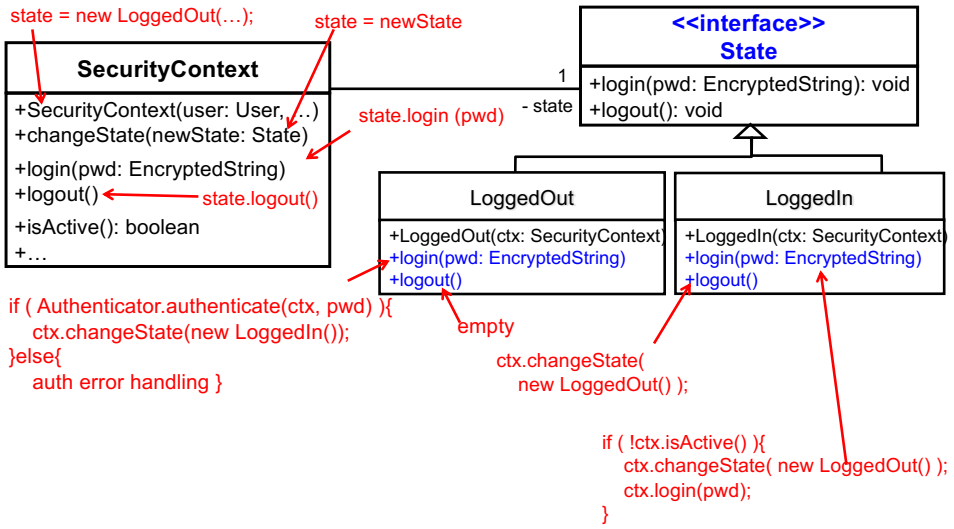
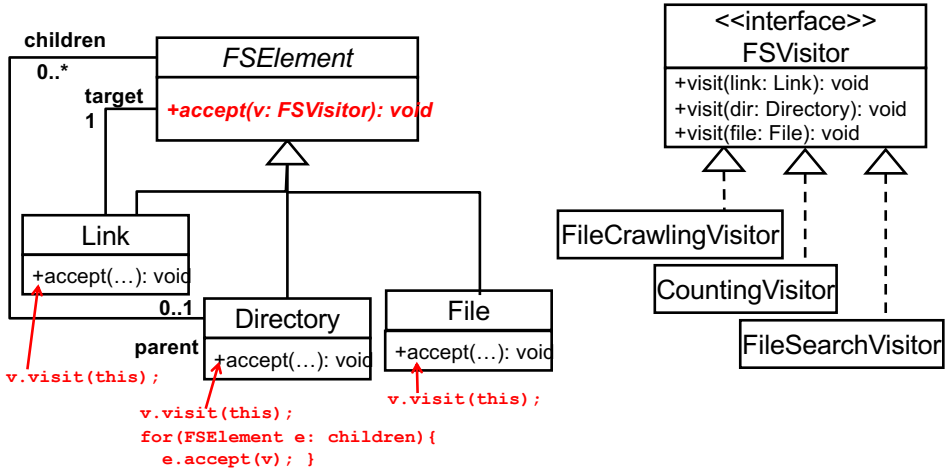
Recap: HW 5

