Videojuego “Sapos y Ranas” en C++

El videojuego “Sapos y Ranas” elaborado en la materia de “Programación Avanzada” de la universidad Mayor, Real y Pontificia de San Francisco Xavier, está basado en el videojuego árcade clásico “Frogger”, que consiste en llevar a una rana a través de ciertos obstáculos evitando que muera.

Para la construcción del juego se utilizó diferentes patrones listados a continuación:

* Abstract factory
* Adapter
* Bridge
* Builder
* Composite
* Factory Method
* Prototype
* Singleton
* Strategy

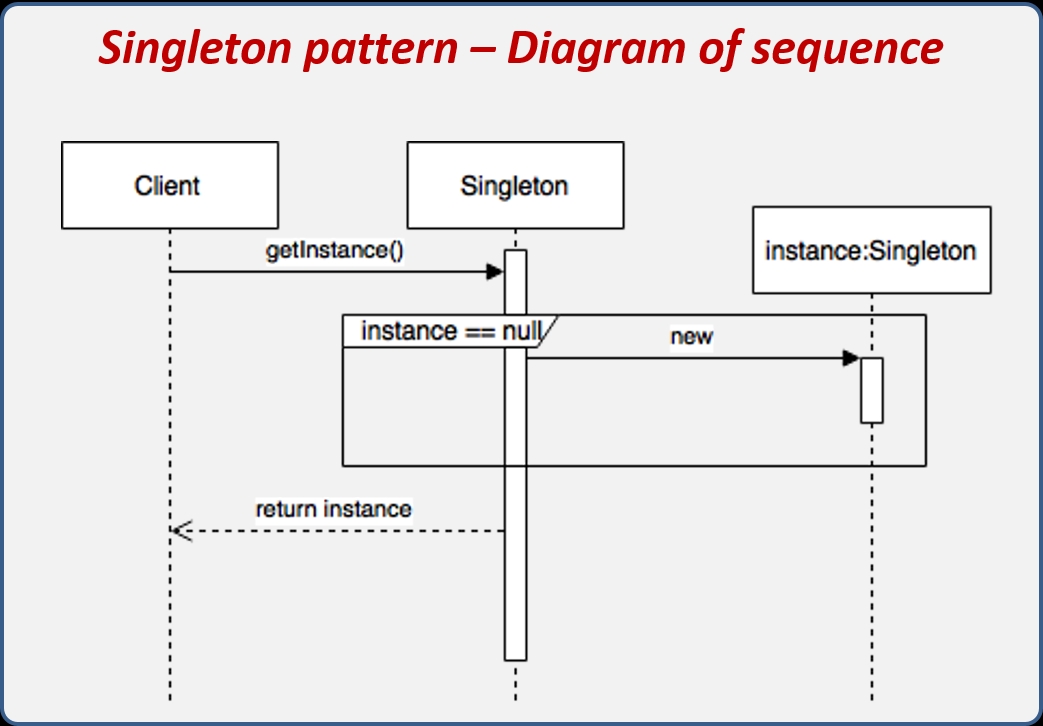
Para una mejor organización de los clases que componen el juego, se las dividió en diferentes carpetas:

* **Actores:** En esta carpeta se encuentran todo lo referente a los diferentes personajes del videojuego, su representación visual e incluido el movimiento que realizaran.
* **Escenario:** Aquí esta organizado todo lo que es referente a la visualización del escenario, los diferentes objetos que aparecerán en el (Bloques, Espinas, CubosEnergia).
* **InterfazTexto:** En esta carpeta se puede encontrar todas las clases referentes a la interfaz visual, al menú y al funcionamiento del videojuego.

# Patrón Singleton

Este patrón limita el número de instancias posibles de una clase en nuestro programa.

Su intención consiste en garantizar que una clase solo tenga una instancia y proporcionar un punto de acceso global a ella.



En el juego se implementa este patrón a la clase GameManager para limitar una solo instancia de esta clase, para ello en GameManager.h se declaran la variable instancia estática de tipo GameManager en esta variable se almacenará la instancia, declaramos constructor privado para evitar instancias y declaramos el método acceso getInstancia estática de tipo GameManager. En el GameManager.cpp implementamos el patrón donde la variable instancia igualamos a cero por que la instancia se inicializara bajo demanda y en el método getInstancia almacenamos y retornamos la instancia de la clase en la variable instancia.

El las clases NivelAvanzado.h, NivelBasico.h y NivelMedio.h llamamos al método accesor getInstancia en vez de instanciar la clase.

