Prototype

Intent

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

Motivation

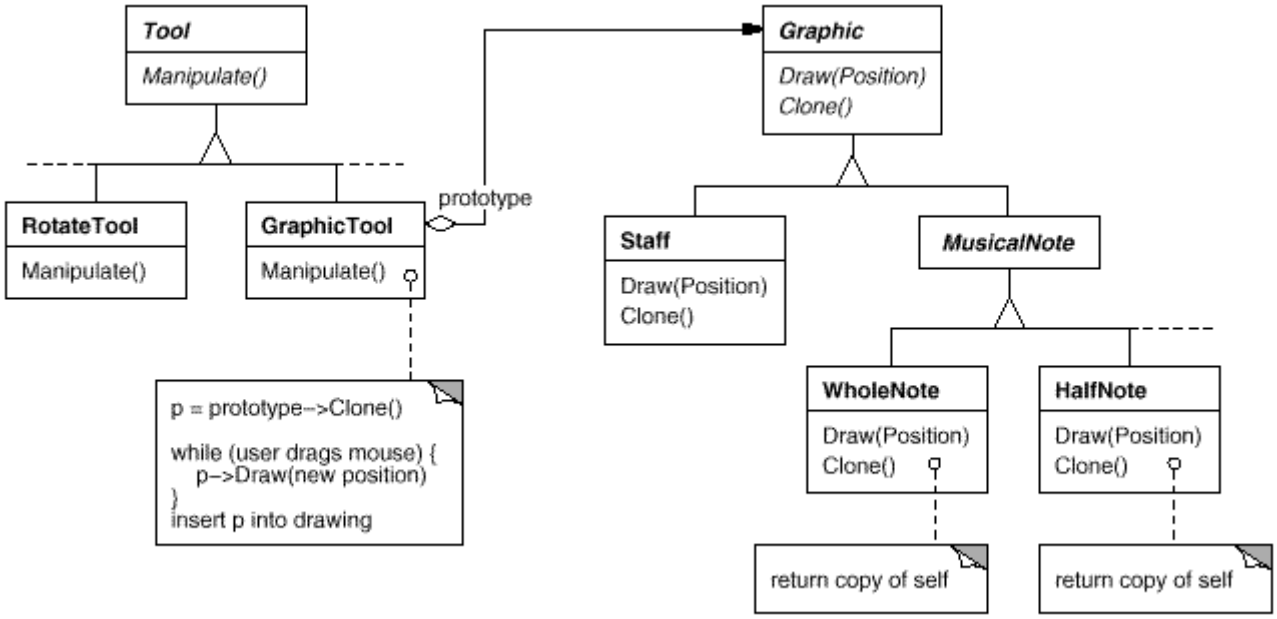
You could build an editor for music scores by customizing a general framework for graphical editors and adding new objects that represent notes, rests, and staves. The editor framework may have a palette of tools for adding these music objects to the score. The palette would also include tools for selecting, moving, and otherwise manipulating music objects. Users will click on the quarter-note tool and use it to add quarter notes to the score. Or they can use the move tool to move a note up or down on the staff, thereby changing its pitch.

Let's assume the framework provides an abstract Graphic class for graphical components, like notes and staves. Moreover, 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, but that would produce lots of subclasses that differ only in the kind of music object they instantiate. We know object composition is a flexible alternative to subclassing. The question is, how can the framework use it to parameterize instances of GraphicTool by the *class* of Graphic they're supposed to create?

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**. GraphicTool is parameterized by the prototype it should clone and add to the document. If all Graphic subclasses support a Clone operation, then the GraphicTool can clone any kind of Graphic.

So in our music editor, each tool for creating a music object is an instance of GraphicTool that's initialized with a different prototype. Each GraphicTool instance will produce a music object by cloning its prototype and adding the clone to the score.



