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

实验报告

开课学期：2022 春季

课程名称：面向对象的软件构造导论

实验名称：飞机大战游戏系统的设计与实现

实验性质：设计型

实验学时：16 地点：

学生班级：计算机类 4 班

学生学号：200110428

学生姓名：杨杰睿

评阅教师：

报告成绩：

实验与创新实践教育中心制

2022 年 4 月

1 实验环境

操作系统：Windows 10
主要开发工具：IntelliJ IDEA 2021.3.2

2 实验过程

2.1 系统功能分析

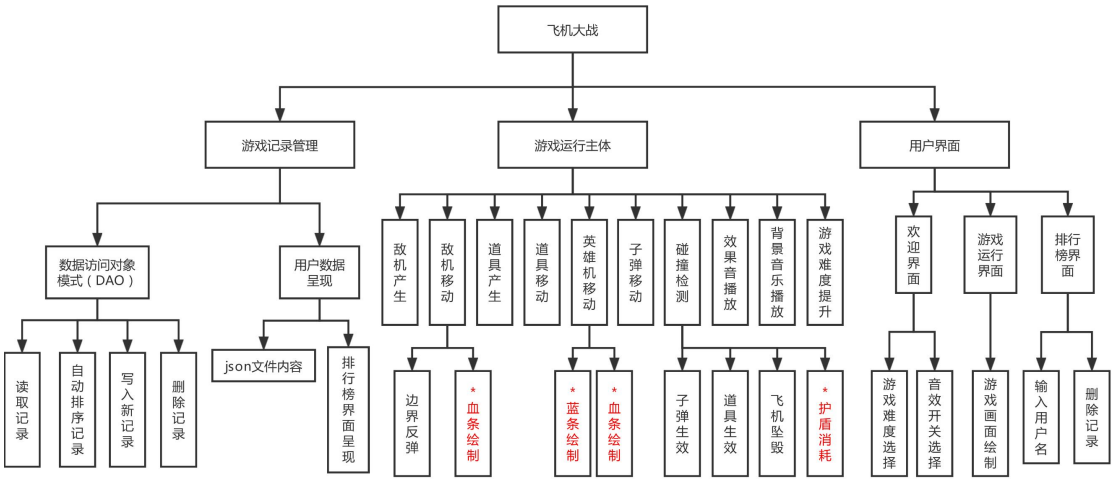

图 1：飞机大战功能层次图

飞机大战系统主要由三个部分组成：游戏记录管理，用户界面，以及游戏运行主体部分。

用户界面提供了游戏内部与用户进行交互的接口，用户通过选择游戏难度、音效，在游戏界面通过鼠标移动控制英雄机移动、攻击敌机，以获取道具并取得得分。

游戏主体部分详述了游戏的主要功能，除基本功能外，添加血条和蓝条绘制以获得更有交互式的游戏体验。添加加血道具的护盾效果，提供了游戏的可玩性。

游戏记录管理部分用于对游戏结束后的玩家和得分进行记录，在排行榜中查看得分以及排名。数据以 json 格式保存，符合了常见配置数据的处理方式，便于阅读和程序处理。

2.2 类的继承关系分析

本部分提供了 2 副 UML 类图，第一幅图为所要求的的 PlantUML 所绘制，但略显零乱；第二幅图为 IDEA 自带 Diagram 工具绘制，更加规整，易于辨识。

AbstractFlyingObject 是所有飞行物品的抽象父类，直接继承的子类有 AbstractAircraft 、 AbstractProp 、 BaseBullet 。 AbstractAircraft 分别由

