assume “bigger” cars are better cars to buy
  - “Bigger” cars as the ones with
    - Lower mileage
    - Higher (more recent) year
    - Lower price

```
public class PriceComparator
  implements Comparator<Car>{
    public int compare(Car car1, Car car2){
      return car2.getPrice() - car1.getPrice();
    }
}
public class YearComparator
  implements Comparator<Car>{
    public int compare(Car car1, Car car2){
      return car1.getYear() - car2.getYear();
    }
}
```



# Thanks to *Strategy*…

- You can define any extra ordering policies without changing existing code
  - e.g., Car, Collections.sort()
  - No conditionals to shift ordering policies.
- You can dynamically change one ordering/comparison policy to another.

```
Collections.sort(usedCars, new PriceComparator());
// printing a list of cars
Collection.sort(usedCars, new YearComparator());
// printing a list of cars
```



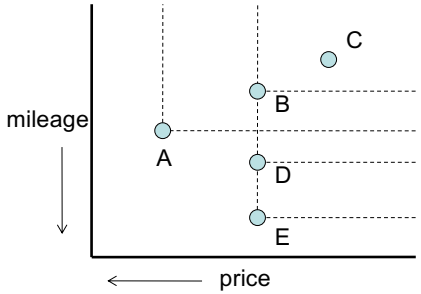
# Used Car Listings


# HW 10

- Step 1: Implement three comparator classes for the Car class
- Step 2: Implement an extra comparator class, `ParetoComparator<Car>`, which performs the *Pareto comparison*.
- Test cases make several cars and sort them with four comparators.
- Due: Nov 11 midnight

# Pareto Comparison

- Given multiple objectives (or criteria),
  - e.g., price, year and mileage
- Car A is said to ***dominate*** (or outperform) Car B iif:
  - A's objective values are superior than, or equal to, B's in all objectives, and
  - A's objective values are superior than B's in at least one objective.




  - Count the number of cars that dominate each car.
    - A: 0 (No cars dominate A.)
    - B: 3 (A, D, E)
    - C: 4 (A, B, D, E)
    - D: 1 (E)
    - E: 0 (No cars dominate E.)
  - Better cars have lower "domination counts."
    - To order cars from the best one(s) to the worst one(s), compare() should treat "better" ones as "smaller" ones.

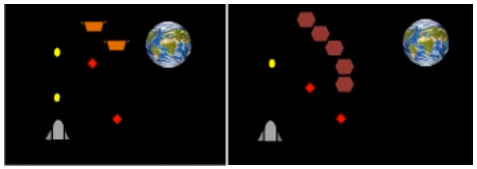


- Implement `setDominationCount()` and `getDominationCount()` in Car.
- When to compute domination counts (i.e., when to call `setDominationCount()`) for individual cars?
  - Before calling sort()

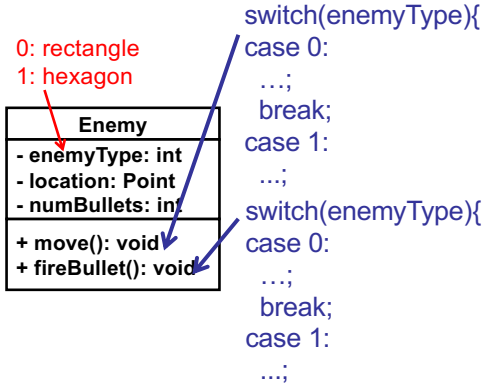
```
// Finish up setting the domination counts for all cars
// by calling setDominationCount() on those cars, and
// then call sort()
for(car: usedCars){
  car.setDominationCount(...); }
Collections.sort(usedCars, new DominationComparator<Car>());
```

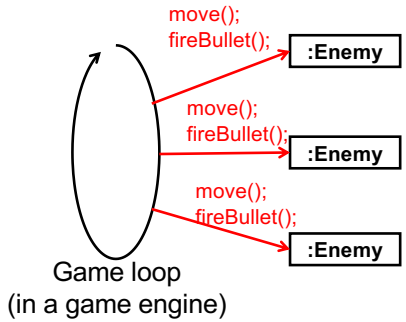


# One More Exercise
# Imagine a Simple 2D Shooting Game



- Each type of enemies has its own attack pattern.
  - e.g. How to move, when to fire bullets, how to fire bullets, etc.

