The builder pattern

COLLABORATORS			
	TITLE : The builder pattern		*REFERENCE* :
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		February 18, 2010	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

1 The builder pattern **1**

1.1 Intent . . . 1

1.2 Solution . . . 1

1.2.1 Responsibility allocation . . . 1

1.2.2 Structure . . . 2

1.2.3 Dynamics . . . 3

1.3 Example applications . . . 4

1.3.1 Non-IT example: fast food outlets . . . 4

1.3.2 UML code generators . . . 4

1.3.3 XML docbook renderers . . . 5

1.3.4 EJB-QL query builders . . . 5

1.3.5 Parsers . . . 5

1.4 Consequences . . . 5

1.5 Implementation guidelines . . . 6

1.6 Related patterns . . . 6

List of Figures

1 The responsibility allocation for the builder pattern 2
2 The structure of the builder pattern 3
3 The dynamics for the builder pattern 4
4 UML code generators often use the Builder pattern 5

1 The builder pattern

The builder pattern is one of the most useful creational patterns introducing clean decoupling of different domains and facilitating the incremental construction of complex objects.

1.1 Intent

- To separate the high-level construction process of a complex, aggregate object from both,
  - the concrete construction of the individual components and
  - the assembling of these components into a product

  such that the same construction process can construct different representations of the same conceptual object.
- To separate the responsibilities of understanding the source domain containing the information from which the objects are constructed from logic required to construct the products.

1.2 Solution

The director consumes the information used to construct a product and issues high-level instructions for the construction process to some or other concrete builder which provides the construction services needed to build a product.

1.2.1 Responsibility allocation

One of the core features of the builder pattern is that it *separates the source and realization domains*. The director only needs to understand the source domain, i.e. the information from which the objects are created while a separate builder is assigned to manage each realization domain.

The following UML use case diagram shows the responsibility allocation across the various components of the builder pattern.

