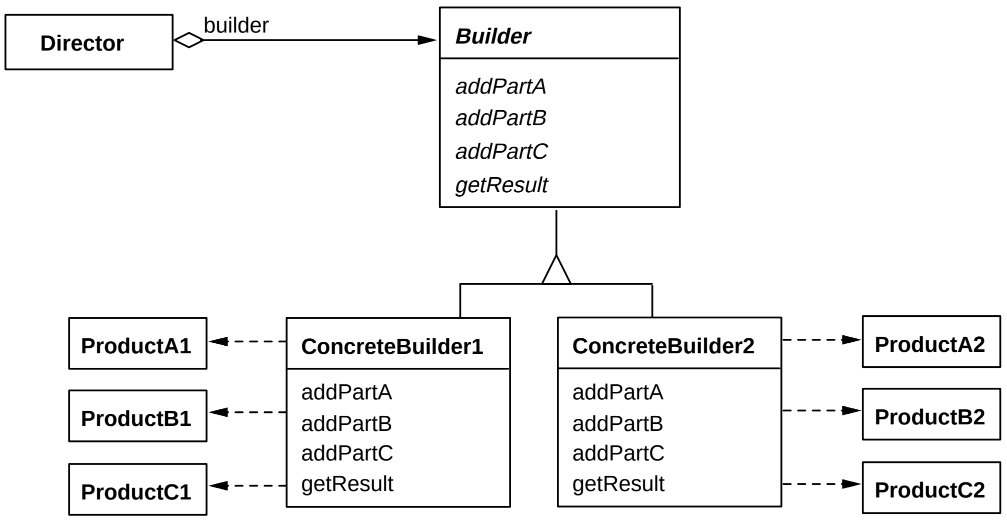
BUILDER (DP 97) Object Creational

Intent

Separate the construction of a complex object from its representation so that the same construction process can create different representations.

Structure



Discussion

First off, we suggest reading our Abstract Factory pattern before continuing here. There are several mutual issues with regard to the problem context—the situation under which the patterns may apply—and the pattern's solution.

Suppose an application needs to build complex objects based on user selections or specifications from other objects. Say it's a car assembly application much like that described in Abstract Factory. The application must be able to build different products, specifically Ford, Toyota, or Porsche cars[1]. We *could* set a flag in the application to specify the type of product to create. Each time we need to create a component (say, an engine) of the overall product, we would perform a conditional based on that flag:

[1] For illustrative purposes, we'll use an oversimplification—of course, there are more than three auto makers, several different models produced by each manufacturer, etc.

```
AnApplication>>createEngine
   "Without Builder"
   manufacturer == #Ford
      ifTrue: [^FordEngine new].
   manufacturer == #Toyota
      ifTrue: [^ToyotaEngine new].
   ^PorscheEngine new
```

We'd need the same sort of logic for every car and car component. We'd also have to give this object the know-how to assemble the subcomponent parts (engines, bodies, transmissions) into a coherent final product (a car). That's a lot of behavior for a single class. And it's not very extensible. Just because we have three known car brands *now* is no guarantee users or the application domain won't require some other component types in the future (e.g., a new car line, Saturn, is introduced to the market). We need a design that lets us extend the application's functionality—that is, add new component types—without modifying the main application itself.

The Builder pattern offers a solution. It makes a separate Builder object responsible for creating and assembling components on the application's behalf. The application can use a Ford Builder to put Fords together, a Toyota Builder to assemble Toyotas, or a Porsche Builder to assemble Porsches.

Builder separates a thing being built and details of how it gets constructed from a client of that thing. The thing being built is called the Product; in our example, this is the overall car. The client is called the Director because it's in control of the overall construction process. The Director knows what generic subcomponents go in the Product (in our example, an engine, a car body, etc.)., but doesn't know which component classes to instantiate for different types of Products (e.g., if a Ford is being constructed and an engine is required, we ought to instantiate the `FordEngine` class). Further, the Director does not know how to put the Product together. So it enlists the help of an external helper, a Builder object—the Builder provides a message interface for both adding subcomponent parts and retrieving the ultimate Product.

