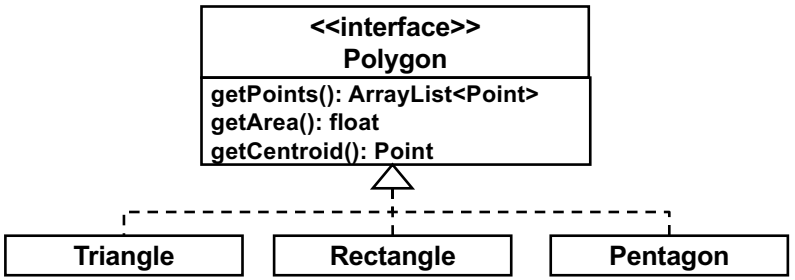
Strategy Design Pattern

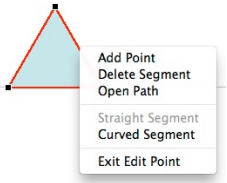
Strategy Design Pattern

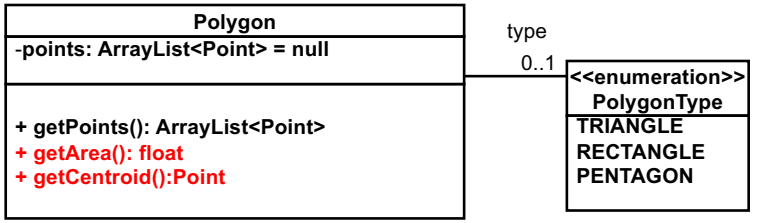
- Intent
  - Implement a family of algorithms as a set of classes
  - Encapsulate each algorithm in a class
  - Separate one algorithm from another
    - Make those algorithms pluggable/interchangeable.

Recap



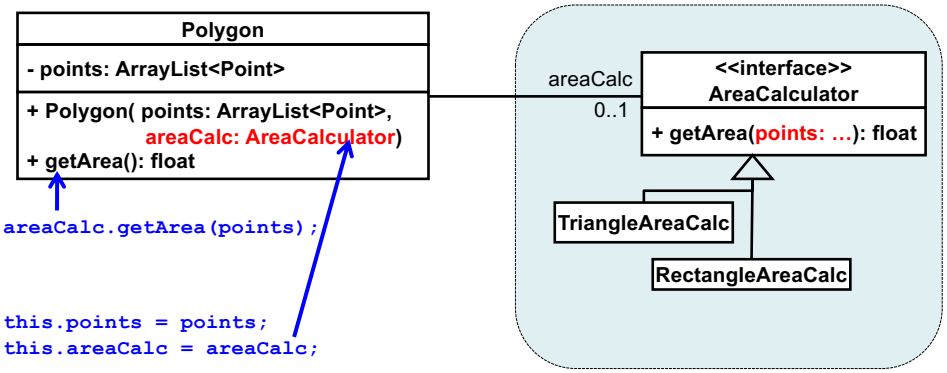
- Can a triangle become a rectangle dynamically?
- If we allow that, eliminate class inheritance





- Need to expect conditionals in some/many methods in Polygon.

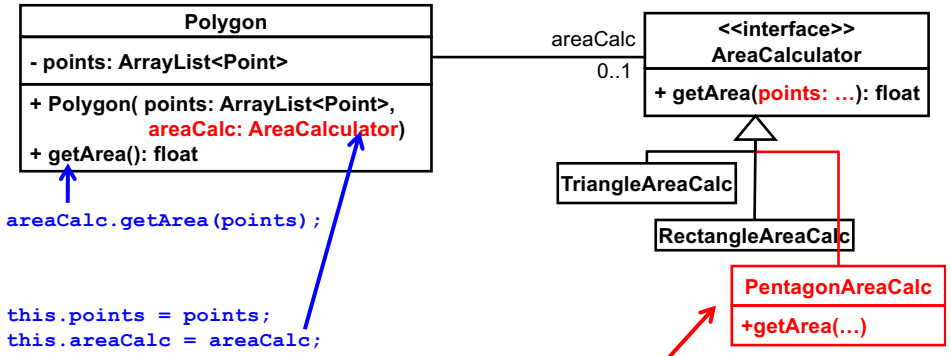
An Example of *Strategy*



Strategy Pattern:
Area calculation is "strategized."

Client of Polygon:

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

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

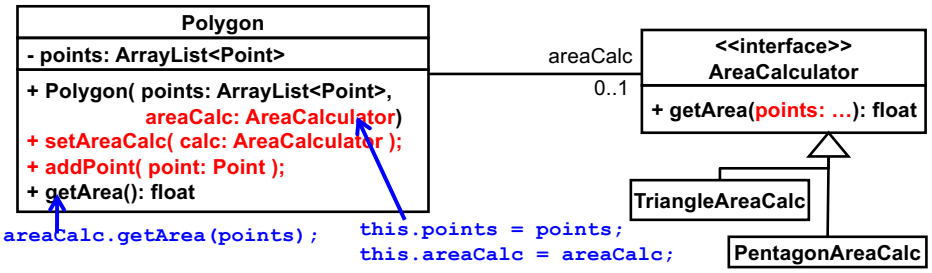



A new area calculator NEVER change existing code.

User/client of Polygon:

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

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


Polygon Transformation

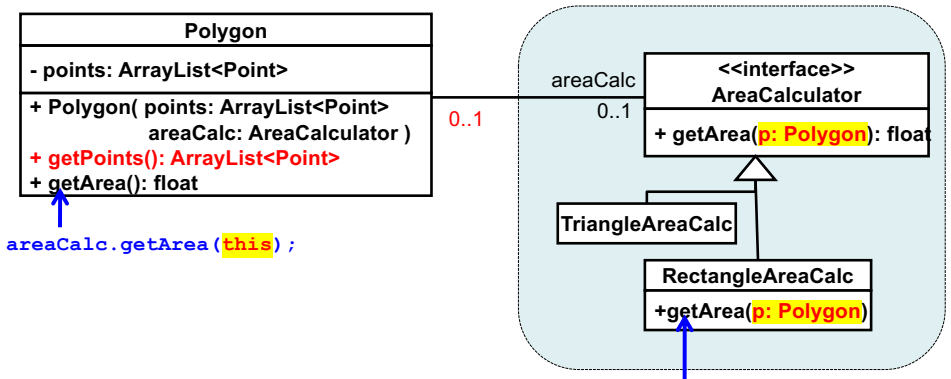


User/client of Polygon:

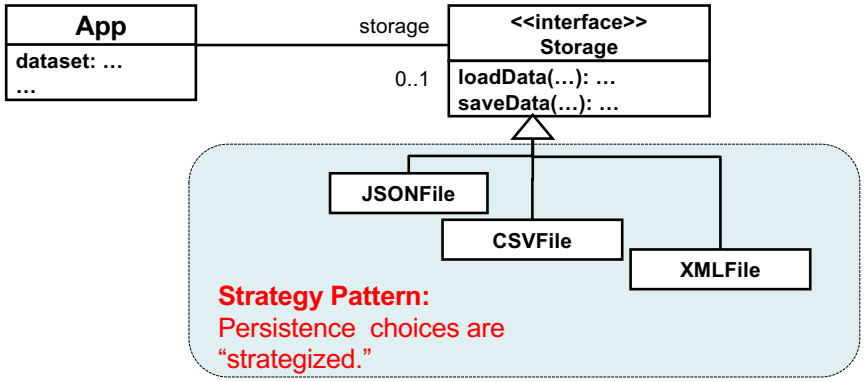
```
ArrayList<Point> al = new ArrayList<Point>();
al.add( new Point(…) ); al.add(...); al.add(...);

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

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

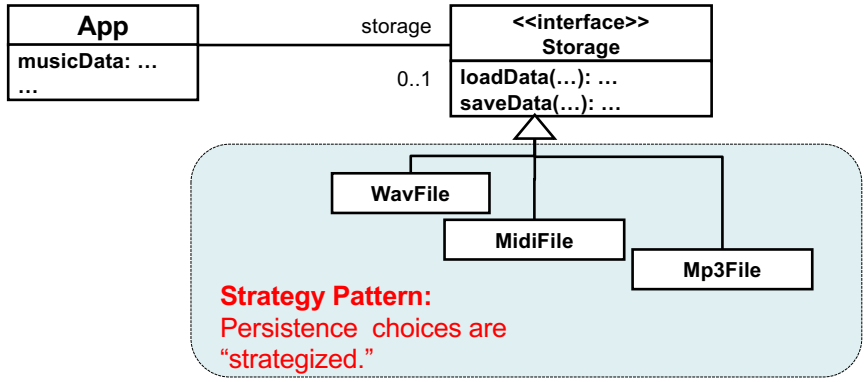
Dynamic polygon transformation. Dynamic replacement of area calculators

An Alternative

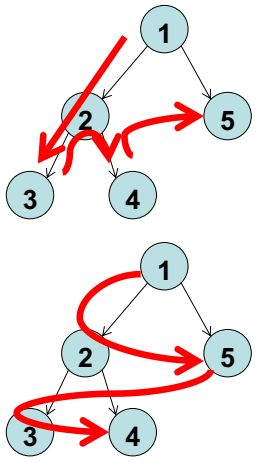


An Example of *Strategy* for Persistence



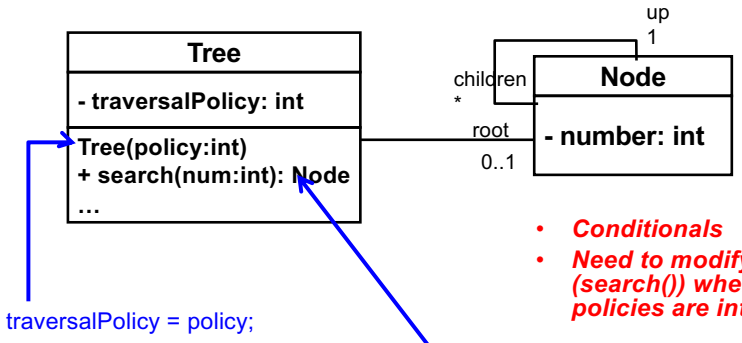


Tree Traversal with *Strategy*



- Tree traversal
  - Visiting all nodes in a tree one by one
  - Many applications:
    - AI engine for strategy games (e.g. tic-tac-toe, 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



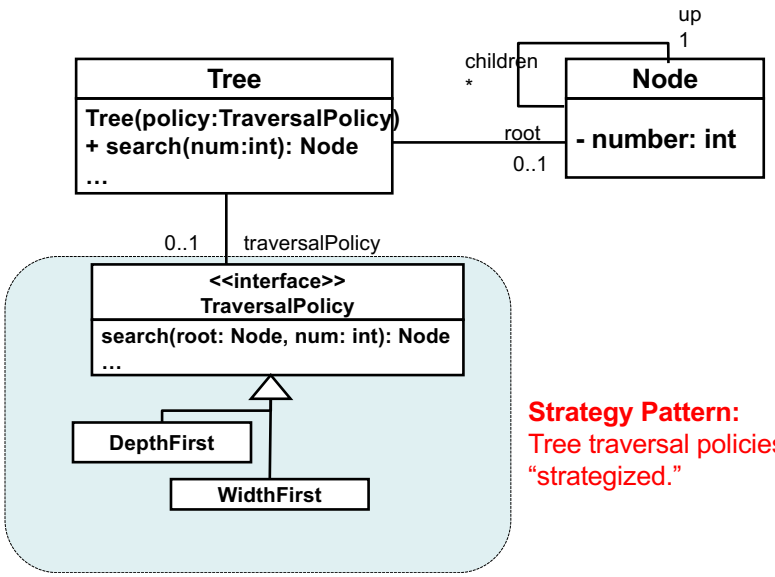
- ***Conditionals***
- ***Need to modify existing code (search()) when new traversal policies are introduced.***

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

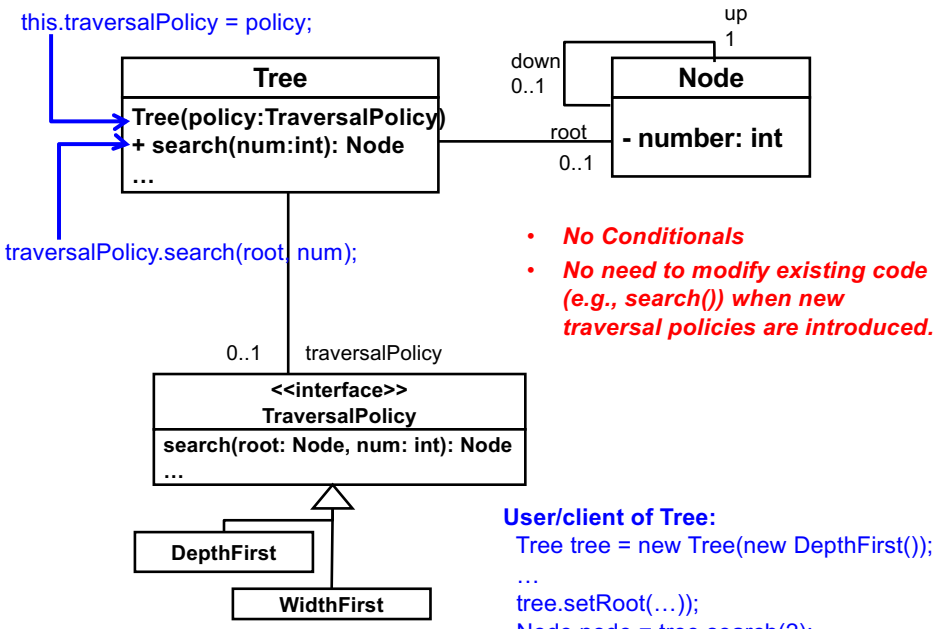
Suggested Read

- 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

With *Strategy* Classes…



Strategy Pattern:
Tree traversal policies are "strategized."



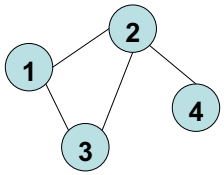
- ***No Conditionals***
- ***No need to modify existing code (e.g., search()) when new traversal policies are introduced.***

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

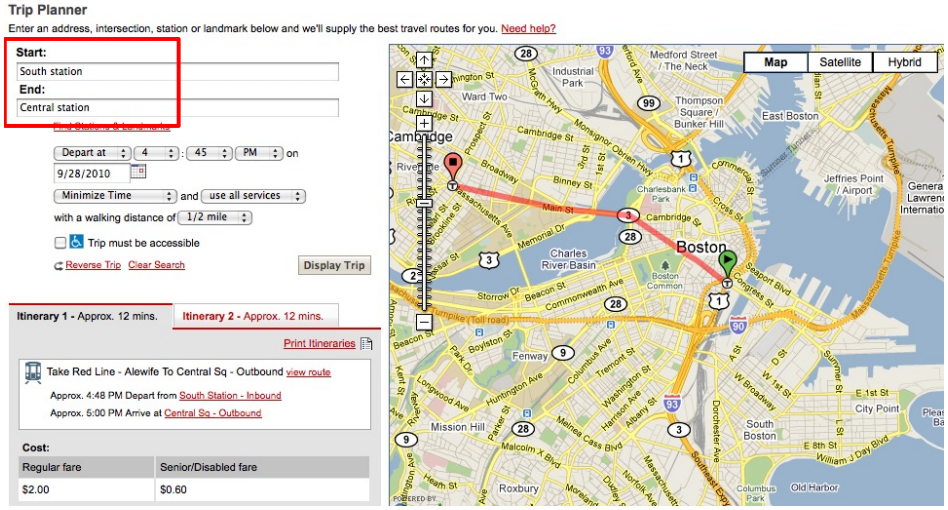


Graph
- graph: ArrayList<ArrayList<Boolean>>
+ Tree(graph:ArrayList<ArrayList<Boolean>>) + getShortestPath(n1:int, n2:int): ArrayList<Integer>

- 0 1 1 0
  1 0 1 1
  1 1 0 0
  0 1 0 0

Connectivity matrix

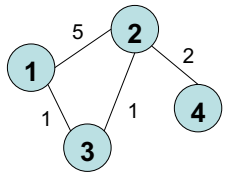
Trip Planner at mbta.org



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

Weighted Graphs

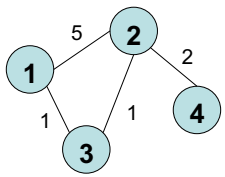
- What if you need to consider the *weighed* shortest path between two nodes?
  - 1 -> 3-> 2-> 4: total weight = 4, between Node 1 and Node 4
  - 1 -> 3 -> 2: total weight = 2, between Node 1 and Node 2



