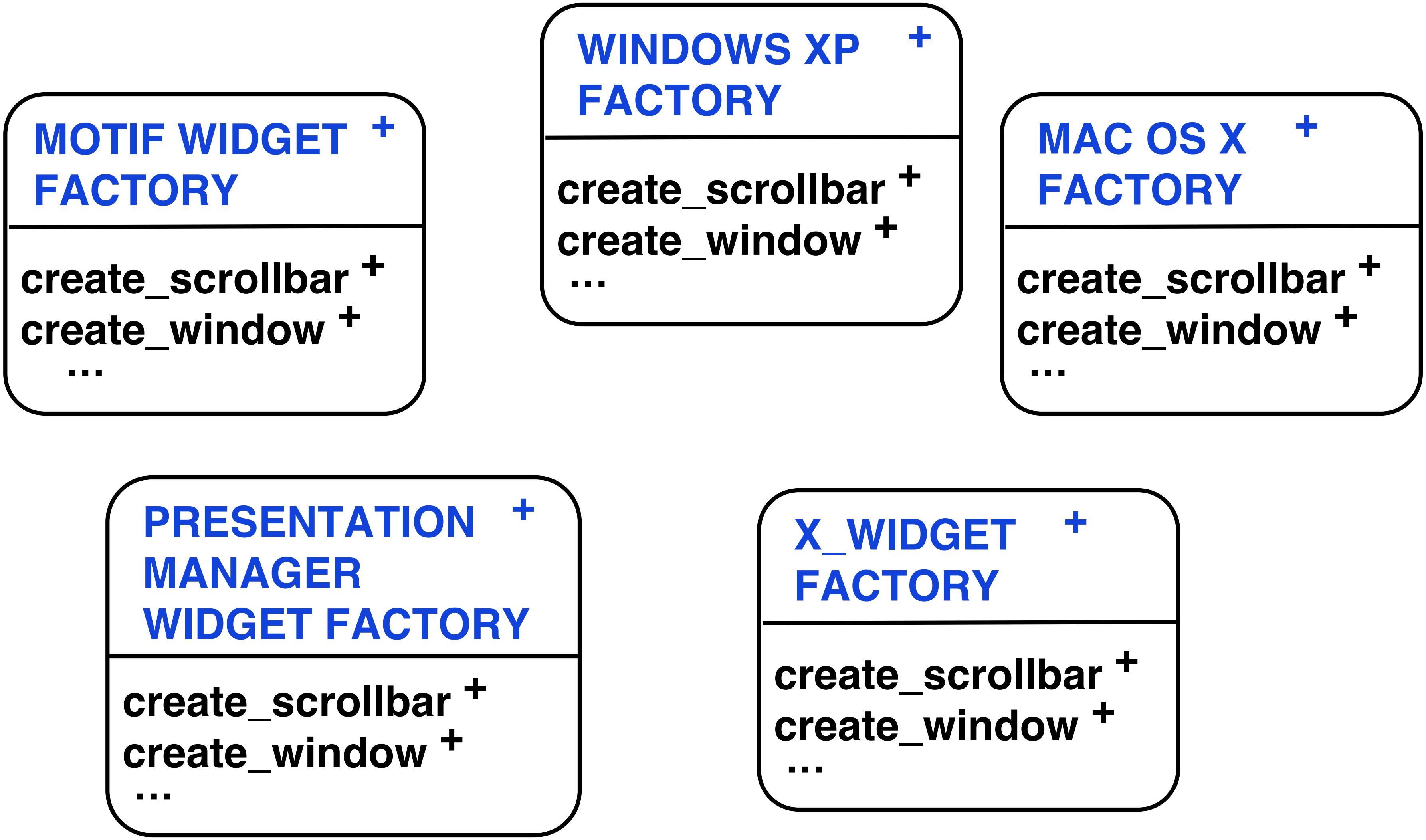
Abstract Factory Pattern – Creational

- Intent

  Provide an interface for creating families of related or dependent objects without specifying their concrete classes

