Supervision 3

1. (1) Static fields and methods are used by "className.fieldName/methodName", but non-static ones are used by "objectName.fieldName/methodName".

   (2) Static methods only have access to static fields and methods, but non-static methods also have access to non-static fields and methods.

2. "==" compares whether two references point to the same object.

   There is a string constant pool in Java. At compilation, for string constants, compiler will check whether it is in the constant pool. If not, put it in the pool. "new String()" will acquire a chunk of memory in the heap to store the reference to the constant pool. But addresses stored in the stack still remain different.

   In the first case, s1 and s2 point to two different chunks of memory so obviously (s1 == s2) returns false. In the second case, s3 and s4 point to the same constant string in string constant pool, so (s3 == s4) returns true.

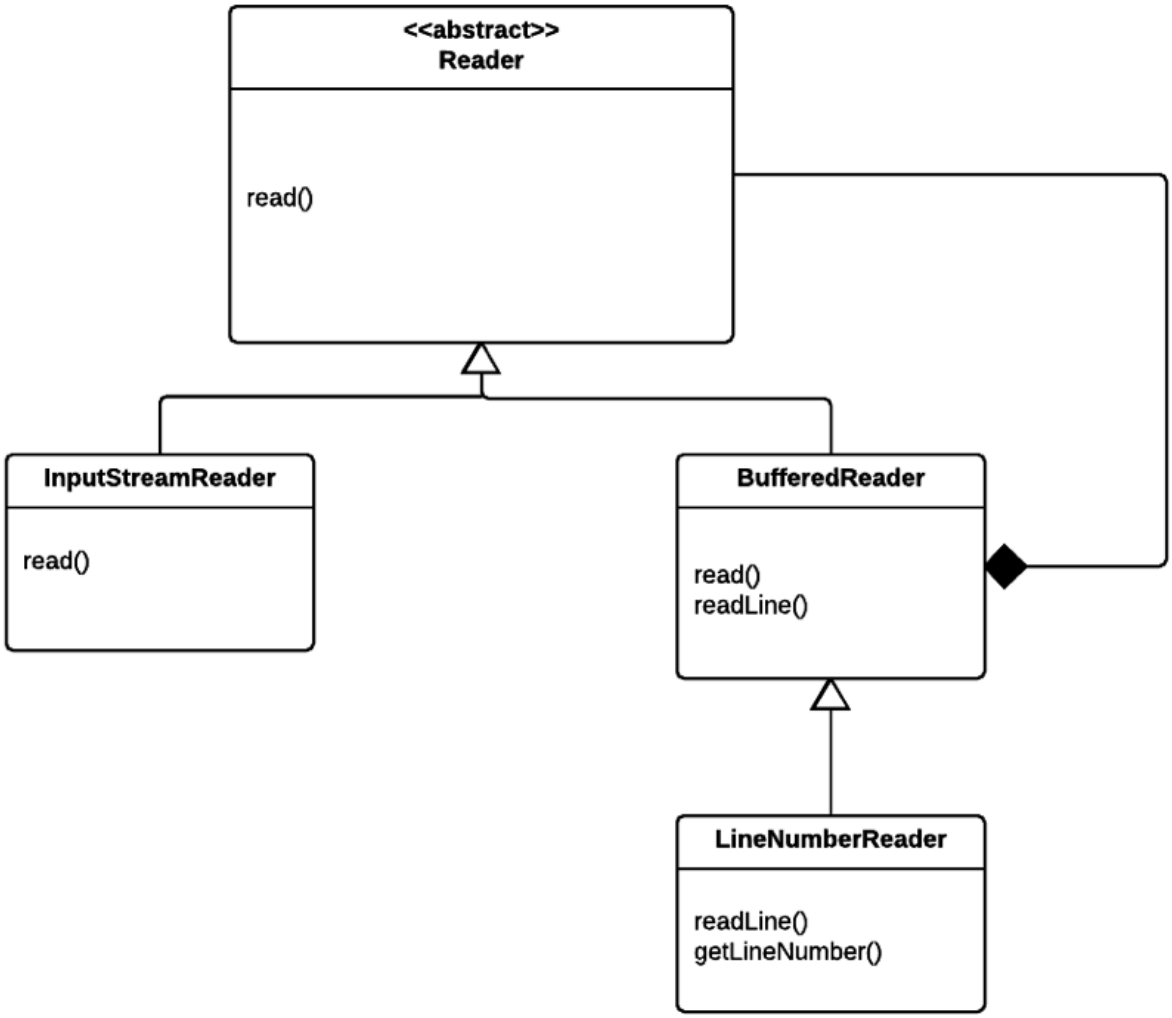
3.