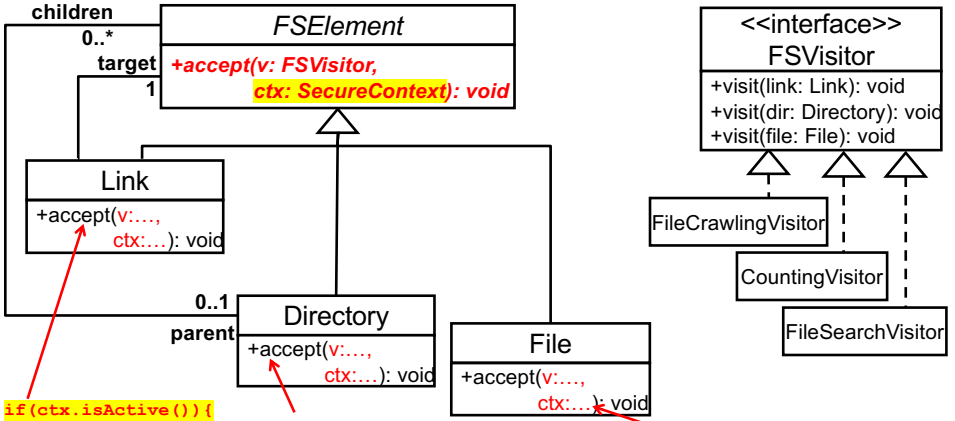


Recap: HW 9



HW 10: Implement Secure Visitors

- Goal: Authenticated traversal of FS elements
  - Have a user login and then pass a visitor to FS elements.
  - Deny access to FS elements if not logged in.
  - Combine your HW 5 and 10 solutions.




```
SecurityContext ctx = new SecurityContext(...);
ctx.login(...);
FileCrawlingVisitor visitor = new FileCrawlingVisitor();
rootDir.accept(visitor);
visitor.getFile();
```

- It is up to you how to implement the “else” block.

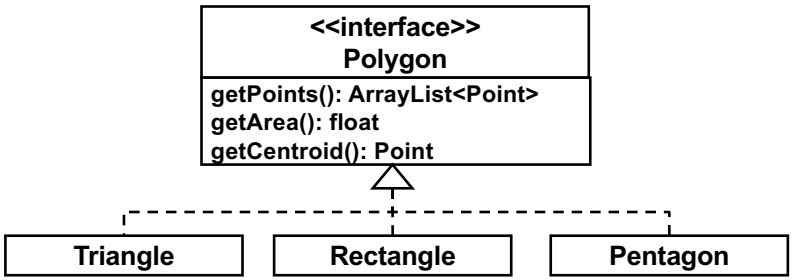
```
if(ctx.isActive()){
  v.visit(this);
}else{
  stop the on-going traversal & print an error msg.
  Or, have the user login again & continue the traversal.
}
```

  - How to stop the on-going traversal?
    - Return accept()?
    - Throw an exception?
  - How to have the user login again?

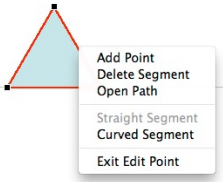
Strategy Design Pattern

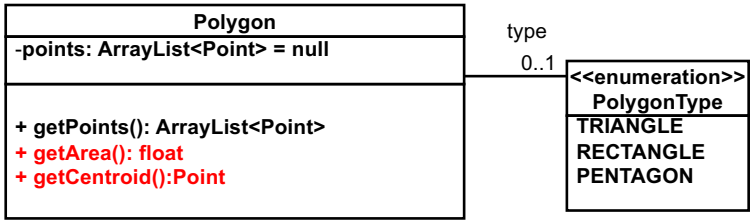
- Intent
  - Implement a family of algorithms
  - Encapsulate each algorithm in a class
  - Separate one algorithm from another
    - Make those algorithms pluggable/interchangeable.

Recap



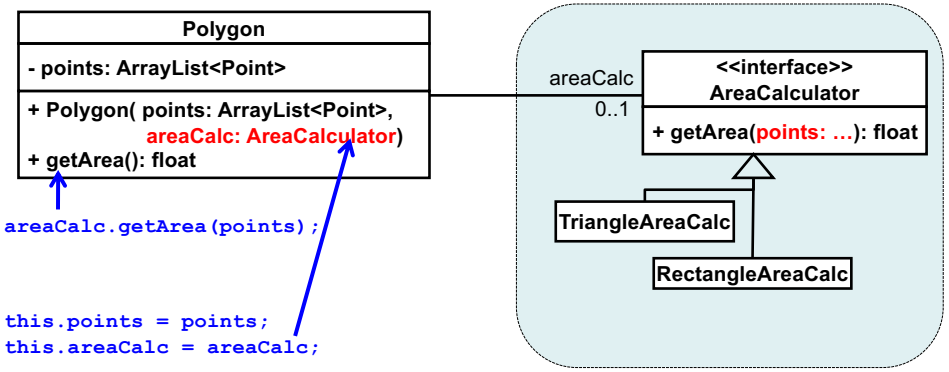
- Can a triangle become a rectangle dynamically?
- If we allow that, eliminate class inheritance





