



JULY 5, 2017

《解救饭堂》 游戏软件设计文档

V1.0

YINGHAO (MICHAEL) WANG

《解救饭堂》游戏开发团队




Table of Contents

1	项目文档简介.....	4
1.1	目的.....	4
1.2	软件概况	4
1.3	本文档组织	4
2	设计考量.....	4
2.1	假设.....	4
2.2	约束条件	4
2.3	系统环境	4
2.4	设计方法论	4
2.5	可能的设计风险	4
3	架构设计.....	4
3.1	架构概况	5
3.2	塔楼建造逻辑设计	5
3.2.1	用户点击建造位置	6
3.2.2	用户点击建造一种塔	6
3.3	塔楼与炮弹关系设计	7
3.4	敌人对象的生成，管理，伤害与击退	7
3.4.1	关系类图.....	7
3.5	游戏伤害，收益，结束条件	9
3.6	Buf（负面效果）设计.....	10
3.6.1	关系图.....	10
3.6.2	顺序图（以 BleedBuf 举例）	10
3.7	血条实现与管理	10
3.8	游戏结束	11
4	用户交互设计.....	12
4.1	玩家引导	12
4.2	交互元素风格.....	12
5	安全性考虑	12

修改记录

版本	姓名	修改原因	日期
1.0	王颖豪	Initial Revision	07/05/2017

确认及许可

姓名	签名	日期
王颖豪	Wang Yinghao	23/06/2017

1 项目文档简介

《解救饭堂》是一款定位在 PC 端的塔防类游戏。玩家通过在游戏中固定的位置建造餐厅，向下课同学们投掷食物来增加学生的饱腹感。吃饱了的学生就不会一拥而上围堵食堂了。

1.1 目的

本文档的根本目的在于细化说明软件需求说明书中规定的设计内容。本文档将从设计的角度详细阐述本游戏的设计思路和部分实现细节。

1.2 软件概况

本软件开发环境为 **Unity 5.6**。开发语言为 **C#**。未使用其他插件。根据需求文档，本软件的部署环境为现代 **Windows** 与 **Macintosh** 系统。