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

public class Car {
    private String manufacturer;
    private int age;

    Car(String manufacturer, int age){
        this.manufacturer = manufacturer;
        this.age = age;
    }

    @Override
    public String toString() {
        return "{" +
            " manufacturer='" + manufacturer +
"'" +
            ", age='" + age + "'" +
            "}";
    }

    public static void main(String[] args) {
        List<Car> list = new ArrayList<>();
        list.add(new Car("Benz", 10));
        list.add(new Car("Benz", 8));
```

```
        list.add(new Car("Porsche", 3));
        list.add(new Car("Ford", 4));
        list.add(new Car("Audi", 15));
        Collections.sort(list, new
Comparator<Car>() {
            @Override
            public int compare(Car a, Car b){
                int v =
a.manufacturer.compareTo(b.manufacturer);
                if(v != 0)
                    return v;
                return Integer.compare(a.age,
b.age);
            }
        });
        for(Car car: list)
            System.out.println(car);
    }
}
```

4. State pattern is to let an object alter its behaviours when its internal state changes. It encapsulates state-dependent behaviours. Strategy pattern is to select an algorithm implementation at runtime. It encapsulates algorithms.
5.

```
public static void main(String[] args) throws
IOException {
    File book = downloadBook();

    try (BufferedReader is =
        new BufferedReader(  // decorator
            new InputStreamReader( // adapter
                new GZIPInputStream( //
decorator
                        new
FileInputStream(book)
                ), StandardCharsets.UTF_8))) {
      String line;
      while ((line = is.readLine()) != null) {
        System.out.println(line);
      }
    }
  }
```



Here Reader would be the Component. InputStreamReader would be the Concrete Component. BufferedReader is the Decorator which contains instance of Reader. And LineNumberReader is a Function Decorator which adds extra readline() behaviour.

6. Singleton pattern is to ensure only one instance of an object is created by developers using our code.

```
import java.util.ArrayList;
import java.util.List;
```

```
import java.util.Map;

interface Processor {

  void debit(int account, int amount);
}

class CreditCardProcessor implements Processor
{

  private static CreditCardProcessor instance
= new CreditCardProcessor();

  private CreditCardProcessor() {}

  public static CreditCardProcessor
getInstance() {
    return instance;
  }

  @Override
  public void debit(int account, int amount) {
    System.out.printf("%d owes me %d in the
future%n", account, amount);
  }
}
```

```
class DebitCardProcessor implements Processor
{

  private static DebitCardProcessor instance =
null;

  private DebitCardProcessor() {}

  // not thread-safe - more in further java
  public static DebitCardProcessor
getInstance() {
    if (instance == null) {
      instance = new DebitCardProcessor();
    }
    return instance;
  }

  @Override
  public void debit(int account, int amount) {
    System.out.printf("%d owe me %d right
now", account, amount);
  }
}

class TestProcessor implements Processor {

  @Override
```

```
  public void debit(int account, int amount)
{}
}

class Item {
  private final String name;
  private final int cost;

  Item(String name, int cost) {
    this.name = name;
    this.cost = cost;
  }

  public int getCost() {
    return cost;
  }
}

public class ShoppingCart {
  private List<Item> items;
  private int accountNumber;
  private Processor processor;

  private ShoppingCart(int accountNumber,
Processor processor) {
    this.accountNumber = accountNumber;
    this.processor = processor;
```

```
    this.items = new ArrayList<>();
  }

  static ShoppingCart create(Map<String,
Processor> processors) {
    return new ShoppingCart(12345,
processors.get("credit"));
  }

  void purchase() {
    for (Item item : items) {
      processor.debit(accountNumber,
item.getCost());
    }
  }

  public static void main(String[] args) {
    // production mode
    ShoppingCart a =
        ShoppingCart.create(
            Map.of(
                "credit",

