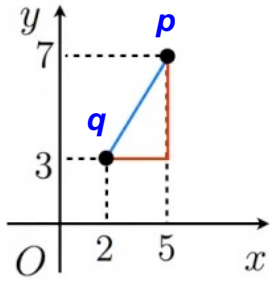
Distance Metrics for ML

- Suppose you are implementing a machine learning (ML) API.
  - You are expected to implement many distance metrics for various ML algorithms
    - e.g., Clustering

Example Distance Metrics

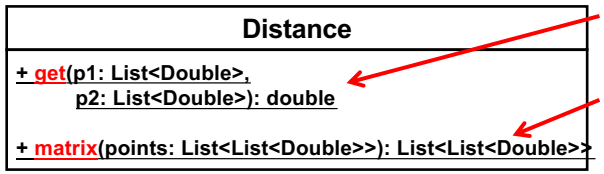
- Euclidean distance
  - $d_e(p,q) = \sqrt{\sum_{i=0}^{n}(p_i - q_i)^2}$
- Manhattan distance
  - $d_m(p,q) = \sum_{i=0}^{n}|p_i - q_i|$



- d_e = sqrt((5 - 2)^2 + (7 - 3)^2)
  = sqrt(3^2+4^2) = 5
- d_m = (5 - 2) + (7 - 3) = 7

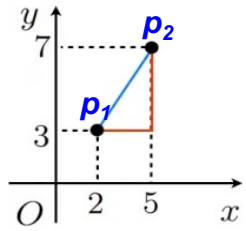
- There exists no "one-size-fits-all" metric.
  - Different algorithms and datasets require different metrics.
    - e.g., Standardized Euclidean, Chebyshev, Mahalanobis, Minkowski, Cosine, Jaccard, Canberra, Kulsinski, etc.
  - You (API designer) will implement extra metrics in the future, for sure.
  - Some API users will want to implement their own (custom) metrics, for sure.
- How to allow new distance metrics to be introduced and maintained in a pluggable way?

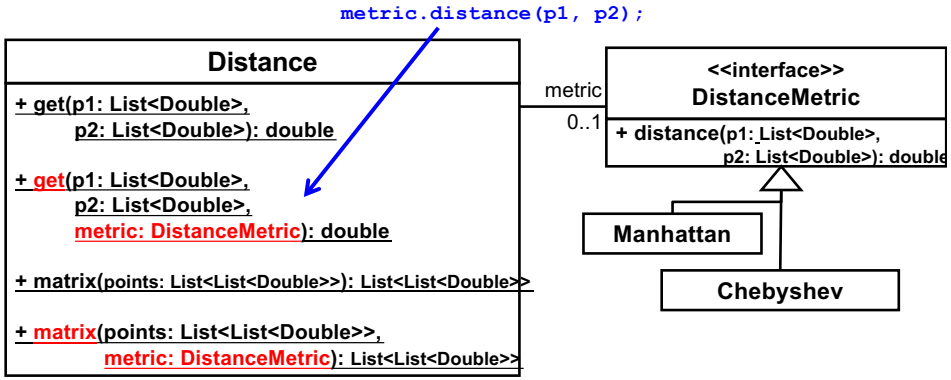
Using *Strategy* for Pluggable Distance Metrics



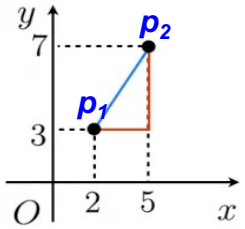
Client of Distance:

```
List<Double> p1, p2;
p1 = Arrays.asList(2.0,3.0);
p2 = Arrays.asList(5.0,7.0);
Distance.get(p1, p2);          // returns 5

List<List<Double>> points = new ArrayList<>();
points.add(p1); points.add(p2);
Distance.matrix(points);
                               // returns [[0,5],
                               //          [5,0]]
```






```
Distance.get(p1, p2);          // returns 5
Distance.get(p1, p2,
             new Manhattan());// returns 7

Distance.matrix(points);       // returns [[0,5],
                               //          [5,0]]
Distance.matrix(points,
                new Manhattan());
                               // returns [[0,7],
                               //          [7,0]]
```



HW 11

- Read `Distance.java` and other source code.
- Implement the Manhattan class, so you can…
  - compute the Manhattan distance b/w `p1` and `p2`
  - compute the distance matrix for given `points` with the Manhattan metric.
- Test Euclidean and Manhattan metrics with 5 or more 3-dimensional points
  - `Distance.matrix()` returns a 5 x 5 matrix.
  - [OPTIONAL] Implement one extra metric.

