Шаблон Состояние (State pattern)

1. Все мы там были – описать код и баги в ситуациях, когда мы описываем прыжок, приседание и атаку в прыжке для нашего персонажа

2. Конечные автоматы – наше спасение: что мы в ярости рисуем на бумажке, как это называется, в чем их суть (4 пункта)

3. Перечисления и переключатели – в чем проблема нашего старого класса Heroine, как мы это исправляем, порядок чего меняется при этом, в каком теперь состоянии не сможет находиться героиня, какую новую проблему мы встречаем (для реализации определенной атаки), как изменяется код и почему это не очень

4. Шаблон Состояние – с чем «ассоциируется» условное ветвление у ООП-ребят, как определяют шаблон Банда Четырех

4.1. Интерфейс состояния – какие методы определяет

4.2. Классы для каждого из состояний – как записываются, в чем фишка с зарядом в присяде

4.3. Делегирование состоянию – что теперь надо сделать, что изменить состояние и как работают методы у класса героини

5. А где же эти объекты состояния? – в чем проблема по сравнению с enum?

5.1. Статические состояния – когда применяем этот метод (можно пойти дальше и заменить класс), куда поместить экземпляр и как менять состояния (код)

5.2. Экземпляры состояний – когда этот метод применяем, как теперь выглядит обработчик ввода (интерфейс функции) и изменение состояния, а так же сам обработчик ввода (код функции), когда что выбирать из этих двух методов

6. Действия для входа и выхода – какая осталась проблема, связанная с переключением состояний, какую вводим доп. функцию в классы состояний, как меняем считывание у Героини, как упрощается для DuckingState код, какую еще можно (противоположную первой) ввести функцию

7. И чего же мы добились? – вроде все супер, но в чем проблема в самих FSM?

8. Машина конкурентных состояний – мы решили добавить героине возможность носить оружие – какую проблему встречаем, как увеличится количество состояний, что мы делаем неправильно, как это исправить (как с точки зрения матана изменяется формула для количества состояний), изменение класса героини и обработчика ввода, но что если состояния взаимодействуют друг с другом?

9. Иерархическая машина состояний – как поступить, если у нас целый букет **похожих** состояний, реализация в коде, как можно по-другому (не через первый прием) реализовать иерархию

10. Автомат с магазинной памятью – что использует, пример со стрельбой и что делать дальше, как называется такая структура данных, как работает

**1. Все мы там были**

Допустим мы работаем над небольшим платформером сайд-скроллером. Наша задача заключается в моделировании героини, которая будет аватаром игрока в игровом мире. Это значит, что она должна реагировать на пользовательский ввод. Нажмите B и она прыгнет. Довольно просто:

void Heroine::handleInput(Input input)

{

if (input == PRESS\_B)

{

yVelocity\_ = JUMP\_VELOCITY;

setGraphics(IMAGE\_JUMP);

}

}

Заметили баг?

Здесь нет никакого кода, предотвращающего "прыжок в воздухе"; продолжайте нажимать B пока она в воздухе и она будет подлетать снова и снова. Проще всего решить это добавлением булевского флага isJumping\_ в Heroine , который будет следить за тем когда героиня прыгнула:

void Heroine::handleInput(Input input)

{

if (input == PRESS\_B)

{

if (!isJumping\_)

{

isJumping\_ = true;

// Jump...

}

}

}

Дальше мы захотели добавить героине игрока возможность пригибаться, когда она находится на земле и вставать, когда кнопка отжимается:

void Heroine::handleInput(Input input)

{

if (input == PRESS\_B)

{

// Jump if not jumping...

}

else if (input == PRESS\_DOWN)

{

if (!isJumping\_)

{

setGraphics(IMAGE\_DUCK);

}

}

else if (input == RELEASE\_DOWN)

{

setGraphics(IMAGE\_STAND);

}

}

А здесь баг заметили?

С помощью этого кода игрок может:

1. Нажать вниз для приседания.

2. Нажать B для прыжка из сидячей позиции.

3. Отпустить вниз, находясь в воздухе.

При этом героиня переключится на графику стояния прямо в воздухе. Придется добавить еще один флаг...

void Heroine::handleInput(Input input)

{

if (input == PRESS\_B)

{

if (!isJumping\_ && !isDucking\_)

{

// Jump...

