实验二报告

一、 单例模式

1. 应用场景分析

单例模式（Singleton Pattern）是一种创建型设计模式，能够保证一个类只有一个实例， 并提供一个访问该实例的全局节点。

从当前的游戏设计来讲，飞机大战中只有一种英雄机且每局游戏只有一架英雄机（尽管本游戏改编的来源中有多个英雄机，本实验大概是为了简化功能所以限定只能有一种英雄机，参考：全民飞机大战）。

而在只允许有一个实例的情况下，原本实验一的代码却存在同时实例化多个英雄机的可能，这是和我们期望不符的，所以需要使用单例模式来约束这个类最多只能有一个对象被实例化。

单例模式常用的设计方案有饿汉式、懒汉式和双重检查锁定，本人采用的是双重锁实现的单例模式。

2. 解决方案

AbstractFlyingObject 是所有飞行物品的抽象父类,用于设计飞行物品基本的方法和属性，直接继承其的子类有 AbstractAircraft 和 BaseBullet；

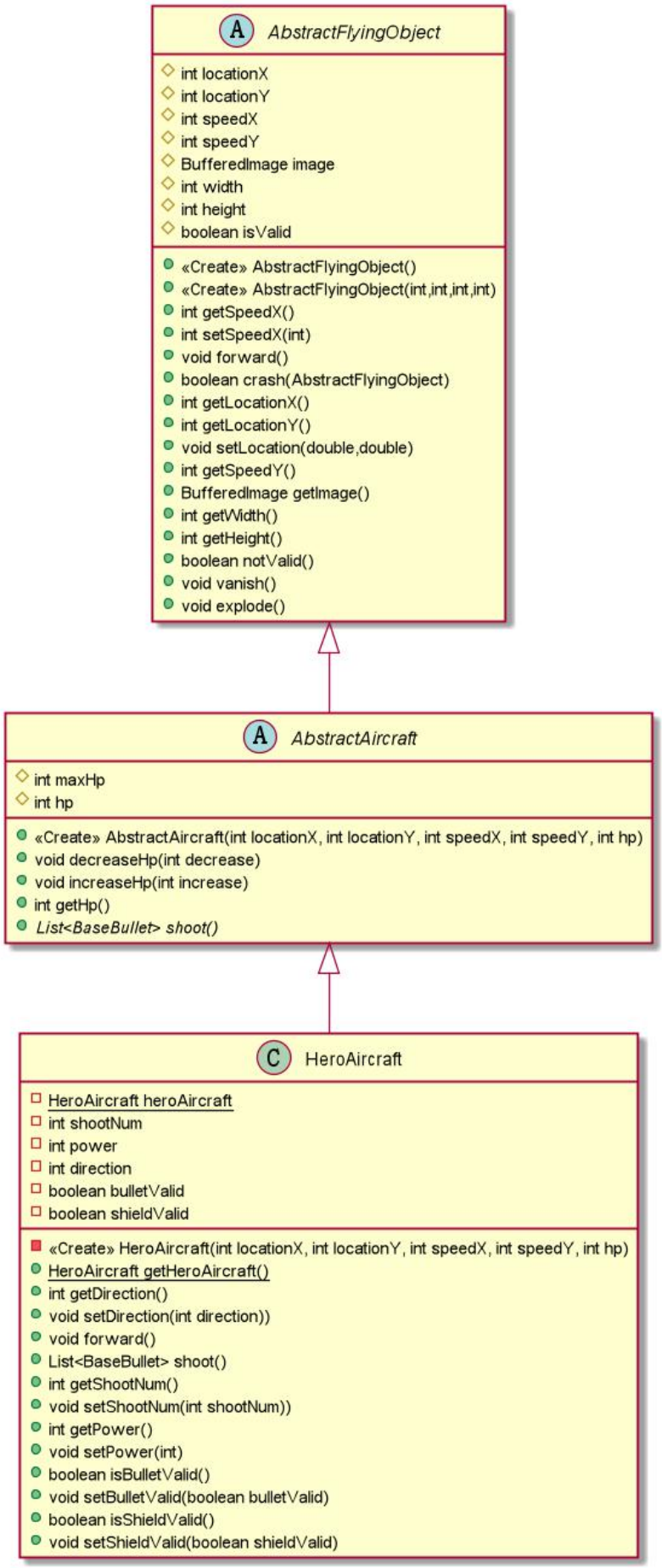
AbstractAircraft 是飞行器的抽象父类，直接继承其的子类有 HeroAircraft 和 AbstractEnemy；