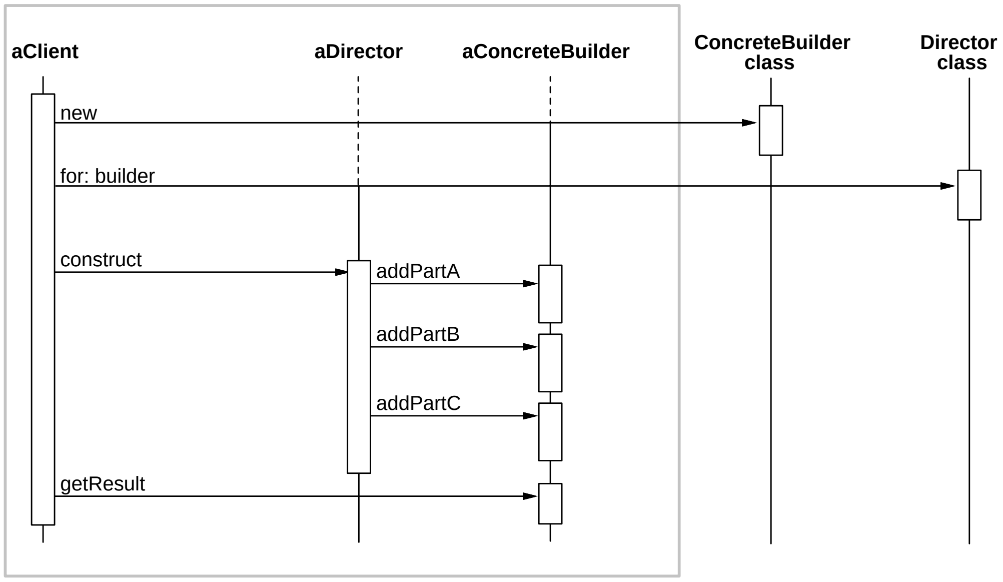
We need several different types of Builders (a Ford, Toyota, and Porsche Builder), and the application will select the one to use at run-time depending on user preference or programmatic circumstances. To make this possible, we will define multiple Builder classes and have them all respond polymorphically to the same set of building messages. Hence the code in the Director won't care what sort of Builder it's talking to, so long as it adheres to the established Builder protocol. Thus by plugging different Builders together with a Director, the same Director can construct very different Products—without having to change the Director object.

The Builder pattern greatly simplifies the implementation of the Director class since the Director has no direct knowledge of the Product's internal representations encapsulated within the Builder, nor of how to put the Product together. Additionally, it's easy to change a Builder's behavior or add new types of Builders to an application because the

Builders have been separated out into separate classes which typically reside in a single subhierarchy. Hence we know where to go to make these modifications, and enhancements to the available Builders require no modification to the Director itself.

Collaborations

The following interaction diagram portrays the Builder pattern's collaborations. Note that the final result may be retrieved from the Builder by the Director or by the Director's client.



This differs slightly from the collaboration diagram on page DP 99 because in Smalltalk we need to account for the run-time messages sent to class objects for instance creation. With regard to the Builder pattern per se, the Director's client creates the Builder object and hands it off to the Director (this is one option; alternatively the Director could instantiate the appropriate Builder itself). Then the client sends the `construct` message to the Director; this method repeatedly asks the Builder object to build new parts (messages `addPartA`, `addPartB`, and `addPartC`). In response, the Builder adds parts to its developing Product. When a subcomponent part is added, the Builder does not respond with—that is, return—anything of interest (of course, since all methods return something, the Builder just returns `self` by default). When the building process is complete, the Director (or in this case the Director's client) asks the Builder for the resultant Product by sending `getResult`.

Implementation

Adding parts

The Builder has some form of internal state: as subcomponent specifications are received, the Builder adds new parts to its encapsulated Product. Actually, the Builder has the choice of instantiating subcomponents each time the Director says “Add part so-and-so,” or of merely saving the “so-and-so” specification and building the final Product in its entirety when the “Return the final Product” message is received. In either case, nonetheless, some data is encapsulated within the Builder until the final Product is retrieved by a client.

There are at least three variations on the “Add part” theme.

