Prototype Pattern – Creational

- Intent

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

Prototype – 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**

Prototype – Motivation – 2

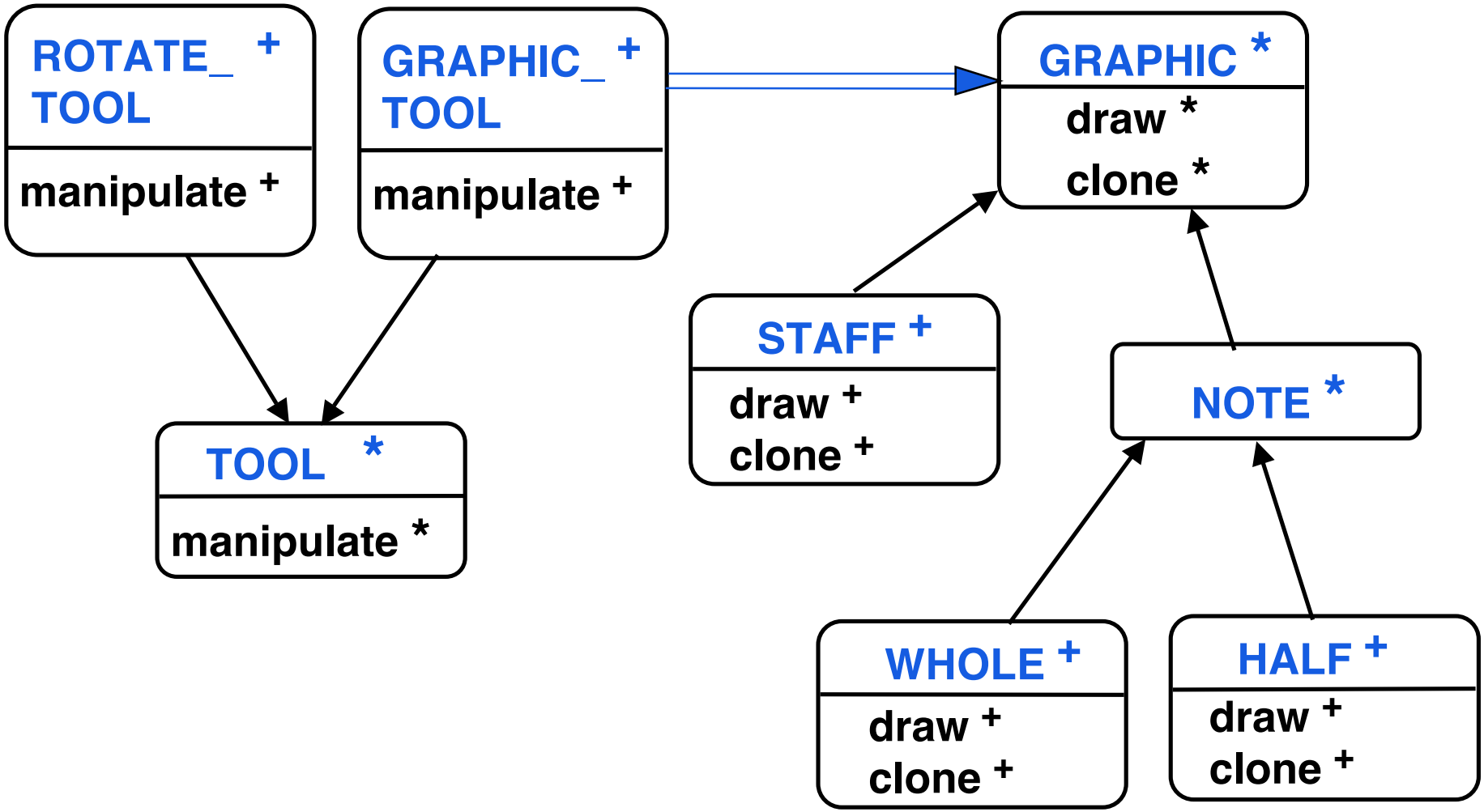
- Graphic_Tool doesn't know how to create instances of music classes