HeroAircraft 是英雄机类，但我们限定该类只能有一个实例，故使用单例模式，将构造函数 HeroAircraft(int,int,int,int,int)设定为私有，只能由静态方法 getHeroAircraft()来实例化对象，同时其中的双重锁限定了对象只能被创建一次，避免了潜在的多次创建英雄机的可能性。

该类中关键属性为 heroAircraft，实际上就是英雄机实例的引用，设置为私有确保了只能由关键方法 getHeroAircraft 来获取实例，而关键方法中的双重锁：

1．引用为空的时候才可能进行创建，非空引用则直接返回 heroAircraft 引用；2．当引用为空的时候，确保只有一个线程的类能创建实例，避免多线程产生多个实例。

uml 类图如下所示：


AbstractFlyingObject
int locationX
int locationY
int speedX
int speedY
BufferedImage image
int width
int height
boolean isValid
«Create» AbstractFlyingObject()
«Create» AbstractFlyingObject(int,int,int,int)
int getSpeedX()
int setSpeedX(int)
void forward()
boolean crash(AbstractFlyingObject)
int getLocationX()
int getLocationY()
void setLocation(double,double)
int getSpeedY()
BufferedImage getImage()
int getWidth()
int getHeight()
boolean notValid()
void vanish()
void explode()
AbstractAircraft
int maxHp
int hp
«Create» AbstractAircraft(int locationX, int locationY, int speedX, int speedY, int hp)
void decreaseHp(int decrease)
void increaseHp(int increase)
int getHp()
List<BaseBullet> shoot()
HeroAircraft
HeroAircraft heroAircraft
int shootNum
int power
int direction
boolean bulletValid
boolean shieldValid
«Create» HeroAircraft(int locationX, int locationY, int speedX, int speedY, int hp)
HeroAircraft getHeroAircraft()
int getDirection()
void setDirection(int direction))
void forward()
List<BaseBullet> shoot()
int getShootNum()
void setShootNum(int shootNum))
int getPower()
void setPower(int)
boolean isBulletValid()
void setBulletValid(boolean bulletValid)
boolean isShieldValid()
void setShieldValid(boolean shieldValid)

二、 工厂模式

1. 应用场景分析

工厂模式（Factory Pattern）也是一种创建型设计模式， 其在父类中提供一个创建对象的方法， 由子类决定实例化对象的类型。

在本实验中，有 3 种类型的敌机：Mob，Elite，Boss。Mob 敌机和 Elite 敌机以一定频率在界面随机位置出现并向屏幕下方移动。Boss 敌机以背景速度 1 向下移动，直至被消灭或者撞毁精英机。敌机通过生命值（血）生存，被英雄机子弹击中损失部分生命值，生命值为 0 时坠毁。

游戏中还有 3 种类型的道具：火力道具、炸弹道具、加血道具。Elite 敌机坠毁后，以一定概率随机在坠毁处出现某种道具。道具以背景速度 1 向屏幕下方移动，与英雄机碰撞或道具移动至界面底部后消失。英雄机碰撞道具后，道具自动触发生效。

三种敌机和三种道具都满足游戏过程中不断产生的过程，而原本实验一中的代码将道具和敌机的生成直接硬写到 Game 类中，一不利于工程的封装和整洁，二会导致可能的对道具和敌机参数的误操作，不安全，这里采用属性和构造分离的设计思想，采用工厂模式完成代码的重构，通过敌机属性在工厂中被设定并构造，避免了敌机属性直接出现在 Game 类中。