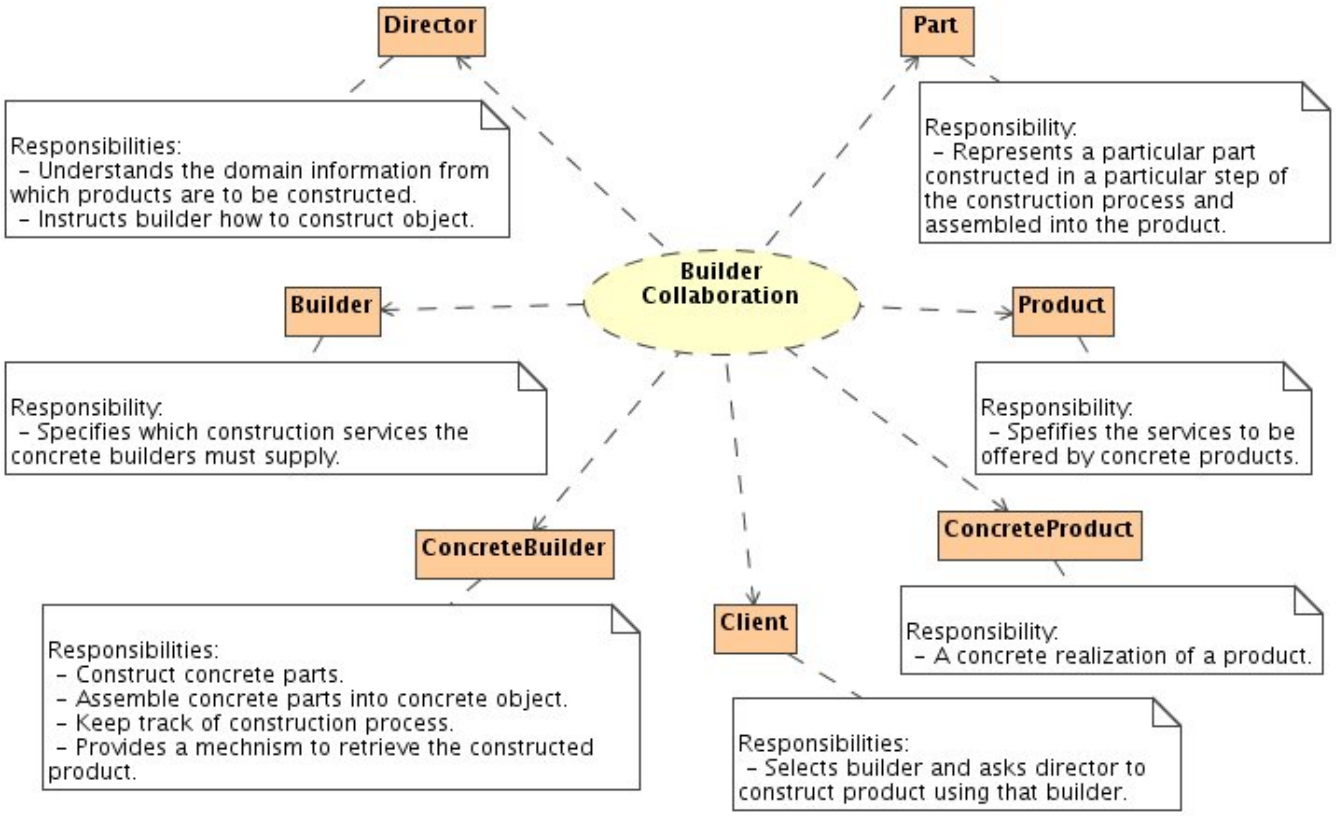

Figure 1: The responsibility allocation for the builder pattern

1.2.2 Structure

The structure of the builder pattern is specified in the following UML class diagram:

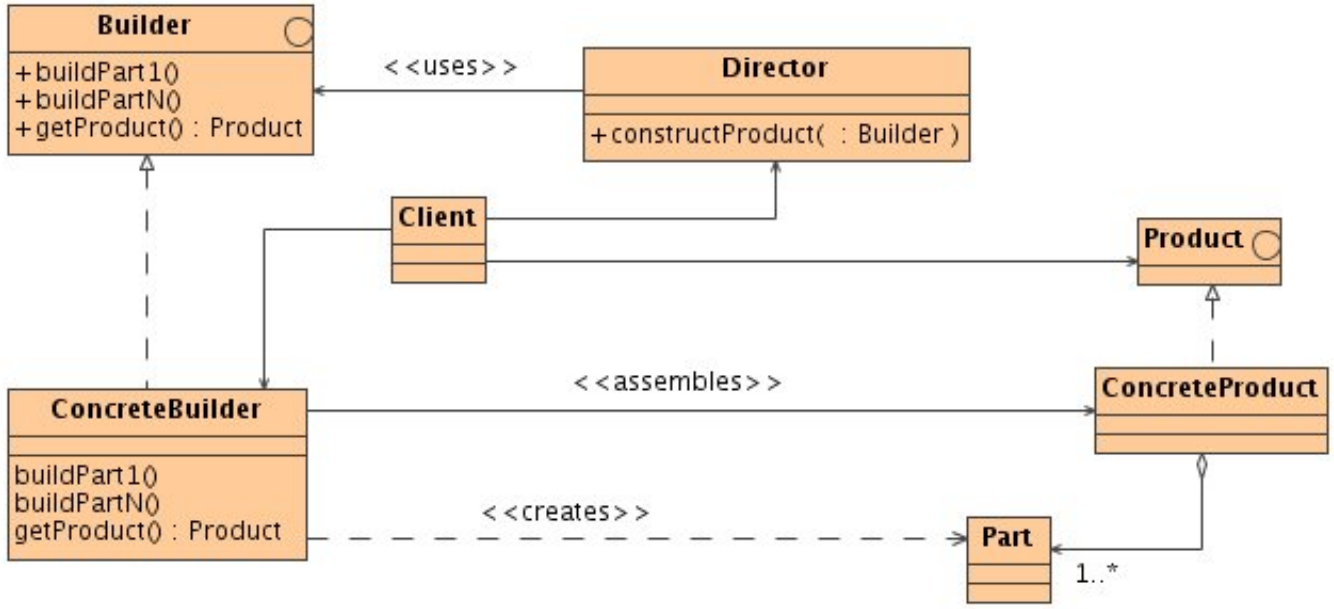

Figure 2: The structure of the builder pattern

Note
The director is completely decoupled from any specific concrete realization of a builder and vice versa. It directs the construction process at a higher, more abstract level.

1.2.3 Dynamics

The client creates a concrete builder and subsequently a director which uses that concrete builder. It then requests the director to construct a product which it does by requesting the builder to construct part for part.

The concrete builder keeps track of the construction process and the constructed product and ultimately the client requests the constructed product from the builder. The dynamics of the builder pattern is shown in the following UML class diagram:

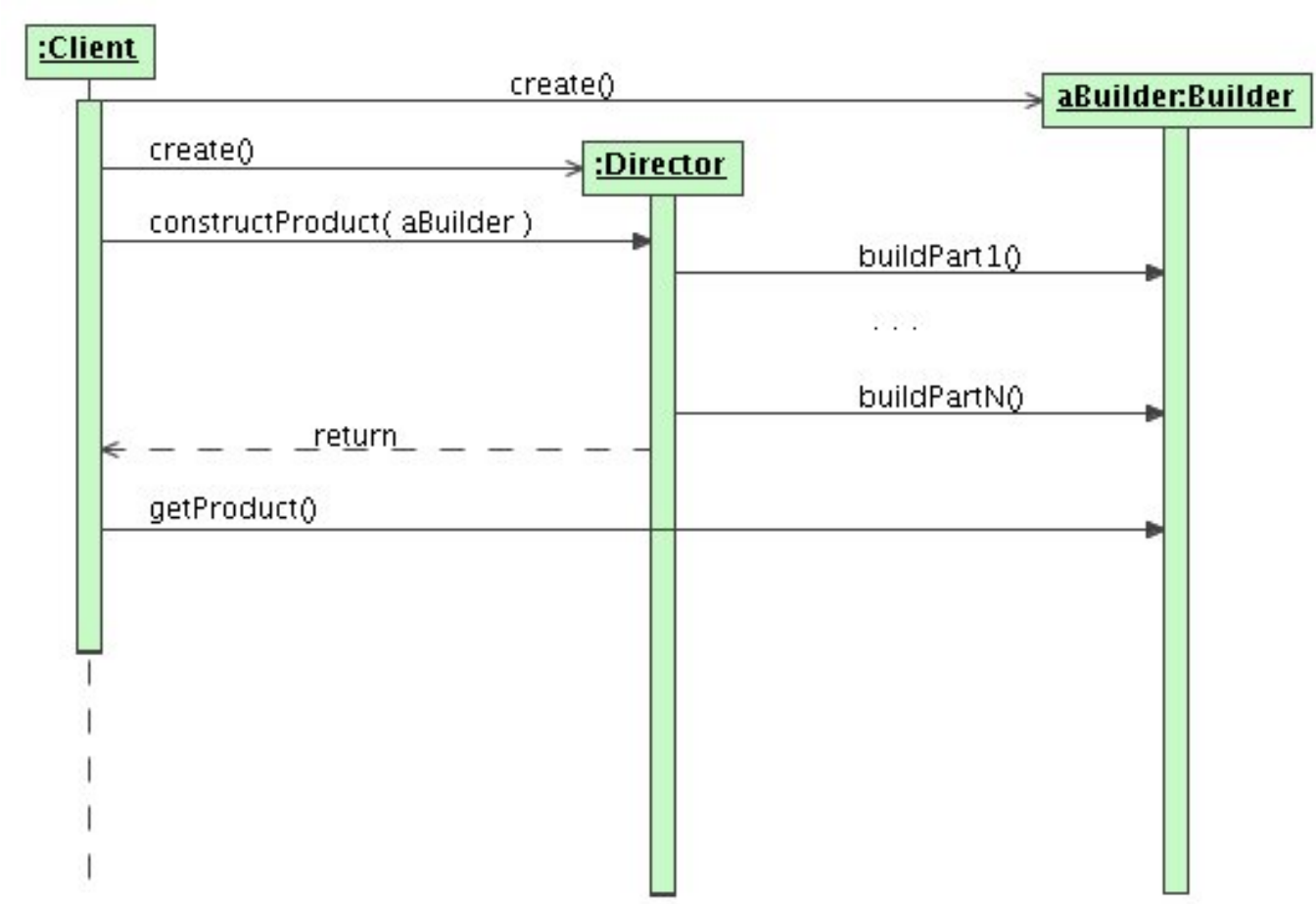

Figure 3: The dynamics for the builder pattern

1.3 Example applications

The builder patterns is used quite widely to abstract the concrete realization of a building process from the more abstract instructions.

1.3.1 Non-IT example: fast food outlets

One can find the use of the builder pattern in real-life processes like those in fast food restaurants which use the builder pattern to construct, for example, childrens meals with a meal, say, consisting of a main item, a side item, a drink, and a toy (e.g., a hamburger, chips, liquid fruit, and water pistol).

There are variations in the content of the children's meal, but the construction process is the same. Whether a customer orders a hamburger, cheeseburger, or chicken drumsticks, the employee at the counter directs the crew to assemble a main item, side item, and toy. These items are then placed in a box with the drink being placed in a separate compartment within the box. Even different fast food outlets placed in a cup and remains outside of the bag. This same process is used at competing restaurants.