- Need to expect conditionals

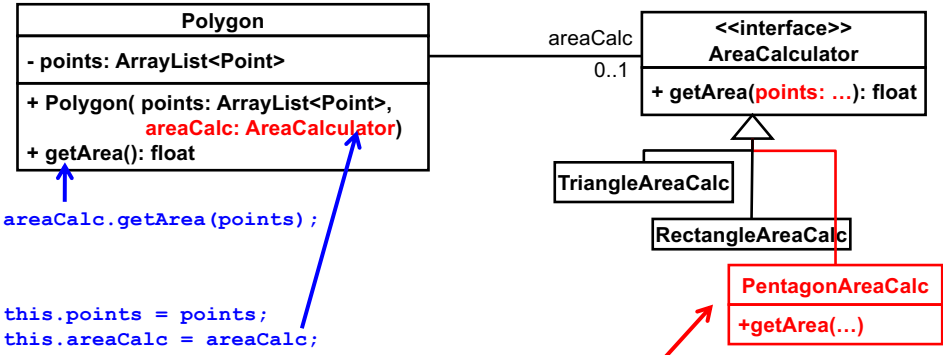
An Example of *Strategy*



Strategy Pattern:
Area calculation is "strategized."

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();
```

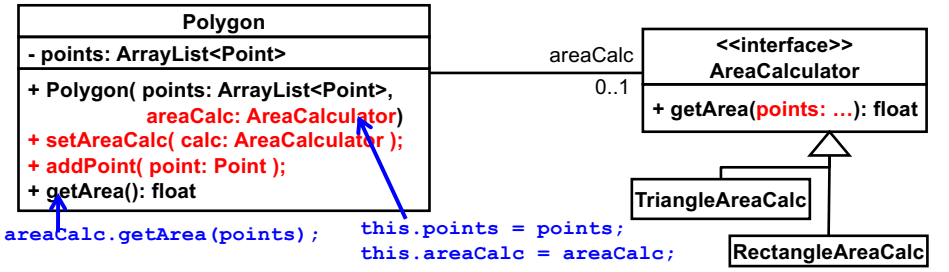



A new area calculator NEVER alter existing code.

User/client of Polygon:

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

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


Polygon Transformation

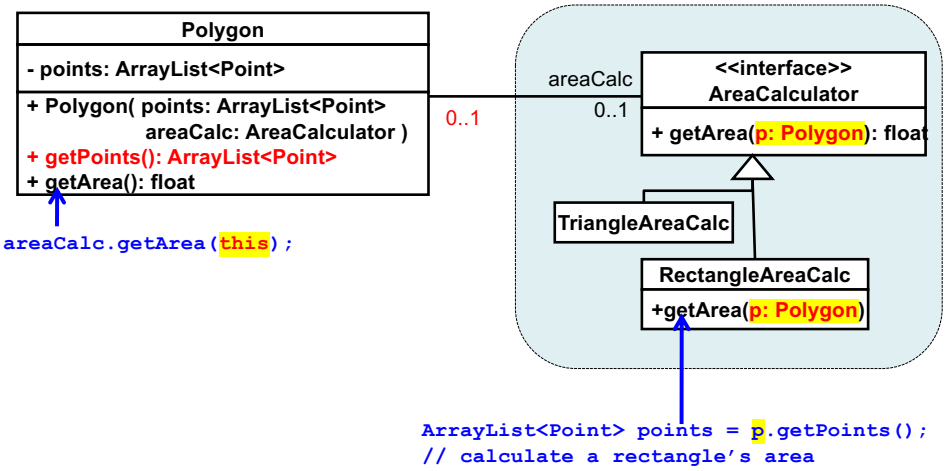


User/client of Polygon:

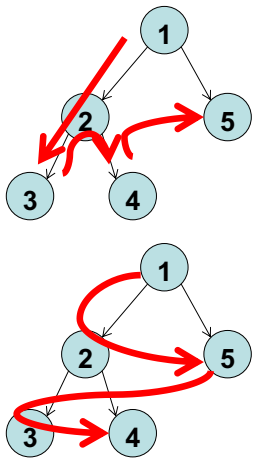
```
ArrayList<Point> al = new ArrayList<Point>();
al.add( new Point(…) ); al.add(...); al.add(...);