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

- 0 5 1 -1
5 0 1 2
1 1 0 -1
-1 2 -1 0

Weighted connectivity matrix



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

graph 1 — 0..1 shortestPathFinder

<<interface>> ShortestPathFinder
getShortestPath(graph:…, n1:int, n2:int): ArrayList<Integer>

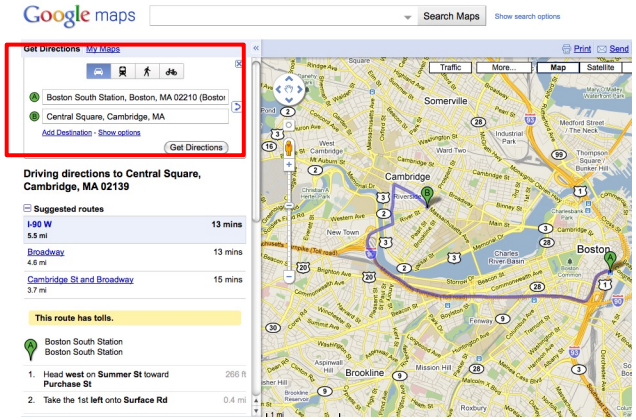
UnweigedShortestPathFinder

WeightedShortestPathFinder

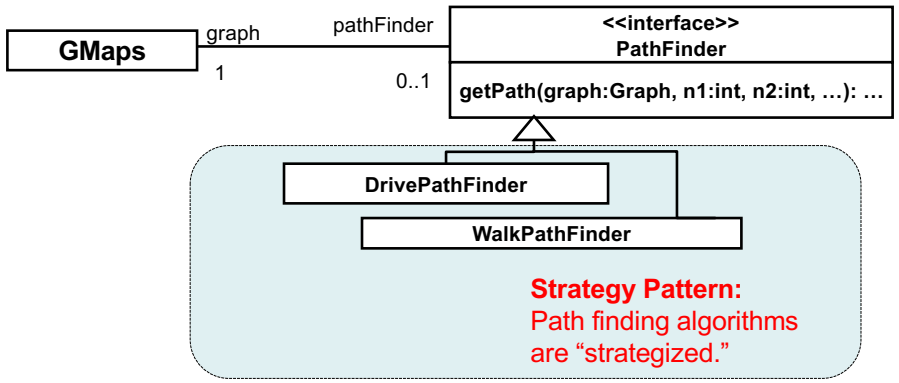
Strategy Pattern:
Shortest path algorithms are "strategized."

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

Google Maps

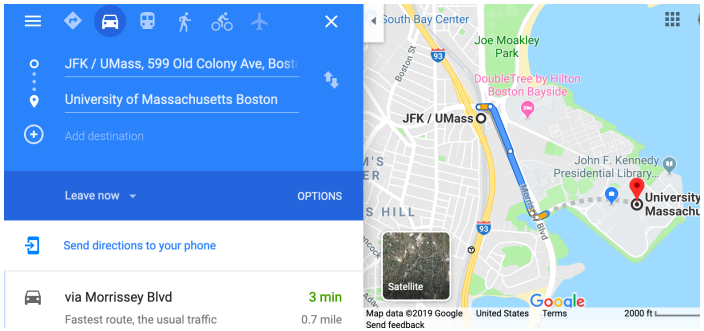


- Directions from one place to another
  - By car
  - By T, walk, bicycle and shared ride (e.g., Uber and Lyft).
  - By car, considering gas consumption.



Recap: an Example Scenario

- Your team is expected to develop a navigation app like G Maps.
  - For users to drive and walk (two navigation features)

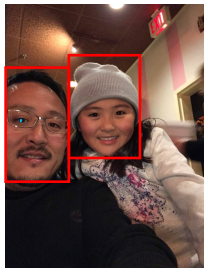


- How can two sub-groups of the team develop these two features *independently* (i.e. *in parallel*)?
  - How can those 2 features be implemented in a *loosely-coupled* manner?
    - NOT in a tightly-coupled manner.
    - To maximize productivity (development efficiency)
  - How can they be *integrated* in the end of the project in a cost-effective manner?
- How can *something common* be implemented in between the 2 features?
  - Basic data structures, algorithms and UI (e.g. maps, landmarks and shortest-path algorithms)

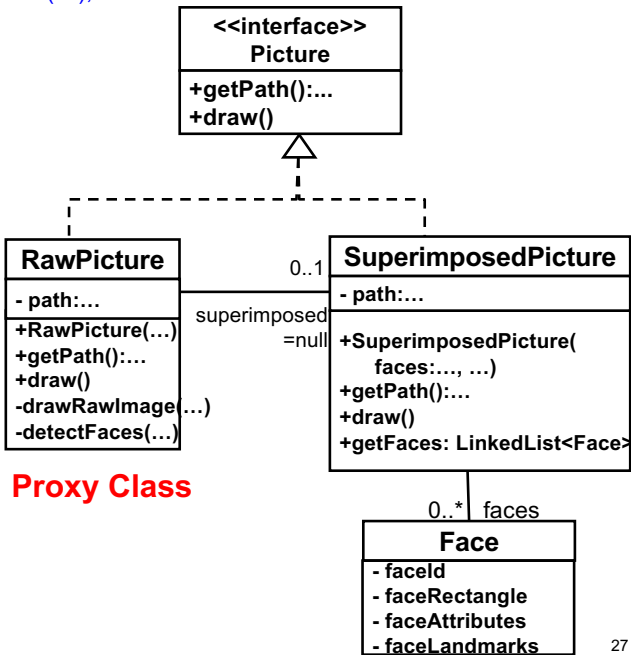
Face Detection in Pictures

- Suppose you are implementing an app to organize, edit and analyze pictures.
  - e.g., Photos from Apple

  - The app loads each raw picture and then superimposes a rectangle on a human face by (dynamically) calling an external face detection/recognition API.
    - e.g., Microsoft Azure Face API, Google Cloud Vision API



Client code (app):
RawPicture pic = new RawPicture(…);
pic.draw();