HeroAircraft 和 AbstractEnemy 继承，其中后者是三种敌机类的抽象父类。

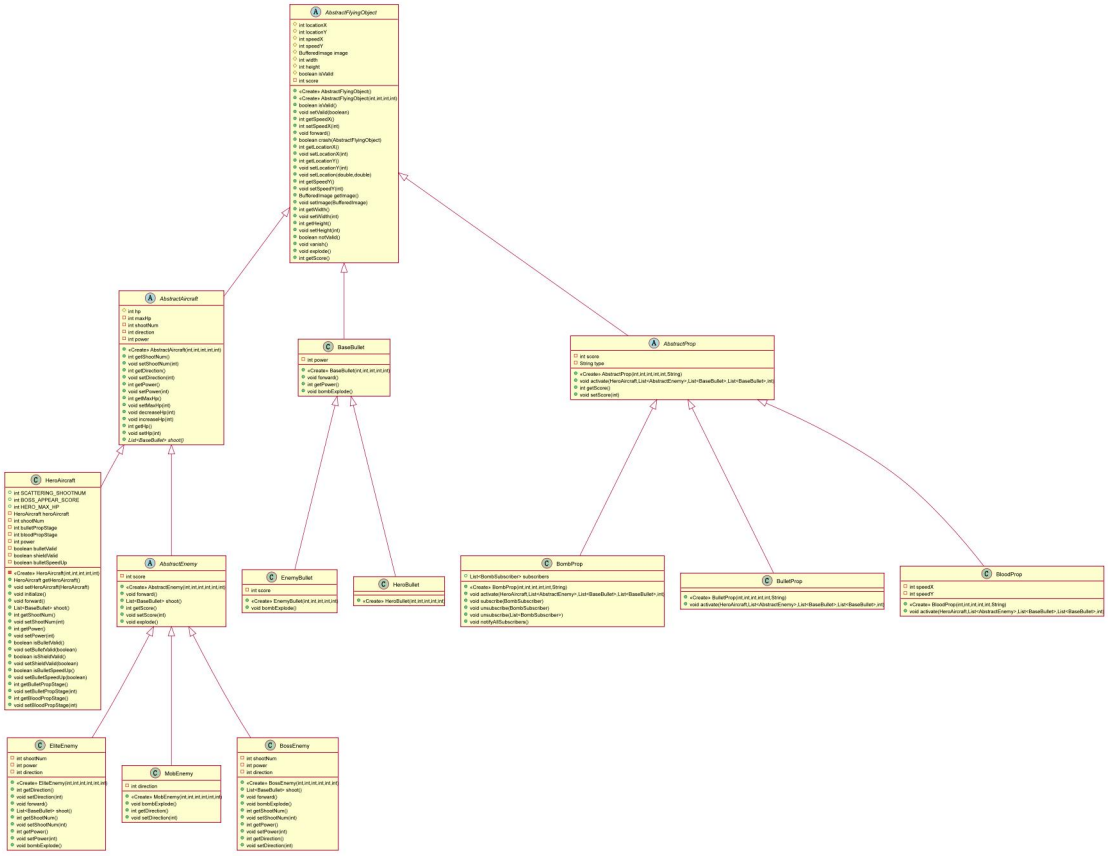

图 2：图为 PlantUML 所绘制继承关系类图

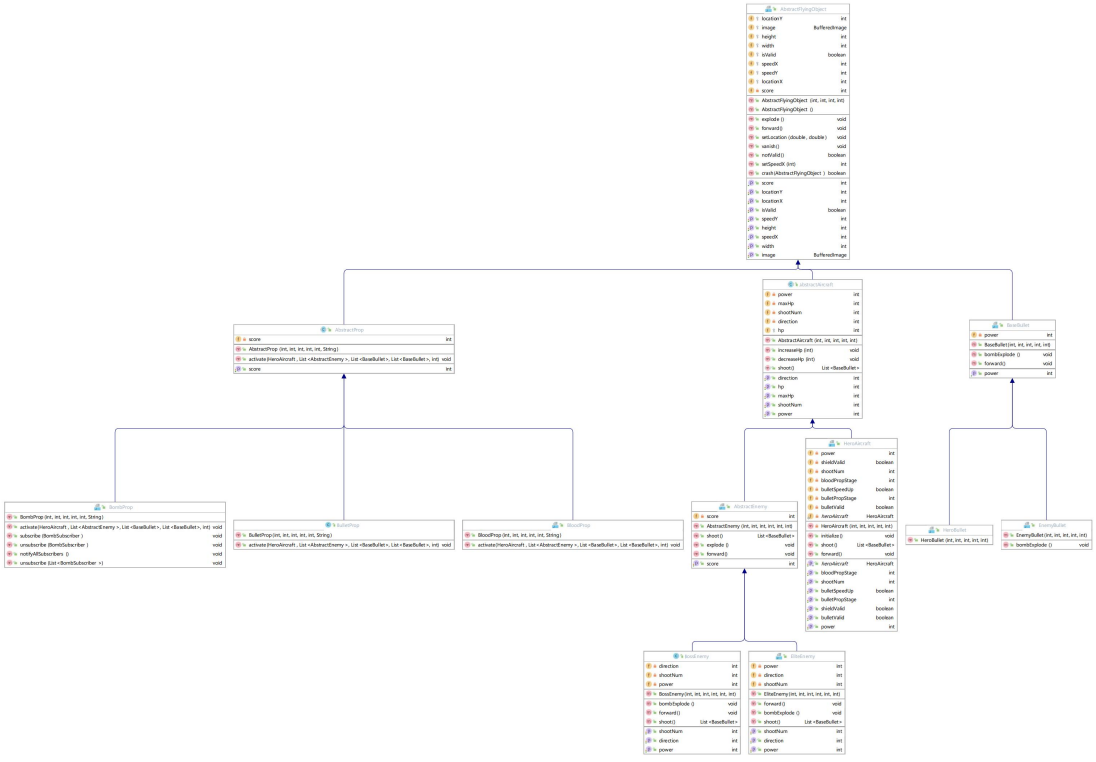

图 3：图为 IDEA 自带 Diagram 工具绘制的继承关系类图，视觉效果更加规整

2.3 设计模式应用

2.3.1 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。这种模式涉及到一个单一的类，该类负责创建自己的对象，同时确保只有单个对象被创建。这个类提供了一种访问其唯一的对象的方式，可以直接访问，不需要实例化该类的对象。