1.3 本文档组织

本文档将分三部分来介绍本游戏的设计概况：项目简介，设计考量，设计架构。

2 设计考量

这部分将根据游戏的开发环境，开发限制，需求文档中的限制来说明软件设计需要考虑的因素。

2.1 假设

玩家具备操作计算机的基本技巧，并能理解常见的游戏提示。

2.2 约束条件

无

2.3 系统环境

本游戏支持在现代 **Windows** 与 **Macintosh** 操作系统平台上运行。要求玩家配置有基本的现代显示驱动。

2.4 设计方法论

MVC 框架与基于组件的编程方法

2.5 可能的设计风险

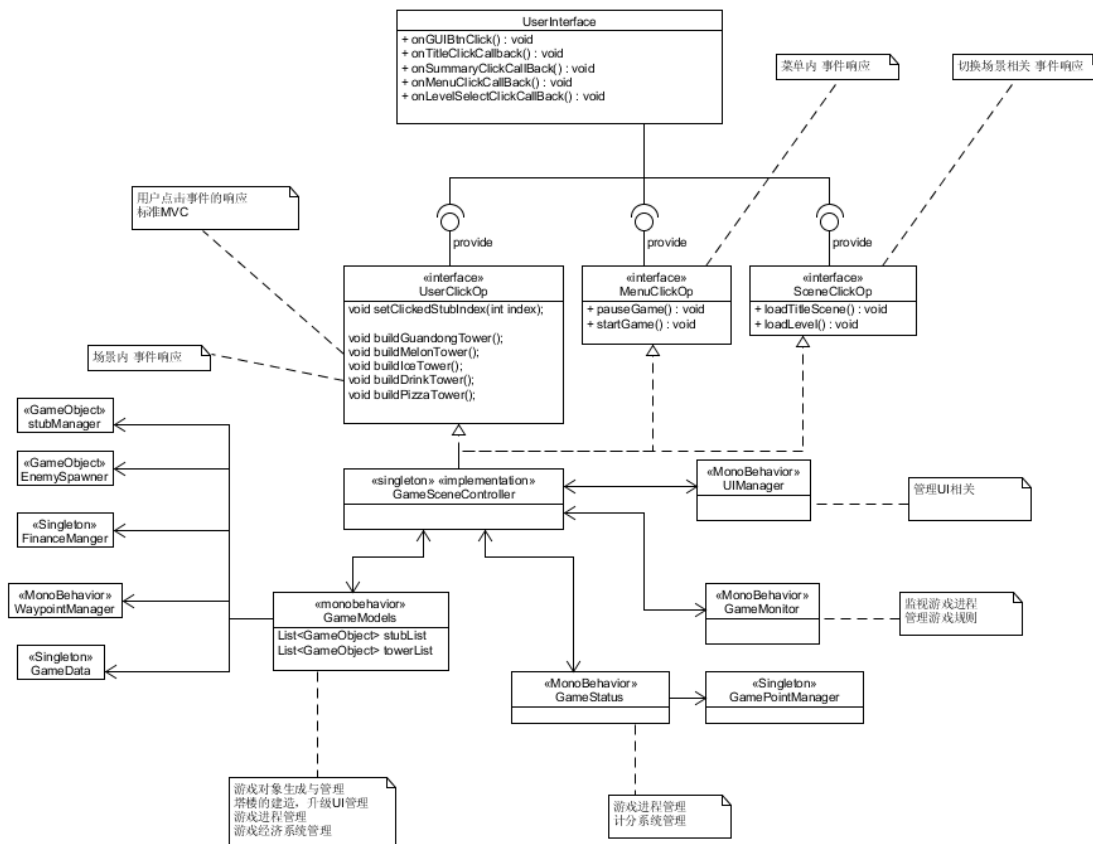
无

3 架构设计

本游戏的设计基于 **MVC** 框架来处理用户的输入。利用基于组件的编程模式来实现需要引擎驱动的功能。

3.1 架构概况

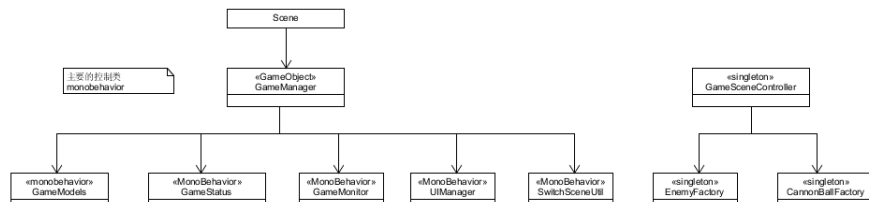
游戏主要控制类关系类图如下:



该设计的核心为 **GameSceneController** 控制器。该控制器作为 MVC 框架中的控制器单元，将用户的输入响应传递给下级的模型单元。

该控制器也很大程度上担任游戏的逻辑中枢，主要的 `Monobehavior` 之间的交互通过 `GameSceneController` 提供的接口来实现。目的在于减少 `GetComponent` 的使用，提高性能。

