The Strategy Pattern

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.

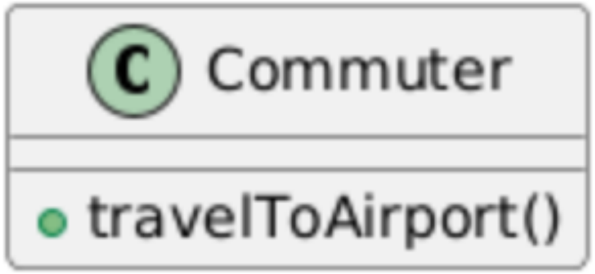
So consider using Strategy if a class should have multiple ways of performing some task.

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

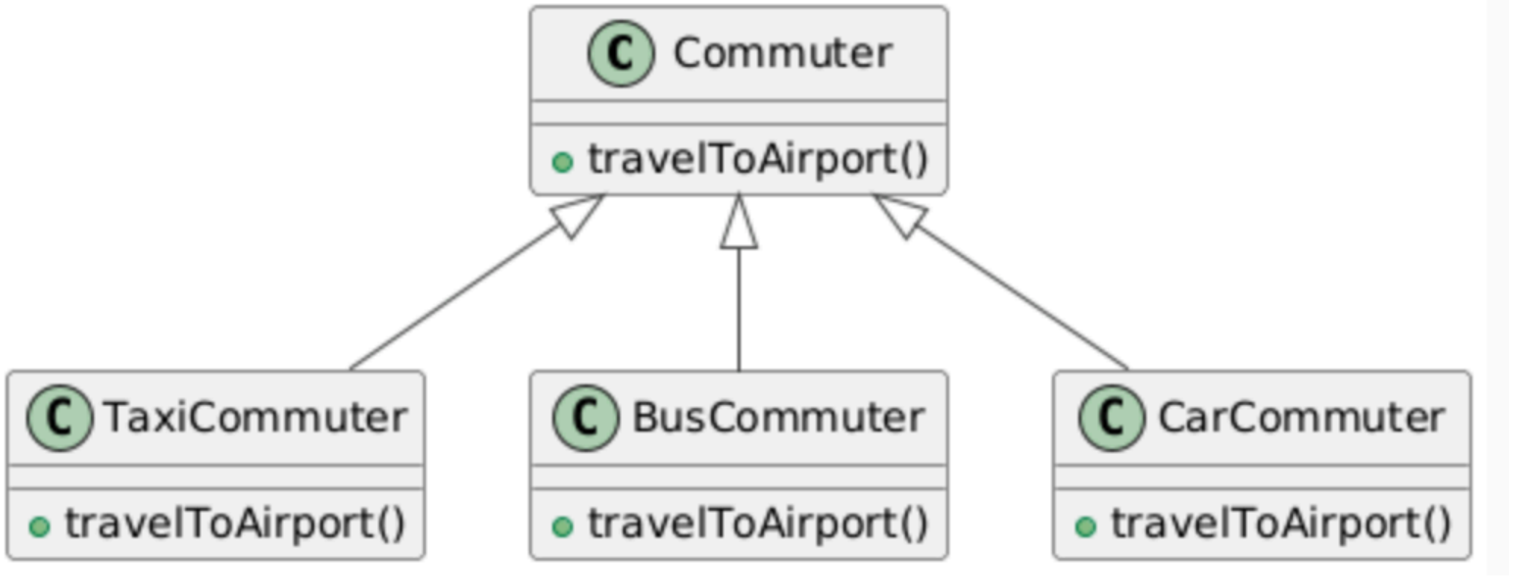
```
public void travelToAirport(String mode) {
  if mode == "car" then
    ... // travel by car
  else if mode == "bus" then
    ... // travel by bus
  else if mode == "taxi"  then
    ... // travel by taxi
}
```

Switching on type is a **code smell**

Method has three unrelated parts.

Gets worse if other methods in the class type check on the `mode` argument as well.

