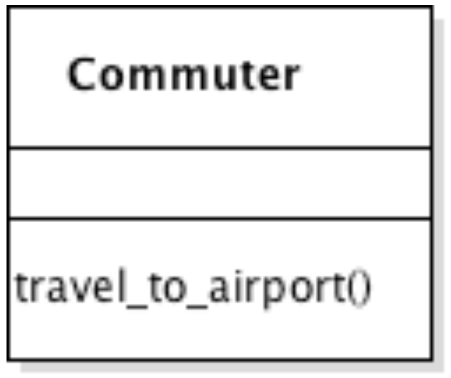
Strategy

- Intent
  - Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it
- So consider using Strategy if a class should have multiple ways of performing a similar task

Simple Commuter Example

- A commuter needs to travel to the airport:



- They might travel by car, bus or taxi.
- How best to model this?

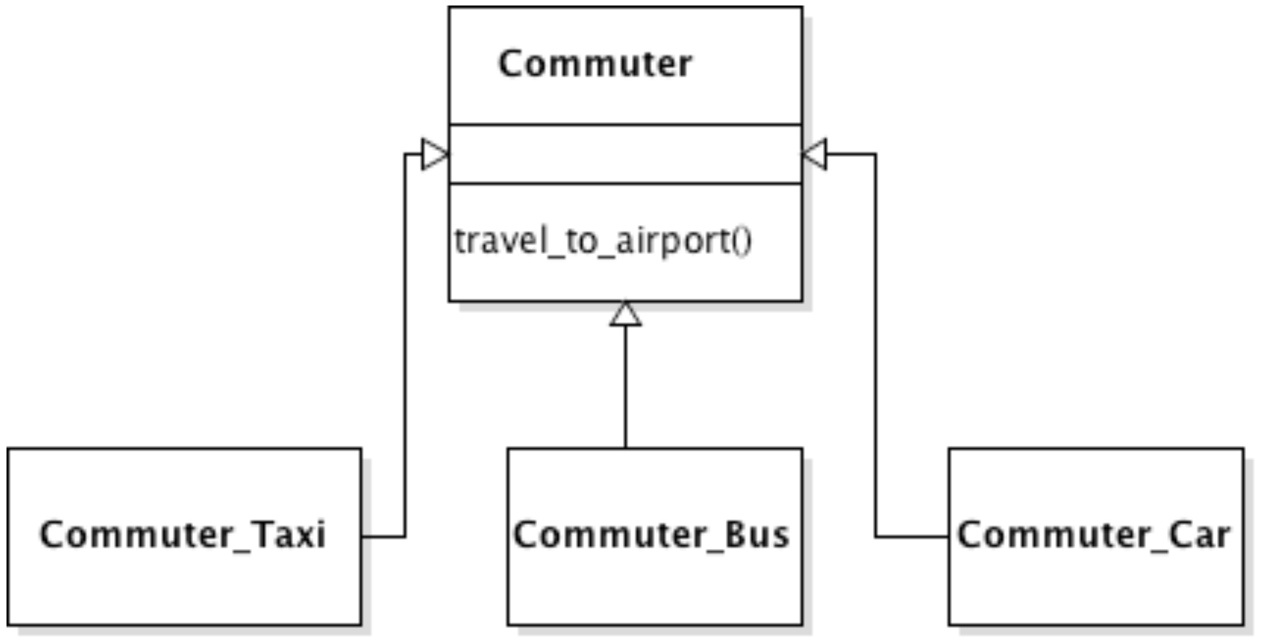
Tell the method what to do — an ugly solution

```
void travel_to_airport(string mode){
  if (mode.equals(“car”)){
    ...
  }
  if (mode.equals(“bus”)){
    ...
  }
  if (mode.equals(“taxi”)){
    ...
  }
}
```