**GameManager.h**

class GameManager {

//simgleton de GameManager

private:

static GameManager\* instancia; //aqui es donde se almacenara la instancia

GameManager(); //constructor privado para evitar instancias

public:

static GameManager\* getInstancia(); //metodo accesor estatico de instancia

}

**GameManager.cpp**

//implementacion del singleton

GameManager\* GameManager::instancia = 0; //nulo por que la instancia se inicializara bajo demanda

GameManager\* GameManager::getInstancia() { //metodo accesor donde gurdamos la instancia no se podra instanciar mietras no se llame al metodo

if (instancia == 0) {

instancia = new GameManager();

}

return instancia;

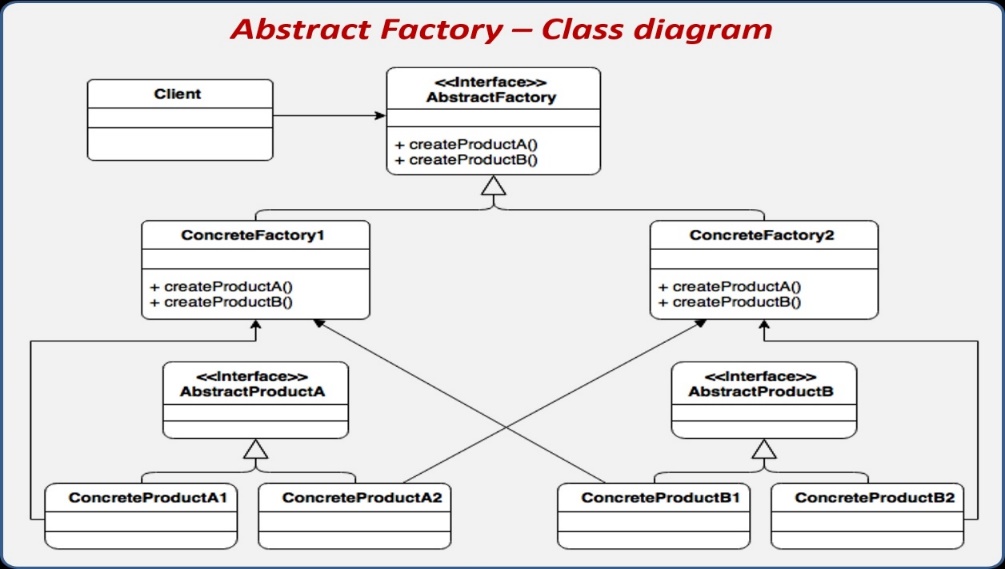
}

GameManager::GameManager() {}

Llamamos al método getInstacia en las clases NivelAvanzado, NivelBasico y NivelIntermedio

# Patrón Abstract Factory (Fabrica Abstracta)

Este patrón nos provee un método (interfaz) que delega la creación de un conjunto de objetos relacionados busca agrupar un conjunto de clases que tiene un funcionamiento en común.

En el Juego se implementa el patrón para fabricar actores, donde la clase Actor es una clase abstracta es abstracta porque tiene un método virtual int dano()=0 esta clase es la clase padre que tiene sus respectivos atributos , métodos y métodos acceso res. Donde las clases hijas Auto, Cocodrilo, Mosca, Sapo, Tortuga y Tronco heredan de clase padre Actor y el método virtual dano() se sobre escribe obligatoriamente en todas las clases hijas.

La Clase padre FabricaActor abstracta tiene atributos y los métodos virtuales de tipo Actor (ej. Virtual Actor\* crearSapo()=0) estos métodos creados sobrecargaran (se sobre escribirá) obligatoriamente las diferentes fabricas especializadas, como FabricaActorNivelAvanzado, FabricaActorNivelBasico y FabricaActorNivelMedio son clases hijas que heredan de la clase padre FabricaActor que a la ves estas clase hijas tiene su propios atributos privado.

La creación de los objetos es gracias a los métodos virtuales mencionados anteriormente, que relacionan a las tres clases que tienen funcionamiento en común.

Las clases que heredan de la clase actor son todos las actores del juego son parte de la aplicación del patrón Abstract Factory

Clase abstracta para la aplicación del patrón que contiene métodos(interfaz) para la creación de un conjunto de objetos.

**FabricaActor.h**

/\*esta es la interfaz necesaria en abstract Factory ->clase padre abstracta\*/

class FabricaActor {

public:

//constructor de la clase

FabricaActor() {

anchoTortuga = anchoCorrect(avatarTortuga);

anchoCocodrilo = anchoCorrect(avatarCocodrilo);

anchoAuto1 = anchoCorrect(avatarAuto1);

anchoAuto2 = anchoCorrect(avatarAuto2);

anchoTronco = anchoCorrect(avatarTronco);

}

//creando metodos que las fabrizas especializadas sobrecargaran obligatoriamente

virtual Actor\* CrearSapo(int \_x, int \_y, int \_dx, int \_dy, MapaElementosPantalla\* m) = 0;

virtual Actor\* CrearCocodrilo(int \_x, int \_y, int \_dx, int \_dy, MapaElementosPantalla\* m) = 0;

virtual Actor\* CrearAuto1(int \_x, int \_y, int \_dx, int \_dy, MapaElementosPantalla\* m) = 0;

virtual Actor\* CrearAuto2(int \_x, int \_y, int \_dx, int \_dy, MapaElementosPantalla\* m) = 0;

virtual Actor\* CrearTortuga(int \_x, int \_y, int \_dx, int \_dy, MapaElementosPantalla\* m) = 0;

virtual Actor\* CrearTronco(int \_x, int \_y, int \_dx, int \_dy, MapaElementosPantalla\* m) = 0;

Las clases FabricaActorNivelAvanzado, FabricaActorNivelBasico y FabricaActorNivelMedio Sobrecargan los métodos creados en FabricaActor(clase padre) para su posterior fabrica de actores