- The pattern is not abstract – just a poor choice of name
  - » **A better names would like one of the following**
    - > **Family factory**
    - > **Style factory**
    - > **Group factory**

Example Families of Products



MOTIF WIDGET FACTORY +

create_scrollbar +
create_window +
…

WINDOWS XP FACTORY +

create_scrollbar +
create_window +
…

MAC OS X FACTORY +

create_scrollbar +
create_window +
…

PRESENTATION MANAGER WIDGET FACTORY +

create_scrollbar +
create_window +
…

X_WIDGET FACTORY +

create_scrollbar +
create_window +
…

Motivation

- Building a user interface toolkit that supports multiple look and feel standards

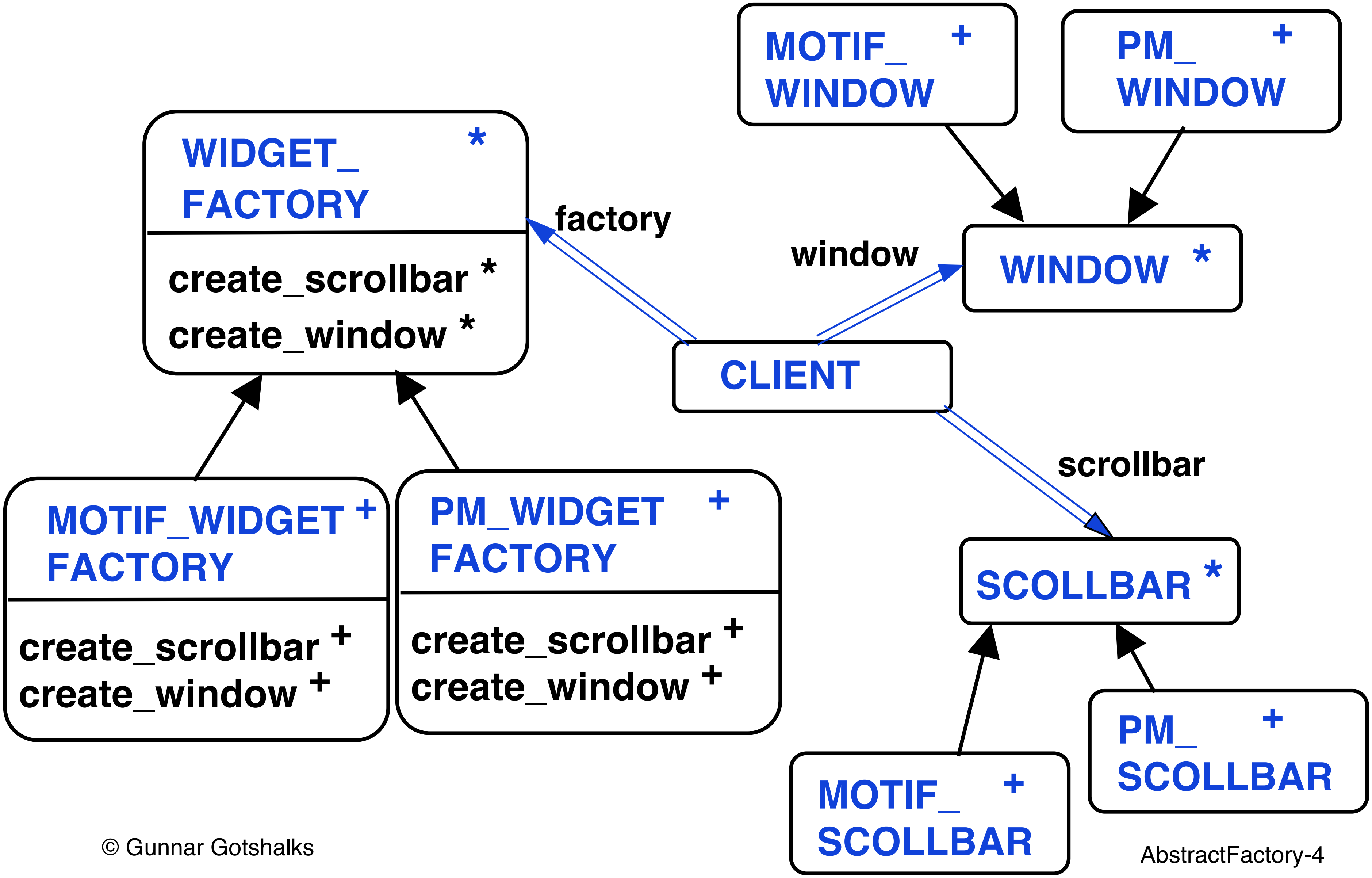
  WINDOWS XP, MAC OS X, Motif,
  Presentation Manager, X Window