}

}

else if (input == PRESS\_DOWN)

{

if (!isJumping\_)

{

isDucking\_ = true;

setGraphics(IMAGE\_DUCK);

}

}

else if (input == RELEASE\_DOWN)

{

if (isDucking\_)

{

isDucking\_ = false;

setGraphics(IMAGE\_STAND);

}

}

}

Теперь будет здорово добавить героине способность атаковать в подкате, когда игрок нажимает вниз, находясь в воздухе:

void Heroine::handleInput(Input input)

{

if (input == PRESS\_B)

{

if (!isJumping\_ && !isDucking\_)

{

// Jump...

}

}

else if (input == PRESS\_DOWN)

{

if (!isJumping\_)

{

isDucking\_ = true;

setGraphics(IMAGE\_DUCK);

}

else

{

isJumping\_ = false;

setGraphics(IMAGE\_DIVE);

}

}

else if (input == RELEASE\_DOWN)

{

if (isDucking\_)

{

// Stand...

}

}

}

Снова ищем баги. Нашли?

У нас есть проверка на то, чтобы было невозможно прыгнуть в воздухе, но не во время подката. Добавляем еще один флаг...

Есть в этом подходе что-то неправильное. Каждый раз, когда мы касаемся кода, у нас что-то ломается. Нам понадобится добавить еще кучу движения, у нас ведь еще даже ходьбы нет, но при таком подходе нам придется преодолеть еще кучу багов.

**2. Конечные автоматы – наше спасение**

В порыве разочарования, вы убираете со стола все, кроме карандаша и бумаги и начинаете чертить блок-схему. Рисуем прямоугольник для каждого действия, что может совершить героиня: стояние, прыжок, приседание и подкат. Чтобы она могла реагировать на нажатие клавиш в любом из состояний, рисуем стрелки между этими прямоугольниками, подписываем над ними кнопки и соединяем между собой состояния.



Поздравляю, вы только что создали **конечный автомат** (finite state machine). Они пришли из области компьютерных наук, называемой теория автоматов (automata theory), в семейство структур которой также входит знаменитая машина Тьюринга. FSM - простейший член этого семейства.

Суть заключается в следующем:

* **У нас есть фиксированный набор состояний, в которых может находиться автомат**. В нашем примере это стояние, прыжок, приседание и подкат.
* **Автомат может находиться только в одном состоянии в каждый момент времени**. Наша героиня не может прыгать и стоять одновременно. Собственно для того чтобы это предотвратить FSM в первую очередь и используется.
* **Последовательность ввода или событий передается автомату**. В нашем примере это нажатие и отпускание кнопок.
* **Каждое состояние имеет набор переходов, каждый из которых связан с вводом и указывает на состояние**. Когда происходит пользовательский ввод, если он соответствует текущему состоянию, автомат меняет свое состояние на то куда указывает стрелка.

Например, если нажать вниз в состоянии стояния, произойдет переход в состояние приседания. Нажатие вниз во время прыжка меняет состояние на подкат. Если в текущем состоянии никакой переход для ввода не предусмотрен — ничего не происходит.

Из этого и состоит весь подход: состояния, ввод и переходы. Можно изобразить их в виде блок-схемы. К сожалению, компилятор таких каракулей не поймет. Так как же в таком случае реализовать конечный автомат?

Моя любимая аналогия FSM — это старый текстовый квест Zork . У вас есть мир, состоящий из комнат, которые соединены между собой переходами. И вы можете исследовать их, вводя команды типа "идти на север".

Такая карта полностью соответствует определению конечного автомата. Комната, в которой вы находитесь — это текущее состояние. Каждый выход из комнаты — переход. Навигационные команды — ввод.

**3. Перечисления и переключатели**

Одна из проблем нашего старого класса Heroine заключается в том, что он допускает некорректную комбинацию булевских ключей: isJumping\_ и isDucking\_, они не могут быть истинными одновременно. А если у вас есть несколько булевских флагов, только один из которых может быть true, не лучше ли заменить их все на enum.