Ejemplo :

FabricaActorNivelAvanzado.h

#include "FabricaActor.h"

//clase hija que hereda de la clase padre fabricaActor relacionados que tienen un funcionamiento en común(aplicando el patron Abstrac Fatory)

class FabricaActorNivelAvanzado :public FabricaActor {

private:

//atributos privados propios de la clase estaticos

static int NumeroEnemigosA1;

static int NumeroEnemigosA2;

static int NumeroEnemigosA3;

static int NumeroEnemigosA4;

static int NumeroEnemigosA5;

public:

//sobrecarga de metodos de la clase padre obligatorio

Actor\* CrearSapo(int \_x, int \_y, int \_dx, int \_dy, MapaElementosPantalla\* m);

Actor\* CrearCocodrilo(int \_x, int \_y, int \_dx, int \_dy, MapaElementosPantalla\* m);

Actor\* CrearAuto1(int \_x, int \_y, int \_dx, int \_dy, MapaElementosPantalla\* m);

Actor\* CrearAuto2(int \_x, int \_y, int \_dx, int \_dy, MapaElementosPantalla\* m);

Actor\* CrearTortuga(int \_x, int \_y, int \_dx, int \_dy, MapaElementosPantalla\* m);

Actor\* CrearTronco(int \_x, int \_y, int \_dx, int \_dy, MapaElementosPantalla\* m);

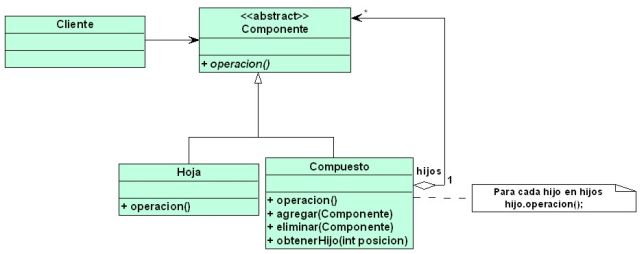
};

}

En la clase FabricaActorNivelAvanzado.cpp solo se esta instanciando los actores, y se está retornando.

# Patrón Composite

El patrón de diseño Composite nos sirve para construir estructuras complejas partiendo de otras estructuras mucho más simples, dicho de otra manera, podemos crear estructuras compuestas las cuales están conformadas por otras estructuras más pequeñas.

 Compone objetos en estructuras de árbol para jerarquías de parte-todo. Permite que los clientes traten de manera uniforme a los objetos individuales y a los compuestos.

## Participantes

* Componente: Generalmente es una interface o clase abstracta la cual tiene las operaciones mínimas que serán utilizadas, este componente deberá ser extendido por los otros dos componentes, hojas, y compuestos.
* Hoja: Representa la parte más simple o pequeña de toda la estructura.
* Compuesto: Define el comportamiento de los componentes hijo, almacena a componentes hijos, implementa las operaciones de la interfaz Componente relacionadas con los hijos

El componente en el juego es la clase ComponenteIT.h que esta heredando de ObjetoIt.h todas sus características.

En el juego se define una clase Composite.h que podrá manejar cualquier tipo de componente usando template, de esta clase podrá contener otras clases hijas de tipo componente ya definiendo sus métodos para poder agregar hijos o eliminarlos y acceder a ellos.

Como el patrón composite puede tener otras hojas tipo composite y estos mismos pueden tener otros componentes compuestos.

using namespace std;

// El template maneja cualquier tipo de clase

template<typename Componente>

// Se está creando la clase componente que desciende de un template tipo componente

class Composite : public Componente

{

protected:

//Va a almacenar hijos de tipo componente

vector<Componente\*> hijos; //Lista de los hijos componentes.

public:

// Se crea un tipo de dato que valga igual vector<Componente\*>::iterator

Typedef typename vector<Componente\*>::iterator IteradorHijo;

// Método para añadir un hijo

void AgregarHijo(Componente\* hijo)

{

// Le pasa un hijo tipo componente

// se busca de principio a fin buscando hijo

// Si lo encuentra lo guarda en itr

// Si no manda la dirección final y añade

IteradorHijo itr = find(hijos.begin(), hijos.end(), hijo);

if (itr == hijos.end())

{

hijos.push\_back(hijo);

}

}

void EliminarHijo(Componente\* hijo)

{

IteradorHijo itr = find(hijos.begin(), hijos.end(), hijo);

if (itr != hijos.end())

{

hijos.erase(itr);

}

}

void VaciarHijos()

{

hijos.clear();

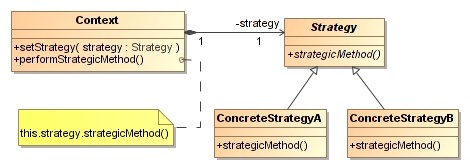
}

};

Las hojas de este Composite seria: Panel.h, Marcador.h, Botón.h, Marco.h, Etiqueta.h, Menu.h, Panel.h; cada uno de estos puede contener sus propias hojas (hijos), además que cada uno tiene sus propios atributos y métodos.

En Interfaz.cpp se añaden unos dentro de otros formando parte del Interfaz de texto.