Note: `Arrays.asList()`

- `static List<T> asList(T... values)`
  - Takes an *arbitrary* number of `T`-typed values.
  - Creates and returns a list containing them.
    - `T`: Type of list elements
    - `T... values` is a syntactic sugar for `T[] values`.

```
- List<String> list = Arrays.asList ("U", "M", "B");
                      // a list of "U", "M" and "B" is returned.
```

```
- String[] strs = {"U", "M", "B"};
  List<String> list = Arrays.asList(strs);
                      // a list of "U", "M" and "B" is returned.
```


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

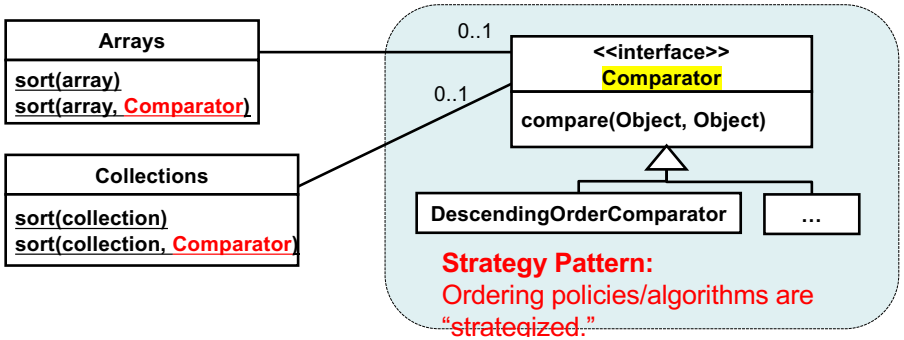
- Sorting collection elements:
  - ```
    ArrayList<Integer> years = new ArrayList<Integer>();
    years.add( new Integer.valueOf(2010) );
    years.add( new Integer.valueOf(2000) );
    years.add( new Integer.valueOf(1997) );
    years.add( new Integer.valueOf(2006) );

    Collections.sort(years);
    for(Integer y: years)
        System.out.println(y);
    ```
  - `java.util.Collections`: a utility class (a collection of static methods) to process collections and collection elements
  - `sort()` sorts collection 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 custom (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()` is designed to sort array/collection elements from "smaller" to "bigger" elements.
  - By default, "smaller" elements mean the elements that have *lower* numbers.
- Descending ordering can be implemented by treating "smaller" elements as the elements that have *higher* numbers.
- `compare()` in a comparator class 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

Type-safe Comparators

```
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:

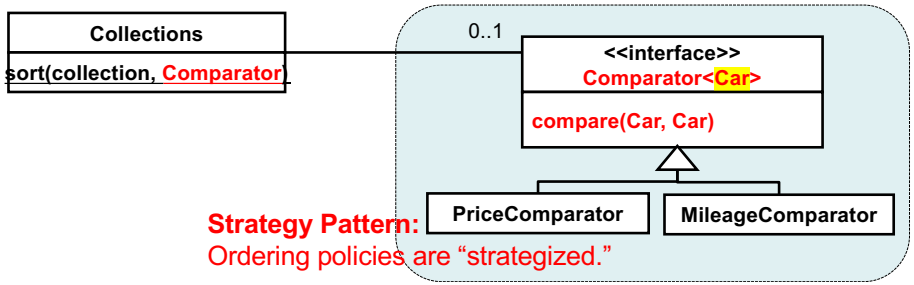
```
public class DescendingOrderComparator{
  implements Comparator<Integer>{
    public int compare(Integer o1, Integer o2){
      return o2.intValue()-o1.intValue();
    }
}
```


