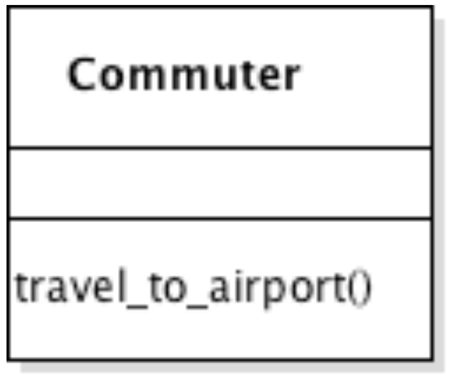
Strategy

- 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 the same task

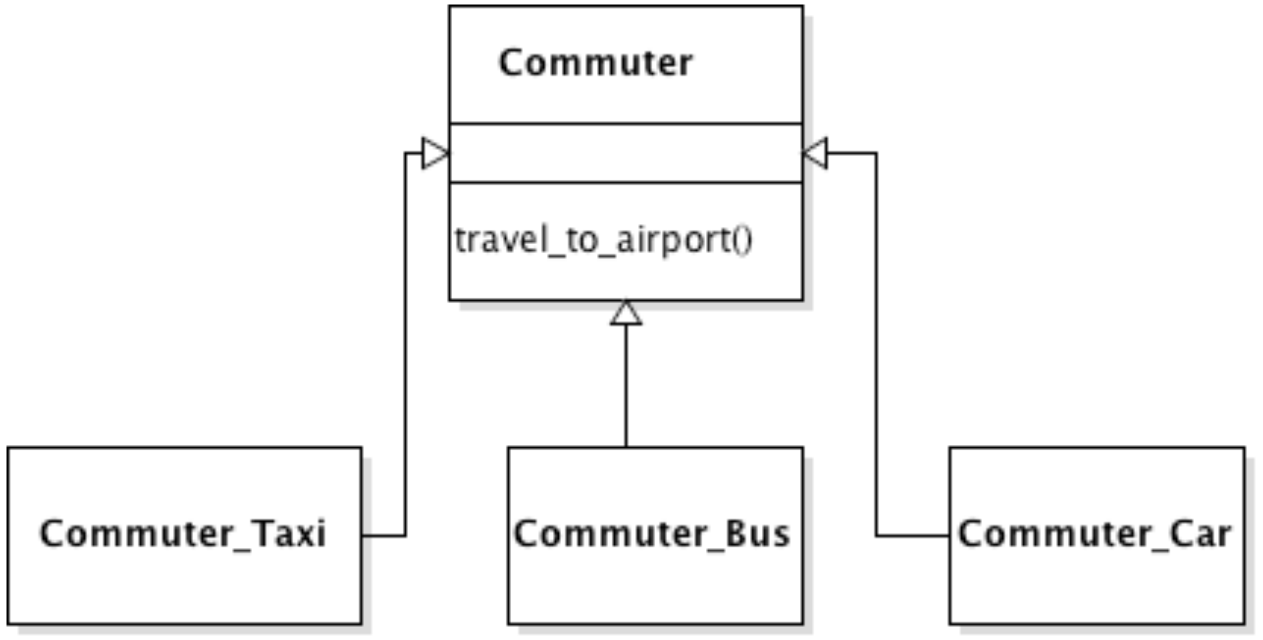
Simple Commuter Example

- A commuter needs to travel to the airport:



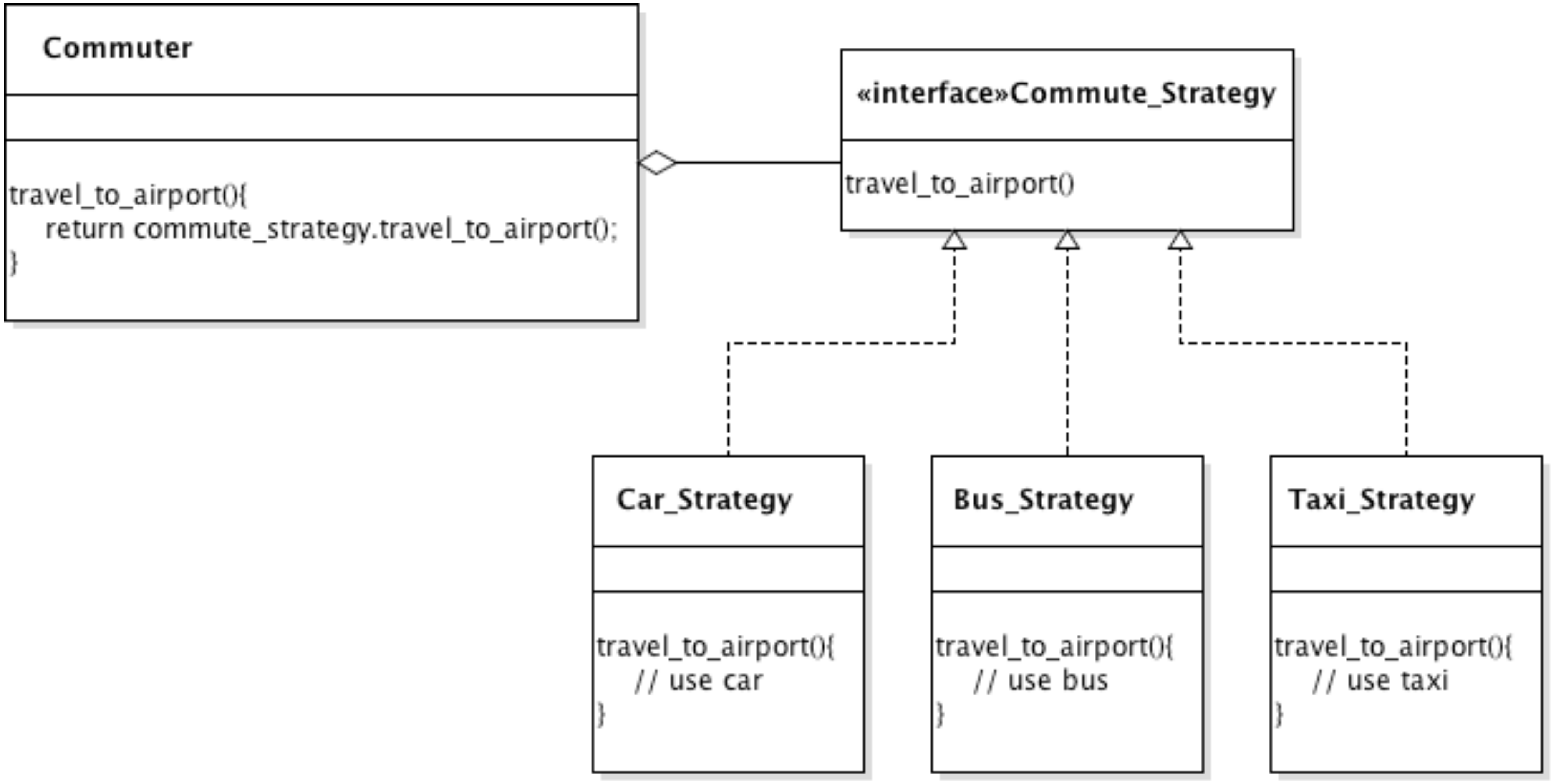
- They might travel by car, bus or taxi.
- How best to model this?

