Prototype - Object Creational

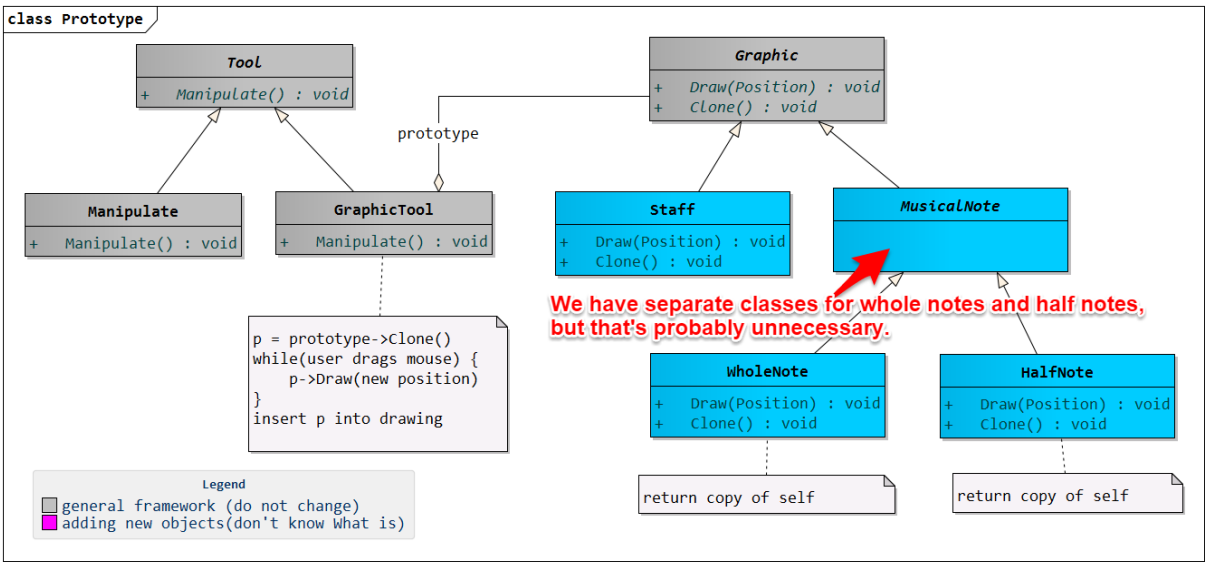
Intent (意图)
Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

Motivation (动机)
- You could build an editor for music scores by customizing a general framework for graphical editors and adding new objects that represent notes, rests, and staves.
- Let's assume the framework provides an abstract **Graphic** class for graphical components, like notes and staves.
- It'll provide an abstract ***Tool*** class for defining tools like those in the palette.
- The framework also predefines a **GraphicTool** subclass for tools that create instances of graphical objects and add them to the document.

But GraphicTool presents a problem to the framework designer.
- The classes for notes and staves are specific to our application, but the GraphicTool class belongs to the framework.
- GraphicTool doesn't know how to create instances of our music classes to add to the score.
- We could subclass GraphicTool for each kind of music object they instantiate, but that would produce lots of subclass that differ only in the kind of music object they instantiate.

Object composition is a flexible alternative to subclassing.

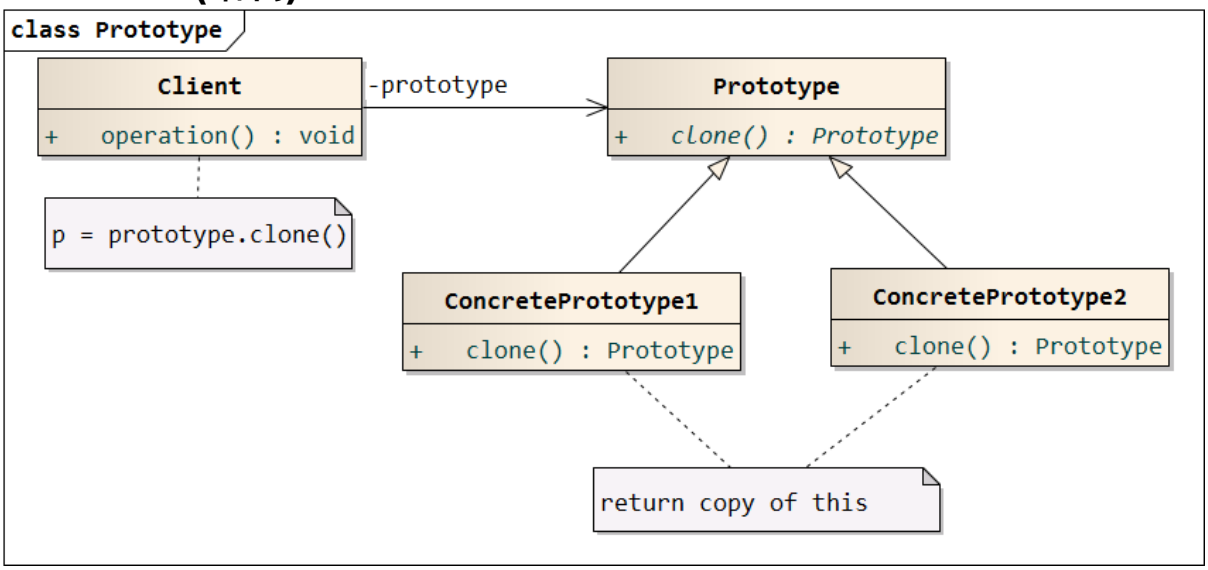
The Solution
lies in making GraphicTool create a new Graphic by copying or "cloning" an instance of a Graphic subclass. We call this instance a **prototype**.