- 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;  }
```

```
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 "smaller" cars are better cars to buy
  - "Smaller" 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 car1.getPrice() - car2.getPrice();
    }
}
public class YearComparator
  implements Comparator<Car>{
    public int compare(Car car1, Car car2){
      return car2.getYear() - car1.getYear();
    }
}
```

- `Collections.sort()` returns the "best" car as the first element.

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 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

Year/Model	Information	Mileage	Seller/Distance	Price
2000 Audi A4 5dr Wgn 1.8T Avant Auto Quattro AWD	Used MPG: 19 Cty / 28 Hwy Automatic Gray	136,636	Dedham Auto Mall (7.4 Miles) Search Dealer Inventory	$4,880
2001 Audi A4	Used	84,297	Herb Connolly Hyundai (18.8 Miles) Search Dealer Inventory	$7,995
2002 Audi A6 4dr Sdn quattro AWD Auto	Used MPG: 17 Cty / 25 Hwy Automatic Blue	84,272	Dedham Auto Mall (7.4 Miles) Search Dealer Inventory	$7,998
2003 Audi A4 1.8T	Used MPG: 20 Cty / 28 Hwy Automatic Blue	78,321	Direct Auto Mall (18.8 Miles) Search Dealer Inventory	$10,697
2002 Audi allroad 5dr quattro AWD Auto	Used MPG: 15 Cty / 21 Hwy Automatic Green	98,362	Lux Auto Plus (8.6 Miles) Search Dealer Inventory	$10,900

Year/Model	Information	Mileage	Seller/Distance	Price
2008 Audi A6	Certified Pre-Owned MPG: 17 Cty / 25 Hwy Automatic	0	Audi Burlington & Porsche of Burlington (14.9 Miles) Search Dealer Inventory	$37,897
2007 Audi A4	Used MPG: 22 Cty / NA Hwy Brilliant Black	6,822	(19.3 Miles)	$24,995
2009 Audi A4	Certified Pre-Owned White	10,120	Audi Burlington & Porsche of Burlington (14.9 Miles) Search Dealer Inventory	$33,497
2009 Audi A4 3.2L Prestige	Certified Pre-Owned MPG: 17 Cty / 26 Hwy Automatic White	12,118	Audi Burlington & Porsche of Burlington (14.9 Miles) Search Dealer Inventory	$39,877
2008 Audi S5	Used Brilliant Black	16,492	(19.3 Miles)	$44,995

HW 12

- Step 1: Implement three comparator classes for the Car class
  - `PriceComparator<Car>`, `YearComparator<Car>` and `MileageComparator<Car>`

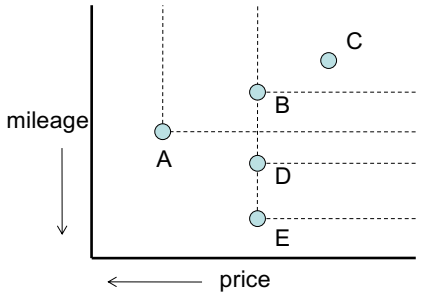