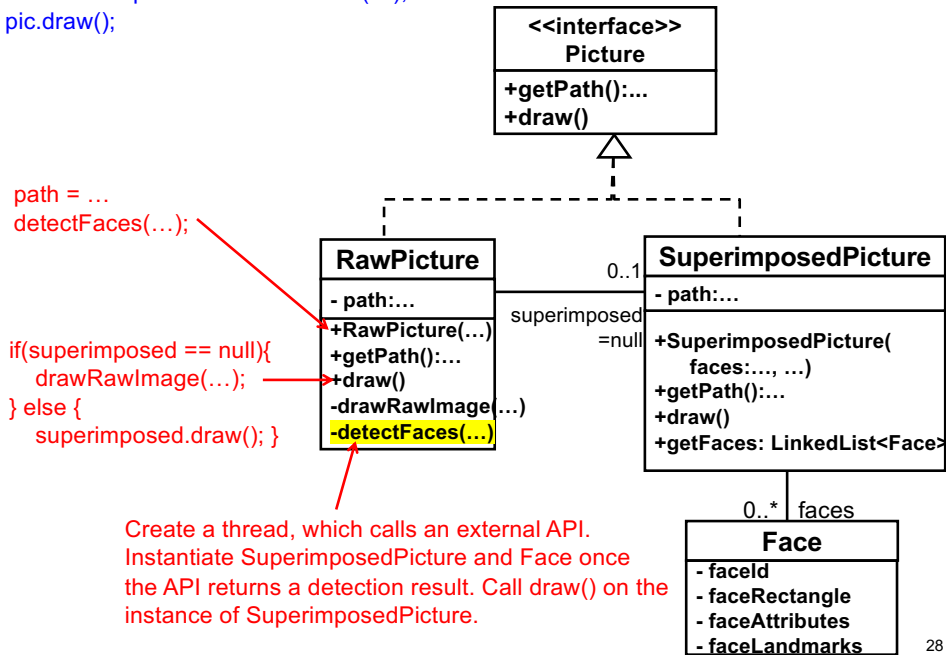


Proxy Class

- Some delay is expected to receive a face detection result from an external API.
  - The user is not patient enough to keep watching a blank app window until receiving a detection result.

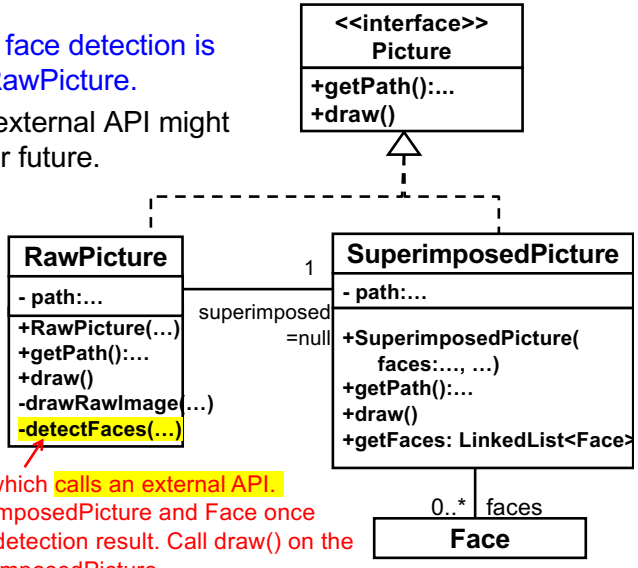
- Lazy loading of detection results
  - Show the user a raw picture first.
  - Call a face detection API in the background
  - Receive a detection result.
  - Replace the raw picture with a superimposed one, which contains a detection result.

Client code (app):
RawPicture pic = new RawPicture(…);
pic.draw();



Create a thread, which calls an external API. Instantiate SuperimposedPicture and Face once the API returns a detection result. Call draw() on the instance of SuperimposedPicture.

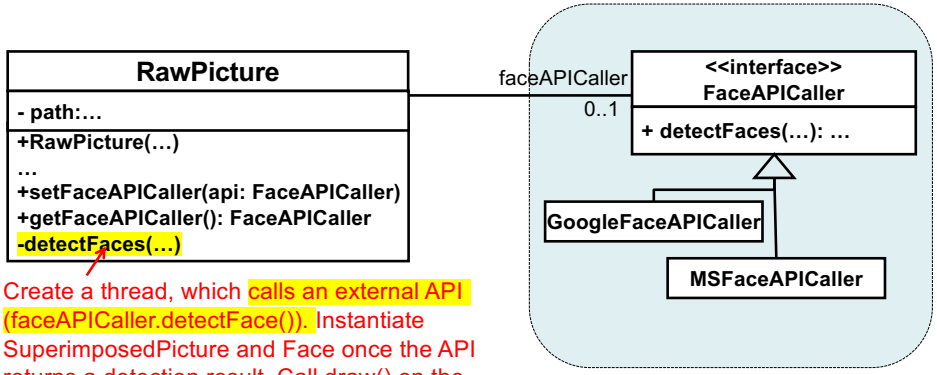
- Issue: An API call for face detection is tightly coupled with RawPicture.
  - The choice of an external API might change in the near future.



Create a thread, which calls an external API. Instantiate SuperimposedPicture and Face once the API returns a detection result. Call draw() on the instance of SuperimposedPicture.

Design Improvements

- An improvement with *Iterator*-inspired design
- Alternative improvement with *Strategy*



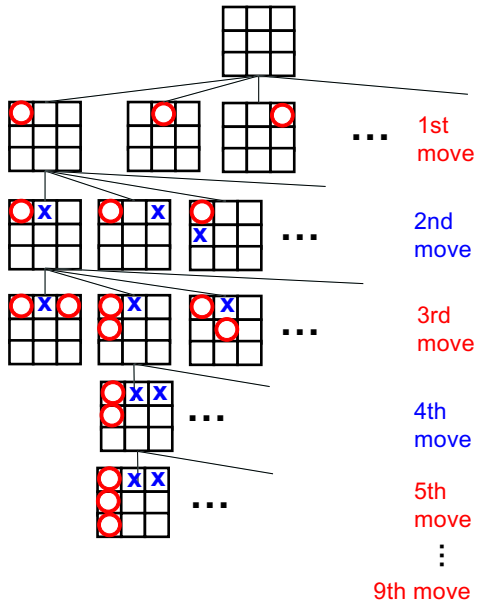
Create a thread, which calls an external API (faceAPICaller.detectFace()). Instantiate SuperimposedPicture and Face once the API returns a detection result. Call draw() on the instance of SuperimposedPicture.

Strategy Pattern:
Face API call is "strategized."

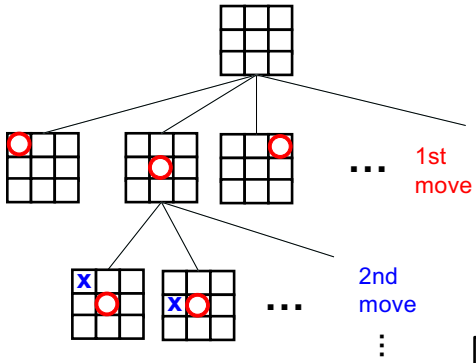
Exercise: Turn-based Strategy Games

- Players take turn in a game.
  - They never play simultaneously.
    - e.g., Tic-Tac-Toe, chess, resersi (othello), checkers, go, backgammon, monopoly, tactical (war) games, etc. etc.
  - In each turn, a player takes a game action(s).
    - e.g., moving a piece.
- A computer (or "AI") player
  - expresses a game as a sequence of game actions ("moves").