1.3.2 UML code generators

From a UML diagram we can generate, for example, an implementation in one of a number of programming languages. Alternatively one can generate an XML schema from a UML class diagram. For each code generator for a particular programming language and the XML schema generator could be a different realization of a builder. This example is illustrated below:

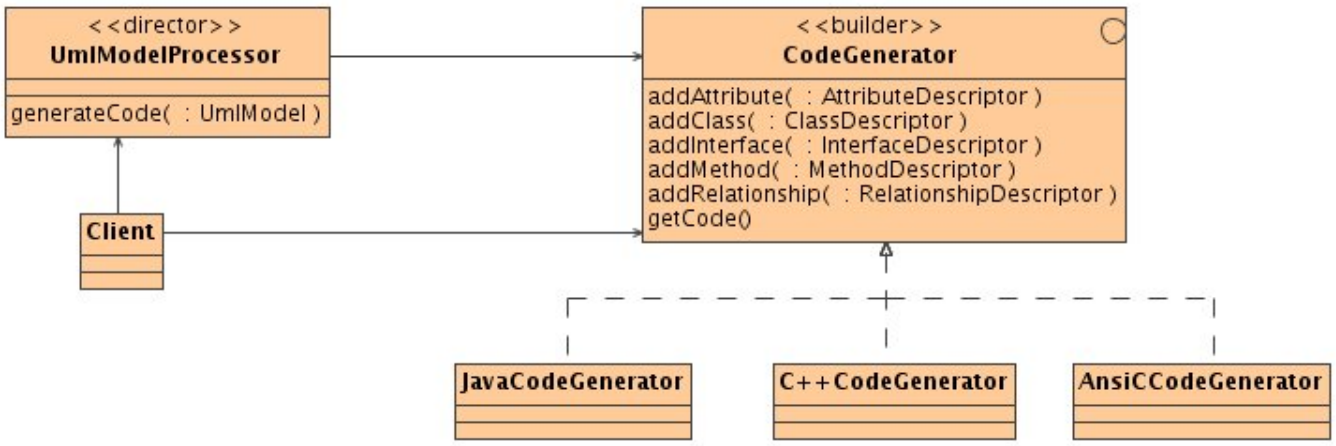

Figure 4: UML code generators often use the Builder pattern

1.3.3 XML docbook renderers

XML docbook renderers read XML docbook tags and generate a rendering in one of a number of technologies. This may include

- HTML,
- PDF,
- LaTeX,
- or even an Open-Office document.

Here the *director* reads the XML tags (e.g. chapter, section or table tags) and requests a concrete builder to construct a rendered dcument from these tags. Different concrete builders construct different realizations of the rendered document.

1.3.4 EJB-QL query builders

When using entity beans with container managed persistence, one specifies the query in *EJB Query Language*, a persistence technology neutral, object-oriented query language. These queries are mapped onto queries in the persistence technology chosen for that entity bean.

For example, if a relational database is used, then SQL queries are constructed from EJB-QL queries. On the other hand, if the relational databose is replaced by an object database, then the SQL-builder can be replaced by an OQL builder.

1.3.5 Parsers

Parsers typically read an input domain and construct a representation in another domain. For example, parsers of a mathematical expression decode the input domain and construct an algorithm for that expression. A builder pattern will separate the expression decoding from the algorithm construction and facilitates different algorithm constructors for different programming languages.

1.4 Consequences

- **Separation of source and destination domains** The director has to only understand the source of the information from which the product is created while the builders only need to understand the construction of the product parts as well as the assembling of these parts into products.

- **Pluggable builders** The pattern allows you to plug in different concrete builders which construct potentially different realizations of a product.
- **Pluggable directors** The pattern allows you to plug in different concrete directors which construct products from potentially different concrete dource domains (e.g. construct a Java data object from either a UML class diagram or a XML schema type).
- **Modeling incremental product construction and assembly** The builder models step-for-step product construction and assembly directly. As such the builder pattern provides control for the construction process.

1.5 Implementation guidelines

The implementation of the builder pattern is for object-oriented languages a straight-forward mapping of the UML diagrams onto code.

- **Should Builder be an interface or an abstract class?** We would recommend to always have the contract for the builder represented by an interface. You could use, in addition to this, an abstract class to encapsulate certain commonalities across some of the concrete builders.
- **The builder interface is determined by the source domain** It is important that the `Builder` interface design is not driven by the requirements of a particular builder constructing a specific realization of a product. Instead the `Builder` should specify services as required by the source domain, i.e. as required by the `Director`.
- **Products may have little in common** The different products produced by different builders may, from a user perspective, have have at times only very little in common and as such the product interface may specify very few functionalities, perhaps even only the ability to obtain a streamable byte representation of the product.

1.6 Related patterns

- **Template method** In many ways the *builder* pattern is a specialization of the *template method* pattern. Both patterns define a high-level algorithm where the individual steps of the algorithm may be realized in different ways.
- **Singleton** Both, the `director` and the `builder` may be a singleton.
- **Abstract factory** Abstract factories may delegate the responsibility of constructing complex components of a family of classes (e.g. instances of classes from a framework) to a builder.