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
```

Dynamic polygon transformation. Dynamic replacement of area calculators

An Alternative

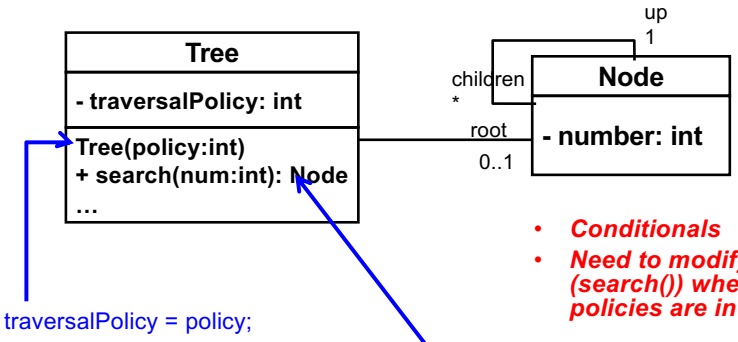


Tree Traversal with *Strategy*



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



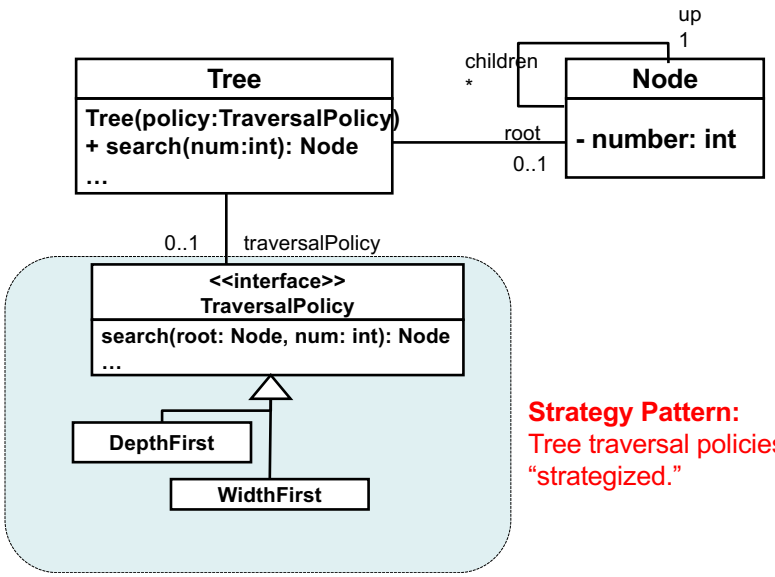
- ***Conditionals***
- ***Need to modify existing code (search()) when new traversal policies are introduced.***

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

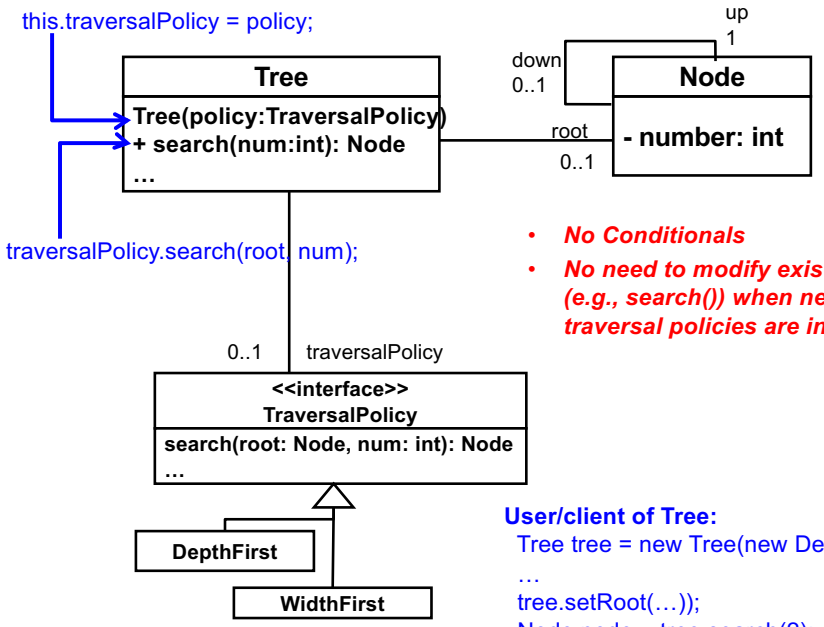

Suggested Read

- 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."



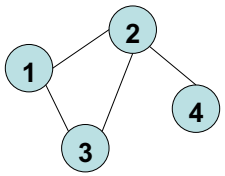
- ***No Conditionals***
- ***No need to modify existing code (e.g., search()) when new traversal policies are introduced.***

