```
    return aMaze;
}
```

Different games can subclass `MazeGame` to specialize parts of the maze. `MazeGame` subclasses can redefine some or all of the factory methods to specify variations in products. For example, a `BombedMazeGame` can redefine the `Room` and `Wall` products to return the bombed varieties:

```
class BombedMazeGame : public MazeGame {
public:
    BombedMazeGame();

    virtual Wall* MakeWall() const
        { return new BombedWall; }

    virtual Room* MakeRoom(int n) const
        { return new RoomWithABomb(n); }
};
```

An `EnchantedMazeGame` variant might be defined like this:

```
class EnchantedMazeGame : public MazeGame {
public:
    EnchantedMazeGame();

    virtual Room* MakeRoom(int n) const
        { return new EnchantedRoom(n, CastSpell()); }

    virtual Door* MakeDoor(Room* r1, Room* r2) const
        { return new DoorNeedingSpell(r1, r2); }
protected:
    Spell* CastSpell() const;
};
```

Known Uses

Factory methods pervade toolkits and frameworks. The preceding document example is a typical use in MacApp and ET++ [WGM88]. The manipulator example is from Unidraw.

Class View in the Smalltalk-80 Model/View/Controller framework has a method defaultController that creates a controller, and this might appear to be a factory method [Par90]. But subclasses of View specify the class of their default controller by defining defaultControllerClass, which returns the class from which defaultContr[illegible]r creates instances. So defaultControllerClass is the real factory method, that is [illegible]he method that subclasses should override.

A more [illegible]oteric example in Smalltalk-80 is the factory method parserClass defined by Beha[illegible]or (a superclass of all objects representing classes). This enables a class