Department of Computer Science

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Tackling Design Patterns
Chapter 22: Builder Design Pattern

Contents

22.1 Introduction

The builder pattern is a creational pattern that adds an additional level of abstraction in order to separate the process of construction of a complex object from the representation of the object. This allows the designer to easily add or change representations without having to change the code defining the process. It will also be possible to change the process without having to change the representations.

22.2 Builder Design Pattern

22.2.1 Identification

Name	Classification	Strategy
Builder	Creational	Delegation
Intent		
Separate the construction of a complex object from its representation so that the same construction process can create different representations. ([1]:97)		

22.2.2 Structure

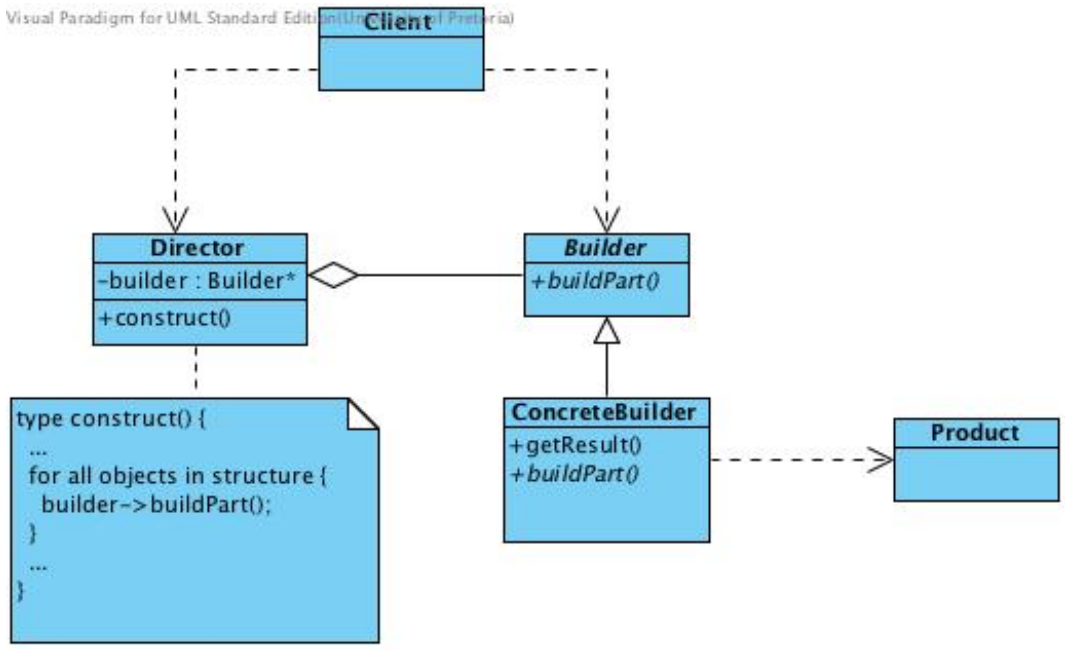

Figure 1: The structure of the Builder Design Pattern

22.2.3 Participants

Builder

- specifies an abstract interface for creating parts of a Product object.

Concrete Builder

- constructs and assembles parts of the product by implementing the Builder interface.
- defines and keeps track of the representation it creates.
- provides an interface for retrieving the product

Director

- constructs an object using the Builder interface.

Product

- represents the complex object under construction. ConcreteBuilder builds the product's internal representation and defines the process by which it's assembled.
- includes classes that define the constituent parts, including interfaces for assembling the parts into the final result.

22.2.4 Problem

An application maintains a complex aggregate and provide for the construction of different representations of the aggregate. There is a need to design the application in such a way that the addition of more representations of the aggregate would require minimal modification of the application.

