Comparison of Declarative and Stateful

First of all, we need to know what it means for declarative, it's like telling you: "What do I want to do" or what to do. In my opinion, declarative can be regarded as an expression corresponding to a higher level of abstraction. Under this understanding, the SQL language as a domain-specific language (DSL) is declarative. On the other hand, my understanding of declarative style is: separation of expression and operation. Expressed that it can not be executed, or even cannot be executed.

```
list = range(0, Infinity);
   list.take(5)
```

for list= range(0, infinity) // Get an array of all integers from 0 to infinity
list. take (5) // Take the first 5 records of the array

It also seems to represent a value that can be obtained in the future. And when we have not really got this value, we can return it as a return value and pass it back and forth in the program as a parameter. Like an estimate of the future

In the traditional imperative programming concept, the return of a function means the execution is complete, and we cannot define the "future value"

```
async function asyncValue(){
    return new Promise((resolve, reject) => {
        setTimeout(() => resolve('a'), 1000)
     });
  }

  var result = asyncValue();
  doSomething(result);

  async function doSomething(input){
     console.log('begin');
     var result = await input;
     console.log('1 second later: result='+result);
  }
```

Although the value of the future has not come yet, the future can be expected after all. But if you don't know whether the future will come, can you assign a form of expression to it?

```
var numbers = [1,2,3,4,5]
var doubled = numbers.map (function (n) {
  return n * 2
})
console.log (doubled) //=> [2,4,6,8,10]
```

If you have a certain programming foundation, you should be familiar with map and reduce. We are very used to pointing out how things should work. For example, "to traverse this list", "if, then". You should start thinking for yourself: Why should we learn this when we can directly tell the computer how to do it?

Perhaps in many situations, the imperative is useful. When doing homework, you only need to use imperatives to complete 80% of your homework.

However, if we take the time to learn declarative styles, we will find that they can bring great convenience to our programming.

First of all, I can write less code and write more simply. This is the shortcut to success. And they allow us to think on a higher level,

```
SELECT * from dogs
INNER JOIN owners
WHERE dogs.owner_id = owners.id
```

for this one, If you can use SQL to write code, you are going to find it's strength. compared to a normal one. you will feel like: " oh, it's easier, reduce my time!"

```
//dogs = [{name: 'Fido', owner_id: 1}, {...}, ... ]
//owners = [{id: 1, name: 'Bob'}, {...}, ...] var dogsWithOwners = []
var dog, owner
for(var di=0; di < dogs.length; di++) {
  dog = dogs[di]
  for(var oi=0; oi < owners.length; oi++) {
    owner = owners[oi]
    if (owner && dog.owner_id == owner.id) {
      dogsWithOwners.push ({
        dog: dog,
        owner: owner
      })
    }
  }}
}
```

So, to summarize the advantages of declarative: easy to manage code and save time.
Disadvantages: Declarative transactions have a limitation. If we want to add transactions to a part of the code block, we need to separate this part of the code block as a method. Easy to lose the original function

However, because of this granularity issue, I do not recommend excessive use of declarative transactions, but it is recommended to use it if it can be used, according to personal needs.

First of all, because declarative transactions are implemented through annotations, and sometimes they can also be implemented through configuration, this will cause a problem, that is, this transaction may be ignored by developers.

Stateful

First of all, stateful, as the name suggests, you should almost understand. It can represent the state in a class, but not only the state, but also the behavior.

Use the simplest example in life:
You should understand these states of aircraft, but I believe you still don't quite understand how to implement specific codes.
Airplanes are: flying, ready to fly, no flight and no power.

You should also understand that you cannot fly when "without power".

You should also understand that you cannot fly again while "flying".

Because of this, we can directly implement different methods in different state subclasses. For example, when "no power" is used, if the "takeoff" member function is used, it will prompt that there is no power and takeoff failed.

