Design Patterns

Composite Pattern

Structural pattern

Problem: we have a collection of objects in which an object can be composed of other objects

Goal: We want to treat all objects (containers and components) uniformly / perform the same action on all the objects in the collection

Composite Pattern

Solution:

- Share behavior/activity across all objects (using an interface)
- Each subclass implements the activity
- Call the activity on the first object in the collection

Sound familiar?

Composite Pattern: Example

Expression Tree

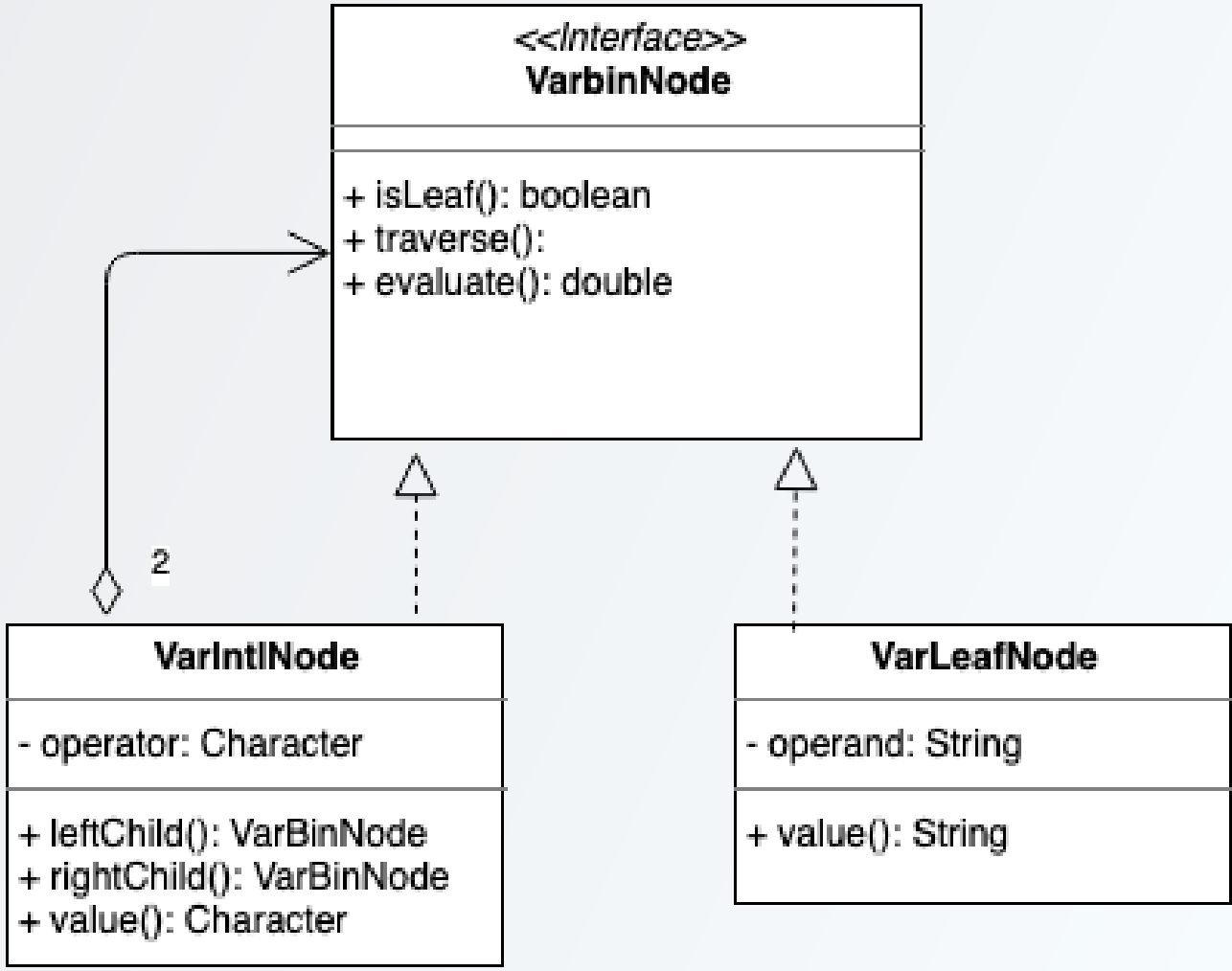
- Collection of Nodes: internal and leaf nodes.
  - Internal nodes composed of (two) internal and/or leaf nodes
- We want to evaluate the expression tree
  - Call `evaluate` on the root of the tree
  - Each internal node invokes `evaluate` of its subtrees with the appropriate operator
  - Leaf node return its value when performing `evaluate`

