Builder Pattern – Creational

- Intent

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

- Motivation

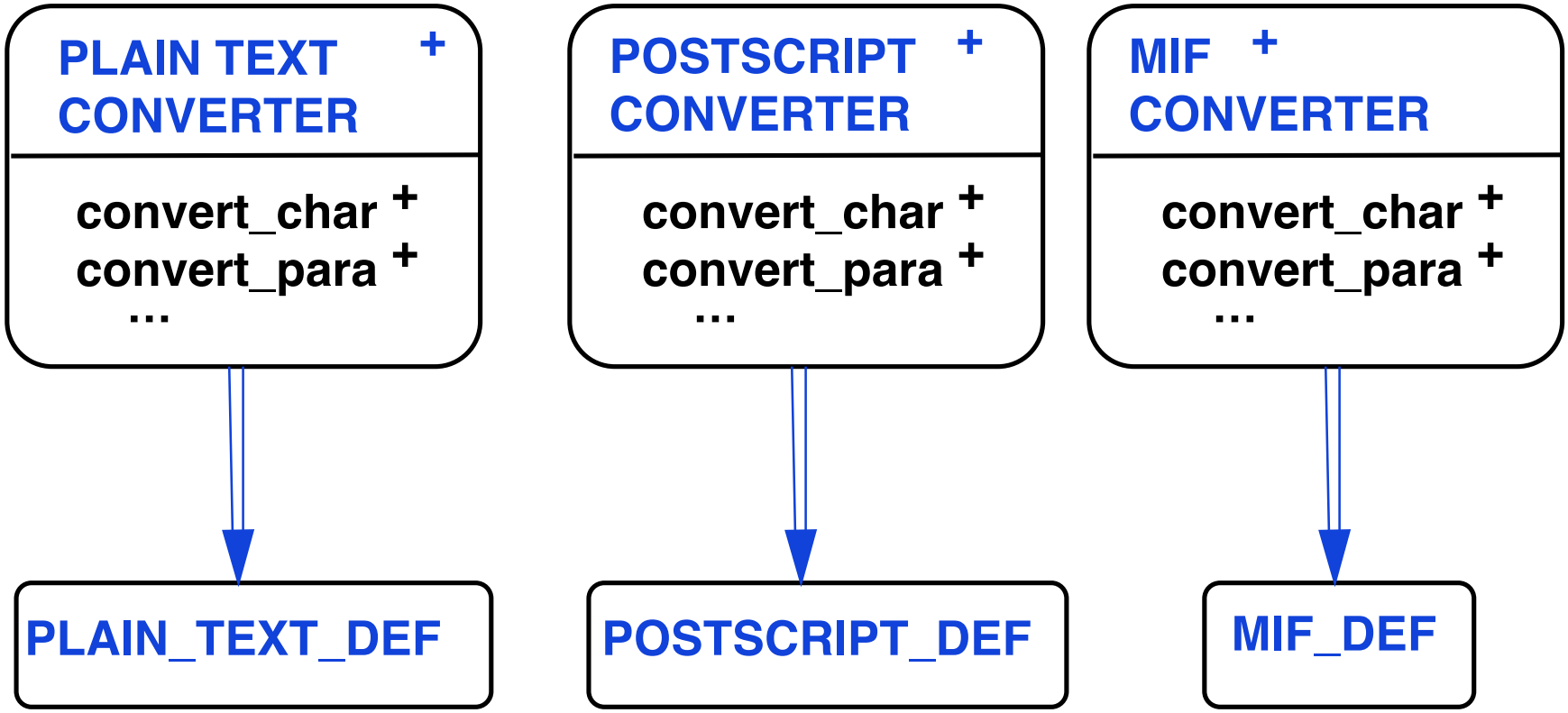
  - **Reader for RTF (Rich Text Format) should be able to convert to any other representation**