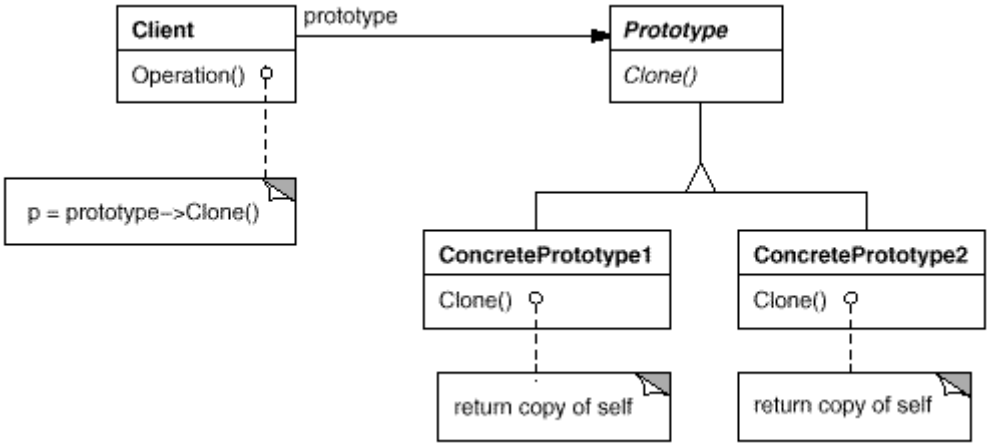
We can use the Prototype pattern to reduce the number of classes even further. We have separate classes for whole notes and half notes, but that's probably unnecessary. Instead they could be instances of the same class initialized with different bitmaps and durations. A tool for creating whole notes becomes just a GraphicTool whose prototype is a MusicalNote initialized to be a whole note. This can reduce the number of classes in the system dramatically. It also makes it easier to add a new kind of note to the music editor.

Applicability

Use the Prototype pattern 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



Participants

- **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

- A client asks a prototype to clone itself.

Consequences

Prototype has many of the same consequences that Abstract Factory (87) and Builder (97) have: It hides the concrete product classes from the client, thereby reducing the number of names clients know about. Moreover, these patterns let a client work with application-specific classes without modification.

Additional benefits of the Prototype pattern are listed below.

1. *Adding and removing products at run-time.* Prototypes let you incorporate a new concrete product class into a system simply by registering a prototypical instance with the client. That's a bit more flexible than other creational patterns, because a client can install and remove prototypes at run-time.
2. *Specifying new objects by varying values.* Highly dynamic systems let you define new behavior through object composition—by specifying values for an object's variables, for example—and not by defining new classes. You effectively define new kinds of objects by instantiating existing classes and registering the instances as prototypes of client objects. A client can exhibit new behavior by delegating responsibility to the prototype.

This kind of design lets users define new "classes" without programming. In fact, cloning a prototype is similar to instantiating a class. The Prototype pattern can greatly reduce the number of classes a system needs. In our music editor, one GraphicTool class can create a limitless variety of music objects.

3. *Specifying new objects by varying structure.* Many applications build objects from parts and subparts. Editors for circuit design, for example, build circuits out of subcircuits.[1] For convenience, such applications often let you instantiate complex, user-defined structures, say, to use a specific subcircuit again and again.

   The Prototype pattern supports this as well. We simply add this subcircuit as a prototype to the palette of available circuit elements. As long as the composite circuit object implements Clone as a deep copy, circuits with different structures can be prototypes.

4. *Reduced subclassing.* Factory Method (107) often produces a hierarchy of Creator classes that parallels the product class hierarchy. The Prototype pattern lets you clone a prototype instead of asking a factory method to make a new object. Hence you don't need a Creator class hierarchy at all. This benefit applies primarily to languages like C++ that don't treat classes as first-class objects. Languages that do, like Smalltalk and Objective C, derive less benefit, since you can always use a class object as a creator. Class objects already act like prototypes in these languages.
5. *Configuring an application with classes dynamically.* Some run-time environments let you load classes into an application dynamically. The Prototype pattern is the key to exploiting such facilities in a language like C++.

   An application that wants to create instances of a dynamically loaded class won't be able to reference its constructor statically. Instead, the run-time environment creates an instance of each class automatically when it's loaded, and it registers the instance with a prototype manager (see the Implementation section). Then the application can ask the prototype manager for instances of newly loaded classes, classes that weren't linked with the program originally. The ET++ application framework [WGM88] has a run-time system that uses this scheme.

The main liability of the Prototype pattern is that each subclass of Prototype must implement the `Clone` operation, which may be difficult. For example, adding `Clone` is 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

Prototype is particularly useful with static languages like C++, where classes are not objects, and little or no type information is available at run-time. It's less important in languages like Smalltalk or Objective C that provide what amounts to a prototype (i.e., a class object) for creating instances of each class. This pattern is built into prototype-based languages like Self [US87], in which all object creation happens by cloning a prototype.

