Exercise 5

This exercise will be set up from the viewpoint of a LeaseManager. Based on his preference, we will be working with a 'green' (hybrid) car or one running on gasoline solely.

- We will be re-using the OnBoardComputer classes (OnBoardComputer, OnBoardComputerGasoline, OnBoardComputerHybrid) from the previous exercise.
- We can also re-use the Car class from the previous exercise, but it has to become ABSTRACT.
- The Car class will have 2 subclasses: GasolineCar and HybridCar. Each subclass will set its own properties. E.g. a hybrid Toyota Prius and a gasoline Audi A5 Sportback.
- Next, we'll need an AbstractCarFactory interface. The only method is createCar.
- There will be 2 implementations for the AbstractCarFactory: 1 for Gasoline cars and one for Hybrid cars.
  The LeaseManager class has a method to reportRemainingMileage:

```
MESSAGE "The car in your fleet can do " objCar:howFarCanIGet() " kilometers"
VIEW-AS ALERT-BOX.
```

To test the LeaseManager class, use the following code in a .p file:

```
DEFINE VARIABLE objLeaseManager AS LeaseManager NO-UNDO.
DEFINE VARIABLE objCarFactory AS AbstractCarFactory NO-UNDO.
DEFINE VARIABLE lTreeHugger AS LOGICAL NO-UNDO.

objLeaseManager = NEW LeaseManager().

MESSAGE "Show green car"
VIEW-AS ALERT-BOX QUESTION BUTTON YES-NO UPDATE lTreeHugger.
CASE lTreeHugger:
    WHEN TRUE THEN
        objCarFactory = NEW HybridCarFactory().
    WHEN FALSE THEN
        objCarFactory = NEW GasolineCarFactory().
END CASE.
objLeaseManager:reportRemainingMileage(objCarFactory).
```

Exercise 6

In this exercise you'll have to build a specific car, and Audi A5 with 18 inch Alloy wheels in the color Black.

Some extra information for this exercise:

- A Car has the attributes Brand (character), Model (character), Color (Enum) and Wheel (Wheel)
- The input for its constructor is a CarBuilder object
- The available colors are Red, Gray and Black
- A Wheel has a Price and a Size
- An AlloyWheel has size 18 and price 180
- A SteelWheel has size 16 and price 76
- The constructor of an abstract CarBuilder gets the Brand and Model as input parameters
- An AudiA5Builder knows how to build a specific car of Brand "Audi", Model "A5" with Alloy Wheels. It gets the optional color as input parameter for its constructor.

Running this from a test procedure would look something like this:

```
objAudiA5Builder = NEW AudiA5Builder(Color:Black).
objCar = objAudiA5Builder:build().
```

Exercise 7 A

In this exercise you'll delegate the creation of classes to a specialized Factory, the CarFactory.
A Car is represented by a simple interface, with 1 property: Brand. We have 2 classes that implement this interface: Audi and BMW.
The CarFactory will return a Car object based on marketing input. Our test procedure testCarFactory looks as follows:

```
DEFINE VARIABLE objVorsprung AS Car NO-UNDO.
DEFINE VARIABLE objFreude    AS Car NO-UNDO.

objVorsprung = CarFactory:createObject("Vorsprung durch technik").
objFreude = CarFactory:createObject("Freude am fahren").

MESSAGE "Vorsprung mit "objVorsprung:Brand ", Freude mit " objFreude:Brand
VIEW-AS ALERT-BOX.
```

Hint: use a case statement in the CarFactory to determine the class to instantiate.

Exercise 7 B

The CASE statement in CarFactory has some disadvantages. Adding new implementations of Car will require a code change. Same goes if we need to make a specialization for a specific environment. And this only works for Cars…
Therefore, you'll have to create a generic Factory based on an input XML file:

- The configuration is in config.xml
- The new generic factory class will be called Factory
- And it uses the xml configuration to determine which class to instantiate, though in a more dynamic way.

You'll also need to create a testFactory.p, much like the testCarFactory. What is the main difference?

Exercise 8

In this exercise you'll have to implement a class that is constructed according to the Singleton pattern. That means it will have a private constructor, a private property to store its instance and a public method to get an instance of the class. To be able to test the singleton feature, the class will have a public integer property MyProperty.

The test procedure looks as below. Both messages should show the value 42:

```
DEFINE VARIABLE objMySingletonClass      AS MySingletonClass NO-UNDO.
DEFINE VARIABLE objMyOtherSingletonClass AS MySingletonClass NO-UNDO.

ASSIGN objMySingletonClass = MySingletonClass:getInstance()
       objMySingletonClass:MyProperty = 42.
MESSAGE objMySingletonClass:MyProperty
VIEW-AS ALERT-BOX.

ASSIGN objMyOtherSingletonClass = MySingletonClass:getInstance().
MESSAGE objMyOtherSingletonClass:MyProperty
VIEW-AS ALERT-BOX.
```

Exercise 9

In this exercise, you will add an Adapter to an existing class.