  Could subclass Graphic_Tool for each kind of music object

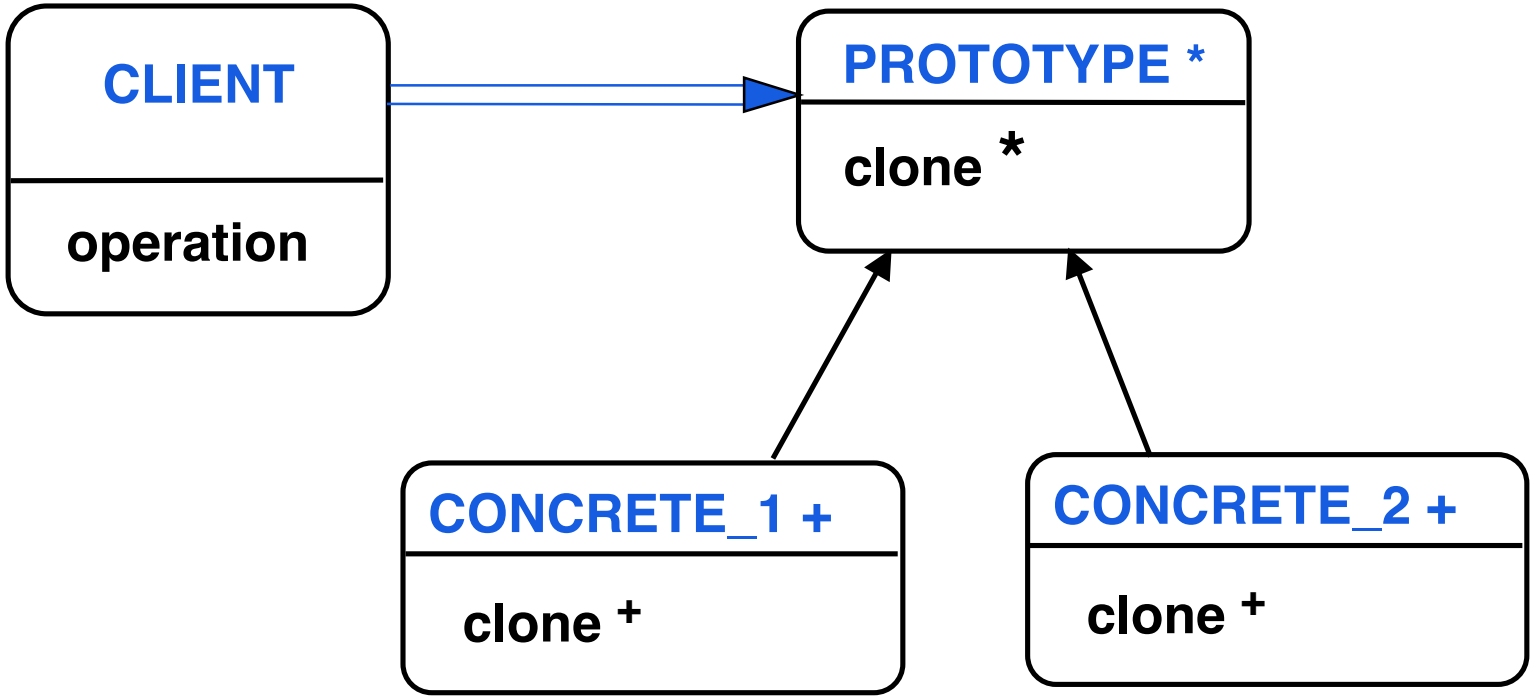
  But have lots of classes with insignificant variations

