



哈爾濱工業大學 (深圳)
HARBIN INSTITUTE OF TECHNOLOGY

实验报告

开课学期: 2023 春季

课程名称: 面向对象的软件构造导论

实验名称: 飞机大战游戏的设计与实现

实验性质: 设计型

实验学时: 16 地点:

学生班级: 21 级计算机 3 班

学生学号: 210110315

学生姓名: 吕弋卿

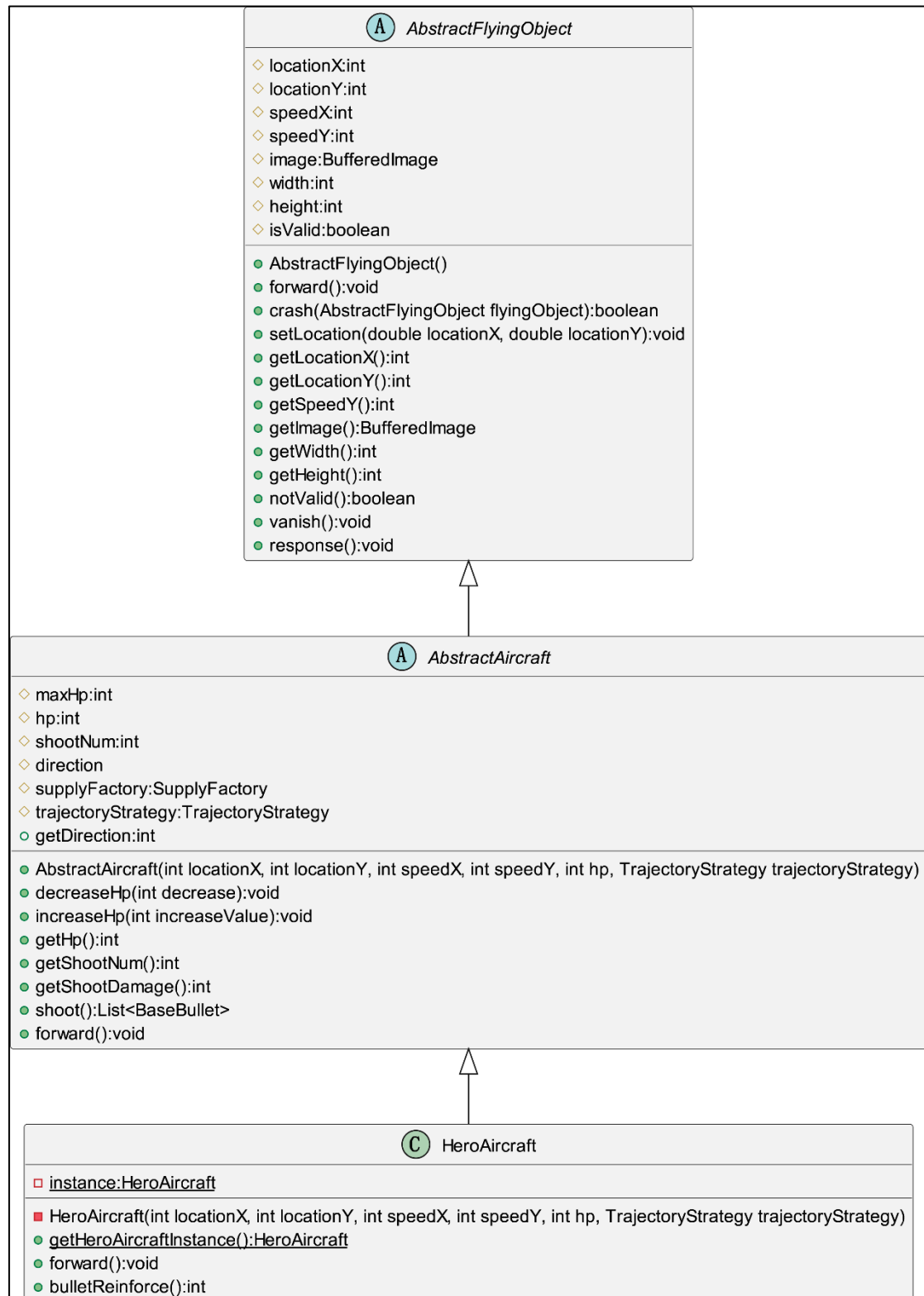
评阅教师:

报告成绩:

实验与创新实践教育中心制

2023 年 4 月

2. 设计模式结构图



- AbstractFlyingObject 与 AbstractAircraft 为已给出的代码，不做赘述。
- HeroAircraft 类是英雄机所属的类：

成员变量：

- 静态变量 instance 用于存储 HeroAircraft 在单例模式下的唯一实例。

成员方法：

- 将构造方法 HeroAircraft 设为私有，这样就不能通过外部调用来创建对象。
- 静态公有方法 getHeroAircraftInstance，用于外部调用并返回该类的对象，此函数首先对 instance 变量进行检查，若已有实例则直接返回；若没有已存在的实例，则创建一个新的实例并用 instance 存储，再返回这个实例。（懒汉式）
- 公有方法 forward，重写了父类的方法，使得英雄机不能自动移动，而通过鼠标指针移动。
- 公有方法 bulletReinforce，用于拾取到 FireSupply 时增加子弹发射数量。

2.3.2 工厂模式

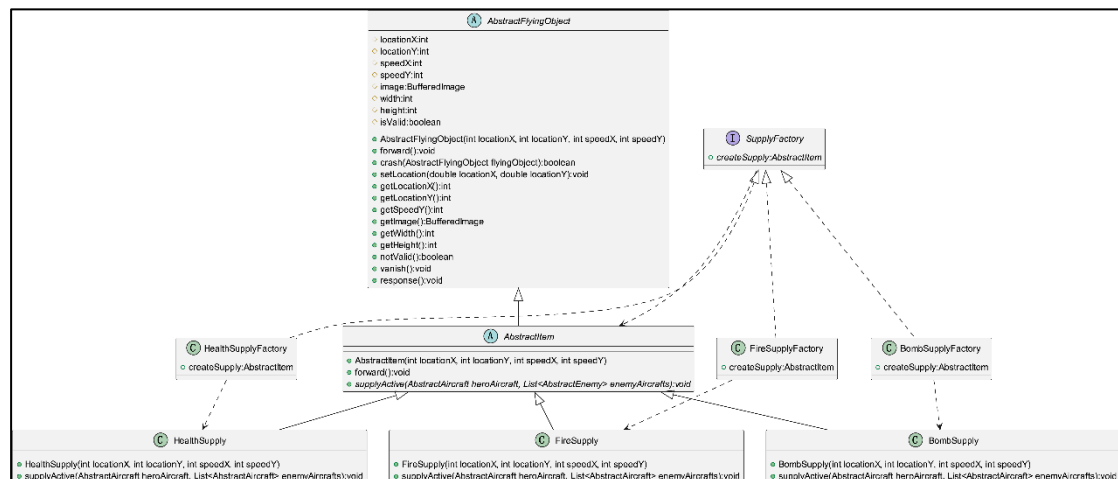
1. 应用场景分析

在飞机大战游戏中，敌机中的普通敌机、精英敌机的实例数量较多；三种掉落道具的实例也会大量产生，因此，上述的几种类的实例化可以采用工厂模式来实现。

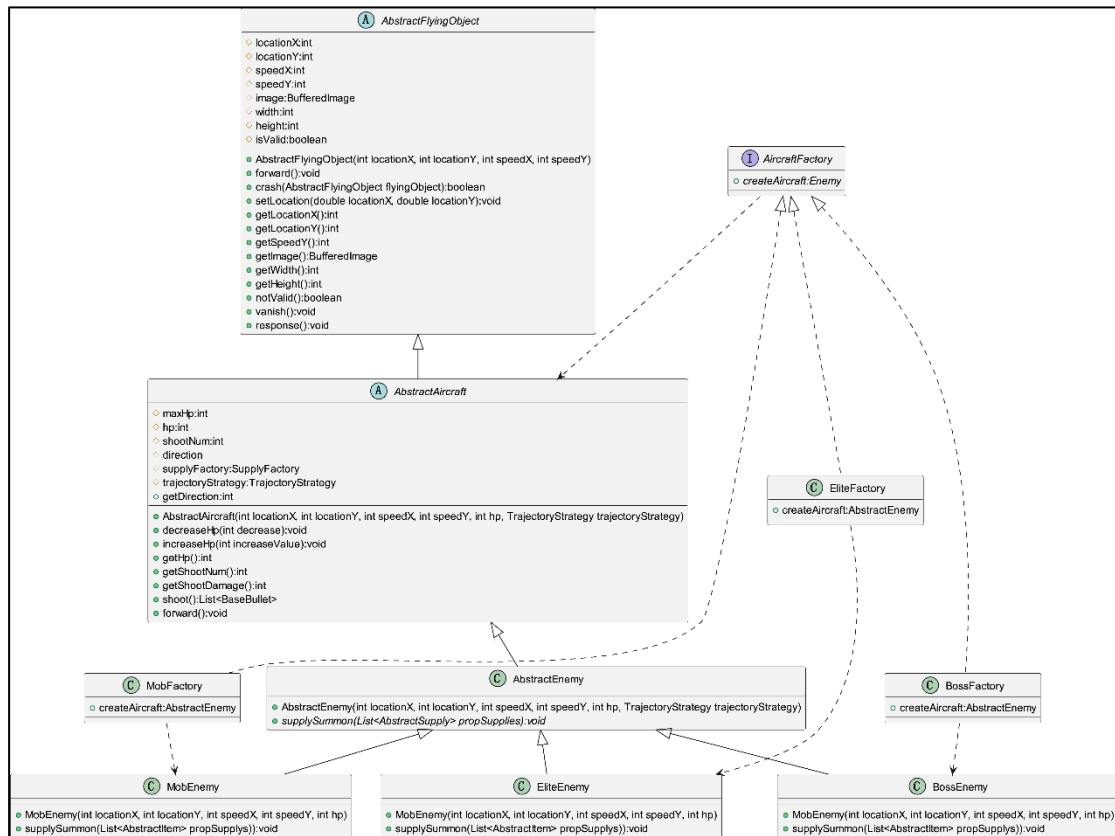
目前问题：若不使用工厂模式，则不符合封装与开闭原则，安全性与可维护性下降；若使用工厂模式，则会增加代码量和程序复杂度。

2. 设计模式结构图

道具及道具工厂：



敌机及敌机工厂：



敌机产品类：

三种敌机都继承自 AbstractEnemy,这里以 EliteEnemy 类为例做说明：

成员方法：

- 公有构造方法 EliteEnemy。
- 公有方法 supplySummon，当精英机被消灭时，生成补给道具。

敌机工厂类：

采用工厂模式(Factory Pattern),由 AircraftFactory 对外提供统一的实例化接口。MobFactory 和 EliteFactory 是具体工厂类，重写抽象工厂方法，使其能够返回具体的子类实例。其中：