В нашем случае с помощью enum можно полностью описать все состояния нашей FSM таким образом:

enum State

{

STATE\_STANDING,

STATE\_JUMPING,

STATE\_DUCKING,

STATE\_DIVING

};

Вместо кучи флагов, у Heroine есть только одно поле state\_. Также нам придется изменить порядок ветвления. В предыдущем примере кода, мы делали ветвление сначала в зависимости от ввода, а потом уже от состояния. При этом мы группировали код по нажатой кнопке, но размывали код, связанный с состояниями. Теперь мы сделаем наоборот и будем переключать ввод в зависимости от состояния. Получим мы вот что:

void Heroine::handleInput(Input input)

{

switch (state\_)

{

case STATE\_STANDING:

if (input == PRESS\_B)

{

state\_ = STATE\_JUMPING;

yVelocity\_ = JUMP\_VELOCITY;

setGraphics(IMAGE\_JUMP);

}

else if (input == PRESS\_DOWN)

{

state\_ = STATE\_DUCKING;

setGraphics(IMAGE\_DUCK);

}

break;

case STATE\_JUMPING:

if (input == PRESS\_DOWN)

{

state\_ = STATE\_DIVING;

setGraphics(IMAGE\_DIVE);

}

break;

case STATE\_DUCKING:

if (input == RELEASE\_DOWN)

{

state\_ = STATE\_STANDING;

setGraphics(IMAGE\_STAND);

}

break;

}

}

Выглядит довольно тривиально, но тем не менее этот код уже гораздо лучше, чем предыдущий. У нас остались некоторые условные ветвления, но зато мы упростили изменяемое состояние до единственного поля. Весь код, управляющий единственным состоянием собран в одном месте. Это самый простой способ реализации конечного автомата и иногда его вполне достаточно.

Теперь героиня уже не сможет быть в неопределенном состоянии. При использовании булевых флагов некоторые комбинации были возможны, но не имели смысла. При использовании enum все значения корректны.

К сожалению, ваша проблема может перерасти такое решение. Допустим, мы захотели добавить нашей героине специальную атаку, для проведения которой героине нужно присесть для подзарядки и потом разрядить накопленную энергию. И пока мы сидим, нам нужно следить за временем зарядки.

Добавляем в Heroine поле chargeTime\_ для хранения времени зарядки. Допустим у нас уже есть метод update(), вызываемый на каждом кадре. Добавим в него следующий код:

void Heroine::update()

{

if (state\_ == STATE\_DUCKING)

{

chargeTime\_++;

if (chargeTime\_ > MAX\_CHARGE)

{

superBomb();

}

}

}

Каждый раз, когда мы приседаем заново, нам нужно обнулять этот таймер. Для этого нам нужно изменить handleInput():

void Heroine::handleInput(Input input)

{

switch (state\_)

{

case STATE\_STANDING:

if (input == PRESS\_DOWN)

{

state\_ = STATE\_DUCKING;

chargeTime\_ = 0;

setGraphics(IMAGE\_DUCK);

}

// Handle other inputs...

break;

// Other states...

}

}

В конце концов, для добавления этой атаки с подзарядкой, нам пришлось изменить два метода и добавить поле chargeTime\_ в Heroine , даже если оно используется только в состоянии приседания. Хотелось бы иметь весь этот код и данные в одном месте. Банда Четырех может нам в этом помочь.

4. Шаблон Состояние

Для людей, хорошо разбирающихся в объектно-ориентированной парадигме, каждое условное ветвление — это возможность для использования динамической диспетчеризации (другими словами, вызова виртуального метода в C++). Думаю, иногда можно слишком глубоко спуститься в эту «кроличью нору». Иногда if — это все что нам нужно.

В нашем примере мы уже добрались до той критической точки, когда нам стоит обратить внимание на что-то объектно-ориентированное. Это подводит нас к шаблону Состояние. Цитирую Банду Четырех:

*«Позволяет объектам менять свое поведение в соответствии с изменением внутреннего состояния. При этом объект будет вести себя как другой класс.»*

Не очень-то и понятно. В конце концов и switch с этим справляется. Применительно к нашему примеру с героиней шаблон будет выглядеть следующим образом:

**4.1. Интерфейс состояния**

Для начала определим интерфейс для состояния. Каждый бит поведения, зависящий от состояния — т.е. все что мы раньше реализовывали при помощи switch — превращается в виртуальный метод этого интерфейса. В нашем случае это handleInput() и update().