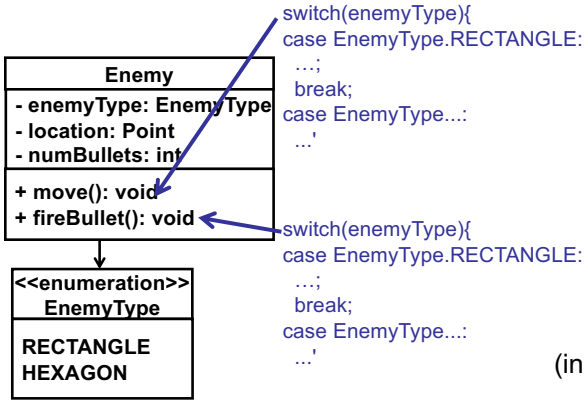


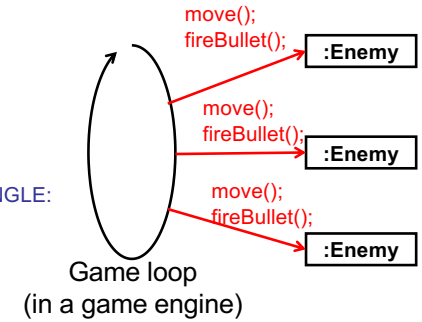






# What's Bad?

- Using magic numbers.
  - Replace them with symbolic constants or an enumeration.



# Still Not Good

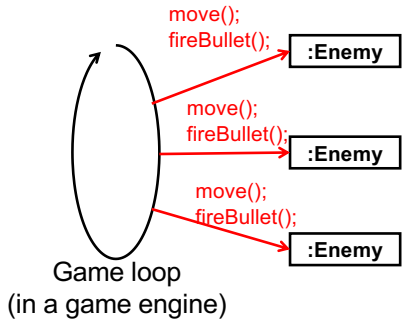
- Conditional blocks. Error-prone to maintain them
  - If there are many enemy types.
  - If new enemy types may be added in the near future.
    - Imagine 3,000 to 5,000 lines of code for each conditional branch
  - If repetitive conditional blocks exist.

- Attack patterns (moving patterns and firing patterns) are *tightly coupled* with Enemy. Hard to maintain them
  - If attack patterns often change.
    - Keeping the same attack pattern for rectangle and hexagonal enemies during a game.
    - Changing rectangle enemy's attack pattern to be more intelligent as you play in a game
    - Introducing a new type of enemies and having them use hexagonal enemy's attack pattern
    - Introducing a new type of enemies and implementing a new pattern for them.



# What We Want are to…

- Eliminate those conditional branches.
- Separate Enemy and its attack patterns (moving patterns and firing patterns).
  - Make Enemy and its attack patterns *loosely coupled*.
- Define a family of attack patterns (algorithms) in a unified way
- Encapsulate each algorithm in a class
- Make algorithms interchangeable



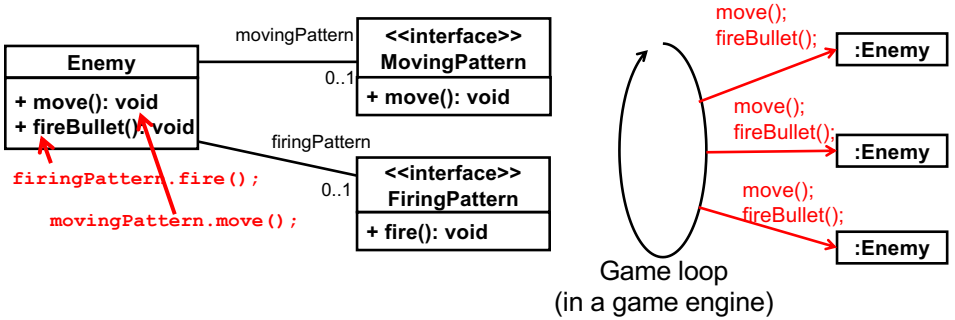


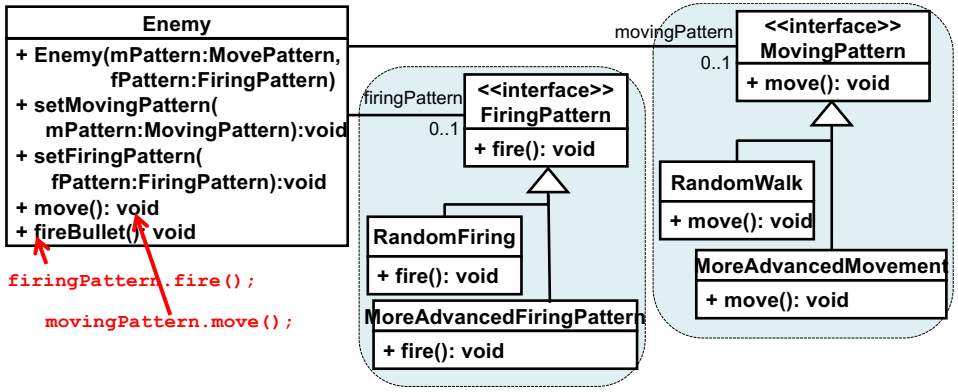

# Suggested Read

- Replace Type Code with Class (incl. enumeration)
  - http://sourcemaking.com/refactoring/replace-type-code-with-class
- Replace Type Code with *Strategy*
  - http://sourcemaking.com/refactoring/replace-type-code-with-state-strategy
- Replace Type Code with Subclasses
  - http://sourcemaking.com/refactoring/replace-type-code-with-subclasses
- Replace Conditional with Polymorphism
  - http://sourcemaking.com/refactoring/replace-conditional-with-polymorphism



