search...

Home » Creational Patterns » Builder   ■▪ Bookmark this on Delicious

Builder Pattern

| Like 65 | G+ | Tweet |

Motivation

The more complex an application is the complexity of classes and objects used increases. Complex objects are made of parts produced by other objects that need special care when being built. An application might need a mechanism for building complex objects that is independent from the ones that make up the object. If this is the problem you are being confronted with, you might want to try using the Builder (or Adaptive Builder) design pattern.

This pattern allows a client object to construct a complex object by specifying only its type and content, being shielded from the details related to the object�s representation. This way the construction process can be used to create different representations. The logic of this process is isolated form the actual steps used in creating the complex object, so the process can be used again to create a different object form the same set of simple objects as the first one.

Intent

   Defines an instance for creating an object but letting subclasses decide which class to instantiate
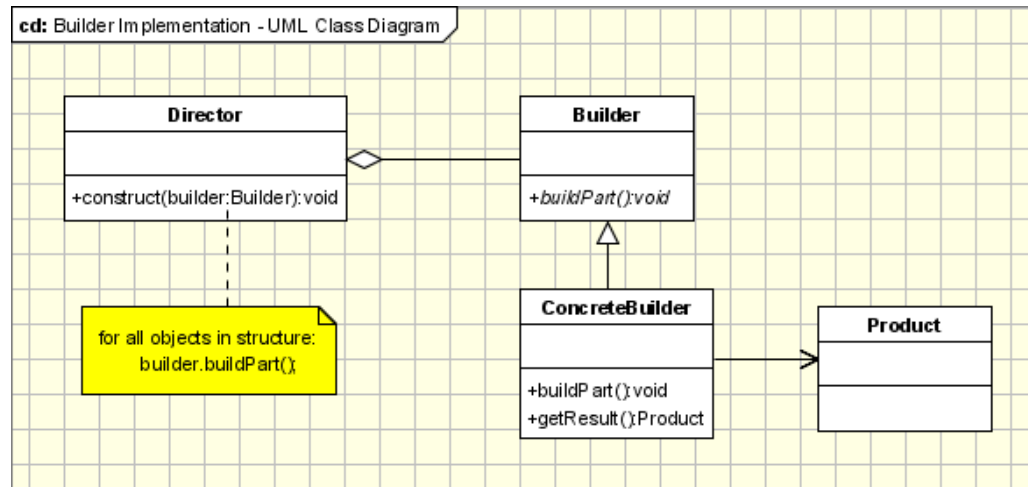   Refers to the newly created object through a common interface

Implementation

The Builder design pattern uses the Factory Builder pattern to decide which concrete class to initiate in order to build the desired type of object, as we will see below in the UML diagram:

cd: Builder Implementation - UML Class Diagram

```
        Director                                 Builder

+construct(builder:Builder):void        +buildPart():void

                                                  △
     for all objects in structure:               │
        builder.buildPart();              ConcreteBuilder          Product

                                         +buildPart():void
                                         +getResult():Product
```

The participants classes in this pattern are:

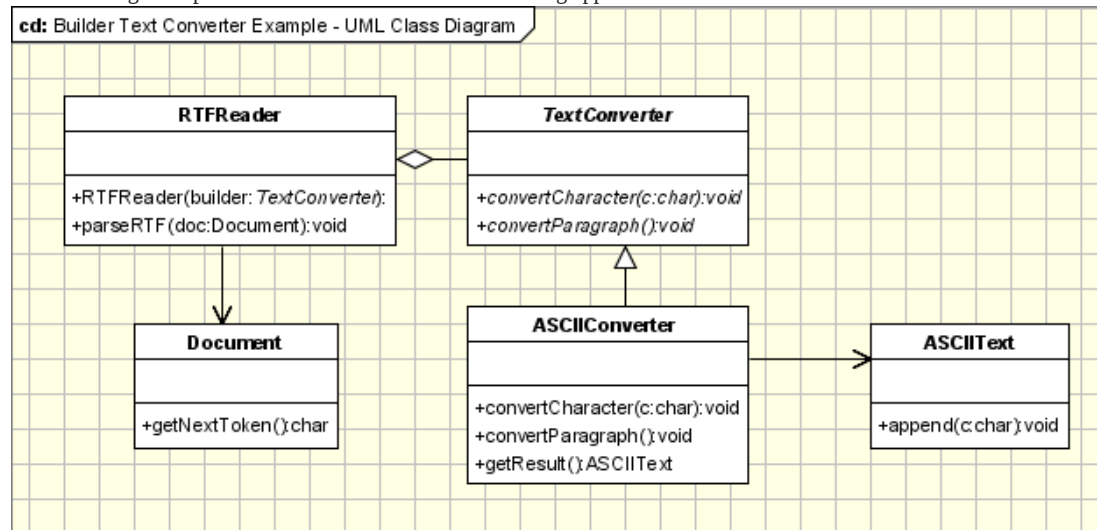The **Builder** class specifies an abstract interface for creating parts of a Product object.

The **ConcreteBuilder** constructs and puts together parts of the product by implementing the Builder interface. It defines and keeps track of the representation it creates and provides an interface for saving the product.

The **Director** class constructs the complex object using the Builder interface.