class HeroineState

{

public:

virtual ~HeroineState() {}

virtual void handleInput(Heroine& heroine, Input input) {}

virtual void update(Heroine& heroine) {}

};

**4.2. Классы для каждого из состояний**

Для каждого состояния мы определяем класс, реализующий интерфейс. Его методы определяют поведение героини в данном состоянии. Другими словами берем все варианты из switch в предыдущем примере превращаем их в класс состояния. Например:

class DuckingState : public HeroineState

{

public:

DuckingState()

: chargeTime\_(0)

{}

virtual void handleInput(Heroine& heroine, Input input) {

if (input == RELEASE\_DOWN)

{

// Change to standing state...

heroine.setGraphics(IMAGE\_STAND);

}

}

virtual void update(Heroine& heroine) {

chargeTime\_++;

if (chargeTime\_ > MAX\_CHARGE)

{

heroine.superBomb();

}

}

private:

int chargeTime\_;

};

Обратите внимание, что мы перенесли chargeTime\_ из класса самой героини в класс DuckingState . И это очень хорошо, потому что этот кусок данных имеет значение только в этом состоянии и наша модель данных явно об этом свидетельствует.

**4.3. Делегирование состоянию**

Дальше мы даем Heroine указатель на текущее состояние, избавляемся от здоровенного switch и делегируем его работу состоянию.

class Heroine

{

public:

virtual void handleInput(Input input)

{

state\_->handleInput(\*this, input);

}

virtual void update()

{

state\_->update(\*this);

}

// Other methods...

private:

HeroineState\* state\_;

};

Чтобы "изменить состояние" нам нужно просто сделать так, чтобы state\_ указывал на другой объект HeroineState . В этом собственно и заключается шаблон Состояние.

**5. А где же эти объекты состояния?**

Я вам кое-что не сказал. Чтобы изменить состояние, нам нужно присвоить state\_ новое значение, указывающее на новое состояние, но откуда этот объект возьмется? В нашем примере с enum думать не о чем: значения enum — это просто примитивы наподобие чисел. Но теперь наши состояния представлены классами и это значит, что нам нужны указатели на реальные экземпляры. Существует два самых распространенных ответа:

**5.1. Статические состояния**

Если объект состояния не имеет никаких других полей, единственное, что он хранит — это указатель на внутреннюю виртуальную таблицу методов, для того чтобы эти методы можно было вызвать. В таком случае, нет никакой необходимости иметь больше одного экземпляра класса: каждый из экземпляров все равно будет одинаковым.

Если у вашего состояния нет полей и только один виртуальный метод, можно еще сильнее упростить шаблон. Заменим каждый класс состояния функцией состояния — обычной функцией верхнего уровня. И соответственно поле state\_ в нашем главном классе превратится в простой указатель на функцию.

Вполне можно обойтись единственным статическим экземпляром. Даже если у вас целая куча FSM , находящихся одновременно в одном и том же состоянии, они могут указывать на один и тот же статический экземпляр, потому что ничего специфичного для конкретного конечного автомата в нем нет.

Куда вы поместите статический экземпляр — это уже ваше дело. Найдите такое место, где это будет уместно. Давайте поместим наш экземпляр в базовый класс. Без всякой причины.

class HeroineState

{

public:

static StandingState standing;

static DuckingState ducking;

static JumpingState jumping;

static DivingState diving;

// Other code...

};

Каждое из этих статических полей — экземпляр состояния, используемого игрой. Чтобы заставить героиню подпрыгнуть, состояние стояния сделает нечто вроде:

if (input == PRESS\_B)

{

heroine.state\_ = &HeroineState::jumping;

heroine.setGraphics(IMAGE\_JUMP);

}

**5.2. Экземпляры состояний**

Иногда предыдущий вариант не взлетает. Статическое состояние не подойдет для состояния присядки. У него есть поле chargeTime\_ и оно специфично для героини, которая будет приседать. Это еще худо-бедно сработает в нашем случае, потому что у нас всего одна героиня, но если мы захотим добавить кооператив для двух игроков, у нас будут большие проблемы.

