Prototype Pattern – Creational

- Intent

  Specify the kinds of objects to create using a prototypical instance and create new objects by copying the prototype

  Use in a mix and match situation
  * **All chairs of the same style**
  * **All desks of the same style**
  * **Desks and chairs can be different styles**

Motivation

- Build an editor for musical scores by customizing a general framework for graphical editors
- Add new objects for notes, rests, staves
- Have a palette of tools

  Click on eight'th note tool and add it to the document

- Assume Framework provides
  - » **Abstract_Graphic class**
  - » **Abstract_Tool class for defining tools**
  - » **Graphic_Tool subclass – create instances of graphical objects and add them to the document**

Motivation – 2

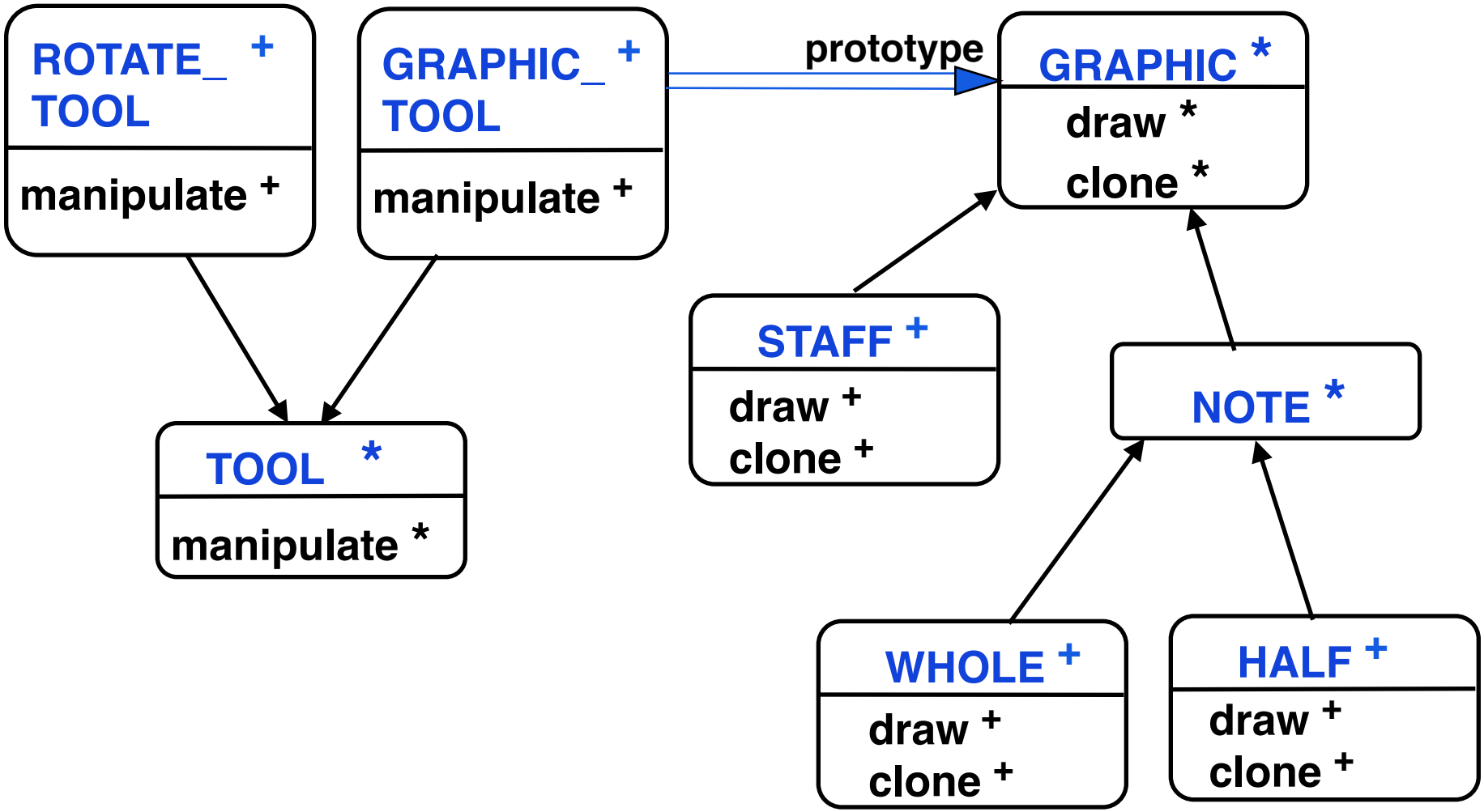
- Graphic_Tool doesn't know how to create instances of music classes

