Strategy

Intent

Define a class hierarchy for a family of algorithms, encapsulate each one, and make them interchangeable.

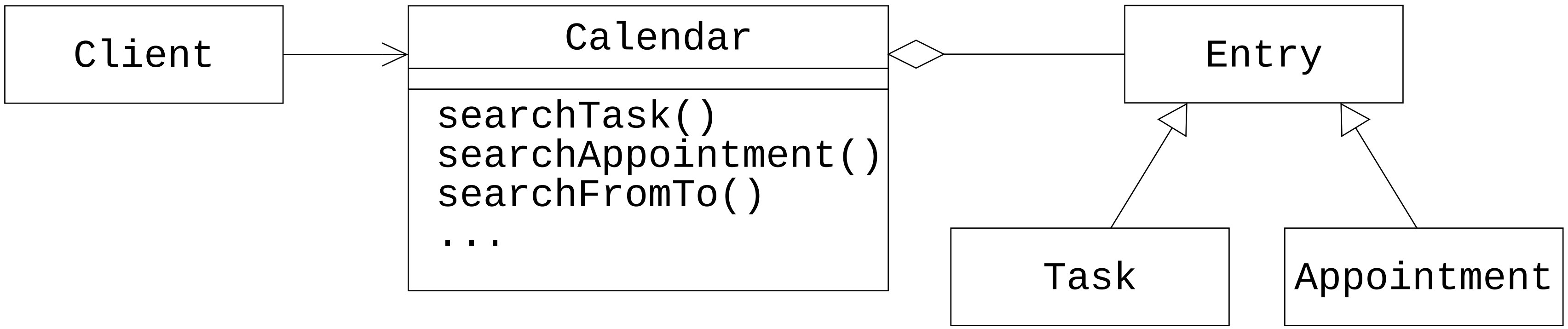
Also Known As

Policy.

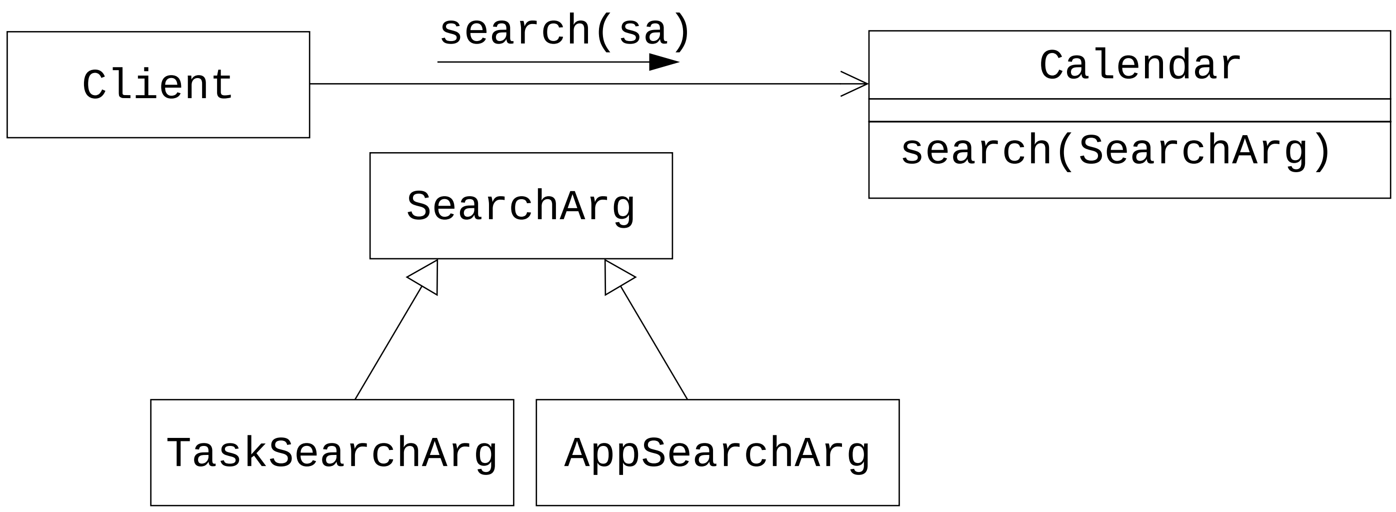
Motivation

Suppose you have write the search function of a calendar application. The calendar stores different kinds of `Entry` objects, and the user's search depends on the kind of `Entry` objects. In addition, a user may choose among several date ranges, and other `Entry` attributes.

