SWEN 262
Engineering of Software Subsystems

Strategy Pattern
### Sorting People

1. A *person* includes a name, an address, and an ID number. For example:
   a. Name: Rutherford B. Hayes
   b. Address: 17 E William St, Delaware, OH 43015
   c. ID Number: 001-00-019

2. People displayed by the application should be sortable based on:
   a. Name
   b. ZIP code and Street Name
   c. ID Number

<table>
<thead>
<tr>
<th>Name</th>
<th>ID Number</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>John Adams</td>
<td>001-00-002</td>
<td>141 Franklin St. Quincy, MA 02169</td>
</tr>
<tr>
<td>John Quincy Adams</td>
<td>001-00-006</td>
<td>141 Franklin St. Quincy, MA 02169</td>
</tr>
<tr>
<td>Chester A. Arthur</td>
<td>001-00-021</td>
<td>4588 Chester A Arthur Rd Fairfield, VT 05455</td>
</tr>
<tr>
<td>James Buchanan</td>
<td>001-00-015</td>
<td>6235 Aughwick Road Fort Loudon, PA 17224</td>
</tr>
<tr>
<td>Grover Cleveland</td>
<td>001-00-022</td>
<td>207 Bloomfield Ave Caldwell, NJ 07006</td>
</tr>
<tr>
<td>Bill Clinton</td>
<td>001-00-042</td>
<td>117 S Hervey St. Hope, AR 71801</td>
</tr>
</tbody>
</table>

Q: How might you go about implementing this requirement?
public void sort(int sortType) {
    switch (sortType) {
        case LAST_NAME:
            // sort people by last name
            break;
        case ZIP_CODE:
            // sort people by ZIP code and then street name
            break;
        case ID_NUMBER:
            // sort people by ID number
    }
}

A: Use a conditional, such as a `switch` statement, to choose between different algorithms.

Q: We already know that conditionals like this can be a code smell. What are the drawbacks of this solution?

A: The class is not very cohesive: it is trying to do too many things. The sorting algorithms are likely to be complex, resulting in a lot of code.

In addition, it would be difficult to add new algorithms without modifying the code (violates OCP).
**Subclassing**

**A:** Create an abstract sort method in the sorting class, and implement the different sorting algorithms in subclasses.

**Q:** What are the drawbacks of this solution?

**A:** The algorithms are hard wired to the subclasses. This requires several different versions of the sorting class that differ only by the sorting algorithm used.

This makes switching between algorithms difficult; a new instance needs to be created and the data copied between them.
Defining a Sorting Strategy

Begin by defining an interface to represent a sorting strategy. This interface will be implemented by any class that can sort a list of people.

```java
public interface PeopleSorter {
    public void doSort(List<Person> people);
}
```

Next, create a concrete strategy for one of the required sorting strategies, e.g. sorting people by last name and then, if the last names match, by first name.

```java
public class SortByName implements PeopleSorter {
    public void doSort(List<Person> people) {
        people.sort(
            Comparator.comparing(Person::getLastName)
            .thenComparing(Person::getFirstName));
    }
}
```

Continue creating concrete strategies for each of the required sorting strategies.
Interchangeable Strategies

Modify the sorting class, i.e. PeopleList, so that it is a context on which the sorting strategy can be changed.

The context will need to aggregate an instance of the strategy interface...

...and provide a method that allows external clients to change the strategy at any time.

When a client calls the method to sort the list of people, the context will delegate the responsibility for handling the sort to the current strategy.
Context

- strategy: Strategy
+ setStrategy(strategy: Strategy)
+ execute()

strategy.algorithm()

Client

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

(Behavioral)

---

Strategy
<<interface>>

# algorithm()

ConcreteStrategyA

# algorithm()

ConcreteStrategyB

# algorithm()

ConcreteStrategyC

# algorithm()
People Sorter Strategy Design

Application
<<Client>>

PeopleList
<<Context>>
- sorter: PeopleSorter
- people: List

+ setSorter(sorter: PeopleSorter)
+ sort()

PeopleSorter
<<Strategy>>

# doSort(people: List)

Once again, note that the classes have appropriate names that are in context for this application...

