1. **需求分析**

用户能够使用键盘操作玩家的飞机并发射子弹进行射击，同时随着时间的增长和消灭的敌方飞机数量获得分数。

* 1. 需求列表：

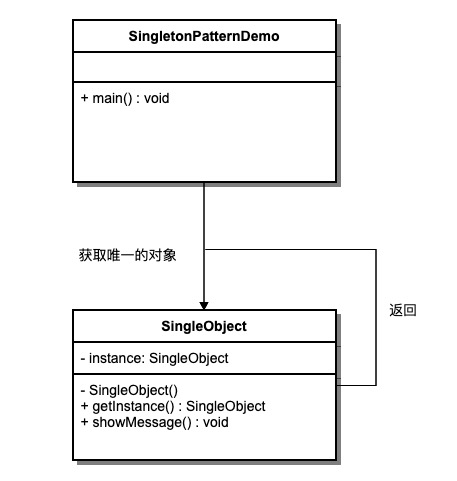
1. 程序能够接受用户的键盘输入，进行简单的界面操作，以及飞机的移动和设计操作。程序需要正确响应用户的单击、按住按键的操作，操作顺畅。
2. 敌人的飞机从屏幕的顶部随机生成，生成的敌人拥有一种以上的移动方式和弹幕射击方式，并且敌人飞机带有一定的生命值，在被玩家攻击数次后才算作被消灭。
3. 玩家会随着游戏时间的增长不断获得分数，在消灭一架飞机后将获得500分数，而每获得2500点分数后玩家会获得1点生命值奖励。玩家的射击速度会根据持有的生命值发生一定的改变。
4. **设计模式**

**2.1 单例模式**

2.1.1 单例模式定义

单例模式属于创建型模式，保证一个类仅有一个实例，并提供一个访问它的全局访问点可以直接访问，不需要实例化该类的对象。

2.1.2 单例模式的结构



单例模式的结构如上图。

在单例模式中我们将创建一个类，此类中包含一个自身的对象实例（指针或引用），。其提供了一个静态方法，供外界获取它的实例。同时，屏蔽（令为私有的）此类的各种构造函数，保证不能从类的静态方法之外获得实例对象。

2.1.3 单例模式在本程序中的应用与理解

本程序需要一个管理器用于存放从文件读取到的资源，然后可以从程序的任意其他位置获取一份共享的资源（因而也带有一定的享元工厂特性）。于是有如下资源管理器类设计：

class ResourcesManager{

public:

// singleton control

static ResourcesManager \*GetInstance();

ResourcesManager(const ResourcesManager &r) = delete;

ResourcesManager(ResourcesManager &&r) = delete;

~ResourcesManager();

private: // singleton content

static ResourcesManager \*sm\_instancePtr;

ResourcesManager() = default;

public:

bool ReadTexture(const std::string &filename, const std::string &alias);

sf::Texture \*GetTexture(const std::string &alias);

bool FreeTexture(const std::string &alias);

private:

std::map<std::string, sf::Texture \*> m\_textureStorage;

};

单例类ResourcesManager通过GetInstance()方法获得内部管理的唯一实例对象的指针。然而，GetInstance()方法并未对应对多线程环境作出任何修改，因为在游戏开始时第一件事就是使用该类把资源加载到内存，而这就是单线程的环境，创建单例唯一对象时并不会出现线程不同步问题。

在C++中，应该对唯一对象进行回收操作，即提供释放唯一对象内存的静态方法。但在本程序中，这些资源的生命周期即为程序的生命周期，即使存在内存泄漏，在用户关闭程序后该内存就会被操作系统回收。

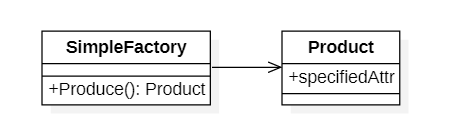
我所理解的单例模式适用场景不仅是要控制一个类的实例存在的数量，更重要的是便于程序员在任意地方能够保留原始状态地、无需重新创建对象地获取到这个类的实例，减少了对象间的依赖。

**2.2 工厂模式**

2.2.1 工厂模式定义

定义一个专门用于创建新对象的类，创建时在直接创建类对象的基础上(即new 等实例化)再对正在创建的类进行一些“出厂设置”，此种工厂模式又称简单工厂模式。在创建对象时不会对客户端暴露创建逻辑，把“出厂设置”之一繁琐的过程隐藏在工厂的“生产”动作内。

2.2.2 工厂模式的结构



工厂模式的结果如上图。

在此简单工厂中，工厂类SimpleFactory负责使用new对象，并设置一些工厂持有者所希望设置的产品身上的特殊参数，如图中Product的specifiedAttr成员。