  Could subclass Graphic_Tool for each kind of music object

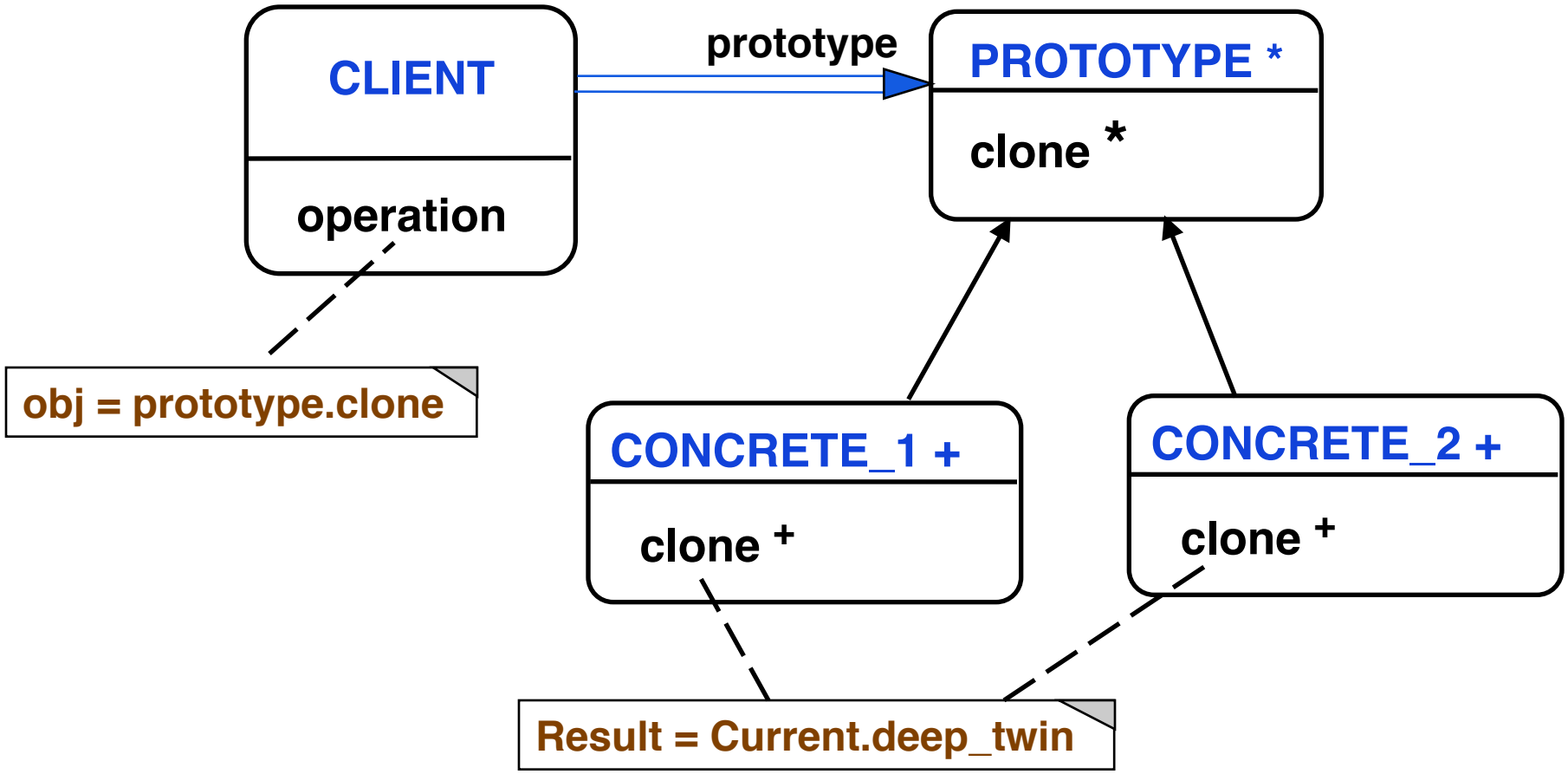
  But have lots of classes with insignificant variations

- Object composition is a flexible alternative to subclassing
  - » **How can we use it in this application?**
  - » **Solution is to copy or clone an instance called a prototype**
- Graphic_Tool is parameterized by the prototype to clone