В таком случае, нам следует создавать объект состояния, когда мы переходим в него. Это позволит каждому FSM иметь собственный экземпляр состояния. Конечно, если мы выделяем память под новое состояние, это значит нам следует освободить занимаемую память текущего. Мы должны быть осторожны, так как код, который вызывает изменения, находится в методе текущего состоянии. Мы не хотим, чтобы удалить this из-под себя.

Вместо этого, мы позволим handleInput() в HeroineState опционально возвращать новое состояние. Когда это произойдет, Heroine удалит старое состояние и поменяет его на новое, например, так:

void Heroine::handleInput(Input input)

{

HeroineState\* state = state\_->handleInput(\*this, input);

if (state != NULL)

{

delete state\_;

state\_ = state;

}

}

Таким образом, мы не удаляем предыдущее состояние, пока мы не вернулись из своего метода. Теперь, состояние стояния может перейти к состоянию нырок путем создания нового экземпляра:

HeroineState\* StandingState::handleInput(Heroine& heroine,

Input input)

{

if (input == PRESS\_DOWN)

{

// Other code...

return new DuckingState();

}

// Stay in this state.

return NULL;

}

Когда у меня получается, я предпочитаю использовать статические состояния, потом что они не занимают память и такты процессора, выделяя объекты при каждом изменении состояния. Для состояний, которые не представляют из себя нечто большее, чем просто состояния — это как раз то что нужно.

**6. Действия для входа и выхода**

Шаблон Состояние предназначен для инкапсуляции всего поведения и связанных с ним данных внутри одного класса. У нас довольно неплохо получается, но остались некоторые невыясненные детали. Когда героиня изменяет состояние, мы также переключаем ее спрайт. Прямо сейчас, этот код принадлежит состоянию, с которого она переключается. Когда состояние переходит от нырка в состояние стояния, то нырок устанавливает ее образ:

HeroineState\* DuckingState::handleInput(Heroine& heroine,

Input input)

{

if (input == RELEASE\_DOWN)

{

heroine.setGraphics(IMAGE\_STAND);

return new StandingState();

}

// Other code...

}

То, что мы действительно хотим, каждое состояние контролировало свою собственную графику. Мы можем добиться этого, добавив в состояние входное действие (entry action):

class StandingState : public HeroineState

{

public:

virtual void enter(Heroine& heroine)

{

heroine.setGraphics(IMAGE\_STAND);

}

// Other code...

};

Возвращаясь к Heroine , мы модифицируем код, добиваясь, чтобы изменение состояния сопровождалось вызовом функции входного действия нового состояния:

void Heroine::handleInput(Input input)

{

HeroineState\* state = state\_->handleInput(\*this, input);

if (state != NULL)

{

delete state\_;

state\_ = state;

// Call the enter action on the new state.

state\_->enter(\*this);

}

}

Это позволит упростить код состояния DuckingState:

HeroineState\* DuckingState::handleInput(Heroine& heroine,

Input input)

{

if (input == RELEASE\_DOWN)

{

return new StandingState();

}

// Other code...

}

Все это делает переключение в стояние и состояние стояния заботится о графике. Теперь наши состояния действительно инкапсулированы. Еще одной приятной особенностью такого входного действия является то, что оно запускается при входе в состояние независимо от состояния, в котором мы находились.

На большинстве графов состояний из реальной жизни присутствует несколько переходов в одно и то же состояние. Например, наша героиня может стрелять из оружия стоя, сидя или в прыжке. А это значит, что у нас может появиться дублирование кода везде, где это происходит. Входное действие позволяет собрать его в одном месте.

Можно по аналогии сделать и выходное действие (exit action). Это будет просто метод, который мы будем вызывать для состояния, перед тем, как покидаем его и переключаемся на новое состояние.

**7. И чего же мы добились?**

Я столько времени потратил, чтобы продать вам FSM , а теперь собираюсь вырвать коврик из-под ног. Все, что я до сих пор говорил — правда и отлично решает проблемы. Но, так уж вышло, что самые главные достоинства конечных автоматов, одновременно являются и их самыми большими недостатками.

Состояние автомата помогает вам серьезно распутать код, организовав его в крайне строгую структуру. Все, что у нас есть — это фиксированный набор состояний, единственное текущее состояние и жестко запрограммированные переходы.