But that each participant also includes its role in <<guillemets>> below the class name...

sortByName
<<ConcreteStrategy>>

# doSort(people: List)

sortByAddress
<<ConcreteStrategy>>

# doSort(people: List)

sortByID
<<ConcreteStrategy>>

# doSort(people: List)

Once again, note that the classes have appropriate names that are in context for this application...

But that each participant also includes its role in <<guillemets>> below the class name...

Application
<<Client>>

PeopleList
<<Context>>
- sorter: PeopleSorter
- people: List

+ setSorter(sorter: PeopleSorter)
+ sort()

PeopleSorter
<<Strategy>>

# doSort(people: List)

Once again, note that the classes have appropriate names that are in context for this application...

But that each participant also includes its role in <<guillemets>> below the class name...

sortByName
<<ConcreteStrategy>>

# doSort(people: List)

sortByAddress
<<ConcreteStrategy>>

# doSort(people: List)

sortByID
<<ConcreteStrategy>>

# doSort(people: List)
<table>
<thead>
<tr>
<th>Class</th>
<th>Role in Pattern</th>
<th>Participant’s Contribution in the context of the application</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>Client</td>
<td>The main user interface to the application. When the user clicks one of the table columns, the application will change the current sorting strategy used to sort people in the UI.</td>
</tr>
<tr>
<td>PeopleList</td>
<td>Context</td>
<td>Maintains a list of people and exposes a method to allow clients to sort the list. Exposes a method that allows clients to change the order in which people are sorted by setting a concrete PeopleSorter.</td>
</tr>
<tr>
<td>PeopleSorter</td>
<td>Strategy</td>
<td>The interface for an algorithm that is capable of sorting people. Concrete implementations of this interface should compare two people to determine the desired natural ordering (e.g. which comes first).</td>
</tr>
<tr>
<td>SortByName</td>
<td>ConcreteStrategy</td>
<td>A PeopleSorter that sorts two people by first comparing the last names. If the last name is the same, the first names of the two people are compared.</td>
</tr>
<tr>
<td>SortByAddress</td>
<td>ConcreteStrategy</td>
<td>A PeopleSorter that sorts two people by first comparing the ZIP code of their addresses. If the ZIP code is the same, the street name is used. If the street name is the same, the house number is used.</td>
</tr>
<tr>
<td>SortByID</td>
<td>ConcreteStrategy</td>
<td>A PeopleSorter that sorts two people by comparing their ID numbers. ID numbers are unique, and so there does not need to be a fallback strategy to handle two people with the same ID.</td>
</tr>
</tbody>
</table>

Deviations from the standard pattern: None

Requirements being covered: 2. People are sortable by 2a. name, 2b. ZIP code and street, 2c. ID number.
This sequence diagram shows a user starting the application, and then changing the sort order.
Strategy

The intent of the Strategy pattern gives it distinguishing characteristics from other patterns.

- "Family" refers to a set of algorithms that perform the same type of operation but use different techniques, for example:
  - Sorting
  - Searching
  - Layout Managers (Swing, JavaFX, Android)

- An external client (a separate class) usually specifies the strategy to use.
- Usually the strategy is relevant to only one aspect of the context’s operation.

In the near future you will notice that some patterns have similar (or even identical) structures, but differ in their intent.

These differences are crucial when choosing the appropriate pattern.
Strategy

There are several *consequences* to implementing the strategy pattern:

- **Families of related algorithms.**
- **An alternative to subclassing.**
- **Elimination of conditional statements.**
- **A choice of different implementations of the same behavior** *(e.g. quicksort vs. merge sort).*
- **Clients must be aware of the alternatives to switch between them.**
- **Communication overhead between Context and Strategy.**
- **Increased number of objects in the system.**

**Things to Consider**

1. How does Strategy affect the overall cohesion in the system?
2. The coupling?
3. How does it support the Open/Closed Principle?
4. What other design principles might strategy make better or worse?
5. When might it not be appropriate to use Strategy?