    Plain Text, MIF (Maker Interchange File), Postscript

  - **Open ended number of representations possible**

  - **Abstract the conversion process**

Example Conversion

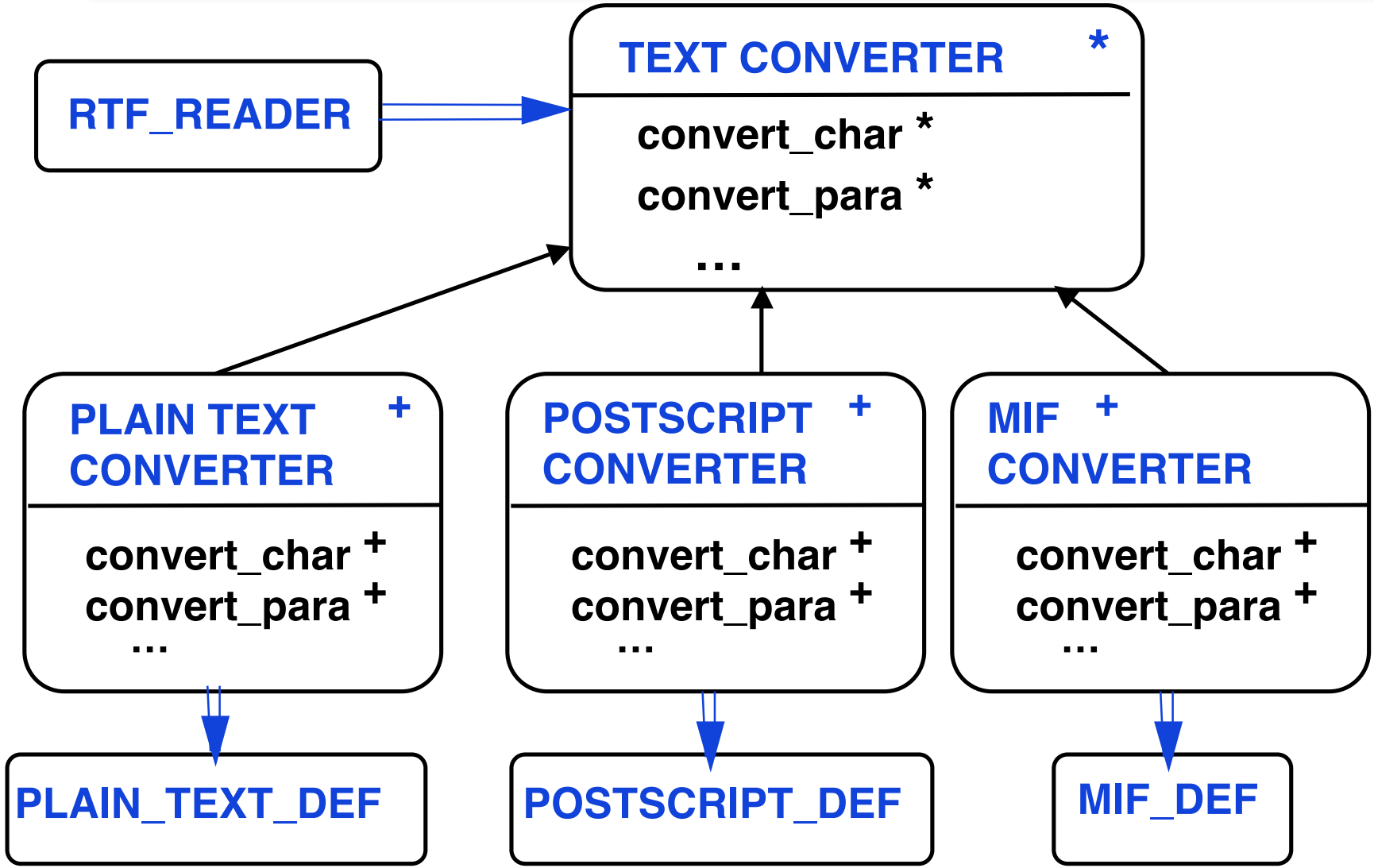


Builder – Applicability