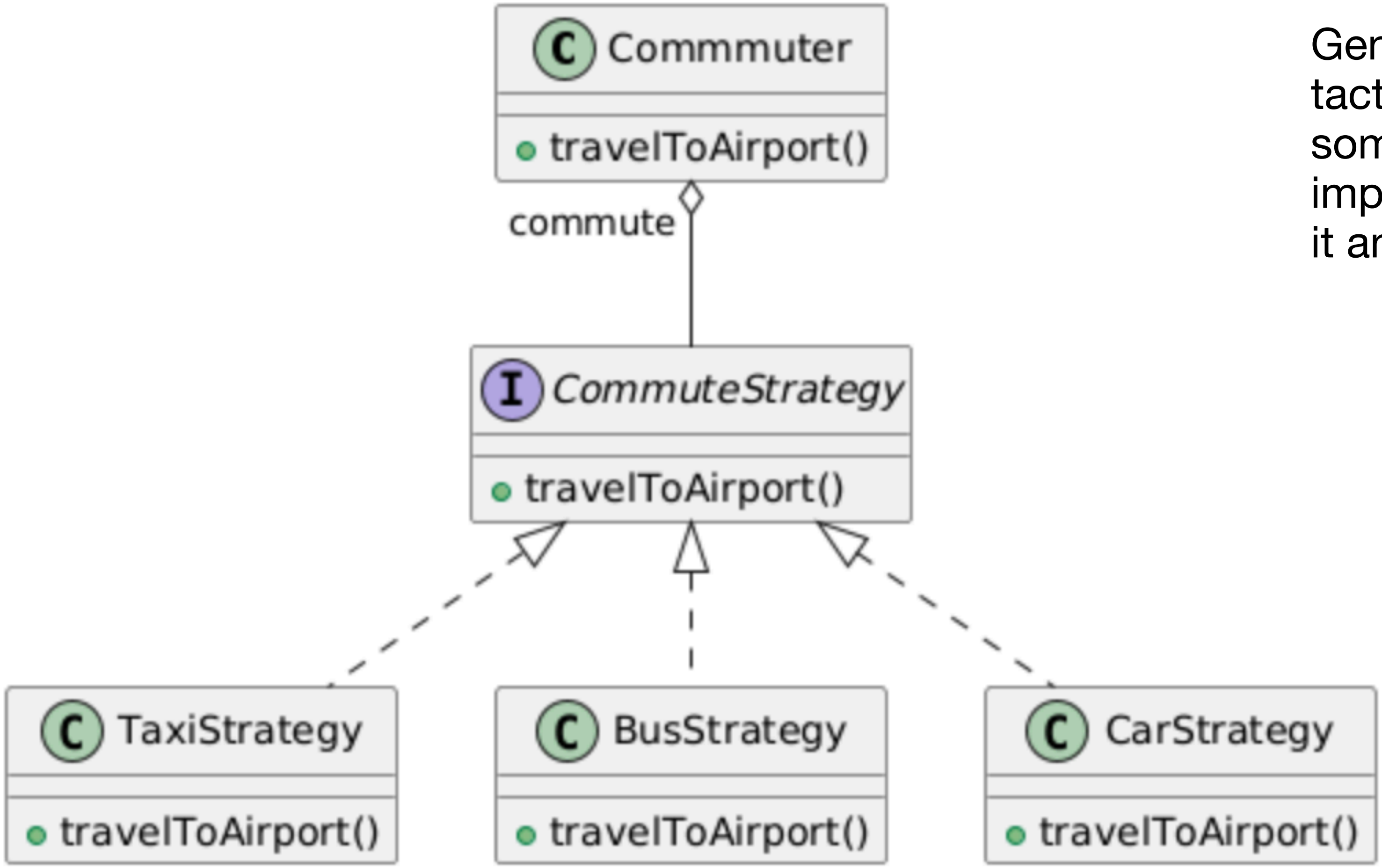
Using inheritance — not a good idea either



Subclassing because just one method is different — 👎.

Also, a commuter may change their mode of transport dynamically, but this model doesn’t facilitate this.

Applying the Strategy pattern



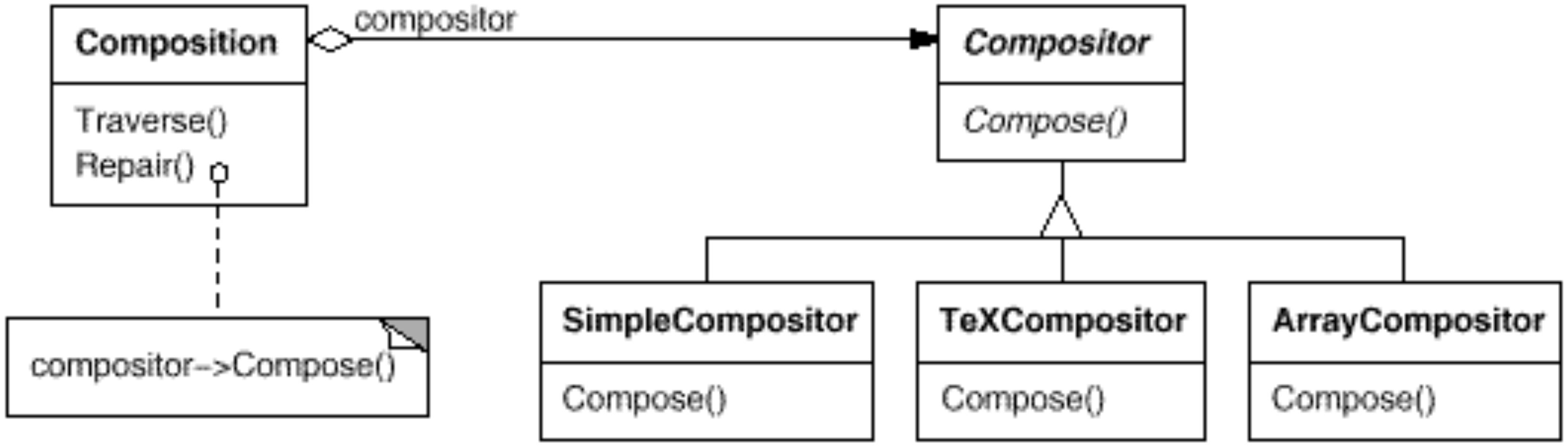
Generalisable tactic: when something is important, make it an object.