Consider the following issues when implementing prototypes:

1. *Using a prototype manager.* When the number of prototypes in a system isn't fixed (that is, they can be created and destroyed dynamically), keep a registry of available prototypes. Clients won't manage prototypes themselves but will store and retrieve them from the registry. A client will ask the registry for a prototype before cloning it. We call this registry a **prototype manager**.

   A prototype manager is an associative store that returns the prototype matching a given key. It has operations for registering a prototype under a key and for unregistering it. Clients can change or even browse through the registry at run-time. This lets clients extend and take inventory on the system without writing code.

2. *Implementing the Clone operation.* The hardest part of the Prototype pattern is implementing the Clone operation correctly. It's particularly tricky when object structures contain circular references.

   Most languages provide some support for cloning objects. For example, Smalltalk provides an implementation of `copy` that's inherited by all subclasses of Object. C++ provides a copy constructor. But these facilities don't solve the "shallow copy versus deep copy" problem [GR83]. That is, does cloning an object in turn clone its instance variables, or do the clone and original just share the variables?

   A shallow copy is simple and often sufficient, and that's what Smalltalk provides by default. The default copy constructor in C++ does a member-wise copy, which means pointers will be shared between the copy and the original. But cloning prototypes with complex structures usually requires a deep copy, because the clone and the original must be independent. Therefore you must ensure that the clone's components are clones of the prototype's components. Cloning forces you to decide what if anything will be shared.

   If objects in the system provide Save and Load operations, then you can use them to provide a default implementation of Clone simply by saving the object and loading it back immediately. The Save operation saves the object into a memory buffer, and Load creates a duplicate by reconstructing the object from the buffer.

3. *Initializing clones.* While some clients are perfectly happy with the clone as is, others will want to initialize some or all of its internal state to values of their choosing. You generally can't pass these values in the Clone operation, because their number will vary between classes of prototypes. Some prototypes might need multiple initialization parameters; others won't need any. Passing parameters in the Clone operation precludes a uniform cloning interface.

   It might be the case that your prototype classes already define operations for (re)setting key pieces of state. If so, clients may use these operations immediately after cloning. If not, then you may have to introduce an `Initialize` operation (see the Sample Code section) that takes initialization

parameters as arguments and sets the clone's internal state accordingly. Beware of deep-copying Clone operations—the copies may have to be deleted (either explicitly or within `Initialize`) before you reinitialize them.

Sample Code

We'll define a `MazePrototypeFactory` subclass of the `MazeFactory` class (page 92). `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.

`MazePrototypeFactory` augments the `MazeFactory` interface with a constructor that takes the prototypes as arguments:

```
class MazePrototypeFactory : public MazeFactory {
public:
    MazePrototypeFactory(Maze*, Wall*, Room*, Door*);

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

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

The new constructor simply initializes its prototypes:

```
MazePrototypeFactory::MazePrototypeFactory (
    Maze* m, Wall* w, Room* r, Door* d
) {
   _prototypeMaze = m;
   _prototypeWall = w;
   _prototypeRoom = r;
   _prototypeDoor = d;
}
```

The member functions for creating walls, rooms, and doors are similar: Each clones a prototype and then initializes it. Here are the definitions of `MakeWall` and `MakeDoor`:

```
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 Wall, new Room, new Door
);

Maze* maze = game.CreateMaze(simpleMazeFactory);
```

To change the type of maze, we initialize `MazePrototypeFactory` with a different set of prototypes. The following call creates a maze with a `BombedDoor` and a `RoomWithABomb`:

```
MazePrototypeFactory bombedMazeFactory(
    new Maze, new BombedWall,
    new RoomWithABomb, new Door
);
```

An object that can be used as a prototype, such as an instance of `Wall`, must support the `Clone` operation. It must also have a copy constructor for cloning. It may also need a separate operation for reinitializing internal state. We'll add the `Initialize` operation to `Door` to let clients initialize the clone's rooms.

Compare the following definition of `Door` to the one on page 83:

```
class Door : public MapSite {
public:
    Door();
    Door(const Door&);

    virtual void Initialize(Room*, Room*);
    virtual Door* Clone() const;

    virtual void Enter();
    Room* OtherSideFrom(Room*);
private:
    Room* _room1;
    Room* _room2;
};

Door::Door (const Door& other) {
    _room1 = other._room1;
    _room2 = other._room2;
}

void Door::Initialize (Room* r1, Room* r2) {
    _room1 = r1;
    _room2 = r2;
}

Door* Door::Clone () const {
    return new Door(*this);
}
```

The `BombedWall` subclass must override `Clone` and implement a corresponding copy constructor.

