The Strategy Pattern

A Behavioural Design Pattern

A presentation by Markus Wichmann

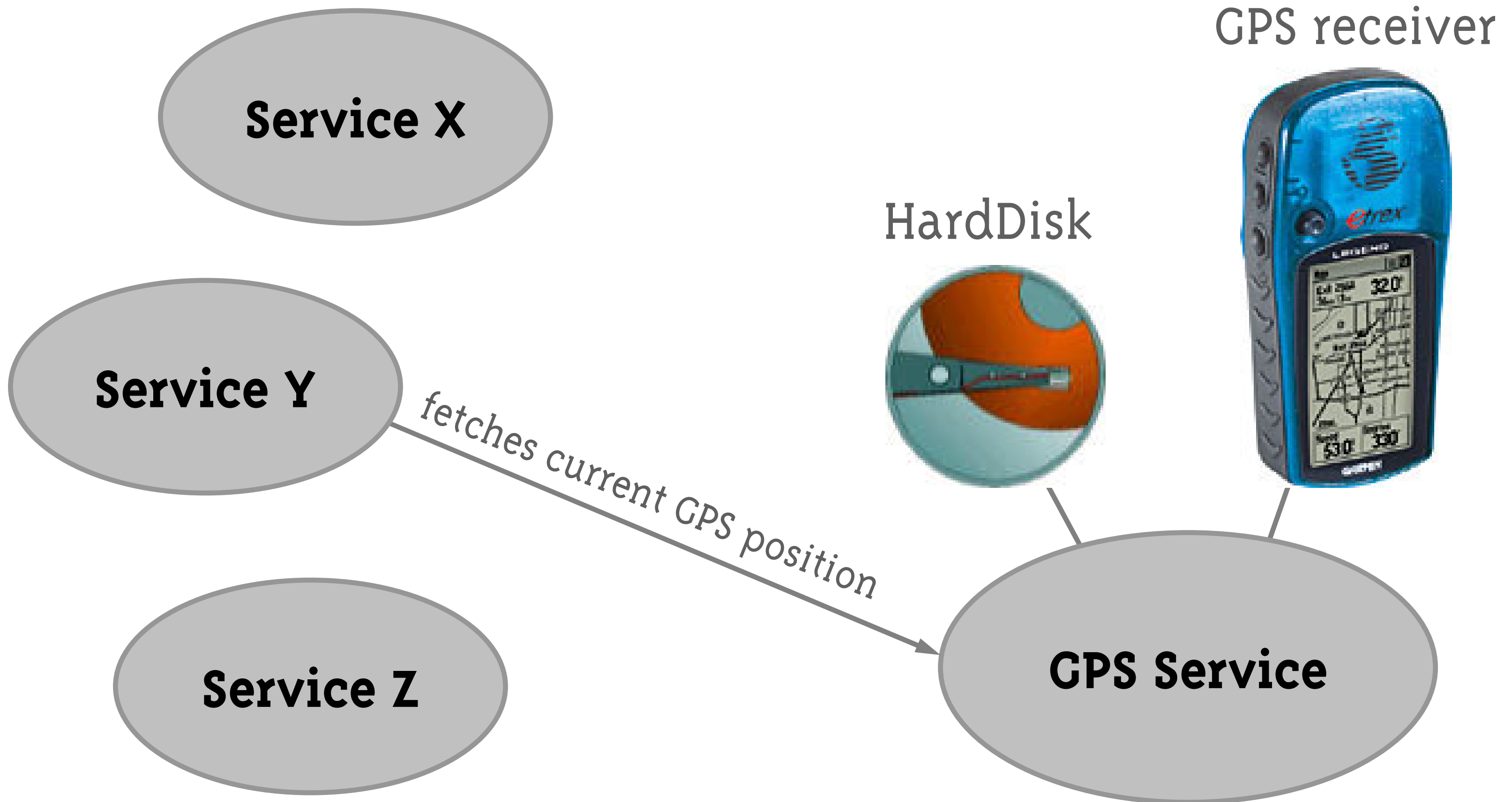
Downloading this presentation

m2w2.de/articles.html

or try markuswichmann.de/patterns

A presentation by Markus Wichmann

The project



Without the Strategy Pattern

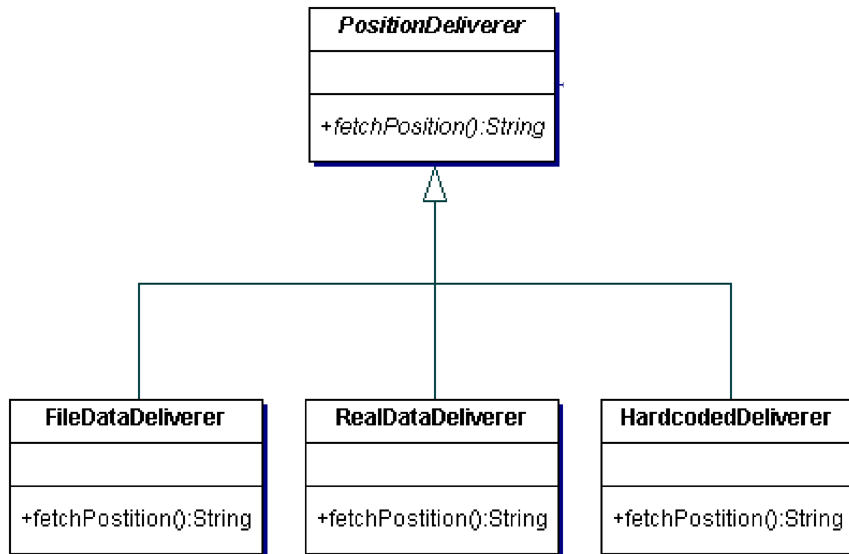
```
switch(dataSource) {
  case FILE_DATA:  { //... }
  case REAL_DATA:  { //... }
  case HARDCODED_DATA: {
     case OPTION_1:     { //... }
     case OPTION_2:     { //... }
     //...
  }
}
```