22.3 Practical Example

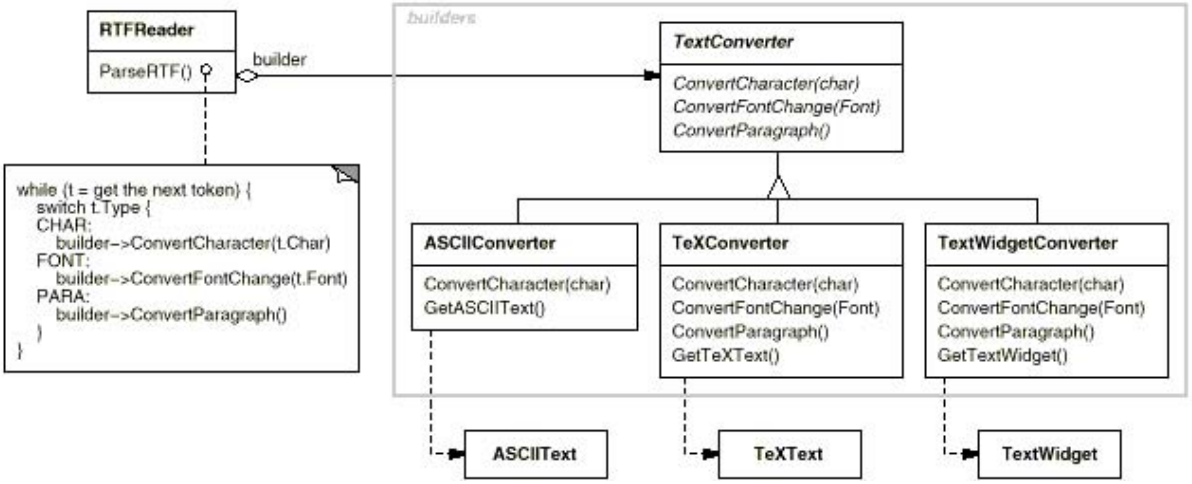

Figure 2: The design of a text converter

An example where the application of the builder pattern is useful offered by [1] is a parser of documents in RTF format that is used to produce documents in ASCII text format,

in TEX format or in a custom format for a text widget. Figure 2 is the class diagram of this application. The participants of the builder pattern in this implementation can be identified as follows:

Participant	**Entity in application**
Director	RTF Reader
Builder	TextConvereter
Concrete Builders	ASCIIConverter, TeXConverter, TextWidgetConverter
Products	ASCIIText, TeXText, textWidget
construct()	parseRTF
buildPart()	convertCharacter(), convertFontChange(), convertPraragraph()
getResult()	getASCIIText(), getTexText(), getTextWidget()

The extension of this system to produce other text formats will entail creating a concrete builder for the required format. For example to add a DocConverter that can create a MS Word document. It will be also easy to re-use these converters by another director. For example if you have built your own LATEXeditor, you can re-use these builders to convert the LATEXsource to any one of the supported formats.

22.4 Builder compared with Factory Method

The following can be observed when the structure of the Builder pattern is compared with that of the Factory Method pattern:

- The Builder pattern contains a Director class, which does not exist in the Factory Method pattern. The `construct()`-method that is defined in the Director participant of the Builder pattern, is the equivalent of the `anOperation()`-method as defined in the Creator participant of the Factory Method pattern. Thus the Builder Pattern requires an additional class in which the algorithm describing the process to construct an object is defined.
- The Builder pattern does not have an abstract product as does the Factory Method pattern. It is explained in [1] that when the Builder Pattern it used it is likely that the concrete products are likely to be so diverse that there is little to gain from giving these products a common parent class. This implies that if the application requires the created products to have a common interface, the Builder design pattern is probably not the most suitable pattern to use for the application.

22.5 Builder Pattern Explained

22.5.1 Interaction and Collaboration

Figure 3 is a sequence diagram that illustrates how the participants of the Builder pattern cooperate to create an object and give the client access to the created object. The client has to create or be given a concrete builder capable of constructing the required product. The client also has to have access to a director which defines the process to construct the

required product and knows the correct concrete builder that will assemble the required product. If all this is in place, the client simply issue a command to the director to construct the required product and retrieve it from the concrete builder when completed.

