Chapter 3

Creational Patterns

Creational design patterns abstract the instantiation process. They help make a system independent of how its objects are created, composed, and represented. A class creational pattern uses inheritance to vary the class that's instantiated, whereas an object creational pattern will delegate instantiation to another object.

Creational patterns become important as systems evolve to depend more on object composition than class inheritance. As that happens, emphasis shifts away from hard-coding a fixed set of behaviors toward defining a smaller set of fundamental behaviors that can be composed into any number of more complex ones. Thus creating objects with particular behaviors requires more than simply instantiating a class.

There are two recurring themes in these patterns. First, they all encapsulate knowledge about which concrete classes the system uses. Second, they hide how instances of these classes are created and put together. All the system at large knows about the objects is their interfaces as defined by abstract classes. Consequently, the creational patterns give you a lot of flexibility in *what* gets created, *who* creates it, *how* it gets created, and *when*. They let you configure a system with "product" objects that vary widely in structure and functionality. Configuration can be static (that is, specified at compile-time) or dynamic (at run-time).

Sometimes creational patterns are competitors. For example, there are cases when either Prototype (117) or Abstract Factory (87) could be used profitably. At other times they are complementary: Builder (97) can use one of the other patterns to implement which components get built. Prototype (117) can use Singleton (127) in its implementation.

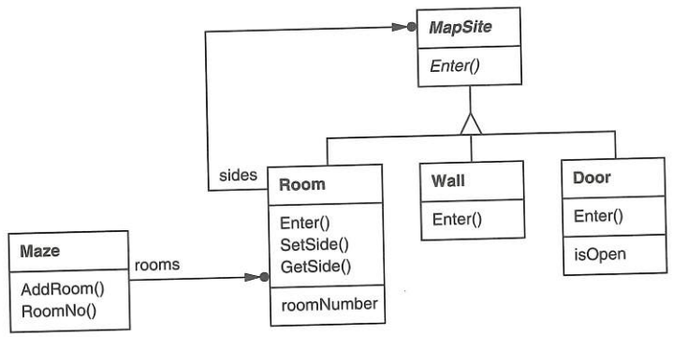
Because the creational patterns are closely related, we'll study all five of them together to highlight their similarities and differences. We'll also use a common example—building a maze for a computer game—to illustrate their implementations. The maze and the game will vary slightly from pattern to pattern. Sometimes the game will be simply to find your way out of a maze; in that case the player will probably only have a local view of the maze. Sometimes mazes contain problems to solve and dangers to

overcome, and these games may provide a map of the part of the maze that has been explored.

We'll ignore many details of what can be in a maze and whether a maze game has a single or multiple players. Instead, we'll just focus on how mazes get created. We define a maze as a set of rooms. A room knows its neighbors; possible neighbors are another room, a wall, or a door to another room.

The classes `Room`, `Door`, and `Wall` define the components of the maze used in all our examples. We define only the parts of these classes that are important for creating a maze. We'll ignore players, operations for displaying and wandering around in a maze, and other important functionality that isn't relevant to building the maze.

The following diagram shows the relationships between these classes:



Each room has four sides. We use an enumeration `Direction` in C++ implementations to specify the north, south, east, and west sides of a room:

```
enum Direction {North, South, East, West};
```

The Smalltalk implementations use corresponding symbols to represent these directions.

The class `MapSite` is the common abstract class for all the components of a maze. To simplify the example, `MapSite` defines only one operation, `Enter`. Its meaning depends on what you're entering. If you enter a room, then your location changes. If you try to enter a door, then one of two things happen: If the door is open, you go into the next room. If the door is closed, then you hurt your nose.

```
class MapSite {
public:
    virtual void Enter() = 0;
};
```

`Enter` provides a simple basis for more sophisticated game operations. For example, if you are in a room and say "Go East," the game can simply determine which `MapSite` is immediately to the east and then call `Enter` on it. The subclass-specific `Enter` operation will figure out whether your location changed or your nose got hurt. In a real game, `Enter` could take the player object that's moving about as an argument.