# Patrón Strategy

Es una alternativa a la herencia múltiple. Las estrategias permiten configurar una clase con un determinado comportamiento de entre muchos posibles. Se Implementa cuando se necesitan distintas variantes de un algoritmo. En lugar de tener muchas condicionales, podemos mover las ramas de estos a sus propias estrategias.

## Participantes

* Estrategia: Declara una interfaz común a todos los algoritmos permitidos. El contexto usa esta interfaz para llamar al algoritmo definido por una Estrategia Concreta.
* Estrategia concreta
* Contexto: Se configura con un objeto estrategia Concreta. Mantiene una referencia a un objeto estrategia Puede definir una interfaz que permite a la Estrategia acceder a sus datos.

En el juego se implementa el patrón strategy con las clases: ObjetoIT.h, NivelBasico.h, NivelMedio.h, NivelAvanzado.h, ObjetoSalir.h, SelecciónObjetoIT.h.

* ***Estrategia*** (SelecciónObjetoIT.h)

Definimos la interfaz Strategy de la cual crearemos estrategias puntuales (presEnter() y fin()) para cada caso.

// clase Abstracta para seleccionar un objeto

class SeleccionObjetoIT

{

public:

virtual void presEnter() = 0;

virtual bool fin() = 0;

};

* ***Estrategia Concreta*** (NivelBasico.h, NivelMedio.h, NivelAvanzado.h, SeleccionSalir.h)

Son las implementaciones de las estrategias, cada nivel será una diferente variante de las estrategias puntuales.

Ejemplos:

// nivel de dificultad del juego basico

class NivelBasico :

public SeleccionObjetoITh

{

public:

FabricaActor\* faNivelBasico = new FabricaActorNivelBasico;

GameManager\* gm01 = GameManager::getInstancia();

void presEnter() {

gm01->IniciarJuego(faNivelBasico);

}

bool fin() {

return gm01->fin();

}

};

//--------------------------------------------------------------------------

// desciende de SeleccionObjetoIT

class ObjetoSalir :

public SeleccionObjetoIT

{

public:

bool pres = false;

void presEnter() {

pres = true;

}

bool fin() {

return pres;

}

};

* ***Contexto*** (ObjetoIT.h)

#pragma once

#include "SeleccionObjetoIT.h"

// clase abstracta de la cual heredaran sus componentes sus clases hijas

class ObjetoIT

{

// Propiedades generales que tiene un objeto

private:

int x;

int y;

int ancho;

int alto;

int colorFondoBase;

int colorCaracterBase;

char caracterBase;

bool seleccionado;

bool visible;

SeleccionObjetoIT\* seleccion;

public:

//se crean constructores según la utilidad

ObjetoIT(int \_x, int \_y, int \_ancho, int \_alto, int \_colorFondoBase, int \_colorCaracterBase, char \_caracterBase);

ObjetoIT(int \_x, int \_y);

ObjetoIT() {};

------------------------------------------

//permite definir el método seleccionado

void setSeleccionObjetoIT(SeleccionObjetoIT\* \_obj) { seleccion = \_obj; }

//permite acceder a sus datos

virtual void preEnter();

void ObjetoIT::preEnter()

{

seleccion->presEnter();

}

virtual bool fin();

bool ObjetoIT::fin()

{

return seleccion->fin()

}

----------------------------------------

virtual void Mostrar() = 0;

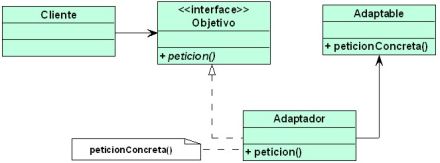
// se crean sus métodos de acceso

# Patrón Adapter

Este patrón permite a dos clases con diferentes interfaces trabajar entre ellas a través de un objeto intermedio con el que se comunican e interactúan.

Para ello, un objeto adaptador reenvía al otro objeto los datos que recibe (a través de los métodos que implementa, definidos en una clase abstracta o interface) tras manipularlos en caso necesario.

Este patrón se aplicó para el actor águila donde las clases que interactúan son **Actor, Águila y AguilaAdapter.** Las clases **Actor y Águila** trabajan entre ellas, aunque tengan interfaces diferentes donde se comunican por la clase **AguilaAdapter** trabaja como objeto intermedio donde interactúan atreves se ella.

La clase **AguilaAdapter** hace herencia de las clases padres **Actor y Aguila** donde hereda sus atributos y el método mostrar de Actor, ya que de la misma manera puede heredar de la clase Águila ya estas interactúan entre ellas.

Las clases Águila y AguilaAdapter están excluidos del proyecto por razones de que este actor no se esta incluido en el juego, pero de esa manera se implementa este patrón si se desea implementar.

