

Abstract Factory Pattern – Creational

- Intent

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

- Motivation

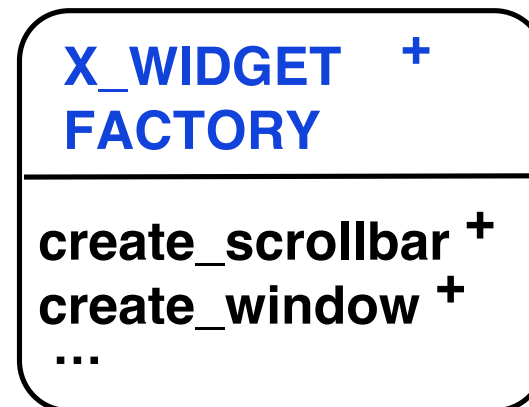
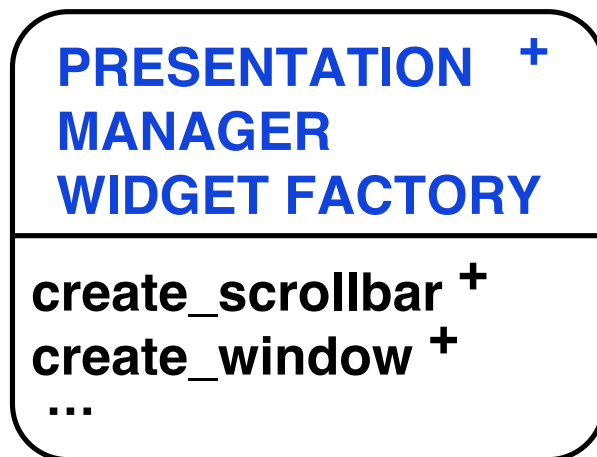
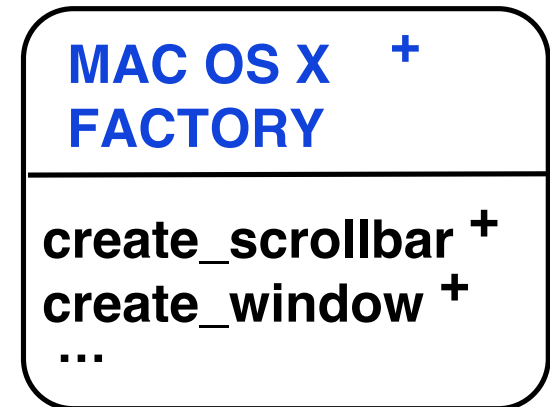
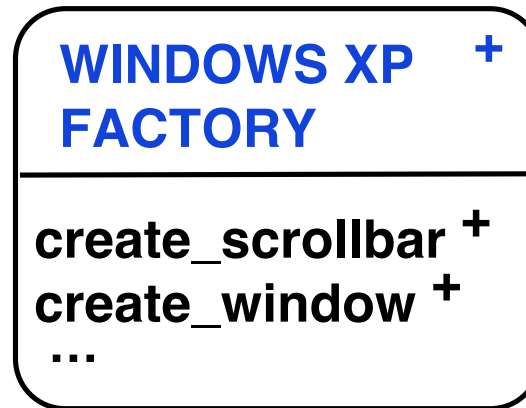
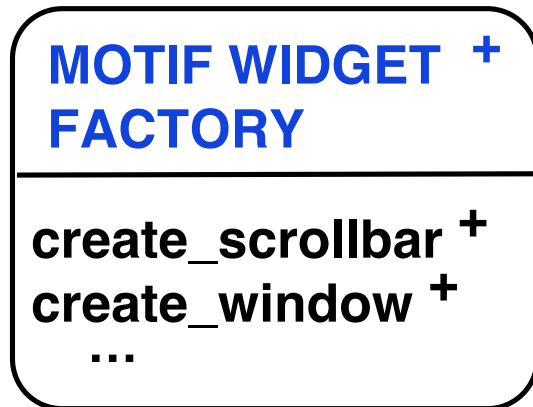
- » **Building a user interface toolkit that supports multiple look and feel standards**

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, ...

