**When Will It Be Needed?**Now, what does Builder class do? Separate the construction of a complex object from its representation so that the same construction process can create different representations.

A reader for the RTF (Rich Text Format) document exchange format should be able

to convert RTF to many text formats. The reader might convert RTF documents into

plain ASCII text or into a text widget that can be edited interactively. The problem,

however, is that the number of possible conversions is open-ended. So it should

be easy to add a new conversion without modifying the reader.

A solution is to configure the RTFReader class with a TextConverter object that

converts RTF to another textual representation. As the RTFReader parses the RTF

document, it uses the TextConverter to perform the conversion. Whenever the

RTFReader recognizes an RTF token (either plain text or an RTF control word),

it issues a request to the TextConverter to convert the token. TextConverter

objects are responsible both for performing the data conversion and for

representing the token in a particular format.   
  
Subclasses of TextConverter specialize in different conversions and formats. For

example, an ASCIIConverter ignores requests to convert anything except plain text.

A TeXConverter, on the other hand, will implement operations for all requests

in order to produce a TeX representation that captures all the stylistic

information in the text. A TextWidgetConverter will produce a complex user

interface object that lets the user see and edit the text

**Each kind of converter class takes the mechanism for creating and assembling a**

**complex object and puts it behind an abstract interface.** The converter is separate

from the reader, which is responsible for parsing an RTF document.

The Builder pattern captures all these relationships. Each converter class is

called a builder in the pattern, and the reader is called the director. Applied

to this example, the Builder pattern separates the algorithm for interpreting

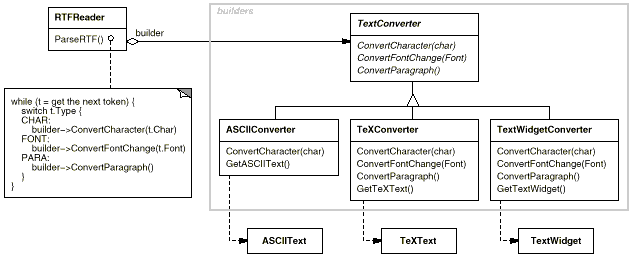
a textual format (that is, the parser for RTF documents) from how a converted

format gets created and represented. This lets us reuse the RTFReader's parsing

algorithm to create different text representations from RTF documents—just

configure the RTFReader with different subclasses of TextConverter.

**RTFReader:**



Now, you will see different Class Diagram symbols included in it:

Like, the relationship between TextConverter and (ASCIIConverter, TeXConverter, TextWidgetConverter)

**That relationship is called Realization/Implementation.**

a relationship between two model elements, in which one model element implements/executes the behavior that the other model element specifies.

Now, there is also Aggregation relationship between RTFReader and TextConverter.

Here, it means RTFReader has TextCoverter but the vice versa is not true.

Now, Note the following things:

RTFHeader constructs the ASCIIConverter, TexConverter, TextWidgets accordingly. **(using the Builder interface TextConverter)**

The final products are ASCIIText, TexText, TextWidget.

(Now, these represent the complex objects under construction. Actually TextConverter is the complex class under construction)

**Applicability:**

Use the Builder pattern when

* the algorithm for creating a complex object should be independent of the

parts that make up the object and how they're assembled.

* the construction process must allow different representations for the

object that's constructed.

**Structure:**

1. **Builder (TextConverter)**
   1. specifies an abstract interface for creating parts of a Product object.
   2. This abstract base class defines all of the steps that must be taken in order to correctly create a product.
   3. Each step is generally abstract as the actual functionality of the builder is carried out in the concrete subclasses.
   4. The GetProduct method is used to return the final product.
2. **ConcreteBuilder (ASCIIConverter, TeXConverter, TextWidgetConverter)**
   1. constructs and assembles parts of the product by implementing the

Builder interface.

* 1. defines and keeps track of the representation it creates.
  2. provides an interface for retrieving the product (e.g., GetASCIIText,

GetTextWidget).

1. **Director (RTFReader)**
   1. constructs an object using the Builder interface.
2. **Product (ASCIIText, TeXText, TextWidget)**
   1. represents the complex object under construction. ConcreteBuilder

builds the product's internal representation and defines the process

by which it's assembled. (It defines the type of the complex object that is to be generated by the builder pattern.)

* 1. includes classes that define the constituent parts, including

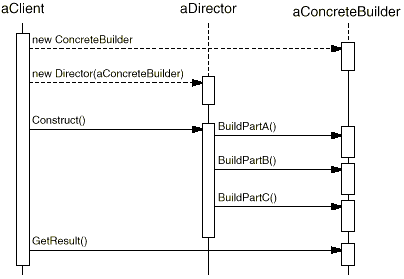
interfaces for assembling the parts into the final result.

**Collaborations:**

* The client creates the Director object and configures it with the desired

Builder object.