Our current application has a **LabelPrinter** class with a *print* method. However, this class requires a certain run-time configuration to specify the printer's properties, e.g. location, type, paper, etc. These properties are private and because the object is rather complex, it uses the Builder pattern to configure itself.

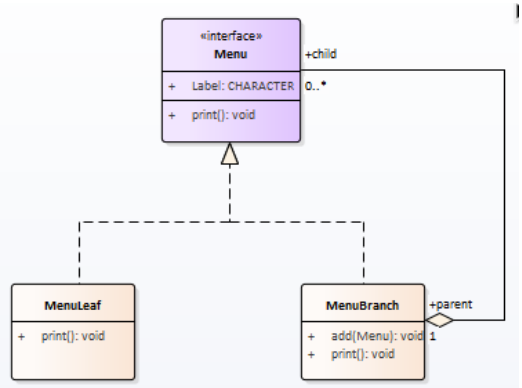
Our new WMS (Warehouse Management System) application needs to print labels but it does not know anything about the existing printers or how to configure them. It has a class named **Scanner** which has an option to print labels. We do not wish to write new label printing code, since we already have a class that does exactly that and it works extremely well.

What should we do?

We most certainly do not want to modify the current **LabelPrinter** class to adapt to the new **Scanner** class. We also do not want to "teach" the **Scanner** class to use the **LabelPrinter** class. Neither class should need to adapt to the other. Instead, we create an adapter class for **Scanner** to use which, in turn, uses the **LabelPrinter** class and will configure the printer. Let's call this adapter class **ScannerPrint**. The adapter will keep both applications simple and separate — the complexities of reusing the existing class will be kept in a single class (Separation of Concerns.)

Exercise 10

In this exercise, we'll build up a menu structure that can 'print' itself using the Composite pattern. The specific methods needed in the Composite class (MenuBranch) will be added only to the class itself.
You will need to implement a Menu interface, a MenuBranch class and a MenuLeaf class. The MenuBranch will have an 'add' method and a temp-table to store a reference to its leafs. Both classes implement the print() method, but of course the MenuBranch class has to propagate this behavior to its leafs.



The testMenu procedure looks like this:

```
DEFINE VARIABLE objRoot   AS MenuBranch NO-UNDO.
DEFINE VARIABLE objBranch AS MenuBranch NO-UNDO.

ASSIGN
    objRoot   = NEW MenuBranch("Root")
    objBranch = NEW MenuBranch("-File").

objRoot:add(objBranch).

objBranch:add(NEW MenuLeaf("--New")).
objBranch:add(NEW MenuLeaf("--Open")).
objBranch:add(NEW MenuLeaf("--Save")).
objBranch = NEW MenuBranch("-Edit").

objRoot:add(objBranch).

objBranch:add(NEW MenuLeaf("--Cut")).
objBranch:add(NEW MenuLeaf("--Copy")).
objBranch:add(NEW MenuLeaf("--Paste")).


OUTPUT TO "menu.txt".
objRoot:print().
```

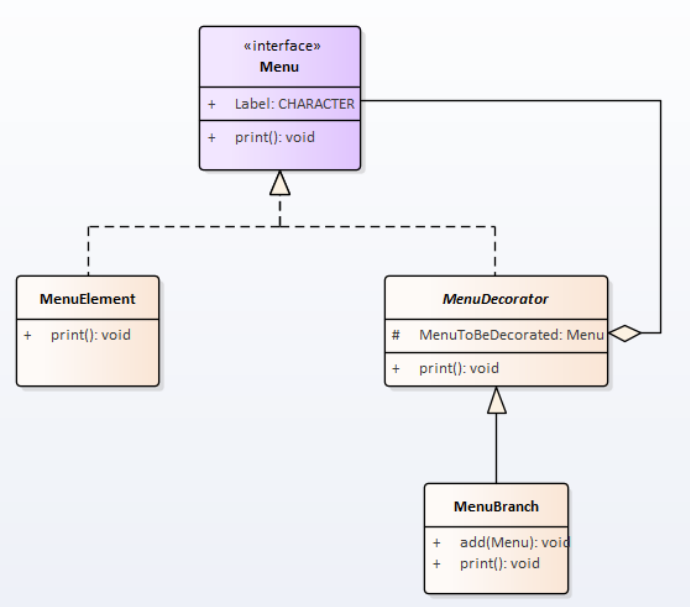
The output in menu.txt looks like this:

Root
-File
--New
--Open
--Save
-Edit
--Cut
--Copy
--Paste

Note: later on, we will refactor the solution to use the Decorator pattern.