- The interaction diagram shows that the Director merely tells the Builder to add specific types of parts to its Product: add a component of type A, add a part of type B, “Add a 4-cylinder engine,” “Add a 2-door coupe body.” The Builder just instantiates the appropriate class and adds the new part to its evolving Product.
- The Director may hand “raw material” to its Builder and tell the Builder to perform some transformation on that raw part and add the result to the Product. *Design Patterns* offers the example of converting the information in a file from one format (e.g., Rich Text Format) to another (e.g., TeX). There, an “Add part” request might imply, “Here’s an RTF font-change token; convert it to a TeX font-change specification and then add it to your Product. Thus the Builder must be implemented with more know-how than simply instantiating the appropriate class.
- The Director can give an abstract specification of a component to the Builder, leaving it to the Builder to interpret the specification, construct an object based on it, and add that object to its Product. This is essentially what’s done in VisualWorks’ `UIBuilder`: the VisualWorks `UIPainter`, which lets users lay out the widgets of a user interface window, generates an abstract specification based on the layout. This includes specifications for all of the window’s widgets, their type, location, size, content (e.g., the text to appear in a label), etc. These specifications are saved in a method of the application class associated with the UI window. At application startup, these “window specs” are passed to a `UIBuilder` which creates each widget and adds it to the overall window.

Multiple Families of Products

We typically have multiple parts families but want a Builder to create parts from a single family. In our example, we want cars and their subparts instantiated from the Ford, Toyota, or Porsche family. In Abstract Factory, we discussed how the code in a factory object may choose among product families; the implementation issues are the same for Builder—we can define multiple Builder classes, each of which hardcodes the component classes to instantiate; we can use part catalogs; we can apply the Factory

Method pattern; we can even define a single Builder class with different methods that create parts from different Product families. We've shown how to implement all of these options in Abstract Factory, but let's look in more detail at the Factory Method solution for Builder and follow that with an alternative solution not considered previously.

Using Factory Method. Different Builders add the same component with the identical message—e.g., to add a 2-door sedan body, we send `add2DoorSedanBody` to the Builder. This method looks the same in all the Builder classes except for the actual class that gets instantiated:

```
CarBuilder>>add2DoorSedanBody
   "Do nothing. Subclasses will override."

FordBuilder>>add2DoorSedanBody
   self car
      addBody: Ford2DoorSedanBody new

ToyotaBuilder>>add2DoorSedanBody
   self car
      addBody: Toyota2DoorSedanBody new

PorscheBuilder>>add2DoorSedanBody
   self car
      addBody: Porsche2DoorSedanBody new
```

Alternatively, we can avoid the duplication of code and implement a factory method in each concrete Builder class, as follows:

```
CarBuilder>>add2DoorSedanBody
   "Define this once for all subclasses. Subclasses will
    override the 'create2DoorSedanBody' factory method."
   self car
      addBody: self create2DoorSedanBody

CarBuilder>>create2DoorSedanBody
   "Factory method; This should only be implemented
    by my concrete subclasses."
   self implementedBySubclass    "Visual Smalltalk"
   "self subclassResponsibility" "VisualWorks, IBM, others"

FordBuilder>>create2DoorSedanBody
   "Factory method; Return an instance of the
    Ford 2-door-sedan body class"
   ^Ford2DoorSedanBody new

ToyotaBuilder>>create2DoorSedanBody
   "Return an instance of the Toyota 2-door-sedan body class"
   ^Toyota2DoorSedanBody new
```

```
PorscheBuilder>>create2DoorSedanBody
   "Return a Porsche 2-door-sedan body object"
   ^Porsche2DoorSedanBody new
```

Using Abstract Factory. There's another alternative method for selecting among Product families. We can apply the Abstract Factory pattern as a secondary pattern within the Builder. Let's look at an existing example of this approach from VisualWorks.

A `UIBuilder` is asked to build the widgets of a window. Since VisualWorks supports multiple platform-specific interface widgets, there are multiple families of widgets—a Motif family (classes like `MotifRadioButton`), an OS/2 CUA family (`CUARadioButton`), and so on. The `UIBuilder` must create widgets from the currently selected look-and-feel family—for example, if the user selected the OS/2 CUA look, the `UIBuilder` must instantiate `CUARadioButton` when a radio button is being created.

One solution to this problem, of course, is what we've already described—define several Builder classes and instantiate the one we want. Here we would define `UIBuilder` as an abstract class, with concrete subclasses such as `MotifUIBuilder`, `CUAUIBuilder`, and `MacUIBuilder`. Instead, VisualWorks applies the Abstract Factory pattern as a subordinate helper pattern.