- 接口类 AircraftFactory 依赖于抽象父类 AbstractAircraft
- 具体工厂类 MobFactory 依赖于产品类 MobEnemy
- 具体工厂类 EliteFactory 依赖于产品类 EliteEnemy
- 具体工厂类 BossFactory 依赖于产品类 BossEnemy

道具产品类：

共有三个子类：HealthSupply, FireSupply, BombSupply, 都继承自抽象父类

AbstractItem。抽象父类 AbstractItem 继承自 AbstractFlyingObject，此外，还拥有一个构造方法和一个抽象方法 supplyActive，这是道具类统一的激活方法，在测试类中有统一的接口，具体实现何种功能则由具体的子类决定。以三个子类中的 HealSupply 为例进行说明：

成员函数：

- 公有构造方法 HealthSupply。
- supplyActive 是对父类抽象方法的具体实现，是多态的体现，其具体定义为：

```
@Override
public void supplyActive(HeroAircraft heroAircraft, List<AbstractEnemy> enemyAircrafts){
    heroAircraft.increaseHp(60);
    MusicProxyThread.musicEffect(1);
    this.vanish();
}
```

其中 increaseHp 是英雄机类的方法，因为要修改其私有变量 hp，只能通过其自身类中的方法来实现。

下面再简述 FireSupply 和 BombSupply 的 supplyActive 方法：

FireSupply 类中：

```
@Override
public void supplyActive(HeroAircraft heroAircraft, List<AbstractEnemy> enemyAircrafts){
    Game.heroFireReinforceFlag = true;
    Game.FIRE_REINFORCE_TIME = Game.TIME;
    MusicProxyThread.musicEffect(1);
    this.vanish();
}
```