Exercise 11

This exercise starts with the solution files of Exercise 10.
In exercise 10 the Menu structure was implemented using 2 separate classes that both implemented the Menu interface in its own way. This causes duplication of code, especially if we would need multiple types of Menu objects. Subclassing would be an alternative, but if the classes require a mix of capabilities, that is also not efficient.
The Decorator pattern holds the solution to this challenge. In this exercise, you will have to refactor the existing classes to the structure below. The Menu class still contains the interface, the MenuElement is the generic baseclass that replaces MenuLeaf. The MenuDecorator is an abstract class that makes use of MenuElement (via the Menu interface) for the standard functionality. A concrete MenuBranch is the concrete class.

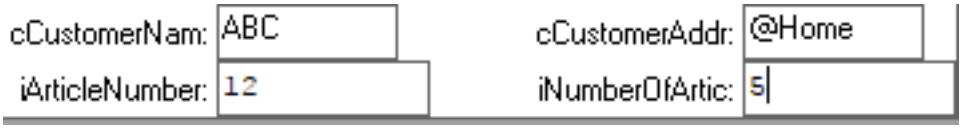


The test procedure now looks like this:

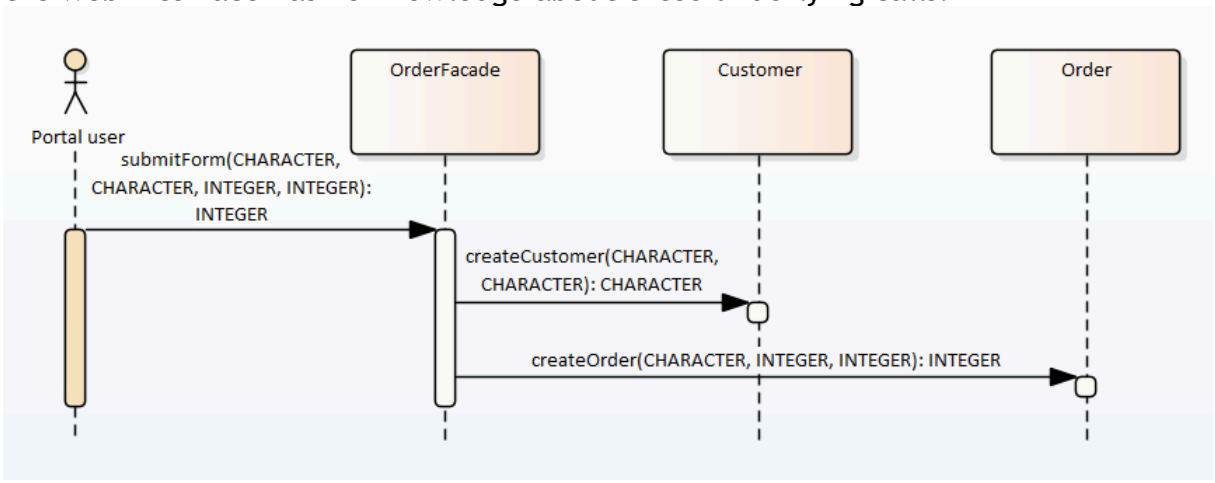
```
DEFINE VARIABLE objRoot   AS MenuBranch NO-UNDO.
DEFINE VARIABLE objBranch AS MenuBranch NO-UNDO.

ASSIGN
    objRoot   = NEW MenuBranch(NEW MenuElement("Root"))
    objBranch = NEW MenuBranch(NEW MenuElement("-File")).
objRoot:add(objBranch).
objBranch:add(NEW MenuElement("--New")).
objBranch:add(NEW MenuElement("--Open")).
objBranch:add(NEW MenuElement("--Save")).
objBranch = NEW MenuBranch(NEW MenuElement("-Edit")).
objRoot:add(objBranch).
objBranch:add(NEW MenuElement("--Cut")).
objBranch:add(NEW MenuElement("--Copy")).
objBranch:add(NEW MenuElement("--Paste")).

OUTPUT TO "menu.txt".
objRoot:print().
```

Exercise 12

Our traditional Sports company has the back end systems in place for order entry. To be able to enter the digital age, the Sports company decides to offer the possibility for customers to order articles via a portal. An advanced mockup is shown below:



The new web service has to re-use the existing backoffice logic. Which means it has to create a customer and use the generated customer id to create the order, the web interface has no knowledge about these underlying calls:



The test procedure has the following content:

```
DEFINE VARIABLE objOrderFacade AS OrderFacade NO-UNDO.

objOrderFacade = NEW OrderFacade().
def var cCustomerName as character no-undo.
def var cCustomerAddress as character no-undo.
def var iArticleNumber as integer no-undo.
def var iNumberOfArticles as integer no-undo.
update cCustomerName cCustomerAddress iArticleNumber iNumberOfArticles with 2 columns.

objOrderFacade:submitForm(cCustomerName, cCustomerAddress, iArticleNumber, iNumberOfArticles).
```