However, when there is no flight, you can use the "take off" member function.

```
class State
{
public:
    State()
    {

    };
    virtual ~State()
    {

    };
    virtual void fly()=0;
    virtual void stopfly()=0;
    virtual void up()=0;
    virtual void down()=0;
```

this is basic class for a state, I think everyone can understand if you learned computer science before

```
1 //state: flying
2 class StateFlying : public State
3 {
4 public:
5    StateFlying()
6    {
7
8    };
9     virtual ~StateFlying()
10   {
```

```
11
12      };
13       virtual void fly()
14      {
15           //plane is flying , u cannot fly
16           cout << "ERROR: Can't do this!! You are not 'stopfly'! " <<endl;
17       };    //take off
18       virtual void stopfly()
19      {
20           //Of course you can land while flying
21           cout << "INFO: OK! stopfly" << endl;
22       };    //stop flying , land!
23       virtual void up()
24      {
25           //now you can do that
26           cout << "INFO: OK! up" << endl;
27       };    //Increase flight altitude
28       virtual void down()
29      {
30           //you can do that
31           cout << "INFO: OK! down" << endl;
32       };    //Lower the flying height
33  };
34
35  //State category: shutdown status
36  class StateNotFly : public State
37  {
38  public:
39      StateNotFly()
40      {
41
42      };
43       virtual ~StateNotFly()
44      {
45
46      };
47       virtual void fly()
48      {
49           //Because it is in a stopped state, it can take off
50           cout << "INFO: OK! " << endl;
51       };    //take off
52       virtual void stopfly()
53      {
54           //Can you shut down after shutting down?
55           cout << "ERROR: Can't do this!! You are not 'Flying'! " <<endl;
56       };    //stop flying, land
57       virtual void up()
58      {
59           //Let's forget about these aerial actions when the machine is down
60           cout << "ERROR: Can't do up!! You are not 'Flying'! " <<endl;
61       };    //Increase flight altitude
62       virtual void down()
63      {
64           //Let's forget about these aerial actions when the machine is down
65           cout << "ERROR: Can't do down!! You are not 'Flying'! " <<endl;
66       };    //Lower the flying height
67  };
68
69  //state: no power
```

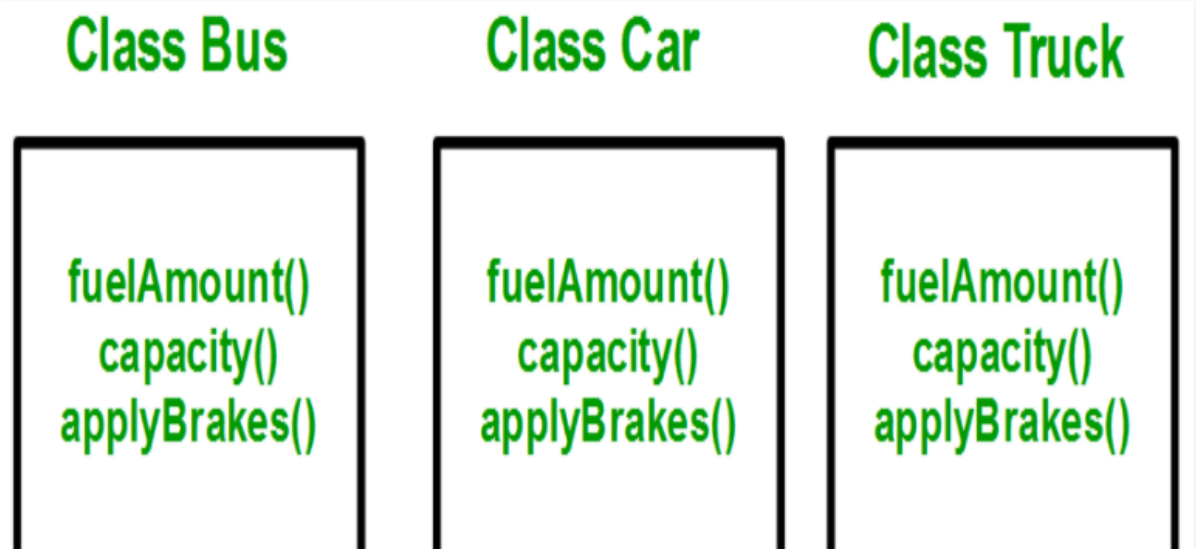
```
70 class StateNotPower : public State
71 {
72 public:
73    StateNotPower()
74    {
75
76    };
77     virtual ~StateNotPower()
78    {
79
80    };
81     virtual void fly()
82    {
83         //No power, unable to take off!
84         cout << "ERROR: Can't fly!! No Power!   " <<endl;
85     };    //take off
86     virtual void stopfly()
87    {
88         //no power, just land
89         cout << "INFO: OK! stopfly" << endl;
90     };    //stop flying just land
91     virtual void up()
92    {
93         //no power
94         cout << "ERROR: Can't take off! No Power!   " <<endl;
95     };    //Increase flight altitude
96     virtual void down()
97    {
98         //I don't have any power, but I can glide down.
99         cout << "INFO: OK! down" << endl;
100     };    //Lower the flying height
101 };
102
103 /*
```