**Aguila.h**

class Aguila

{

//atributos propios de la clase de tipo privado

private:

string NombreAguila;

string RepresentacionVisual;

int PosicionXI;

int PosicionXD;

int PosicionYS;

int PosicionYI;

public:

//constructor de la clase publica

Aguila(string Nombre, string RepresentacionVisual, int \_PosicionXI, int \_PosicionXD, int \_PosicionYS, int PosicionYI);

//metodos publicos declarados de la clase

void VisualizarEnPantalla();

void AtraparAnimal() {};

};

**Aguila.cpp**

Aguila::Aguila(string \_NombreAguila, string \_RepresentacionVisual, int \_PosicionXI, int \_PosicionXD, int \_PosicionYS, int \_PosicionYI) : NombreAguila(\_NombreAguila), RepresentacionVisual(\_RepresentacionVisual),

PosicionXI(\_PosicionXI), PosicionXD(\_PosicionXD),

PosicionYS(\_PosicionYS), PosicionYI(\_PosicionYI)

{

}

void Aguila::VisualizarEnPantalla()

{

gotoxy(PosicionXI, PosicionYI);

cout << RepresentacionVisual << "\nBBBBB" << "\nAAAAAA" << endl;

}

AguilaAdapter.h esta clase hace herencia de las clases Actor y Aguila donde trabajan juntas

#include "Actores.h"

#include "Aguila.h"

//clase hija que interactua como objeto intermedio para que las clases actor y Aguila trabajen entre ellas

class AguilaAdapter :

public Actores, public Aguila

{

public:

//contructor de la clase

AguilaAdapter(vector<string> \_avatar, int \_x, int \_y, int \_dx, int \_dy, int \_limiteSuperior, int \_limiteInferior, int \_limiteIzquierda, int \_limiteDerecha, MapasCaracteres\* \_mce);

void Mostrar();//metodo heredado de actor

void MostrarInformacionClase();//metodo propio de la clase

//metodo inline de representacion visual del avatar de tipo string que retorna valor del mismo tipo

inline string AvatarARepresentacionVisual(vector<string> \_avatar) {

string rvt = "";

for (int i = 0; i < \_avatar.size(); i++) {

for (int j = 0; j < \_avatar[i].length(); j++) {

rvt = rvt + \_avatar[i][j];

}

//rvt = rvt + '\n';

}

return rvt;

}

AguilaAdapter.cpp

#include "AguilaAdapter.h"

//inplementacion del constructor de la clase

AguilaAdapter::AguilaAdapter(vector<string> \_avatar, int \_x, int \_y, int \_dx, int \_dy, int \_limiteSuperior, int \_limiteInferior, int \_limiteIzquierda, int \_limiteDerecha, MapasCaracteres\* \_mce) :Aguila("NombreAguila", AvatarARepresentacionVisual(\_avatar), \_x, \_x + 5, \_y, \_y), Actores(\_avatar, \_x, \_y, \_dx, \_dy, \_limiteSuperior, \_limiteInferior, \_limiteIzquierda, \_limiteDerecha, \_mce) {

}

//inplementacion del metodo heredado de actor

void AguilaAdapter::Mostrar()

{

VisualizarEnPantalla();

}

//inplementacion del metodo propio de la clase

void AguilaAdapter::MostrarInformacionClase()

{

}

# Patrón de Creación – Factory Method

Este patrón de creación define una interfaz para crear un objeto, pero delega la creación del objeto a sus subclases. En la aplicabilidad que se le puede dar a este patrón, se la puede usar cuando no se puede prever cuantas clases de objetos serán utilizados y cuando se quiere delegar su creación a una interfaz que los vaya manejando.

Para comenzar a utilizar el patrón Factory Method, se requiere de una clase que sea el creador con una función static que se encargará de devolver el objeto que sea requerido, en este caso del videojuego “Sapos y Ranas” el creador será la clase Objetos, una clase abstracta de la cual se heredara el método Mostrar que es una función virtual pura, esta clase se define de la siguiente forma:

**Implementación del constructor y el método static (Objeto.cpp)**

//Aquí se puede observar la implementación de la función static que creara los objetos según el "tipoObjeto" que se le mande y retornara ese objeto creado con los diferentes valores que requiere el cliente"

Objetos\* Objetos::getObjeto(string \_tipoObjeto, int \_x, int \_y, int \_ancho, int \_alto)

{

if (\_tipoObjeto == "Espina")

return new Espinas(1, "espina01", \_x, \_y, \_ancho, \_alto, 'X', 0);

else if (\_tipoObjeto == "Bloque")

return new Bloques(1, "bloque01", \_x, \_y, \_ancho, \_ancho, '#', 0, 0);

else if (\_tipoObjeto == "CuboEnergia")

return new CubosEnergia(1, "cuboenergia01", \_x, \_y, \_ancho, \_alto, '@', 5, 1);

else

return NULL;

}

Como puede observarse en el código, es necesario definir e implementar una función static que será la interfaz de creación de los diferentes objetos, en base al tipo de objeto que se requiera.

Y para finalizar su implementación en el cliente (GameManager) se hace de la siguiente manera:

**GameManager.h**

//Para facilitar la visualización de este objeto se crea una lista que almacenara los diferentes objetos creados

list<Objetos\*> listaObjetos;

**Implementación del patrón – GameManager.cpp**

//Para la creacion de este objeto se puede observar como se le pasa el tipo de objeto requerido y se lo añade a la lista

listaObjetos.push\_back(Objetos::getObjeto("Espina", 5, 15, 4, 3));

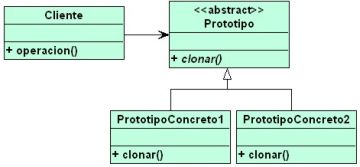
listaObjetos.push\_back(Objetos::getObjeto("Bloque", 20, 4, 10, 5));

listaObjetos.push\_back(Objetos::getObjeto("CuboEnergia", 50, 4, 1, 1));

Como se puede observar la implementación de este patrón es sencilla, pero se excluye del juego ya que los diferentes objetos que son requeridos en el juego como ser: “Espinas”, “CubosEnergia” y “Bloque”, ya han sido implementados con el patrón Builder y se crean a partir de ahí, sin embargo, este es otra opción con la cual se puede obtener el mismo resultado.