Example Architecture



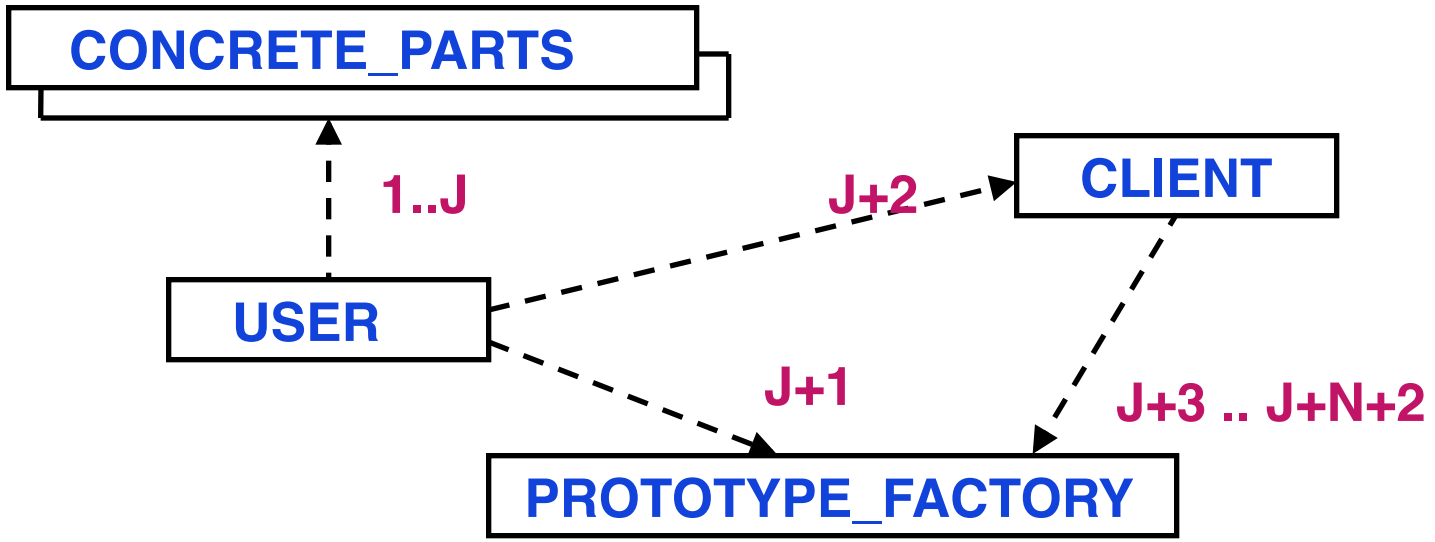
Abstract Architecture



Scenario

Scenario: Build a product

1..J create parts_i.make
J+1 create proto_factory.make(…parts…)
J+2 create client.make(proto_factory)
J+3 proto_factory.make_part_1 (…)
…
J+N+2 proto_factory.. make _part_2 (…)



Participants

- Prototype

  Declares an interface for cloning itself

- Concrete prototype

  Implements operation for cloning itself

- Client

  Creates new object by asking prototype to clone itself

Applicability

- Use when a system should be independent of how its products are

  created, composed and represented

  and

  > **When classes to instantiate are specified at run time**

    dynamic loading

  > **or – To avoid building a class hierarchy of factories that parallels the class hierarchy of products**

  > **or – When instances of a class can have one of a few different combinations of state**

    – **More convenient to install corresponding number of prototypes and clone them – undo command case study**

Consequences

- Many of the same consequences as Builder and Abstract Factory
- Benefits
  - » **Hides concrete product classes from the client**
    - > **Reduces number of names client needs to know**
    - > **Work with application specific classes without modification**
- Liability

  Each subclass of Prototype implements clone which can be difficult with circular references

Additional Benefits

- Adding & removing products at run time
  - » **Register a prototype instance with client**

- Specify new objects by varying values
  - » **Define new behaviour through object composition**
  - » **Specify objects variables with new values not new classes**
  - » **Effectively define new kinds of objects**
  - » **Client exhibits new behaviour by delegating responsibility to the prototype**

Additional Benefits– 2

- Specify new objects by varying structure
  - » **Build objects as parts and subparts**
  - » **User defines new groupings that can be reused**

- Configuring an application with classes dynamically

  C++ lets you load classes dynamically

Additional Benefits– 3

- Reduced sub-classing
  - » **Factory Method produces hierarchy of creator classes that parallels product classes**
  - » **Cloning avoids parallel hierarchy**
    - > **Biggest benefit in languages like C++ that do not treat classes as first class citizens (not real objects themselves)**
    - > **Less benefit in Smalltalk and Objective C as classes are their own prototype**

Maze Prototype Factory

```
class MAZE_PROTOTYPE_FACTORY    create make
feature
    prototype_maze : MAZE
    prototype_room : ROOM
    prototype_door : DOOR
    prototype_wall : WALL

// Note parameterization with prototypes
    make ( m : MAZE ; r : ROOM ; d : DOOR ; w : WALL )
    do
       prototype_maze := m ; prototype_door := d
       prototype_room:= r ; prototype_wall := w
    end

     -- next slide for the make components methods
end
```