In this way, our three states are fine. You can also see one thing from the code, each state has its own specific processing. So, do you understand now? State mode. You can reuse it easily. You can even define N states and N different airplanes, and the airplanes and states can be reused easily!

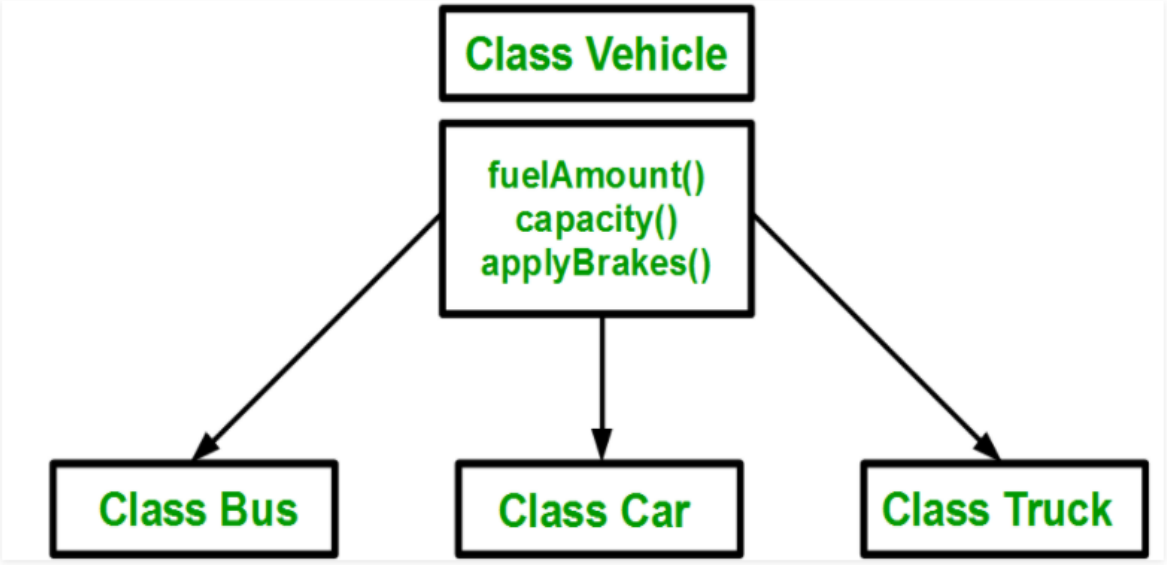
Because the state inherits the State abstract class, and the flight only uses and defines the state, and does not need to manage different behaviors in different states.
But if you look at it from the opposite side, we have to use a lot of template files, and a lot of repeated codes to describe the state.