Abstract of the Strategy Pattern

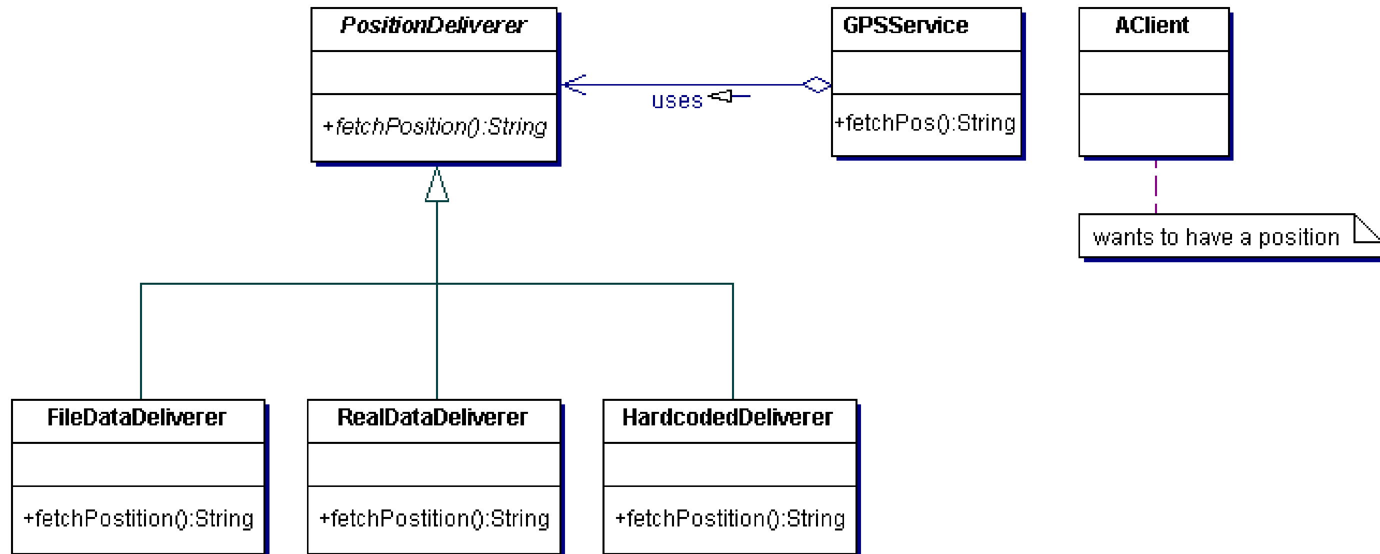
» Define a family of algorithms,
encapsulate each one,
and make them interchangeable.
Strategy lets the algorithm vary
independently from clients that use it. «

With the Strategy Pattern

With the Strategy Pattern



With the Strategy Pattern



1. AClient instantiates appropriate subclass of `PositionDeliverer`
2. AClient passes that new object to `GPSService`
3. Each time `GPSService` calls `fetchPosition()` on that object polymorphism delegates that call to the corresponding class

Implementation details

in class `AClient`:

```
PositionDeliverer posDel = new RealDataDeliverer();
GPSService gps = new GPSService(posDel);
gps.fetchPos(); // (internally calls posDel.fetchPosition
                    → polymorphism delegates)
```

When the code of (one of) the deliverers change(s), the clients don't have to be changed at all because they don't know anything about the deliverers' implementation.

When a different strategy is desired only the `RealDataDeliverer` in the example above has to be replaced by e.g. `FileDataDeliverer` (dynamically or hard-coded).

Pros & Cons of the Strategy Pattern

- + simplifies client code (exchange is only needed in one single line of code)
- + very easily switchable algorithms
- + very easily extendable
- + elimination of conditional statements (switch, if...else)
- + common functionality of different algorithms can be moved to a common superclass
- + clients do not need to get in contact with provider implementation
- + families of algorithms can be ordered hierarchically

- - providers (here: GPSService) should not change interface
- - clients must know of existance of different strategies (aaargh!)
- - number of objects increases
- - imagine a Strategy class with many many methods. Every concrete subclass of that Strategy class, even the most simple one, knows about those methods. But maybe a simple subclass never needs them.

Alternatives in some cases:

- Facade pattern
- Chain of responsibility

Thanks!

A presentation by Markus Wichmann