Maze Prototype Factory – 2

```
new_wall : WALL
do     Result := prototype_wall.deep_twin    end

new_door ( id : STRING ; r1 , r2 : ROOM ) : DOOR
do    Result := prototype_door. deep_twin
      Result.set_parameters (id, r1, r2 )
end

new_room ( id : INTEGER ) : ROOM
do    Result := prototype_room. deep_twin
      Result.set_parameters ( id )
end

new_maze : MAZE
do    Result := prototype_maze. deep_twin end
```

Enchanted Maze Factory

```
class ENCHANTED_MAZE_FACTORY
   inherit MAZE_PROTOTYPE_FACTORY
           redefine new_room , new_door end

new_room ( id : INTEGER ) : ENCHANTED_ROOM
do
      if attached {ENCHANTED_ROOM} prototype_room as room
      then Result := room. deep_twin end
      Result.set_parameters ( id , last_spell_cast )
end

new_door ( id : STRING ; r1 , r2 : ROOM ) : LOCKED_DOOR
do
      if attached {LOCKED_DOOR} prototype_door as door
      then Result := door. deep_twin end
     Result.set_parameters (id , r1 , r2 )
end
```

Bombed Maze Factory

```
class BOMBED_MAZE_FACTORY
   inherit MAZE_PROTOTYPE_FACTORY
           redefine new_room , new_wall end

new_room ( id : INTEGER ) : BOMBED_ROOM
do
      if attached {BOMBED_ROOM} prototype_room as room
      then Result := room. deep_twin end
      Result.set_parameters ( id, False )
end

new_wall ( id : STRING ) : BOMBED_WALL
do
      if attached {BOMBED_WALL} prototype_wall as wall
      then Result := wall. deep_twin end
      Result.set_parameters (id )
end
```

Common Prototype

```
class COMMON_PROTOTYPE

feature

make deferred end

maze : MAZE

create_maze ( factory : MAZE_PROTOTYPE_FACTORY ) : MAZE
local
   r1, r2 : ROOM
   door : DOOR
do
      … Common maze creation …
end
```

Basic Prototype Maze

```
class BASIC_PROTOTYPE_MAZE
feature
make
  local
    maze_prototype_factory : MAZE_PROTOTYPE_FACTORY
    proto_maze : MAZE     proto_wall : WALL
    proto_room : ROOM     proto_door : DOOR
  do
    create proto_maze.make   create proto_wall.make
    create proto_room.make   create proto_door.make
    create maze_prototype_factory.make
            ( proto_maze, proto_wall, proto_room, proto_door )
    maze := create_maze (maze_prototype_factory)
  end
end
```

Enchanted Prototype Maze

```
class ENCHANTED_PROTOTYPE_MAZE
feature
make
  local
    maze_prototype_factory : ENCHANTED_MAZE_FACTORY
    proto_maze : MAZE     proto_wall : WALL
    proto_room : ROOM     proto_door : DOOR
  do
    create proto_maze.make
    create proto_wall.make
    create { ENCHANTED_ROOM } proto_room.make
    create { LOCKED_DOOR } proto_door.make
    create maze_prototype_factory.make
            ( proto_maze, proto_wall, proto_room, proto_door )
    maze := create_maze (maze_prototype_factory)
  end
end
```

Bombed Prototype Maze

```
class ENCHANTED_PROTOTYPE_MAZE
feature
make
  local
    maze_prototype_factory : BOMBED_MAZE_FACTORY
    proto_maze : MAZE     proto_wall : WALL
    proto_room : ROOM     proto_door : DOOR
  do
    create proto_maze.make      create proto_door.make
    create {BOMBED_WALL } proto_wall.make
    create {BOMBED_ROOM } proto_room.make
    create maze_prototype_factory.make
            ( proto_maze, proto_wall, proto_room, proto_door )
    maze := create_maze (maze_prototype_factory)
  end
end
```

Prototype Client

```
class PROTOTYPE_CLIENT
feature
make
  local
    maze_1 : BASIC_PROTOTYPE_MAZE
    maze_2 : ENCHANTED_PROTOTYPE_MAZE
    maze_3 : BOMBED_PROTOTYPE_MAZE
  do
    create maze_1.make
    maze_1.describe
    create maze_2.make
    maze_2.describe
    create maze_3.make
    maze_3.describe
  end
end
```

Related Patterns

- Abstract Factory and Prototype can be used together

  Abstract Factory can store set of prototypes which are cloned to return product objects

- When Composite or Decorator are used, then Prototype can often be of use