A `UIBuilder` is configured with a `UILookPolicy` object to which it delegates its widget-creation tasks—that is, the Builder asks its look policy to instantiate each of the widgets in the window. Different concrete subclasses of the abstract `UILookPolicy` class implement the same widget-creation messages to construct widgets according to a particular look-and-feel—`MacLookPolicy` creates radio buttons which appear as they do on a Macintosh screen, `MotifLookPolicy` instances create Motif-like buttons. Depending on the desired look-and-feel, one of these subclasses is instantiated as the `UIBuilder`'s look policy object. The `UIBuilder` has no knowledge of what sort of look is being constructed, nor of what sort of look policy object it is talking to; it merely sends the widget-creation messages defined in the abstract `UILookPolicy` protocol.

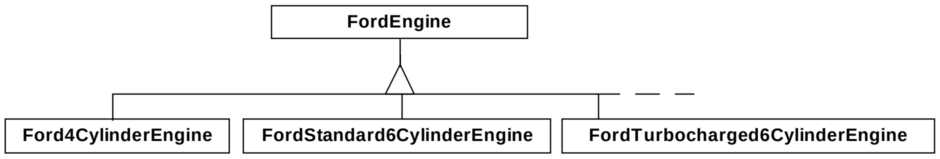
So the `UILookPolicy` object acts as a factory for the `UIBuilder,` and the `UIBuilder`/`UILookPolicy` framework is an instance of the Abstract Factory pattern. Although there is a single Builder class, `UIBuilder`, it can create Products from multiple Product (widget) families.

Sample Code

Now let's look at an example that expands on our car assembly application. Suppose we have an application that allows a user to walk into a generic car sales showroom, step up to a computer in a kiosk, and assemble an order for a car. This application might alternatively be deployed on the Web as a virtual car sales showroom. Imagine this showroom does not sell cars made by a particular company; rather, the customer can select *any* brand of car. She may also choose the options she desires. Imagine a typical user scenario wherein the user selects a Honda automobile and options such as the two-door sedan body type, six cylinder engine, automatic transmission, air-conditioning, luxury audio package, and deluxe paint trim, and then selects a menu item that says "Order."

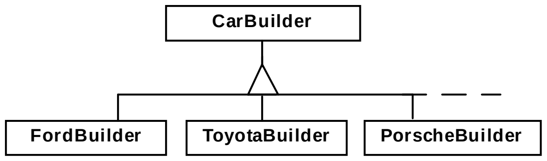
We'd like this application to have a common user interface for all cars and options, and we'd like to assemble a virtual car based on the user's selections. That is, as in the Abstract Factory example, the application will render a 3D, navigable image of the car on the screen, allowing the customer to view and "tour" the car. Before actually ordering the car, she can change the car's configuration and view it again. Finally, the application generates an order for the car as the user described it.

Assume Product classes similar to those used in Abstract Factory, except there will be more of them. Since we're allowing for the selection of options, we'll have classes like the following rather than just a `FordEngine` class:



We'll have similar subhierarchies under `ToyotaEngine`, `PorscheEngine`, `FordBody`, `ToyotaBody`, `PorscheBody`, and likewise for all other components.

With regard to the Builder portion of this application, we start by defining a `CarBuilder` hierarchy.



```
Object subclass: #CarBuilder
   instanceVariableNames: 'car'
   classVariableNames: ''
   poolDictionaries: ''

CarBuilder class>>new
   ^self basicNew initialize
```

```
CarBuilder>>car
   "getter method"
   ^car

CarBuilder>>car: aCar
   "setter method"
   car := aCar

CarBuilder subclass: #FordBuilder
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''

CarBuilder subclass: #ToyotaBuilder
   ...

CarBuilder subclass: #PorscheBuilder
   ...
```

The `car` instance variable in `CarBuilder` references the Builder's Product. When a Builder is first instantiated, `car` is initialized to an instance of the appropriate `Car` subclass (an empty shell; a `Car` with no subcomponents yet):

```
FordBuilder>>initialize
   self car: FordCar new.

ToyotaBuilder>>initialize
   self car: ToyotaCar new.

PorscheBuilder>>initialize
   self car: PorscheCar new.
```