`Commuter` delegates to its `CommuteStrategy`:

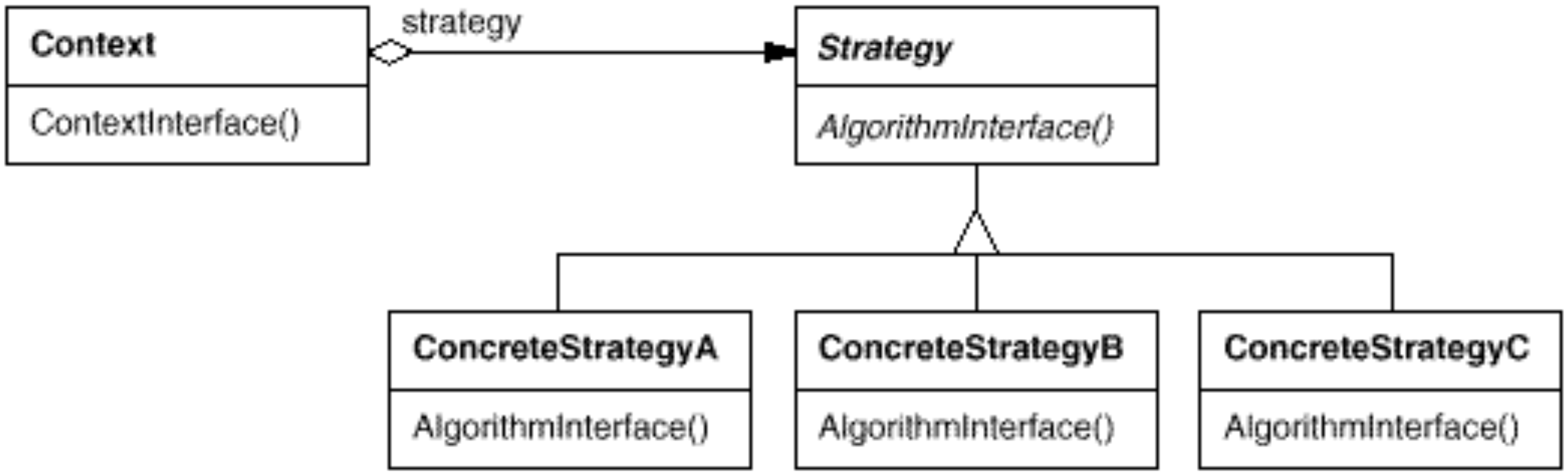
```
public void travelToAirport() {
  commute.travelToAirport();
}
```

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 Commuter {
  public void travelToAirport(){
    commute.travelToAirport();
  }
  private CommuteStrategy commute;
  private Wallet myWallet;
}

class BusStrategy implements CommuteStrategy {
  public void travelToAirport(){
    ...
  }
  private payDriver(){
    // needs to access myWallet, but how?
  }
}
```

Option (1): Pass it as an argument

```
class Commuter {
  public void travelToAirport(){
    commute.travelToAirport(myWallet);
  }
  private CommuteStrategy commute;
  private Wallet myWallet;
}

class BusStrategy implements CommuteStrategy {
  public void travelToAirport(Wallet wallet){
    ...
  }
  private payDriver(Wallet wallet){
    wallet.pay(...);
  }
}
```

Option (2): Pass reference to the Context object

```
class Commuter {

  public void travelToAirport(){
    commute.travelToAirport(this);
  }

  public void payDriver(){
    ...
  }

  private CommuteStrategy commute;
  private Wallet myWallet;
}
```

An additional public method has to be added

Option (2): Pass reference to the Context object

```
class BusStrategy implements CommuteStrategy {

  public void travelToAirport(Commuter commuter){
    this.commuter = commuter;
    ...
  }

  private payDriver(){
    ...
    commuter.payDriver();
    ...
  }

  private Commuter commuter;
}
```

Strategy -- Applicability

Consider using 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.
  - Instead of many conditionals, move related conditional branches into their own Strategy class.

Strategy -- A Functional Implementation

Strategy can also be implemented using an **anonymous function.**

In the example above, `Commuter` would simply take a `commute` function as argument, and invoke this function where needed.

This is very easy, and leads some to claim that Strategy is no longer a valid pattern, more a programming idiom.

However, if the strategy is more complicated, using the class-based approach makes more sense.

Strategy -- Consequences

Provides an alternative to subclassing the Context class to create a variety of algorithms or behaviours.

Eliminates repeated conditional statements.

Provides a choice of implementations for the same behaviour.

Clients must be aware of different Strategies.

May increase the number of objects in the system, but Strategies tend to be stateless and so a single instance can be shared (the **Flyweight** pattern).

All algorithms must provide the same Strategy interface.

Strategy -- Comments

Related algorithms are grouped, and **Template Method** (covered in COMP30950) can be used to capture their commonality.

Strategy can be implemented as a stateless object, so it can be shared by several contexts (**Flyweight**).

Strategy is a very common pattern, e.g. Java.util.zip uses Strategy to enable two algorithms be used for performing a checksum on a stream: Adler32 and CRC32.