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

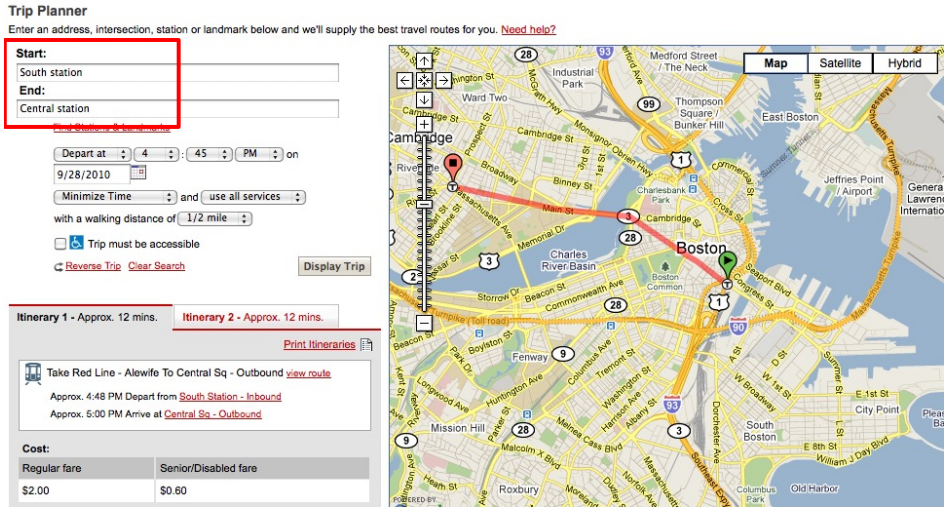


Graph
- graph: ArrayList<ArrayList<Boolean>>
+ Tree(graph:ArrayList<ArrayList<Boolean>>) + getShortestPath(n1:int, n2:int): ArrayList<Integer>

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

  Connectivity matrix

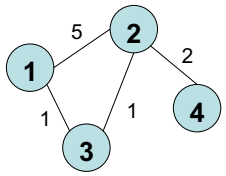
Trip Planner at mbta.org



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

Weighted Graphs

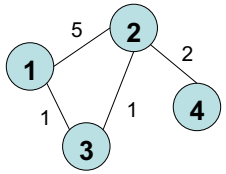
- What if you need to consider the *weighed* shortest path between two nodes?
  - 1 -> 3-> 2-> 4: total weight = 4, between Node 1 and Node 4
  - 1 -> 3 -> 2: total weight = 2, between Node 1 and Node 2



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>

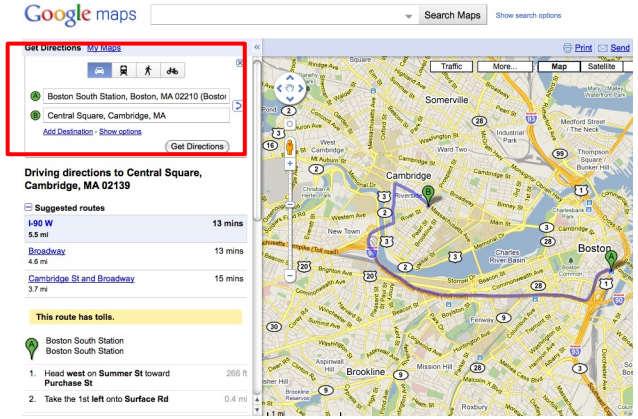
UnweigedShortestPathFinder