```
class BombedWall : public Wall {
public:
    BombedWall();
    BombedWall(const BombedWall&);

    virtual Wall* Clone() const;
    bool HasBomb();
private:
    bool _bomb;
};

BombedWall::BombedWall (const BombedWall& other) : Wall(other) {
    _bomb = other._bomb;
}

Wall* BombedWall::Clone () const {
    return new BombedWall(*this);
}
```

Although `BombedWall::Clone` returns a `Wall*`, its implementation returns a pointer to a new instance of a subclass, that is, a `BombedWall*`. We define `Clone` like this in the base class to ensure that clients that clone the prototype don't have to know about their concrete subclasses. Clients should never need to downcast the return value of `Clone` to the desired type.

In Smalltalk, you can reuse the standard `copy` method inherited from `Object` to clone any `MapSite`. You can use `MazeFactory` to produce the prototypes you'll need; for example, you can create a room by supplying the name `#room`. The `MazeFactory` has a dictionary that maps names to prototypes. Its `make:` method looks like this:

```
make: partName
    ^ (partCatalog at: partName) copy
```

Given appropriate methods for initializing the `MazeFactory` with prototypes, you could create a simple maze with the following code:

```
CreateMaze
    on: (MazeFactory new
        with: Door new named: #door;
        with: Wall new named: #wall;
        with: Room new named: #room;
        yourself)
```

where the definition of the `on:` class method for `CreateMaze` would be

```
on: aFactory
    | room1 room2 |
    room1 := (aFactory make: #room) location: 1@1.
    room2 := (aFactory make: #room) location: 2@1.
    door := (aFactory make: #door) from: room1 to: room2.

    room1
        atSide: #north put: (aFactory make: #wall);
        atSide: #east put: door;
        atSide: #south put: (aFactory make: #wall);
        atSide: #west put: (aFactory make: #wall).
    room2
```

```
        atSide: #north put: (aFactory make: #wall);
        atSide: #east put: (aFactory make: #wall);
        atSide: #south put: (aFactory make: #wall);
        atSide: #west put: door.
    ^ Maze new
        addRoom: room1;
        addRoom: room2;
        yourself
```

Known Uses

Perhaps the first example of the Prototype pattern was in Ivan Sutherland's Sketchpad system [Sut63]. The first widely known application of the pattern in an object-oriented language was in ThingLab, where users could form a composite object and then promote it to a prototype by installing it in a library of reusable objects [Bor81]. Goldberg and Robson mention prototypes as a pattern [GR83], but Coplien [Cop92] gives a much more complete description. He describes idioms related to the Prototype pattern for C++ and gives many examples and variations.

Etgdb is a debugger front-end based on ET++ that provides a point-and-click interface to different line-oriented debuggers. Each debugger has a corresponding DebuggerAdaptor subclass. For example, GdbAdaptor adapts etgdb to the command syntax of GNU gdb, while SunDbxAdaptor adapts etgdb to Sun's dbx debugger. Etgdb does not have a set of DebuggerAdaptor classes hard-coded into it. Instead, it reads the name of the adaptor to use from an environment variable, looks for a prototype with the specified name in a global table, and then clones the prototype. New debuggers can be added to etgdb by linking it with the DebuggerAdaptor that works for that debugger.

The "interaction technique library" in Mode Composer stores prototypes of objects that support various interaction techniques [Sha90]. Any interaction technique created by the Mode Composer can be used as a prototype by placing it in this library. The Prototype pattern lets Mode Composer support an unlimited set of interaction techniques.

The music editor example discussed earlier is based on the Unidraw drawing framework [VL90].

Related Patterns

Prototype and Abstract Factory (87) are competing patterns in some ways, as we discuss at the end of this chapter. 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 (163) and Decorator (175) patterns often can benefit from Prototype as well.