Если же вы попробуете использовать машину состояний для чего-либо более сложного, как например игровой AI , вы сразу уткнетесь в ограничения этой модели. К счастью, наши предшественники научились обходить некоторые препятствия. Я закончу эту главу несколькими такими примерами.

**8. Машина конкурентных состояний**

Мы решили добавить нашей героине возможность носить оружие. Хотя она теперь вооружена, она по-прежнему может делать все, что делала раньше: бегать, прыгать, приседать и т.д. Но теперь, делая все это, она еще может и стрелять из оружия.

Если мы захотим вместить такое поведение в рамки FSM , нам придется удвоить количество состояний. Для каждого из состояний нам придется завести еще одно такое же, но уже для героини с оружием: стояние, стояние с оружием, прыжок, прыжок с оружием.... Ну вы поняли.

Если добавить еще несколько видов оружия, количество состояний увеличится комбинаторно. И это не просто куча состояний, а еще и куча повторов: вооруженное и безоружное состояния практически идентичны за исключением части кода, отвечающей за стрельбу.

Проблема здесь в том, что мы смешиваем две части состояния — что она делает и что держит в руках — в один автомат. Чтобы смоделировать все возможные комбинации, нам нужно завести состояние для каждой пары. Решение очевидно: нужно завести два отдельных конечных автомата.

Если мы хотим объединить n состояний действия и m состояний того, что держим в руках в один конечный автомат — нам нужно n × m состояний. Если у нас будет два автомата — нам понадобится n + m состояний.

Наш первый конечный автомат с действиями мы оставим без изменений. А в дополнение к нему создадим еще один автомат для описания того, что героиня держит. Теперь у Heroine будет две ссылки на "состояние", по одной для каждого автомата.

class Heroine

{

// Other code...

private:

HeroineState\* state\_;

HeroineState\* equipment\_;

};

Когда героиня делегирует ввод состояниям, она передает перевод обеим конечным автоматам:

void Heroine::handleInput(Input input)

{

state\_->handleInput(\*this, input);

equipment\_->handleInput(\*this, input);

}

Более сложные системы могут иметь в своем составе конечные автоматы, которые могут поглощать часть ввода таким образом чтобы другие автоматы его уже не получали. Это позволит нам предотвратить ситуацию, когда несколько автоматов реагируют на один и тот же ввод.

Каждый конечный автомат может реагировать на ввод, порождать поведение и изменять свое состояние независимо от других автоматов. И когда оба состояния практически не связаны между собой это отлично работает.

На практике вы можете встретить ситуацию, когда состояния взаимодействуют друг с другом. Например, она не может выстрелить в прыжке или например выполнить атаку с подкатом когда вооружена. Чтобы обеспечить такое поведение и координацию автоматов в коде, вам придется вернуться к той же самой грубой проверке через if другого конечного автомата. Не самое элегантное решение, но по крайней мере работает.

9. Иерархическая машина состояний

После дальнейшего оживления поведения героини, у нее наверняка появится целый букет похожих состояний. Например, у не могут быть состояния стояния, ходьбы, бега и скатывания со склонов. В любом из этих состояний нажатие на B заставляет ее подпрыгнуть, а нажатие вниз — присесть.

В простейшей реализации конечного автомата мы дублировали этот код для всех состояний. Но конечно было бы гораздо лучше, если бы нам нужно было написать код всего один раз и после этого мы могли бы использовать его повторно для всех состояний.

Если бы это был просто объектно-ориентированный код, а не конечный автомат, можно было бы использовать такой прием разделения кода между состояниями, как наследование. Можно определить класс для состояния "на земле", который будет обрабатывать подпрыгивание и приседание. Стояние, ходьба, бег и скатывание для него наследуется и добавляет свое дополнительное поведение.

В таком виде получившаяся структура будет называться иерархическая машина состояний (или иерархический автомат). А у каждого состояния может быть свое суперсостояние (само состояние при этом называется подсостоянием). Когда наступает событие и подсостояние его не обрабатывает, оно передается по цепочке суперсостояний вверх. Другими словами, получается подобие переопределения унаследованного метода.

