Шаблона Command (Команда)

1. Теория, что такое команда, в чем смысл этого шаблона (на чем базируется), как называют этот шаблон банда четырех.

2. Практика

2.1. Настройка ввода – примитивная функция анализа ввода и вызов функций, когда вызывается (еще один шаблон, будущий), что многие игры позволяют делать игрокам, соответственно, как меняем код (что создаем, что хранит обработчик ввода, как теперь выглядит считывание)

2.2. Указания для актеров – в чем недостаток реализованного решения, как поменяем из-за этого код (сам шаблон команды и считывание ввода, как теперь вызываем команды), для чего такая реализация часто используется, как теперь можно организовать поток команд (подсказка в самом вопросе)

2.3. Отмена и повтор – пример реализации команды MoveUnitCommand, реализации считывания ввода, как реализуем отмену и повтор введенных команд, что нужно сделать, чтобы позволить игроку отменить движение (случай с единичной отменой и множественной)

**Теория**

**Команда — это материализация вызова метода.**

**"Материализовать", чтобы вы знали, означает буквально "сделать реальным". Еще один термин материализации — это объявление чего либо "объектом первого класса".**

**Оба термина означают, что мы возьмем некую концепцию и превратим ее в набор данных — объект, который можно поместить в переменную, передать в функцию и т.д. Таким образом, если мы говорим, что команда — это "материализация вызова метода"— это означает, что вызов метода оборачивается некоторым объектом.**

**Есть много разных названий: "обратный вызов", "функция первого класса", "указатель на функцию", "замыкание (closure)", "частично примененная функция (partially applied function)", в зависимости от языка, к которому вы привыкли. Однако все это одного поля ягоды. Банда четырех немного дальше уточняет:**

**Команда — это объектно-ориентированная замена обратного вызова.**

**Практика**

**Настройка ввода**

Внутри каждой игры есть код, отвечающий за считывание пользовательского ввода — нажатия на кнопки, клавиатурные события, нажатия мыши и т.д. Этот код обрабатывает ввод и преобразует его в соответствующие действия в игре:



Самая примитивная реализация выглядит следующим образом:

void InputHandler::handleInput()

{

if (isPressed(BUTTON\_X)) jump();

else if (isPressed(BUTTON\_Y)) fireGun();

else if (isPressed(BUTTON\_A)) swapWeapon();

else if (isPressed(BUTTON\_B)) lurchIneffectively();

}

Такая функция обычно вызывается на каждом кадре внутри игрового цикла (GameLoop). Думаю, вам понятно, что она делает. Здесь мы видим жесткую привязку пользовательского ввода с действиями в игре. Однако многие игры позволяют пользователям настраивать, какие кнопки за что отвечают.

Для того, чтобы это стало возможным нам нужно преобразовать прямые вызовы jump() и fireGun() в нечто, что мы сможем свободно менять местами. "Менять местами" звучит как присвоение значений переменным, поэтому нам нужен объект, который будет представлять игровое действие. И тут в дело вступает шаблон Команда.

Для начала определим базовый класс, представляющий запускаемую игровую команду:

class Command

{

public:

virtual ~Command() {}

virtual void execute() = 0;

};

Теперь создадим дочерние классы для каждой из различных игровых команд:

class JumpCommand : public Command

{

public:

virtual void execute() { jump(); }

};

class FireCommand : public Command

{

public:

virtual void execute() { fireGun(); }

};

// You get the idea...

В нашем обработчике ввода мы будем хранить указатели на команду для каждой кнопки:

class InputHandler

{

public:

void handleInput();

// Methods to bind commands...

private:

Command\* buttonX\_;

Command\* buttonY\_;

Command\* buttonA\_;

Command\* buttonB\_;

};

Теперь обработка ввода сводится к делегированию такого вида:

void InputHandler::handleInput()

{

if (isPressed(BUTTON\_X)) buttonX\_->execute();

else if (isPressed(BUTTON\_Y)) buttonY\_->execute();

else if (isPressed(BUTTON\_A)) buttonA\_->execute();

else if (isPressed(BUTTON\_B)) buttonB\_->execute();

}

Там, где раньше пользовательский ввод напрямую вызывал функции, теперь у нас появился промежуточный слой косвенности:



В этом и заключается сущность шаблона Команда.

