Unit-V

Behavioral patterns part-II (cont'd): state
strategy
Template Method
Visitor
Discussion of Behavioral pattern

What to expect from Design patterns: A brief history
the pattern community an invitation
A parting thought.

State:

the state of an object is a combination of the current values of its attributes.

the intent of the state pattern is to distribute state-specific logic across classes that represent an object's state

Modeling States:

the state pattern offers a cleaner, simpler approach using a distributed operation

To see State at work, it will help to first look at a system that models states without using the state pattern.

Consider the Oozinoz s/w that models the state of a carousel door:

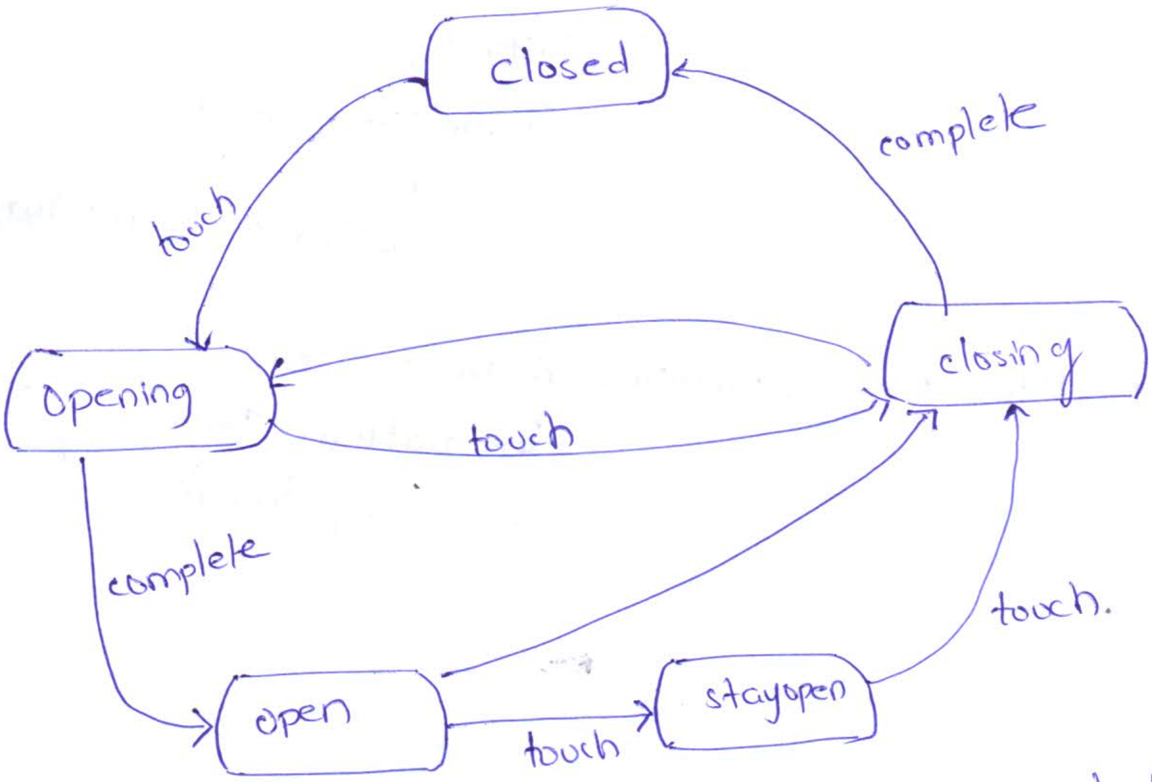


fig: states & transitions of the carousel door.

A carousel is a large, smart rack that accepts material through a doorway and stores the material acc. to a bar code ID on it.

→ you can prevent this by touching the button again when the door is open

→ It is a UML state machine

You can supply the carousel s/w with a Door object that the carousel s/w will update with state changes in the carousel