`Room` is the concrete subclass of `MapSite` that defines the key relationships between components in the maze. It maintains references to other `MapSite` objects and stores a room number. The number will identify rooms in the maze.

```
class Room : public MapSite {
public:
    Room(int roomNo);

    MapSite* GetSide(Direction) const;
    void SetSide(Direction, MapSite*);

    virtual void Enter();

private:
    MapSite* _sides[4];
    int _roomNumber;
};
```

The following classes represent the wall or door that occurs on each side of a room.

```
class Wall : public MapSite {
public:
    Wall();

    virtual void Enter();
};

class Door : public MapSite {
public:
    Door(Room* = 0, Room* = 0);

    virtual void Enter();
    Room* OtherSideFrom(Room*);

private:
    Room* _room1;
    Room* _room2;
    bool _isOpen;
};
```

We need to know about more than just the parts of a maze. We'll also define a `Maze` class to represent a collection of rooms. `Maze` can also find a particular room given a room number using its `RoomNo` operation.

```
class Maze {
public:
    Maze();

    void AddRoom(Room*);
    Room* RoomNo(int) const;
private:
    // ...
};
```

`RoomNo` could do a look-up using a linear search, a hash table, or even a simple array. But we won't worry about such details here. Instead, we'll focus on how to specify the components of a maze object.

Another class we define is `MazeGame`, which creates the maze. One straightforward way to create a maze is with a series of operations that add components to a maze and then interconnect them. For example, the following member function will create a maze consisting of two rooms with a door between them:

```
Maze* MazeGame::CreateMaze () {
    Maze* aMaze = new Maze;
    Room* r1 = new Room(1);
    Room* r2 = new Room(2);
    Door* theDoor = new Door(r1, r2);

    aMaze->AddRoom(r1);
    aMaze->AddRoom(r2);

    r1->SetSide(North, new Wall);
    r1->SetSide(East, theDoor);
    r1->SetSide(South, new Wall);
    r1->SetSide(West, new Wall);

    r2->SetSide(North, new Wall);
    r2->SetSide(East, new Wall);
    r2->SetSide(South, new Wall);
    r2->SetSide(West, theDoor);

    return aMaze;
}
```

This function is pretty complicated, considering that all it does is create a maze with two rooms. There are obvious ways to make it simpler. For example, the `Room` constructor could initialize the sides with walls ahead of time. But that just moves the code somewhere else. The real problem with this member function isn't its size but its *inflexibility*. It hard-codes the maze layout. Changing the layout means changing this member function, either by overriding it—which means reimplementing the whole thing—or by changing parts of it—which is error-prone and doesn't promote reuse.

The creational patterns show how to make this design more *flexible*, not necessarily smaller. In particular, they will make it easy to change the classes that define the components of a maze.

Suppose you wanted to reuse an existing maze layout for a new game containing (of all things) enchanted mazes. The enchanted maze game has new kinds of components, like `DoorNeedingSpell`, a door that can be locked and opened subsequently only with a spell; and `EnchantedRoom`, a room that can have unconventional items in it, like magic keys or spells. How can you change `CreateMaze` easily so that it creates mazes with these new classes of objects?

In this case, the biggest barrier to change lies in hard-coding the classes that get instantiated. The creational patterns provide different ways to remove explicit references to concrete classes from code that needs to instantiate them:

- If `CreateMaze` calls virtual functions instead of constructor calls to create the rooms, walls, and doors it requires, then you can change the classes that get instantiated by making a subclass of `MazeGame` and redefining those virtual functions. This approach is an example of the Factory Method (107) pattern.
- If `CreateMaze` is passed an object as a parameter to use to create rooms, walls, and doors, then you can change the classes of rooms, walls, and doors by passing a different parameter. This is an example of the Abstract Factory (87) pattern.
- If `CreateMaze` is passed an object that can create a new maze in its entirety using operations for adding rooms, doors, and walls to the maze it builds, then you can use inheritance to change parts of the maze or the way the maze is built. This is an example of the Builder (97) pattern.
- If `CreateMaze` is parameterized by various prototypical room, door, and wall objects, which it then copies and adds to the maze, then you can change the maze's composition by replacing these prototypical objects with different ones. This is an example of the Prototype (117) pattern.

The remaining creational pattern, Singleton (127), can ensure there's only one maze per game and that all game objects have ready access to it—without resorting to global variables or functions. Singleton also makes it easy to extend or replace the maze without touching existing code.

Chapter 4

Structural Patterns

Structural patterns are concerned with how classes and objects are composed to form larger structures. Structural *class* patterns use inheritance to compose interfaces or implementations. As a simple example, consider how multiple inheritance mixes two or more classes into one. The result is a class that combines the properties of its parent classes. This pattern is particularly useful for making independently developed class libraries work together. Another example is the class form of the Adapter (139) pattern. In general, an adapter makes one interface (the adaptee's) conform to another, thereby providing a uniform abstraction of different interfaces. A class adapter accomplishes this by inheriting privately from an adaptee class. The adapter then expresses its interface in terms of the adaptee's.

