Builder - Object Creational

Intent (意图)

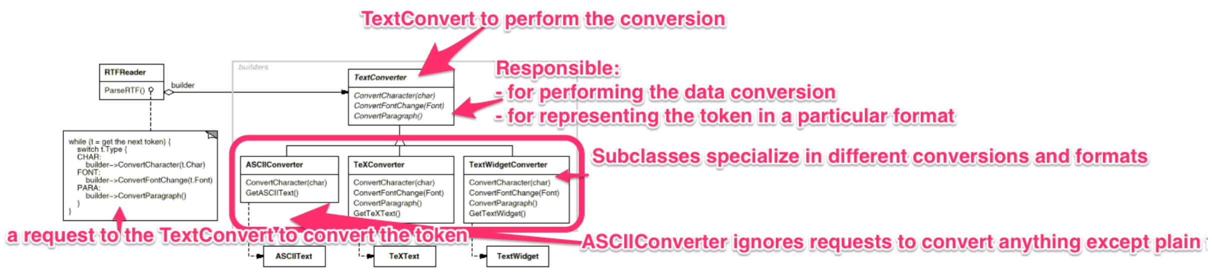
Separate the construction of a complex object from its representation so that the same construction process can create different representations.

Motivation (动机)

- A reader for the RTF(Rich Text Format) document exchange format should be able to convert RTF to many text formats.
- The problem, 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:

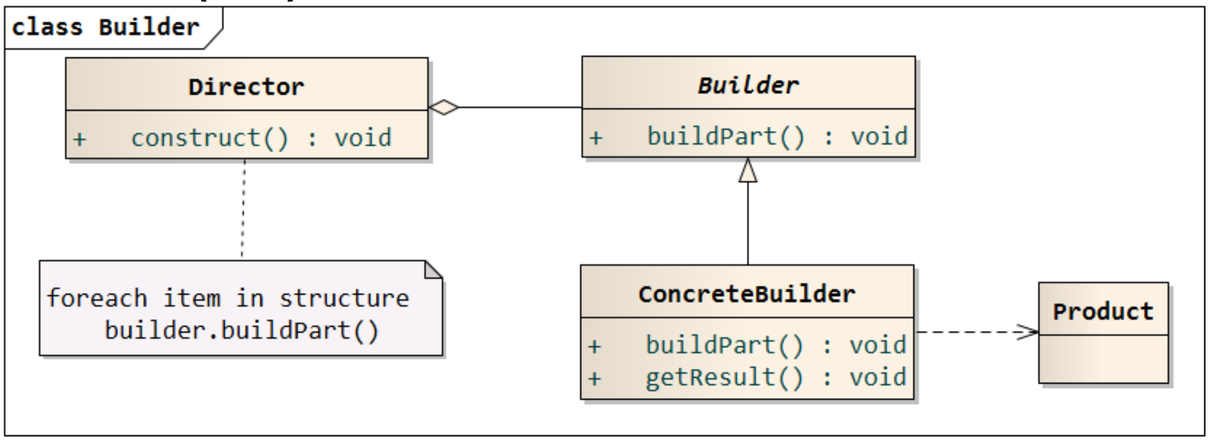
- configure the RTFReader class with a TextConverter object that converts RTF to another textual representation.
- As the RTFReader parses the RTF document, it uses the TextConverter to perform the conversion.


TextConvert to perform the conversion
Responsible:
- for performing the data conversion
- for representing the token in a particular format
Subclasses specialize in different conversions and formats
a request to the TextConvert to convert the token
ASCIIConverter ignores requests to convert anything except plain
RTFReader
ParseRTF()
builder
builders
TextConverter
ConvertCharacter(char)
ConvertFontChange(Font)
ConvertParagraph()
while (t = get the next token) {
switch t.Type {
CHAR:
builder->ConvertCharacter(t.Char)
FONT:
builder->ConvertFontChange(t.Font)
PARA:
builder->ConvertParagraph()
}
}
ASCIIConverter
ConvertCharacter(char)
GetASCIIText()
TeXConverter
ConvertCharacter(char)
ConvertFontChange(Font)
ConvertParagraph()
GetTeXText()
TextWidgetConverter
ConvertCharacter(char)
ConvertFontChange(Font)
ConvertParagraph()
GetTextWidget()
ASCIIText
TeXText
TextWidget

Applicability (适用性)

- the algorithm for creating a complex object should be independent of the parts that make up the object and how they'er assembled.
- the construction process must allow different representation for the object that's constructed.

Structure (结构)



Participants (参与者)

- **Builder (TextConverter)**

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

- **ConcreteBuilder (ASCIIConverer, TeXConverter, TextWidgetConverter)**

  - 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 (e.g., GetASCIIText, GetTextWidget)