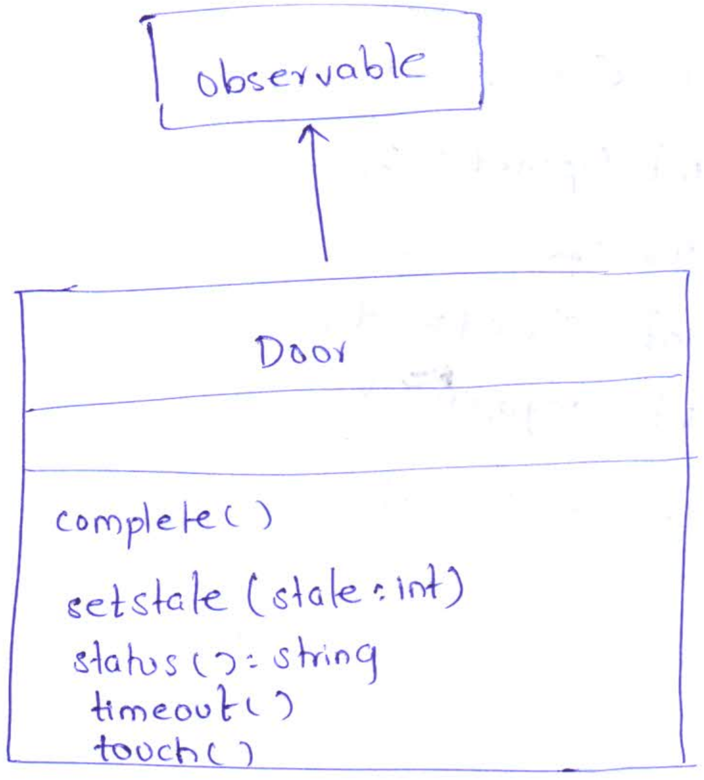


fig: Door class.

The door class is observable so that clients, such as a GUI can display the status of a door.

the class defination establishes the states that a door can enter:

```
package com.oozinoz.carousel;
Import java.util.Observable;
public class Door extends Observable
 {
    public final int Closed = -1
    public final  int Opened = -2;
    public final  int open  = -3;
    public final  int Closing = -4;
     public final int Stayopen = -5;
     private int state = closed;
   //...
 }
 public String status()
   {
     switch (state)
       {
       case Opening: return "Opening";
        case Open: return "Open";
         case closing: return "closing";
          case Stayopen: return "StayOpen";
        default:
              return "closed";
        }
    }
```

Refactoring of state :

The code for Door is somewhat complex because the use of the state variable is spread throughout the class.

→ the state pattern can help you to simplify this code, to apply state & make each state of the door a separate class.