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）。

实际问题是，在只允许有一个实例的情况下，原本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。

使用该模式，其优势在于在内存中只有一个实例，减少了内存的开销，与此同时避免了对资源的多重占用。当然也存在一定的缺点，因为没有接口，不能继承，与单一职责原则冲突，对于一个类而言，应该只关心内部逻辑，而不关心外面怎么样来实例化。

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，本人采用的是懒汉式实现的单例模式。

2. 设计模式结构图

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。

UML 类图如下所示：

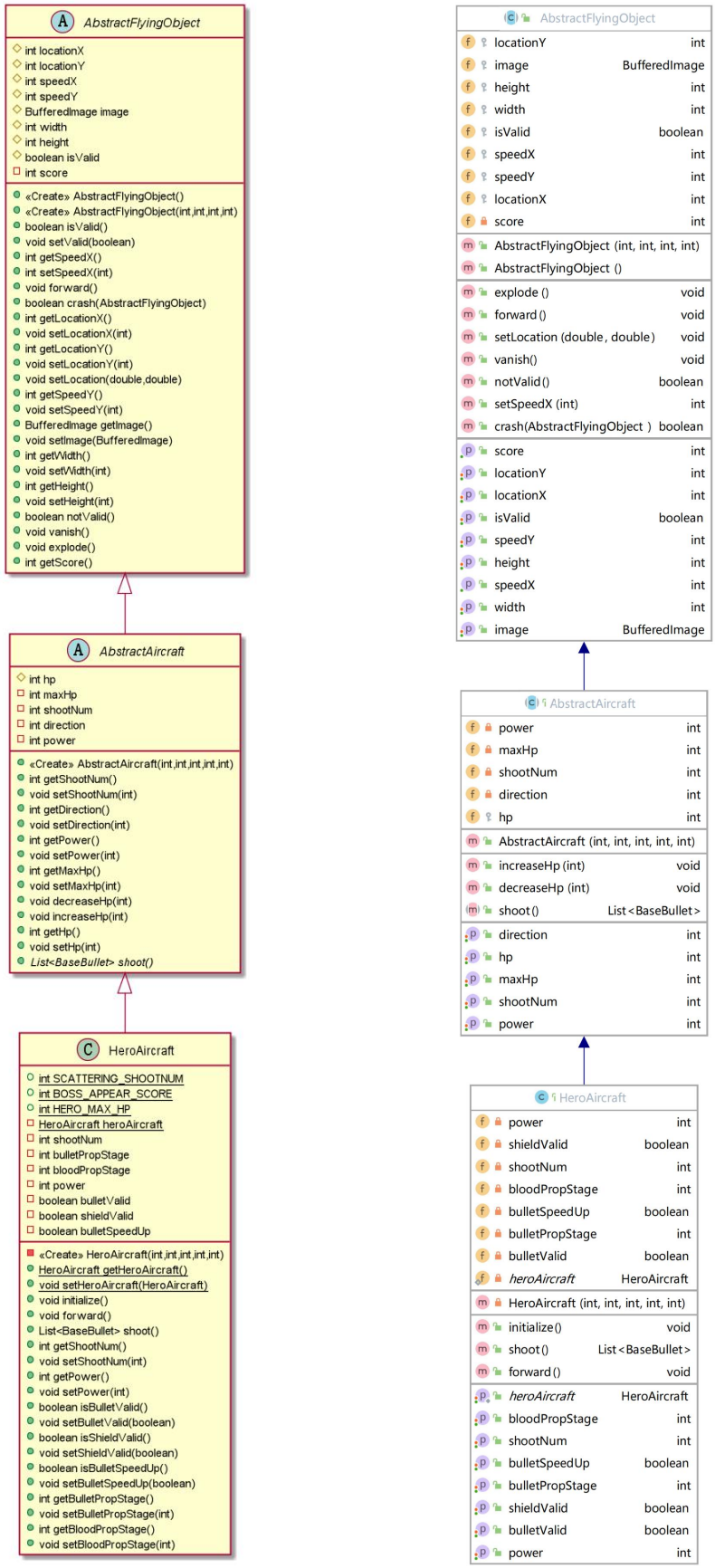

图 4：左图为 PlantUML 所绘制英雄机单例模式类图，右图为 IDEA 自带 Diagram 绘制类图

2.3.2 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，是 Java 中最常用的设计模式之一。在工厂模式中，我们在创建对象时不会对客户端暴露创建逻辑，并且是通过使用一个共同的接口来指向新创建的对象。