Rather than composing interfaces or implementations, structural *object* patterns describe ways to compose objects to realize new functionality. The added flexibility of object composition comes from the ability to change the composition at run-time, which is impossible with static class composition.

Composite (163) is an example of a structural object pattern. It describes how to build a class hierarchy made up of classes for two kinds of objects: primitive and composite. The composite objects let you compose primitive and other composite objects into arbitrarily complex structures. In the Proxy (207) pattern, a proxy acts as a convenient surrogate or placeholder for another object. A proxy can be used in many ways. It can act as a local representative for an object in a remote address space. It can represent a large object that should be loaded on demand. It might protect access to a sensitive object. Proxies provide a level of indirection to specific properties of objects. Hence they can restrict, enhance, or alter these properties.

The Flyweight (195) pattern defines a structure for sharing objects. Objects are shared for at least two reasons: efficiency and consistency. Flyweight focuses on sharing for space efficiency. Applications that use lots of objects must pay careful attention to the cost of each object. Substantial savings can be had by sharing objects instead of replicating them. But objects can be shared only if they don't define context-dependent

state. Flyweight objects have no such state. Any additional information they need to perform their task is passed to them when needed. With no context-dependent state, Flyweight objects may be shared freely.

Whereas Flyweight shows how to make lots of little objects, Facade (185) shows how to make a single object represent an entire subsystem. A facade is a representative for a set of objects. The facade carries out its responsibilities by forwarding messages to the objects it represents. The Bridge (151) pattern separates an object's abstraction from its implementation so that you can vary them independently.

Decorator (175) describes how to add responsibilities to objects dynamically. Decorator is a structural pattern that composes objects recursively to allow an open-ended number of additional responsibilities. For example, a Decorator object containing a user interface component can add a decoration like a border or shadow to the component, or it can add functionality like scrolling and zooming. We can add two decorations simply by nesting one Decorator object within another, and so on for additional decorations. To accomplish this, each Decorator object must conform to the interface of its component and must forward messages to it. The Decorator can do its job (such as drawing a border around the component) either before or after forwarding a message.

Many structural patterns are related to some degree. We'll discuss these relationships at the end of the chapter.

Chapter 5

Behavioral Patterns

Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects. Behavioral patterns describe not just patterns of objects or classes but also the patterns of communication between them. These patterns characterize complex control flow that's difficult to follow at run-time. They shift your focus away from flow of control to let you concentrate just on the way objects are interconnected.

Behavioral class patterns use inheritance to distribute behavior between classes. This chapter includes two such patterns. Template Method (325) is the simpler and more common of the two. A template method is an abstract definition of an algorithm. It defines the algorithm step by step. Each step invokes either an abstract operation or a primitive operation. A subclass fleshes out the algorithm by defining the abstract operations. The other behavioral class pattern is Interpreter (243), which represents a grammar as a class hierarchy and implements an interpreter as an operation on instances of these classes.

Behavioral object patterns use object composition rather than inheritance. Some describe how a group of peer objects cooperate to perform a task that no single object can carry out by itself. An important issue here is how peer objects know about each other. Peers could maintain explicit references to each other, but that would increase their coupling. In the extreme, every object would know about every other. The Mediator (273) pattern avoids this by introducing a mediator object between peers. The mediator provides the indirection needed for loose coupling.

Chain of Responsibility (223) provides even looser coupling. It lets you send requests to an object implicitly through a chain of candidate objects. Any candidate may fulfill the request depending on run-time conditions. The number of candidates is open-ended, and you can select which candidates participate in the chain at run-time.

The Observer (293) pattern defines and maintains a dependency between objects. The classic example of Observer is in Smalltalk Model/View/Controller, where all views of the model are notified whenever the model's state changes.

Other behavioral object patterns are concerned with encapsulating behavior in an object and delegating requests to it. The Strategy (315) pattern encapsulates an algorithm in an object. Strategy makes it easy to specify and change the algorithm an object uses. The Command (233) pattern encapsulates a request in an object so that it can be passed as a parameter, stored on a history list, or manipulated in other ways. The State (305) pattern encapsulates the states of an object so that the object can change its behavior when its state object changes. Visitor (331) encapsulates behavior that would otherwise be distributed across classes, and Iterator (257) abstracts the way you access and traverse objects in an aggregate.