In OOP, a switch statement is a code smell

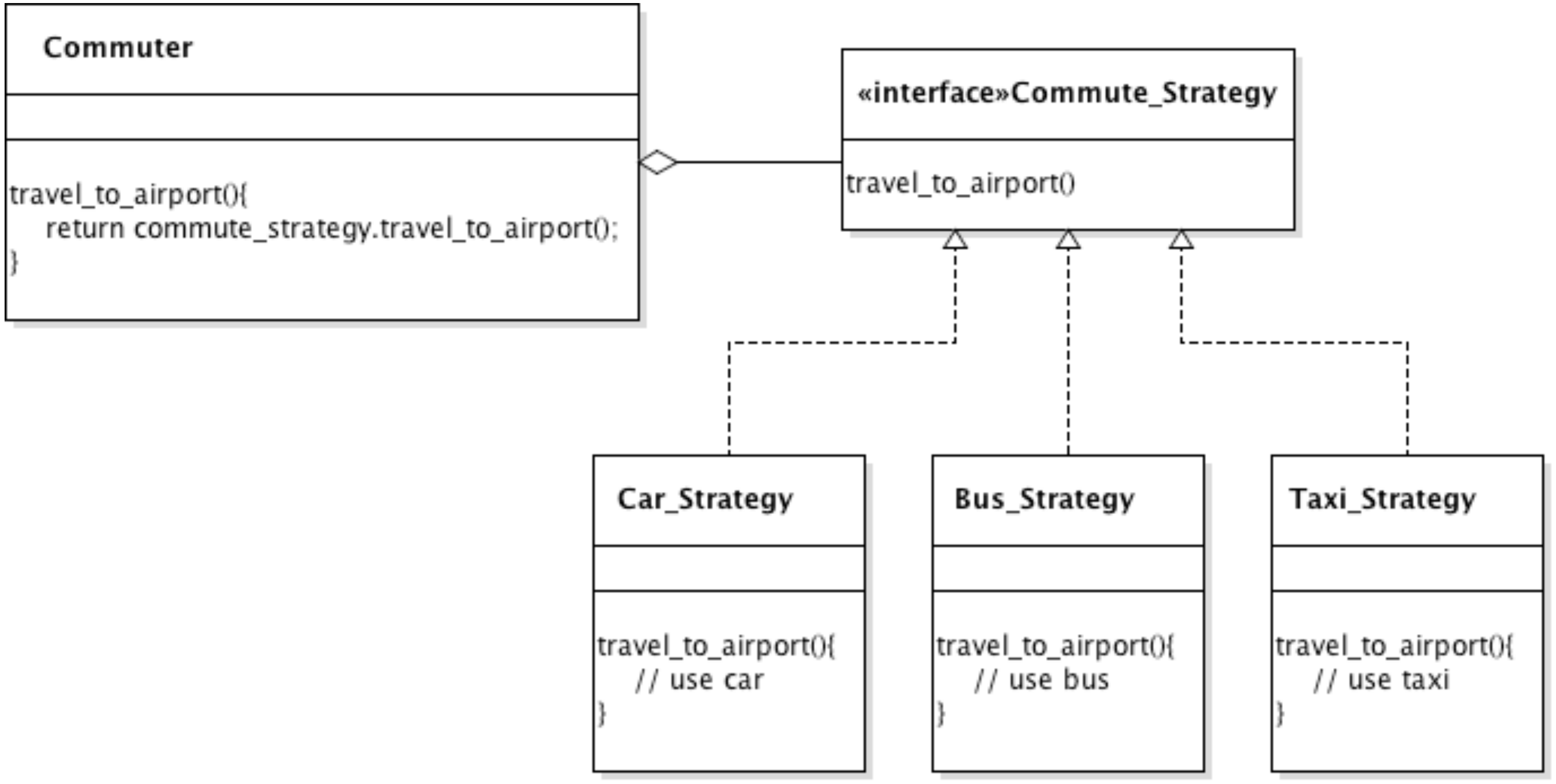
- Method has three unrelated parts
- Even worse if other methods in the class employ type checking on the `mode` variable as well

Using inheritance — not a good idea either



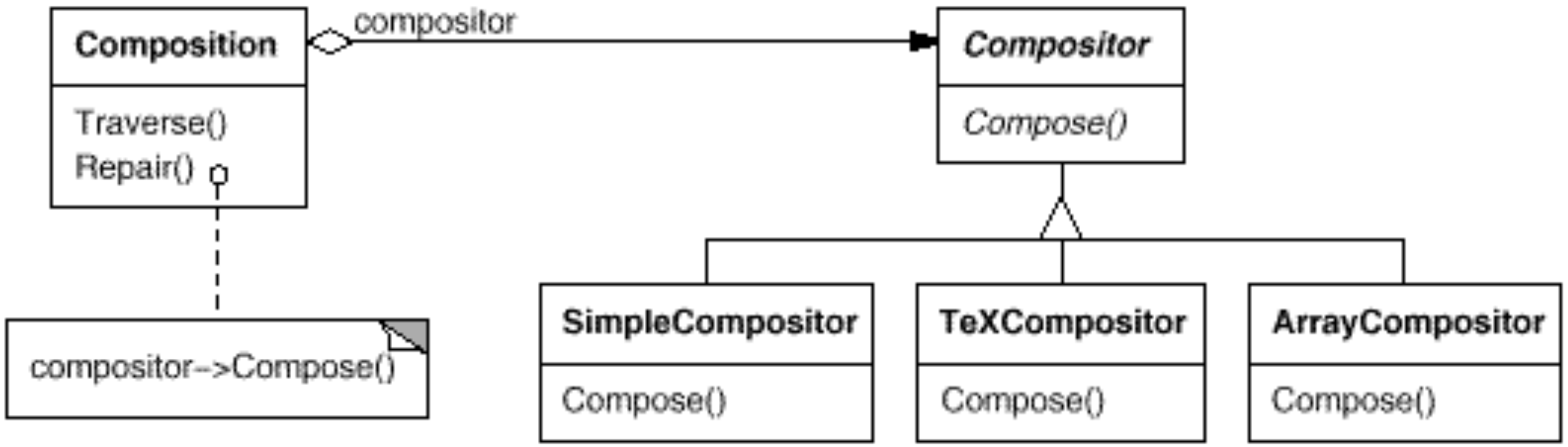
- Subclassing a key class because just one method is different — yeuch!
- In reality, a commuter may decide on mode of transport at the last minute, but this model doesn't facilitate this

