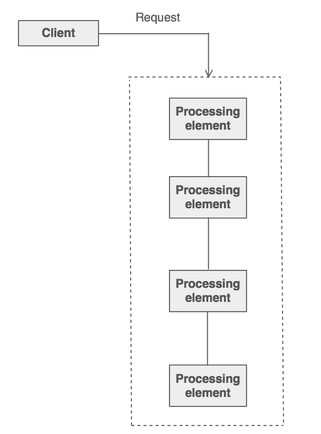
**Chain of responsibility**

Всеки един обект, измежду обекти от един тип (произхождащи от един и същ абстрактен клас), съдържа в себе си логика, която определя какви точно действия може да изпълни обектът. Ако изискващото се действие не може да бъде изпълнено от обекта, то тогава той го предава на стоящия над него в ейрархията обект(следващия по отговорност/важност/компетенция/възможности). Същото важи и за следващия обект - ако може изпълнява командата, ако не я предава на следващия. Това продължава докато командата бива изпълнена или докато се достигне до последния (стоящия най-високо в йерархията) обект. Ако той не може да изпълни командата не вика никой, а изпълнява определен за случая метод/действие.

Самата йерархия не се изгражда от наследени последователно класове (като например Person, който бива наследен от Seller, който пък е наследен от Distributor, а той е родител на RegionalDistributor), а става посредством пропърти идващо от абстрактния клас, който записва в поле, кой е следващия елемент. Така в горния пример (с продавачите), за да се изгради такава йерархия (всеки да знае за стоящия над него) трябва просто след създаването на обектите да се запише, чрез полето, кой е следващия в йерархията. В крайна сметка се получава нещо като свързан списък, където всеки обект знае кой е след него, но не знае за останалите.

Ако не съм го обяснил добре може би примера с малко код ще помогне.

Нека имаме енумерация, която представлява тип трудност на зададена задача

enum Task { Easy , Moderate, Hard, TeamProject }

Имаме базовия клас Developer, който е абстрактен и може да върши задачи (Task). Освен всичко друго, което може би съдържа(като име, години, образование, умения и каквото друго се сетите), това което е нужно, за да се направи връзка към стоящия над него е следното - метод, който да записва следващия в йерархията, пропърти, което да го пази и метод, който да се извиква, за да се провери дали обектът е способен да свърши тази задача.

public abstract class Developer

{

......

protected Developer Successor {get;set;}

public void SetSuccessor (Developer successor)

{

this.Successor = successor;

}

public abstract void HandleTask (Task task);

......

}

Сега си представете 3 класа, които наследяват Developer

public class JuniorDeveloper : Developer

{

public override void HandleTask (Task task)

{

if((int)task < 2)

{

DoYourJob();

}

else

{

this.Successor.HandleTask(task);

}

}

}

public class RegularDeveloper : Developer

{

public override void HandleTask (Task task)

{

if((int)task < 3)

{

DoYourJob();

}

else

{

this.Successor.HandleTask(task);

}

}

}

public class SeniorDeveloper : Developer

{

public override void HandleTask (Task task)

{

if((int)task < 4)

{

DoYourJob();

}

else

{

GatherTeamToSolveTask(task);

}

}

}

Вижда се, че всеки един клас може да работи само по определен тип задача и ако задачата е над възможностите му го прехвърля на този, който е над него в йерархията.

Самата йерархия става при създаването на обектите

JuniorDeveloper gosho = new JunioDeveloper();

RegularDeveloper pesho = new RegularDeveloper();

SeniorDeveloper kolio = new SeniorDeveloper();

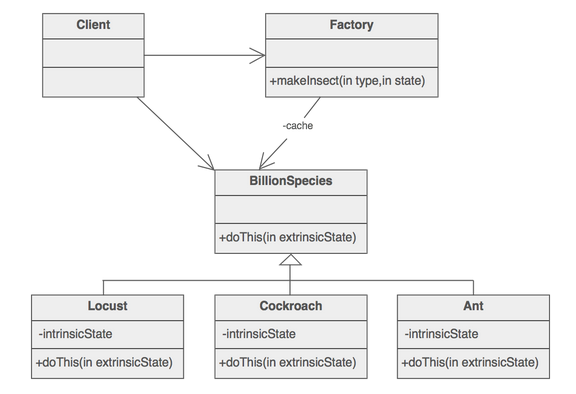
gosho.SetAccessor(pesho);

pesho.SetAccessor(kolio);

По този начин вече всеки обект знае кой е след в йерархията. В така написания пример, ако се каже на обекта gosho gosho.HandleTask(Task.Hard), gosho ще види, че Task е извън неговите възможности и ще я подаде на pesho. Той също ще види, че не може да се справи и ще я подаде на kolio, който ще работи по Task.

Подобни патърни не мога да сетя.

**Flyweight**



Това, което прави Flyweight е да преизползва вече създадени обекти, когато се налага да създадем много на брой еднакви обекти. Накратко този патърн създава обект, който се съхранява в паметта и когато трябва да се създаде нов такъв, със същите пропъртита, се подава само референцията към вече създадения обект. Така например може да имаме 100 еднакви обекта като реално сме използвали памет само за един.

За да бъде този патърн ефективен (приемаме, че се е стигнало до извода, че патърна е нужен) обектите трябва да се създават през Factory, а не с ключовата дума new. Това, с което ни помага фрабриката, е че в себе си пази речник или хеш-таблица, където записва вече създадените обекти. Ако обектът, който искаме да създаден вече съществува, не се създава нов обект, а се подава този, който вече е записан.