- Step 2: Implement an extra comparator class, `ParetoComparator<Car>`, which performs the *Pareto comparison*.

- Write and run 4 test cases to sort multiple Car instances with 4 comparators.

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.



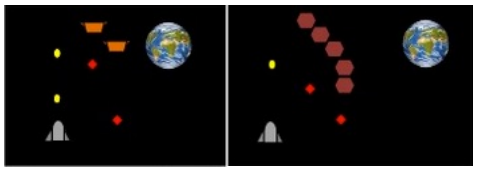
  - Count the number of cars that dominate each car.
    - A: 0 (No cars dominate A.)
    - B: 3 (A, D, E)
    - C: 4 (A, B, D, E)
    - D: 1 (E)
    - E: 0 (No cars dominate E.)
  - Better cars have lower "domination counts."
    - To order cars from the best one(s) to the worst one(s), compare() should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

- Implement `setDominationCount()` and `getDominationCount()` in Car.

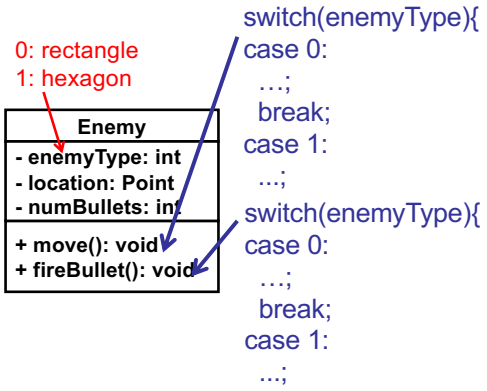
- When to compute domination counts (i.e., when to call `setDominationCount()`) for individual cars?
  - Before calling sort()

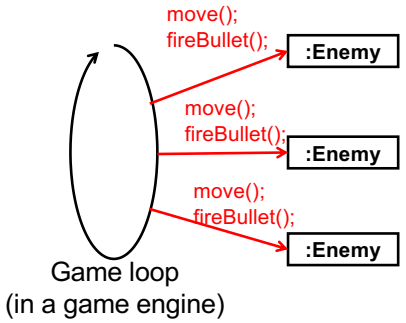
```
// Set domination counts for all cars by calling
// setDominationCount() on those cars, and
// then call sort()
for(car: usedCars){
  car.setDominationCount(...); }
Collections.sort(usedCars, new ParetoComparator<Car>());
```


One More Exercise
Imagine a Simple 2D Shooting Game



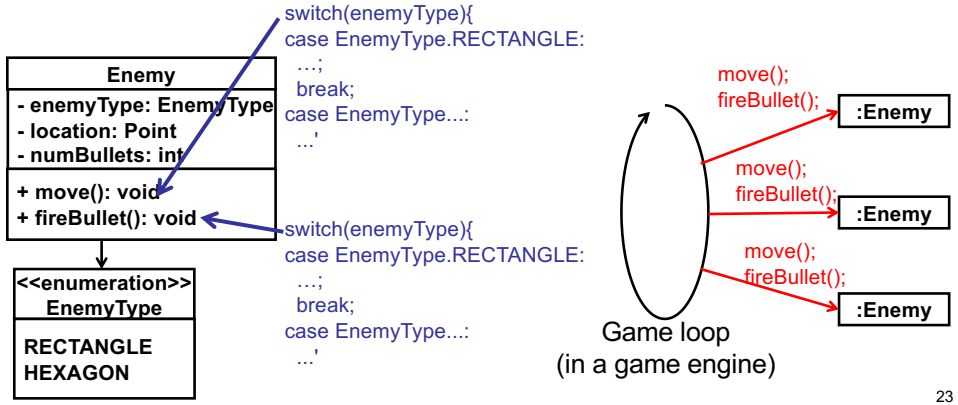
- Each type of enemies has its own attack pattern.
  - e.g. How to move, when to fire bullets, how to fire bullets, etc.





What's Bad?

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



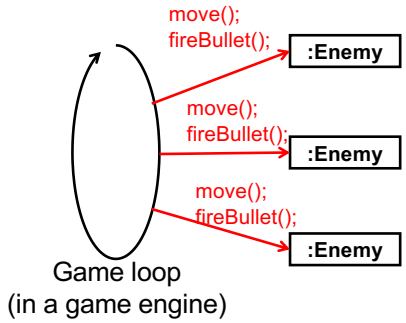
Still Not Good

- 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…

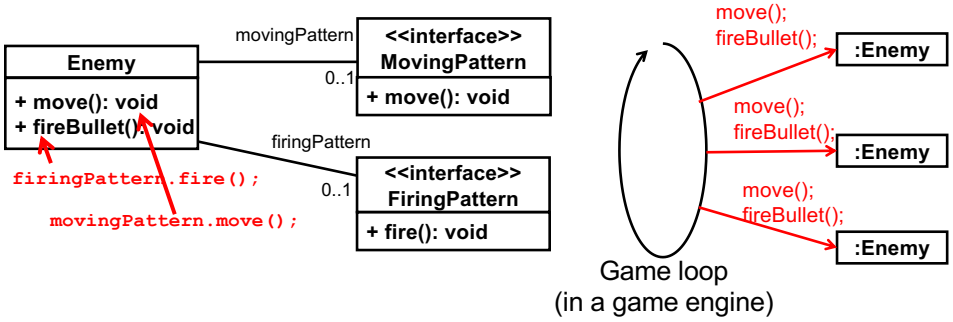
- Eliminate those conditional branches.
- Separate Enemy and its attack patterns (moving patterns and firing patterns).
  - Make Enemy and its attack patterns *loosely coupled*.
- Define a family of attack patterns (algorithms) in a unified way
- Encapsulate each algorithm in a class
- Make algorithms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

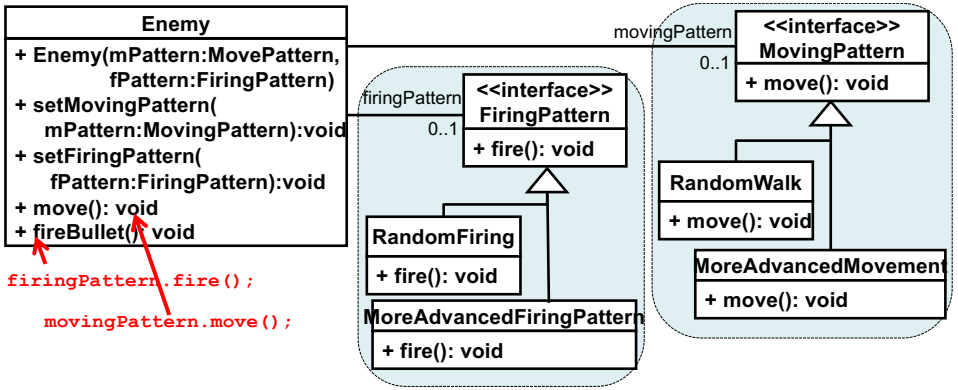


Suggested Read

- 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

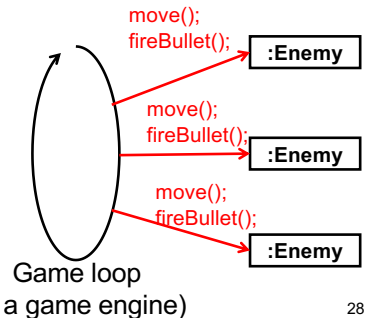
Revised Design with *Strategy*





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(); }
```



