

## Intent

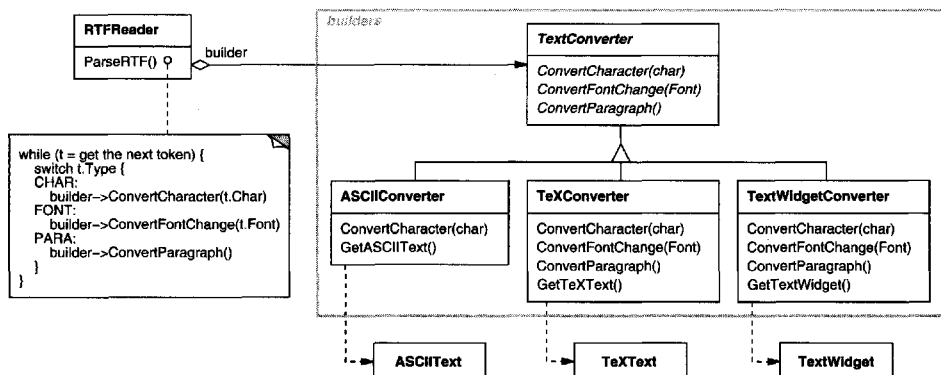
Separate the construction of a complex object from its representation so that the same construction process can create different representations.

## Motivation

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.

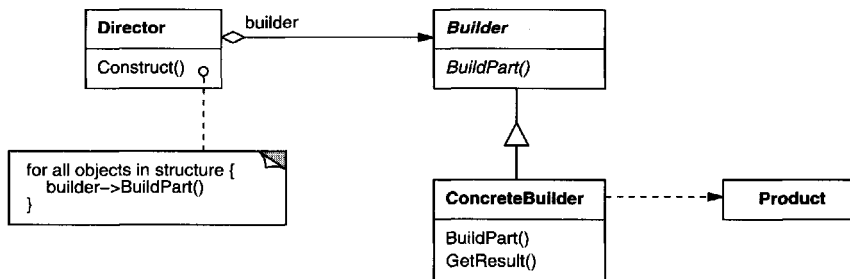
The Builder pattern captures all these relationships. Each converter class is called a **builder** in the pattern, and the reader is called the **director**. Applied to this example, the Builder pattern separates the algorithm for interpreting a textual format (that is, the parser for RTF documents) 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.

## 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



## Participants

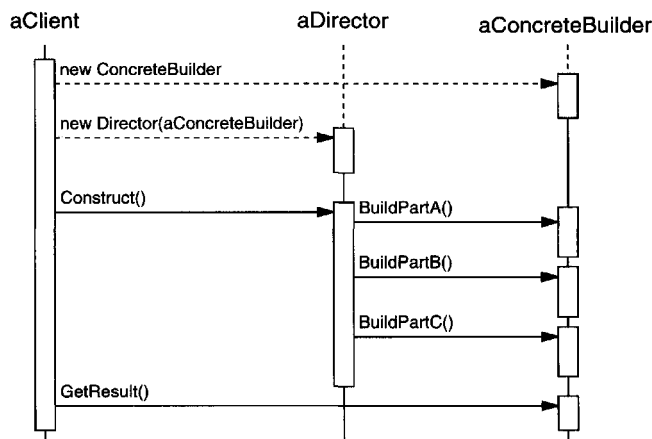
- **Builder** (TextConverter)
  - specifies an abstract interface for creating parts of a Product object.

- **ConcreteBuilder** (ASCIIConverter, 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

- 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.

The following interaction diagram illustrates how Builder and Director cooperate with a client.



## Consequences

Here are key consequences of the 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 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.
2. *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.
3. *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.

## Implementation

Typically there's an abstract Builder class that defines an operation for each component that a director may ask it to create. The operations do nothing by default. A ConcreteBuilder class overrides operations for components it's interested in creating.

Here are other implementation issues to consider:

1. *Assembly and construction interface.* 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. In the RTF example, the builder converts and appends the next token to the text it has converted so far.

