CHAPTER 1

CHAPTER 2

Experimenting with Prototypes

In this chapter we want to build with you step by step a little prototype language named Propha. In prototype kernels, a single object can have instance specific state and more fundamental instance specific behavior.

The kernel of Propha that we will build does not unify state and behavior as in Self. In particular, it means that the basic clone operation only clones state and delegation only look for behavior in another object when it does not find it in the current object. With this separation, we avoid the problem that when a parent changes its state, its sons are impacted. From that perspective Propha is closed to the model of Pepsi [**?**]. We can build other operations that would clone state as well as delegate to the cloned object.

Our implementation is based on the use of anonymous class that we can dynamically inject in the superclass chain (between an instance and its class) and where we can then compile instance specific behavior.

2.1 A glimpse at our implementation

The implementation we propose is simple: for state we will use a dictionary and for methods we will use anonymous classes.

Object state

We will model object state with a dictionary and use the traditional instance variable semantic: each object has its own value. So each prototype will have its own dictionary of variables.

Now for the behavior we will use anonymous classes. As we mentioned earlier, we can define hidden subclasses to a given class. Such subclasses have

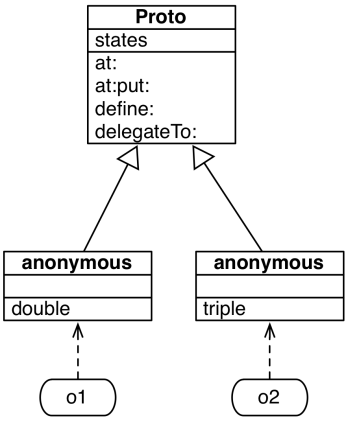

Figure 2.1 Using anonymous subclasses to have instance specific method dictionary.

all the same format for their instances and they have their own method dictionary (see Figure 2.1).

Delegation

Delegation can be easily implemented with anonymous classes. When an object delegates to another one, it means that when a method is not found in its method pool, the lookup should look in the ones of his parent or prototype. We will reuse the Pharo method dispatch and dynamically change the superclass chain as shown in Figure 2.2. When `o2` delegates to `o1`, its behavior should point to the one of `o1` (notice that in Propha state is not shared by delegation). At the implementation level, the superclass of anonymous class of `o2` will point the anonymous class of `o1`.

2.2 Starting with ExNihilo creation

There are different ways to create prototypes and our language supports two ways: the exnihilo creation and the cloning. Let us start with the exnihilo one. Let us write some tests to make sure that we do not get semantic regression during our little journey.

```
TestCase subclass: #ProphaTest
  ...
```

and a class `Proto`

```
Object subclass: #Proto
  instanceVariableNames: 'states'
  classVariableNames: ''
```

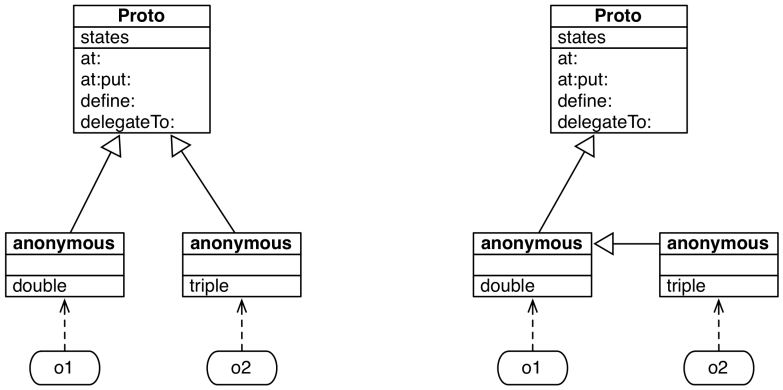

Figure 2.2 Changing superclass chain to model delegation.

```
  package: 'Proto'
```