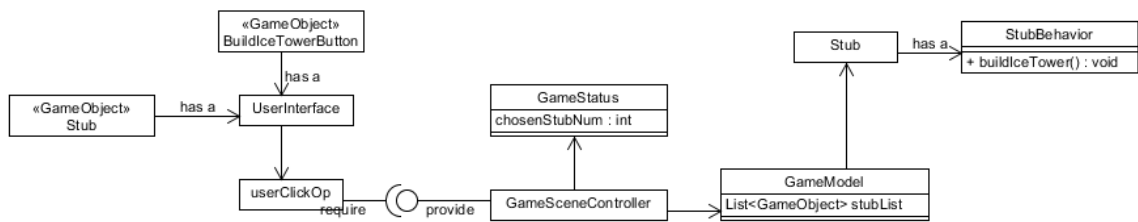
该控制器一定程度担任游戏单例对象的生命周期管理。见下图。



游戏主要控制类别的 `MonoBehavior` 的驱动，通过挂载在永久的游戏对象 `GameManager` 上实现。

3.2 塔楼建造逻辑设计

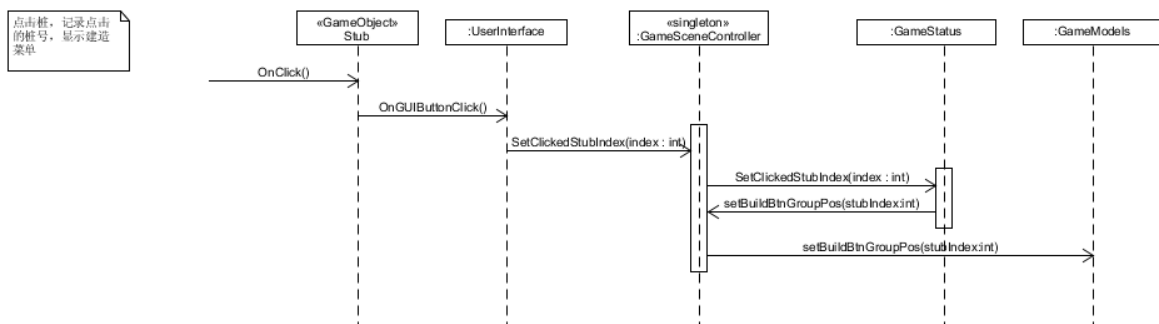
与塔楼建造相关的类图:



本图非标准 UML 类图。主要区别如下：为表示一个 **GameObject** 下挂载了这个组件，我们使用实线箭头加“has a”标注来表示，下同。

3.2.1 用户点击建造位置

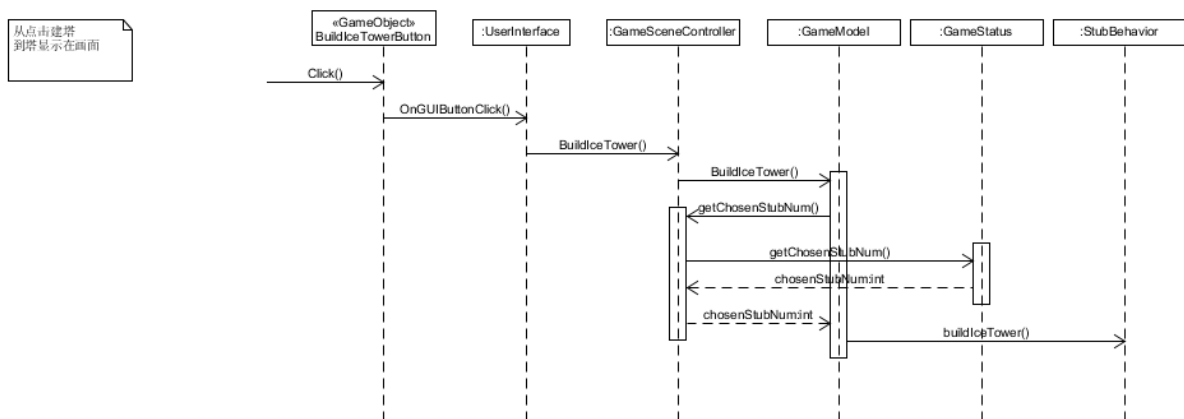
用户点击建造位置后，需要记录下当前点击过的位置并显示建造菜单。这部分的顺序图如下：