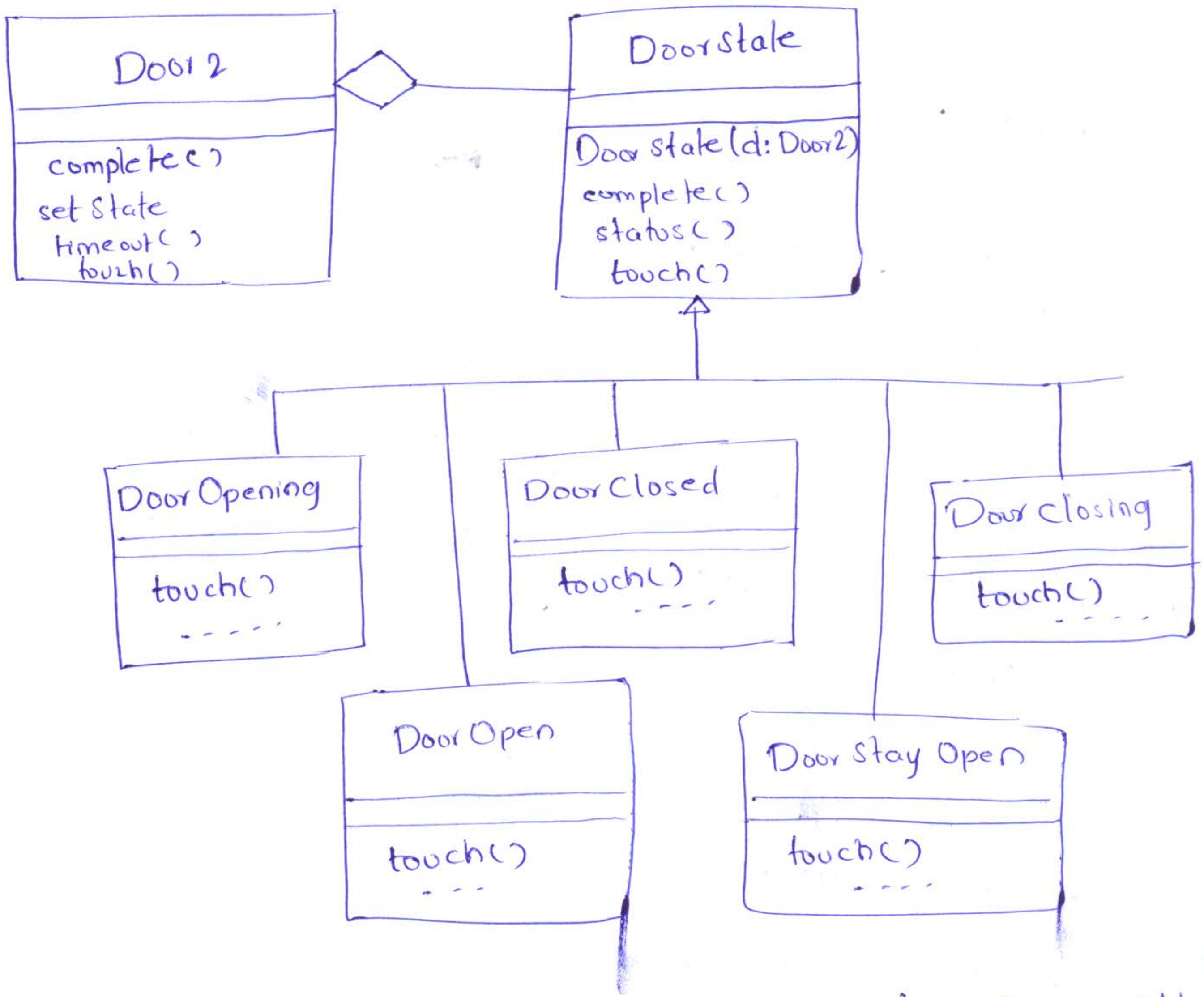


fig:- door's states as classes in an arrangement that mirror the door's state machine

It creates a special class for each state that the door might be in, each of these classes contains the logic for responding to a touch

```
public DoorClosed (Door2 door)
  {
    super (door);
  }
  public void touch ()
   {
     door.setState (door.Opening);
   }
}
```

the current design requires the generation of a Door 2 object to be accompanied by the creation of a suite of states that belong to door.

```
package com.oozinoz. carousel;
Import java.util. Observable;
public class Door2 extends Observable
{

}
```

Making states constant

the state pattern moves state-specific logic across classes that represent an objects state

state does not specify how to manage communication and dependencies btw the state objects. and the central object object to which they apply

you might prefer a design that created a single, static set of Doorstate objects and require the Door state to manage all updates resulting from state changes.

In this design the Door class's touch () method for example updates the state variable as follows:

```
public void touch ()
{
  state = state. touch ();
}
```

the DoorOpen class's touch () method now reads as follows

```
public Doorstate touch()
{
 return Door state. STAYOPEN;
}
```