* Director notifies the builder whenever a part of the product should be built
* Builder handles requests from the director and adds parts to the product. **(Adding parts in proper order is important)**
* The client retrieves the product from the builder



interface HousePlan

{

public void setBasement(String basement);

public void setStructure(String structure);

public void setRoof(String roof);

public void setInterior(String interior);

}

**/\*Final Product class\*/**

class House implements HousePlan

{

private String basement;

private String structure;

private String roof;

private String interior;

public void setBasement(String basement)

{

this.basement = basement;

}

public void setStructure(String structure)

{

this.structure = structure;

}

public void setRoof(String roof)

{

this.roof = roof;

}

public void setInterior(String interior)

{

this.interior = interior;

}

}

**/\*The Abstract Builder Class\*/**

interface HouseBuilder

{

public void buildBasement();

public void buildStructure();

public void bulidRoof();

public void buildInterior();

public House getHouse();

}

**/\* The Concrete Builder class\*/**

class IglooHouseBuilder implements HouseBuilder

{

private House house;

public IglooHouseBuilder()

{

this.house = new House();

}

public void buildBasement()

{

house.setBasement("Ice Bars");

}

public void buildStructure()

{

house.setStructure("Ice Blocks");

}

public void buildInterior()

{

house.setInterior("Ice Carvings");

}

public void bulidRoof()

{

house.setRoof("Ice Dome");

}

public House getHouse()

{

return this.house;

}

}

**/\*Another Concrete Builder class\*/**

class TipiHouseBuilder implements HouseBuilder

{

private House house;

public TipiHouseBuilder()

{

this.house = new House();

}

public void buildBasement()

{

house.setBasement("Wooden Poles");

}

public void buildStructure()

{

house.setStructure("Wood and Ice");

}

public void buildInterior()

{

house.setInterior("Fire Wood");

}

public void bulidRoof()

{

house.setRoof("Wood, caribou and seal skins");

}

public House getHouse()

{

return this.house;

}

}

**/\*Director class\*/**

**/\***

**The client creates the Director object and configures it with the desired**

**Builder object.**

**\*/**

**/\*See, that Client Class configures the Director Object CivilEngineer with the Desired Builder HouseBuilder object\*/**

class CivilEngineer

{

private HouseBuilder houseBuilder;

**//configures the director class with the desired Builder Object**

public CivilEngineer(HouseBuilder houseBuilder)

{

this.houseBuilder = houseBuilder;

}

**//return the final product**

public House getHouse()

{

return this.houseBuilder.getHouse();

}

**//Now, from client class, it must be called before obtaining the final product here**

public void constructHouse()

{

this.houseBuilder.buildBasement();

this.houseBuilder.buildStructure();

this.houseBuilder.bulidRoof();

this.houseBuilder.buildInterior();

}

}

**/\*Client class\*/**

class Builder

{

public static void main(String[] args)

{

HouseBuilder iglooBuilder = new IglooHouseBuilder();

CivilEngineer engineer = new CivilEngineer(iglooBuilder);

engineer.constructHouse();

House house = engineer.getHouse();

System.out.println("Builder constructed: "+ house);

}

}

**The Same Builder Design in C++**

#include<bits/stdc++.h>

using namespace std;

/\*Abstract product class\*/

class HousePlan

{

/\*that's an abstract class\*/

public:

virtual void set\_basement(string)=0;

virtual void set\_structure(string)=0;

virtual void set\_roof(string)=0;

virtual void set\_interior(string)=0;

//virtual void set\_exterior(string)=0;

};

/\*Base product class\*/

class House:public HousePlan

{

private:

string basement;

string structure;

string roof;

string interior;

public:

/\*this were pure virtual functions in the Abstract product class

\* So, this functions must be overridden. Otherwise, House will also become an abstract class\*/

void set\_basement(string );

void set\_structure(string );

void set\_roof(string );

void set\_interior(string );

void display();

};

void House::set\_basement(string basement)

{

this->basement=basement;

}

void House::set\_structure(string structure)

{

this->structure=structure;

}

void House::set\_roof(string roof)

{

this->roof=roof;

}

void House::set\_interior(string interior)

{

this->interior=interior;

}

void House::display()

{

cout<<"The basement is set to: "<<basement<<endl;

cout<<"The structure is set to: "<<structure<<endl;

cout<<"The roof is set to: "<<roof<<endl;

cout<<"The interior is set to: "<<interior<<endl;

}

/\*Abstract Builder class\*/

class HouseBuilder

{

public:

virtual void build\_basement()=0;

virtual void build\_structure()=0;

virtual void build\_roof()=0;

virtual void build\_interior()=0;

virtual House\* get\_house()=0;

};

/\*Builder class\*/

class TipiBuilder: public HouseBuilder

{

/\*Now, you will see the difference\*/

/\*In AbstractFactory design pattern, the concreteFactory classes contain the same data members

\* as Product. Because, user passes the product specification.

