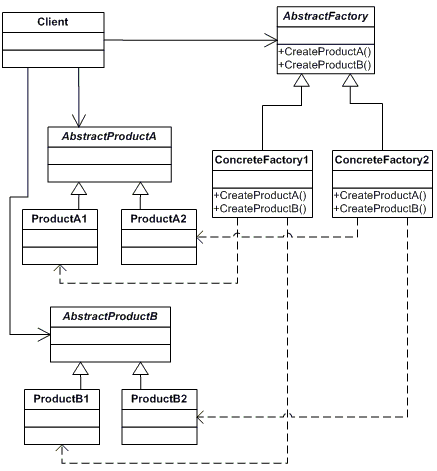
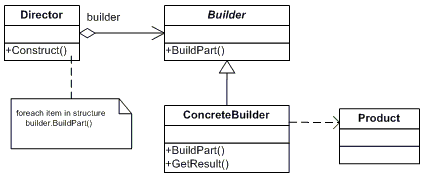
**Creational Patterns**

**1). Abstract Factory**



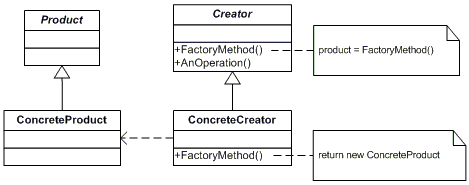
En el diagrama anterior el cliente es un elemento que interactuara haciendo uso de las demás clases, abtract factory es el encargado de definir como interface, los métodos por los cuales las concretefactorys crean objetos de porductos y el abstract product se encarga de definir como interface, un objeto de producto que creara cada concrete Factory correspondiente, es decir que las abtractproduct A y B definen correspondientemente a los product A1,A2 y B1,B2 correspondientemente.

**2). Builder**



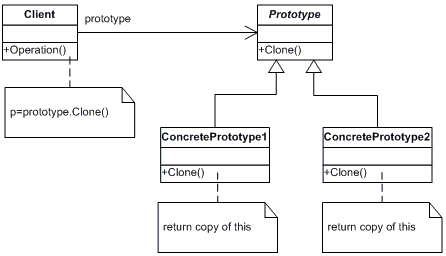
El director manda la orden de crear un objeto, Builder a su vez genera una interface que concrete builder debe aplicar y este último se encarga de construir un objeto producto que implementa la interface y guarda el resultado de la creación.

**3). Factory Method**



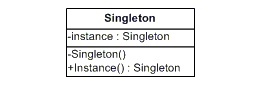
Product es abstracto y define una interface que concreteproduct debe implementar, creator define el factory method y cualquier operación que usa concretecreator el cual le devuelve un nuevo objeto tipo product con las especificación echas en creator.

**4). Prototype**



Client manda la orden de creación de prototype y este a su vez crea a concreteprototype 1 y 2 clonándose a sí mismo.

**5). Singleton**



Solo existe una clase con una única instancia y un solo punto de acceso que es el singleton.

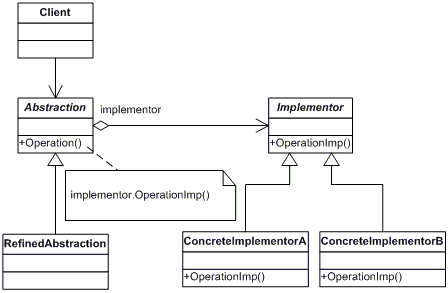
**Behavioral Patterns**

**6). Adapter**



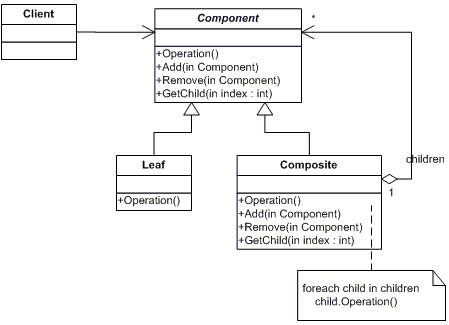
El client busca utilizar el target, adapter implementa la petición hecha por target y adaptee sobrescribe la petición a una más específica.