 CreditCardProcessor.getInstance(),
                "debit",

 DebitCardProcessor.getInstance()));
```

```
    // test mode
    ShoppingCart b =
        ShoppingCart.create(
            Map.of(
                "credit", new TestProcessor(),
                "debit", new
TestProcessor()));
  }
}
```

We want to use credit card or debit card to make purchase. For security reasons, we hope only one instance is created when processing. To apply Open-Closed Principle, we let CreditCardProcessor and DebitCardProcessor implement Processor interface. And within class, we set the constructor as private so that we can't create instances outside the class. And we define a static instance inside the class and that should be the only instance of the class and can only be accessed by calling getInstance().

Advantages:

(1) Instance control: Singleton prevents other objects from instantiating their own copies of the Singleton object, ensuring that all objects access the single instance.

(2) Lazy instantiation support: The instance of Singleton class can be created only when it's called. This saves memory and other resource costs.

Disadvantages:

(1) Test inconvenience: Singleton pattern only allows one instance so you have to write a lot more extra code to test your program.

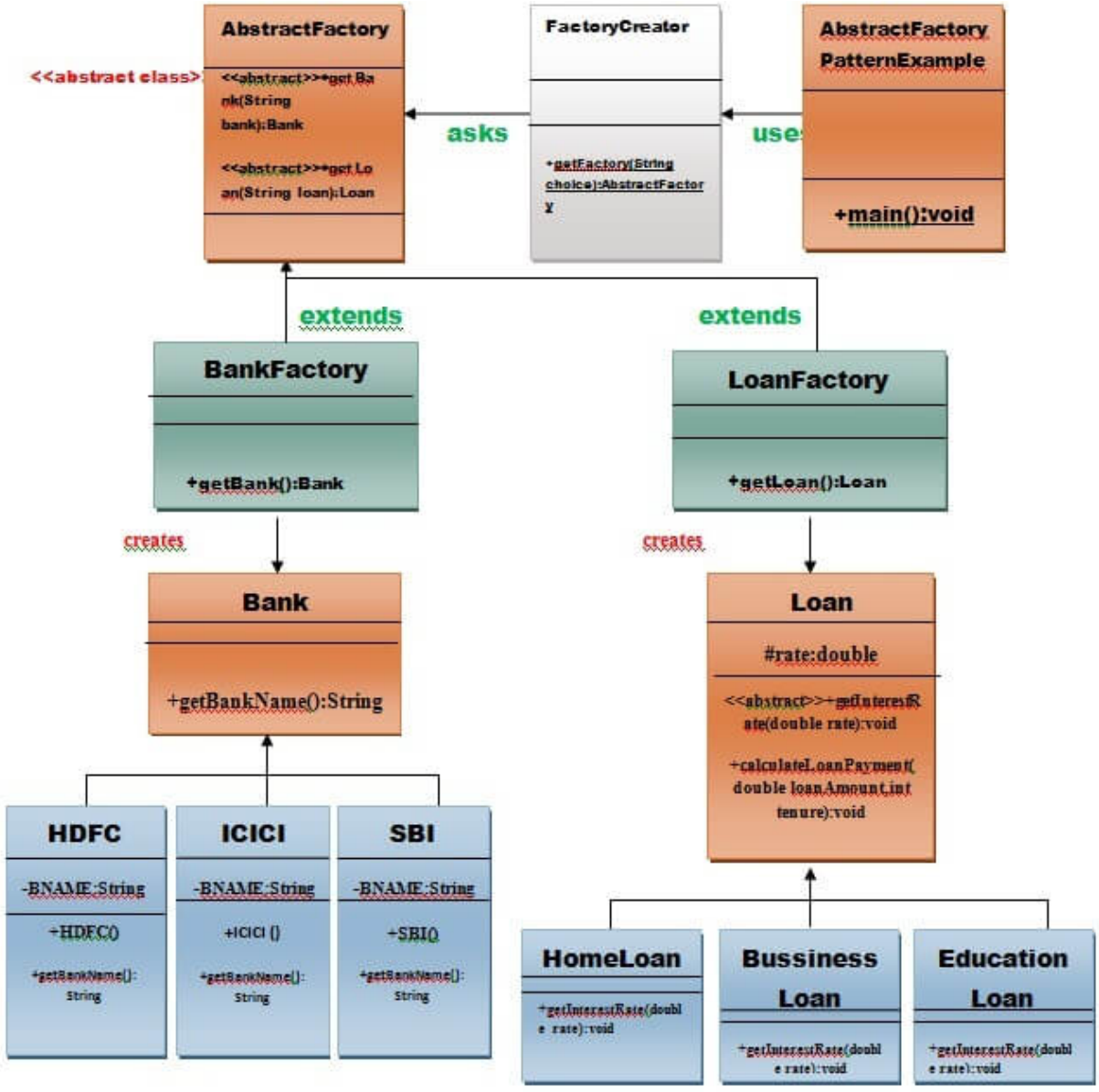
(2) Lazy instantiation is not thread-safe.

(3) You can't pass any parameter to the constructor of the Singleton pattern so the flexibility is limited.

7. **Abstract Factory Pattern**

   An interface or abstract class can be used to represent a family of related classes that can have related objects. For example, Car can be an interface. And it can represent different brands (classes) such as Audi or Porsche. And a factory can generate objects of concrete classes whichever we want, for example we want a Benz car.

An abstract factory is an interface for creating different factories. For example, factory for cars and factory for planes.



(1) We are going to create interfaces or abstract classes as well as their subclasses (implementations).

(2) Create AbstractFactory class.

(3) Create concrete factories that extends Abstract

Factory class.

(4) We can define a FactoryCreator class to have instances of different factories.

(5) We can use instances of different factories to get different objects.

Here is an example of calculating the loan payment for different loan modes and for different banks.