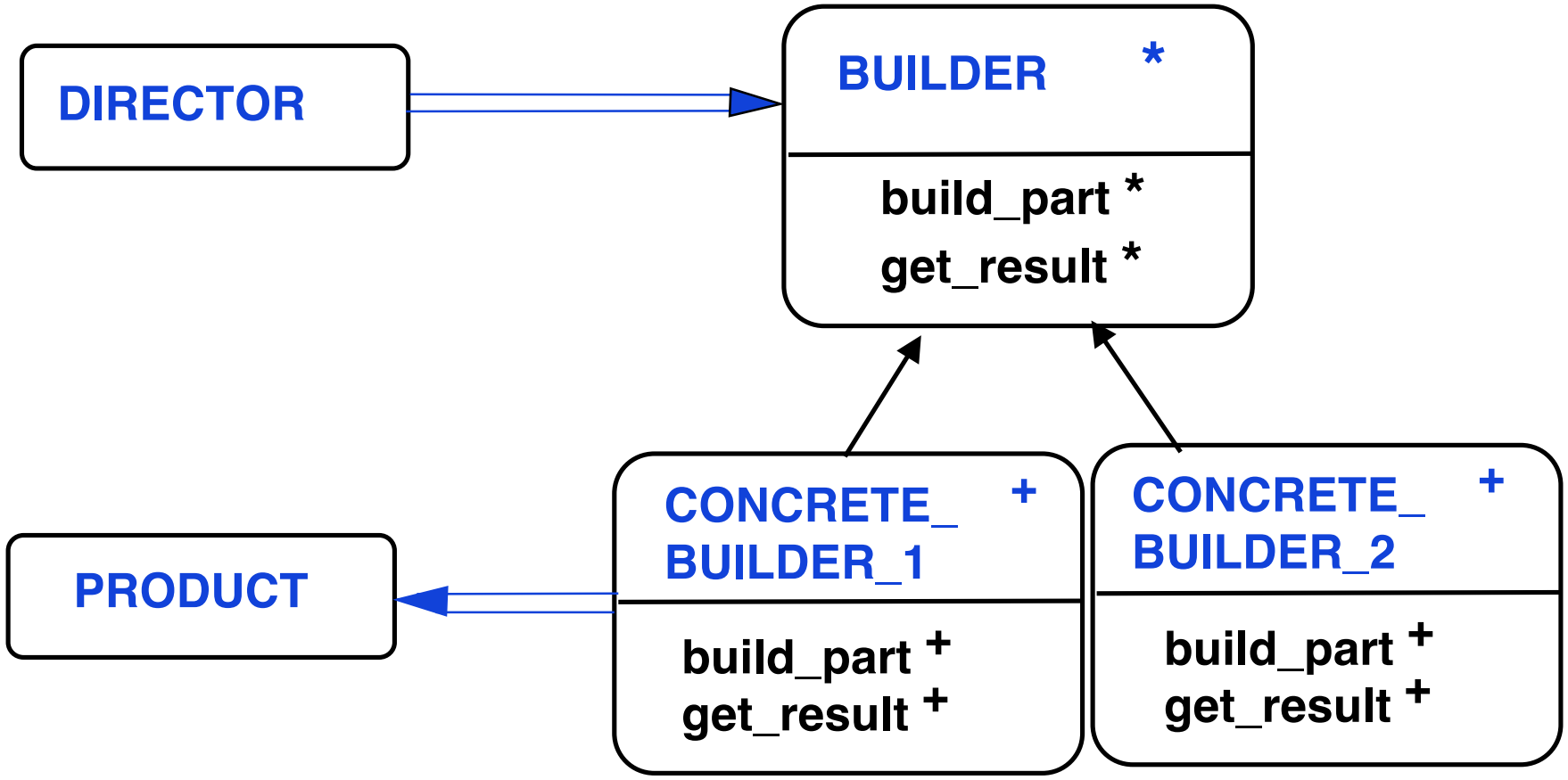
- When algorithm for creating a complex object should be independent of the parts that make up the object and how they are assembled

- The construction process must allow different representations for the object that is constructed