2.2.3 工厂模式在本程序中的应用与理解

本程序在多处使用到了工厂模式和简单工厂模式，其中典型的是用于生成有贴图和不同速度子弹的子弹工厂。产品和工厂类如下：

class Bullet : public WorldObject {

public:

sf::Vector2f velocity = {0, 0};

void MoveForward();

};

class BulletFactory{

public:

Bullet \*Produce(const sf::Vector2f &from, float angle);

public:

void SetBulletTexture(sf::Texture \*bulletTexture);

void SetBulletTexture(sf::Texture \*bulletTexture, const sf::IntRect &cutting);

private:

sf::Texture \*m\_bulletTexture = nullptr;

bool m\_isCut = false;

sf::IntRect m\_cutting = {0, 0, 0, 0};

};

其中，工厂的Produce方法实现如下：

Bullet \*BulletFactory::Produce(const sf::Vector2f &from, float angle) {

if (m\_bulletTexture == nullptr) throw std::exception("Invalid Texture for Factory");

auto bullet = new Bullet;

bullet->velocity.x = -1 \* cosf(fmod(angle, 360.0f) \* 3.1415926f / 180);

bullet->velocity.y = -1 \* sinf(fmod(angle, 360.0f) \* 3.1415926f / 180);

bullet->setPosition(from);

bullet->setTexture(\*m\_bulletTexture);

if (m\_isCut) bullet->setTextureRect(m\_cutting);

return bullet;

}

子弹工厂BulletFactory让工厂的持有者，如玩家或敌人对象，不需要关心子弹是如何创建的，不需要手动设置繁琐的参数。同时，子弹工厂也封装了子弹的速度计算和贴图设置，可以很方便地创建各种方向的子弹而制作各种样式的子弹发射效果。

**2.3 模板方法（策略模板）**

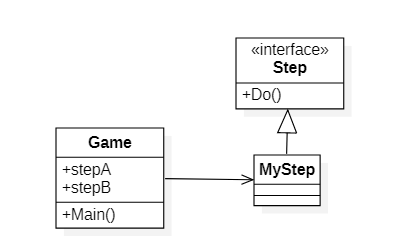
2.3.1模板方法定义

模板方法中，一个类公开定义了执行它的方法的方式，将这些通用流程抽象出来，并提供接口供重写和替换其中的方法，实现执行方法的可变。而策略模式又容易与模板方法相结合使用，即模板方法中不同的子步骤通过策略模式再提供不同的实现。

2.3.2 模板方法的结构



上图为模板方法的结构，而结合策略模式后，结构类似下图：



相比之下更加灵活。

2.3.3 模板方法（策略模板）在本程序中的应用与理解

本程序在实现敌人飞机的多种移动方式和多种射击方式上使用了该设计方法，类的框架如下代码：

class DoerStrategy{

public:

virtual bool Do() = 0;

};

class Doer{

public:

DoerStrategy \*performStrategy, \*intervalSleepStrategy, \*controlStrategy;

public:

Doer();

~Doer();

public:

void Start();

void Stop();

private:

void ThreadFunc();

bool m\_isThreadKeepContinue = false;

std::thread \*m\_thread = nullptr;

};

类Doer中实现了一个模板方法，用户可以通过实现DoerStrategy接口实现自己的各种策略，然后替换Doer中的performStrategy、intervalSleepStrategy、controlStrategy任何一个来灵活实现各种效果。

同时，类Doer为适应游戏“基于循环”的设计，将模板方法的循环置于另一线程上执行，因此可以实现某些基于时间的行为效果而不阻塞主线程的渲染和事件处理或增加额外的时间判断造成性能下降。

class RandomDancerMover : public DoerStrategy {

public:

RandomDancerMover(int rangeLeft, int rangeRight, Enemy \*obj)

: m\_moveRangeLeft(rangeLeft), m\_moveRangeRight(rangeRight), m\_obj(obj) {

if (m\_moveRangeLeft <= 0) m\_moveRangeLeft = 1;

if (m\_moveRangeRight >= WIN\_WIDTH - 32) m\_moveRangeRight = WIN\_WIDTH - 32;

}

bool Do() override {

sf::Vector2f move = {0, 1.0f};

if (m\_obj->getPosition().x < m\_moveNextWayPoint + 1.0f && m\_obj->getPosition().x > m\_moveNextWayPoint - 1.0f) {

std::default\_random\_engine randomEngine(std::chrono::system\_clock::now().time\_since\_epoch().count());

std::uniform\_int\_distribution<int> wayPointDistribution(m\_moveRangeLeft, m\_moveRangeRight);

m\_moveNextWayPoint = wayPointDistribution(randomEngine);