But sometimes you might need access to parts of the product constructed earlier. In the Maze example we present in the Sample Code, the MazeBuilder

interface lets you add a door between existing rooms. Tree structures such as parse trees that are built bottom-up are another example. In that case, the builder would return child nodes to the director, which then would pass them back to the builder to build the parent nodes.

2. *Why no abstract class for products?* In the common case, the products produced by the concrete builders differ so greatly in their representation that there is little to gain from giving different products a common parent class. In the RTF example, the `ASCIIText` and the `TextWidget` objects are unlikely to have a common interface, nor do they need one. 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.
3. *Empty methods as default in Builder.* In C++, the build methods are intentionally not declared pure virtual member functions. They're defined as empty methods instead, letting clients override only the operations they're interested in.

## Sample Code

We'll define a variant of the `CreateMaze` member function (page 84) that takes a builder of class `MazeBuilder` as an argument.

The `MazeBuilder` class defines the following interface for building mazes:

```
class MazeBuilder {
public:
    virtual void BuildMaze() { }
    virtual void BuildRoom(int room) { }
    virtual void BuildDoor(int roomFrom, int roomTo) { }

    virtual Maze* GetMaze() { return 0; }
protected:
    MazeBuilder();
};
```

This interface can create three things: (1) the maze, (2) rooms with a particular room number, and (3) doors between numbered rooms. The `GetMaze` operation returns the maze to the client. Subclasses of `MazeBuilder` will override this operation to return the maze that they build.

All the maze-building operations of `MazeBuilder` do nothing by default. They're not declared pure virtual to let derived classes override only those methods in which they're interested.

Given the `MazeBuilder` interface, we can change the `CreateMaze` member function to take this builder as a parameter.

```

Maze* MazeGame::CreateMaze (MazeBuilder& builder) {
    builder.BuildMaze();

    builder.BuildRoom(1);
    builder.BuildRoom(2);
    builder.BuildDoor(1, 2);

    return builder.GetMaze();
}

```

Compare this version of `CreateMaze` with the original. Notice how the builder hides the internal representation of the Maze—that is, the classes that define rooms, doors, and walls—and how these parts are assembled to complete the final maze. Someone might guess that there are classes for representing rooms and doors, but there is no hint of one for walls. This makes it easier to change the way a maze is represented, since none of the clients of `MazeBuilder` has to be changed.

Like the other creational patterns, the Builder pattern encapsulates how objects get created, in this case through the interface defined by `MazeBuilder`. That means we can reuse `MazeBuilder` to build different kinds of mazes. The `CreateComplexMaze` operation gives an example:

```

Maze* MazeGame::CreateComplexMaze (MazeBuilder& builder) {
    builder.BuildRoom(1);
    // ...
    builder.BuildRoom(1001);

    return builder.GetMaze();
}

```

Note that `MazeBuilder` does not create mazes itself; its main purpose is just to define an interface for creating mazes. It defines empty implementations primarily for convenience. Subclasses of `MazeBuilder` do the actual work.

The subclass `StandardMazeBuilder` is an implementation that builds simple mazes. It keeps track of the maze it's building in the variable `_currentMaze`.

```

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

    virtual void BuildMaze();
    virtual void BuildRoom(int);
    virtual void BuildDoor(int, int);

    virtual Maze* GetMaze();
private:
    Direction CommonWall(Room*, Room*);
    Maze* _currentMaze;
};

```

`CommonWall` is a utility operation that determines the direction of the common wall between two rooms.

