**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

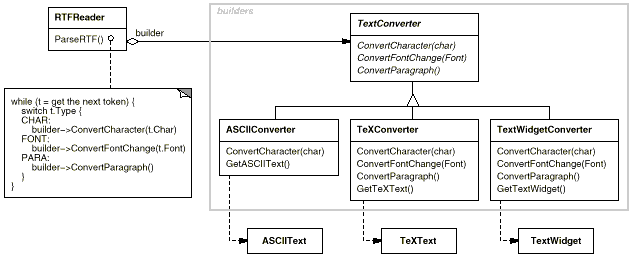
a textual format (that is, the parser for RTF documents) from how a converted

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

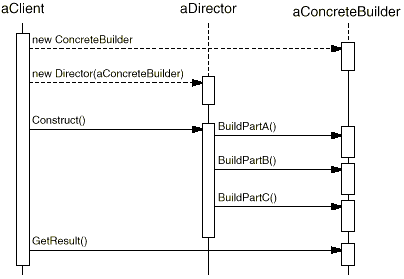
interfaces for assembling the parts into the final result.

**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);

}

}

**A Similar Example in c++  
  
#include <iostream>**

**using namespace std;**

**/\* Interface that is implemented by the final product\*/**

class HousePlan

{

public:

**//all members are pure virtual function**

  virtual void setWindow(string window)=0;

  virtual void setDoor(string door)=0;

  virtual void setBathroom(string bathroom)=0;

  virtual void setKitchen(string kitchen)=0;

  virtual void setFloor(string floor)=0;

};

**/\* Concrete class for the HousePlan interface \*/**

**/\*This is will be the final product returned\*/**

class House:public HousePlan

{

private :

  string window, door, kitchen, bathroom, floor;

public:

  void setWindow(string window)

  {

   this->window = window;

  }

  void setDoor(string door)

  {

   this->door = door;

  }

  void setBathroom(string bathroom)

  {

   this->bathroom = bathroom;

  }

  void setKitchen(string kitchen)

  {

   this->kitchen = kitchen;

  }

  void setFloor(string floor)

  {

   this->floor = floor;

  }

};

**/\* Builder Class \*/**

**class HouseBuilder**

**{**

**public:**

**/\* Abstract functions to build parts \*/**

virtual void buildWindow()=0;

  virtual void buildDoor()=0;

  virtual void buildKitchen()=0;

  virtual void buildBathroom()=0;

  virtual void buildFloor()=0;

**/\* The product is returned by this function \*/**

virtual House\* getHouse()=0;

**/\*A pointer to the House\*/**

**};**

**/\* Concrete class for the builder interface \*/**

class LavishHouse:public HouseBuilder

{

private:

  House \*house;

public:

  LavishHouse()

  {

   house = new House();

  }

  void buildWindow()

  {

   house->setWindow("French Window");

  }

  void buildDoor()

  {

   house->setDoor("Wooden Door");

  }

  void buildBathroom()

  {

   house->setBathroom("Modern Bathroom");

  }

  void buildKitchen()

  {

   house->setKitchen("Modular Kitchen");

  }

  void buildFloor()

  {

   house->setFloor("Wooden Floor");

  }

  House\* getHouse()

  {

   return this->house;

  }

};

**/\* Another Concrete class for the builder interface \*/**

class NormalHouse:public HouseBuilder

{

private:

  House \*house;

public:

  NormalHouse()

  {

   house = new House();

  }

  void buildWindow()

  {

   house->setWindow("Normal Window");

  }

  void buildDoor()

  {

   house->setDoor("Metal Door");

  }

  void buildBathroom()

  {

   house->setBathroom("Regular Bathroom");

  }

  void buildKitchen()

  {

   house->setKitchen("Regular Kitchen");

  }

  void buildFloor()

  {

   house->setFloor("Mosaic Floor");

  }

  House\* getHouse()

  {

   return this->house;

  }

};

**/\* The Director. Constructs the house \*/**

class Contractor

{

private:

  HouseBuilder \*houseBuilder;  
 **//base class object**

public:

  Contractor(HouseBuilder \*houseBuilder)

  {

   this->houseBuilder = houseBuilder;

  }

House \*getHouse()

  {

   return houseBuilder->getHouse();

  }

void buildHouse()

  {

   houseBuilder->buildWindow();

   houseBuilder->buildDoor();

   houseBuilder->buildBathroom();

   houseBuilder->buildKitchen();

   houseBuilder->buildFloor();

  }

};

**/\* Example on how to use the Builder design pattern \*/**

int main()

{

  HouseBuilder \*lavishHouseBldr = new LavishHouse();

  HouseBuilder \*normalHouseBldr = new NormalHouse();

**/\*It’s like choosing a HouseBuilderPlan and giving the plan to (CivilEngineer)Contractor to execute the plan and get the final product: House\*/**

Contractor \*ctr1 = new Contractor(lavishHouseBldr);

  Contractor \*ctr2 = new Contractor(normalHouseBldr);

  ctr1->buildHouse();

  House \*house1 = ctr1->getHouse();

  cout<<"Constructed: "<<house1;

ctr2->buildHouse();

  House \*house2 = ctr2->getHouse();

  cout<<"Constructed: "<<house2;

}

* **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 Contractor knows the step by step construction of a House. Client does not need to know that)