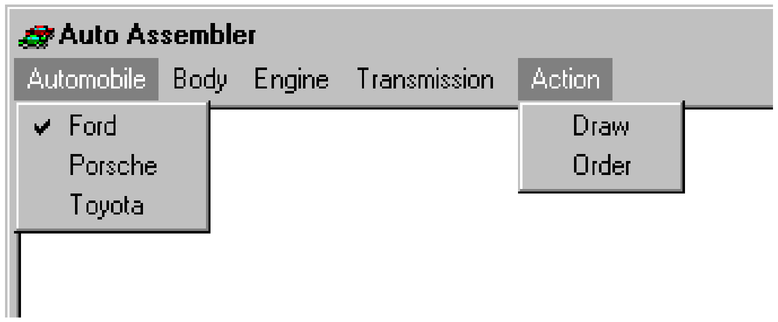
As the Builder receives requests to add components, it adds them to `car.` Let's define the "Add a subcomponent" messages: we define these messages in the abstract superclass to do nothing, overriding them as necessary in concrete Builder subclasses:

```
CarBuilder>>add4CylinderEngine
   "Do nothing. Subclasses will override."

FordBuilder>>add4CylinderEngine
   self car
      addEngine: Ford4CylinderEngine new

ToyotaBuilder>>add4CylinderEngine
   self car
      addEngine: Toyota4CylinderEngine new
```

```
PorscheBuilder>>add4CylinderEngine
   self car
      addEngine: Porsche4CylinderEngine new


CarBuilder>>addStandard6CylinderEngine
   "Do nothing. Subclasses will override."

FordBuilder>>addStandard6CylinderEngine
   self car
      addEngine: FordStandard6CylinderEngine new

ToyotaBuilder>>addStandard6CylinderEngine
   self car
      addEngine: ToyotaStandard6CylinderEngine new

PorscheBuilder>>addStandard6CylinderEngine
   self car
      addEngine: PorscheStandard6CylinderEngine new
```

We're assuming here that `Car`s know how to add subcomponents to themselves with messages like `addEngine:`, `addBody:`, etc.

Now let's talk about the user interface of our sample application. Users will be able to select an automobile manufacturer from a menu and then choose the engine type, body type, and other features from corresponding menus. When the user wants to view what has been built so far, he selects "Draw" from the "Action" menu. In response, the application renders the car in the graphics pane (below the menu bar). When the user wants to order the car, he selects "Order" from the same menu. The UI might look like the following (where additional pulldown menus would be added for "Audio System," "Trim," and other subcomponents):



Let's say the user interface is implemented by an `AutoAssemblerUI` object. The `AutoAssemblerUI` is configured with a Car`Builder` instance, and the `AutoAssemblerUI` tells this Builder to create and add car components based on the user's selections. Here's some of the code that implements this application in Visual Smalltalk. First, we define the UI application as a subclass of `ViewManager` and then add methods to construct and open the window:

```
ViewManager subclass: #AutoAssemblerUI
   instanceVariableNames: 'builder'
   classVariableNames: ''
   poolDictionaries: ''

AutoAssemblerUI>>open
   self
      owner: self;
      label: 'Auto Assembler';
      createView;
      openWindow
```

The window will include the menus described earlier, along with a single graphics pane in which the car will be drawn.

```
AutoAssemblerUI>>createView
   "Private - create the panes for the receiver window."
   | pane |
   self mainView
      when: #menuBarBuilt send: #rebuildMenuBar to: self.

   pane := GraphPane new.
   pane
      setName: #graphOutput:;
      when: #needsContents
         send: #graphOutput:
         to: self
         with: pane;
      framingRatio:
         (Rectangle leftTopUnit extentFromLeftTop: 1@1).
   self addSubpane: pane

AutoAssemblerUI>>rebuildMenuBar
   "Fixup the menu bar with the pulldowns required for the
    AutoAssembler application."
   | menuWindow |
   menuWindow := self menuWindow.
   menuWindow
      removeMenu: (self menuTitled: 'File');
      removeMenu: (self menuTitled: 'Edit');
      removeMenu: (self menuTitled: 'Smalltalk');
      addMenu: self carMenu;
      addMenu: self bodyMenu;
      addMenu: self engineMenu;
      addMenu: self transmissionMenu;
      addMenu: self actionMenu
```