- **Director (RTFReader)**

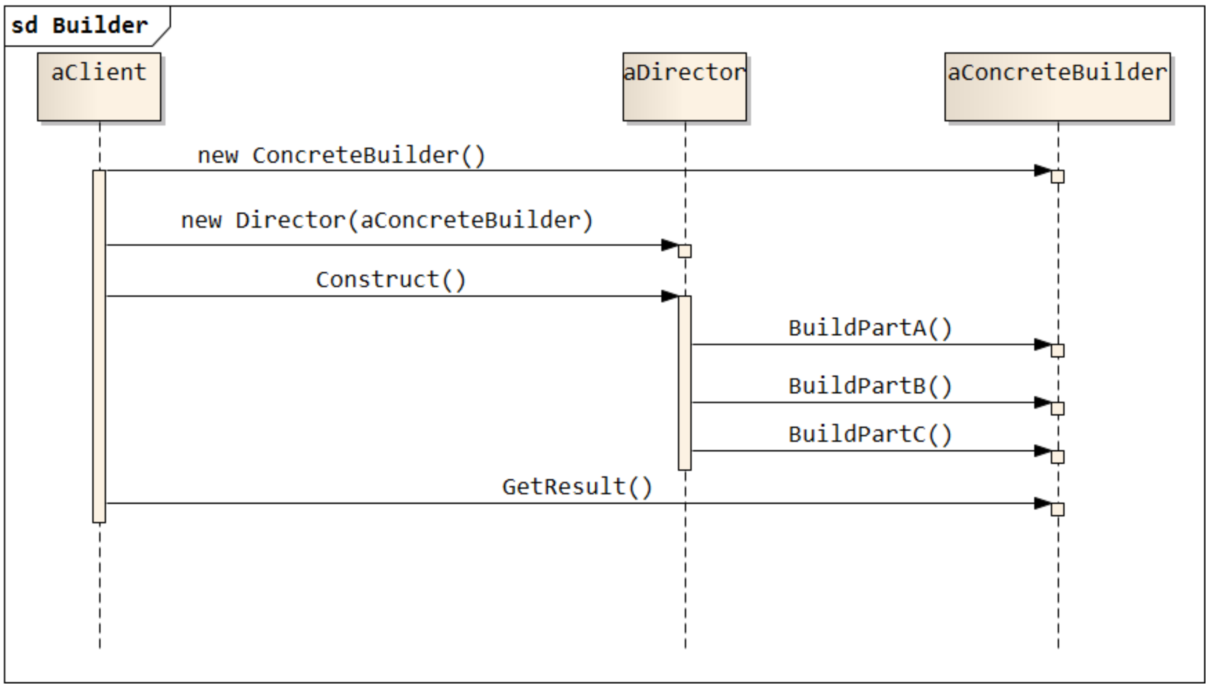
  - constructs an object using the Builder interface.

- **Product (ASCIIText, TexText, TextWidget)**

  - 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.

Collaborations (协作)

- 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.
- The client retrieves the product from the builder.



Consequences (结果)

- It lets you vary a product's internal representation.
- It isolates code for construction and representation.
- It gives you finer control over the construction process.

Implementation (实现)

- Assembly and construction interface.

A key design issue concerns the model for the construction and assembly process.

- Why no abstract class for products?

Because **the client** usually configures **the director** with the proper concrete builder, the client is in a position to know which concrete subclass of Builder is in use and can handle its products accordingly.

- Empty methods as default in Builder.

In C++, the build methods are intentionally not declared pure virtual member functions.

Sample Code (代码示例)

```
class MazeBuilder
{
public:
//This interface can create three things:
virtual void buildMaze();
virtual void buildRoom(int room);
virtual void buildDoor(int roomFrom, int roomTo);

//returns the maze to the client.
virtual Maze *getMaze() { return 0 };
protected:
MazeBuilder();
};

//Director
Maze *MazeGame::createMaze(MazeBuilder &builder)
{
builder.buildMaze();

builder.buildRoom(1);
builder.buildRoom(2);
builder.buildDoor(1, 2);

return builder.getMaze();
}

Maze *MazeGame::createComplexMaze(MazeBuilder &builder)
{
builder.buildRoom(1);
// ...
builder.buildRoom(1001);

return builder.getMaze();
}

class StandardMazeBuilder : public MazeBuilder
{
public:
StandardMazeBuilder();

virtual void buildMaze() override;
virtual void buildRoom(int room) override;
virtual void buildDoor(int roomFrom, int roomTo) override;

virtual Maze *getMaze() override;

private:
Direction commonWall(room *, room *);
Maze *_currentMaze;
};

StandardMazeBuilder::StandardMazeBuilder()
: _currentMaze(0)
{

}

//instantiates a Maze
void StandardMazeBuilder::buildMaze()
{
_currentMaze = new Maze;
}
```

```
//return to the client
void StandardMazeBuilder::getMaze()
{
return _currentMaze;
}

//creates a room and builds the walls around it.
void StandardMazeBuilder::buildRoom(int room)
{
if (!_currentMaze->getRoom(n)) {
Room *r = new Room(n);
_currentMaze->addRoom(r);

r->SetSide(North, new Wall);
r->SetSide(South, new Wall);
r->SetSide(East, new Wall);
r->SetSide(West, new Wall);
}
}

//to build a door between two rooms.
void StandardMazeBuilder::buildDoor(int roomFrom, int roomTo)
{
Room *r1 = _currentMaze->getRoom(roomFrom);
Room *r2 = _currentMaze->getRoom(roomTo);
Door *d = new Door(r1, r2);

r1->SetSide(commonWall(r1, r2), d);
r2->SetSide(commonWall(r2, r1), d);
}


void clientUse()
{
//Clients can now use CreateMaze in conjunction with
StandardMazeBuilder to create a maze.
Maze *maze;
MazeGame game;
StandardMazeBuilder builder;

maze = game.CreateMaze(builder);
}
```

Known Uses (已知应用)

...

Related Patterns (相关模式)

Abstract Factory is similar to **Builder** in that it too may construct complex objects.

- **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.
- **Abstract Factory**, the product gets returned immediately.

A **Composite** is what the **Builder** often builds.