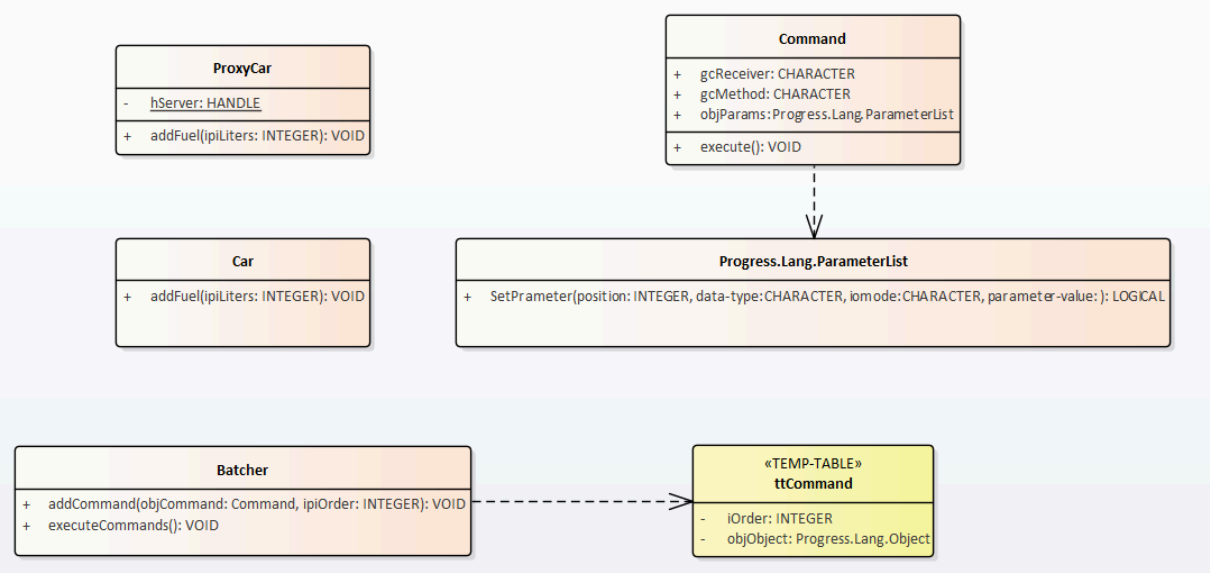
Create a Customer and Order class with the specific methods (just dummy implementations) and use the façade pattern to create the new submitForm service.

Exercise 13

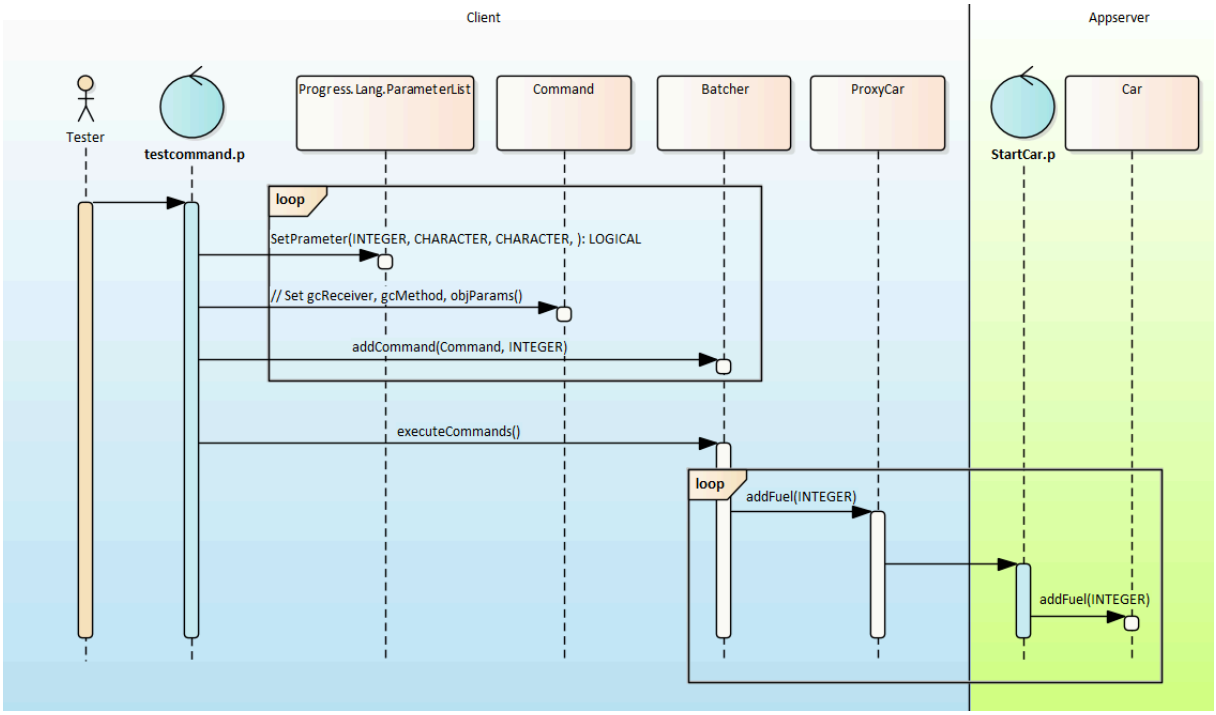
In this exercise you will be combining the Chain of Responsibility pattern with the Command pattern:

- A test procedure will create 2 Command objects.
- These Command objects hold all the information to
    - call a method (addFuel)
    - in a class (proxyCar)
    - with parameters (ipiLiters)
    - and their values (10 and 25)
    - in a given order (1 and 2)
- The Command objects will be added to a Batcher
- When executeCommands is called the Batcher will execute the Command objects
- As we want the logic to run on an appserver, the Commands will execute methods in a proxy class.
- The proxy class will run the given method asynchronously on the appserver using a remote .p file.

The next classes are involved. Note that the Command will be completely dynamic. It uses reflection to execute the given method!



The next overview shows the runtime behavior of the solution.



And finally the test procedure testcommand.p:

```
DEFINE VARIABLE objCommand       AS Command                     NO-UNDO.
DEFINE VARIABLE objParameterList AS Progress.Lang.ParameterList NO-UNDO.
DEFINE VARIABLE objBatcher       AS Batcher                     NO-UNDO.

ASSIGN
    objBatcher             = NEW Batcher()
    objCommand             = NEW Command()
    objCommand:gcReceiver  = "DesignPrinciples.ProxyCar"
    objCommand:gcMethod    = "addFuel"
    objParameterList       = NEW Progress.Lang.ParameterList(1).
objParameterList:SetParameter(1,"INTEGER","INPUT", 25).
objCommand:objParams = objParameterList.
objBatcher:addCommand(objCommand,1).

ASSIGN
    objCommand             = NEW Command()
    objCommand:gcReceiver  = "DesignPrinciples.ProxyCar"
    objCommand:gcMethod    = "addFuel"
    objParameterList       = NEW Progress.Lang.ParameterList(1).
objParameterList:SetParameter(1,"INTEGER","INPUT", 10).
objCommand:objParams = objParameterList.
objBatcher:addCommand(objCommand,2).

objBatcher:executeCommands().
```

Before starting the implementation, make sure the standard Progress AppServer (oepas1) is up and running. The propath should include the directory where your sources/r-code are situated.

Exercise 14

In this exercise you will create an Iterator. However, the Iterator will only act as 'cursor' for the Aggregate (Collection) as we don't want to expose all internals of the Collection itself.

- The basis is a Car class with 2 properties: Brand and Model.
- The Car objects will be stored in a CarCollection class:
    - this class has a temp-table to hold the Car object and
    - a field to hold the sequence of the specific 'record'.
    - The collection also 'knows' the number of cars it holds (iNumberOfCars) and it offers access to its Iterator (CarIterator) via a property.
    - In the constructor 6 cars will be created hardcoded. Don't forget to set the number of cars property.
    - The class only requires 1 method: getCar. This method returns a Car object based on the sequence number.
- The CarIterator
    - Has its property objCarCollection set in the constructor
    - Holds a 'cursor' (iCurrent)
    - Offers at least methods for
        - getting the first Car in the Collection
        - getting the last Car in the Collection
        - determining if the current Car is the last one in the Collection

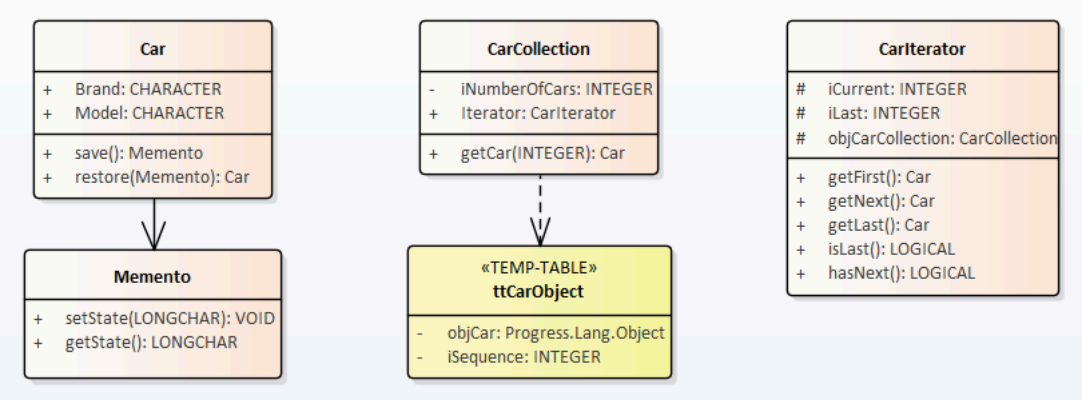
The testCarIterator looks as follows:

```
DEFINE VARIABLE objCarCollection AS CarCollection NO-UNDO.
DEFINE VARIABLE objCar           AS Car           NO-UNDO.

objCarCollection = NEW CarCollection().
objCar = objCarCollection:Iterator:getFirst().
MESSAGE objCar:Brand objCar:Model
    VIEW-AS ALERT-BOX.
objCar = objCarCollection:Iterator:getLast().
MESSAGE objCar:Brand objCar:Model
    VIEW-AS ALERT-BOX.
MESSAGE objCarCollection:Iterator:isLast()
VIEW-AS ALERT-BOX.
```