**7). Bridge**



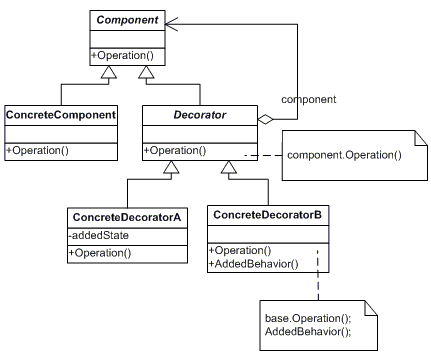
Client hace uso de abstracción, abstracción define la interface de los implementors que hacen uso de la operación, implementor define la interface para cada implementor concreto de la operación y refinedabtraccion extiende la interface.

**8). Composite**



El cliente manipula los objetos creados en composite, component declara una interface eh implementa los comportamientos comunes a las demás clases, composite guarda los childs creados y ejecuta operaciones con ellos según component lo defina.

**9). Decorator**



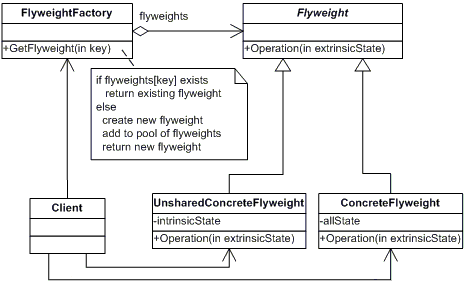
Component se encarga de definr la interface de los objetos creados, decorator mantiene la referencia de component y define el comportamiento de su operación que se aplica a todos los objetos component, concrete decorator A esta alamacenando un estado de la operación demarcada en el Decorator y concrete decorator añade otro comportamiento al decorator, por ultimo concrete component define un objeto que hereda todas las responsabilidades.

**10). Facade**



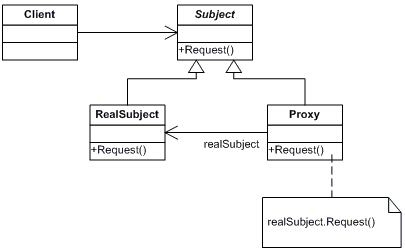
Facade define una interface para un subsistema de interfaces, es decir define una interface de uso general que hace más fácil el uso del subsistema

**11). Flywheight**



Flyweightfactory devuelve objetos flywheight existentes si los tiene o si no los crea, flywheight define la interface de los objetos que hereden su comportamiento, concreteflyweight conserva un estado de sí mismo que puede ser compartido mientras que el unchared define un estado para sí mismo que no necesita ser compartido, client mantiene las relaciones

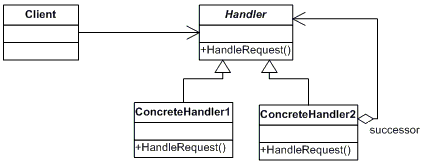
**12). Proxy**



Client mantiene una relación con subjec, subject define la interface que tanto proxy como realsubject siguen, proxy lanza la petición a ejecutar con realsubject

**Behavioral Patterns**

**13). Chain of Responsability**



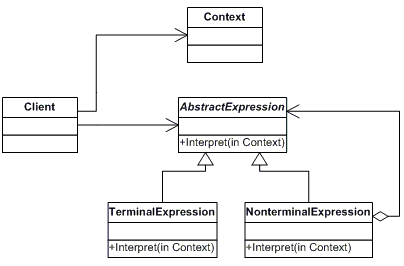
Client manda la petición a handler, handler define la interface de los objetos tipo handler que intentaran recibir la petición y si cada concretehandler intenta tomar la petición.

**14). Command**



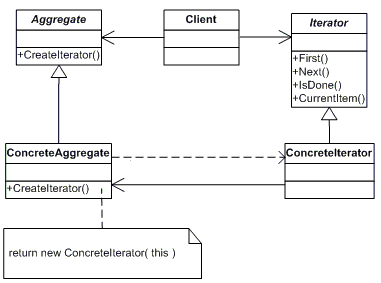
Client crea al concretecommand y setea su recibidor, comand a su vez plantea la interface a ejecutar por concretecomand, invoker pide a coman que cargue con la petición y recivier lleva a la actividad a ejecutar con la petición recibida.

