Е. А. Нестерова

# ПРИМЕНЕНИЕ ПРИНЦИПОВ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ SOLID В UNITY-РАЗРАБОТКЕ

*В работе приводится описание применения принципов SOLID для решения практических задач разработки игры на движке Unity.*

## Введение

В начале 2000-х годов известный консультант и автор в области разработки ПО Роберт Мартин обозначил пять основных принципов объектно-ориентированного программирования и проектирования, также известных как SOLID. Принципы SOLID применяются как при создании программных систем, так и в работе над уже существующим ПО с целью его улучшения. Внедрение указанных принципов получило широкое применение в разработке на Unity, которая на сегодняшний день является одной из популярнейших сред для разработки компьютерных игр, поддерживающей до 27 платформ.

## I. Распространенные проблемы на пути разработки кода

После получения заказа на разработку игры по прошествии некоторого времени разработчик неизбежно сталкивается с этапом правок от заказчика. И зачастую именно в последствии внесения изменений в коде проявляются его "слабые места". Для наглядности приведем образный список нередко проявляющихся проблем:

- "Создание нового класса повлекло за собой чрезмерную множественность изменений остального кода".
- "Переиспользование интерфейсов оказалось слишком времязатратным".
- "Изменение 1 строчки кода привело к появлению десятка багов".
- "Внедрение новых фитч привело к "поломке"старых".
- "Расширение функционала классов "сломало"их".

В результате общая картина с исходным кодом такова, что его любые, даже самые незначительные изменения в одном месте неизбежно оказывают последствия на весь остальной код и, каждый раз внося изменения в какую-то отдельную его часть, мы рискуем "сломать"все остальное. Как следствие, исходный код становится практически непригодным для тестирования.

## II. Решение проблем кода с использованием принципов SOLID

Решать упомянутые проблемы гораздо проще и быстрее, если структура исходного кода изначально предполагает возможность отдельного модульного тестирования (separate unit testing). И эту возможность нам дает внедрение в разработку принципов SOLID. На примере вышепредставленного списка приведем примеры решения задач с использованием данных принципов.

### Проблема множественности изменений кода

Изначально у нас был лишь один публичный класс Shot("Выстрел"), наследуемый от MonoBehaviour, содержащий переменную Damage("Урон"). Однако в последствии заказчик захотел, чтобы вариантов стрельбы было больше: чтобы стрелять можно было не только из пистолета, но и из боевой установки, расположенной на борту вертолета. Появляется новый класс HelicopterShot("Стрельба с вертолета"), который также будет содержать переменную Damage("Урон"). Сама переменная Damage используется во множестве других классов, в зависимости от того, куда стреляет игрок: Wall("Стена") или Enemy("Враг"). Однако такая логика приведет к тому, что каждый из подобных Wall и Enemy классов будет обращаться все время к Damage-переменной класса Shot, даже если урон будет боевой установкой вертолета. Следуя Принципу инверсии зависимостей (The Dependency Inversion Principle or D.I.P.), который гласит:"Зависимость на Абстракциях. Нет зависимости на что-то конкретное решение данной проблемы становится очевидным - использовать абстрактный класс. В итоге мы имеем абстрактный класс ShotBehaviour, наследуемый от MonoBehaviour и содержащий переменную Damage, и обычные классы Shot и HelicopterShot, которые в свою очередь будут унаследованы от класса ShotBehaviour. В результате сколько бы еще классов оружия мы не добавляли, все остальные классы продолжают исправно работать.

### Проблема времязатратности переиспользования интерфейсов

Исходный код содержит структуру интерфейса IContent, посредством которой каждый объект может сохранять информацию о себе в файл, а затем использовать данную информацию в игре в режиме реального времени (сама

информация может представлять из себя данные различных типов: string, byte, Object и т.д.). Но каждый раз, когда интерфейс используется в каком-либо классе, каждая из имеющихся в нем функций должна быть переопределена. Согласно Принципу разделения интерфейса (The Interface Segregation Principle or I.S.P.):"Много интерфейсов, специально предназначенных для клиентов, лучше, чем один интерфейс общего назначения". Поэтому гораздо более оптимальным будет организовать отдельный интерфейс для каждого типа данных (IString, IByte и т.д.) и использовать в каждом классе только необходимые.

### Проблема множественности багов вследствие одного изменения

Причиной таких последствий прежде всего является то, что 90 процентов работы кода исполняется в одном классе. Например, имеется публичный класс Game ("Игра"), наследуемый от MonoBehaviour, который содержит переменные врага, переменные героя, здоровья и еще много всего прочего, что используется в игре. Такой подход к структуре кода приводит к сложностям в идентификации багов в будущем. Поэтому не стоит забывать о Принципе единственной ответственности (The Single Responsibility Principle or S.R.P.):"Каждый класс выполняет лишь одну задачу"и организовать многокомпонентную структуру классов. Для каждого компонента (врага, героя,здоровья) - свой класс, свой .cs файл.

### Проблема поломки старых фитч

Изначально наш код содержит публичный класс ShapeTools ("Инструменты формы"), в котором есть функция расчета площади Area, возвращающая произведение ширины на высоту, и класс Rectangle("Прямоугольник"), наследуемый от ShapeBehaviour. Совершенно очевидно, что если мы будем иметь дело с любыми другими фигурами, расчет площади которых будет производиться иным образом, произойдет ошибка. Казалось бы решить это можно путем использования блоков с оператором if: если прямоугольник - считай так, круг - считай иначе и т.д. Но чем больше новых фигур будет появляться, тем более громоздким будет становиться класс. В этом случае оптимально модифицировать код можно опираясь на принцип Принцип открытости/закрытости (The Open Closed Principle or O.C.P.):"Программные сущности должны быть открыты для расширения, но закрыты для модификации". Следует организовать отдельный абстрактный класс ShapeBehaviour, который будет содержать абстрактный метод Area. Для каждой фигуры будет определен свой отдельный класс, в котором метод Area будет переопределен (override) с учетом конкретной фигуры.

### Проблема поломки классов вследствие расширения их функционала

На начальном этапе игра предполагалась исключительно в 2D-формате. Управление персонажем происходит посредством класса GameBoard, который унаследован от MonoBehaviour и содержит в себе функцию установки двумерных координат SetTile. В ходе правок заказчик потребовал включить бонусные 3D-уровни. Для решения задачи был создан класс GameBoard3D, наследуемый от класса GameBoard и содержащий в себе функцию установки уже трехмерных координат SetTile. Однако в этом случае работа относительно трехмерных координат будет некорректной: обращение в итоге будет осуществляться к двухмерным координатам. Подход к пониманию сути проблемы и ее решению дает Принцип подстановки Барбары Лисков (The Liskov Substitution Principle or L.S.P.):"Наследующий класс должен дополнять, а не изменять базовый". В случае организации наследования класса GameBoard3D непосредственно от MonoBehaviour конфликта координат удается избежать.

### Список литературы


[1] CLR via C Sharp. Jeffrey Richter

[2] Grome Terrain Modeling with Ogre3D UDK and Unity3D. Richard A. Hawley



*Нестерова Екатерина Андреевна*, студентка 3 курса факультета информационных технологий и управления Белорусского государственного университета информатики и радиоэлектроники, kateplain17@gmail.com.

*Научный руководитель: Рак Татьяна Александровна*, ассистент кафедры вычислительных методов и программирования Белорусского государственного университета информатики и радиоэлектроники, tatianarak@bsuir.by.