BombSupply 类中：

```
@Override
public void supplyActive(HeroAircraft heroAircraft, List<AbstractEnemy> enemyAircrafts){
    Game.GLOBAL_OBSERVED_SUBJECTS.notifyObserver();
    MusicProxyThread.musicEffect(2);
    this.vanish();
}
```

2.3.3 策略模式

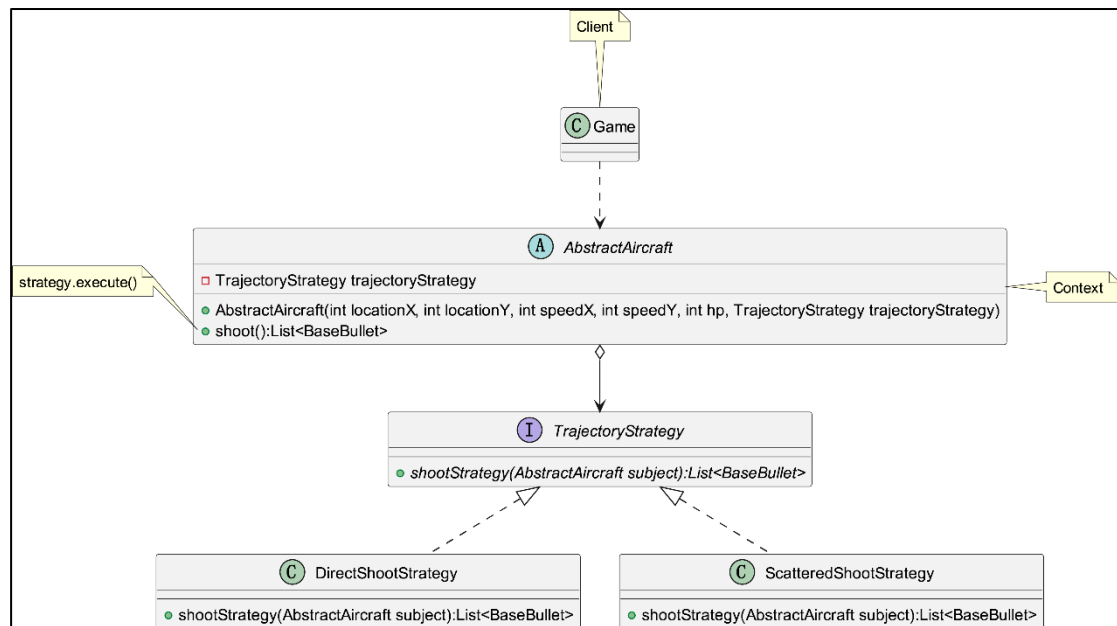
1. 应用场景分析

游戏中，射击模式有直射与散射两种，且会有多个类包含射击方法，使用策略模式将这两种射击模式封装起来，能够使算法的对象可以相互替换，从而使算法可以独立于使用它的对象，组织调用好算法能让程序的安全性、灵活性、可复用性、可扩展性提高。

如果不用策略模式，就要在每个飞机具体类中单独编写其射击方法，可复用性差；且如果要修改则要直接在飞机类中修改，不符合开闭原则。

2. 设计模式结构图

结合飞机大战实例，绘制该场景下具体的解决方案（UML 类图）。描述你设计的 UML 类图结构中每个角色的作用，并指出它的关键属性和方法。



- 环境类 **AbstractAircraft**:持有一个接口类的引用 **trajectoryStrategy**,可以在 `shoot()`方法中调用。
- 接口类 **TrajectoryStrategy**:定义了策略方法的规范
- 具体方法类 **DirectShootStrategy**(直射策略)与 **ScatteredShootStrategy**(散射策略), 这里以散射策略为例:

```
public class ScatteredShootStrategy implements TrajectoryStrategy{
    @Override
    public List<BaseBullet> shootStrategy(AbstractAircraft subject){
        List<BaseBullet> res = new LinkedList<>();
        int x = subject.getLocationX();
        int y = subject.getLocationY() + subject.getDirection()*2;
        if(subject instanceof AbstractEnemy){
            EnemyBullet bullet;
            for(int i=0; i<subject.getShootNum(); i++){
                bullet = new EnemyBullet(x + (i*2 - subject.getShootNum() + 1)*10, y, 4*i-4, 15, subject.getShootDamage());
                res.add(bullet);
            }
        }
        if(subject instanceof HeroAircraft){
            HeroBullet bullet;
            for(int i=0; i<subject.getShootNum(); i++){
                bullet = new HeroBullet(x, y, i-(int)(0.5*(subject.getShootNum()-1)), -10, subject.getShootDamage());
                res.add(bullet);
            }
        }
        return res;
    }
}
```