\* Here, User chooses a Builder class for providing functiality

\* That Builder class will serve a representation (a fixed one) of the product

\* So, here, In Builder design pattern, we will rather have a pointer to the house class

\* Again, another difference, since, there is only one product here, and it is the builder's

\* responsibility to provide a different representation of that class, we will have a pointer to the base product class

\* in fact, in java, the base product class only implements interface to provide combined set of functionality

\* here, in c++, there is no concept of interface. But, we can have multiple inheritance

\* though, it would not be theoritically same. Because, interface supports has a relationship while it is implemented

\* by a class. While, with abstract class it is a is a relationship

\*/

private:

House \*house;

public:

TipiBuilder();

void build\_basement();

void build\_structure();

void build\_roof();

void build\_interior();

House\* get\_house();

};

TipiBuilder::TipiBuilder()

{

this->house=new House;

}

void TipiBuilder::build\_basement()

{

house->set\_basement("Wooden Poles");

}

void TipiBuilder::build\_structure()

{

house->set\_structure("Wood and ice");

}

void TipiBuilder::build\_roof()

{

house->set\_roof("Fire wood");

}

void TipiBuilder::build\_interior()

{

house->set\_interior("Wood, caribou and seal skins");

}

House\* TipiBuilder::get\_house()

{

return this->house;

}

class IglooBuilder: public HouseBuilder

{

private:

House \*house;

public:

IglooBuilder();

void build\_basement();

void build\_structure();

void build\_roof();

void build\_interior();

House\* get\_house();

};

IglooBuilder::IglooBuilder()

{

this->house = new House();

}

void IglooBuilder::build\_basement()

{

house->set\_basement("Ice Bars");

}

void IglooBuilder::build\_structure()

{

house->set\_structure("Ice Blocks");

}

void IglooBuilder::build\_interior()

{

house->set\_interior("Ice Carvings");

}

void IglooBuilder::build\_roof()

{

house->set\_roof("Ice Dome");

}

House\* IglooBuilder::get\_house()

{

return this->house;

}

/\*Director class\*/

class CivilEngineer

{

private:

HouseBuilder \*houseBuilder;

//configures the director class with the desired Builder Object

public:

CivilEngineer(HouseBuilder \*);

House\* get\_house();

void construct\_house();

};

CivilEngineer::CivilEngineer(HouseBuilder \*houseBuilder)

{

this->houseBuilder=houseBuilder;

}

House\* CivilEngineer::get\_house()

{

houseBuilder->get\_house();

}

void CivilEngineer::construct\_house()

{

/\*Now, see, the order in which director chooses to build a product step by step, is director specific. Client

\* Only cares about Product. And, again, it is director specific. Not builder specific. Builder provides the necessary abstraction

\* between the actual director class and how that class builds a product. Nothing else\*/

houseBuilder->build\_basement();

houseBuilder->build\_structure();

houseBuilder->build\_roof();

houseBuilder->build\_interior();

};

int main()

{

HouseBuilder \*iglooBuilder=new IglooBuilder();

CivilEngineer engineer(iglooBuilder);

//passed the pointer as a ddescription that how client wants his house to be built

engineer.construct\_house();

House \*house=engineer.get\_house();

//since, it will return a pointer. We could try to improvise and return a reference

//for now, let it be pointer

house->display();

return 0;

}

* **Consequences Of Builder Pattern:**

1. It lets you vary a product's internal representation. The Builder object

provides the director with an abstract interface for constructing the

product. The interface **(The abstract builder class?)**lets the builder hide the representation and internal structure of the product. It also hides how the product gets assembled. Because the product is constructed through an abstract interface, all you have to do to change the product's internal representation is define a new kind of builder.

1. **It isolates code for construction and representation.** The Builder pattern

improves modularity by encapsulating the way a complex object is constructed

and represented. Clients needn't know anything about the classes that define

the product's internal structure; such classes don't appear in Builder's interface.

Each ConcreteBuilder contains all the code to create and assemble a

particular kind of product. The code is written once; then different

Directors can reuse it to build Product variants from the same set of parts.

In the earlier RTF example, we could define a reader for a format other

than RTF, say, an SGMLReader, and use the same TextConverters to generate

ASCIIText, TeXText, and TextWidget renditions of SGML documents.

(So, since director class (RTFReader) is separate that the Builder class (TextConverter), we could define a new Director class (a reader for a format other than RTF, sa SGMLReader, and use the same builder class (TextCoverters) to generate ASCIIText, TeXText, and TextWidget renditions of SGML documents)

1. **It gives you 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 director retrieve it from the builder. Hence the Builder interface reflects the process of constructing the product more than other creational patterns. This gives you finer control over the construction process and consequently the internal structure of the resulting product.

(Like, in my c++ example, only CivilEngineer knows the step by step construction of a House. Client does not need to know that)