Указания для актеров

Классы команд, которые мы только что определили, отлично работают для примера выше, но их возможности все-таки сильно ограничены. Проблема в том, что мы предполагаем что у нас уже есть готовые функции высокого уровня jump() , fireGun() и т.д., которые сами знают, как найти персонаж игрока и заставить его плясать под нашу дудку.

Такое предположение значительно снижает применимость наших команд. Получается что команда JumpCommand — это единственное, что способно заставить прыгать только нашего игрока. Давайте избавимся от этого ограничения. Вместо того, чтобы запускать функцию, которая будет сама искать объект для воздействия, мы сами передадим ей объект, которым хотим управлять:

class Command

{

public:

virtual ~Command() {}

virtual void execute(GameActor& actor) = 0;

};

Здесь в качестве GameActor выступает наш класс "игровой объект", представляющий игрока в игровом мире. Мы передаем его в execute() и таким образом изолированная команда получает возможность вызвать метод выбранного нами актера:

class JumpCommand : public Command

{

public:

virtual void execute(GameActor& actor)

{

actor.jump();

}

};

Теперь мы можем использовать этот единственный класс, чтобы заставить прыгать любого в нашей игре. Правда у нас пока еще нет прослойки между обработчиком ввода и командой, которая собственно получает команду и применяет ее к нужному объекту. Для начала мы изменим handleInput() таким образом, чтобы она возвращала команду:

Command\* InputHandler::handleInput()

{

if (isPressed(BUTTON\_X)) return buttonX\_;

if (isPressed(BUTTON\_Y)) return buttonY\_;

if (isPressed(BUTTON\_A)) return buttonA\_;

if (isPressed(BUTTON\_B)) return buttonB\_;

// Nothing pressed, so do nothing.

return NULL;

}

Функция не может выполнить команду немедленно, потому что не знает какого актера ей передать. Зато мы можем воспользоваться тем преимуществом команды, что это **материализованный** **вызов** — мы можем **отложить** выполнение.

Теперь нам нужен код, который получит команду и передаст в нее актера, представляющего игрока. Нечто наподобие:

Command\* command = inputHandler.handleInput();

if (command)

{

command->execute(actor);

}

Предполагая, что actor указывает на персонажа игрока, мы получаем корректную реализацию того, чего добивались, т.е. мы вернулись к тому же поведению, что и в самом первом примере. Добавив слой косвенности между командой и актером, который ее выполняет, мы получили еще одну приятную способность: **теперь мы можем позволить игроку управлять любым актером в игре, просто подменяя актера, к которому применяется команда**.

На практике такая возможность используется не слишком часто. Но похожий вариант использования все равно часто всплывает. До сих пор мы упоминали только управляемых игроком персонажей. А что насчет остальных? Тех, которые управляются игровым ИИ. Мы можем использовать тот же самый шаблон в качестве интерфейса между движком ИИ и актерами: код ИИ просто будет вызывать объекты Command.

Уменьшение связности в данном случае, когда ИИ выбирает команду, а код актера ее выполняет, дает нам дополнительную гибкость. Мы получаем возможность использовать разные модули ИИ для разных актеров. Или же мы можем их смешивать и выстраивать ИИ для разных стилей поведения. Вам нужен более агрессивный противник? Просто подключите более агрессивный ИИ, чтобы им управлять. На самом деле мы можем даже передать на попечение ИИ персонаж игрока, что довольно удобно для демо-режима, когда игра работает на автопилоте.

Делая команду, управляющую актером объектом первого класса, мы избавляемся от жесткой привязки прямого вызова методов. Вместо этого можете думать об этом, как об очереди или потоке команд.



Некоторый код (обработчик ввода или ИИ) генерирует команды и добавляет их в поток. Другой код (диспетчер или сам актер) поглощает команды и вызывает их. Поместив такую очередь в середину, мы уменьшили связность между производителем с одной стороны и потребителем с другой.

**Отмена и повтор**

Последний пример — это самый известный способ применения данного шаблона. Если объект команда может выполнять действия, значит мы уже сделали маленький шаг к тому, чтобы получить возможность их отменять. Отмену можно встретить в некоторых стратегических играх, когда вы имеете возможность отменить последнее не понравившееся вам действие. Такая функциональность обязательно присутствует и в инструментах, которые используются для создания игр.

Без шаблона Команда, реализация отмены довольно сложна. С ним — пара пустяков. Для примера предположим, что мы разрабатываем однопользовательскую пошаговую игру и хотим разрешить игроку отменять ходы, чтобы он мог больше сосредоточиться на стратегии, а не на угадывании.