后置条件：点击过的建造位置被记录；建造菜单显示在建造的位置上。

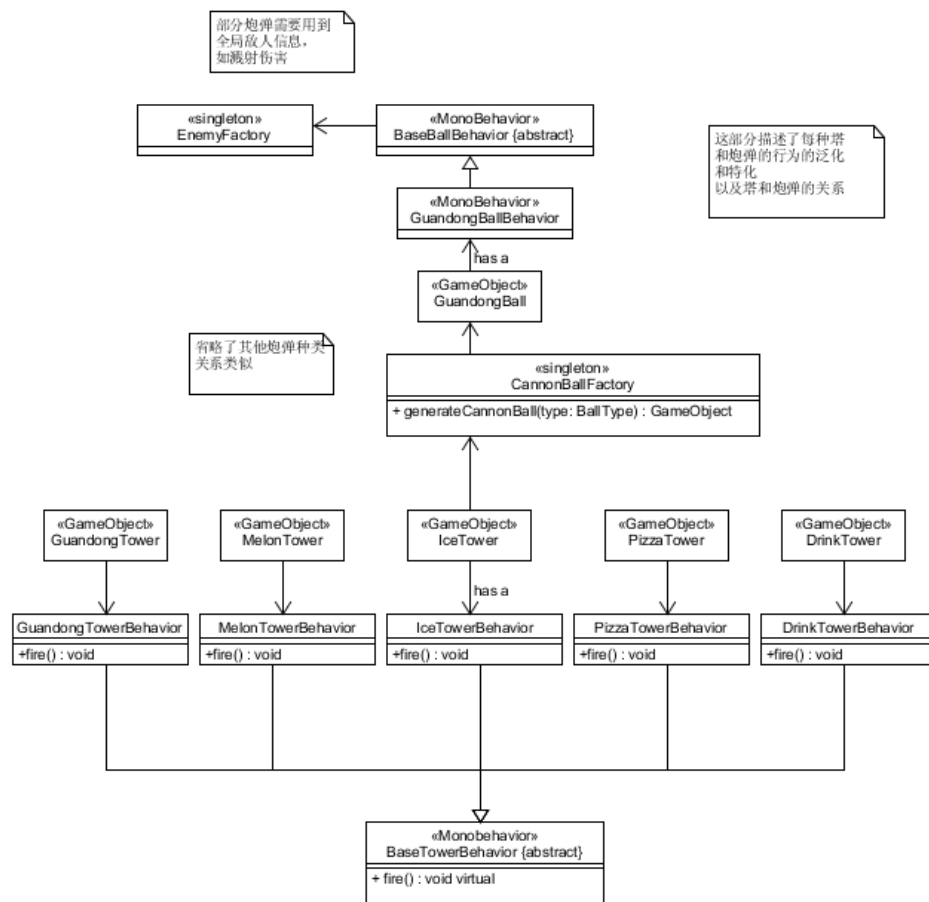
3.2.2 用户点击建造一种塔

用户点击建造某一种塔后，需要将该位置的桩的游戏对象替换为用户需要建造的塔的对象。这部分的顺序图如下：



后置条件：建造桩的对象被用户需要建造的塔所替代。

3.3 塔楼与炮弹关系设计

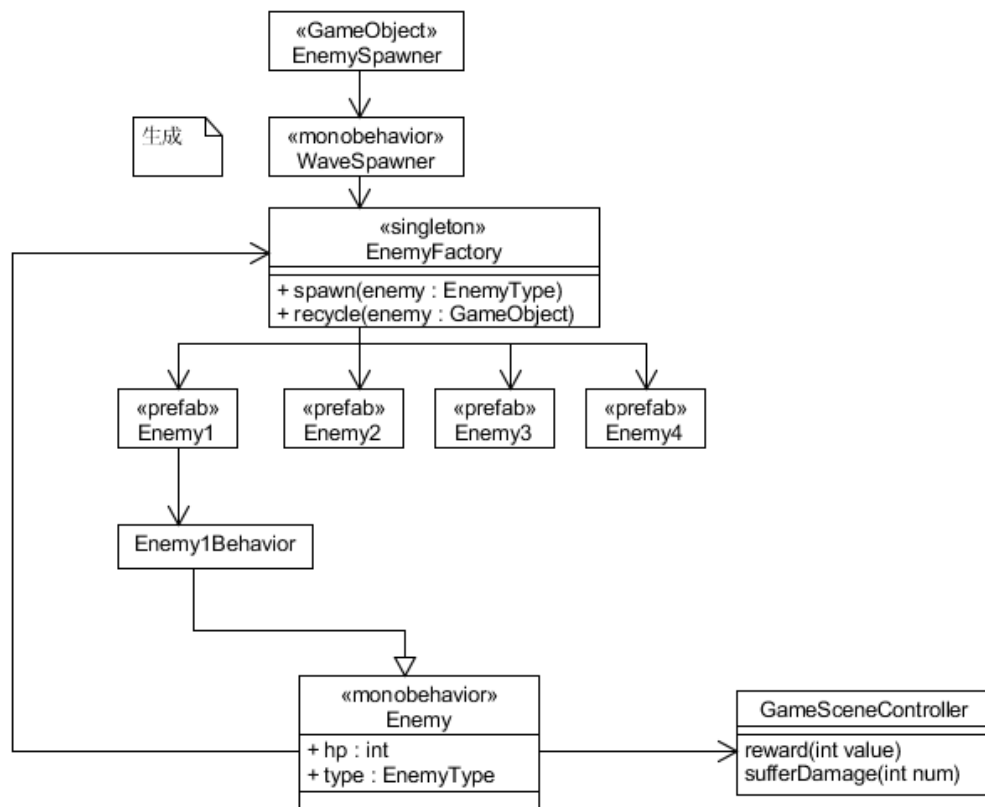


塔楼都有一个特化的 **Behavior** 组件，这个组件里重载了 **fire** 方法，因而每种塔楼都有自己独特的发射炮弹的方法。所有的 **TowerBehavior** 均继承 **BaseTowerBehavior** 基类。