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 敌机不向下移动，在屏幕上方横向来回移动直至被消灭。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，而原本代码将道具和敌机的生成直接硬写到原始 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游戏逻辑执行的类中。

工厂模式的优势在于，对于调用者而言可以屏蔽产品的具体实现，调用者只关心产品接口，调用时只需要知道对象的名称，同时可扩展性高，如果再后续游戏过程中想要添加一种道具，只要扩展一个工厂类就可以，对游戏本身的代码结构没有影响。

2. 设计模式结构图

下方第一幅图是道具的工厂模式类图。

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，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)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。

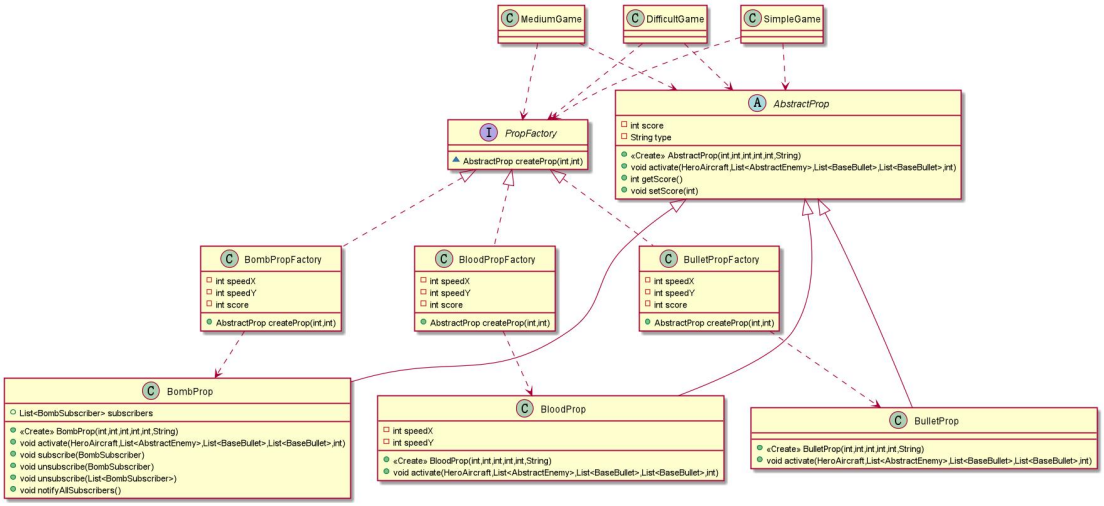

图 5：道具工厂模式类图

下方第二幅图是敌机的工厂模式类图，定义 EnemyFactory 的接口，将敌机的构造函数统一实现在接口下 createEnemy(double)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，实现了工厂模式。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，三个工厂子类实现了 EnemyFactory 的接口，工厂与接口间是实现关系。

