FLYWEIGHT:

# Problem & situation

* **Situation:**

When you need to create a greater number of objects (>100 000) => the Using of memory is not efficient if we allocate a new memory block for each object being created.

It is necessary to have a way to reduce redundant memory

* **Problem:**

Imagine you are an owner of a gun shop, you have to custom your gun as a request of customer. Your gun’s attributes are number of bullets and type of the gun. The bullet’s attributes are speed and damage. Many guns use a same type of bullet so we only need to design each bullet type once. Besides, the guns are different depend on customer request.

# Solution (without using Flyweight pattern):

If you implement those ideas in traditional way, you have to create new object of bullet for each gun which is redundant and not very efficient when it come to producing a large number of guns while the bullet type can be share between guns.

Allocate new bullet object whenever we create a gun.

For example:

#include<iostream>

#include<vector>

#include<string>

using namespace std;

#define small 0

#define medium 1

#define large 2

#define Ak47 0

#define M4a1 1

class Bullet

{

private:

int speed;

int damage;

Bullet() {};

public:

static int count;

Bullet(int speed, int damage)

{

++count;

this->speed = speed;

this->damage = damage;

};

int getSpeed() { return speed; };

int getDamage() { return damage; };

};

int Bullet::count = 0;

class Gun

{

public:

virtual void setNumberOfBullets(int size) {};

virtual int getType() { return 0; };

virtual void fire() {};

virtual Bullet\* getBullet() { return nullptr; };

};

class Ak47Gun :public Gun

{

private:

Bullet \* bullet;

int numOfBullet;

int type;

Ak47Gun() {};

public:

Ak47Gun(Bullet \*bullet)

{

this->bullet = bullet;

type = Ak47;

};

void setNumberOfBullets(int num)

{

numOfBullet = num;

};

Bullet\* getBullet()

{

return bullet;

}

int getType() { return type; };

void fire()

{

cout << "Gun type:" << type << " fire with " << numOfBullet <<

"speed,dame:" << bullet->getSpeed() << ","

<< bullet->getDamage()<< endl;

};

};

class M4a1Gun :public Gun

{

private:

Bullet \* bullet;

int numOfBullet;

int type;

M4a1Gun() {};

public:

M4a1Gun(Bullet \*bullet)

{

this->bullet = bullet;

type = M4a1;

};

void setNumberOfBullets(int num)

{

numOfBullet = num;

};

Bullet\* getBullet()

{

return bullet;

}

int getType() { return type; };

void fire()

{

cout << "Gun type:" << type << " fire with " << numOfBullet <<

"speed,dame:" << bullet->getSpeed() << ","

<< bullet->getDamage()<< endl;

};

};

int main()

{

Ak47Gun \*g = new Ak47Gun(new Bullet(small, large));

M4a1Gun \*g2 = new M4a1Gun(new Bullet(large, large));

system("PAUSE");

return 0;

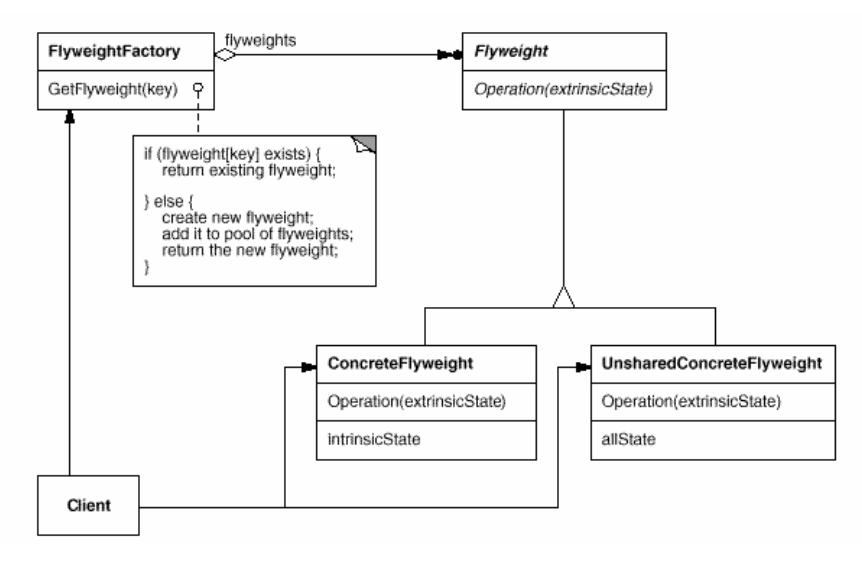
}

# Weakness of above solution:

* Not efficient
* waste memory
* longer creating time.

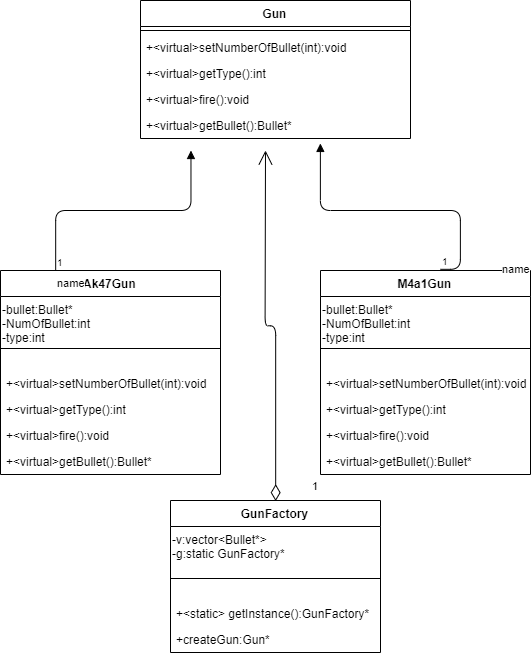
# Introduction to flyweight pattern:

* Flyweight is a structural design pattern.
* Flyweight uses sharing to support large numbers of fine-grained objects efficiently.
* Flyweight pattern divide an object into 2 parts:
  + Intrinsic state: all constant attributes (immutable object)of that object ,they can’t be changed once created.
  + Extrinsic state: all attributes which can be changed in running time.
* We can only get object through flyweight factory which manage all processes under the initialization of objects. Flyweight will return the same intrinsic state part if it had been created before. Otherwise, it will allocate new memory for that intrinsic part.



