Прототип Prototype

1. Шаблон проектирования прототип – как выглядит структура кода для реализации различных видов монстров, мы добавляем каждому спаунеры, как выглядит реализация, чем плох такой код, какое решение предлагает шаблон Прототип, что мы добавляем нашему базовому классу монстра, как определяем поведение этого в дочерних, сколько теперь надо классов спаунеров (+ синтаксис), синтаксис создания спаунера

2. Насколько хорошо он работает? – нам теперь не нужно создавать отдельный класс спаунер для каждого монстра, но что нам все равно надо сделать + подводные камни при клонировании

3. Функции спаунера – как по-другому можно реализовать спаунер для каждого класса (не через класс), как выглядит базовый класс спаунер и что хранит, синтаксис создания спаунера в таком случае

4. Шаблоны (Templates) – как написать код через шаблоны в С++ и создать объект спаунера для определенного типа монстров

5. Прототипы для моделирования данных – описание атрибутов монстров через JSON, как выглядит в плохой реализации, хорошим с использованием прототипа, создавали ли мы базового гоблина, пример с боссами и уникальными предметами

**1. Шаблон проектирования прототип**

Давайте представим, что мы делаем игру в стиле Gauntlet. У нас есть всякие существа и демоны, роящиеся вокруг героя и норовящие откусить кусочек его плоти. Эти незваные сотрапезники появляются через "спаунеры" (spawners, тут, источники- генераторы создания существ, прим.пер.) и для каждого типа врагов есть отдельный тип спаунера.

Для упрощения примера давайте сделаем предположение, что для каждого типа монстра в игре имеется отдельный тип. Т.е. у нас есть C++ классы для Ghost, Demon, Sorcerer и т.д.:

class Monster

{

// Stuff...

};

class Ghost : public Monster {};

class Demon : public Monster {};

class Sorcerer : public Monster {};

Спаунер конструирует экземпляры одного из типов монстров. Для поддержки всех монстров в игре мы можем использовать прямолинейный подход и заведем класс спаунер для каждого класса монстра. В результате получится следующая иерархия:



Реализация будет выглядеть так:

class Spawner

{

public:

virtual ~Spawner() {}

virtual Monster\* spawnMonster() = 0;

};

class GhostSpawner : public Spawner

{

public:

virtual Monster\* spawnMonster()

{

return new Ghost();

}

};

class DemonSpawner : public Spawner

{

public:

virtual Monster\* spawnMonster()

{

return new Demon();

}

};

// You get the idea...

Если вам конечно не платят за каждую строчку кода, использовать такой подход совсем не весело. Куча классов, куча похожего кода, куча избыточности, куча дублей, куча самоповторов...

Шаблон Прототип предлагает решение. Ключевой мыслью является **создание объекта, который может порождать объекты, похожие на себя**. Если у вас есть один призрак, вы можете с его помощью получить кучу призраков. Если есть демон, можно сделать больше демонов. Любого монстра можно трактовать как прототипируемого монстра, используемого для генерации новых версий его самого.

Для реализации этой идеи, мы дадим нашему базовому классу Monster абстрактный метод clone():

class Monster

{

public:

virtual ~Monster() {}

virtual Monster\* clone() = 0;

// Other stuff...

};

Каждый подкласс монстра предоставляет свою реализацию, которая возвращает объект, идентичный по классу и состоянию ему самому. Например:

class Ghost : public Monster {

public:

Ghost(int health, int speed)

: health\_(health),

speed\_(speed)

{}

virtual Monster\* clone()

{

return new Ghost(health\_, speed\_);

}

private:

int health\_;

int speed\_;

};

Как только все монстры будут его поддерживать, нам больше не нужен будет отдельный класс спаунер для каждого класса монстров. Вместо этого мы обойдемся всего одним:

class Spawner

{

public:

Spawner(Monster\* prototype)

: prototype\_(prototype)

{}

Monster\* spawnMonster()

{

return prototype\_->clone();

}

private:

Monster\* prototype\_;

};

Внутри себя он содержит монстра, скрытого извне, который используется спаунером в качестве шаблона для штамповки новых монстров ему подобных. Получается нечто наподобие матки пчел, никогда не покидающей своего улья.



Для создания спаунера призраков, мы просто создаем прототипируемый экземпляр призрака и затем создаем спаунер, который будет хранить этот прототип:

Monster\* ghostPrototype = new Ghost(15, 3);

Spawner\* ghostSpawner = new Spawner(ghostPrototype);

Интересна одна особенность этого шаблона заключается в том, что он не просто клонирует класс прототипа, но и клонирует его состояние. Это значит, что мы можем сделать спаунер для быстрых призраков, для слабых, для медленных, просто создавая соответствующего прототипируемого призрака.

**2. Насколько хорошо он работает?**

Итак, нам не нужно создавать отдельный класс спаунер для каждого монстра и это хорошо. Но при этом нам нужно реализовывать метод clone() в каждом классе монстров. Кода там примерно столько же сколько и в спаунере.

К сожалению, если вы попытаетесь написать корректную реализацию clone() , вы быстро наткнетесь на несколько подводных камней. Должен это быть глубокий клон или приблизительный? Другими словами, если демон держит вилы, должны ли мы клонировать вилы при клонировании демона?