WeightedShortestPathFinder

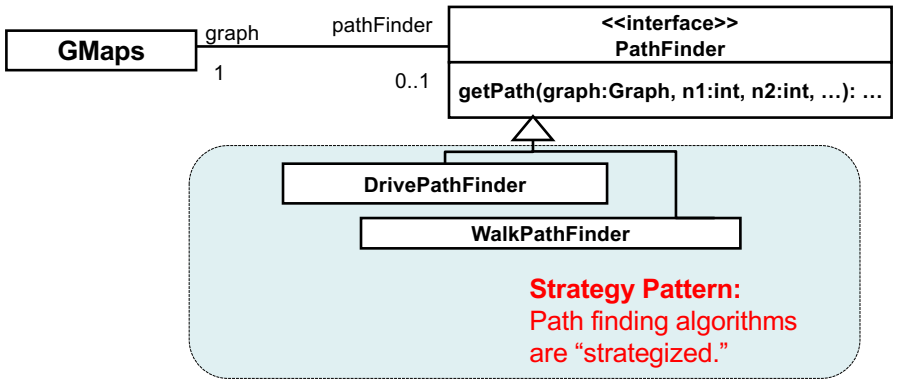
Strategy Pattern:
Shortest path algorithms are "strategized."

- ***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

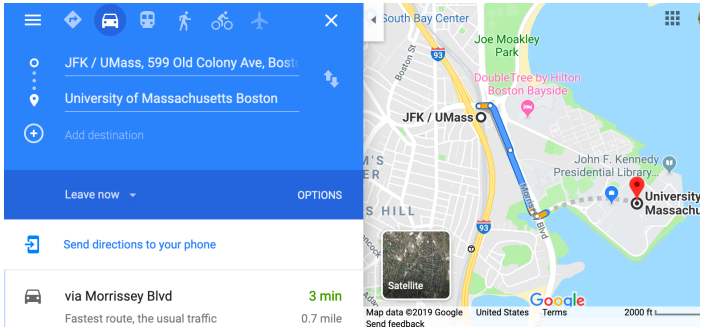


- Directions from one place to another
  - By car
  - By T, walk, bicycle and shared ride (e.g., Uber and Lyft).
  - By car, considering gas consumption.



Recap: an Example Scenario

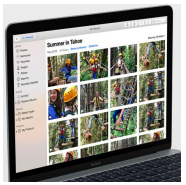
- Your team is expected to develop a navigation app like G Maps.
  - For users to drive and walk (two navigation features)



- 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

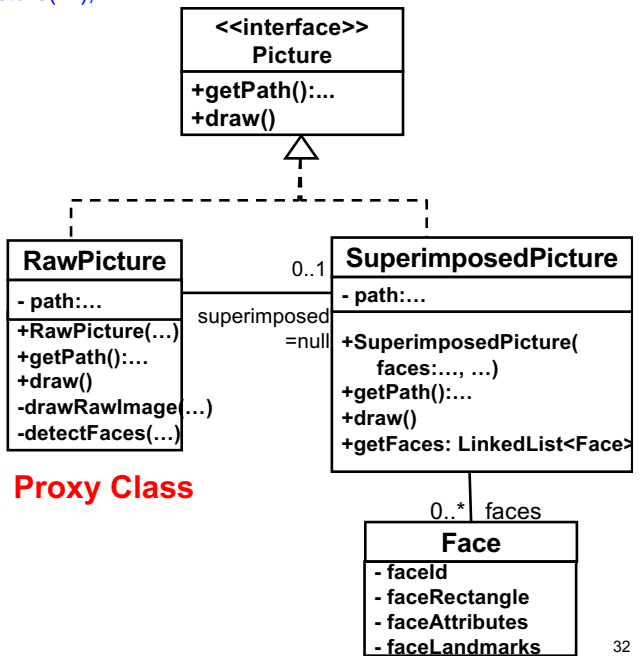
- Suppose you are implementing an app to organize, edit and analyze pictures.
  - e.g., Photos from Apple