# Prototype

Permite crear objetos prediseñados sin conocer detalles de cómo crearlos. Esto lo logra especificando los prototipos de objetos a crear. El concepto de este patrón es simple: en lugar de crear un objeto, se clona, es decir, se realiza una copia exacta de otro objeto dado, el cual tiene como finalidad crear nuevos objetos duplicándolos, clonando una instancia creada previamente, lo cual permite reducir la herencia y la cantidad de clases innecesarias.

Uno de los puntos del patrón es que un programa puede dinámicamente añadir y borrar objetos prototipo en tiempo de ejecución. Esta es una ventaja que no ofrece ninguno de los otros patrones de creación, esto mismo esconde los nombres de los productos específicos al cliente.

## Implementación en el código

Para la implementación de este patrón creamos las clases, que relacionándolo con el esquema tenemos:

* Potenciadores (abstract)
* PotenciadoresConcretosTipo1 (PrototipoConcreto1)
* PotenciadoresConcretosTipo2 (PrototipoConcreto2)

Clase Potenciadores:

Para la implementación del patrón la clase debe ser una clase abstracta, el cual nos permite clonar, esta debe ser de tipo virtual puro, es decir, que no se puede declarar en la clase Potenciadores, pero sí de forma obligada en sus clases hijas.

class Potenciadores :

public Objetos

{

private:

string tipo;

int velocidad;

int salto;

int energia;

int escudo;

int vidas;

public:

//Constructor

Potenciadores(int \_x, int \_y, int \_ancho, int \_alto, char \_caracter, int \_colorFondo, int \_colorCaracter, MapaElementosPantalla\* \_mc,

int \_velocidad, int \_salto, int \_energia, int \_escudo, int \_vidas);

//acorde al patron prototype, debemos crear un metodo de tipo virtual y abstracto para poder clorar

virtual Potenciadores\* Clonar() = 0;

void Mostrar();}

Dentro de las clases Hijas:

Dentro del constructor luego de definir los parámetros concretos que se está utilizando para una determinada clase, definimos un método asesor” setTipo” que permita determinar el tipo concreto que es, esta puede ser un entero, string, etc.

Dentro del método Clonar, retorna un nuevo objeto de la clase donde está definido el método clon, a esta se le pasa un puntero que hace referencia a todos los parámetros que recibe en la clase.

PotenciadoresContretosTipo1::PotenciadoresContretosTipo1(int \_x, int \_y, int \_ancho, int \_alto, char \_caracter, int \_colorFondo, int \_colorCaracter, MapaElementosPantalla\* \_mc,int \_velocidad, int \_salto, int \_energia, int \_escudo, int \_vidas) :Potenciadores(\_x, \_y, \_ancho, \_alto, \_caracter, \_colorFondo, \_colorCaracter, \_mc, \_velocidad, \_salto, \_energia, \_escudo, \_vidas)

{

setTipo("PotenciadorTipo1");

}

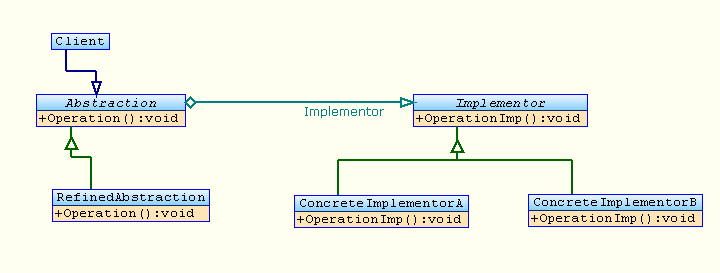
Potenciadores\* PotenciadoresContretosTipo1::Clonar()

{

return new PotenciadoresContretosTipo1(\*this);

}

# Bridge

* “También conocido como Handle/Body, es una técnica usada en programación para desacoplar una abstracción de su implementación, no une permanentemente una implementación a una interfaz, sino que la implementación puede configurarse en tiempo de ejecución de manera que ambas puedan ser modificadas independientemente sin necesidad de alterar por ello la otra”.
* Logra minimizar el acoplamiento, hacer que las clases dependan de abstracciones en lugar de depender de implementaciones, prefiere el uso de composición antes que el uso de herencia.
* Mejora la extensibilidad, podemos extender las jerarquías de Abstracción y de Implementador de forma independiente.
* Oculta detalles de implementación a los clientes, se puede aislar a los clientes de los detalles de implementación, como el compartimiento de objetos implementadores y el correspondiente mecanismo de conteo de referencias.