**3. Функции спаунера**

Даже если у нас для каждого типа монстра имеется свой класс, есть другой способ

"поймать кота". Вместо того, чтобы делать отдельный класс спаунер для каждого

монстра, можно организовать функцию спаунер:

Monster\* spawnGhost()

{

return new Ghost();

}

Это уже не настолько примитивный подход, как создание отдельного класса для

каждого нового типа монстров. Теперь единственный класс-спаунер может просто

хранить указатель на функцию:

typedef Monster\* (\*SpawnCallback)();

class Spawner

{

public:

Spawner(SpawnCallback spawn)

: spawn\_(spawn)

{}

Monster\* spawnMonster()

{

return spawn\_();

}

private:

SpawnCallback spawn\_;

};

И для создания спаунера призраков нужно будет всего лишь вызвать:

Spawner\* ghostSpawner = new Spawner(spawnGhost);

**4. Шаблоны (Templates)**

Нашему классу спаунеру нужно создать экземпляр определенного класса, но мы не хотим жестко прописывать в коде определенный класс монстра. Естественным решением этой задачи будет воспользоваться возможностями шаблонов и добавить параметр типа:

class Spawner

{

public:

virtual ~Spawner() {}

virtual Monster\* spawnMonster() = 0;

};

template <class T>

class SpawnerFor : public Spawner

{

public:

virtual Monster\* spawnMonster() { return new T(); }

};

Применение выглядит следующим образом:

Spawner\* ghostSpawner = new SpawnerFor<Ghost>();

**5. Прототипы для моделирования данных**

Давайте представим себе, что мы определяем модель данных для бессовестного клона Gauntlet , о котором я писал выше. Геймдизайнеру нужны какие-то файлы, в которые он сможет поместить описание атрибутов монстров и предметов.

Можно использовать JSON : сущности данных будут представлены в виде maps или мешков со свойствами (property bags) или еще дюжиной терминов, потому что программисты просто обожают придумывать для одного и того же разные имена.

Итак, гоблин в игре описан следующим образом:

{

"name": "goblin grunt",

"minHealth": 20,

"maxHealth": 30,

"resists": ["cold", "poison"],

"weaknesses": ["fire", "light"]

}

Довольно прямолинейный подход и даже не любящие писать дизайнеры могут справиться. Можно, например, добавить еще парочку сестринских описаний в славном семейном дереве зеленых гоблинов:

{

"name": "goblin wizard",

"minHealth": 20,

"maxHealth": 30,

"resists": ["cold", "poison"],

"weaknesses": ["fire", "light"],

"spells": ["fire ball", "lightning bolt"]

}

{

"name": "goblin archer",

"minHealth": 20,

"maxHealth": 30,

"resists": ["cold", "poison"],

"weaknesses": ["fire", "light"],

"attacks": ["short bow"]

}

Если бы это был обычный код, наше чувство прекрасного уже заставило бы нас беспокоиться. У этих сущностей слишком много общей дублирующейся информации, а хорошо натренированные программисты это просто ненавидят. Данные занимают слишком много места и требуют слишком много времени на написание. Даже для того, чтобы выяснить одинаковые ли это данные, вам нужно тщательно их прочитать. Их поддержка — настоящая головная боль. Если мы захотим сделать всех гоблинов в игре сильнее, нам нужно будет не забыть обновить значение здоровья для них всех. Плохо, плохо, плохо.

Если бы это был код, мы могли бы создать абстракцию "гоблин" и использовать ее между всему типами гоблинов. Но тупой JSON ничего об этом не знает. Давайте попробуем сделать его чуточку умнее.

Определим для каждого объекта поле " prototype " и поместим туда имя объекта, к которому он делегирует. Любые свойства, отсутствующие у первого объекта нужно будет смотреть в прототипе.

Это позволит нам упростить описание нашей оравы гоблинов:

{

"name": "goblin grunt",

"minHealth": 20,

"maxHealth": 30,

"resists": ["cold", "poison"],

"weaknesses": ["fire", "light"]

}

{

"name": "goblin wizard",

"prototype": "goblin grunt",

"spells": ["fire ball", "lightning bolt"]

}

{

"name": "goblin archer",

"prototype": "goblin grunt",

"attacks": ["short bow"]

}

Так как и лучник, и чародей имеют в качестве прототипа пехотинца, нам не нужно указывать заново здоровье, сопротивляемости и уязвимости для каждого из них. Добавленная нами в данные логика предельно проста — мы просто добавили простейшее делегирование и сразу смогли избавиться от кучи повторов.

Хочу обратить ваше внимание на то, что мы не стали добавлять четвертого "базового гоблина" в качестве абстрактного прототипа, к которому будут делегировать остальные три. Вместо этого, мы просто взяли одного из гоблинов, который является простейшим и делегируем к нему.

Такой подход является естественным для систем на основе прототипов, где каждый объект можно использовать для клонирования нового объекта с уточненными свойствами и смотрится натуральным и здесь. Применительно к игровым данным такой подход тоже удобен, потому что здесь часто приходится создавать объекты, лишь немного отличающиеся от остальных.

Подумайте о боссах и уникальных предметах. Очень часто они являются лишь немного измененной версией обыкновенных игровых объектов и прототипирование с делегированием очень хорошо подходит для их описания. Магический Sword of Head- Detaching (Меч-голова с плеч) можно описать как длинный меч с определенными бонусами:

{

"name": "Sword of Head-Detaching",

"prototype": "longsword",

"damageBonus": "20"

}

Такие дополнительные возможности для описания данных могут облегчить жизнь вашим дизайнерам и добавить больше вариативности предметам и популяции монстров в игре, а это именно то, что может понравиться игрокам.