Composite Pattern: Example

Class	Purpose
`VarBinNode`	Interface for a binary node
`VarIntlNode`	Class for an internal node, implements `VarBinNode`
`VarLeafNode`	Class for a leaf node, implements `VarBinNode`

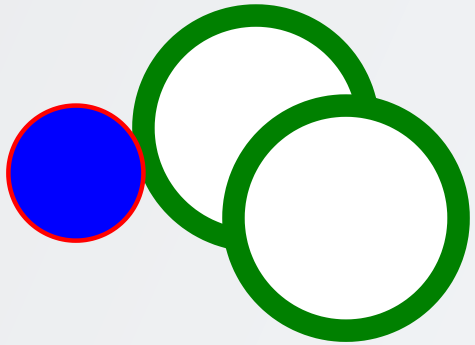
Composite Pattern: Example

Class diagram



Composite Pattern: SVG `g` Element

```
<svg viewBox="0 0 100 100" xmlns="http://www.w3.org/2000/svg">
  <!-- Using g to inherit presentation attributes -->
  <g fill="white" stroke="green" stroke-width="5">
    <circle cx="40" cy="40" r="25" />
    <circle cx="60" cy="60" r="25" />
  </g>
</svg>
```



How to Draw an SVG?

- For each element in the SVG, we can call the `draw` method using:
  - its style properties
  - any style properties of its ancestors
- If the element is a single object, it will draw itself
- If the element is a group (a `g`), it will recursively draw each of its children

Strategy Pattern

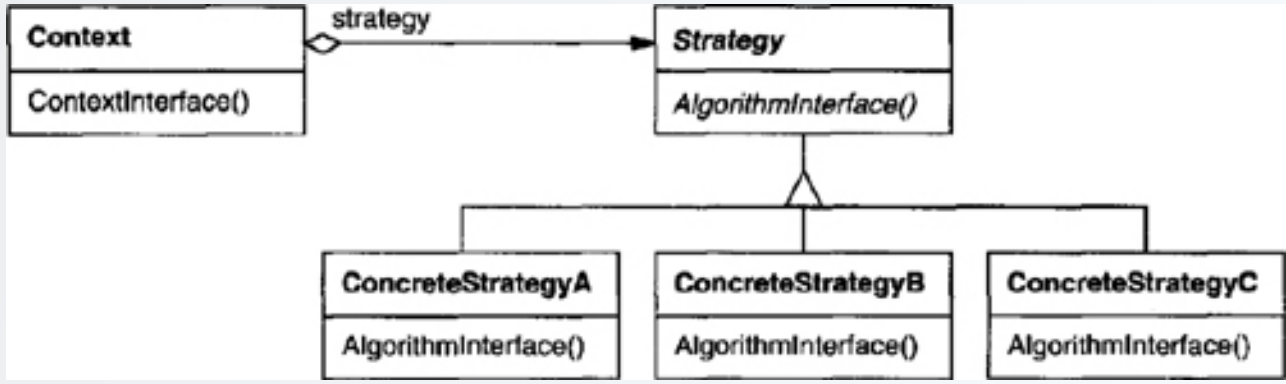
Behavioral pattern

Problem: We want to use a general algorithm; some part of it may vary depending on the context. We want to avoid multiple slightly different version of the same algorithm

Goal: Improved reusability of the code

Strategy Pattern

Solution: Use a class that represents the strategy and pass an instance to a single method that implement the rest of the algorithm.