**15). Interpreter**



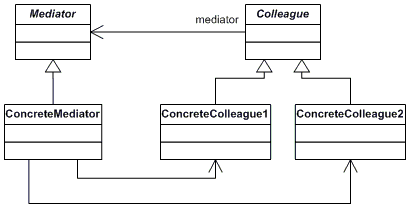
Client crea context y plantea abstract exprecion que genera un interface para terminal y nonterminal de manera que los expression puedan interpretar el contexto.

**16). Iterator**



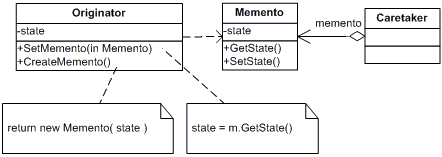
Client mantiene conexión al la misma velocidad con agragate como con iterato, agregate Define la interface para la creación de objetos tipo iterator e iterator define la interface para el acceso a los objetos creados, concrete aggregate implementa la interface y retorna los objetos tipo concreteiterator y concrete iterator mantiene la posición de los objetos creados.

**17). Mediator**



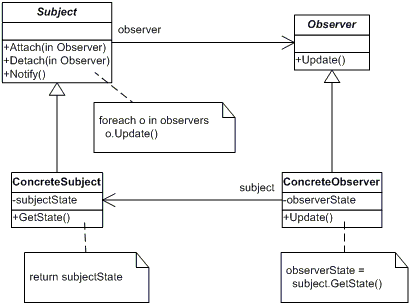
Mediator plentea la interface de los concrete mediator plantando loas formas por las cuales se van a comunicar los objetos colleague, colleague define la interface que implementan los concrete colleague y los concrete colleague implementan la interface conociendo como comunicarse con el mediator.

**18). Memento**



Originator se encarga de crear el memnto con un estado definido, memento guarda este estado para sí mismo y permite devolver el estado del originator al inicial, caretaker se enfoca en la protección del memnto teniendo muy poco acceso a su contenido.

**19). Observer**



Subject define una interface para que objetos tipo observer puedan estar ligados a el o desligados si es requerido y plantea el método por el cual los observer son notificados, concretesubject implementa la interface guardando y updateando el estado del subject, opserver plantea la interface de los objetos tipo observer y plantea su método para actualizarse y concrete observer guarda el estado actualizado por subject.

**20). State**



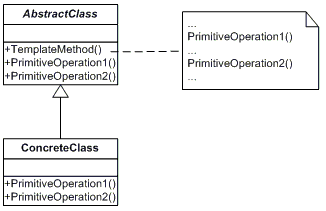
Context plantea request con un estado definido, state define la interface que maneja un comportamiento según el estado de context actual y las concrete state implementan comportamientos asociados al estado actual.

**21). Strategy**



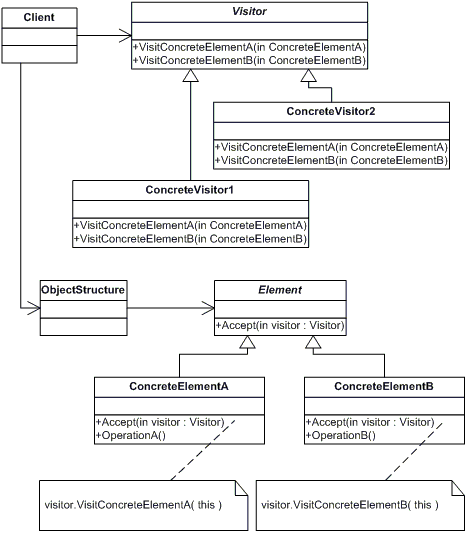
Strategy define la interface de los algoritmos soportados por el programa, context está ligado a strategy y maneja las conexiones con los concrete strategy los culés implementan la interface de strategy y definen los algoritmos a ser implementados.

**22). Template Method**



Define una interface genérica para su implementación en las concrete clases que se requieran o sean necesarias, concrete class implementa la interface definiendo las operations específicas.