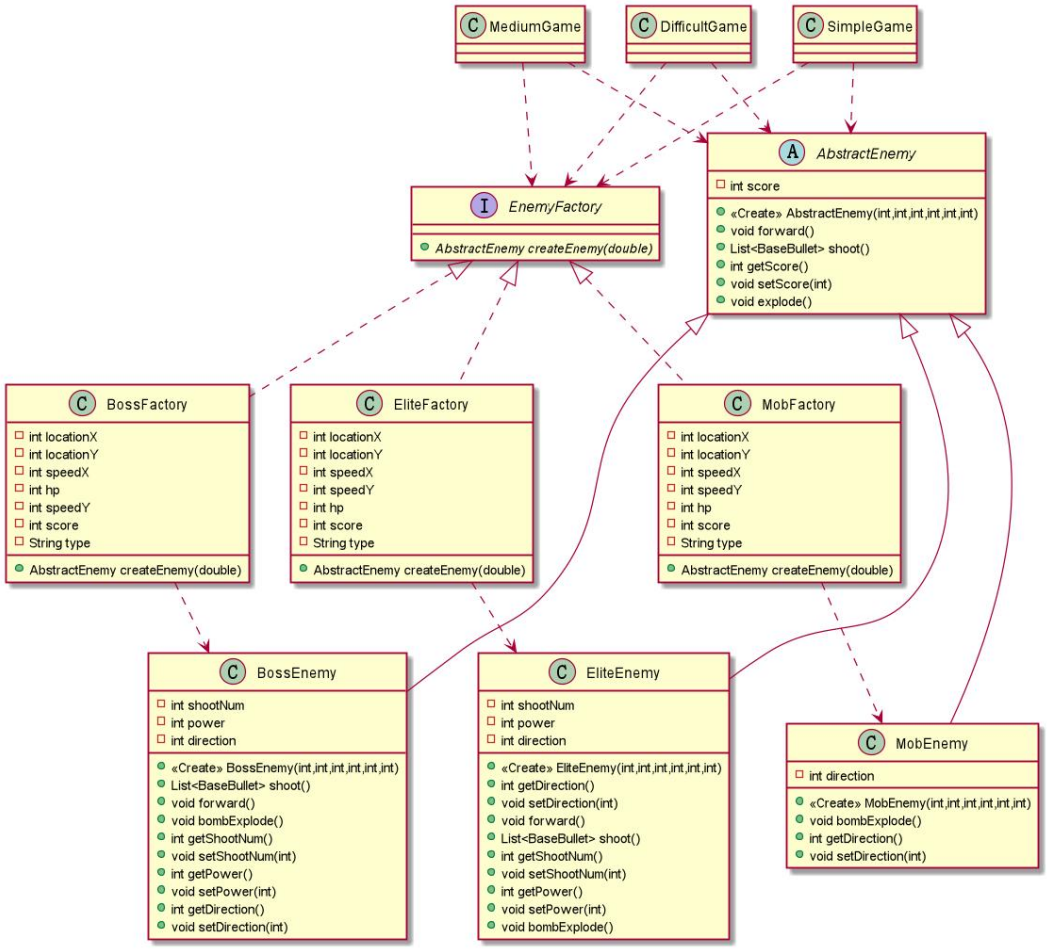

图 6：敌机工厂模式类图

2.3.3 策略模式

1. 应用场景分析

策略模式（Strategy Pattern）是一种行为型设计模式，它能让你定义一系列算法，并将每种算法分别放入独立的类中，以使算法的对象能够相互替换。

本例中，原来的代码直接由飞机调用 shoot 方法，无法使不同的射击方法分开，不利于后续添加弹道以及弹道代码的复用，原代码中（未删除，但本例实验中未调用）的 shoot 方法虽然实现了直射和散射的功能，但较为臃肿，采用策略模式能够使得代码模块化，同时不同策略的代码能够复用到不同类型的飞机上，提高了代码的复用性。

使用策略模式的优点在于，能灵活的切换算法而不需要额外的代码逻辑修改，同时避免了使用多重条件的判断，具有良好的可扩展性。

2．设计模式结构图

使用策略模式完成代码编写，uml 图如下：

Game 类：

策略模式外层的调用类，作为 client 存在

Context 类：

关键属性：

shootStrategy：用于记录当前条件下需要执行的策略类型，在本例于 Game 类中实例化了 enemyShootStrategy 和 heroShootStrategy，用于根据条件改变飞行器射击的方式

关键方法：

executeShootStrategy：用于执行对应的飞行器发射子弹的方法，具体而言通过将飞行器引用传入本方法，再根据飞行器类别（hero，enemy）、Context 的属性（即 shootStrategy 属性）来执行相应的射击方法。