This kind of requirement could easily lead to following class diagram:



An alternative way is to provide one method `search` which takes an object of a subclass of a (potentially) abstract class `SearchArg`:



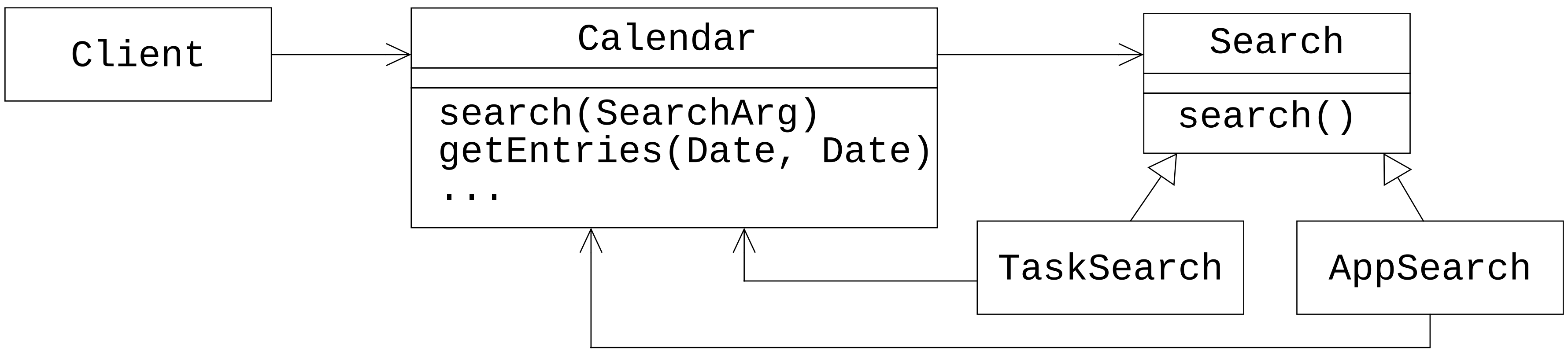
The burden of implementing the searching now lies in method `search`. Several cases need to be considered:

- kind of concrete subclass of the search argument
- values of the attributes of the search argument.

It is easy to guess that the `search` method becomes quite complicated in terms of many nested `if`-statements.

In addition, it should be clear that introducing a new kind of search function requires to extend an already complicated method, becoming even more complicated.

An alternative is to decompose the `search` method that use several specialized classes:



Method `search` of class `Calendar` does not compute the search, but determines which search algorithm (a concrete object of class `Search`) is applicable. It then *delegates* the search to that particular object and applies method `search` on it.

A concrete object of a subclass of `Search`, in turn, uses the `Calendar` object and its context, denoted by the `getEntries` method.

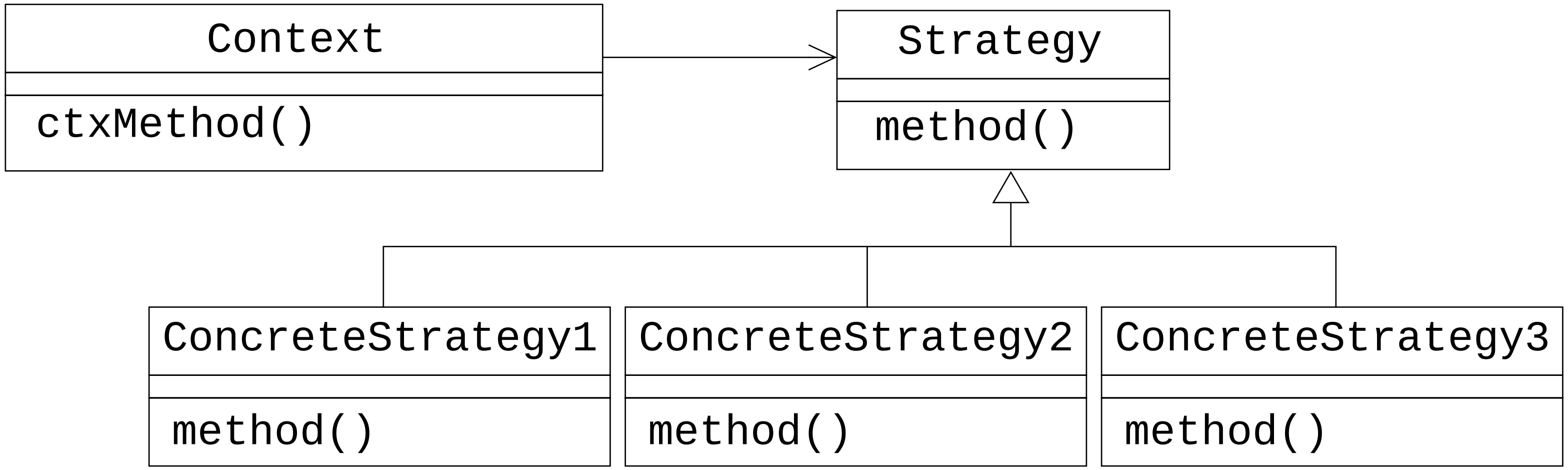
Class `TaskSearch` implements an algorithm to find entries of kind `Task` only. Analogous to class `AppSearch`: This class returns only `Appointment` entries matching the criterion given.