Applying the Strategy pattern

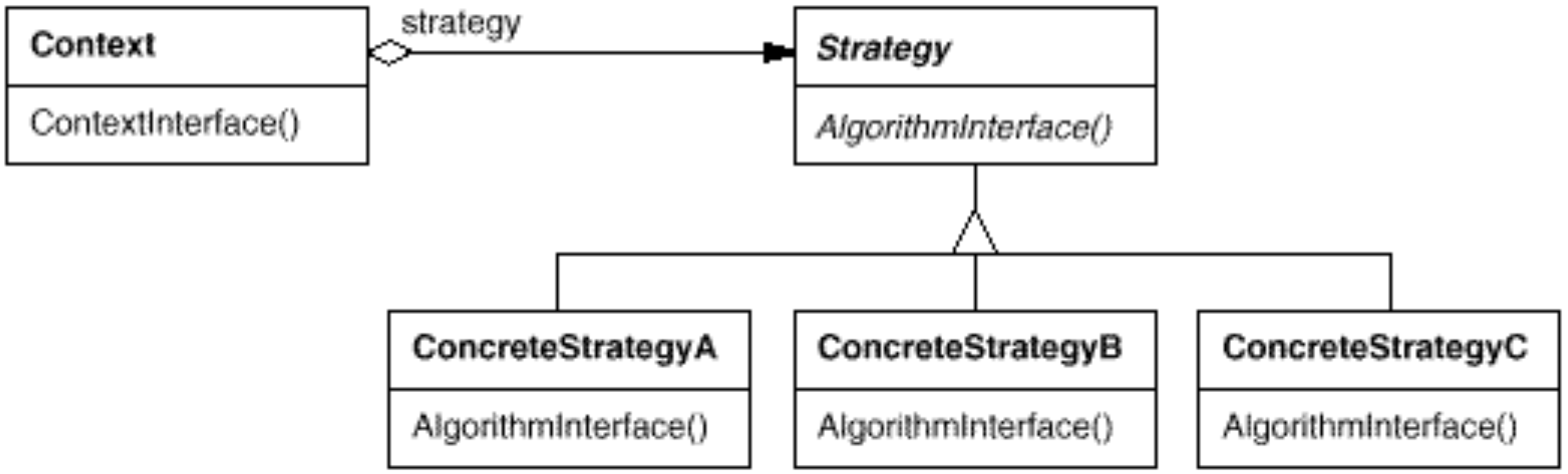


Another Example -- GoF Motivating Example

- Many algorithms exist for breaking a stream of text into lines. How can we configure an application to dynamically choose which one to use?



Strategy -- Typical Structure



How does the Strategy class access its Context class?

```
class Commuter {
  void travel_to_airport(){
    commute.travel_to_airport();
  }
  CommuteStrategy commute;
  Wallet my_wallet;
}


class Bus implements CommuteStrategy {
  void travel_to_airport(){
    ...
    driver.pay(){
      // needs to access my_wallet, but how?
    }
  }
 }
```

Strategy -- Applicability

Use Strategy whenever:

- Several related classes differ only in their behaviour.
- A class needs several variants of an algorithm.
- An algorithm uses data that clients shouldn’t know about. Use Strategy to avoid exposing complex, algorithm-specific data structures.
- A class defines many behaviours, and these appear as multiple conditional statements in its methods (this is a **code smell**).
  - Instead of many conditionals, move related conditional branches into their own Strategy class.

Strategy -- Consequences

- Provides an alternative to subclassing the Context class to create a variety of algorithms or behaviours.
- Eliminates large conditional statements.
- Provides a choice of implementations for the same behaviour.
- Increases the number of objects in the system.
- All algorithms must use the same Strategy interface.

Strategy -- Comments

- Related algorithms are grouped, and **Template Method** can be used to capture their commonality.
- Reduces subclassing of the Context class.
- Strategy can be implemented as a stateless object, so it can be shared by several contexts (**Flyweight**).
- Java.util.zip uses Strategy to enable two algorithms be used for performing a checksum on a stream: Adler32 and CRC32.