The actual pulldown menus are constructed by methods invoked above in `rebuildMenuBar`. The `carMenu` method is an interesting one to look at—it constructs the menu programmatically based on the existing concrete `CarBuilder` subclasses. A menu item is constructed for each Builder class; its label is obtained by sending the `manufacturer` message to each Builder. The associated action is a message that invokes the `AutoAssemblerUI>>userSelectedBuilder:` method, passing the corresponding `CarBuilder` subclass as its argument:

```
AutoAssemblerUI>>carMenu
   "Build the car-manufacturers menu."
   | menu |
   menu := Menu new
      title: 'Automobile';
      owner: self.
   CarBuilder subclasses do: [:aClass |
      menu
        appendItem: aClass manufacturer
        selector: (Message
                     receiver: self
                     selector: #userSelectedBuilder:
                     arguments: (Array with: aClass))].
   ^menu
```

When the user selects a manufacturer from the “Automobile” menu, `userSelectedBuilder:` creates the appropriate Builder by instantiating the class passed as the message argument. Subsequently, this Builder is used for creating and assembling all of the car’s subcomponent parts:

```
AutoAssemblerUI>>userSelectedBuilder: builderClass
   | menu |
   builder := builderClass new.

   "Only after an automobile manufacturer is selected may
    the user select subcomponent parts and options from
menus:"
   self enableSubcomponentMenus.

   "Uncheck all manufacturer menu items and check the
    one just selected:"
   menu := self menuTitled: 'Automobile'.
   menu items do: [:menuItem |
      menu uncheckItem: menuItem selector].
   menu checkItem: builderClass manufacturer.
```

We also need to define the `manufacturer` method in each of the `CarBuilder` classes:

```
CarBuilder class>>manufacturer
   self implementedBySubclass
```

```
FordBuilder class>>manufacturer
   ^'Ford'

ToyotaBuilder class>>manufacturer
   ^'Toyota'

PorscheBuilder class>>manufacturer
   ^'Porsche'
```

Now we need the menus for other car components. We’re making the simplifying assumption that all `Car`s may be constructed with the same generic parts—all may include either a 4-cylinder, standard 6-cylinder, or turbocharged 6-cylinder engine, for example. But, of course, for each type of car, the Builders instantiate different classes for each engine type. In the case of 4-cylinder engines, for example, the application must instantiate `Ford4CylinderEngine`, `Toyota4CylinderEngine`, or `Porsche4CylinderEngine`.

```
AutoAssemblerUI>>engineMenu
   ^Menu new
      title: 'Engine';
      owner: self;
      appendItem: '4-Cylinder'
         selector: #engineIs4Cylinder;
      appendItem: '6-Cylinder Standard'
         selector: #engineIsStandard6Cylinder;
      appendItem: '6-Cylinder Turbocharged'
         selector: #engineIsTurbocharged6Cylinder;
      "Gray-out the menu until a manufacturer has
       been selected:"
      disableAll;
      yourself
```

So when the user selects, say, “4-Cylinder” from the “Engine” pulldown, the `AutoAssemblerUI>>engineIs4Cylinder` method is invoked. It simply sends the generic `add4CylinderEngine` message to its Builder:

```
AutoAssemblerUI>>engineIs4Cylinder
   "The user has selected the '4-cylinder' menu item
    from the 'Engine' pulldown menu. Tell my Builder."
   self builder add4CylinderEngine
```

Each Builder class implements `add4CylinderEngine` to instantiate the appropriate class depending on the Builder’s parts family (these methods were defined earlier). The methods for other components are implemented in a similar fashion:

```
AutoAssemblerUI>>engineIsStandard6Cylinder
   "The user has selected the 'Standard 6-cylinder'
    menu item from the 'Engine' pulldown menu."
   self builder addStandard6CylinderEngine
```

```
AutoAssemblerUI>>engineIsTurbocharged6Cylinder
   "The user has selected the 'Turbocharged 6-cylinder'
    menu item from the 'Engine' pulldown menu."
   self builder addTurbocharged6CylinderEngine

AutoAssemblerUI>>bodyIs2DoorCoupe
   "The user has selected the '2-Door Coupe' menu item
    from the 'Body' pulldown menu."
   self builder add2DoorCoupeBody

AutoAssemblerUI>>bodyIs2DoorSedan
   "The user has selected the '2-Door Sedan' menu item
    from the 'Body' pulldown menu."
   self builder add2DoorSedanBody
```