```
import java.io.*;

interface Bank{
    String getBankName();
}

class HDFC implements Bank {
    private final String BNAME;

    public HDFC() {
            BNAME="HDFC BANK";
    }

    public String getBankName() {
        return BNAME;
    }
}
```

```
class ICICI implements Bank {
    private final String BNAME;

    public ICICI() {
            BNAME="ICICI BANK";
    }

    public String getBankName() {
        return BNAME;
    }
}

class SBI implements Bank {
    private final String BNAME;

    public SBI() {
            BNAME="SBI BANK";
    }

    public String getBankName() {
        return BNAME;
    }
}

abstract class Loan {
    protected double rate;
```

```
    abstract void getInterestRate(double rate);

    public void calculateLoanPayment(double
loanamount, int years) {
        double EMI;
        int n;

        n=years*12;
        rate=rate/1200;
        EMI=
((rate*Math.pow((1+rate),n))/((Math.pow((1+rate)
,n))-1))*loanamount;

        System.out.println("your monthly EMI is
"+ EMI +" for the amount"+loanamount+" you have
borrowed");
    }
}

class HomeLoan extends Loan {
    public void getInterestRate(double r) {
        rate=r;
    }
}

class BussinessLoan extends Loan {
```

```
    public void getInterestRate(double r) {
        rate=r;
    }
}

class EducationLoan extends Loan {
    public void getInterestRate(double r) {
        rate=r;
    }
}

abstract class AbstractFactory {
    public abstract Bank getBank(String bank);

    public abstract Loan getLoan(String loan);
}

class BankFactory extends AbstractFactory{
    public Bank getBank(String bank){
        if(bank == null){
            return null;
        }
        if(bank.equalsIgnoreCase("HDFC")){
            return new HDFC();
        } else
if(bank.equalsIgnoreCase("ICICI")){
            return new ICICI();
```

```
        } else if(bank.equalsIgnoreCase("SBI")){
            return new SBI();
        }
        return null;
    }

    public Loan getLoan(String loan) {
        return null;
    }
}

class LoanFactory extends AbstractFactory{
    public Bank getBank(String bank){
        return null;
    }

    public Loan getLoan(String loan){
        if(loan == null){
            return null;
        }
        if(loan.equalsIgnoreCase("Home")){
            return new HomeLoan();
        }
        else
if(loan.equalsIgnoreCase("Business")){
            return new BussinessLoan();
```

```
        } else
if(loan.equalsIgnoreCase("Education")){
            return new EducationLoan();
        }
    return null;
    }
}

class FactoryCreator {
    public static AbstractFactory
getFactory(String choice){
        if(choice.equalsIgnoreCase("Bank")){
            return new BankFactory();
        } else
if(choice.equalsIgnoreCase("Loan")){
            return new LoanFactory();
        }
        return null;
   }
}

public class AbstractFactoryExample {
    public static void main(String args[])throws
IOException {
        BufferedReader br=new BufferedReader(new
InputStreamReader(System.in));
```

```
        System.out.print("Enter the name of Bank
from where you want to take loan amount: ");
        String bankName=br.readLine();

        System.out.print("\n");
        System.out.print("Enter the type of loan
e.g. home loan or business loan or education
loan : ");