- Have different appearances and behaviour for a large set of subclasses

  scroll bars, windows, buttons, ...

- Java API uses AF pattern in java.awt.Toolkit
  - » **Button and Canvas classes are platform independent**
    - > **Use classes ButtonPeer and CanvasPeer that are platform specific**

Example Architecture



WIDGET_ FACTORY *

create_scrollbar *

create_window *

factory

window

CLIENT

scrollbar

MOTIF_ WINDOW +

PM_ WINDOW +

WINDOW *

MOTIF_WIDGET FACTORY +

create_scrollbar +

create_window +

PM_WIDGET FACTORY +

create_scrollbar +

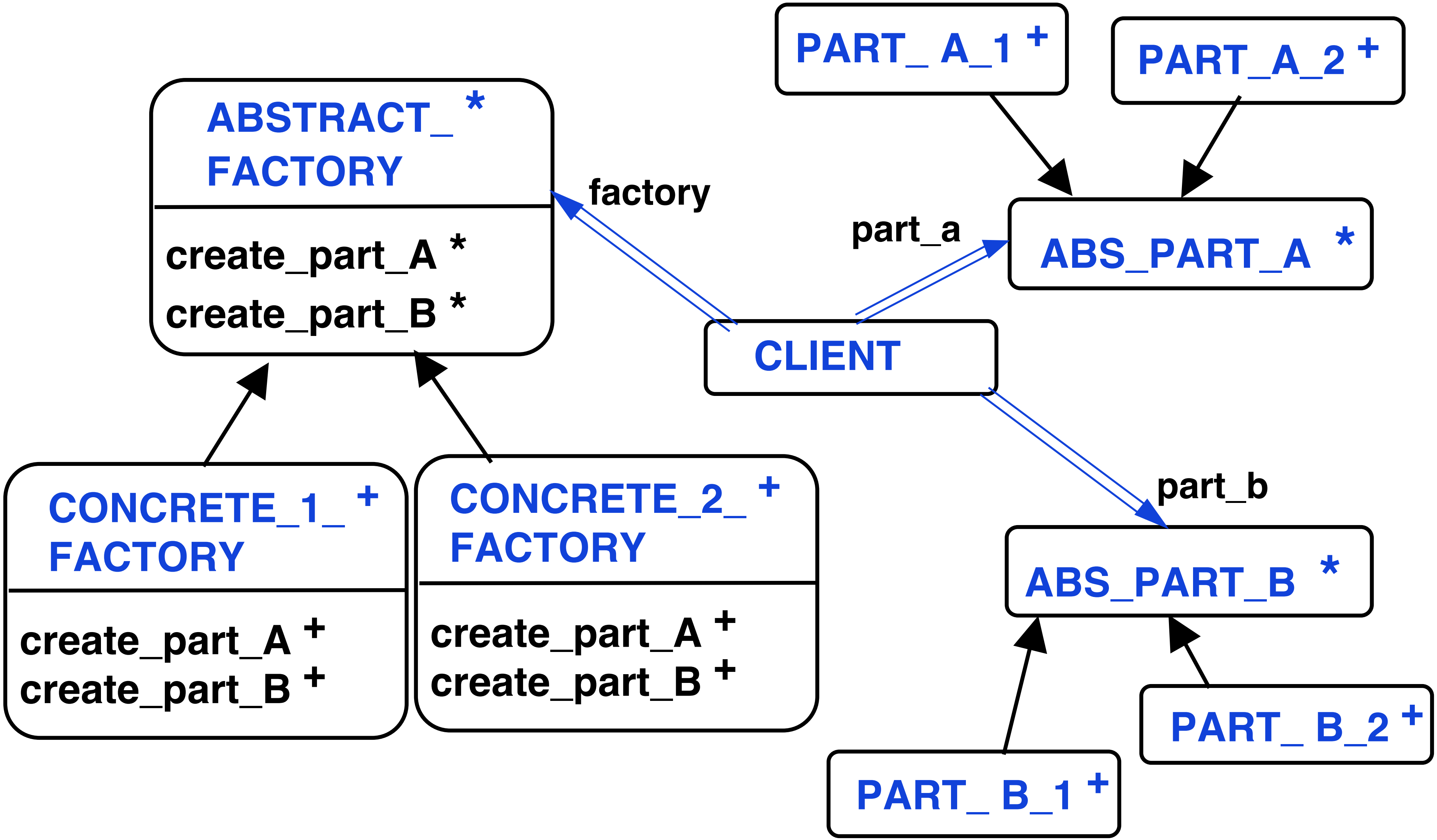
create_window +

SCOLLBAR *