与塔楼设计类似，炮弹也有特化的 **Behavior** 组件，也均继承 **BaseBallBehavior** 基类。

3.4 敌人对象的生成，管理，伤害与击退

3.4.1 关系类图



敌人对象的生成通过场景中永久挂载的游戏对象 **EnemySpawner** 上的 **Monobehavior – WaveSpawner** 以及单例 **EnemyFactory** 来实现。

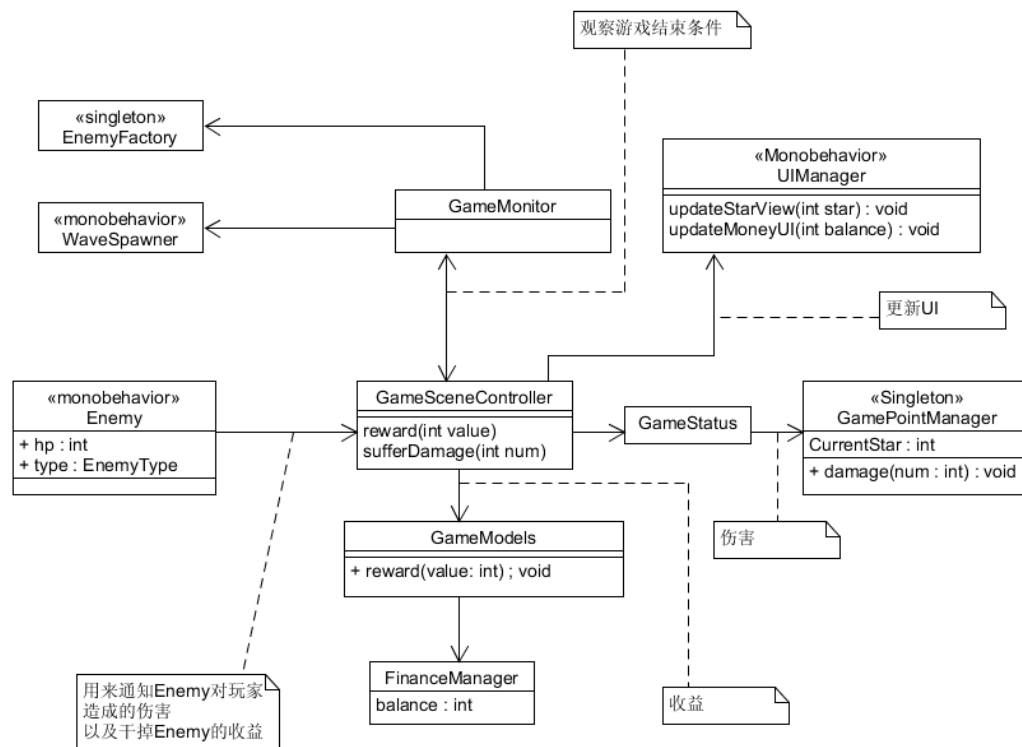
WaveSpawner 管理生成的节奏。

WaveSpawner 调用 **EnemyFactory**，生成并重复利用 **Enemy** 对象。工厂模式提高性能。

每一个 **Enemy** 挂载特化的 **Behavior**，因而可以定制每种 **Enemy** 的行为特点。他们继承基类 **Enemy**。

Enemy 能向 **GameSceneController** 发送消息以通知该 **Enemy** 对玩家造成的伤害，或 **Enemy** 被干掉时玩家获得的收益。

3.5 游戏伤害，收益，结束条件



GameSceneController 通知 GameModels 处理收益相关事项。通知 GameStatus 处理玩家生命值（剩余星数）事项。
每次玩家遭受损伤或者获得收益，GameSceneController 通过 UIManager 更新 HUD 显示。