Create the Car, Carcollection and CarIterator classes using the Iterator pattern. Make sure not to break the encapsulation rules: the CarIterator cannot access the collection directly!

Exercise 15

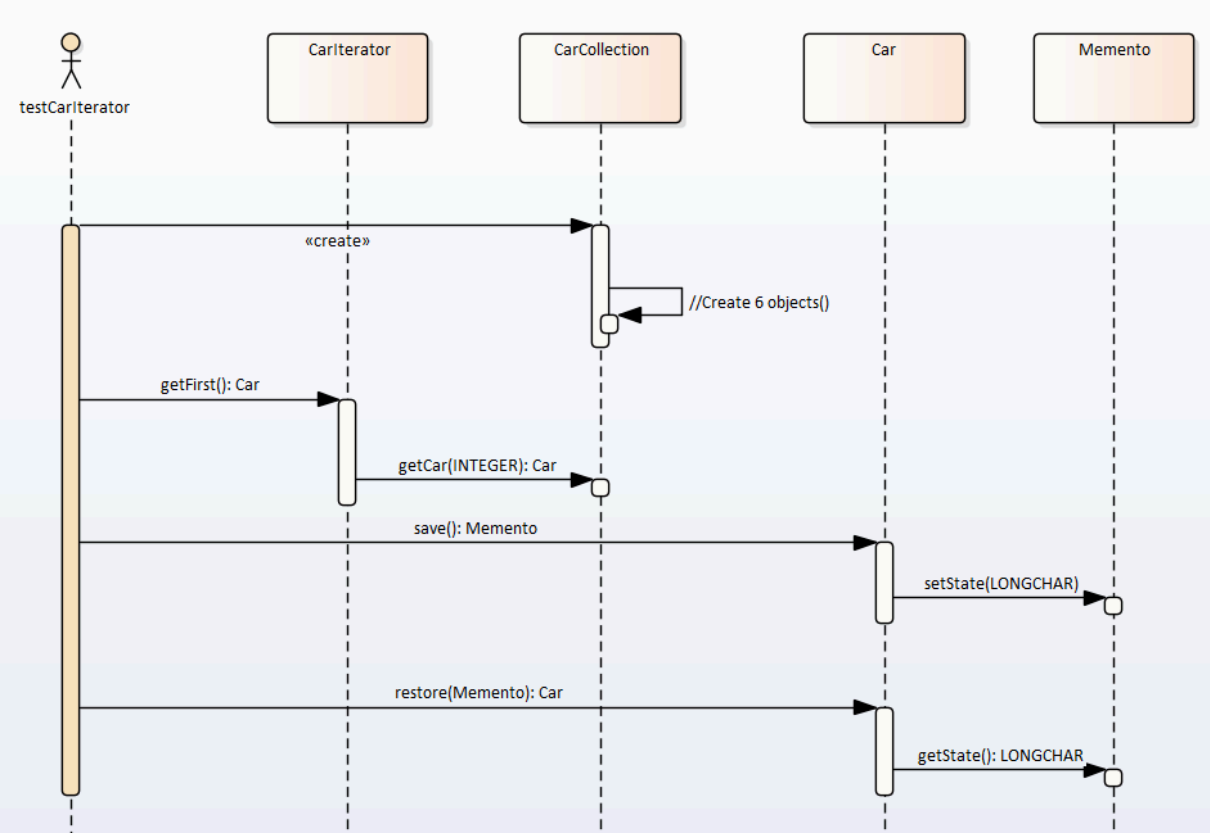
In this exercise you will extend the solution of exercise 14. To make sure you're aligned with this exercise, copy the provided solution.
The exercise will start with a test procedure that creates the same CarCollection as before and gets the first Car object.
Before making any changes to the Car Model, let the Car object save its state using a Memento object. Then make some changes to the Model and restore the Car using the saved Memento.
The next diagram shows the participating classes:



The test procedure testCarIterator.p:

```
DEFINE VARIABLE objCarCollection AS CarCollection NO-UNDO.
DEFINE VARIABLE objCar           AS Car           NO-UNDO.
DEFINE VARIABLE objMemento       AS Memento       NO-UNDO.

objCarCollection = NEW CarCollection().
objCar = objCarCollection:Iterator:getFirst().
MESSAGE objCar:Brand objCar:Model
    VIEW-AS ALERT-BOX.

objMemento = objCar:save().
ASSIGN
    objCar:Model = "A7".
MESSAGE objCar:Model
    VIEW-AS ALERT-BOX.

objCar = objCar:restore(objMemento).
MESSAGE objCar:Model
    VIEW-AS ALERT-BOX.
```

The sequence of events looks as follows.