setShootStrategy：用于设置 Context 的属性，由 Game 中传入的参数进行设置

ShootStrategy 接口：

关键方法：

shoot：射击方法的接口，对于不同的射击方法拥有相同的参数和返回值

ScatterShoot 类：

关键方法：

shoot：散射方法的具体实现，implement 了 ShootStrategy 接口，具体的实现而言就是在子弹上添加横向的速度，默认子弹初始位置构成抛物线

StraigtShoot 类：

关键方法：

shoot：直射方法的具体实现，implement 了 ShootStrategy 接口，具体的实现为固定子弹的初始位置为关于飞机中轴坐标对称的抛物线

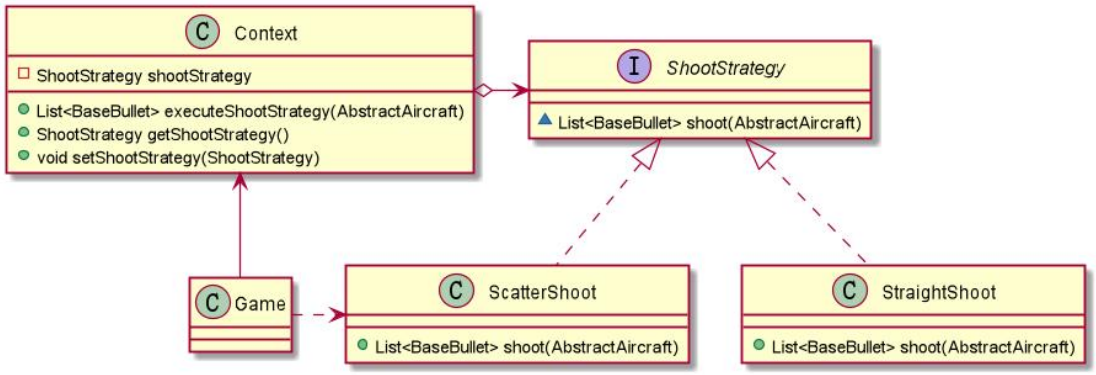

图 7：射击的策略模式类图（Game 这里沿用了此前未使用模板模式的结构，目的在于简洁呈现本部分策略模式的结构）

2.3.4 数据访问对象模式

1. 应用场景分析

数据访问对象模式（Data Access Object Pattern）用于把低级的数据访问 API 和操作从高级的业务服务中分离出来。

本例中因为需要将排行榜每轮游戏结束时进行更新，需要使用此模式将数据的访问部分和游戏上层的结构分离出来，有利于模块化的管理和流程的控制。

本例中使用的 DAO 模式，采用 json 保存 java 对象 Record 类，Record 类的读取由 DAO 管理，使得不同的功能相分离，提高了代码的复用性和封装性，有助于对代码上层结构的修改而不改变底层的读写逻辑。

2. 设计模式结构图

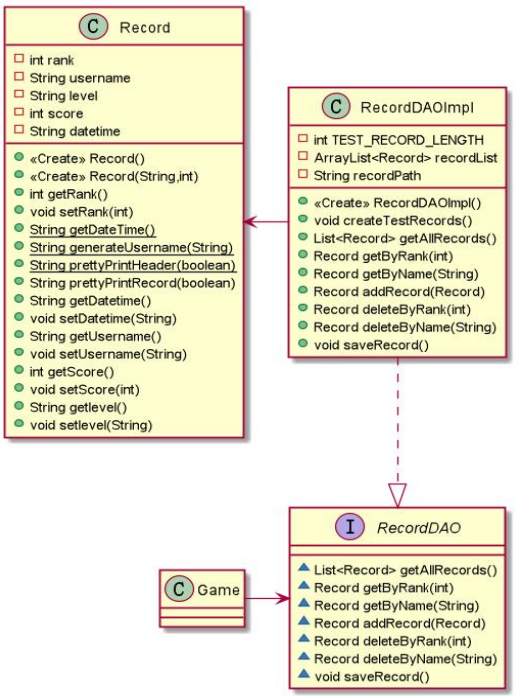

图 8：数据访问对象模式类图

使用 DAO 模式构建底层读写逻辑，puml 类图如上所示：

Game 类：

操作 DAO 模式的上层逻辑，游戏结束后将得分记录写入 record.json 保存

Record 类：

用于记录游戏得分情况的类，提供了展示数据，写入数据的方法。

关键属性：

- rank：为了方便 DAO 完成数据的读取，添加了在增删数据时排序以获得准确 rank 值
- username：输入的用户名
- level：选择的难度，当前全为“MEDIUM”
- score：游戏结束时的得分
- datetime：当前的系统时间

关键方法：

- 空参构造函数：构造一个随机指定的记录，通常没有作用只是用于测试时填充数据
- 含参构造函数：根据用户名和得分以及当前游戏难度，获取系统时间得到当前记录的属性（不含 rank）
- 静态方法
  - getDateTime：获取当前系统时间的格式化字符串表示，注意与实例的方法重名，当然不影响使用。
  - generateUsername：随机数和当前系统时间组成的 String 对象 SHA256 值的前 8 位作为随机生成的用户名，仅用于测试
  - prettyPrintHeader：打印记录的 header，控制了精确的间隔以呈现较好的格式
- prettyPrintRecord：打印记录，因前提是构建了实例，故为非静态方法，控制了精确的间隔以呈现较好的格式。

RecordDAO 接口（详细描述见 RecordDAOImpl）：

关键方法：

- `List<Record> getAllRecords();`
- `Record getByRank(int rank);`
- `Record getByName(String name);`
- `Record addRecord(Record record);`
- `Record deleteByRank(int rank);`
- `Record deleteByName(String name);`
- `void saveRecord();`

RecordDAOImpl 方法：

实现了 RecordDAO 接口，用于完成数据的访问（读写、排序、删改操作等）

关键方法：

- `List<Record> getAllRecords()`
  返回记录的 ArrayList 引用，默认使用 Record 实例的 prettyPrintRecord 方法打印每一条记录
- `Record getByRank(int rank)`
  返回按排名查询得到的记录，需要排序获得正确的得分顺序（效率更高的做法应当是构建一个排序标记，只要无序状态时读取才进行排序）
- `Record getByName(String name)`

返回按用户名查询所得记录。（此处未进行排序，非正确做法。）

- `Record addRecord(Record record)`
  添加一条记录，因需要写入 rank 属性，故需要进行排序，但非有效率做法。
- `Record deleteByRank(int rank)`
  根据排名删除一条记录，未进行排序。
- `Record deleteByName(String name);`
  根据用户名删除一条记录，未进行排序
- `void saveRecord();`