首先使用 `instanceof` 关键字来判断传入 `subject` 的类型，如果为英雄机则使用 **HeroBullet**，若为敌机则使用 **EnemyBullet**。

散射的弹道实现，即将 `speedX` 设为一个非 0 值。

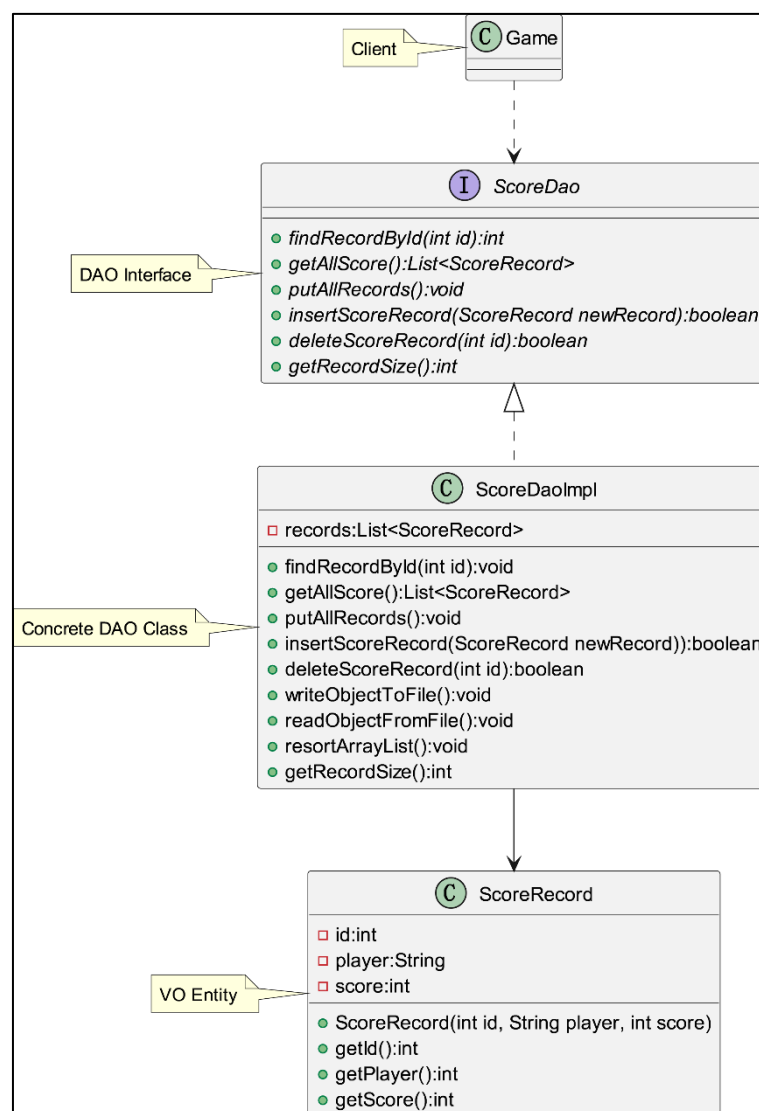
2.3.4 数据访问对象模式

1. 应用场景分析

描述飞机大战游戏中哪个应用场景需要用到此模式，设计中遇到的实际问题，使用该模式解决此问题的优势。

游戏中实现分数排行榜和文件读写需要 DAO 模式，把数据访问这一功能从高级业务中分离出来并封装，使得业务代码更简洁，DAO 功能的修改也符合开闭原则。

2. 设计模式结构图



1. 数据实体类 ScoreRecord:

属性:

➤ int id 记录 id

- String player 记录玩家昵称
- int score 记录得分

方法:

- ScoreRecord() 构造方法
- int getId() 返回 id
- String getPlayer() 返回玩家昵称
- int getScore() 返回得分

2. DAO 接口类 ScoreDao:

- abstract void findRecordById(int id):
 用 id 查找记录
- abstract List<ScoreRecord> getAllScore()
 返回记录列表
- abstract void putAllRecords()
 按照得分降序打印所有记录
- abstract boolean insertScoreRecord(ScoreRecord newRecord)
 插入一个新的记录并按照分数降序排序,插入成功返回 true, 否则 false
- abstract boolean deleteScoreRecord(int id)
 根据 id 删除记录, 删除成功返回 true, 否则 false
- abstract getRecordSize():int
 返回记录总条数

3. DAO 实体类 ScoreDaoImpl:

含有一个数据对象实例: records

包含 DAO 接口中的抽象方法实现, 此外, 还有以下方法:

- void writeObjectToFile() 将记录写入文件
- void readObjectFromFile() 从文件读取记录
- void resortArrayList() 按分数降序重新排序记录

2.3.5 观察者模式

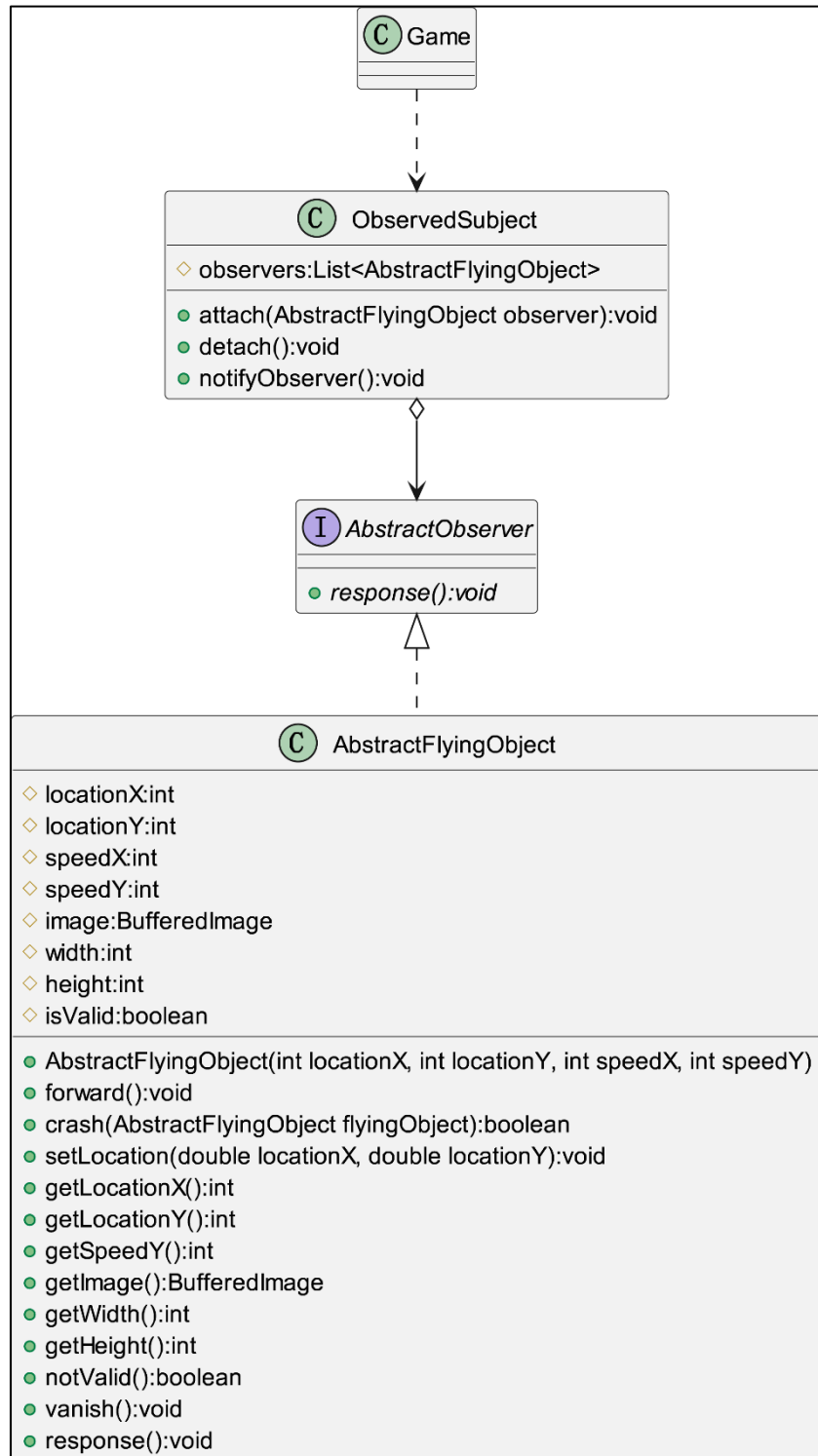
1. 应用场景分析

飞机大战游戏中, 炸弹道具爆炸时, 会清除屏幕中现有的精英敌机、普通敌机、敌机子弹, 并扣除 BOSS 敌机血量, 也即这些对象会对炸弹道具激活这一事件做出 response, 因此, 很适合用观察者模式来实现此功能, 优势有:

- 可以实现表示层和数据逻辑层的分离