  - maintains possible sequences as a tree structure.

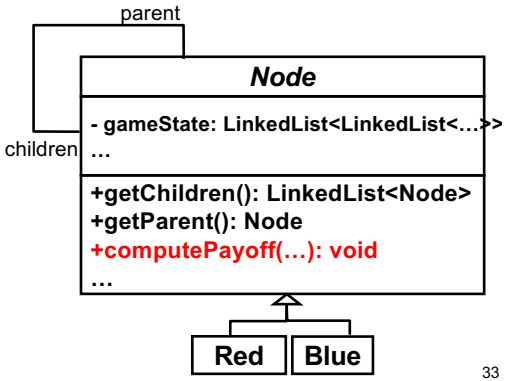
An Example Game: Tic-Tac-Toe

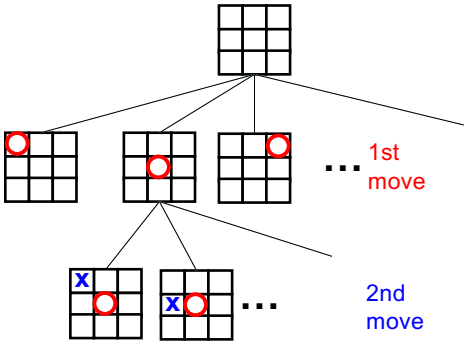


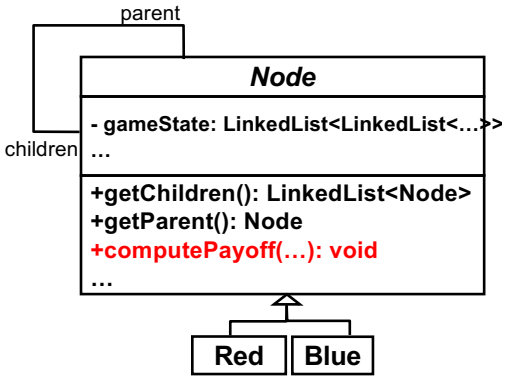
- Board size (# of positions/cells): 9
- Up to 9 moves in a game
  - 9 possible positions for the 1st move
  - 8 possible positions for the 2nd move
  - …
  - Up to 9! (362,880) root-to-leaf paths

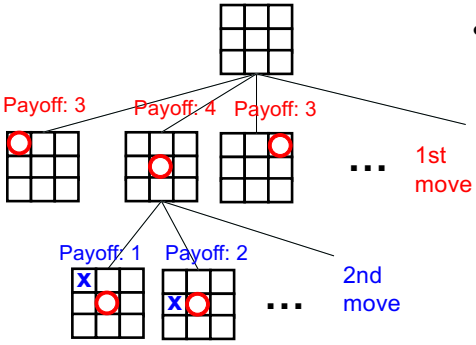


- Design a class diagram to express this kind of tree structures.
  - Define the **`Node`** class and its subclasses: **`Red`** and **`Blue`**.





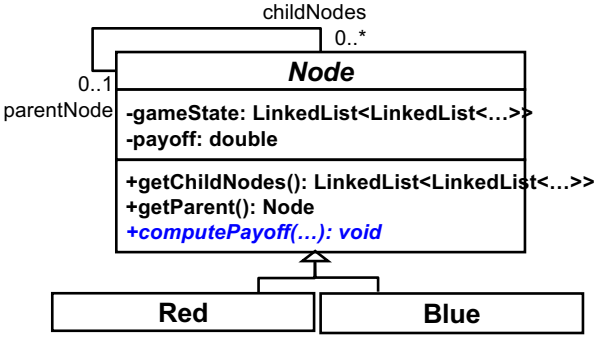




- Each node indicates
  - the current state of the game board.
    - Where Os are placed
    - Where Xs are placed
  - the payoff for a player.
    - "Goodness" of the current game state for the player.
    - The higher, the better, for the red (O) player
    - The lower, the better, for the blue (X) player.
- Payoff is computed with a certain evaluation logic.

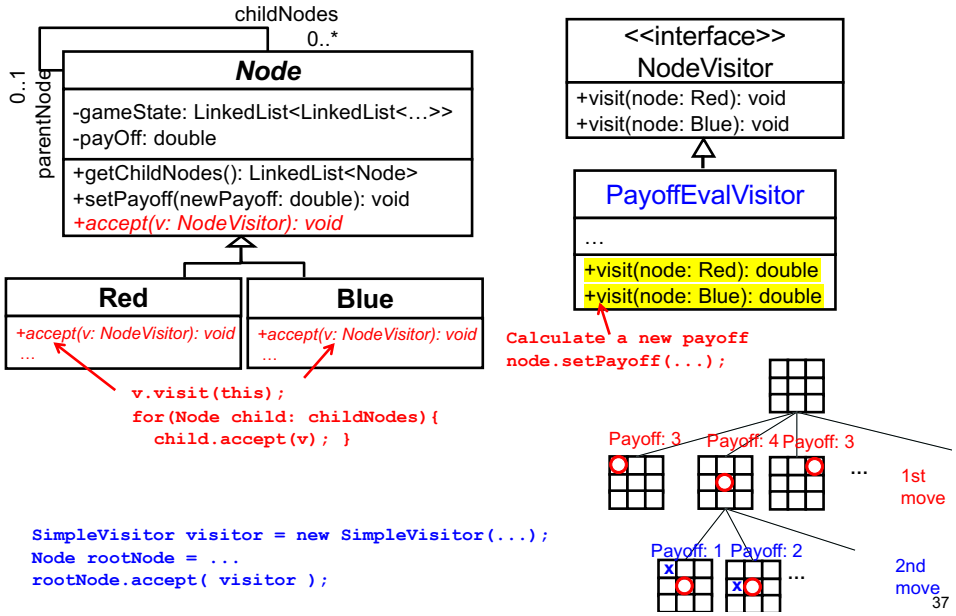
- The red (O) player plays to maximize its payoff.
- The blue (X) player plays to minimize its payoff.

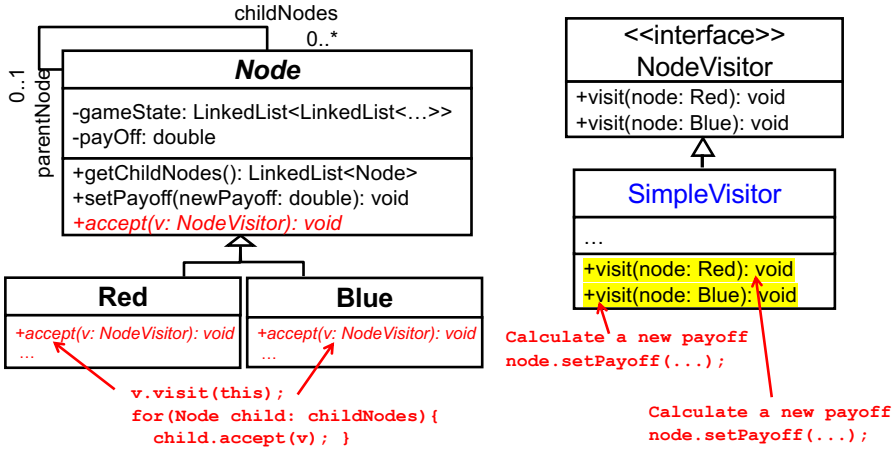
Design Issue: Payoff Evaluation Logic Changes VERY Often



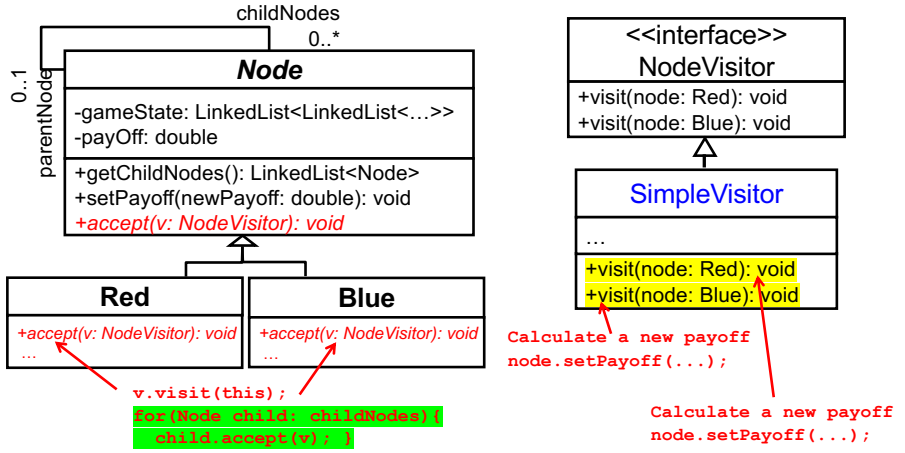
- It doesn't make sense to modify `Red`'s and `Blue`'s `computePayOff()`.
- It is not that good either to subclass `Red` and `Blue`
  - You would end up with too many subclasses.
  - They need to take care of other concerns, not only payoff evaluation logic.
    - e.g. tree traversal logic, multi-threading, time mgt, etc.

Separating Evaluation Logic with *Visitor*





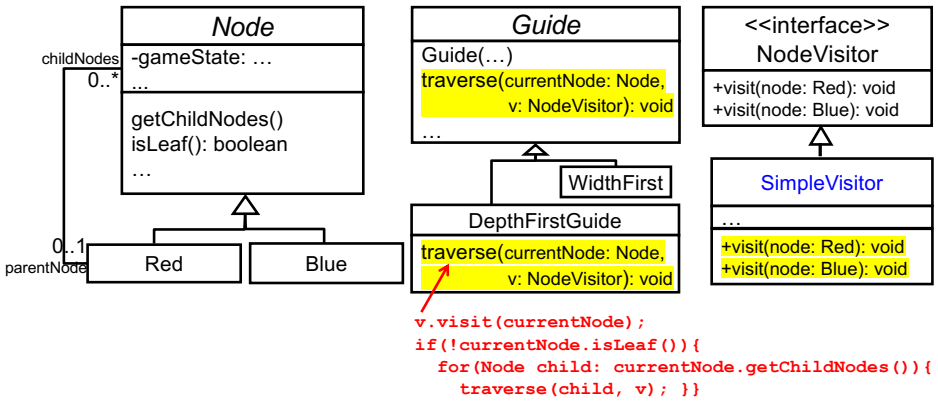
- This design separates a set of foundation objects and the operations to be performed on those objects.
  - You can implement the operations (i.e. evaluation operations) in a pluggable way (i.e. without changing foundation objects)



- Notice that this design does NOT allow you to implement tree traversal algorithms in a pluggable way.