  - 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

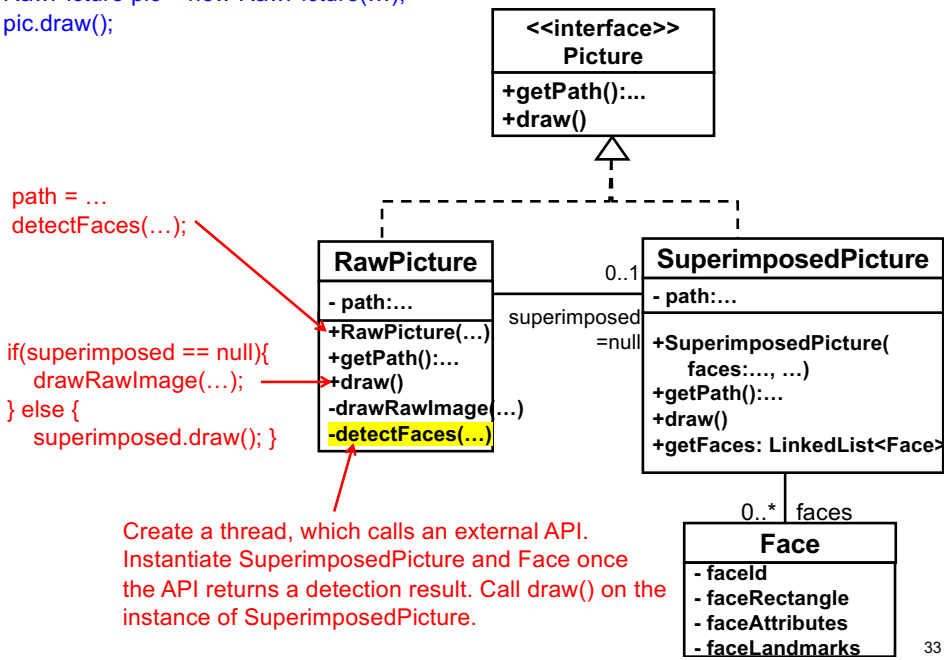
- Some delay is expected to receive a face detection result from an external API.
  - The user is not patient enough to keep watching a blank app window until receiving a detection result.
- Lazy loading of detection results
  - Show the user a raw picture first.
  - Call a face detection API in the 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();

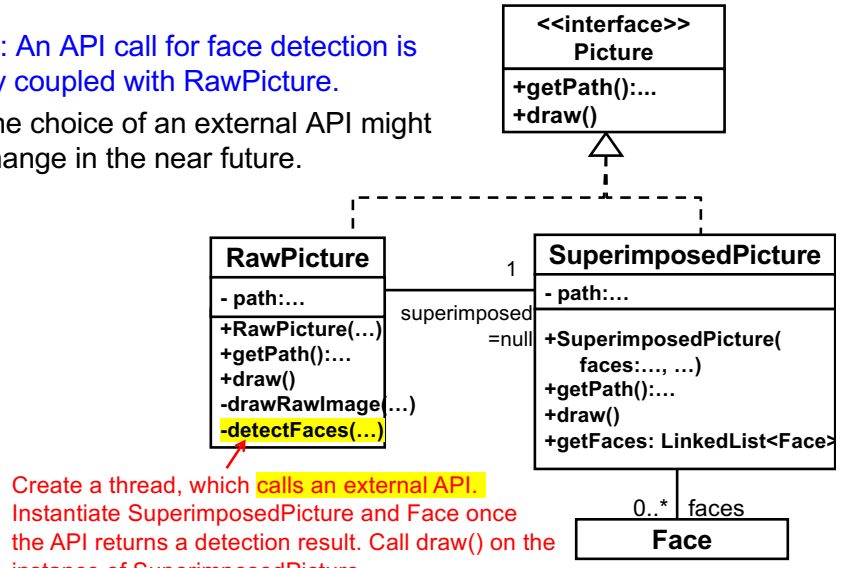


Proxy Class

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

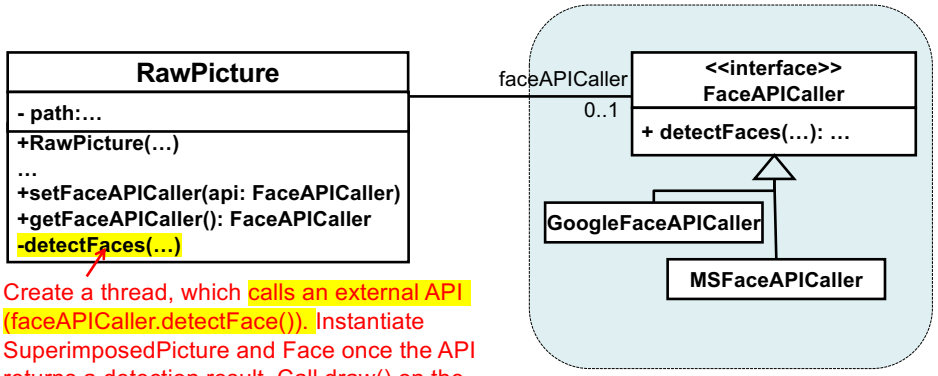


- Issue: An API call for face detection is tightly coupled with RawPicture.
  - The choice of an external API might change in the near future.



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."