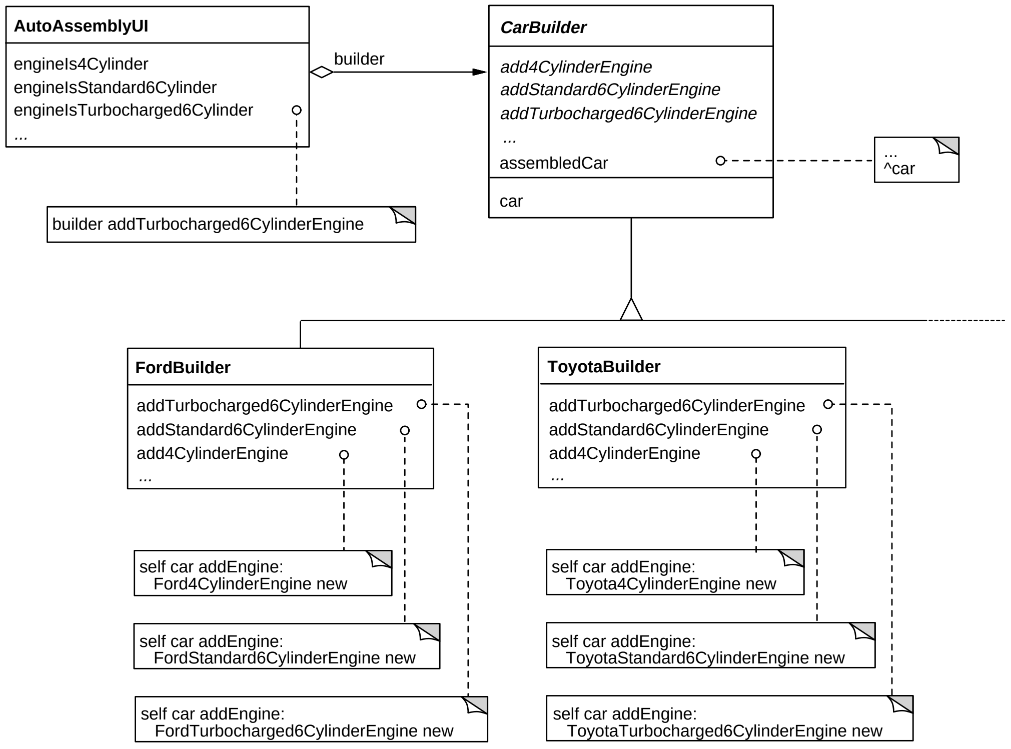
We also need to define a method to build our “Action” menu. When the user clicks on the “Order” menu item, the application will retrieve the Builder’s Product (the assembled car), pass it to an object that draws the car, and then construct an order based on the assembled car.

```
AutoAssemblerUI>>actionMenu
   ^Menu new
      title: 'Action';
      owner: self;
      appendItem: 'Draw' selector: #drawCar;
      appendItem: 'Order' selector: #orderCar;
      disableAll;  "Gray it out at window-open time"
      yourself

AutoAssemblerUI>>orderCar
   "The user has selected the 'Order' menu item, signaling
   all car/components selections have been made."

   | car |
   "Get the assembled car from my Builder:"
   car := builder assembledCar.
   car isNil ifTrue: [^MessageBox message:
     'You haven''t finished assembling a complete car yet!'].

   "Draw it for the user:"
   CarRenderer new
      render: car
      using: (self paneAt: #graphOutput:) pen.

   "Assemble and print an invoice for the assembled car:"
   CarInvoiceMaker new printInvoiceFor: car
```

`CarBuilder>>assembledCar` is the Builder’s “Return the final Product” method. It verifies the car has all essential subcomponents, such as an engine and body, and then returns the completed car.

```
CarBuilder>>assembledCar
   "Return my final Product after verifying there's
    a completed Product to return."
   car isNil ifTrue: [^nil].
   car engine isNil ifTrue: [^nil].
   ...
   ^car
```

Structurally, our application looks like the following diagram. Notice the difference between this and the similar Abstract Factory diagram. In Abstract Factory, the factory can be asked to create an individual component and it returns the appropriate object. If the factory client wants to, it may add each of these parts to a larger product, but the factory itself has no knowledge of this. A Builder can also be asked to create an individual component, but for each such request the Builder returns nothing of interest; instead, the newly created component is added to the Product encapsulated within the Builder. Later, when the all subcomponents have been added, the Builder can be asked for its ultimate Product.