        String loanName=br.readLine();
        AbstractFactory bankFactory =
FactoryCreator.getFactory("Bank");
        Bank b=bankFactory.getBank(bankName);

        System.out.print("\n");
        System.out.print("Enter the interest
rate for "+b.getBankName()+ ": ");

        double
rate=Double.parseDouble(br.readLine());
        System.out.print("\n");
        System.out.print("Enter the loan amount
you want to take: ");

        double
loanAmount=Double.parseDouble(br.readLine());
        System.out.print("\n");
```

```
        System.out.print("Enter the number of
years to pay your entire loan amount: ");
        int
years=Integer.parseInt(br.readLine());

        System.out.print("\n");
        System.out.println("you are taking the
loan from "+ b.getBankName());

        AbstractFactory loanFactory =
FactoryCreator.getFactory("Loan");
        Loan l=loanFactory.getLoan(loanName);
        l.getInterestRate(rate);

 l.calculateLoanPayment(loanAmount,years);
    }
}
```

8. (1) Usage of null references.

   Disadvantages:

   a. Every time you get an object as an input you must check whether it is null or a valid object reference. Your program would become complicated and hard to maintain.

b. Ambiguity is inevitable for a code reader since they may not be aware of the possibility of null references.

c. From the view of an object, it makes less sense to see a null reference. An exception or an object representing null looks more reasonable.

(2) Making objects mutable.

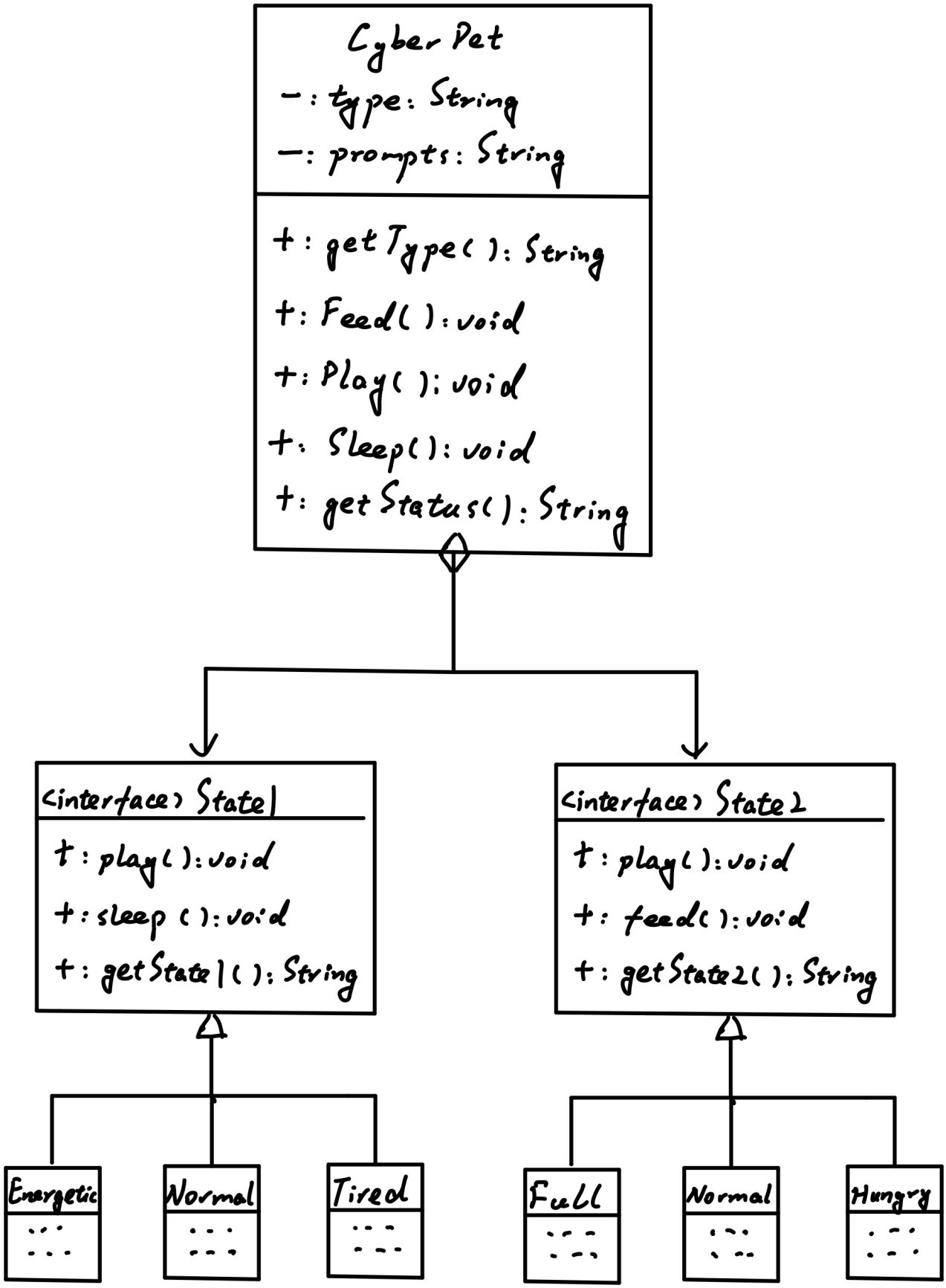
Disadvantages:

a. Its leads to side effects that lead to bugs and maintainability issues.

b. If objects are used as keys in maps, when being modified, we could lose an entry in the map which is annoying and difficult to debug.

c. If it throws an exception in the middle of the modification, the consistency of states would be broken.

9.



0.