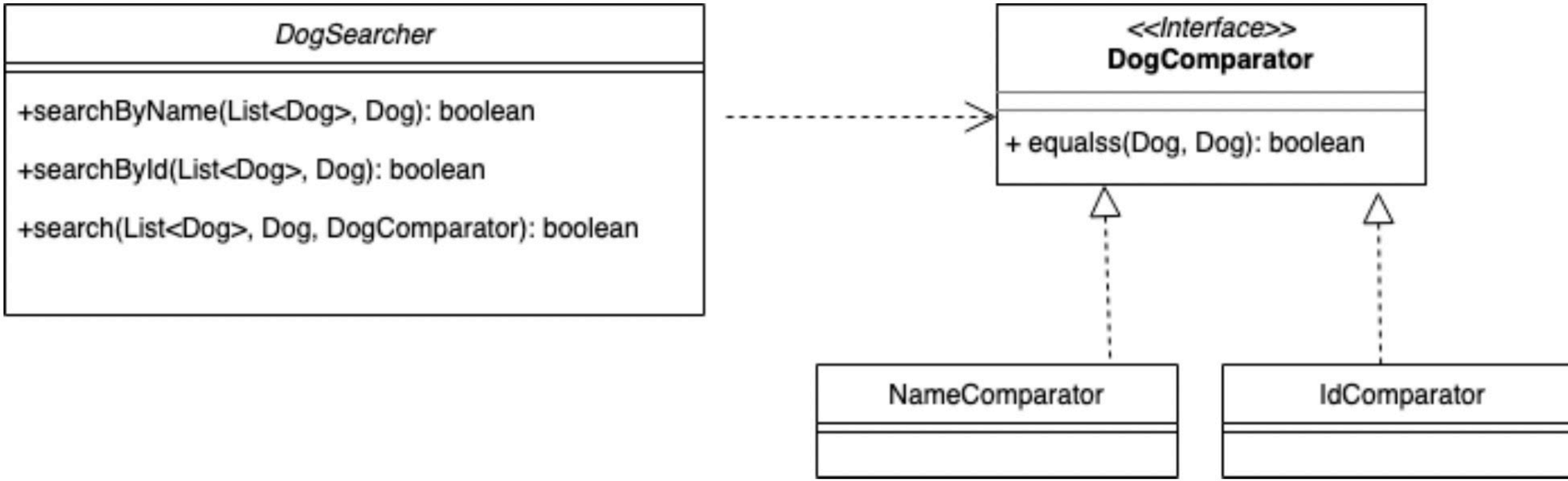


Strategy Pattern: An Old Example

We want to search for an element (target) in a collection of objects (dogs) based on different criteria: by name, or by id

We want to have one implementation of the search algorithm, and specify the strategy to use every time we are calling the algorithm

- The "strategy" here is just the use of one **Comparator** over another!



Created with draw.io (www.draw.io)

Functional Interfaces

`Comparator` is an example of a **functional interface**

- An interface that has only one unimplemented method
- An implementing class for a functional interface can consist of one function
- An **anonymous class** implementing a functional interface is basically a single function definition!

```
TreeSet<Treasure> ts = new TreeSet(new Comparator() {
        public int compare(Treasure t1, Treasure t2) {
                return t1.getValue() - t2.getValue();
        }
});
```

Anonymous Functions (Lambdas)

Java 8 introduced **lambdas** to simplify the syntax for implementing functional interfaces

```
TreeSet<Treasure> ts = new TreeSet((t1, t2) -> t1.getValue() - t2.getValue());
```

Here, `(t1, t2) -> t1.getValue() - t2.getValue()` represents a concise implementation of the `compare` method, which itself is a concise expression of an entire new `Comparator` class!

```
(inputOne, inputTwo, ...) -> oneLineExpressionGivingValueToReturn;
```

Passing "Functions"

You can also "pass functions" (not really, but close enough) like so:

```
TreeSet<Treasure> ts = new TreeSet(Comparator.comparing(Treasure::getValue));
```

`Comparator.comparing()` is a method that takes in **another method** that selects the proper value from the objects by which to compare them. It returns a `Comparator` that uses that method to compare the objects.