AutoAssemblyUI
engineIs4Cylinder
engineIsStandard6Cylinder
engineIsTurbocharged6Cylinder
...
builder
builder addTurbocharged6CylinderEngine
CarBuilder
add4CylinderEngine
addStandard6CylinderEngine
addTurbocharged6CylinderEngine
...
assembledCar
car
...
^car
FordBuilder
addTurbocharged6CylinderEngine
addStandard6CylinderEngine
add4CylinderEngine
...
ToyotaBuilder
addTurbocharged6CylinderEngine
addStandard6CylinderEngine
add4CylinderEngine
...
self car addEngine: Ford4CylinderEngine new
self car addEngine: FordStandard6CylinderEngine new
self car addEngine: FordTurbocharged6CylinderEngine new
self car addEngine: Toyota4CylinderEngine new
self car addEngine: ToyotaStandard6CylinderEngine new
self car addEngine: ToyotaTurbocharged6CylinderEngine new

Known Smalltalk Uses

Design Patterns mentions three known uses of the Builder pattern in VisualWorks (page DP 105). We'll describe one of these (`ClassBuilder`) in detail along with additional examples.

MenuBuilder

`MenuBuilder` in VisualWorks implements methods to add individual menu parts (menu items and separator lines). The information about each part is appended to the `MenuBuilder`'s internal representation of its product. When all the parts have been added, the client asks the `MenuBuilder` for the finished product by sending the `menu` message. In response, the `MenuBuilder` constructs and returns the completed `Menu` instance.

UIBuilder

A `UIBuilder` in VisualWorks constructs user interface windows and their subcomponent widgets. Specifications for individual interface widgets can be supplied to a `UIBuilder` with the `add:` message. When all the widget specifications are added, the `UIBuilder` can be asked to `open` the resultant product, a window. A large set of examples can be found in the `UIBuilder` class in the VisualWorks image (see the class methods in the *examples* category).

ClassBuilder

In VisualWorks, `ClassBuilder` instances are called on to create new classes or modify existing ones. For example, a typical class creation message looks like this:

```
ASuperclass subclass: #ASubclass
   instanceVariableNames: 'var1 var2'
   classVariableNames: 'ClassVar1'
   poolDictionaries: ''
   category: 'Companion Examples'
```

Here's the implementation of this message directly from the VisualWorks image:

```
Class>>subclass: t instanceVariableNames: f
classVariableNames: d poolDictionaries: s category: cat
   "This is the standard initialization message for
    creating a new class as a subclass of an existing
    class (the receiver)."
```

```
      ^self classBuilder
         superclass: self;
         environment: self environment;
         className: t;
         instVarString: f;
         classVarString: d;
         poolString: s;
         category: cat;
         beFixed;
         reviseSystem
```

The first message creates an instance of `ClassBuilder` using a factory method. The `classBuilder` method looks like this (class `Class` inherits from `Behavior`):

```
Behavior>>classBuilder
   ^ClassBuilder new
```

Subsequently, this `ClassBuilder` instance is sent a bunch of messages to set various characteristics of the class being created or modified. For example, the class' superclass is set and the class' name is assigned. Finally, in `reviseSystem` the `ClassBuilder` either constructs a new class object (if the class does not already exist) or modifies the existing class. This is the "Return the final Product" Builder message.

Related Patterns

Strategy

Builder is similar to the Strategy pattern; the difference between the two is their intended use. Builder is used to construct new objects, bit by bit, on behalf of a client; different types of Builder objects implement the same generic building protocol but may actually instantiate different classes. On the other hand, Strategy is used to provide an abstract interface to an *algorithm*, that is, a strategy object is a reification of an algorithm as an object; different strategy objects provide alternative implementations of the same generic service.

Abstract Factory

We've already seen that the Builder and Abstract Factory creational patterns are closely related. They are both used in situations where we want to instantiate Products from one of several Product families. The difference is that an Abstract Factory is called on to instantiate and return *all* component parts—each time it is invoked, the factory returns a Product—and its client *may* assemble them into a more complex object. A Builder is called upon in a piecemeal fashion to add components to an ultimate Product, and the Product is encapsulated within the Builder object. The Builder, rather than its client, is the Product assembler now. When the client has added all the component parts it requires, it asks the Builder for its final Product.