The `StandardMazeBuilder` constructor simply initializes `_currentMaze`.

```
StandardMazeBuilder::StandardMazeBuilder () {
    _currentMaze = 0;
}
```

`BuildMaze` instantiates a `Maze` that other operations will assemble and eventually return to the client (with `GetMaze`).

```
void StandardMazeBuilder::BuildMaze () {
    _currentMaze = new Maze;
}

Maze* StandardMazeBuilder::GetMaze () {
    return _currentMaze;
}
```

The `BuildRoom` operation creates a room and builds the walls around it:

```
void StandardMazeBuilder::BuildRoom (int n) {
    if (!_currentMaze->RoomNo(n)) {
        Room* room = new Room(n);
        _currentMaze->AddRoom(room);

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

To build a door between two rooms, `StandardMazeBuilder` looks up both rooms in the maze and finds their adjoining wall:

```
void StandardMazeBuilder::BuildDoor (int n1, int n2) {
    Room* r1 = _currentMaze->RoomNo(n1);
    Room* r2 = _currentMaze->RoomNo(n2);
    Door* d = new Door(r1, r2);

    r1->SetSide(CommonWall(r1,r2), d);
    r2->SetSide(CommonWall(r2,r1), d);
}
```

Clients can now use `CreateMaze` in conjunction with `StandardMazeBuilder` to create a maze:

```

Maze* maze;
MazeGame game;
StandardMazeBuilder builder;

game.CreateMaze(builder);
maze = builder.GetMaze();

```

We could have put all the `StandardMazeBuilder` operations in `Maze` and let each `Maze` build itself. But making `Maze` smaller makes it easier to understand and modify, and `StandardMazeBuilder` is easy to separate from `Maze`. Most importantly, separating the two lets you have a variety of `MazeBuilders`, each using different classes for rooms, walls, and doors.

A more exotic `MazeBuilder` is `CountingMazeBuilder`. This builder doesn't create a maze at all; it just counts the different kinds of components that would have been created.

```

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

    virtual void BuildMaze();
    virtual void BuildRoom(int);
    virtual void BuildDoor(int, int);
    virtual void AddWall(int, Direction);

    void GetCounts(int&, int&) const;
private:
    int _doors;
    int _rooms;
};

```

The constructor initializes the counters, and the overridden `MazeBuilder` operations increment them accordingly.

```

CountingMazeBuilder::CountingMazeBuilder () {
    _rooms = _doors = 0;
}

void CountingMazeBuilder::BuildRoom (int) {
    _rooms++;
}

void CountingMazeBuilder::BuildDoor (int, int) {
    _doors++;
}

void CountingMazeBuilder::GetCounts (
    int& rooms, int& doors
) const {
    rooms = _rooms;
    doors = _doors;
}

```



Here's how a client might use a `CountingMazeBuilder`:

```
int rooms, doors;
MazeGame game;
CountingMazeBuilder builder;

game.CreateMaze(builder);
builder.GetCounts(rooms, doors);

cout << "The maze has "
      << rooms << " rooms and "
      << doors << " doors" << endl;
```

## Known Uses

The RTF converter application is from ET++ [WGM88]. Its text building block uses a builder to process text stored in the RTF format.

Builder is a common pattern in Smalltalk-80 [Par90]:

- The Parser class in the compiler subsystem is a Director that takes a `ProgramNodeBuilder` object as an argument. A Parser object notifies its `ProgramNodeBuilder` object each time it recognizes a syntactic construct. When the parser is done, it asks the builder for the parse tree it built and returns it to the client.
- `ClassBuilder` is a builder that Classes use to create subclasses for themselves. In this case a Class is both the Director and the Product.
- `ByteCodeStream` is a builder that creates a compiled method as a byte array. `ByteCodeStream` is a nonstandard use of the Builder pattern, because the complex object it builds is encoded as a byte array, not as a normal Smalltalk object. But the interface to `ByteCodeStream` is typical of a builder, and it would be easy to replace `ByteCodeStream` with a different class that represented programs as a composite object.

The Service Configurator framework from the Adaptive Communications Environment uses a builder to construct network service components that are linked into a server at run-time [SS94]. The components are described with a configuration language that's parsed by an LALR(1) parser. The semantic actions of the parser perform operations on the builder that add information to the service component. In this case, the parser is the Director.

## Related Patterns

Abstract Factory (87) 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.

A Composite (163) is what the builder often builds.