Applicability

Use the strategy pattern when

- many related classes differ only in their behavior. Strategy allows to configure a class with one or many behaviors.
- you need different variants of an algorithm.
- a class defines many behaviors, and these appear as multiple conditional statements in its operations.

Structure


Context
ctxMethod()
Strategy
method()
ConcreteStrategy1
method()
ConcreteStrategy2
method()
ConcreteStrategy3
method()

Participants

- `Context (Calendar)`:
  - provides a service to its clients
  - uses a particular `Strategy` object
  - if necessary, defines an interface that lets `Strategy` objects access its data
- `Strategy (Search)`:
  - declares an interface common to all supported algorithms. `Context` uses this interface to call the algorithm
- `ConcreteStrategy (TaskSearch, AppSearch, ...)`:
  - implements the algorithm using the `Strategy` interface

Collaborations

- `Strategy` and `Context` interact to implement the chosen algorithm.
- A `Context` forwards request from its clients to its concrete `Strategy` object.

Consequences

- A family of related algorithms can be expressed in a class hierarchy.
- The algorithm can be varied dynamically by changing the concrete `Strategy` object.
- The strategy pattern offers an alternative to conditional statements for selecting behaviors.
- Clients must be aware of different strategies. This is achieved in section “Motivation” by providing concrete objects of a subclass of `SearchArg`.

Implementation

The definition of the `Strategy` and `Context` interface:

- have the `Context` object pass data to the `Strategy` method.
- have the `Context` object to pass itself as an argument, an let the `Strategy` object request data from the context explicitly.

Sample Code

We briefly sketch the code of the example given in the “Motivation” section. We start defining the `Search` class. Notice that we pass as arguments the `Calendar` object as well as the `SearchArg` object.

```
public abstract class Search {
   public abstract List search(Calendar cal, SearchArg sa);
}
```

A sketch of the class hierarchy of `SearchArg` might look like:

```
public class SearchArg implements Serializable {
   public abstract Search makeStrategy(); // optional variant
}
public class TaskSearchArg extends SearchArg {
   public Date from, to;
   public Search makeStrategy() { return new TaskSearch(); }
}
```

Class `Calendar` offers two methods: Method search is for clients. Method `getEntries` is used by concrete `Strategy` objects:

```
public class Calendar {
   ...
   List getEntries(Date from, Date to) { // helper
      // Returns ordered list of any kind of Entry objects,
      // details not shown.
   }

   public List search(SearchArg sa) {
      // We assume that the SearchArgument has a factory
      // method which creates and returns the matching Strategy
      // object.
      List l = sa.makeStrategy() .search(this, sa);
      return l;
   }
}
```

In method `search` above, we assume that each `SearchArgument` object has a matching `Strategy` object. This then simplifies method search! Alternatively, if no factory methods are provided, then the exact type of the given `SearchArg` object can be queried, and the corresponding `Strategy` object can then be created and used:

```
// in class Calendar, second variant:
public List search(SearchArg sa) {
   if (sa instanceof TaskSearchArg)
      return new TaskSearch() .search(this, sa);
   else if ...
   else // Oops, we got an argument we cannot handle!
      return new ArrayList();
}
```

Note that the above variant depends on the concrete `SearchArg` classes!

Finally, we sketch the concrete `TaskSearch` class:

```
public class TaskSearch extends Search {

   // Note that can safely assume the specialization of
   // the SearchArg argument. This allows us to access its
   // fields (perhaps, via getters).
   public List search(Calendar cal, TaskSearchArg sa) {
      // get the Date fields from and to, then:
      List tempres = cal.getEntries(from, to);
      List res = removeNonTasks(tempres);
      return res;
   }

   protected List removeNonTaks(List tempres) {
      List tasks = ...;
      // returns a subset of the argument given,
      // containing only Entry objects of kind Task.
      // Details omitted.
      return tasks;
   }
}
```

Related Patterns

- Flyweight: Strategy objects make good flyweights.
- Factory method: Allows the `SearchArg` to return appropriate `Search` object. Either `SearchArg` is the concrete factory object, or it uses a concrete factory, accessible as singleton, for example.