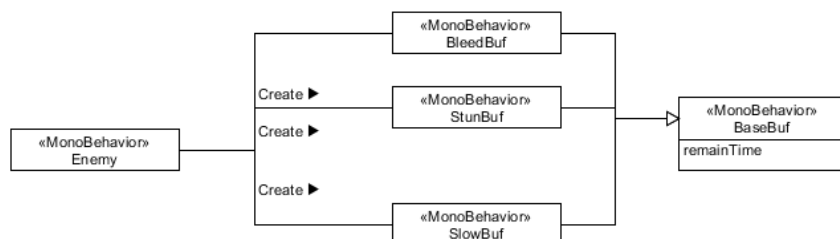
游戏对象 GameManager 上挂载永久脚本 GameMonitor，记录该关剩余时间，通过监察 WaveSpawner 和 EnemyFactory，当满足游戏结束（胜利或失败）通知 GameSceneController。

游戏胜利与失败条件：

当前关卡时间结束，玩家生命值不为 0	胜利
所有波数生成结束，所有敌人被消灭，玩家生命值不为 0	胜利
玩家生命值为 0	失败
临界条件：玩家生命值为 0，关卡时间刚好结束	胜利

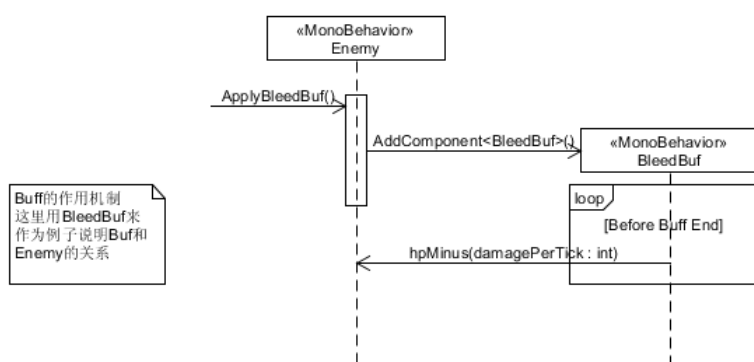
3.6 Buf（负面效果）设计

3.6.1 关系图



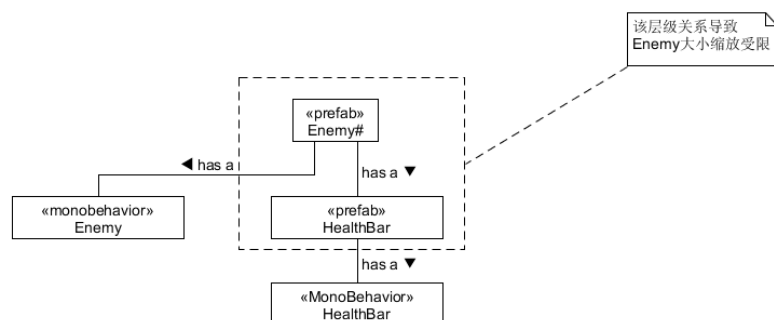
当敌人受到负面效果时，动态创建 Buf Monobehavior。

3.6.2 顺序图（以 BleedBuf 举例）

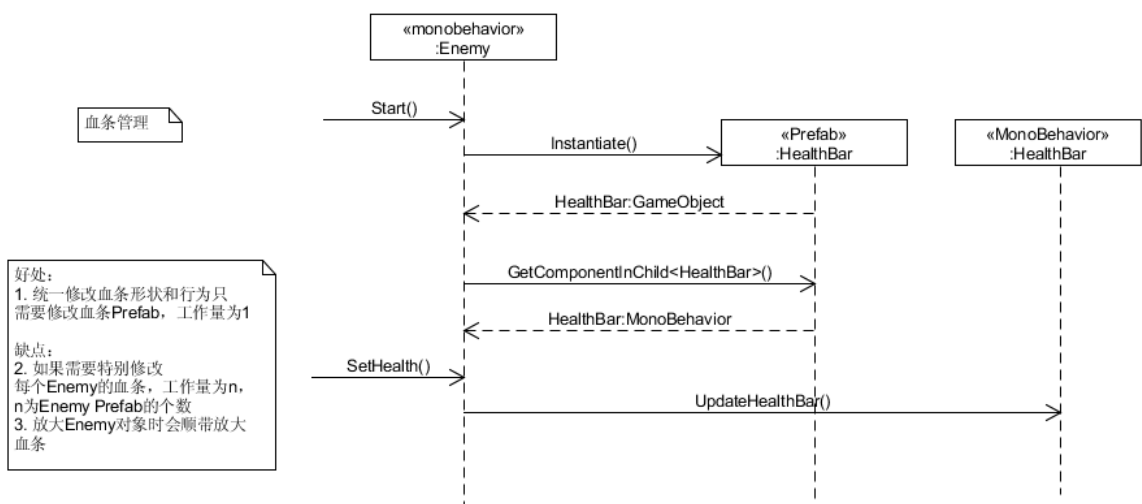


（炮弹）作用在 `Enemy` 上，炮弹本身具有 `BleedBuf` 的效果，就会调用 `ApplyBleedBuf()` 方法，该方法动态创建 `BleedBuf MonoBehavior`。该 Buf 的效果是固定时间削减 `Enemy` 的生命值。