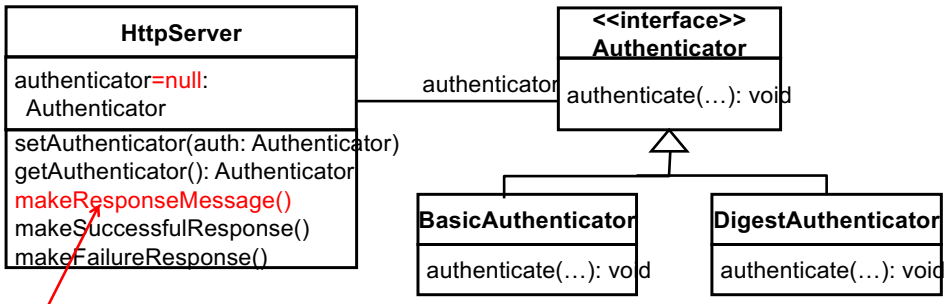
Null Object Design Pattern

Null Object Design Pattern

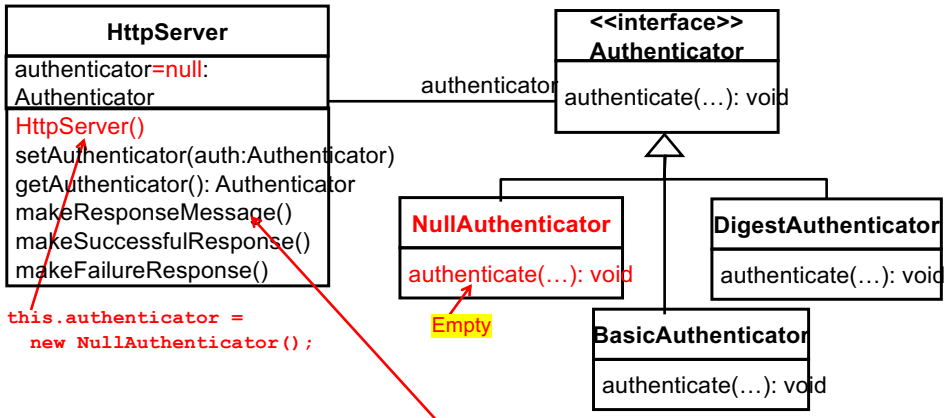
- Intent
  - Encapsulate the implementation decisions of how to do nothing and hide those details from clients
  - Replace a *null-checking* (i.e., conditional) with a neutral/default object that does nothing.

  - 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




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


```
this.authenticator =
  new NullAuthenticator();
```

```
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.