Inheritance

I believe that the students may be more or less exposed to "inheritance" in the previous courses. For example, it often appears in C++. for example:



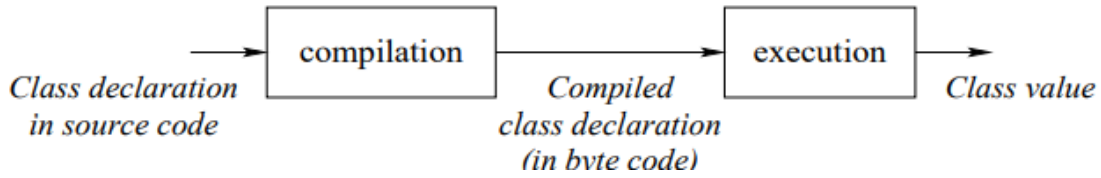
From here we can see that the same code above is repeated 3 times. This increases data redundancy. So how to avoid it? I recommend using inheritance. For example, we create a class like: Vehicle and then code in the vehicle class. Then we can get:



With this method, we only need to write the function once, which greatly reduces the time we spend. Because we have inherited the other three classes from (Vehicle).

So how to define inheritance in the Oz language? First of all, we first understand the usage of a class in Oz: it can be inherited from one or more classes. Use some keywords to indicate inheritance
From the class declaration. At the same time, the inheritance of a class can also be called single inheritance. We can also call more than one class multiple inheritance.

For example: C appears in the from statement of A.
• C is the superclass of the class that declares A.



we can use this chain to declare their relation. Like the superclass, we can also call it a directed graph. Because inheritance is legal, there are two requirements. 1. Inheritance
The relationship is directed and acyclic. Therefore, the following situations are not allowed:

```
class A from B ... end
class B from A ... end
```

Secondly, after deletion, it should be unique and listed as a separate class.
C in the following example is illegal because the two methods marked m still exist:

```
class A1 meth m(...) ... end end
class B1 meth m(...) ... end end
class A from A1 end
class B from B1 end
class C from A B end
```

Through the explanation in the book, we can know: The Mozart system blurs the difference between runtime and compile time. Because the compiler is also a part of the runtime system. So the declaration of the class is an executable statement, and a class is created by compiling and executing it. Then we can use it to pass the value to the new object. I believe that you have also learned to classify and partition multiple types in the previous study, and know how to optimize.

In order to use inheritance correctly, we also need to know: Oz basically will not find the problem of restricting multiple inheritance, because Oz will enforce a method to require that the methods defined in multiple superclasses must be locally Define the method to override the method that caused the conflict.

When superclasses share a common ancestor class that is stateful (that is, with attributes), there is another problem with multiple inheritance: people will repeat the same operations during initialization when we have to initialize. This problem is called the realization sharing problem.

In Oz, we did not directly define the protected methods. We can solve this problem privately through inheritance.

```
class C from ...
  attr pa:A
  meth A(X) ... end
  meth a(...) {self A(5)} ... end
  ...
end
```

As you can see, We created a subclass C1 to access method of A.

```
class C1 from C
  meth b(...) L=@pa in {self L(5)} ... end
  ...
end
```

This allows method b to access method A in this way.

Static and dynamic binding

When executing inside an object, we will consider using another object by calling another object's method. It looks like a kind of recursion in C++. Some students might think that recursion would be simple. But when it comes to inheritance, it becomes complicated.
We defined that the purpose of inheritance is a new ADT that is extended with the existing ADT. It sounds very simple, but if you want to achieve it, you must use static and dynamic binding.

example:
We have created an account class to carry out the transfer function

```
class LoggedAccount from Account
   meth transfer(Amt)
      {LogObj addentry(transfer(Amt))}
      ...
   end
end
```

```
LogAct={New LoggedAccount transfer(100)}
```

To pass batchTransfer, a new method must be used in LoggedAccount.
This is called dynamic binding. This can be kept open using dynamic binding and guarantees the possibility that the account can be expanded

inherit. It can be guaranteed that the new ADT is inherited from the old ADT, that is to say, this new one has all the functions of the old one and has been developed for more functions.

```
class LoggedAccount from Account
   meth transfer(Amt)
```



```
            {LogObj addentry(transfer(Amt))}
            Account,transfer(Amt)
         end
      end
```

What is static binding? When shipping with a new account, we cannot use transfer, which will automatically be bound to new.
So we use Account, transfer (Amt)
Find out the method transmission in the Account of the class.