- 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

Prototype – Example Architecture



Prototype – Abstract Structure



Prototype – 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

Prototype – 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

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

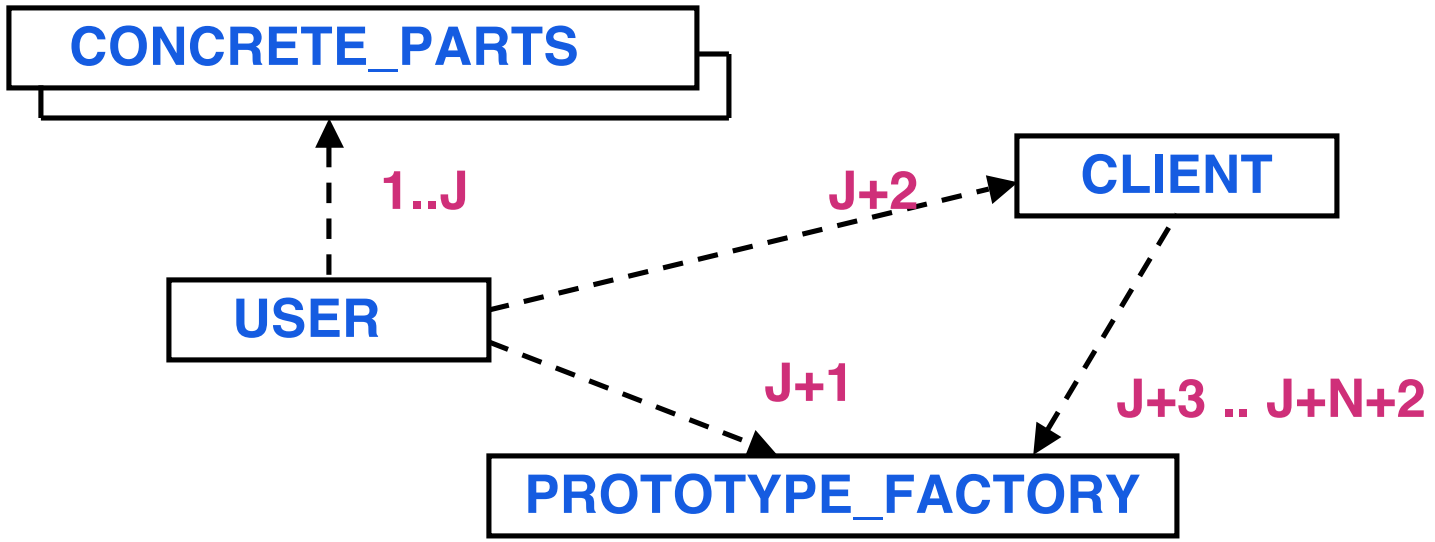
  > **When instances of a class can have one of a few different combinations of state**

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

Prototype – 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 (…)



Prototype – Consequences

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

  Adding & removing products at run time

  Register a prototype instance with client

Prototype – Consequences – 2

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

Prototype – Consequences – 3

- Reduced subclassing
  - Factory Method produces hierarchy of creator classes that parallels product classes
  - Cloning avoids parallel hierarchy

    Biggest benefit is 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
- Configuring an application with classes dynamically

  C++ lets you load classes dynamically

Prototype – Consequences – 4

- Liability

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

Prototype – Implementation

```
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 ) is
    do
        prototype_maze := m ; prototype_door := d
        prototype_room:= r ; prototype_wall := w
    end
     -- next slide for the make components methods
end
```

Prototype – Implementation – 2

```
make_wall : WALL is
do      Result := prototype_wall.twin
end

make_door ( r1 : ROOM ; r2 : ROOM ): DOOR is
do    Result := prototype_door.twin
      Result.set_rooms (r1, r2 )
end

make_room ( id : INTEGER ) : ROOM is
do    Result := prototype_room.twin
      Result.set_id ( id )
end

make_maze : MAZE is
do   Result := prototype_maze.twin
end
```

Prototype – Implementation – 3

```
// Client uses the prototype -- assuming subclasses are
// implemented

game : MAZE_GAME
proto_factory : MAZE_PROTOTYPE_FACTORY
m : MAZE ; d : DOOR ; w : WALL ; r : ROOM
create m.make  ; create d.make
create w.make  ; create r.make
create proto _factory.make ( m, r, d, w )
// create_maze expects a prototype instead of
// abstract factory
create game . create_maze (proto _factory )
```

Prototype – Implementation – 4

```
// To get a different types of mazes create with different
// prototypes

m : ENCHANTED_MAZE
d : DOOR_NEEDING_SPELL
w : WALL
r : ROOM

create m.make  ; create d.make
create w.make  ; create r.make

create prot_factory.make ( m, r, d, w )
create game . create_maze (proto _factory )
```

Prototype – Related Patterns

- Abstract Factory and Protoype can be used together

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

- When Composite and Decorator are used together, then Prototype can also be used