Let first test that two exNihilo created objects have a distinct identity.

```
ProphaTest >> testExNihiloCreateDifferentObjects

  self deny: Proto exNihilo == Proto exNihilo
```

In addition, we want to make sure that each created object has its own behavior so that it can have instance specific method.

```
testExNihiloCreateDifferentImplementationClasses

  | o1 o2 |
  o1 := Proto exNihilo.
  o2 := Proto exNihilo.
  self deny: o1 == o2.
  self deny: o1 class == o2 class.
```

Let us start by defining a class method to generate an anonymous class based on the `Proto` Pharo class.

```
Proto class >> createBehavior
  "Define an anonymous subclass of Proto"
  ^ Proto newAnonymousSubclass
```

Now we can define the `exNihilo` message.

```
Proto class >> exNihilo
  "Return a prototype without link to other prototype objects."
  ^ self createBehavior new
```

About class

The test `testExNihiloCreateDifferentImplementationClasses` uses the message `class` to access to the behavior (the anonymous class in fact) and we do not really like that. So we propose to define an extra method.

```
Proto >> behavior
  ^ self class
```

```
ProtoTest >> testBehavior

  | b |
  b := Proto createBehavior.
  self assert: (b new behavior == b)
```

```
ProtoTest >> testExNihiloCreateDifferentImplementationClasses

  | p1 p2 |
  p1 := Proto exNihilo.
  p2 := Proto exNihilo.
  self deny: p1 == p2.
  self deny: p1 behavior == p2 behavior.
```

Another solution would be to define one prototype as the root of delegation similar to `Lobby` in Self. With this solution we would have to convert all the methods that we are defining the class `Proto` into such prototype. We could do that by copying all the methods.

2.3 State

The following snippet shows that we create two objects from nothing and that we can set to them different values and retrieve them.

```
| o1 o2 |
o1 := Proto exNihilo.
o1 at: #foo put: 42.
o1 at: #foo.
>>> 42

o2 := Proto exNihilo.
o2 at: #foo put: 33.
o2 at: #foo.
>>> 33
```

Let us turn that into a test, of course!

```
ProtoTest >> testExNihiloCreation

  | o1 |
  o1 := Proto exNihilo.
  o1 at: #foo put: 42.
```

```
  self assert: (o1 at: #foo) equals: 42.
```

And we should add a test to verify that two different states are created.

```
ProtoTest >> testExNihiloCreationDifferentState

  | o1 o2 |
  o1 := Proto exNihilo.
  o1 at: #foo put: 42.
  o2 := Proto exNihilo.
  o2 at: #foo put: 33.
  self assert: (o1 at: #foo) equals: 42.
  self assert: (o2 at: #foo) equals: 33.
```

Since we do not want to touch the Pharo compiler we represent object variables as a dictionary.

```
Proto >> initialize
  states := Dictionary new.
```

We add two methods `at:` and `at:put:` to access and set value to a variable.

```
Proto >> at: aSymbol
  ^ states at: aSymbol

Proto >> at: aSymbol put: aValue
  ^ states at: aSymbol put: aValue
```

Since we love representation of implementation and introspective systems, let's define a little API for it. Let us set that the message `variableNamed` returns the list of variables of the object.

```
ProtoTest >> testStructuralReflection

  | o1 o2 |
  o1 := Proto exNihilo.
  o1 at: #foo put: 42.
  self assert: (o1 variableNames includes: #foo).
  o2 := o1 clone.
  self assert: (o2 variableNames includes: #foo).
```

The method implementation is straigthforward.

```
Proto >> variableNames
  "Return the variables used in the receiver"
  ^ states keys
```

2.4 Instance creation as cloning

Now we are ready to implement object instantiation based on cloning. Cloning an object may have different semantics and implementation. We decide

to go for creation-time sharing: it means that the new object will share its state with its originator but that as soon as we modify a field, the value is not shared anymore.