Builder – Example Architecture



Builder – Abstract Architecture



Builder – Participants

- Builder

  Specifies abstract interface for creating parts of a product object

- Concrete builder

  Constructs and assembles parts of the product by implementing the Builder interface

- Director

  Constructs an object using the Builder interface

- Product

  - **The complex object under construction**
  - **Includes classes that define the parts and interfaces for assembling parts into a final result**

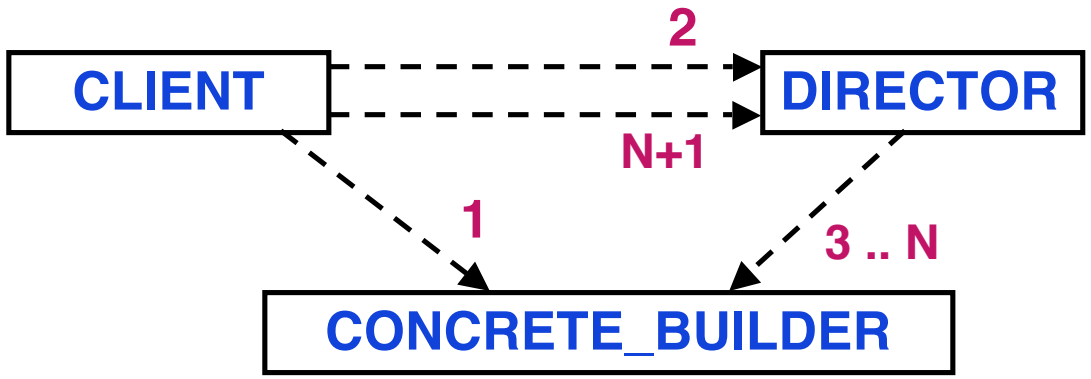
Builder – Collaboration

- The client creates a Director object and configures it with the desired Builder object
- Director notifies Builder whenever a part of the product should be built
- Builder handles requests from the Director and adds parts to the product
- Client retrieves the product from the Builder

Builder – Scenario

Scenario: Build a product

1 create aBuilder.make
2 director.set(aBuilder)
3 aBuilder.build_part_1
4 aBuilder.build_part_2
…
N aBuilder.build_part_N
N+1 director.get_product



Builder – Implementation

```
class MAZE_BUILDER
feature
    build_maze deferred end
    build_room ( id : INTEGER ) deferred end
    build_door ( id1 : INTEGER ;
                 id2 : INTEGER  ) deferred end
    maze : MAZE   // builder adds to the structure
    get_result : MAZE do Result := maze end
end
```

Builder – Implementation – 2

```
class MAZE_GAME      create create_maze
feature
   create_maze ( builder : MAZE_BUILDER )
   do
     the_builder := builder
     builder.build_room ( 1 )
     builder.build_room ( 2 )
     builder.build_door ( 1 , 2 )
   end

   the_builder : MAZE_BUILDER

   get_result : MAZE do Result := the_builder.get_result end

end
```

Builder – Implementation – 3

```
class DEFAULT_MAZE_BUILDER
inherit BUILDER
feature
    build_room ( id : INTEGER )
    local room : ROOM
    do
        create room.make ( id )      -- make with four walls default
        maze.add_room (room )
    end

    ...
```

Builder – Implementation – 4

```
build_door ( id1 : INTEGER ; id2 : INTEGER )
local r1 , r2 : ROOM  ;  door : DOOR
do
   r1 := maze.get_room ( id1 )
   r2 := maze.get_room ( id2 )
   create  door.make ( id1, id2 )
   r1.set_side ( common (r1, r2 ) , door )
   r2.set_side ( common (r1, r2 ) , door )
end

end -- DEFAULT_MAZE_BUILDER
```

Builder – Implementation – 5

```
// Client

maze : MAZE
game : MAZE_GAME
builder : DEFAULT_MAZE_BUILDER

create builder
create game.create_maze ( builder )
maze := game.get_result
```

Builder – Related Patterns

- Abstract Factory focuses on families of product objects, while Builder focuses on step by step construction of complex objects

- Builder frequently builds a Composite