Следващият пример ще покаже как работи Flyweight

Имаме абстрактен клас, който извършва някаква операция по подадено число

class Flyweight

{

public abstract void Operation(int someNumber);

}

Самата фабрика изглежда по начина описан по-долу. При създаването на нова фабрика автоматично се съзадава хеш-таблица с обектите, които знаем, че ще ползваме. Разбира се може да се направи и по-ефективно (това е само един от възможните примери, погледнете демото за Flyweight от лекцията, за да видите пример с речник).

class FlyweightFactory

{

private Hashtable flyweights = new Hashtable();

public FlyweightFactory()

{

flyweights.Add("X", new Flyweight ());

flyweights.Add("Y", new Flyweight ());

flyweights.Add("Z", new Flyweight ());

}

public Flyweight GetFlyweight(string key)

{

return((Flyweight)flyweights[key]);

}

}

И самата работа с Flyweight изглежда по следния начин

static void Main()

{

FlyweightFactory factory = new FlyweightFactory();

Flyweight fx = factory.GetFlyweight("X");

Flyweight fy = factory.GetFlyweight("Y");

Flyweight fz = factory.GetFlyweight("Z");

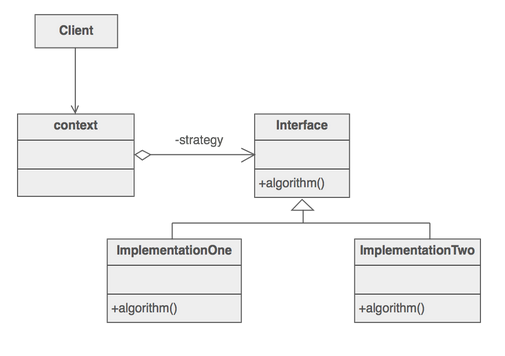
}

Ако поискаме да създаем 50 други инстанции на ‘Z’, когато извикаме нов ‘Z’ обект от фабриката, тя ще ни върне отново този, който вече е бил направен при съзадаването на фабриката.

Ако се нуждаем от ‘Z’ обекти, които имат различни свойства (размер на шрифта, цвят, шрифт, бордър) и се опитваме да променим обекта, който сме получили от фабриката, ще имаме накрая отново всички обекти с едни и същи свойства (тъй като това е 1 обект), който ще изглежда така както е след последната направена промяна. Всички сочат към една и съща референция.

Искам да подчертая, че това не е най-добрата имплементация на Flyweight, а просто за да се добие представа за патърна. По-правилния начин е да се създаде речник вместо хеш-таблица. При всяко създаване на обект през фабриката се прави проверка дали обектът вече не е бил създаван т.е. дали не е в речника. Ако не е тогава се създава нов обект и се добавя в речника. Ако вече е създаден просто се връща вече създадения.

**Strategy**

Strategy позволява да бъде скрита, от клиента, имплементацията на различни алгоритми, които в същото време да бъдат взаимно заменяеми. Т.е. представете си, че трябва да сготвите месо да речем (не смейте на примера ми). Месо е доста абстрактно понятие нали ? В зависимост какво е месото, ще го сготвите по различен начин. Свинско на скара, пилешко на тиган, телешко на фурна. Представете си че готвенето е абстрактния ни клас, който в себе ще включва ‚готви пилешко‘, ‚готви свинско‘, ‚готви телешко‘. Това, което ще каже клиента е просто сготви ми месото и в зависимост какво е то ще бъде сготвено по различен начин.

Малко код за обяснение (кодът не е работещ и оптимизиран просто е за обяснение)

Имаме готвач, който има определена техника на готвене. Имаме метод CookMeat(), който готви месото според техниката за готвене.

class Chef

{

private Cook cookTechnique;

public Chef()

{

}

public CookTehchnique

{

set

{

this.cookTechnique = value;

}

}

public void CookMeat(Meat meat)

{

this.cookTechnique.CookTheMeat(meat);

}

}

Имаме абстрактен клас Cook, който ще бъде базов за различните техники за готвене. В себе си има базовия метод CookTheMeat(), който всеки наследник ще имплементира по свой си начин.

abstract class Cook

{

public abstract void CookTheMeat(Meat meat)

{

}

}

Следват 3 класа имплементиращи базовия и съоветно метода CookTheMeat() по свой собствен начин.

class CookChicken : Cook

{

public override void CookTheMeat(Meat meat)

{

Console.WriteLine("Chicken cooked")

}

}

class CookPork : Cook

{

public override void CookTheMeat(Meat meat)

{

Console.WriteLine("Pork cooked")

}

}

class CookBeef : Cook

{

public override void CookTheMeat(Meat meat)

{

Console.WriteLine("Beef cooked")

}

}

static void Main()

{

Chef chef = new Chef());

chef.cookTechnique = new CookBeef();

chef.CoockMeat(Meat.Beef);

chef.cookTechnique = new CookPork();

chef.CoockMeat(Meat.Pork);

chef.cookTechnique = new CookChicken();

chef.CoockMeat(Meat.Chicken);

}

Примера показва как готвача може да готви различни меса като използва един и същ метод. Въпреки това месото ще бъде сготвено по различни начини.

Друг пример е Array.Sort() метода, който използва различен алгоритъм в зависимост от това с каква дължина е масивът, който трябва да се сортира.