将 Java 对象写以 json 格式写入文件，此处使用了 gson 包来完成 json 的读写。

2.3.5 观察者模式

1. 应用场景分析

观察者模式（Observer Pattern）也是一种行为型设计模式，允许定义一种订阅机制，可在对象事件发生时通知多个“观察”该对象的其他对象。这种模式主要应用在对象间存在一对多的关系时，而这种关系十分普遍。

在本实验中，炸弹道具生效时，需要广播其作用到所有敌机类和敌机子弹，分别使各个对象完成在炸弹道具生效后的行为。原本设计时，没有专门思考到使用观察者模式进行实现，采用了类似观察者模式的方式，完成了程序设计，本次中将代码通过观察者模式重构，统一实现了观察者接口完成收到订阅后的行为。

使用观察者模式能使得观察者和被观察者是抽象耦合的，在本例中使用观察者模式建立的一套触发机制能有效的管理观察者的行为，提高了代码的可重用性。

2. 设计模式结构图

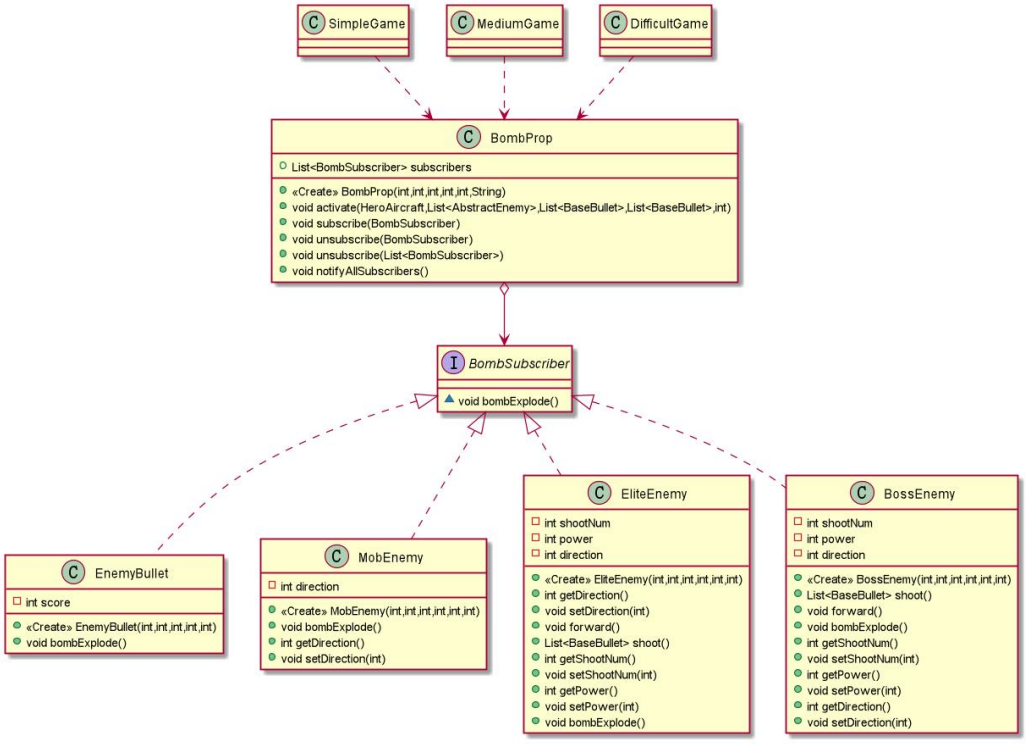

图 9：观察者模式类图，注意此处 Boss 机也实现了观察者接口

UML 类图如上所示，BombSubscriber 接口由需要接受炸弹消息的 4 个类分别实现，在 Game 类中，通过调用 notifyAllSubstribers()方法完成对炸弹消息的传播，借助原有的 activate 方法，完成 Subscriber 的订阅过程。BombProp 类内部添加了列表用于保存需要通知的订阅者，因原代码已经实现相关功能，此处为改用观察者模式重构。

2.3.6 模板模式

1. 应用场景分析

模板模式（Template Pattern）是一种行为型设计模式，它在抽象类中定义了一个算法的框架，允许子类在不修改结构的情况下重写算法的特定步骤。这种方式定义一个操作中的算法的骨架，而将一些步骤延迟到子类中，子类的重写定制化了操作的实现，但遵循了父类的执行步骤。

本次实验中，需要对游戏的难度进行划分，而不同的难度需要有不同的功能实现。原本的代码中，将难度的划分集成到一个 level 变量进行控制，对于不同难度的不同事件触发使用的是条件语句，这样提高了运行时的损耗，也使得代码较为冗长，不利于维护。