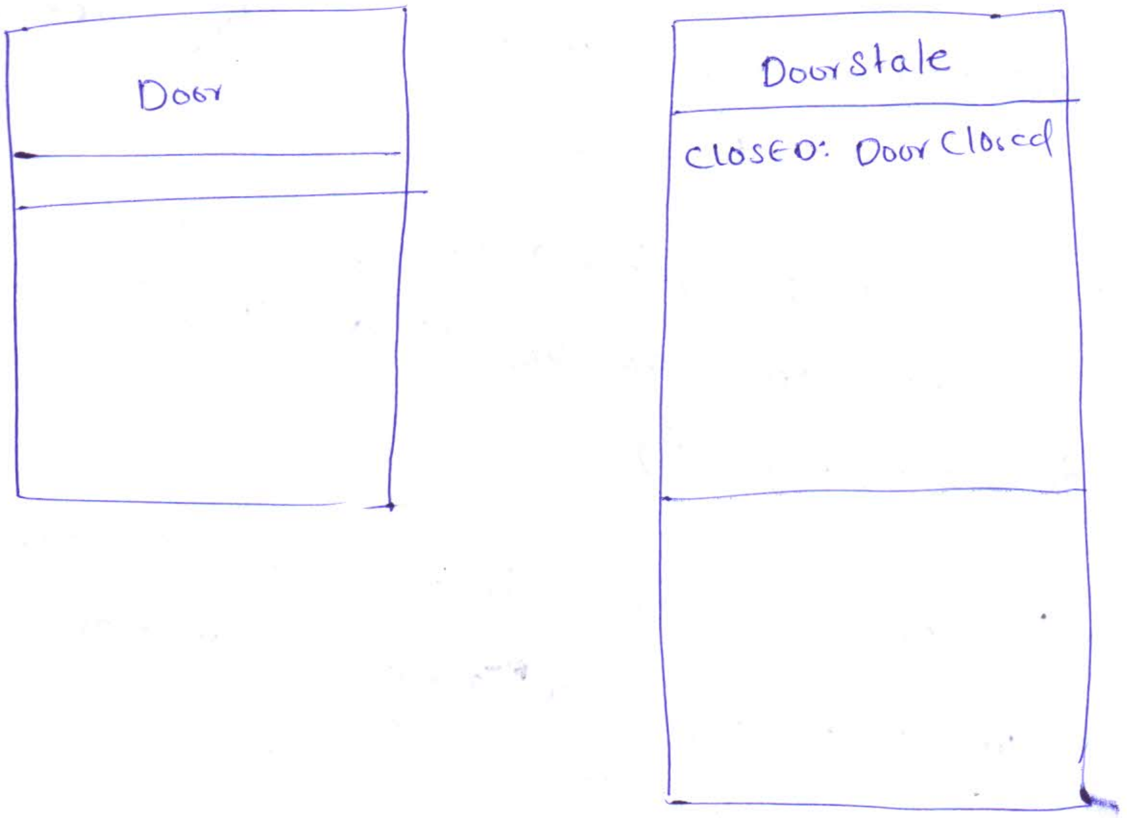


fig: show a design that makes door states constants.

You can also make the state subclasses mere information providers that determine the next state but that do not update the central object.

which ~~app~~ approach you use may depend on the context of your application or on aesthetics.

Strategy:

A strategy is a plan (or) approach for achieving an aim given certain input conditions

A strategy is thus similar to an algorithm a procedure that produces an o/p from a set of inputs.

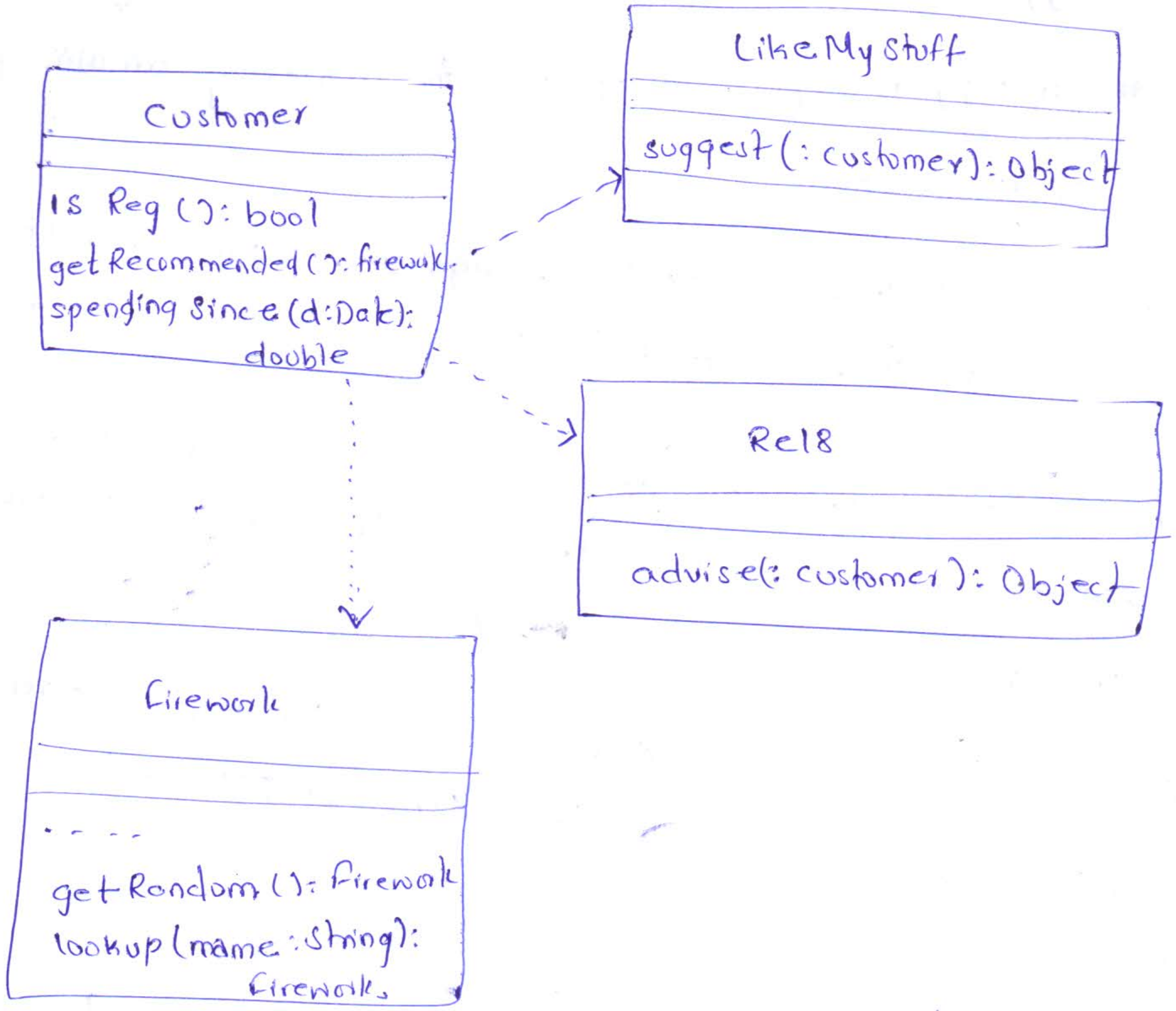
→ When multiple strategies appear in a computer program, the code may become complex, the logic that surrounds the strategies must select a strategy.