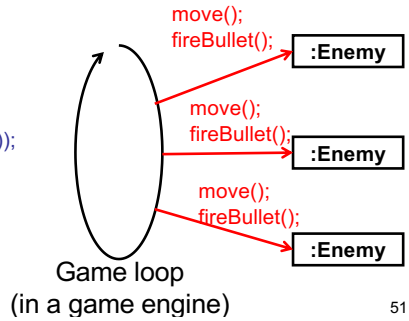
# Revised Design with *Strategy*






**User/client of Enemy:**

```
Enemy e1 = new Enemy(new RandomWalk(…),
                     new RandomFiring(…));
Enemy e2 = new Enemy(new RandomWalk(…),
                     new MoreAdvancedPattern(…));
Enemy e3 = …
ArrayList<Enemey> el = new ArrayList<Enemy>();
el.add(e1); el.add(e2); el.add(e3);
for(Enemy e: el){
   e.move();
   e.fireBullet(); }
```






# Null Object Design Pattern

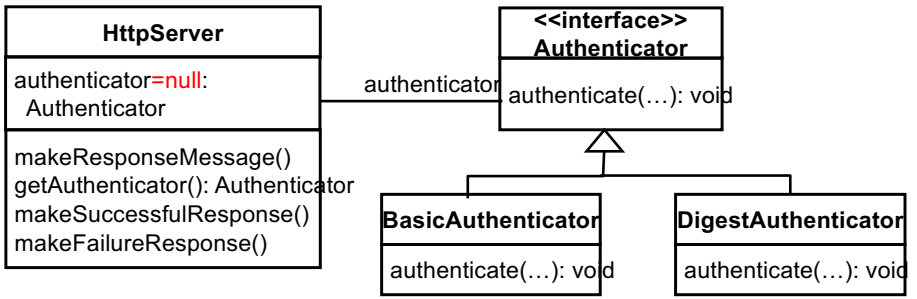


# Null Object Design Pattern

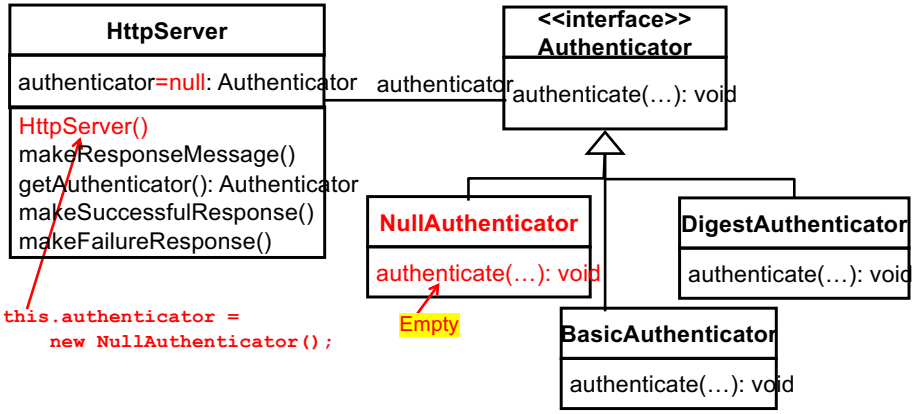
- Intent
  - Encapsulate the implementation decisions of how to do nothing and hide those details from clients
  - Replace a *null-checking* (i.e., conditional) with a neutral/default object that does nothing.
  - B. Woolf, "Null Object," Chapter 1, PLoP 3, Addison-Wesley, 1998.
  - Refactoring: Introduce Null Object
    - http://sourcemaking.com/refactoring/introduce-null-object



# An Example: Authentication in HTTP




```
makeResponseMessage(...){              // if an authenticator has been set,
  if( getAuthenticator() != null ){    // do authentication with it. Otherwise,
     try{                              // skip authentication.
        getAuthenticator().authenticate(...);
    }catch(AuthenticationFailedException afe){
        makeErrorResponse(afe);
    }
  }
  makeSuccessfulResponse(...);
}
```




```
makeResponseMessage(...){
  try{
    getAuthenticator().authenticate(...);
  }catch(AuthenticationFailedException afe){
    makeErrorResponse(afe);
  }
  makeSuccessfulResponse(...);
}
```



# *Null Object* as *Strategy*

- Null object
  - A variant/application of *Strategy* that focuses on "doing nothing" by default.