Static and dynamic binding is required when overriding methods through inheritance. This dynamic binding allows the new ADT to extend the old one normally
ADT, and you can also call new methods

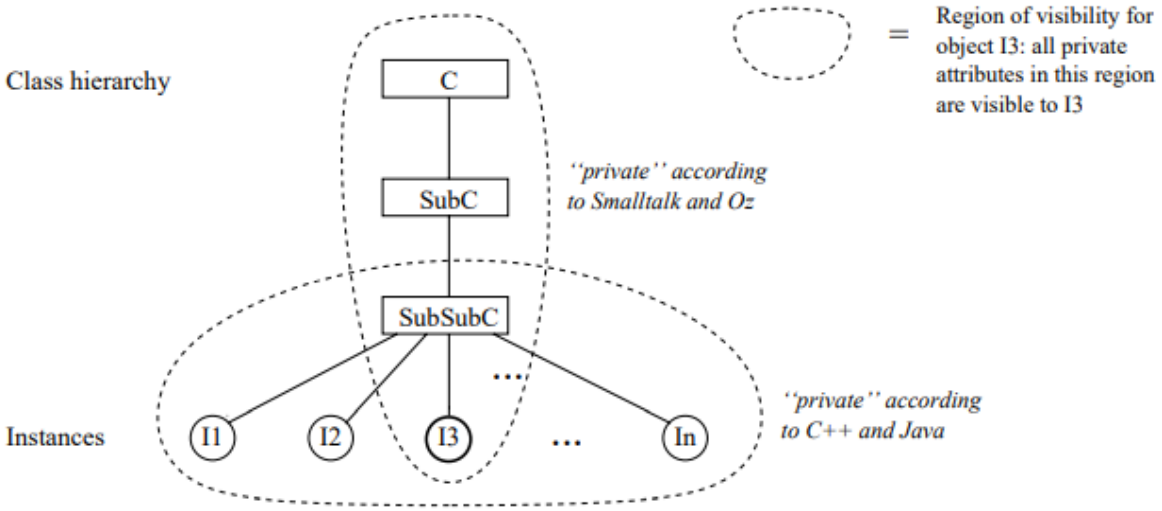
Here I will give a separate explanation of the two bindings:

Dynamic binding: {self M}. Means to match the M visible in the current object. And also take into account
The coverage has been completed. Dynamic means to change.
Static binding: C, M, C are expression classes
The method of matching M is defined. Like binding C and M together. They can no longer make changes.

Controlling encapsulation
A very important point in object-oriented languages.
for example. In life, we need structural composition to build a house: steel bar, concrete, cement, sand, stone, brick

Decoration composition: ceramic tiles, paint, wallpaper, doors and windows, etc.

Understanding encapsulation from the perspective of the interface designer. I think the printf() function in C language is a classic example.
In other words, put the functions of the same function in a class for unified management



Private and public scopes
Is private and public,
Private members are members that are only visible in the class.
Public members are members that are visible anywhere in the program.
In Oz, attributes are private and methods are public. This is very similar to other languages

Constructing other scopes

Two concepts: lexical scope and name value. Private and public scopes.
Scopes can also use name values to represent the same meaning.
For example,
Private and protected ranges contained in the most familiar C++

Private method
In C++, when the method header is a name value, its scope is limited to all instance classes, which is completely private.

```
class C
   meth A(X)
      % Method body
   end
end
```