Мы уже оценили удобство использования команды для абстрагирования пользовательского ввода, поэтому каждый ход игрока у нас уже инкапсулирован в команду. Например, движение юнита может выглядеть следующим образом:

class MoveUnitCommand : public Command

{

public:

MoveUnitCommand(Unit\* unit, int x, int y)

: unit\_(unit),

x\_(x),

y\_(y)

{}

virtual void execute()

{

unit\_->moveTo(x\_, y\_);

}

private:

Unit\* unit\_;

int x\_, y\_;

};

Обратите внимание на небольшое отличие от нашей предыдущей команды. В предыдущем примере мы хотели абстрагировать команду от актера, на которого она действует. В этом же случае мы специально хотим привязать ее к актеру, которого она двигает. Экземпляр этой команды — это не обобщенная операция "перемещающая что либо", которую можно применить в самом разном контексте, а конкретный отдельный шаг в очереди шагов игры.

Это показывает, насколько вариативным может быть применение данного шаблона. В некоторых случаях, как наша первая парочка примеров, команда — это многоразовый (reusable) объект, представляющий действие, которое можно выполнить. Наш первый пример обработки ввода сводился к единственному вызову метода execute() по нажатию нужной кнопки.

А вот более специфическая команда. Она описывает вещи, которые можно сделать в определенный момент. Это значит, что код обработчика ввода будет создаваться каждый раз, когда игрок решит двинутся. Выглядеть это будет следующим образом:

Command\* handleInput()

{

Unit\* unit = getSelectedUnit();

if (isPressed(BUTTON\_UP)) {

// Move the unit up one.

int destY = unit->y() - 1;

return new MoveUnitCommand(unit, unit->x(), destY);

}

if (isPressed(BUTTON\_DOWN)) {

// Move the unit down one.

int destY = unit->y() + 1;

return new MoveUnitCommand(unit, unit->x(), destY);

}

// Other moves...

return NULL;

}

Тот факт, что команды получаются одноразовыми дает нам определенные преимущества. Чтобы сделать команды отменяемыми, мы определим еще одну операцию, которую должен реализовывать каждый класс команд:

class Command

{

public:

virtual ~Command() {}

virtual void execute() = 0;

virtual void undo() = 0;

};

Метод undo() возвращает игру в то состояние, в котором она была до выполнения соответствующего метода execute() . Вот наша последняя команда, дополненная поддержкой отмены:

class MoveUnitCommand : public Command

{

public:

MoveUnitCommand(Unit\* unit, int x, int y)

: unit\_(unit),

xBefore\_(0),

yBefore\_(0),

x\_(x),

y\_(y)

{}

virtual void execute()

{

// Remember the unit's position before the move

// so we can restore it.

xBefore\_ = unit\_->x();

yBefore\_ = unit\_->y();

unit\_->moveTo(x\_, y\_);

}

virtual void undo()

{

unit\_->moveTo(xBefore\_, yBefore\_);

}

private:

Unit\* unit\_;

int xBefore\_, yBefore\_;

int x\_, y\_;

};

Обратите внимание, что мы добавили в класс больше состояний. После того, как мы переместили юнит, ему неоткуда узнать, где он был раньше. Чтобы иметь возможность отменить перемещение, нам нужно запомнить предыдущую позицию самостоятельно. Вот для этого мы и добавляем в команду xBefore\_ и yBefore\_.

Чтобы позволить игроку отменить движение, нам нужно сохранить последнюю выполненную им команду. И потом, когда мы жмакнем Ctrl-Z , мы просто вызовем метод undo() . (Если мы уже выполнили отмену, то по нажатию на ту же кнопку можно выполнить команду повтор и выполнить команду снова.)

Поддержка множественной отмены не намного сложнее. Вместо того, чтобы просто запоминать последнюю команду, мы будем хранить список команд и ссылку на "текущую". Когда игрок выполняет команду, она добавляется в список команд и помечается как "текущая".



Когда игрок выбирает "Отмена", мы отменяем текущую команду и сдвигаем указатель на одну позицию назад. Когда мы выполняем повтор, мы перемещаем указатель на позицию вперед и выполняем команду. Когда игрок после отмены выполняет новую команду, все содержимое списка после текущей команды выбрасывается.