The **Product** represents the complex object that is being built.

The client, that may be either another object or the actual client that calls the main() method of the application, initiates the Builder and Director class. The Builder represents the complex object that needs to be built in terms of simpler objects and types. The constructor in the Director class receives a Builder object as a parameter from the Client and is responsible for calling the appropriate methods of the Builder class. In order to provide the Client with an interface for all concrete Builders, the Builder class should be an abstract one. This way you can add new types of complex objects by only defining the structure and reusing the logic for the actual construction process. The Client is the only one that needs to know about the new types, the Director needing to know which methods of the Builder to call.

The following example discusses the case of a text converting application:

cd: Builder Text Converter Example - UML Class Diagram

```
        RTFReader                            TextConverter

+RTFReader(builder: TextConverter):   +convertCharacter(c:char):void
+parseRTF(doc:Document):void          +convertParagraph():void

                                                 △
                                                 │
        Document                          ASCIIConverter             ASCIIText

+getNextToken():char                  +convertCharacter(c:char):void  +append(c:char):void
                                      +convertParagraph():void
                                      +getResult():ASCIIText
```

The Client needs to convert a document from RTF format to ASCII format. There for, it calls the method createASCIIText that takes as a parameter the document that will be converted. This method calls the concrete builder, ASCIIConverter, that extends the Builder, TextConverter, and overrides its two methods for converting characters and paragraphs, and also the Director, RTFReader, that parses the document and calls the builder�s methods depending on the type of token encountered. The product, the ASCIIText, is built step by step, by appending converted characters.

```
//Abstract Builder
class abstract class TextConverter{
        abstract void convertCharacter(char c);
        abstract void convertParagraph();
}

// Product
class ASCIIText{
        public void append(char c){ //Implement the code here }
}

//Concrete Builder
class ASCIIConverter extends TextConverter{
        ASCIIText asciiTextObj;//resulting product

        /*converts a character to target representation and appends to the resulting*/
        object void convertCharacter(char c){
                char asciiChar = new Character(c).charValue();
                        //gets the ascii character
                asciiTextObj.append(asciiChar);
        }
        void convertParagraph(){}
        ASCIIText getResult(){
                return asciiTextObj;
        }
}

//This class abstracts the document object
class Document{
        static int value;
        char token;
        public char getNextToken(){
                //Get the next token
                return token;
        }
}

//Director
class RTFReader{
        private static final char EOF='0'; //Delimitor for End of File
        final char CHAR='c';
        final char PARA='p';
        char t;
        TextConverter builder;
        RTFReader(TextConverter obj){
                builder=obj;
        }
        void parseRTF(Document doc){
                while ((t=doc.getNextToken())!= EOF){
                        switch (t){
                                case CHAR: builder.convertCharacter(t);
                                case PARA: builder.convertParagraph();
```

```
                        }
                    }
                }
}

//Client
public class Client{
        void createASCIIText(Document doc){
                ASCIIConverter asciiBuilder = new ASCIIConverter();
                RTFReader rtfReader = new RTFReader(asciiBuilder);
                rtfReader.parseRTF(doc);
                ASCIIText asciiText = asciiBuilder.getResult();
        }
        public static void main(String args[]){
                Client client=new Client();
                Document doc=new Document();
                client.createASCIIText(doc);
                system.out.println("This is an example of Builder Pattern");
        }
}
```

Applicability & Examples

Builder Pattern is used when:
    the creation algorithm of a complex object is independent from the parts that actually compose the object
    the system needs to allow different representations for the objects that are being built

Example 1 - Vehicle Manufacturer.

Let us take the case of a vehicle manufacturer that, from a set of parts, can build a car, a bicycle, a motorcycle or a scooter. In this case the Builder will become the VehicleBuilder. It specifies the interface for building any of the vehicles in the list above, using the same set of parts and a different set of rules for every type of type of vehicle. The ConcreteBuilders will be the builders attached to each of the objects that are being under construction. The Product is of course the vehicle that is being constructed and the Director is the manufacturer and its shop.

Example 1 - Students Exams.

If we have an application that can be used by the students of a University to provide them with the list of their grades for their exams, this application needs to run in different ways depending on the user that is using it, user that has to log in. This means that, for example, the admin needs to have some buttons enabled, buttons that needs to be disabled for the student, the common user. The Builder provides the interface for building form depending on the login information. The ConcreteBuilders are the specific forms for each type of user. The Product is the final form that the application will use in the given case and the Director is the application that, based on the login information, needs a specific form.

Specific problems and implementation

Builder and Abstract Factory

The Builder design pattern is very similar, at some extent, to the Abstract Factory pattern. That�s why it is important to be able to make the difference between the situations when one or the other is used. In the case of the Abstract Factory, the client uses the factory�s methods to create its own objects. In the Builder�s case, the Builder class is instructed on how to create the object and then it is asked for it, but the way that the class is put together is up to the Builder class, this detail making the difference between the two patterns.

Common interface for products

In practice the products created by the concrete builders have a structure significantly different, so if there is not a reason to derive different products a common parent class. This also distinguishes the Builder pattern from the Abstract Factory pattern which creates objects derived from a common type.

< Prev   **Next >**

[Back]