На самом деле, если мы используем оригинальный шаблон Состояние для реализации FSM , мы уже можем использовать наследование классов для реализации иерархии. Определим базовый класс для суперсостояния:

class OnGroundState : public HeroineState

{

public:

virtual void handleInput(Heroine& heroine, Input input)

{

if (input == PRESS\_B)

{

// Jump...

}

else if (input == PRESS\_DOWN)

{

// Duck...

}

}

};

А теперь каждое подсостояние будет его наследовать:

class DuckingState : public OnGroundState

{

public:

virtual void handleInput(Heroine& heroine, Input input)

{

if (input == RELEASE\_DOWN)

{

// Stand up...

}

else

{

// Didn't handle input, so walk up hierarchy.

OnGroundState::handleInput(heroine, input);

}

}

};

Конечно, это не единственный способ реализации иерархии. Но, если вы не используете шаблон Состояние Банды Четырех, это не сработает. Вместо этого вы можете смоделировать четкую иерархию текущих состояний и суперсостояний с помощью стека состояний вместо единственного состояния в главном классе.

Текущее состояние будет находиться вверху стека, под ним его суперсостояние, дальше суперсостояние для этого суперсостояния и т.д. И когда вам нужно будет реализовать специфичное для состояния поведение, вы начнете с верха стека спускаться по нему вниз, пока состояние его не обработает. (А если не обработает — значит вы его просто игнорируете).

10. Автомат с магазинной памятью

Есть еще одно обычное расширение конечных автоматов, также использующее стек состояния. Только здесь стек представляет совершенно другую концепцию и используется для решения других проблем.

Проблема в том, что у конечного автомата нет концепции истории. Вы знаете в каком состоянии вы находитесь, но у вас нет никакой информации о том, в каком состоянии вы были. И соответственно нет простой возможности вернуться в предыдущее состояние.

Вот простой пример: Ранее мы позволили нашей бесстрашной героине вооружиться до зубов. Когда она стреляет из своего оружия, нам нужно новое состояние для проигрывания анимации выстрела, порождения пули и сопутствующих визуальных эффектов. Для этого мы создаем новое FiringState и делаем в него переходы из всех состояний, в которых героиня может стрелять по нажатию кнопки стрельбы.

Так как это поведение дублируется между несколькими состояниями, здесь как раз можно применить иерархическую машину состояний для повторного использования кода. Сложность здесь в том, что нужно каким-либо образом понять в какое состояние нужно перейти после стрельбы. Героиня может выстрелить всю обойму, пока она стоит на месте, бежит, прыгает или приседает. Когда последовательность стрельбы закончена, ей нужно вернуться в состояние, в котором она была до стрельбы.

Если мы привязываемся к чистому FSM , мы сразу забываем в каком состоянии мы были. Чтобы за этим следить, нам нужно определить множество практически одинаковых состояний — стрельба стоя, стрельба в беге, стрельба в прыжке и т.д. Таким образом, у нас образуются жестко закодированные переходы, переходящие в правильное состояние по своему окончанию.

Что нам на самом деле нужно — так это возможность хранить состояние, в котором мы находились до стрельбы и после стрельбы вспоминать его снова. Здесь нам снова может помочь теория автоматов. Соответствующая структура данных называется Автомат с магазинной памятью (Pushdown Automaton).

Там, где в конечном автомате у нас находится единственный указатель на состояние, в автомате с магазинной памятью находится их стек. В FSM переход к новому состоянию заменяет собой предыдущий. Автомат с магазинной памятью тоже позволяет это делать, но добавляет сюда еще две операции:

1. Вы можете поместить (push) новое состояние в стек. Текущее состояние всегда будет находиться вверху стека, так что это и есть операция перехода в новое состояние. Но при этом старое состояние остается прямо под текущим в стеке, а не исчезает бесследно.

2. Вы можете извлечь (pop) верхнее состояние из стека. Состояние пропадает и текущим становится то что находилось под ним.



Это все что нам нужно для стрельбы. Мы создаем единственное состояние стрельбы. Когда мы нажимаем кнопку стрельбы, находясь в другом состоянии, мы помещаем (push) состояние стрельбы в стек. Когда анимация стрельбы заканчивается, мы извлекаем (pop) состояние и автомат с магазинной памятью автоматически возвращает нас в предыдущее состояние.