Using inheritance — a bad idea



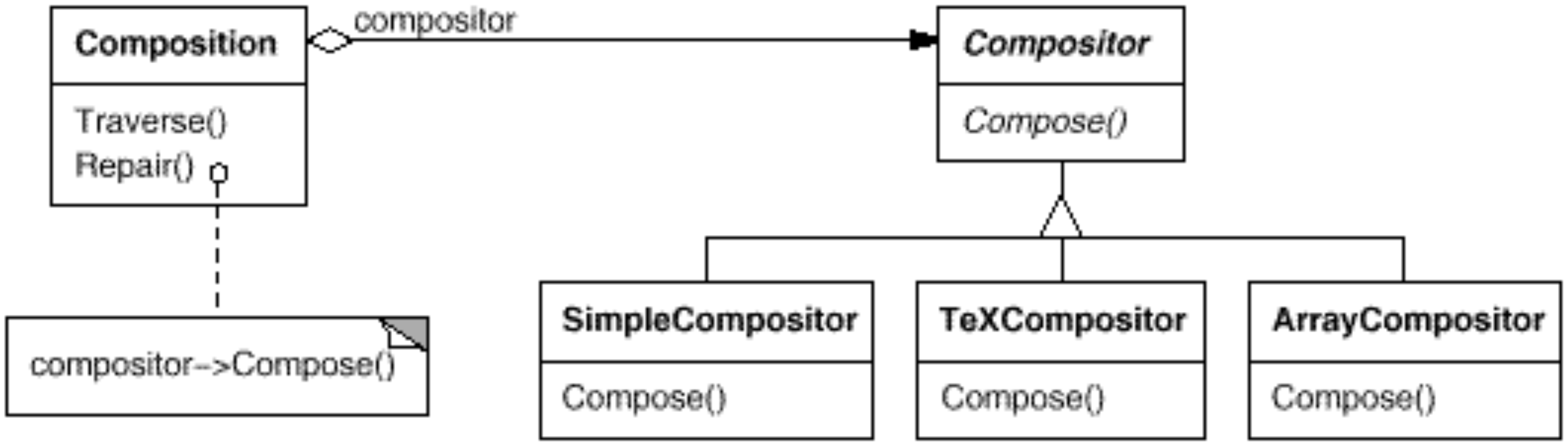
- Subclassing key class on basis of small differences — yeuch!
- Commuter may decide on mode of transport at last minute.

Applying the Strategy pattern

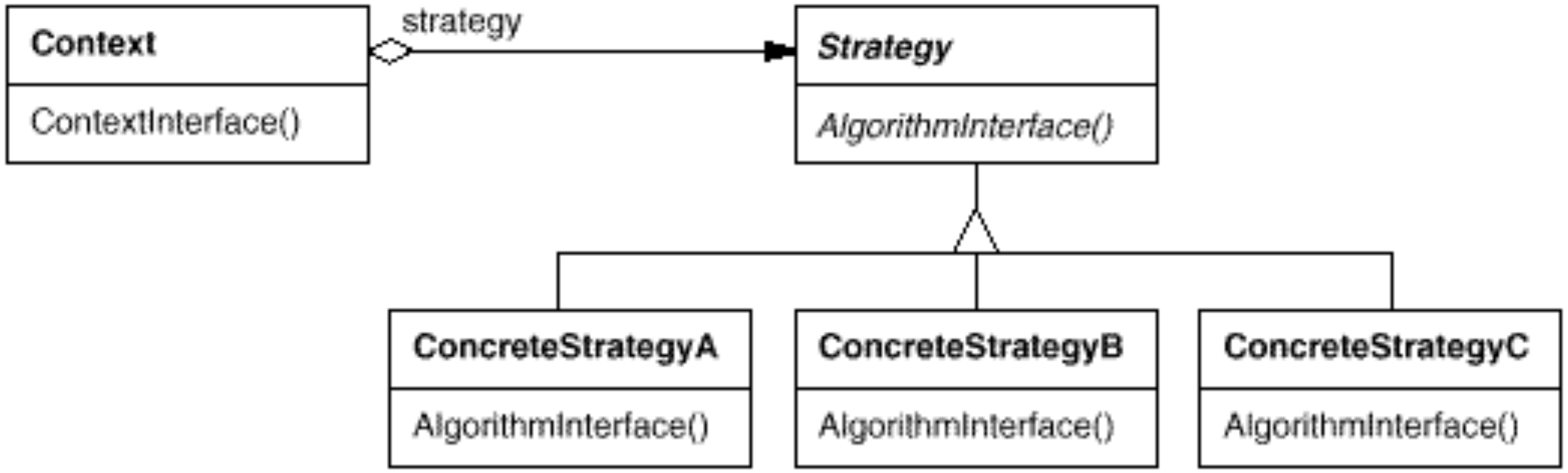


Strategy -- GoF Motivating Example

- 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



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.