→ If the choice and execution of various strategies lead to complex code, you can apply the strategy pattern to cleanit up.

Modeling strategies:

the strategy pattern helps to organize and simplify code by encapsulating diff approaches to a problem in different classes.

we refactor this code, applying strategy to improve the quality of the code



Customer
is Reg (): bool
getRecommended (): firework
spending Since (d:Date): double
LikeMyStuff
suggest (: customer): Object
Rel8
advise(: customer): Object
firework
- - - -
getRandom (): firework
lookup (name : String): firework

fig: custome class relies on other classes.

the getRecommended () method expects that if a promotion is on . it will be named in a strategy

promote = J squirrel

the getRecommended () code will use the Rel8 engine if the customer is registered.

Refactoring to Strategy

The getRecommended() method presents several problems First it's long - so long that comments have to explain its various parts.

→ short methods are easy to understand, seldom need explanation and are usually preferable to long methods.

→ the getRecommended() method both chooses a strategy and executes it

→ these are two different and separable functions. you can clean up this code by applying strategy

→ Create an interface that defines the strategic operation

→ Implement the interface with classes that represent each strategy

→ Refactor the code to select and use an instance of the right stoategy class

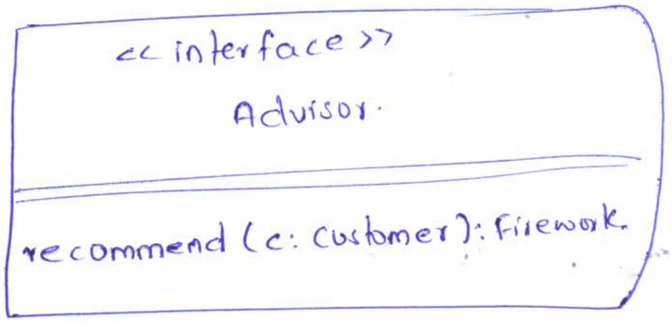
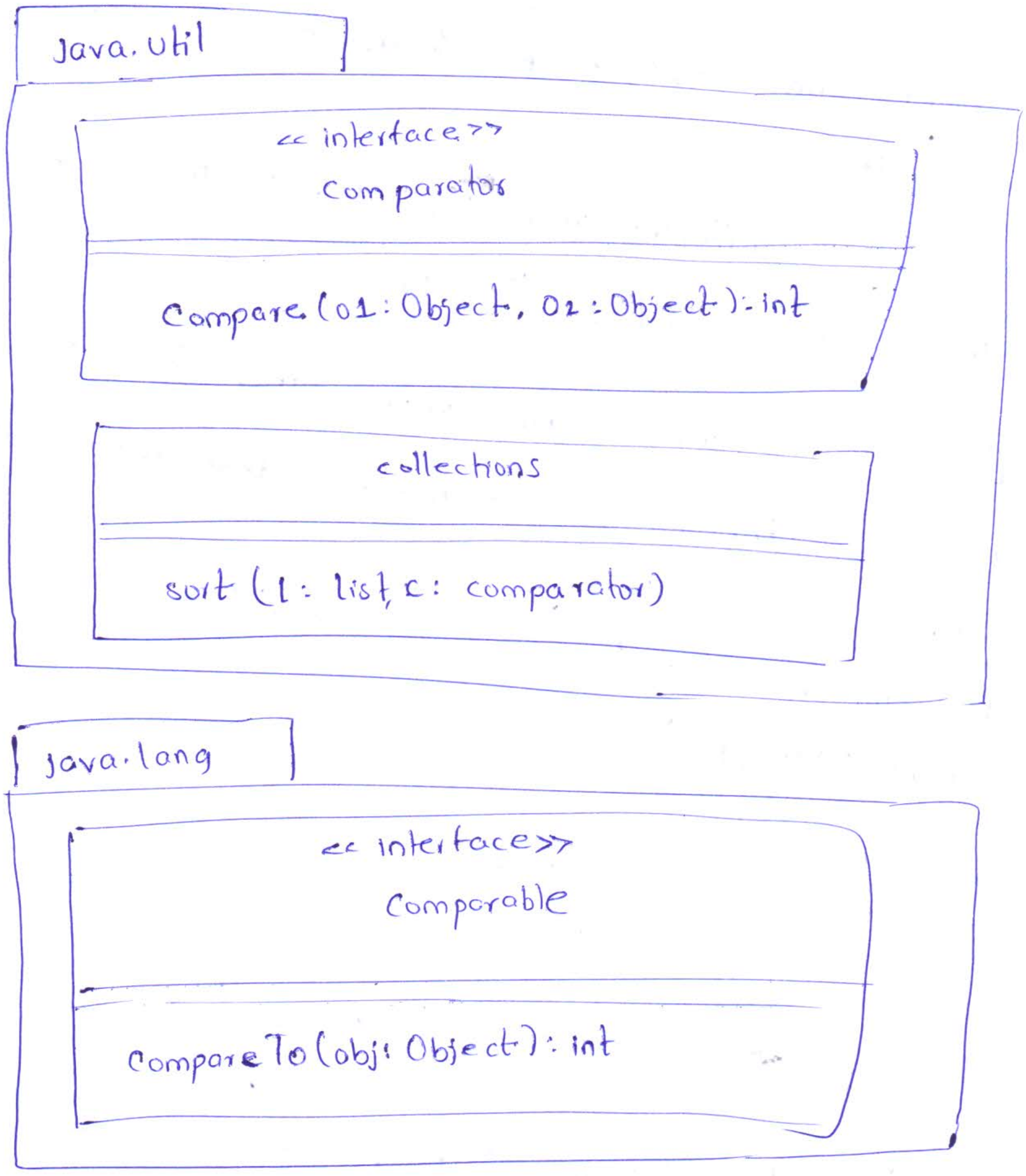


fig: the advisor interface defines an operation that various classes can implement with diff. strategies

Template method :

the intent of Template method is to implement an algorithm in a method, deferring the definition of some steps of the algorithms so that other classes can redefine them.

A classic example : Sorting :



Java.util
<< interface>>
Comparator
Compare(o1: Object, o2: Object): int
collections
sort (l: list, c: comparator)
java.lang
<< interface>>
Comporable
CompareTo(obj: Object): int

Visitor:

The intent of visitor is to let you define a new operatio for a hierarchy without changing the hierarchy classes.

visitor is most often layered over a composite, you may want to review that pattern, as well use it throughout this chapter.