2. 解决方案

将 PlantUML 插件绘制的类图截图到此处，并对 UML 类图中每个类、接口，以及其关键属性和方法进行简单说明。

下方第一幅图是敌机的工厂模式类图，略过已经介绍过的 AbstractFlyingObject 和 AbstractAircraft 。

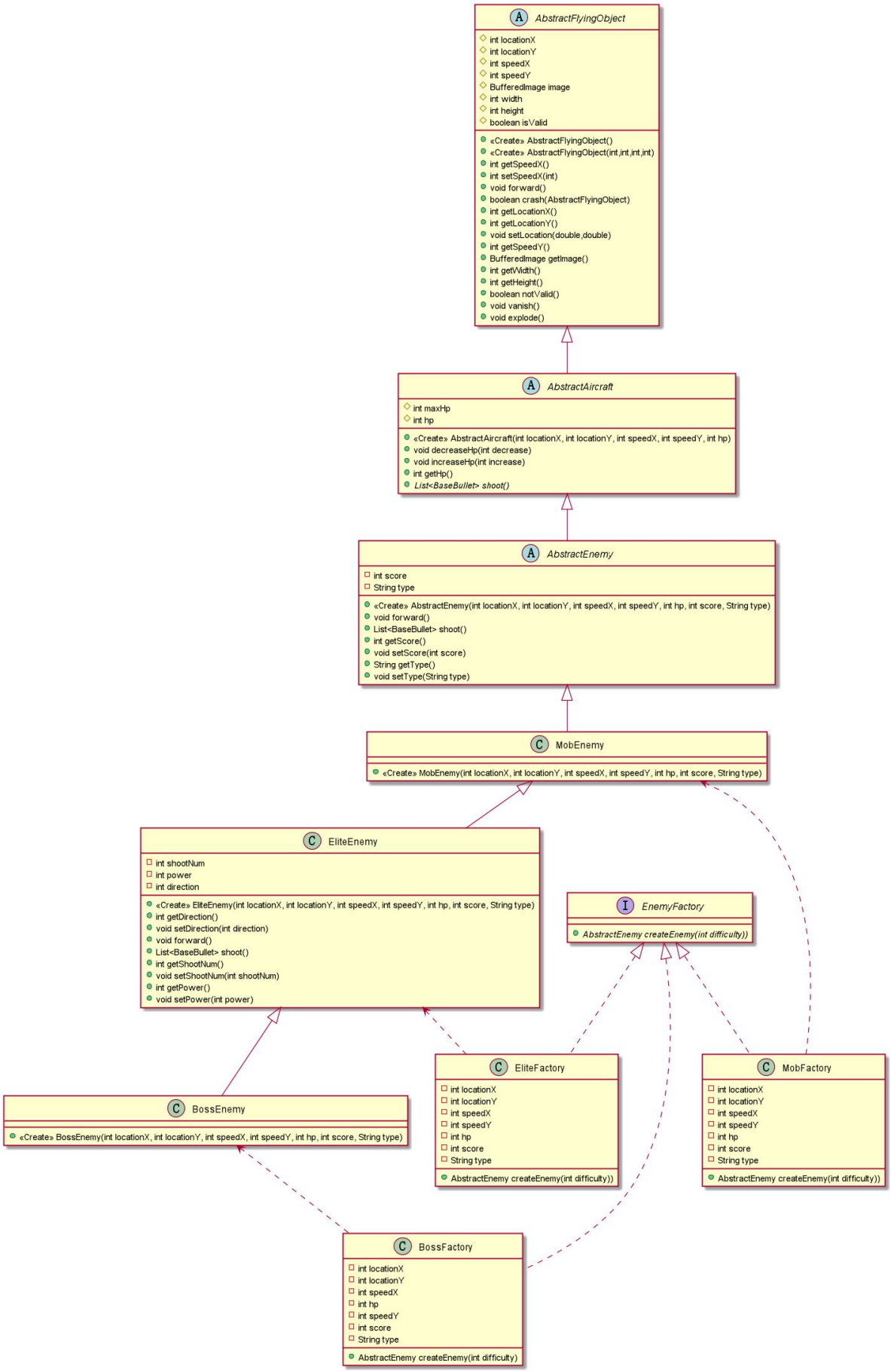
AbstractEnemy 类是所有敌方物件的抽象父类，属性 score 和 type 为所有子类均继承的属性，是敌方物品（敌机，敌机坠毁后的道具）都各自具有的属性；