Note: you will need the StringInputStream and StringOutputStream classes supplied in the solution package. These will serialize the Car object to a LONGCHAR, and deserialize it later on from a LONGCHAR to a Car object.
Example code for the Car:save method:

```
ASSIGN
    objSerializer   = NEW Progress.IO.JsonSerializer(FALSE)
    objStringStream = NEW StringOutputStream().
objSerializer:Serialize(THIS-OBJECT, objStringStream).
objStringStream:Close().
```

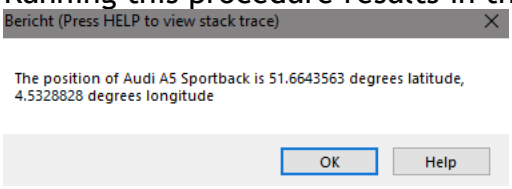
Exercise 16

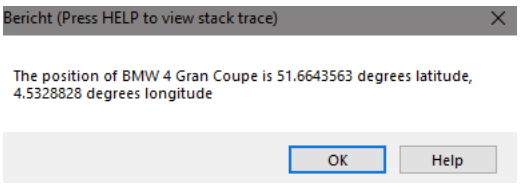
In this exercise you will implement the very basics of the Observer pattern using the OpenEdge built-in Publish/Subscribe capabilities.
The goal here is to create a FleetOwner that owns multiple cars. At any given time, the FleetOwner wants to know the GPS position (latitude/longitude) of each of his cars.

The test procedure (see below) will create a FleetOwner and 2 Cars. The constructors of the Car class need an override so that they get the FleetOwner object, this enables the Car objects to Subscribe to the event.
The Event itself does not need any parameters.

```
DEFINE VARIABLE objAudi       AS Car        NO-UNDO.
DEFINE VARIABLE objBMW        AS Car        NO-UNDO.
DEFINE VARIABLE objFleetOwner AS FleetOwner NO-UNDO.

objFleetOwner = NEW FleetOwner().

objAudi = NEW Car(objFleetOwner).
ASSIGN
    objAudi:Brand = 'Audi'
    objAudi:Model = "A5 Sportback".
objBMW = NEW Car(objFleetOwner).
ASSIGN
    objBMW:Brand = 'BMW'
    objBMW:Model = "4 Gran Coupe".

objFleetOwner:showPositions().
```

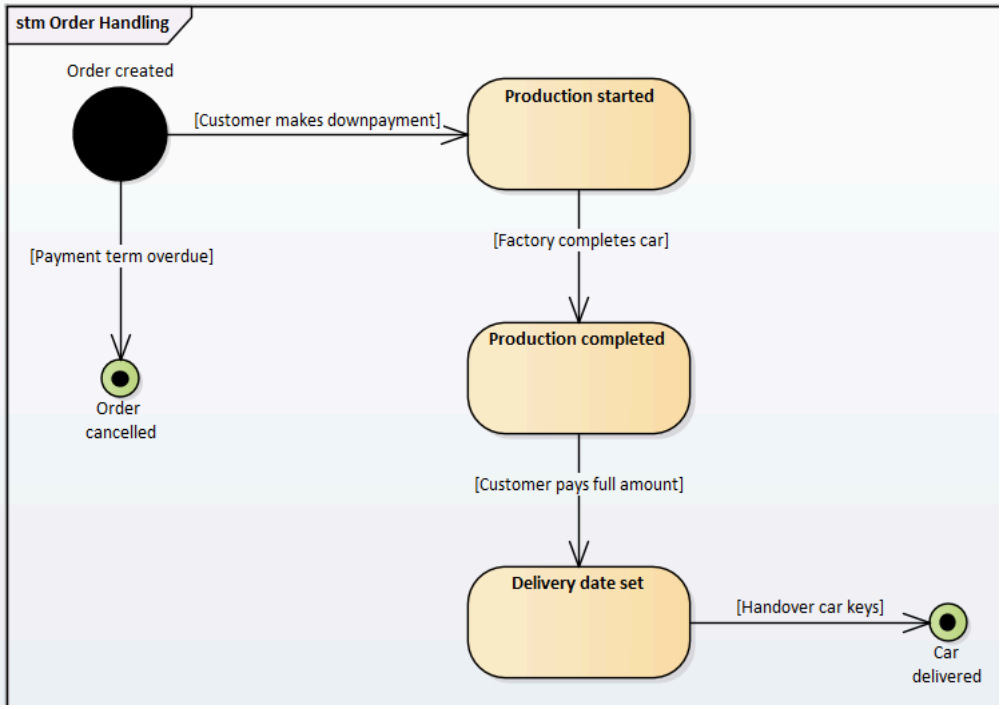
Running this procedure results in the next 2 dialogs:



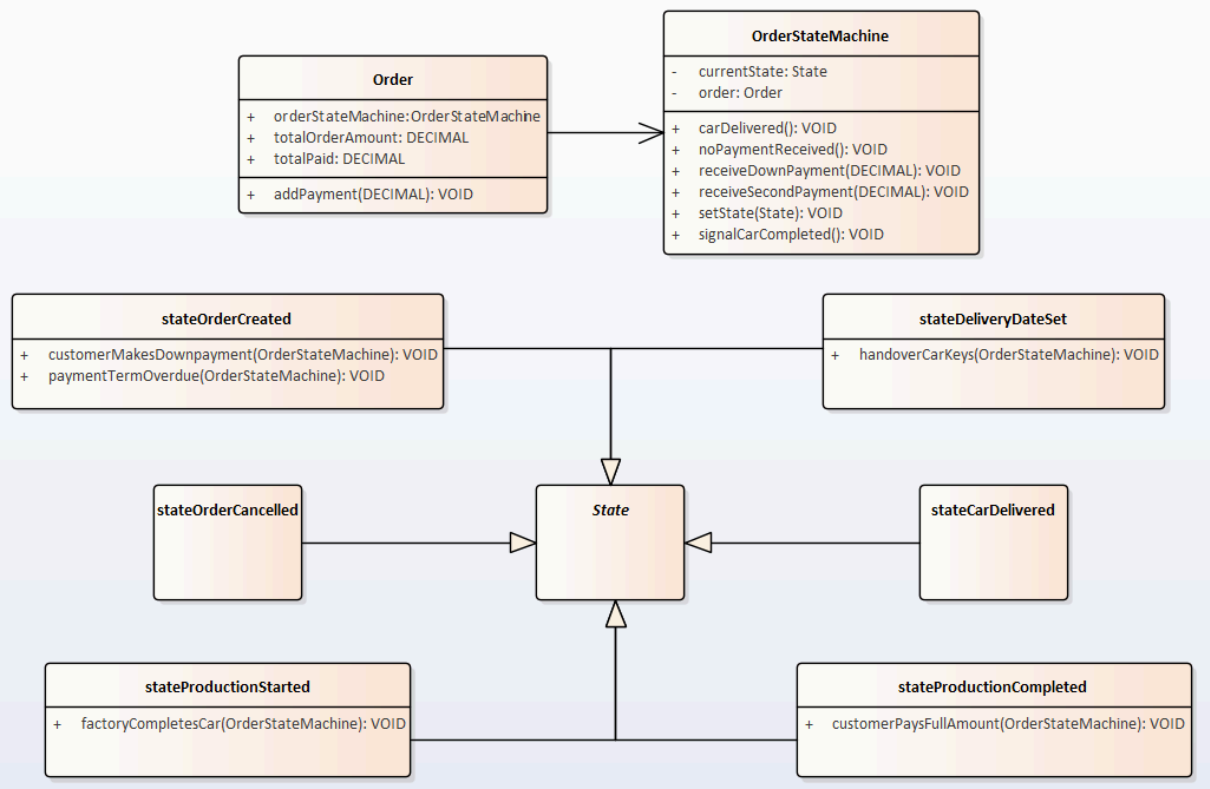


Exercise 17

In this exercise you will create an Object Oriented Finite State Machine.
The process you have to create is the handling of a Car Order. The States and Actions to be implemented are described in the next State Machine diagram:



The Order class contains the logic on payments (totalOrderAmount, totalPaid and the addPayment method). Everything that has to do with State changes and Actions is delegated to an OrderStateMachine. This machine keeps a pointer to the current State and the logic for all the Events (carDelivered, signalCarCompleted etc.). This includes making decisions if the downpayment has been paid or not.
The State abstract class only contains logic in its constructor to present a message for testing our .p file. The concrete State classes tell the State Machine what the next State will be:



The test procedure looks as follows. The code follows a complete process of creating an order, paying downpayment etc.. Optionally you can end the process by using the noPaymentReceived action:

```
DEFINE VARIABLE objOrder AS Order NO-UNDO.

objOrder = NEW Order().
ASSIGN
    objOrder:totalOrderAmount = 10000.

/* nothing paid */
//objOrder:orderStateMachine:noPaymentReceived().

/* pay some money, but not enough */
objOrder:orderStateMachine:receiveDownpayment(1500).

/* pay some money, just enough */
objOrder:orderStateMachine:receiveDownpayment(500).

/* complete production */
objOrder:orderStateMachine:signalCarCompleted().

/* pay remainder */
objOrder:orderStateMachine:receiveSecondPayment(8000).

/* get the keys */
objOrder:orderStateMachine:carDelivered().
```