The build process as defined in the director is executed in terms of a series of calls to the concrete builder which will create the product and assemble it by adding parts to it. The process of assembling the product as well as the internal structure of the product is hidden from the client.

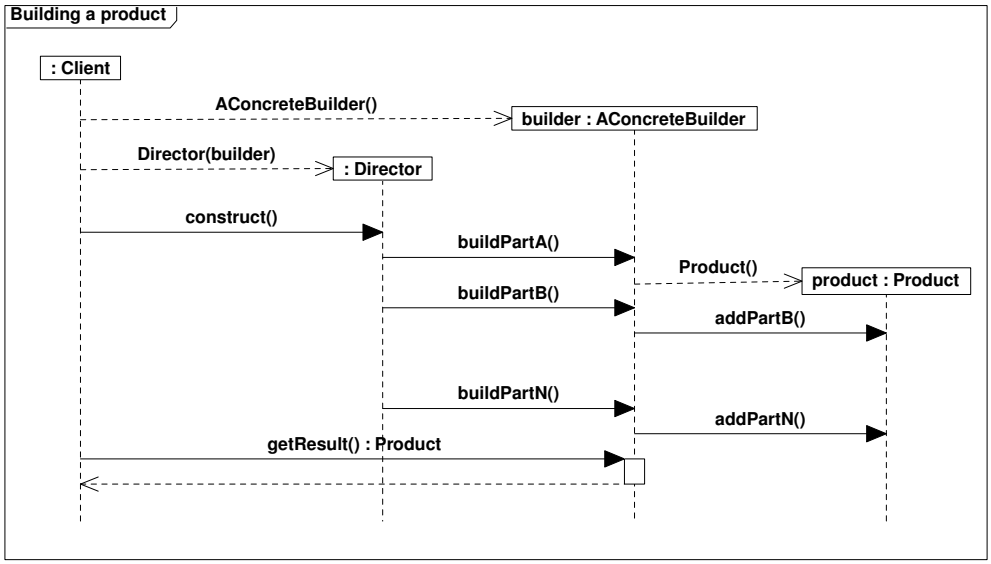

Figure 3: Cooperation of the participants of the Builder Pattern

22.5.2 Improvements achieved

[1] offers the following consequences of the application of the builder design pattern:

- **Variation product's internal representation**
  Builder object provides the director with an abstract interface for constructing the product. The interface lets the builder hide the representation and internal structure of the product. It also hides how the product gets assembled.

- **Separation of code for construction and representation**
  The Builder pattern improves modularity by encapsulating the way a complex object is constructed and represented. Clients need not know anything about the classes that define the product's internal structure.

- **Finer control over the construction process**
  Unlike creational patterns that construct products in one shot, the Builder pattern constructs the product step by step under the director's control. Only when the product is finished does the client retrieve it from the builder.

22.5.3 Common Misconception

Programmers are often under the impression that the application of a complicated algorithm for the construction of multi-part objects constitutes the application of the Builder pattern. However, if this algorithm is implemented in the abstract class of the 'Concrete Builder' objects, it is in fact an implementation of the Factory Method pattern. Thus, we do not agree with [2] who states that "directors can actually be the builder themselves". To be an implementation of the Builder Pattern, this algorithm has to be implemented in a separate 'Director' class.

22.6 Implementation Issues

22.6.1 Creating a product

Each concrete builder has the responsibility to define its own process to create a product in terms of the methods defined in the abstract builder. Each time a product is created, it has to be created from scratch. This can be done either by creating a default product in the first method that is executed by the director or by always having a default product handy.

The option to create a new default product in the first method that is executed by the director is less versatile since it it prescriptive in what method the director should always execute first. However, it is more robust because it is easier to ensure that the copy of the product under construction was not altered in a previous use of the concrete constructor, especially if the same instance of a concrete constructor is re-used by different directors.

The option to have a default product handy can be achieved by instantiating a default product on construction of the concrete builder. This is a viable option if the concrete builder is destroyed after creating an instance of a product and recreated each time it is needed. In situations where the same instance of a concrete builder is re-used, the option to create a product in a method call issued by the director is a better option.