The following snippet illustrates such behavior.

```
| o1 o2 |
o1 := Proto exNihilo.
o1 at: #foo put: 42.
o1 at: #foo.
>>> 42

o2 := o1 clone.
o2 at: #foo put: 33.
o2 at: #foo.
"> 33"
o1 at: #foo.
"> 42"
```

Now let us write some tests. First let us ensure that both objects have access to the same variable.

```
ProtoTest >> testAfterCloningClonedObjectHasBeSameState

  | o1 o2 |
  o1 := Proto exNihilo.
  o1 at: #foo put: 42.
  o2 := o1 clone.
  self assert: (o1 at: #foo) equals: 42.
  self assert: (o2 at: #foo) equals: 42.
```

Now this test is a bit weak since it does not show that the sharing is only during creation time. The following test verifies that the value is shared forever.

```
ProtoTest >> testAfterCloningClonedObjectDoNotShareState

  | o1 o2 |
  o1 := Proto exNihilo.
  o1 at: #foo put: 42.
  o2 := o1 clone.
  self assert: (o1 at: #foo) equals: 42.
  o2 at: #foo put: 33.
  self assert: (o2 at: #foo) equals: 33.
  self assert: (o1 at: #foo) equals: 42.
```

Finally we want to make sure that modifying the state of the parent does not impact the objects that got cloned from it. It shows that Propha offers at the same time behavior sharing and state creation time sharing but in addition Propha avoids parent state change propagation.

```
ProtoTest >> testAfterCloningNoParentStatePropagation
```

```
  | o1 o2 |
  o1 := Proto exNihilo.
  o1 at: #foo put: 42.
  o2 := o1 clone.
  self assert: (o1 at: #foo) equals: 42.
  o1 at: #foo put: 666.
  self assert: (o2 at: #foo) equals: 42.
  self assert: (o1 at: #foo) equals: 666.
```

A first implementation of clone

So let us implement the clone method. A possible (and wrong) solution is the following. Let us make the current test green and we will think later where is the problem.

```
Proto >> clone
  ^ self copy
```

But now we are sharing the object state! So we should copy the state too.

```
Proto >> postCopy
  states := self states copy.
```

We will see that this is not correct because now our two objects are sharing the same anonymous subclass and the same behavior. So adding one method will add this behavior to both objects while it should not. We are not in class-based languages.

2.5 About behavior

Now we are ready to define method, i.e., behavior of our prototypes. The following snippet illustrates the expected behavior.

```
o1 := Proto exNihilo.
o1 at: #foo put: 42.
o1 at: #foo.
>>> 42

o1 define: 'double  ^ (self at: #foo) * 2'.
o1 double.
>>>84
```

```
ProtoTest >> testBehaviorDefinition

  | o |
  o := Proto exNihilo.
  o at: #foo put: 42.
  o define: 'double  ^ (self at: #foo) * 2'.
  self assert: o double equals: 84
```

The easiest way is to simply compile the method in the anynomous class as follows:

```
Proto >> define: aString
  self behavior compile: aString classified: 'methods'
```

2.6 Instance specific behavior

Now we should verify that the behavior that we define is really specific to the object and that only when using delegation such behavior is shared.

The following snippet shows that our implementation is wrong. Cloning an object should not share the behavior only the state.

```
o1 := Proto exNihilo.
o1 at: #foo put: 42.
o1 at: #foo.
>>> 42

o1 define: 'double  ^ (self at: #foo) * 2'.
o1 double.
>>>84

o2 := o1 clone.
o2 at: #foo put: 33.
o2 at: #foo.
>>> 33

o2 double.
>>> 66
```

This is not really correct since it is more like class-based where multiple classes share common behavior. Here `double` should not be available to `o2` because we want `clone` to be only be about state and not behavior.