- 在观察目标和观察者之间建立一个抽象的耦合

- 支持广播通信，简化了一对多系统设计的难度
- 符合开闭原则，增加新的具体观察者无须修改原有系统代码，在具体观察者与观察目标之间不存在关联关系的情况下，增加新的观察目标也很方便

2. 设计模式结构图



1. 观察者接口类 AbstractObserver:

- `public abstract void response()` 观察者作出反应的接口方法

2. 观察对象类 `ObservedSubject`:

属性字段:

- `private List<AbstractObserver> obs` 观察者的引用列表

方法:

- `public void attach()` 添加观察者
- `public void detach()` 移除观察者
- `public void notifyObserver()` 通知观察者做出反应

3. 具体观察者类 `AbstractFlyingObject`:

由分析知, 需要做出反应的对象类有敌机、敌方子弹, 其共有的超类是 `AbstractFlyingObject`, 因此此类 implements 了 `AbstractObserver` 接口, 并重写了 `response` 方法。

4. 客户端 `Game`:

`Game` 类中持有一个 `ObservedSubject` 实例 `GLOBAL_OBSERVED_SUBJECTS`, 在三种敌机以及敌机子弹产生时, 将其加入 `GLOBAL_OBSERVED_SUBJECTS` 的观察者列表中, 当 `BombSupply` 激活时, `GLOBAL_OBSERVED_SUBJECTS` 将通知列表中的观察者作出反应。

2.3.6 模板模式

1. 应用场景分析

在飞机大战游戏中, 不同游戏难度的游戏流程不会改变, 只是其中的特定步骤发生了变化, 因此很适合用模板模式来实现。模板模式将不变的行为放在超类, 去除了子类中的重复代码, 也提高了代码复用, 是多态的良好应用。