**23). Visitor**



Visitor plantea la interface de realcion entre los visitor y los elemt parte en los concrete visitor y la contraparte de visita en los element, los concretevisitor implementan la interface y guardan los algoritmos y estados de su visita a los element, Element define la interface que plante la operación en la cual acepta al visitor como argumento, concrete element implementa la operación de visita y objecstructure puede enumerar sus elementos.

3. ejemplos de patrones en videojuegos

Creacional: Singleton

public sealed class GameZombies  
{  
    // Singleton instance  
    private static GameZombies \_gameZombies;  
  
    // Privately stored list of all zombies  
    private List() \_zombies;  
  
    // Public property to access zombie list  
    public List() Zombies { get { return \_zombies; } }  
  
    private GameZombies()  
    {  
  
    }  
  
    // Public property to get the single instance of this class  
    public GameZombies Instance  
    {  
        get {  
            if (\_gameZombies == null) {  
                \_gameZombies = new GameZombies();  
            }  
            return \_gameZombies;  
        };  
    }  
  
    // Adds a new zombie to the list of game zombies  
    public void Add(Zombie zomb)  
    {  
        \_zombies.Add(zomb);  
    }  
}

public class DeadRisingGame  
{  
    static void Main(string[] args)  
    {  
        //Create 3 new zombies  
        GameZombies.Instance.Add(new PlaidZombie());  
        GameZombies.Instance.Add(new WomanZombie());  
        GameZombies.Instance.Add(new BrownShirtZombie());  
  
        //Print some statements  
        foreach(Zombie zomb in GameZombies.Instance.Zombies)  
        {  
            Console.WriteLine("Type: " + zomb.ZombieType);  
            Console.WriteLine("Movement Type: " + zomb.MovementType());  
        }  
    }  
}

OUTPUT:  
Type: Plaid-Zombie  
Movement Type: Walk slowly  
Type: Woman-Zombie  
Movement Type: Stumble around  
Type: Brown-Shirt-Zombie  
Movement Type: Yell and run

<http://www.gamasutra.com/blogs/MattChristian/20101013/88205/OOPsie_Patterns_The_Singleton_Pattern.php>

Estructural: proxy

#include "crossbowproxy.h"

int main()

{

cout << endl;

cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl;

cout << "\*\* Game Patterns \*\*" << endl;

cout << "\*\* Proxy \*\*" << endl;

cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl;

cout << "\*\* Christoph Stallmann \*\*" << endl;

cout << "\*\* University of Pretoria \*\*" << endl;

cout << "\*\* COS121 - 2012 \*\*" << endl;

cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl;

cout << endl;

Player player;

Weapon \*weapon = new CrossbowProxy(&player);

do

{

player.increaseLevel();

cout << "Player level " << player.level() << ": ";

}

while(!weapon->buy());

delete weapon;

cout << endl;

return 0;

}

#include "crossbowproxy.h"

CrossbowProxy::CrossbowProxy(Player \*player)

{

mPlayer = player;

mSubject = NULL;

}

CrossbowProxy::~CrossbowProxy()

{

if(mSubject != NULL)

{

delete mSubject;

}

}

bool CrossbowProxy::buy()

{

if(mPlayer->level() >= 5)

{

if(mSubject == NULL)

{

mSubject = new Crossbow();

}

return mSubject->buy();

}

else

{

cout << "Crossbow is unavailable." << endl;

return false;

}

}

#include "crossbow.h"

bool Crossbow::buy()

{

cout << "Buying a crossbow." << endl;

return true;

}

<https://www.youtube.com/watch?v=whJi3fhwbW0>

Nota: Código en la descripción

Comportamental: Observer

class Observer

{

public:

virtual ~Observer() {}

virtual void onNotify(const Entity& entity, Event event) = 0;

};

class Achievements : public Observer

{

public:

virtual void onNotify(const Entity& entity, Event event)

{

switch (event)

{

case EVENT\_ENTITY\_FELL:

if (entity.isHero() && heroIsOnBridge\_)

{

unlock(ACHIEVEMENT\_FELL\_OFF\_BRIDGE);

}

break;

// Handle other events, and update heroIsOnBridge\_...

}

}

private:

void unlock(Achievement achievement)

{

// Unlock if not already unlocked...

}

bool heroIsOnBridge\_;

};

<http://gameprogrammingpatterns.com/observer.html>