} else {

if (m\_obj->getPosition().x > m\_moveNextWayPoint + 0.5f) move.x = -1.0f;

if (m\_obj->getPosition().x < m\_moveNextWayPoint - 0.5f) move.x = 1.0f;

}

m\_obj->move(move);

return true;

}

private:

int m\_moveRangeLeft, m\_moveRangeRight, m\_moveNextWayPoint = 200;

Enemy \*m\_obj;

};

该实现通过在另一线程上，不断生成新的路径点以及判断是否到达路径点，同时移动对象，来实现敌方飞机的不规则移动。

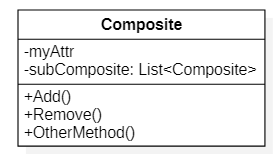
**2.4 组合模式**

2.4.1 组合模式定义

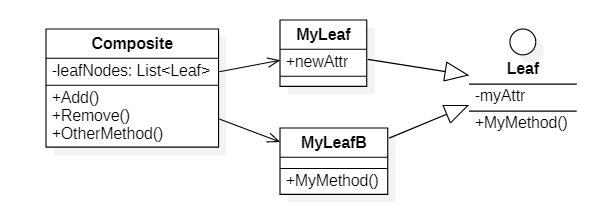
组合模式又叫部分整体模式，是用于把一组相似的对象当作一个单一的对象。组合模式依据树形结构来组合对象，用来表示部分以及整体层次。这种类型的设计模式属于结构型模式，它创建了对象组的树形结构。

2.4.2 组合模式的结构

如果使用树状的结构来管理组合对象或构件，那么结构将会如下图：



如果简化为只有一层，那么能用以下结构实现：



2.4.3组合模式在本程序中的应用与理解

本程序使用了上述简化后的只有一层的组合模式，因为需要组合的对象并没有嵌套关系或主从关系，都是平行的对象。本程序使用组合模式来存放上一模板模式不同的行为，使之能有多种模板共同生效，代码如下：

class DoerComposite {

public:

~DoerComposite();

public:

void Add(Doer \*doer);

void StartAll();

void StopAll();

private:

std::vector<Doer \*> m\_doerList;

};

程序中的敌人持有这样的“组合容器”来存放只有一层的、没有子构件的具体构件。并且提供了StartAll()和StopAll()方法便捷地启停Doer中的线程。因为每一个具体构件的生命周期往往跟随者持有者，因而没有实现删除构件的操作，而是在析构函数中一次性全部清理。

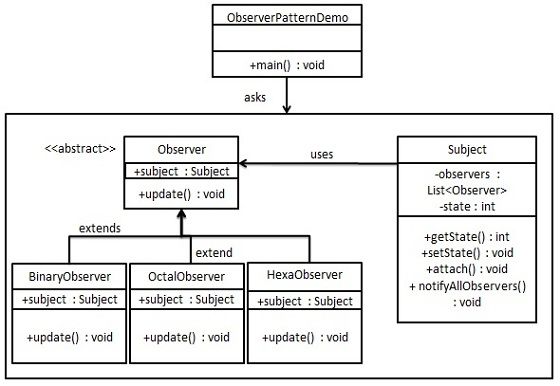
组合模式更适合用于带有层级结构的系统，比如GUI界面应用等，否则难以按照所谓的“规范”实现一个结构上完全类似的模式。

**2.5 观察者模式**

2.5.1 观察者模式定义

当一个对象被修改时，则会自动通知依赖它的对象，并且被通知的、被依赖的对象往往是多个，这种模式被称为观察者模式。一个对象状态改变给其他对象通知的问题，而且要考虑到易用和低耦合，保证高度的协作。

2.5.2 观察者模式的结构



观察者模式使用Subject、Observer 类，Subject 对象存放了绑定的观察者Observer 对象，其他类可通过Subject类的notifyAll()方法通知所有监听这一Subject的对象。

2.5.3观察者模式在本程序中的应用与理解

本程序在响应子弹命中和生命值归零事件上使用了观察者模式，代码如下：

class ObserverInterface {

public:

ObserverInterface() = default;

virtual ~ObserverInterface() = default;

public:

virtual void Response() = 0;

};

class ObserverNotifier {

public:

virtual ~ObserverNotifier();

public:

void NotifyAll();

public:

std::vector<ObserverInterface \*> m\_observers;

};

在C++中并不常用类似Java的ActionListener的“间接回调”的观察者模式，而是使用“直接回调”即用回调函数来实现，同时，事件的响应也依赖程序员手动地分发事件到具体的类。这导致了很高的耦合性和不便。

在使用观察者模式后，我们能够以低耦合的方式去实现让我们的类去“关注”某些事件。