For example, A is bound to a name. This variable A is only visible inside
But an instance of C can call method A in any other instance.
But C. Method A is not visible to the subclass definition. . It corresponds to private

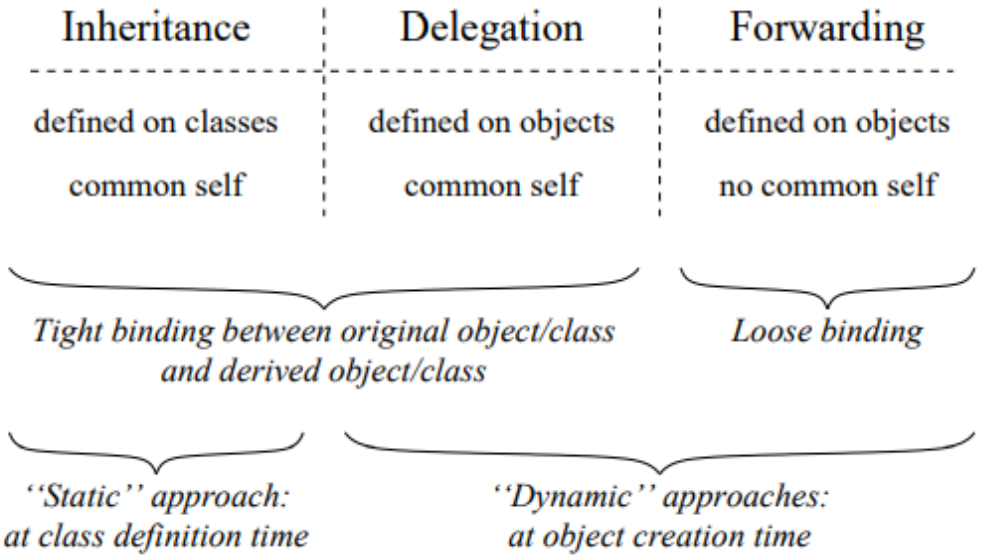
```
local
   A={NewName}
in
   class C
      meth !A(X)
         % Method body
      end
   end
end
```

This is a same method, but in different variables. When we define, put the name directly at the front, and then the method header is bound to it.

Protected method
This can basically be called "public". For example, in C++, the class and descendants that define it are protected

```
class C
   attr pa:A
   meth A(X) skip end
   meth foo(...) {self A(5)} end
end
```

Attribute scopes
The attributes are private. If you want to make it public, you can only give access to its attributes, which can be operated by dynamic type.



This picture clearly illustrates the specific relationship between inheritance, delegation and forwarding

Delegation

Purpose of delegation: use methods as parameters

The simplest explanation of delegation: a variable.

Like character variables accept character data!

Numerical variables accept numerical data!

The delegate accept method is the value!

we always use new delegation (it's also called syntax sugar), The compiler does something to make the code look more concise

```
[Serializable, ClassInterface(ClassInterfaceType.AutoDual), ComVisible
public abstract class Delegate : ICloneable, ISerializable
{
    // Fields
    internal object _methodBase;
    [ForceTokenStabilization]
    internal IntPtr _methodPtr;
    [ForceTokenStabilization]
    internal IntPtr _methodPtrAux;
    [ForceTokenStabilization]
    internal object _target;

    // Methods
    private Delegate();
    [SecuritySafeCritical]
    protected Delegate(object target, string method);
```

```
string str = "";

int i = 2;

public delegate void TestDelegate(string name);

TestDelegate gd;

void testfunction(string name)

{

    Console.Write(name);

}

gd = testfunction;
```

Forward:

Use to forward to another object. When some of the members are forwarded to another object. The other members are used directly.

```
local
   class ForwardMixin
      attr Forward:none
      meth setForward(F) Forward:=F end
      meth otherwise(M)
         if @Forward==none then raise undefinedMethod end
         else {@Forward M} end
      end
   end
in
   fun {NewF Class Init}
      {New class $ from Class ForwardMixin end Init}
   end
end
```