对于三种敌机类型，我采用的是继承的实现方式。

Mob 敌机直接继承抽象父类，不添加额外的参数，同时作为 Elite 敌机的父类；

Elite 敌机继承 Mob 敌机类，同时改写了父类默认的属性和方法，用于实现敌机的射击和左右移动；

Boss 敌机继承 Elite 敌机类，当前没有添加除射击以外的其他内容，修改了 shotNum；


AbstractFlyingObject
int locationX
int locationY
int speedX
int speedY
BufferedImage image
int width
int height
boolean isValid
«Create» AbstractFlyingObject()
«Create» AbstractFlyingObject(int,int,int,int)
int getSpeedX()
int setSpeedX(int)
void forward()
boolean crash(AbstractFlyingObject)
int getLocationX()
int getLocationY()
void setLocation(double,double)
int getSpeedY()
BufferedImage getImage()
int getWidth()
int getHeight()
boolean notValid()
void vanish()
void explode()
AbstractAircraft
int maxHp
int hp
«Create» AbstractAircraft(int locationX, int locationY, int speedX, int speedY, int hp)
void decreaseHp(int decrease)
void increaseHp(int increase)
int getHp()
List<BaseBullet> shoot()
AbstractEnemy
int score
String type
«Create» AbstractEnemy(int locationX, int locationY, int speedX, int speedY, int hp, int score, String type)
void forward()
List<BaseBullet> shoot()
int getScore()
void setScore(int score)
String getType()
void setType(String type)
MobEnemy
«Create» MobEnemy(int locationX, int locationY, int speedX, int speedY, int hp, int score, String type)
EliteEnemy
int shootNum
int power
int direction
«Create» EliteEnemy(int locationX, int locationY, int speedX, int speedY, int hp, int score, String type)
int getDirection()
void setDirection(int direction)
void forward()
List<BaseBullet> shoot()
int getShootNum()
void setShootNum(int shootNum)
int getPower()
void setPower(int power)
EnemyFactory
AbstractEnemy createEnemy(int difficulty))
BossEnemy
«Create» BossEnemy(int locationX, int locationY, int speedX, int speedY, int hp, int score, String type)
EliteFactory
int locationX
int locationY
int speedX
int speedY
int hp
int score
String type
AbstractEnemy createEnemy(int difficulty))
MobFactory
int locationX
int locationY
int speedX
int speedY
int hp
int score
String type
AbstractEnemy createEnemy(int difficulty))
BossFactory
int locationX
int locationY
int speedX
int hp
int speedY
int score
String type
AbstractEnemy createEnemy(int difficulty)

对于敌机工厂模式的改写，我采用的是定义 EnemyFactory 的接口，将敌机的构造函数统一到实现了该接口下 createEnemy(int)的三个子类中：MobFactory，EliteFactory，BossFactory。三个敌机工厂内置了敌机的属性参数，并且分别调用三种敌机的构造函数，用于构造敌机，实现了敌机类本身属性和敌机类构造的分离，即实现了工厂模式的功能。三个工厂调用的是敌机类的构造函数，故工厂与敌机间存在依赖关系，而三个工厂子类实现了 EnemyFactory 的接口，故工厂与接口间是实现关系。

下方第二幅图是道具的工厂模式类图，除了三种道具间不存在继承关系外，与上述所讲敌机与工厂的结构类似，同样略去已经阐述过的部分。

三种道具类分别为 BombProp，BulletProp，BloodProp，三种道具的主要功能由 activate(HeroAircraft，List<AbstractEnemy>，List<BaseBullet>，int) 方法实现，用于激活道具的功能，故需要对参数列表中的对象进行修改。

本实验中对于工厂模式的代码重构部分，类似敌机的工厂模式设计，我采用的是定义 PropFactory 的接口，将道具的构造函数统一到实现了该接口下 createProp(int,int,String)的三个子类中：BombFactory，BulletFactory，BloodFactory。三个道具工厂内置了道具的大部分属性参数，除了需要获取的敌机坠毁坐标和产生的道具种类，分别调用三种道具的构造函数，用于构造道具，实现了道具类本身属性和道具类构造的分离，即实现了工厂模式的功能。三个道具工厂调用的是道具类的构造函数，故道具工厂与道具间存在依赖关系，而三个工厂子类实现了 PropFactory 的接口，故工厂与接口间是实现关系。