  - A depth-first traversal algorithm is tightly-coupled with foundation objects (i.e. hard-coded in `accept()`).

Towards a More Flexible Design

- Is it possible to separate
  - Foundation objects,
  - Tree traversal algorithms, and
  - Evaluation operations,
  - so that you can implement BOTH traversal algorithms and evaluation operations in a pluggable manner?
- Use *Strategy* and *Visitor* at the same time.



- Each "guide" specializes in a specific traversal algorithm. It guides (or escorts) a visitor to visit nodes in a tree based on that algorithm.
  - `SimpleVisitor visitor = new SimpleVisitor(...);`
    `Node rootNode = ... ;`
    `Guide guide = new DepthFirstGuide(...);`
    `guide.traverse( rootNode, visitor );`
- If `traverse()` is defined as a static method…
  - `SimpleVisitor visitor = new SimpleVisitor(...);`
    `Node rootNode = ... ;`
    `DepthFirstGuide.traverse( rootNode, visitor );`

Comparators in Java API

- Sorting array elements:
  - `int years[] = {2020, 2010, 2000, 1990};`

    `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.
    - 1990 -> 2000 -> 2010 -> 2020

- Sorting collection elements:
  - `ArrayList<Integer> years = Arrays.asList(`
    `2020, 2010, 2000, 1990);`

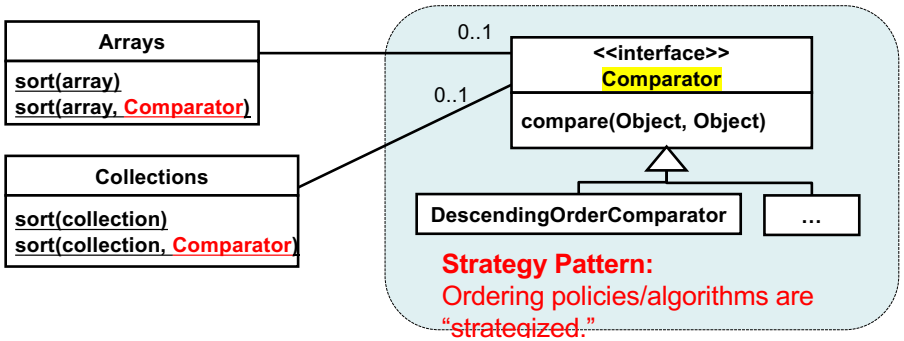
    `Collections.sort(years);`
    `for(Integer y: years)`
    `System.out.println(y);`
  - `java.util.Collections`: a utility class (a collection of static methods) to process collections and collection elements
  - `sort()` sorts collection elements in an ascending order.
    - 1990 -> 2000 -> 2010 -> 2020

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 = Arrays.asList(
                                    2020, 2010, 2000, 1990);
Collections.sort(years);
for(Integer y: years)
    System.out.println(y);

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

- 1990 -> 2000 -> 2010 -> 2020
- 2020 -> 2010 -> 2000 -> 1990

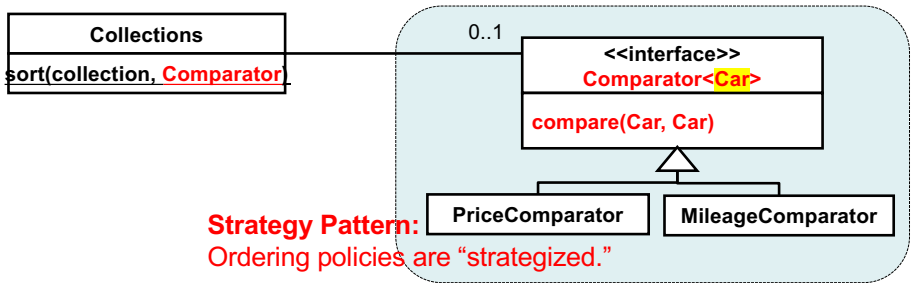
Type-safe Comparators