Let us write a test to cover such aspect. Here we checks that `double` is not available and similarly that `triple` when defined in a clone does not impact the parent.

```
ProtoTest >> testCloningShouldNotShareBehavior

  | o1 o2 |
  o1 := Proto exNihilo.
  o1 at: #foo put: 42.
  o1 define: 'double  ^ (self at: #foo) * 2'.
  self assert: o1 double equals: 84.
  o2 := o1 clone.
  o2 at: #foo put: 33.
  self should: [ o2 double] raise: MessageNotUnderstood.
  o2 define: 'triple  ^ (self at: #foo) * 3'.
```

```
  self should: [ o1 triple ] raise: MessageNotUnderstood.
  self assert: o2 triple equals: 99.
```

So we should create a new prototype with its own anonymous class and copy the the state of its mother prototype. Then after the two objects are autonomous.

```
Proto >> clone

  | newObject |
  newObject := self class exNihilo.
  newObject states: self states copy.
  ^ newObject
```

```
Proto >> states: aDictionary

  states := aDictionary
```

```
testCloningShouldNotShareBehavior

  | o o1 |
  o := Proto exNihilo.
  o at: #foo put: 42.
  o define: 'double  ^ (self at: #foo) * 2'.
  self assert: o double equals: 84.
  o1 := o clone.
  o1 at: #foo put: 33.
  self should:  [ o1 double] raise: MessageNotUnderstood
```

2.7 Delegation

In class-based language, a class reuses and extends the behavior of other class using inheritance. In prototype-based language the pendant is delegation. With prototype delegation when a method is not found in the receiver it is looked up in its proto or parent following a delegation chain and when it is found, it is applied to the receiver of the message.

> **Note** It should be noted that delegation in class-based language is not the same as delegation in prototype languages because in class-based language delegation the receiver of the delegated message change. In prototype language delegation similarly than with class-based inheritance a method is looked in different behavior container and applied to the receiver of the message.

Let us write some test to cover the semantics we want to implement.

```
ProtoTest >>testDelegation

  | o1 o2 |
  o1 := Proto exNihilo.
```

```
  o1 at: #foo put: 42.
  o1 define: 'double  ^ (self at: #foo) * 2'.
  o2 := Proto exNihilo.
  self should:  [ o2 double] raise: MessageNotUnderstood.
  o2 delegateTo: o1.
  o2 at: #foo put: 33.
  self assert: o2 double equals: 66
```

```
Proto >> delegateTo: anObject

   self behavior superclass: anObject behavior
```

In the model we implemented we make a difference between state and behavior. Cloning only clones the state and does not clone the behavior and delegation only share behavior. The following test shows that applying a parent method on a children not having the correct field raise an error. This is expected when creating an object ex nihilo.

```
ProtoTest >> testDelegationOnlyShareBehavior

  | o1 o2 |
  o1 := Proto exNihilo.
  o1 at: #foo put: 42.
  o1 define: 'double  ^ (self at: #foo) * 2'.
  o2 := Proto exNihilo.
  self should: [ o2 double] raise: MessageNotUnderstood.
  o2 delegateTo: o1.
  self should: [ o2 double] raise: KeyNotFound
```

The following test show that cloning and delegating together are a better way to achieve sharing.

```
ProtoTest >> testDelegationWorksBetterWithCloning

  | o1 o2 |
  o1 := Proto exNihilo.
  o1 at: #foo put: 42.
  o1 define: 'double  ^ (self at: #foo) * 2'.
  o2 := o1 clone.
  self should: [ o2 double] raise: MessageNotUnderstood.
  o2 delegateTo: o1.
  self shouldnt: [ o2 double] raise: KeyNotFound
```

2.8 Conclusion

We have implemented an interesting little prototype kernel. Propha offers at the same time behavior sharing via implicit delegation and state creation-time sharing but in addition it avoids parent state change propagation. There are plenty of prototype models and alternate implementations. So Propha is

just one possible model but it has nice semantics and can be implemented in 9 small methods on top of Pharo and we hope that we gave you the curiosity to experiment more.