## Implementación en el código

Para la implementación del patrón, utilizamos las siguientes clases:

* ActorMovimiento (Clase Padre)
* ActorMovimientoAleatorio(Clase Hija)
* ActorMovimientoNormal(Clase Hija)

Clase Padre:

Dentro de la clase padre, creamos un método el cual sea virtual puro “Mostrar Mover” el cual obligatoriamente debe implementarse en las clases hijas.

Clases Hijas:

Dentro de las clases hijas, implementamos las clases Mostrar y Mover, nótese que en cada clase permite mover y mostrar al actor pero de diferente manera, esto es lo que permite el patrón bridge, el cual logra que cada hija logre implementar el mismo método pero de diferente manera.

### En La Clase Actormovimientoaleatorio:

void ActorMovimientoAleatorio::Mostrar(){//aquí dentro se implementa un método de mostrar movimiento aleatorio}

### En la Clase ActorMovimientoNormal:

void ActorMovimientoNormal::Mostrar() {//aquí dentro se implementa un método de mostrar el movimiento normal

//}

Ahora bien ¿Cómo hacemos para que las clases sepan que movimiento deben implementar? En este caso creamos el constructor de la clase por defecto y dentro de ella creamos una variable “movimientoActor” la cual permite crear un nuevo objeto, y le digo que tipo de comportamiento o implementación le voy a asignar, permitiendo así conocer cómo se va amostrar y mover.

en la clase auto, toma el movimiento normal

Auto::Auto(vector<string> \_avatar, int \_x, int \_y, int \_dx, int \_dy, int \_colorCaracter, int \_colorFondo, MapaElementosPantalla\* m)

:Actor(\_avatar, \_x, \_y, \_dx, \_dy, \_colorCaracter, \_colorFondo, m) {

movimientoActor = new ActorMovimientoNormal(&\_avatar, &\_x, &\_y, &\_dx, &\_dy, &\_colorCaracter, &\_colorFondo, m);

}

Sin embargo, para la clase Mosca toma un movimiento Aleatorio Mosca::Mosca(vector<string> \_avatar, int \_x, int \_y, int \_dx, int \_dy, int \_colorCaracter, int \_colorFondo, MapaElementosPantalla\* m) :

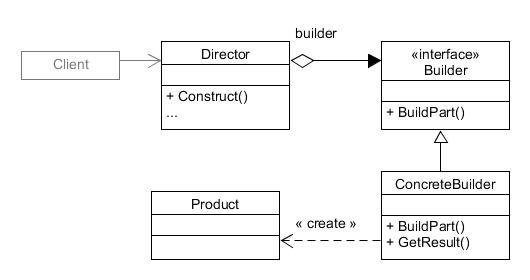
Actor(\_avatar, \_x, \_y, \_dx, \_dy, \_colorCaracter, \_colorFondo, m) {

movimientoActor = new ActorMovimientoAleatorio(&\_avatar, &\_x, &\_y, &\_dx, &\_dy, &\_colorCaracter, &\_colorFondo, m);

}

# Patrón de Creación - Builder

Este patrón tiene la función que tiene este patrón es la construcción de objetos complejos a partir de la abstracción de los elementos que lo compondrán. Como ejemplo consideremos que se quiere construir un auto, para la construcción de este objeto complejo es necesaria sus partes como podrían ser: motor, neumáticos, carrocería, batería. Al igual con este patrón se puede crear diferentes representaciones de este objeto con solo algunas partes de la cual lo componen.



## Esquema Base del Patrón

En el videojuego “Sapos y ranas”, este patrón se implementó en la construcción de las diferentes franjas que hay en el escenario puesto en el juego, como los que serían “FranjaInicioBuilder”, “FranjaCalleBuilder”, “FranjaMedioBuilder”, “FranjaCasasBuilder” y “FranjaRioBuilder”.

Primero necesitamos una clase que sea el producto final de todo el patrón, esta vendría a ser “FranjasEscenario” con los diferentes componentes que se utilizaran en el objeto a construir.

### FranjasEscenario.h

class FranjasEscenario

{

private:

Panel\* panelFE;

vector<Bloques\*> bloquesFE;

vector<Espinas\*> espinasFE;

public:

void Mostrar();

Panel\* getPanelFE() { return panelFE; }

vector<Bloques\*> getBloquesFE() { return bloquesFE; }

vector<Espinas\*> getEspinasFE() { return espinasFE; }

void setPanelFE(Panel\* \_panelFE) { panelFE = \_panelFE; }

void setBloquesFE(vector<Bloques\*> \_bloquesFE) { bloquesFE = \_bloquesFE; }

void setEspinasFE(vector<Espinas\*> \_espinasFE) { espinasFE = \_espinasFE; }

};

### Implementación de (FranjasEscenario.cpp)

#include "FranjasEscenario.h"

void FranjasEscenario::Mostrar() {

//Metodo para mostrar el panel

panelFE->Mostrar();

//Metodos para mostrar los objetos almacenados en los vectores

for (auto it = std::begin(bloquesFE); it != std::end(bloquesFE); ++it) {

(\*it)->Mostrar();

}

for (auto it = std::begin(espinasFE); it != std::end(espinasFE); ++it) {

(\*it)->Mostrar();

