

Análisis del rendimiento de los patrones de diseño creacionales aplicado al desarrollo de Videojuegos

Jesús Fermín Villar Ramírez
ESNE Escuela Universitaria, Madrid, Spain.
hello@pokoidev.com

Abstract

Los patrones de diseño facilitan la labor del programador, permitiendo una mejor lectura y reusabilidad del código.

La mayoría de ellos se basan en capsular las operaciones concretas, separando las operaciones de las representaciones de las mismas.

Aquí se va a analizar qué gasto tiene realizar estos ajustes para la comodidad del desarrollador y qué impacto tiene sobre el rendimiento del programa.

I. INTRODUCCIÓN

En videojuegos, el paradigma actual es el de la programación orientada a objetos, por lo que el enfoque de esta investigación estará centrada en dicho paradigma.

La programación orientada a objetos está fundamentada en el papel de los objetos. Cada uno de estos, es un paquete de datos y operaciones de dichos datos. A su vez, los objetos pueden relacionarse con otros objetos de diversas maneras.

Un patrón de diseño se puede definir como una solución simple, y comúnmente

extendida, a un problema específico con una gran flexibilidad y reusabilidad. Por tanto, no todos los patrones solucionan todos los problemas o los mismos problemas, pero sí hay algunos patrones que solucionan problemas similares.

Siguiendo la clasificación establecida por el grupo Gang of Four (GoF) en su libro “*Design Patterns. Elements of Reusable Object-Oriented Software*”, los patrones de diseño se pueden organizar según su propósito en:

- Patrones creacionales: Abstract Factory, Builder, Factory Method, Prototype, Singleton.
- Patrones estructurales: Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy.
- Patrones de comportamiento: Chain of responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method, Visitor.

El propósito de este artículo es analizar el rendimiento del uso de tres de los principales patrones creacionales (Abstract Factory, Builder y Prototype) frente a no usar ninguno.

II. MÉTODOS

A. Modelado del entorno

Para realizar las pruebas se ha modelado un entorno de creación de bosques simples con tres tipos de elementos distintos: árboles, piedras y senderos.

a) Árboles. Los árboles se han modelado con tres características principales: altura, anchura y un conjunto de vértices (1000 vértices por árbol) a modo de simulación de la malla del árbol.

b) Piedras. Las piedras se han modelado con tres características principales: peso, tamaño y un conjunto de vértices (500 vértices por piedra) a modo de simulación de la malla de la piedra.

c) Senderos. Los senderos se han modelado con cuatro características principales: unas coordenadas tridimensionales de inicio, coordenadas tridimensionales de final, anchura y número de curvaturas.

Este modelado es un modelado muy simple, casi anecdótico, que deja al lado aspectos importantes en videojuegos como las transformadas de cada elemento, componentes de colisión etcétera. El objetivo de esta investigación se vale de un modelado tan simple como el presentado.

B. Versión estándar

Lo primero para poder evaluar la consistencia y rendimiento de los patrones de diseño comentados es poder tener un punto de partida neutro con el que comparar. A esto se le ha llamado la versión estándar.

En esta versión se construye el bosque de una manera tradicional, llamando a los constructores de cada uno de los elementos en la función principal.

A continuación se muestra un fragmento del código (en C++) utilizado para realizar esta prueba:

```
void CreateForest()
{
    forest = new Forest;

    for (size_t i = 0; i < number_of_trees; i++)
    {
        Tree* tree = new Tree();
        forest->add_tree(tree);
    }

    for (size_t i = 0; i < number_of_stones; i++)
    {
        Stone* stone = new Stone();
        forest->add_stone(stone);
    }

    for (size_t i = 0; i < number_of_paths; i++)
    {
        Path* path = new Path();
        forest->add_path(path);
    }
}
```

Generación estándar del bosque

C. Versión Abstract Factory

La versión de Abstract Factory se apoya en la creación de una clase auxiliar que implemente una serie de métodos virtuales que realicen la creación de elementos concretos.

La implementación del patrón de Abstract Factory podría apoyarse en el uso del patrón Singleton para garantizar acceso global a la factoría, aunque en el ejemplo descrito no se implementa para mezclar lo mínimo posible los diferentes patrones de diseño y hacer un análisis más acertado.

El principio del Abstract Factory es implementar interfaces de creación de objetos y que sean las subclases las que se encarguen de crearlas de manera concreta. Para simplificar el proceso, en el ejemplo se realiza todo en la clase base.

Esto se realiza de una manera similar a la siguiente:

```
class AbstractFactoryForest
{
public:
    AbstractFactoryForest() {}

    virtual Forest * MakeForest() const
    { return new Forest; }
    virtual Path * MakePath () const
    { return new Path; }
    virtual Stone * MakeStone () const
    { return new Stone; }
    virtual Tree * MakeTree () const
    { return new Tree; }

};
```

Abstract Factory

Permitiendo así generar todos los objetos a partir de una misma factoría:

```
void CreateForest(AbstractFactoryForest& factory)
{
    forest = factory.MakeForest();

    for (size_t i = 0; i < number_of_trees; i++)
    {
        Tree * tree = factory.MakeTree();
        forest->add_tree(tree);
    }

    for (size_t i = 0; i < number_of_stones; i++)
    {
        Stone* stone = factory.MakeStone();
        forest->add_stone(stone);
    }

    for (size_t i = 0; i < number_of_paths; i++)
    {
        Path* path = factory.MakePath();
        forest->add_path(path);
    }
}
```

Generación de un bosque mediante Abstract Factory

D. Versión Builder

La versión de Builder, de manera parecida a la anterior, se encapsula la creación de los componentes mediante una clase auxiliar.