```
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
```

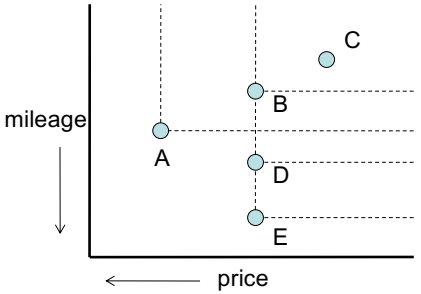

Used Car Listings

HW 11

- 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.

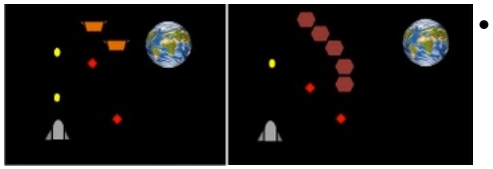


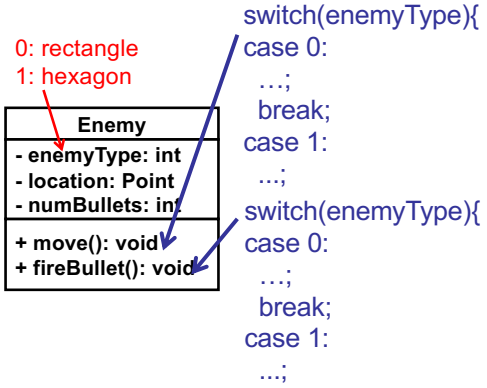
  - Count the number of cars that dominate each car.
    - A: 0 (No cars dominate A.)
    - B: 3 (A, D, E)
    - C: 4 (A, B, D, E)
    - D: 1 (E)
    - E: 0 (No cars dominate E.)
  - Better cars have lower "domination counts."
    - To order cars from the best one(s) to the worst one(s), compare() should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

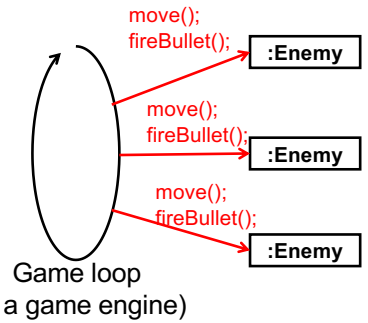
```
// Set domination counts for all cars by calling
// setDominationCount() on those cars, and
// then call sort()
for(car: usedCars){
  car.setDominationCount(...); }
Collections.sort(usedCars, new ParetoComparator<Car>());
```


One More Exercise
Imagine a Simple 2D Shooting Game



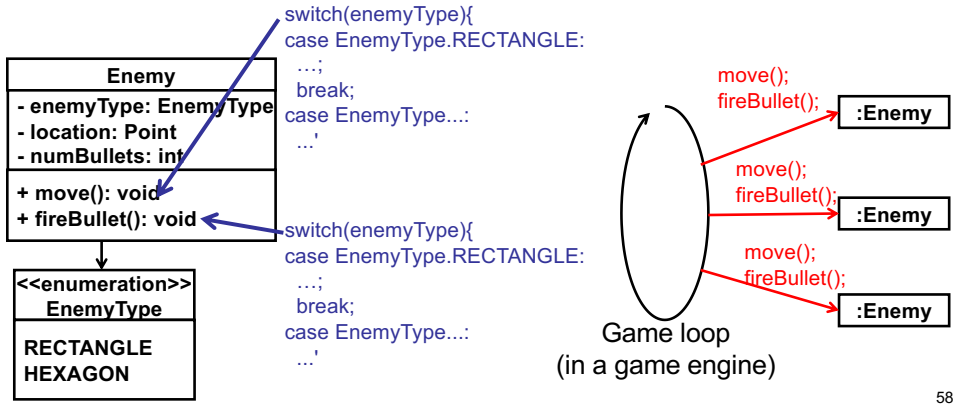


- Each type of enemies has its own attack pattern.
  - e.g. How to move, when to fire bullets, how to fire bullets, etc.



What's Bad?

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



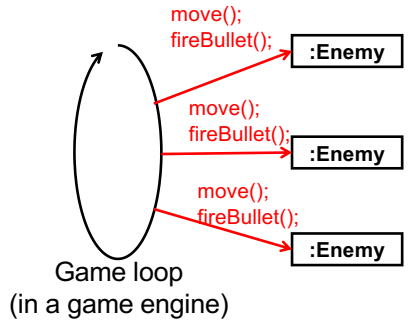
Still Not Good

- 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…

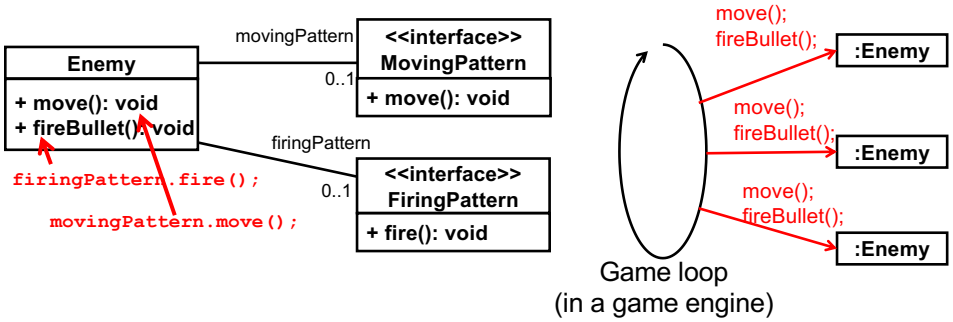
- Eliminate those conditional branches.
- Separate Enemy and its attack patterns (moving patterns and firing patterns).
  - Make Enemy and its attack patterns *loosely coupled*.
- Define a family of attack patterns (algorithms) in a unified way
- Encapsulate each algorithm in a class
- Make algorithms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

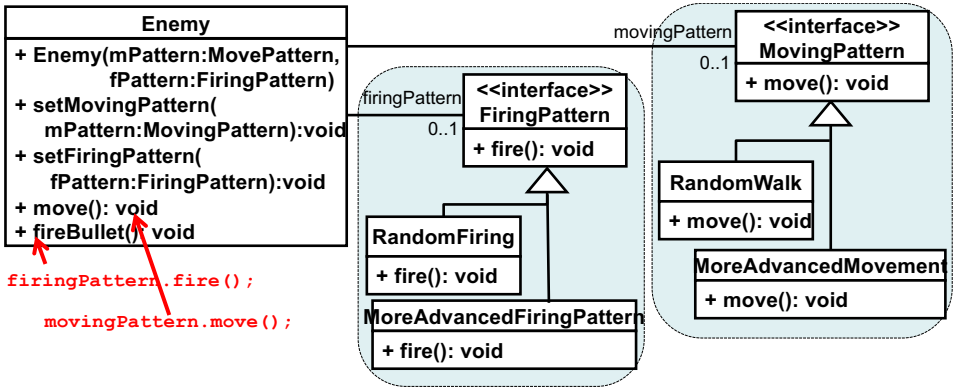


Suggested Read

- 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

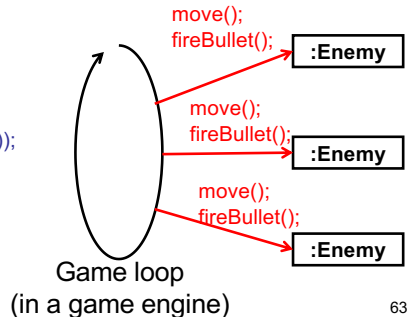
Revised Design with *Strategy*





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(); }
```