Applicability (适用性)
when a system should be independent of how its products are created, composed, and

represented; and
- when the classes to instantiate are specified at run-time, for example, by 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 only a few different combinations of state. It may be more convenient to install a corresponding number of prototypes and clone them rather than instantiating the class manually, each time with the appropriate state.

Structure (结构)



Participants (参与者)

- Prototype (Graphic)
  declares an interface for cloning itself.
- ConcretePrototype (Staff, WholeNote, HalfNote)
  implements an operation for cloning itself.
- Client (GraphicTool)
  creates a new object by asking a prototype to clone itself.

Collaborations (协作)

A client asks a prototype to clone itself.

Consequences (结果)

Prototype has many of the same consequences that **Abstract Factory** and **Builder** have:
- It hides the concrete product classes from the client
- thereby reducing the number of names clients know about.
- let a client work with application-specific classes without modification.

Additional benefits of the **Prototype** pattern are listed below:
- Adding and removing products at run-time.
- Specifying new objects by varying values.
- Specifying new objects by varying structure.

Such applications reflect the **Composite** and **Decorator** patterns.
- Reduced subclassing.
- Configuring an application with classes dynamically.

The main liability:
- each subclass of **Prototype** must implement the **Clone** operation, which may be difficult.
    - when the classes under consideration already exist.
    - Implementing **Clone** can be difficult when their internals include objects that don't support copying or have circular references.

Implementation (实现)
- Using prototype manager.
    When the number of prototypes in a system isn't fixed, keep a registry of available prototypes.
- Implementing the Clone operation
    Note that shallow copy versus deep copy
- Initializing clones.
    You generally can't pass these values in the Clone operation, because their number will vary between classes of prototypes.
    Passing parameters in the Clone operation precludes a uniform cloning interface.

Sample Code (代码示例)
- Maze Game

```
//MazePrototypeFactory will be initialized with prototypes of the
objects it will create so that we don't have to subclass it just
to change the classes of walls or rooms it creates.
class MazePrototypeFactory : public MazeFactory
{
public:
MazePrototypeFactory(Maze *, Room *, Wall *, Door *);

virtual Maze *MakeMaze() const override;
virtual Room *MakeRoom(int) const override;
virtual Wall *MakeWall() const override;
virtual Door *MakeDoor(Room *, Room *) const override;

private:
Maze *_prototypeMaze;
Room *_prototypeRoom;
Wall *_prototypeWall;
Door *_prototypeDoor;
};

//the new constructor simply initializes its prototypes
MazePrototypeFactory::MazePrototypeFactory(Maze *m, Room *r, Wall *w,
Door *d)
: _prototypeMaze(m)
, _prototypeRoom(r)
```

```
, _prototypeWall(w)
, _prototypeDoor(d)
{

}

//Each clones a prototype and then initializes it.
Wall *MazePrototypeFactory::MakeWall() const
{
return _prototypeWall->clone();
}

Door *MazePrototypeFactory::MakeDoor(Room *r1, Room *r2) const
{
Door *door = _prototypeDoor->clone();
door->initialize(r1, r2);

return door;
}


//We can use MazePrototypeFactory to create a prototypical or
default maze just by initializing it with prototypes of basic maze
components
MazeGame game;
MazePrototypeFactory simpleMazeFactory(new Maze, new Room, new Wall,
new Door);
Maze *maze = game.createMaze(simpleMazeFactory);

//To change the type of maze, we initialize MazePrototypeFactory
with a diferent set of prototypes.
MazePrototypeFactory bombedMazeFactory(new Maze, new RoomWithABomb,
new BombedWall, new Door);
```

Known Uses (已知应用)

...

Related Patterns (相关模式)

- **Prototype** and **Abstract Factory** are competing patterns in some ways, They can also be used together, however. An **Abstract Factory** might store a set of prototypes from which to clone and return product objects.
- Designs that make heavy use of the ***Composite*** and **Decorator** patterns often can benefit from **Prototype** as well.