* Structure of Flyweight includes:
  + Flyweight: declares an interface through which flyweights can receive and act on extrinsic state.
  + ConcreteFlyweight: implements the Flyweight interface and adds storage for intrinsic state, if any. A ConcreteFlyweight object must be sharable. Any state  
    it stores must be intrinsic; that is, it must be independent of the  
    ConcreteFlyweight object's context.
  + UnsharedConcreteFlyweight: not all Flyweight subclasses need to be shared. The Flyweight interface enables sharing; it doesn't enforce it. It's common for  
    UnsharedConcreteFlyweight objects to have ConcreteFlyweight objects as  
    children at some level in the flyweight object structure.
  + FlyweightFactory: creates and manages flyweight objects. It ensures that flyweights are shared properly. When a client requests a  
    flyweight, the FlyweightFactory object supplies an existing instance  
    or creates one, if none exists.

# Uml & code:



Code:

#include<iostream>

#include<vector>

#include<string>

using namespace std;

#define small 0

#define medium 1

#define large 2

#define Ak47 0

#define M4a1 1

class Bullet

{

private:

int speed;

int damage;

Bullet() {};

public:

static int count;

Bullet(int speed, int damage)

{

++count;

this->speed = speed;

this->damage = damage;

};

int getSpeed() { return speed; };

int getDamage() { return damage; };

};

int Bullet::count = 0;

class Gun

{

public:

virtual void setNumberOfBullets(int size) {};

virtual int getType() { return 0; };

virtual void fire() {};

virtual Bullet\* getBullet() { return nullptr; };

};

class Ak47Gun:public Gun

{

private:

Bullet \* bullet;

int numOfBullet;

int type;

Ak47Gun() {};

public:

Ak47Gun(Bullet \*bullet)

{

this->bullet = bullet;

type = Ak47;

};

void setNumberOfBullets(int num)

{

numOfBullet = num;

};

Bullet\* getBullet()

{

return bullet;

}

int getType() { return type; };

void fire()

{

cout << "Gun type:" << type << " fire with " << numOfBullet <<

"speed,dame:" << bullet->getSpeed() << "," << bullet->getDamage()

<< endl;

};

};

class M4a1Gun :public Gun

{

private:

Bullet \* bullet;

int numOfBullet;

int type;

M4a1Gun() {};

public:

M4a1Gun(Bullet \*bullet)

{

this->bullet = bullet;

type = M4a1;

};

void setNumberOfBullets(int num)

{

numOfBullet = num;

};

Bullet\* getBullet()

{

return bullet;

}

int getType() { return type; };

void fire()

{

cout << "Gun type:" << type << " fire with " << numOfBullet <<

"speed,dame:" << bullet->getSpeed() << "," << bullet->getDamage()

<< endl;

};

};

class GunFactory

{

private:

vector<Bullet\*> v;

GunFactory() {};

static GunFactory \* g;

public:

static GunFactory \*getInstance()

{

if (g == nullptr)

g = new GunFactory;

return g;

}

Gun \*createGun(int type,int NumberOfBullet, int bulletSpeed, int bulletDame)

{

Bullet\* b = nullptr;

Gun \*g;

for (vector<Bullet\*>::iterator it = v.begin();it!=v.end();++it)

{

if ((\*it)->getDamage() == bulletDame && (\*it)->getSpeed() == bulletSpeed)

{

b = \*it;

break;

}

};

if (b == nullptr)

{

b = new Bullet(bulletSpeed, bulletDame);

v.push\_back(b);

}

switch (type)

{

case Ak47:

g = new Ak47Gun(b);

break;

case M4a1:

g = new M4a1Gun(b);

break;

default:

return nullptr;

break;

}

g->setNumberOfBullets(NumberOfBullet);

v.push\_back(b);

return g;

}

};

GunFactory\* GunFactory::g = nullptr;

int main()

{

GunFactory \* g = GunFactory::getInstance();

Gun \*gun = nullptr;

for (int i = 0; i < 1000; ++i)

{

gun=g->createGun(rand() %2, rand() % 3, rand() % 3, rand() % 3);

gun->fire();

}

cout << "Number of count:" << Bullet::count;

system("PAUSE");

return 0;

}

Other problems using this pattern:

Modern web browsers use the intrinsic state of this pattern to prevent loading same images more than once. When browser loads a web page, it will check through all

images store in internal cache if they have been loaded before.

Cons and pros:

* Pros:
  + Save memory.
  + Faster than traditional method.
  + Using memory efficiently.
* Cons:
  + More code.
  + Harder to debug if there is something wrong.

# References:

<https://www.geeksforgeeks.org/flyweight-design-pattern/>

<https://sourcemaking.com/design_patterns/flyweight/cpp/2>

<https://refactoring.guru/design-patterns/flyweight>

<https://www.tutorialspoint.com/design_pattern/flyweight_pattern.htm>