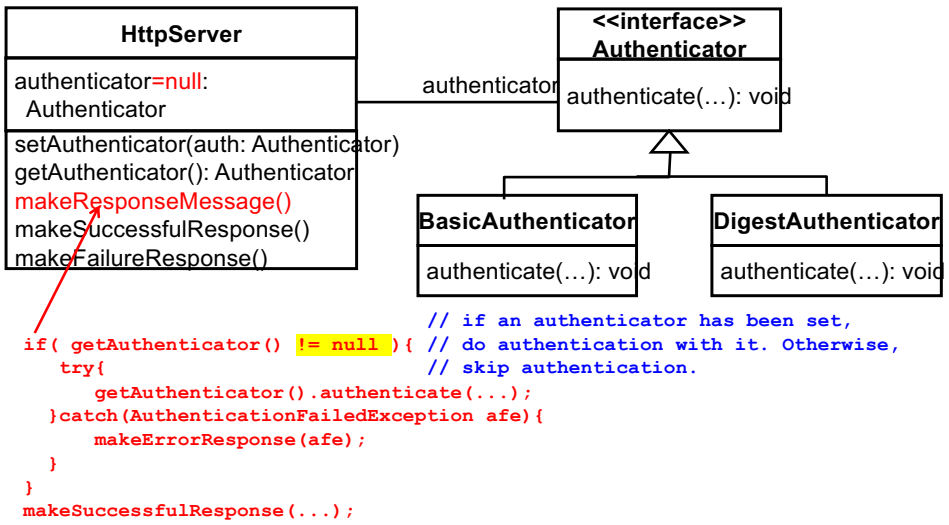


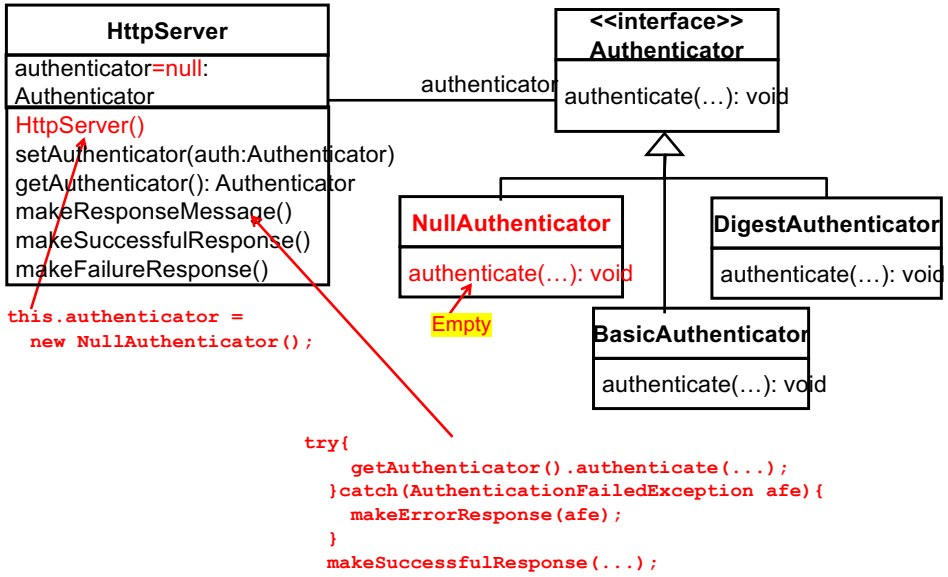
Null Object Design Pattern

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





Null Object as *Strategy*

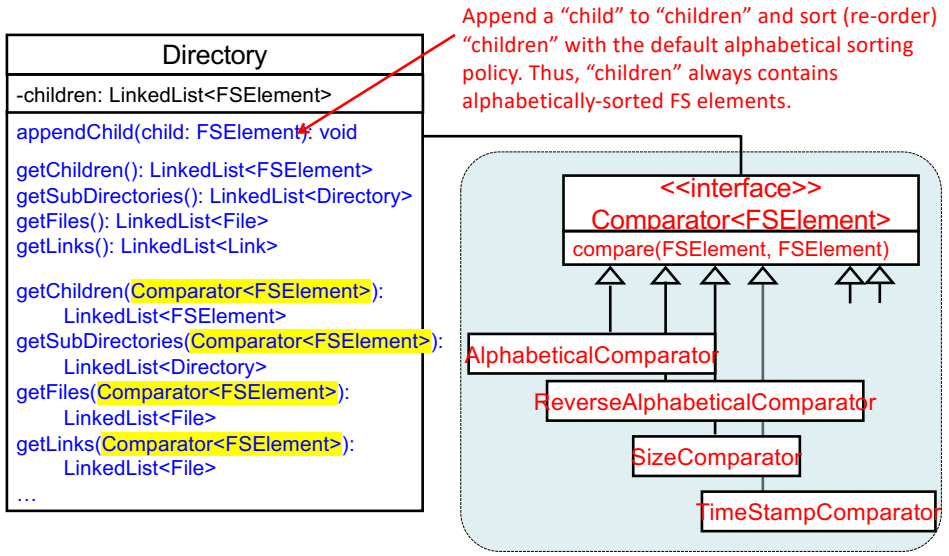
- Null object
  - A variant/application of *Strategy* that focuses on "doing nothing" by default.

HW 12: Sorting FS Elements

- In prior HWs, FS elements were never sorted.
- Let's have the file system sort FS elements.
- Example sorting policies
  - Alphabetical
  - Reverse alphabetical
  - Size-based
  - Timestamp-based (e.g. "creation time"-based)
  - Element kind based (e.g. directories listed first followed by files and links)

Soring FS Elements with `Comparator`

- It is not a good idea to hardcode sorting policies in `Directory`.
  - Whenever a new sorting policy is required, FS developers have to modify the internals of `Directory`.
- Better idea: Separate `Directory` and sorting policies
  - Allow FS developers to add (or plug-in) new sorting policies.
    - Have them add extra code (classes) rather than modifying code (i.e., `Directory`).
  - Allow each FS user to choose his/her favorite sorting policies dynamically
- Solution: Use *Strategy*
  - Use `Comparator` and `Collections.sort()`.



Strategy design pattern

- `getChildren()`
  `getSubDirectories()`
  `getFiles()`
  `getLinks()`
  - Returns a `LinkedList` whose elements are sorted with the default alphabetical sorting policy.
- `getChildren(Comparator<FSElement>)`
  `getSubDirectories(Comparator<FSElement>)`
  `getFiles(Comparator<FSElement>)`
  `getLinks(Comparator<FSElement>)`
  - Re-orders FS elements based on a custom (non-default) sorting policy, which is indicated by the method parameter, and returns re-ordered FS elements.
    - Use `Collections.sort(List, Comparator<FSElement>)`.
    - `Directory` does not have to retain the re-ordered elements.
    - Implement at least 3 custom sorting policies