MOTIF_ SCOLLBAR +

PM_ SCOLLBAR +

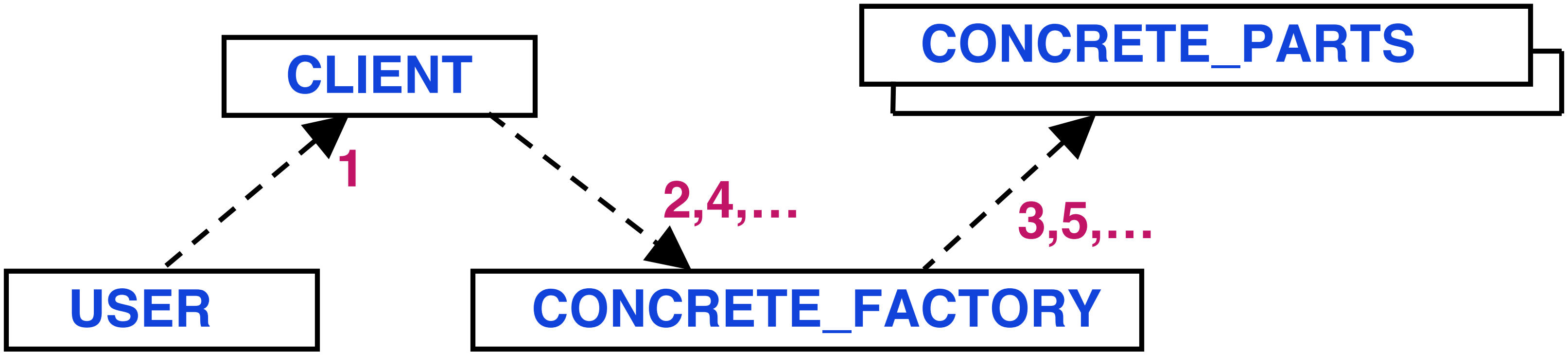
Abstract Architecture



Scenario

Scenario: Build a product

1 create client.make(a_Factory)
2 a_Factory.make_part_1(…)
3 part_1.make (…)
4 a_Factory. make _part_2 (…)
5 part_2.make (…)
…



Participants

- Abstract factory

  Declares interface for operations that create abstract parts

- Concrete factory

  Implements operations to create parts

- Abstract part

  Declares an interface for a type of part

Participants – 2

- Concrete part
  - » **Defines part to be created by the corresponding concrete factory**
  - » **Implements Abstract_Part interface**