```
import java.util.Scanner;
```

```
class TooTired extends Exception{
    public TooTired(String msg){
        super(msg);
    }
}

class TooHungry extends Exception{
    public TooHungry(String msg){
        super(msg);
    }
}

class TooFull extends Exception{
    public TooFull(String msg){
        super(msg);
    }
}

public class CyberPet {
    private static final String begin =
"Hello! I'm your pet. Please choose the type
of your pet.";
    private static final String prompt =
"Press 1 to feed. Press 2 to play. Press 3 to
let your pet sleep. Press q to quit.";
```

```
    private String type;

    static Scanner scanner = new
Scanner(System.in);

    interface Status1{
        void play() throws TooTired;
        void sleep();
        String energy();
    }

    private Status1 state1;

    class Energetic implements Status1{
        public void play(){
            state1 = new Normal1();
        }

        public void sleep(){
            state1 = new Energetic();
        }

        public String energy(){
            return "Energetic";
        }
    }
```

```
    class Normal1 implements Status1{
        public void play(){
            state1 = new Tired();
        }

        public void sleep(){
            state1 = new Energetic();
        }

        public String energy(){
            return "Normal";
        }
    }

    class Tired implements Status1{
        public void play() throws TooTired{
            throw new TooTired("Tired to death
lol.");
        }

        public void sleep(){
            state1 = new Normal1();
        }

        public String energy(){
            return "Tired";
        }
```

```
    }

    interface Status2{
        void play() throws TooHungry;
        void feed() throws TooFull;
        String food();
    }

    private Status2 state2;

    class Full implements Status2{
        public void play(){
            state2 = new Normal2();
        }

        public void feed() throws TooFull{
            throw new TooFull("Eat too
much!");
        }

        public String food(){
            return "Full";
        }
    }

    class Normal2 implements Status2{
        public void play(){
```

```
            state2 = new Hungry();
        }

        public void feed(){
            state2 = new Full();
        }

        public String food(){
            return "Normal";
        }
    }

    class Hungry implements Status2{
        static final String food = "Hungry";

        public void play() throws TooHungry{
            throw new TooHungry("Hungry to
death lol.");
        }

        public void feed(){
            state2 = new Normal2();
        }

        public String food(){
            return "Hungry";
        }
```

```
    }

    CyberPet(String type){
        this.type = type;
        this.state1 = new Energetic();
        this.state2 = new Full();
    }

    String getType(){
        return type;
    }

    void Feed() throws Exception{
        state2.feed();
    }

    void Play() throws Exception{
        state1.play();
        state2.play();
    }

    void Sleep() throws Exception{
        state1.sleep();
    }

    String getStatus(){
```

```
        return state1.energy() + " " +
state2.food();
    }

    public static void main(String[] args)
throws Exception{
        System.out.println(begin);

        String type = scanner.nextLine();
        CyberPet pet = new CyberPet(type);

        while(true){
            System.out.println(pet.getType() +
" " + pet.getStatus());
            System.out.println(prompt);
            int p =
Integer.parseInt(scanner.nextLine());

            if(p == 1)
                pet.Feed();
            else if(p == 2)
                pet.Play();
            else if(p == 3)
                pet.Sleep();
            else break;
        }
    }
```

```
}
```