本次实验重写了此前难度控制的代码，使用 AbstractGame 控制游戏执行的步骤，同时将部分需要子类重写的方法延迟到子类进行实现，三种难度分属三个独立的子类，这样封装了不变的部分到抽象父类中，扩展了可变的部分并延迟实现到子类中，同时提取了公共代码，提高了可维护性，而已经由父类指定的流程控制，避免了子类中流程的错误，在流程异常时能较为方便的查错。

2. 设计模式结构图

重写了难度控制的代码，模板方法为 public final void action()，固定了程序执行的流程，多数可重用的方法略去不加详细阐述，以下仅介绍本人额外添加的方法：

- `public void gameOverCheck();`
  用于检测游戏结束时的音乐停止播放方法
- `public void paintEnemyLife(Graphics g);`
  用于绘制敌机生命条，Mob 敌机无血条，Elite 敌机血条悬浮在飞机上侧，Boss 敌机血条悬浮在屏幕最上沿，Boss 机存活时显现，死亡后消失
- `public void paintHeroAttributes(Graphics g);`
  用于绘制英雄机属性条，当前由两条细矩形绘制：下方为生命值条，可显示我方生命值和当前护盾有效时间，处于护盾有效时碰撞敌机扣除一定有效时间，碰撞 Boss 机扣除全部有效时间，在护盾失效后碰撞敌机本机坠毁；上方为蓝条，可随火力道具叠加而显示不同火力阶段可持续时间。

需要由子类重写的抽象方法如下所示，这些方法决定了难度的控制：

- `abstract public void aircraftsMoveAction();`
- `abstract public void generateEnemyAircrafts();`
- `abstract public void playBGM();`
- `abstract public void shootAction();`
- `abstract public void crashCheckAction();`

具体的难度控制表如下所示：

进度	内容	简单	普通	困难
√	Boss 敌机	无	有 血量固定	有 血量逐次提升
√	难度随时间增加	否	是	是
√	敌机最大数目	3(固定)	5	7
√	敌机初始速度	Mob：5 Elite：3	Mob：7 Elite：4	Mob：10 Elite：6
√	敌机初始血量	Mob：30 Elite：60	Mob：30 Elite：84 Boss：4000	Mob：30 Elite：120 Boss：8000
√	敌机产生概率	Mob：70% Elite：30%	Mob：50% Elite：50%	Mob：30% Elite：70%
√	Boss 敌机产生	不产生	每获得 500＊level 分	每获得 500*level/3 分
√	敌机越过本方防线扣分	不扣分	Mob：0.5 倍 hp Elite：0.75 倍 hp	Mob：0.75 倍 hp Elite：0.875 倍 hp
√	道具产生概率	Blood：33% Bomb：33% Bullet：33%	Blood：30% Bomb：30% Bullet：30%	Blood：20% Bomb：20% Bullet：30%
√	道具效果消退时间	333 单位	285 单位	222 单位
√	加血道具的护盾效果	碰撞敌机不减少持续时间	- 碰撞后道具有效时间减去敌机血量 -Elite 和 Mob 损毁，Boss 减少与剩余时间相关的血量 - 碰撞后道具有效时间至少保持 5 时间单位	- 碰撞后道具有效时间减去敌机血量 - 敌机减少与剩余时间相关的血量 - 碰撞后道具有效时间至少保持 5 时间单位

表 1：游戏难度控制细则

图 10：模板模式类图，游戏的主体逻辑在 AbstractGame 类中完成编写

3 收获和反思

本次 Java 实验主要是时间较为紧张，每周连续的实验任务在完成尤其是完成实验五的任务时将会十分的紧迫，所给截止时间往往是实验开始后的一周。所幸期间其他科目所给实验时间较为充裕，可以通过周转延后完成以及时完成 Java 实验任务。

实验代码评分要求不明确，虽提供了需要完成的实验要点，但对于具体要点的评判和所占分数对学生不可见，不利于学生在提交前自行评估自己的实验得分。

实验报告评分要求不明确，导致起初完成实验报告时过多的对于类图的分析而花费了大量的时间，同时缺乏具体的评分细则，不利于学生在提交时自行估计实验得分。

除此以外，对于面向对象程序设计国内外早已有较为完善的公开课程和实验资料，却采用独立设计课程的方式进行，虽也体现了本校教师的优秀教学水平，但缺乏对于经过打磨和反复测试的课程内容的参考，尤其是作业和实验部分。作业部分往往过于简单，或者习题与课程所学主体内容相去甚远，尤其是选择题部分，而实验难度参差不齐，实验五跨度极大，花费了大量的时间在自行摸索学习过程中，较为浪费时间，而这本应该是成熟的课程体系已经规避了的问题。国内外 OOP 相关课程很容易搜索到，比如 Standford-CS108。