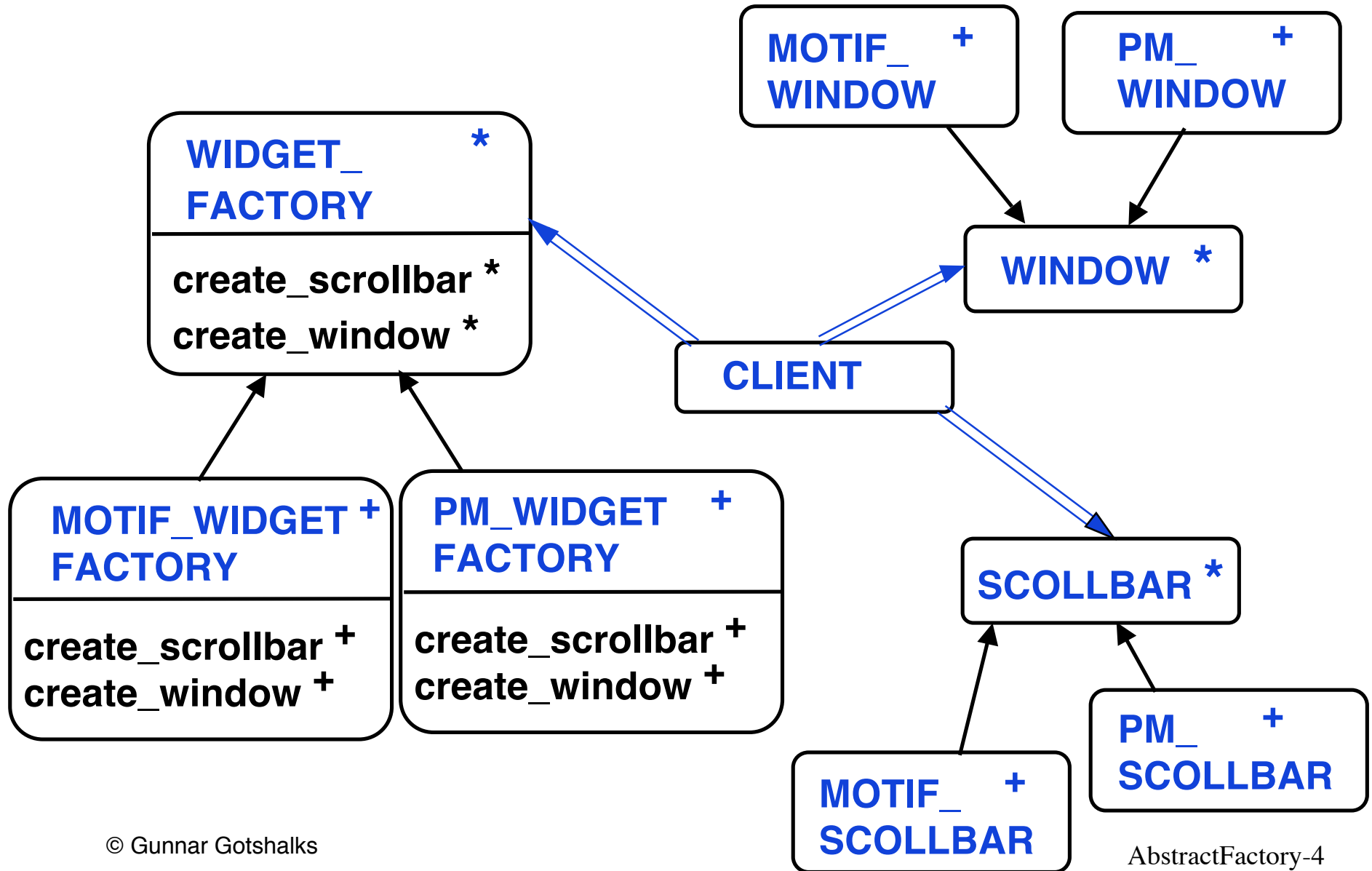
Example of a set of Families of Products