}

}

A continuación, se crea un constructor, en la cual se define la interfaz para crear las partes de un objeto Producto. En el videojuego este constructor es “BuilderFranjasEscenario” una clase abstracta, en el cual se implementará los métodos virtuales puros que se heredaran a los constructores concretos:

// clase abstracta pura

class BuilderFranjasEscenario

{

public:

// Cambiamos la clase superficie por la clase Panel y usamos MapaElementosPantalla para guardar caracter, color y posiciones

virtual Panel\* getPanelFranjaEscenario(MapaElementosPantalla\*) = 0;

virtual vector<Bloques\*> getBloquesFranjaEscenario(MapaElementosPantalla\*) = 0;

virtual vector<Espinas\*> getEspinasFranjaEscenario(MapaElementosPantalla\*) = 0;

};

Aquí se definen las partes que se usaran en la composición del objeto (Panel\*, vector<Bloques\*>, vector<Espinas\*>), todos definidos con punteros.

Para el siguiente paso necesitamos un **Director** el cual construirá el objeto a partir de la interfaz del constructor, en el videojuego esta definido como “DirectorFranjasEscenario”.

### Definición del director (DirectorFranjasEscenario.h)

class DirectorFranjasEscenario{

private:

BuilderFranjasEscenario\* builder;

public:

void setBuilder(BuilderFranjasEscenario\* \_builder) { builder = \_builder; }

//Método que devolverá la creación del objeto con los objetos que fueron pedidos por el cliente FranjasEscenario\* getFranjaEscenario(MapaElementosPantalla\*\_mce);

};

### Implementación de los métodos (DirectorFranjasEscenario.cpp)

// El director te crea una franja con los objetos que tu le dices y te retorna un franja

FranjasEscenario\* DirectorFranjasEscenario::getFranjaEscenario(MapaElementosPantalla\* \_mce)

{

FranjasEscenario\* franjaEscenario = new FranjasEscenario;

franjaEscenario->setPanelFE(builder->getPanelFranjaEscenario(\_mce));

franjaEscenario->setBloquesFE(builder->getBloquesFranjaEscenario(\_mce));

franjaEscenario->setEspinasFE(builder->getEspinasFranjaEscenario(\_mce));

return franjaEscenario;

}

Para el siguiente paso será necesaria la creacion de los constructores concretos, los cuales los definimos como “FranjaInicioBuilder”, “FranjaCalleBuilder”, “FranjaMedioBuilder”, “FranjaCasasBuilder” y “FranjaRioBuilder”.

Estos constructores concretos se crean a partir del **builder** que en nuestro videojuego seria “BuilderFranjasEscenario”, de este se hereda los métodos de creación de los diferentes componentes para asi armar el **Constructor Concreto**.

### Ejemplo: FranjaInicioBuilder.h

class FranjaInicioBuilder :

public BuilderFranjasEscenario

{

Panel\* getPanelFranjaEscenario(MapaElementosPantalla\* \_mce);

vector<Bloques\*> getBloquesFranjaEscenario(MapaElementosPantalla\*);

vector<Espinas\*> getEspinasFranjaEscenario(MapaElementosPantalla\*);

};

**Implementación de los métodos según las partes que requiera el usuario (FranjasInicioBuilder.cpp)**

En la fraja de inicio lo que se necesitaría seria un panel, 3 “Bloques” almacenados en un vector y 3 objetos “Espinas” igualmente almacenados en un vector.

#include "FranjaInicioBuilder.h"

Panel\* FranjaInicioBuilder::getPanelFranjaEscenario(MapaElementosPantalla\* \_mce) {

Panel\* panelFI = new Panel(1, 26, 118, 3, 5, 6, '\*', 120, ' ', 0, \_mce);

return panelFI;

}

vector<Bloques\*> FranjaInicioBuilder::getBloquesFranjaEscenario(MapaElementosPantalla\* m)

{

vector<Bloques\*> bloquesFI;

bloquesFI.push\_back(new Bloques(1, 26, 20, 2, 'X', 5, 2, m));

bloquesFI.push\_back(new Bloques(50, 26, 10, 2, 'B', 5, 2, m));

bloquesFI.push\_back(new Bloques(99, 26, 20, 2, 'X', 5, 2, m));

return bloquesFI;

}

}

Para finalizar el ejemplo se lo implementa en el cliente (GameManager) para su creación y visualización en el juego. Se define la franja de inicio como un puntero a la clase del producto final (FranjasEscenario) y se declara igualmente el constructor concreto (FranjaInicioBuilder) que es necesario para que el director cree el objeto en base a ello.

**GameManager.h**

FranjasEscenario\* FEInicio;

FranjaInicioBuilder franjaInicioBuilder;

**GameManager.cpp**

//pasando el tipo de franja al director de BUILDER

DirectorFE.setBuilder(&franjaInicioBuilder);

// Franja Inicio

FEInicio = DirectorFE.getFranjaEscenario(mapaCaracteresEscenario);

Este patrón es muy utilizado como se menciono antes para la construcción de un objeto en base a sus partes e incluso para crear diferentes representaciones como se hizo en el juego (“FranjaInicioBuilder” es muy diferente a “FranjaRioBuilder”).