Visitor mechanics:

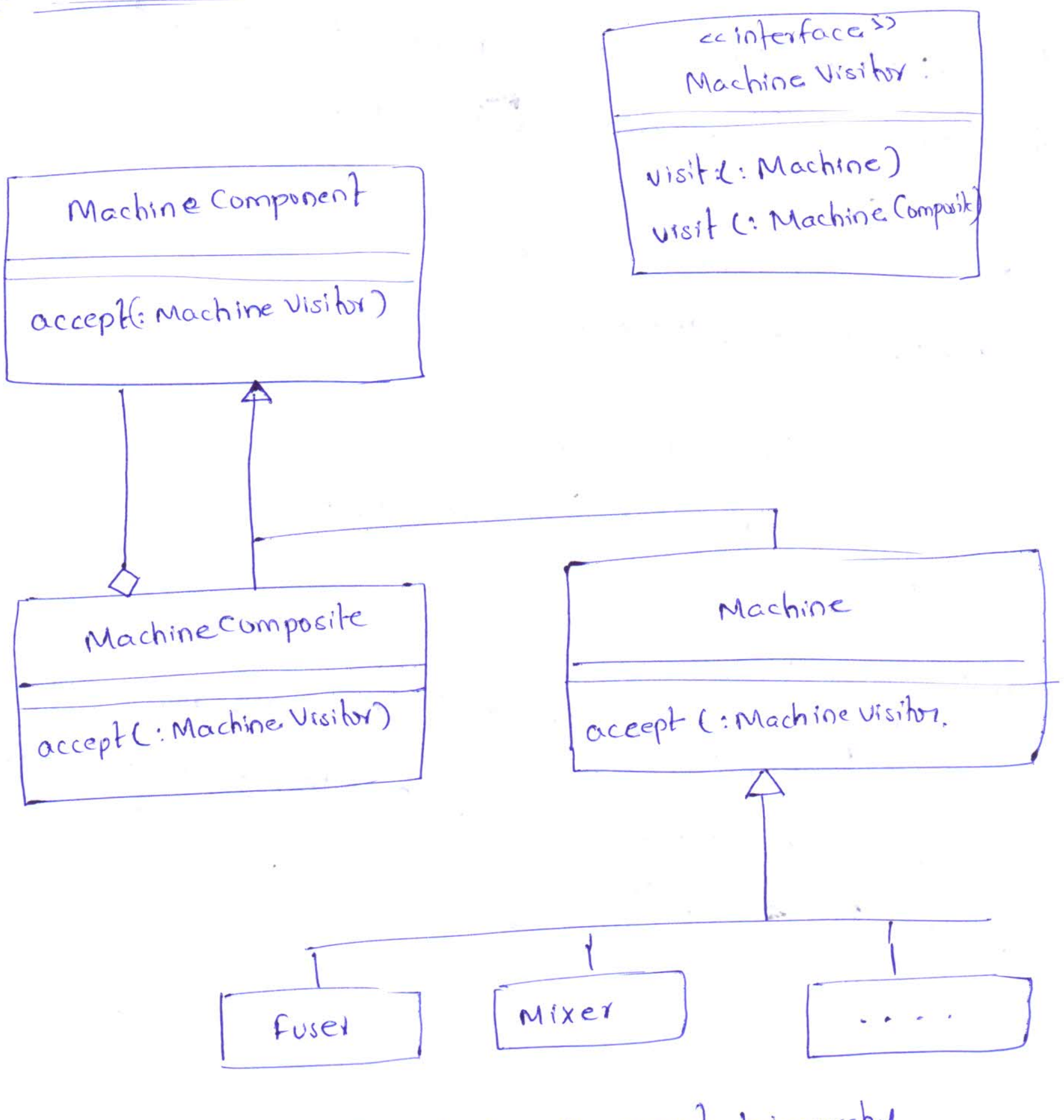


fig: Machine Component hierarchy

the accept method in the machine Component class is abstract
both subclasses this method with exactly the same code:

```
public void accept (Machine Visitor v)
    {
      v.visit(this);
    }
```

the machine Visitor interface requires implements to define methods for visiting machines & machine composites.

```
package com.oozinoz.machine;
public interface MachineVisitor
  {
    void Visit (Machine m);
    void visit (Machine Composite mc);

  }
```

<u>An ordinary visitor</u>:
Displaying the factory's machines requires building an instance of machine tree model from the factory composite & wrapping this model in swing components..

Discussion of Behavioural patterns

Encapsulating variation is a theme of many behavioural patterns

→ the patterns usually define an abstract class that describes the encapsulating object. & the pattern derives its name from that object

Ex:

→ a strategy object encapsulates an algorithm

→ a state object encapsulates a state-dependent behaviour

→ an Iterator object encapsulates the way you access & traverse the components of an aggregate object.

Objects as arguments: Several patterns introduce an object that's always used as argument.

→ It can be encapsulated or distributed

→ Decoupling senders and receivers.