22.6.2 Model for constructing a product

Builders construct their products in step-by-step fashion. Therefore the Builder class interface must be general enough to allow the construction of products for all kinds of concrete builders. A key design issue concerns the model for the construction and assembly process. A model where the results of construction requests are simply appended to the product is usually sufficient. But sometimes you might need access to parts of the product constructed earlier. In that case, more methods to enable communication between the builder and the director is needed to enable the director to retrieve parts, modify them and pass them back to the builder.

22.6.3 Extending a product

Each concrete builder creates a unique product. A concrete builder is allowed to define and add parts to a product that is not controlled by the director. Concrete builders usually

define and maintain instance variables that can eliminate the need for the director to pass many values by means of parameters to the methods that assemble the product.

22.6.4 Varying construction

Since the code for construction and code for representation is separated from one another the design allows exchanging the construction process. Thus, different directors can use the same concrete builders in different ways to build product variants from the same set of parts.

22.7 Related Patterns

Composite
: Builder usually construct composite objects.

Abstract Factory
: Abstract Factory is similar to Builder in that it too may construct complex objects. The primary difference is that the Builder pattern focuses on constructing a complex object step by step. Abstract Factory's emphasis is on families of product objects (either simple or complex). Builder returns the product as a final step, but as far as the Abstract Factory pattern is concerned, the product gets returned immediately.

22.8 Example

In a real application of the builder pattern a director may depend on data that specify the detail of the aggregate. The process of creating a new product involve interpreting the data and issuing commands related to this interpretation to a concrete builder. Different concrete builders are able to create different variations of a product through different implementations of these commands issued by the director. It is important to note that in many cases concrete builders implement only the operations they need and omit the others.

Since the interpretation of data that specifies the aggregate is not part of the pattern, our example assumes a small hard coded aggregate (a soft toy) that has exactly five parts (name, body, stuffing, heart and voice).

Figure 4 is a class diagram of our example implementation. It is a nonsense program that implements the builder structure to illustrate how different directors can use the same concrete builders to create variations of the products that are produced. The different products deliberately have different internal structures and different interfaces to illustrate how this pattern allows for the creation of divers products by the same director when using a different concrete builder.