- Client

  Uses only the interfaces declared by Abstract_Factory and Abstract_Part

Applicability

- System should be independent of how its products are created, composed and represented
- System should be configured with one of multiple families of products
- Family of related product objects is designed to be used together and you need to enforce this constraint
- Provide a class library of products and you want to reveal just their interfaces not their implementations

Collaborations

- A single instance of Concrete_Factory is created at run time
  - » **Creates parts having a particular implementation**
  - » **To create different parts, use a different concrete factory**

- Abstract_Factory defers creation of parts to its Concrete_Factory subclass

Consequences

- Isolates concrete classes
  - » **Factory encapsulates responsibility and process of creating parts**
  - » **Isolates clients from implementation classes**
- Exchanging product families easy

  Concrete factory appears once where it is instantiated
- Promotes consistency among products
- Supporting new kinds of products is difficult

  Fixes set of parts to be created

Implementation

```
class MAZE_FACTORY feature
  make_maze : MAZE
    do  create Result  end

  make_room ( id : INTEGER ) : ROOM
    do create Result.make ( id ) end

  make_door ( r1 : ROOM ; r2 : ROOM ) : DOOR
   do create Result.make ( r1, r2 ) end

  make_wall : MAZE
    do create Result.make end

end
```

Implementation – 2

```
-- Client program

class MAZE_MAKER     create make_with
feature
   make_with ( factory : MAZE_FACTORY )
      local  maze : MAZE  ;  r1, r2 : ROOM  ;  door : DOOR
      do
         maze := factory.make_maze
         r1  := factory.make_room (1)
         r2  := factory.make_room(2)
         door := factory.make_door ( r1, r2 )
         maze.add_room (r1 )  ;  maze.add_room ( r2 )
-- Construct contents of maze – next slide

      end
end
```

Implementation – 3

```
-- Construct contents of maze

r1.set_side ( North , factory.make_wall )
r1.set_side ( East , door )
r1.set_side ( South , factory.make_ wall)
r1.set_side ( West , factory.make_ wall)

r2.set_side ( North , factory.make_ wall)
r2.set_side ( East , factory.make_ wall )
r2.set_side ( South , factory.make_ wall)
r2.set_side ( West , door)
```

Implementation – 4

```
class ENCHANTED_MAZE_FACTORY inherits MAZE_FACTORY
feature

   make_room ( id : INTEGER ) : ROOM
      local room : ENCHANTED_ROOM
       do
            cast_a_spell(id)
            create  room.make ( id, spell )  ;  Result := room
       end

   make_door ( r1 : ROOM ; r2 : ROOM ) : DOOR
       local door : DOOR_NEEDING_SPELL
       do
            create door.make ( r1, r2 )  ;  Result := door
       end
end
```

Implementation – 5

```
-- Imagine a subclass of wall is damaged if a bomb goes off
-- Have a subclass of room with a bomb in it

class BOMBED_MAZE_FACTORY inherits MAZE_FACTORY
feature
 make_wall : WALL
       local  wall : BOMBED_WALL
      do     create wall.make  ;  Result := wall   end

 make_room ( id : INTEGER ) : ROOM is
       local  room : ROOM_WITH_BOMB
       do   create room.make ( id )  ;  Result := room   end
end
```

Implementation – 6

```
-- Create various mazes

the_maze : MAZE_MAKER

factory_1 : ENCHANTED_MAZE_FACTORY   -- Maze 1
create factory_1
create the_maze . make_from ( factory_1 )
the_maze.maze.describe

factory_2 : BOMBED_MAZE_FACTORY      -- Maze 2
create factory_2
create the_maze . create_maze ( factory_2 )
the_maze.maze.describe
```

Related Patterns

- Abstract Factory classes can be implemented with Factory Method or Prototype

- Concrete factories are often Singletons

Abstract Factory in Java API

- java.awt.Toolkit uses the Abstract Factory pattern
  - » **Classes such as Button and Canvas are platform independent**
  - » **Peer classes ButtonPeer and CanvasPeer contain platform specific program text**