Esta hace todas las operaciones necesarias a la hora de construir los objetos.

Se realiza de una manera similar a la siguiente:

```
class BuilderForest
{
private:
    Forest* current_forest;

public:
    BuilderForest()
    {
        BuildForest();
    }

    virtual void BuildForest()
    {
        current_forest = new Forest;
    }

    virtual void BuildTree()
    {
        Tree* tree = new Tree;
        current_forest->add_tree(tree);
    }

    virtual void BuildStone()
    {
        Stone* stone = new Stone;
        current_forest->add_stone(stone);
    }

    virtual void BuildPath()
    {
        Path* path = new Path;
        current_forest->add_path(path);
    }

    Forest* GetForest() { return current_forest; }

};
```

Builder

Este proceso está aún más encapsulado que el anterior, permitiendo un mayor control a la hora de crear los objetos y una mayor abstracción del lado del cliente:

```
void CreateForest(BuilderForest & builder)
{
    forest = builder.GetForest();

    for (size_t i = 0; i < number_of_trees; i++)
```

```

    {
        builder.BuildTree();
    }

    for (size_t i = 0; i < number_of_stones; i++)
    {
        builder.BuildStone();
    }

    for (size_t i = 0; i < number_of_paths; i++)
    {
        builder.BuildPath();
    }
}

```

Generación de un bosque mediante Builder

E. Versión Prototype

La versión de Prototype, la más diferente a las anteriores, se basa en el concepto de que un objeto tenga la capacidad de crear más objetos del mismo tipo, dependiendo así de los propios objetos su formación .

Para encapsular este proceso al cliente, se puede realizar algo parecido a lo siguiente:

```

class PrototypeForest
{
private:
    Forest * prototype_forest;
    Stone * prototype_stone;
    Tree * prototype_tree;
    Path * prototype_path;

public:

    PrototypeForest(Forest* forest, Stone* stone,
Tree* tree, Path* path)
    {
        prototype_forest = forest;
        prototype_stone = stone;
        prototype_tree = tree;
        prototype_path = path;
    }

    virtual Forest* MakeForest() const
    { return prototype_forest->Copy(); }
    virtual Stone * MakeStone () const
    { return prototype_stone ->Copy(); }
    virtual Tree * MakeTree () const
    { return prototype_tree ->Copy(); }
    virtual Path * MakePath () const
    { return prototype_path ->Copy(); }
};

```

Prototype

Esta implementación necesita de la implementación de operaciones de clonado en cada clase de los objetos a crear. Además se precisa de unos objetos primarios ya creados con los que crear el resto.

En el lado del cliente tendríamos al final algo tal que lo que sigue:

```

void CreateForest(PrototypeForest& prototype)
{
    for (size_t i = 0; i < number_of_trees; i++)
    {
        Tree* tree = prototype.MakeTree();
        forest->add_tree(tree);
    }

    for (size_t i = 0; i < number_of_stones; i++)
    {
        Stone* stone = prototype.MakeStone();
        forest->add_stone(stone);
    }

    for (size_t i = 0; i < number_of_paths; i++)
    {
        Path* path = prototype.MakePath();
        forest->add_path(path);
    }
}

```

Generación de un bosque mediante Prototype

III. RESULTADOS

Tras realizarse dos pruebas, se han obtenido los siguientes resultados de memoria ocupada y tiempo de ejecución.

A. Prueba 1

El objetivo de la primera prueba era crear un bosque con las siguientes especificaciones:

- 1 000 árboles
- 2 000 piedras
- 10 senderos

Obteniéndose los resultados mostrados en la siguiente tabla:

Version	Memory used (kB)	Execution time (seconds)
Standard	24 086.97	0.48
Abstract Factory	24 086.97	0.47
Builder	24 087.99	0.47
Prototype	24 104.86	0.48

Resultados de la primera prueba

B. Prueba 2

El objetivo de la segunda prueba era crear un bosque con las siguientes especificaciones:

- a) 10 000 árboles
- b) 20 000 piedras
- c) 100 senderos

Obteniéndose los resultados mostrados en la siguiente tabla:

Version	Memory used (kB)	Execution time (seconds)
Standard	240 419.55	4.74
Abstract Factory	240 419.55	4.66
Builder	240 419.55	4.67
Prototype	240 437.44	4.64

Resultados de la segunda prueba

IV. CONCLUSIONES

Los resultados son bastante estables en todas las versiones, solo siendo mayores en el caso

del prototype ya que necesita de unos objetos iniciales para poder realizar las copias.

Además, todos los objetos creados mediante el patrón Prototype son idénticos, ya que son copias o clones de uno inicial. Esto, sin embargo, aunque es algo negativo en este escenario, es una ventaja en otros posibles escenarios como en los spawners de enemigos, ya que modificando las estadísticas del prototipo se pueden crear nuevas unidades mejoradas.

Con respecto al tiempo de ejecución, se ha probado que son más rápidos los procedimientos mediante patrones de diseño.

Por último, la claridad de código y la encapsulación que ofrecen los patrones Abstract Factory y Builder (siendo más relevante la de este segundo patrón) hacen muy cómodo el uso y reutilización de los procedimientos.

V. BIBLIOGRAFÍA

Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. ISBN 978-0-201-63361-0.

Nystrom, Robert (2014). *Game Programming Patterns*. Genever Benning. ISBN 978-0990582908.