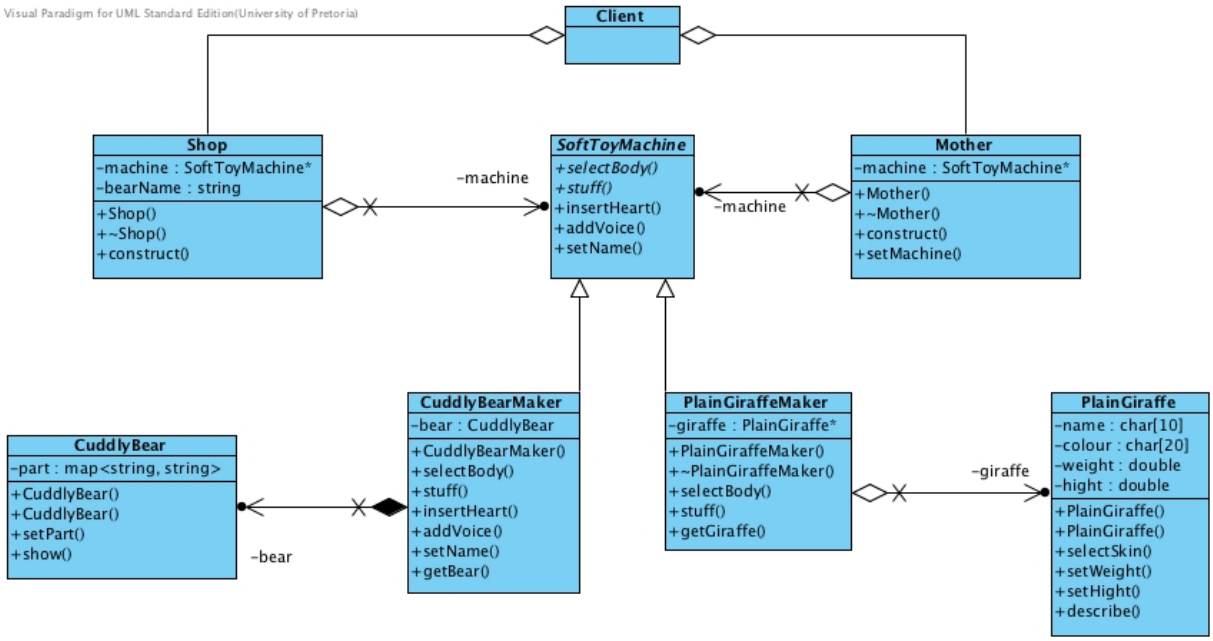

Figure 4: Class Diagram of a soft toy builder nonsense program

Participant	**Entity in application**
Directors	Mother, Shop
Builder	SoftToyMachine
Concrete Builders	PlainGiraffeMaker, CuddlyBearMaker
Products	PlainGiraffe, CuddlyBear
construct()	construct()
buildPart()	selectBody(), stuff(), insertHeart(), addVoice(), setName()
getResult()	getGirrafe(), getBear()

Directors

- The `Mother` class and the `Shop` class are different directors. Both are implemented to use instances of the same concrete builders to create products.
- The `Mother` class has a method that allows the client to equip a `Mother` object with another concrete builder on the fly, while the `Shop` class is instantiated with its concrete builder on construction. The only way to equip a `Shop` object with another concrete builder is by recreating it.
- The `construct()` methods of these two classes are different. The `Shop` class calls all the methods in the interface and make use of parameters to specify high quality products while the `Mother` class omits some of the methods and calls other with default values.

Builder

- The `SoftToyMachine` class act as the builder. It defines the union of all operations needed by the different concrete builders. Those that are not necessarily required are provided with empty implementations.

- This interface specifies the methods that has to be implemented by the concrete builders. All its methods are virtual to allow concrete builders to override them.
- `selectBody()` and `stuff()` are pure virtual. Each concrete builder is required to implement these.
- `insertHeart()`, `addVoice()` and `setName()` has default empty implementations. Usually most of the methods specified in a builder should be specified as such, to allow a concrete builder to omit them if they are not required in the products created by the concrete builder.

Concrete Builders

- The classes `CuddlyBearMaker` and `PlainGiraffeMaker` act as concrete builders.
- Each of these classes provides its own implementation of the building process. They implement the common interface that is defined in `SoftToyMachine` to adapt the methods in the concrete products to the methods defined in `SoftToyMachine`.
- `CuddlyBearMaker` instantiates its product on construction. It is therefore not reusable. For this reason different instances of this class is used by the different directors in this example.
- `PlainGiraffeMaker` instantiates its product in the `selectBody()` method. It is therefore required that each director should call this method first in its `construct()` method. Notice that it deletes any previous instance of the product (if it exists) before creating a new one. This is done to avoid a memory leak. Also notice how its `getGiraffe()` method returns a copy of this product, rather than the product itself. This is done because the product that is created will be destroyed when this concrete builder is reused. The copy is owned by the client and is destroyed by this concrete builder.

Products

- The products are PlainGirrafe and CuddlyBear. You will notice that these products do not share a common abstract interface. This is a distinct feature of the situation where the builder design pattern is deemed appropriate.

Client

- The client constructs instances of directors, concrete builders and products. It then illustrates how the different directors uses the same concrete builders to create different variants of the products. The output is the detail of the products that was created by the different directors.

22.9 Exercise

1. Draw a class diagram showing the participants of the builder pattern to implement dynamic context sensitive creation of menus in a word processing program.

References

[1] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns : elements of reusable object-oriented software*. Addison-Wesley, Reading, Mass, 1995.

[2] Christopher G. Lasater. *Design Patterns*. Wordware Publishing Inc., Texas, USA, 2007.