2. 三种难度

请描述你的三种游戏难度是如何设计的，影响游戏难度的因素有哪些。

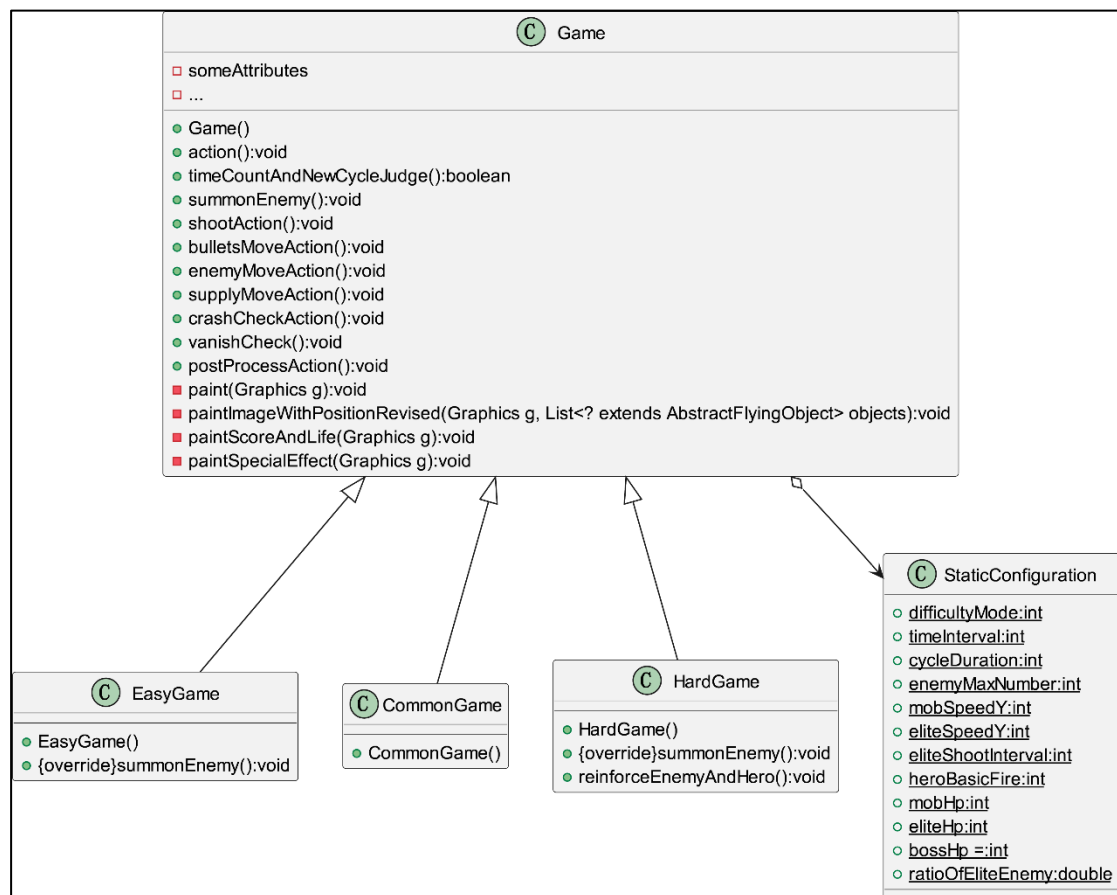
在三种游戏难度中，变化的参数放在 StaticConfiguration 中，由三种难度的游戏子类的构造函数来修改这些参数。

因素	简单模式	普通模式	困难模式
敌机数量最大值	5	7	9
敌机刷新频率及行动速率	较慢	中等	较快
敌机行动速率	较慢	中等	较快
敌机的射击频率	正常	正常	较快
普通敌机血量	30	30	$30 + 5 * \text{Boss 出现次数}$ (Max Value = 60)
精英敌机血量	30	60	$60 + 15 * \text{Boss 出现次数}$ (Max Value = 150)
Boss 敌机血量	不生成 Boss 机	1200	$1200 + 300 * \text{Boss 出现次数}$ (Max Value = 3000)
精英敌机刷新概率	50%	65%	75%

值得注意的是，为了增加游戏趣味性与平衡性，在困难模式中，英雄机也会随时间得到增强，具体为：

分数阈值	强化内容
500	基础子弹变为 2 颗
1000	提高子弹伤害（且更换了子弹贴图：  ）
1500	基础子弹变为 3 颗
2000	提高子弹伤害（且更换了子弹贴图：  ）
3000	提高子弹伤害（且更换了子弹贴图：  ）

3. 设计模式结构图



1. 静态设置参数类 StaticConfiguration:

用于存储一些会在不同难度中有所不同的参数，这些参数将在不同难度游戏的构造函数中被修改。

2. 模板父类 Game:

在 `action()` 中，规定了游戏的流程。`summonEnemy()` 等某些函数将在子类中被重写。

3. 模板子类 EasyGame, CommonGame, HardGame (以 Hard 为例):

`HardGame()` 修改了 `StaticConfiguration` 中的难度参数，`summonEnemy` 重写了父类的方法，使其能够在每次产生 Boss 时强化敌机，`reinforceEnemyAndHero()` 使得英雄机和敌机能随时间得到强化。

3 收获和反思

本次实验中，我深刻体会到了面向对象编程的三大特性：封装、继承、多态的优越性，并且在各种设计模式的实验中灵活运用了它们，这使我的编程思维和能力得到了极大的提升，也深化了我对 Java 这门语言的理解。

同时，我的编码中也存在一些不足，希望老师能够批评指正。感谢各位实验老师一直以来的帮助，祝好。