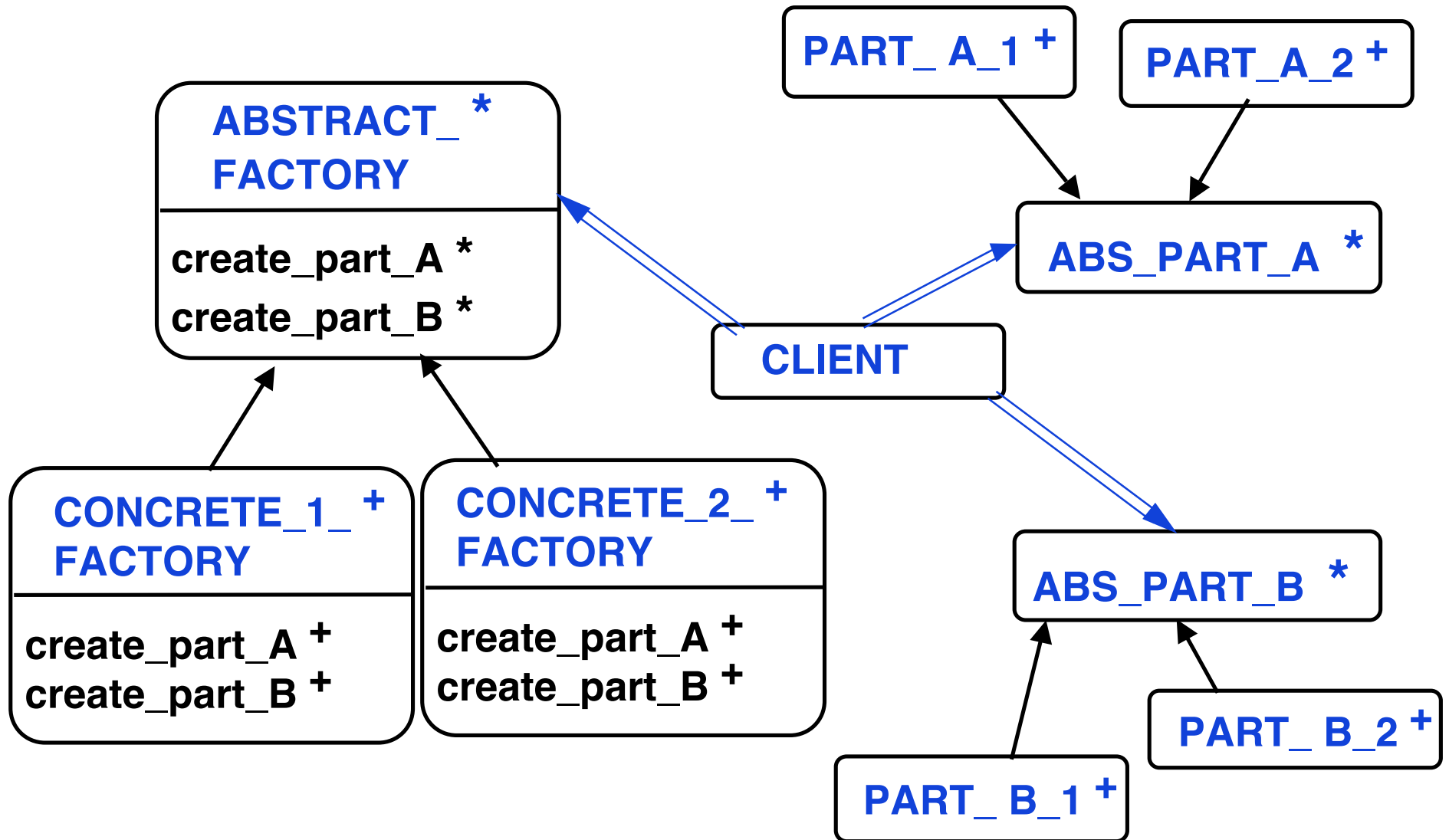
AF – Applicability

- 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

AF – Example Architecture



AF – Abstract Architecture



AF – Participants

- Abstract factory

Declares interface for operations that create abstract parts

- Concrete factory

Implements operations to create parts

AF – Participants – 2

- Abstract part
 - Declares an interface for a type of part**
- 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**

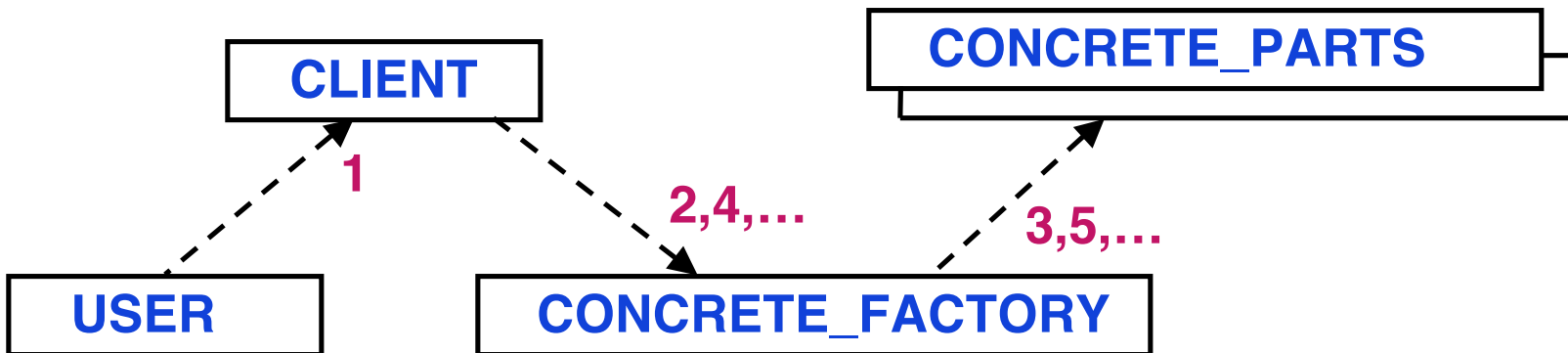
AF – Collaborations

- 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

AF – Scenario

Scenario: Build a product

- 1 create client.make(aFactory)
- 2 aFactory.make_part_1(...)
- 3 part_1.make (...)
- 4 aFactory. make_part_2 (...)
- 5 part_2.make (...)
- ...



AF – 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**

AF – 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
```

AF – Implementation – 2

-- Client program

```
class MAZE_GAME    create create_maze
feature
  create_maze ( 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 )
    end
end
```

-- Construct contents of maze – next slide

AF – 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 )
```

AF – 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
```

AF – 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

AF – Implementation – 6

-- Create various games

game : MAZE_GAME

factory_1 : ENCHANTED_MAZE_FACTORY // Game 1

create factory_1

create game . create_maze (factory_1)

factory_2 : BOMBED_MAZE_FACTORY // Game 2

create factory_2

create game . create_maze (factory_2)

Abstract Factory – Related Patterns

- Abstract Factory classes can be implemented with Factory Method or Prototype
- Concrete factories are often Singletons