3.7 血条实现与管理

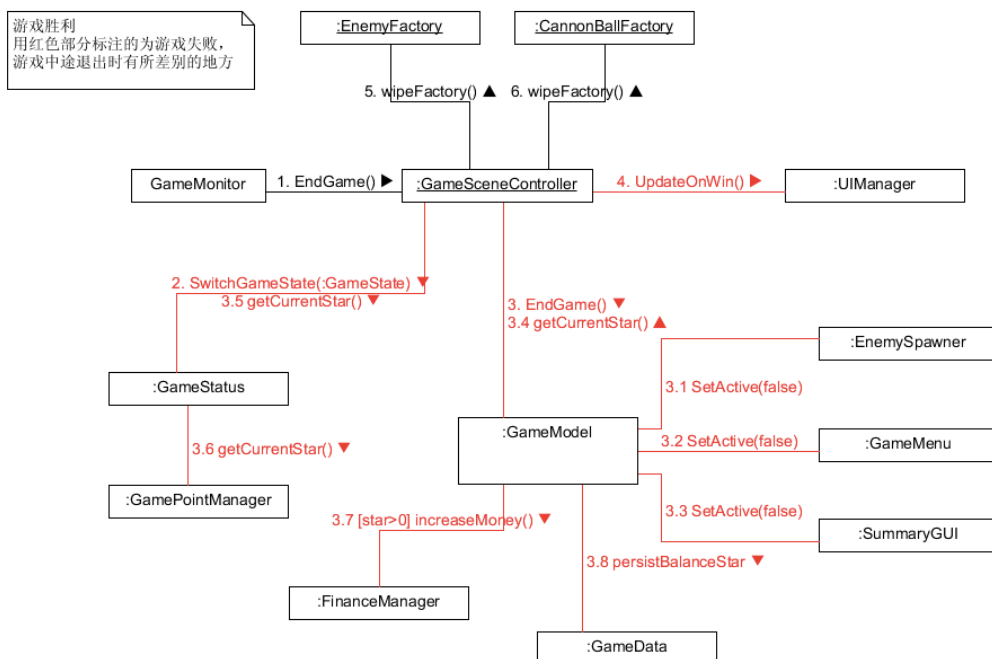


我们将血条设置为 `Enemy` 的一个子对象。这样血条对象能跟着 `Enemy` 移动。血条对象是动态创建的。由于不同的 `Enemy` 对象共用同一个血条对象，因此用代码控制只需一行。



可以看到，Enemy 对象中长期持有 MonoBehaviour HealthBar 的一个实例，因此能随时更新。

3.8 游戏结束



由于涉及对象过多，这里采用 Communication Diagram 来代替 Sequence Diagram。当 GameMonitor 发出游戏结束请求时，GameSceneController 需要做三件事情：

1. 通知 UIManager 更新 UI
2. 通知 GameModel 进入游戏结束逻辑
3. 清空工厂类中的游戏对象（否则会因访问销毁的游戏对象而出现异常）。

4 用户交互设计

需求文档中并未明确指出用户交互设计的定义。因此，本文档将根据通用的用户交互设计准则，以及项目实际开发约束来给出交互设计的定义。

4.1 玩家引导

本游戏应具备基本的故事背景介绍，应尽可能早将故事背景显示给用户阅读。

本游戏从启动到进行游戏，到退出游戏，都应当有清晰的用户交互元素供用户满足其目的。“清晰”的定义为：具备基本计算机操作技巧的用户，在不需额外说明的情况下，能自行使用。

4.2 交互元素风格

本游戏所有交互元素应具备统一的交互元素风格。

5 安全性考虑

本游戏无特殊安全性考虑。

本游戏无外部数据访问，因此也无特殊数据安全性考虑。