Наблюдатель Observer

1. Достижение разблокировано – что мы хотим, чтобы умела делать игра, с чем должно быть связано это поведение и почему это плохо делать напрямую, как нам здесь помогает шаблон наблюдатель (что он делает), пример кода, как синергируют движок и система получения достижений

2. Наблюдатель – реализация класса Observer, что принимает метод специальный, реализация класса Achievements

3. Объект – через что запускается метод уведомления (как его называет Банда четырех), что хранит, public API для добавления и удаления наблюдателей, что дает нам хранение наблюдателей в виде массива, метод для передачи объектом уведомлений

4. Наблюдаемая физика – как теперь подцепить сюда движок, что это позволяет делать, как бросаем уведомления из движка

5. Слишком быстро? – про что не стоит забывать при использовании Наблюдателя, чем это опасно, реакция на событие синхронно и асинхронно, как можно нарваться на взаимную блокировку

6. «Здесь слишком много динамического выделения памяти» - какой массив мы использовали в примере, какой он на самом деле, когда происходит выделения памяти при применении, посылка уведомлений

7. Связанные в цепочку наблюдатели – как мы теперь решаем хранить список наблюдателей, как изменяется класс объекта, класс наблюдателя (какие поля изменяются/добавляются, пока без методов), код метода для добавления наблюдателя (в каком порядке добавляются, в каком уведомляются соответственно), код метода для удаления наблюдателя, код метода для посылки уведомления, как можно изменить поведение последнего метода, какой остался один недочет у нашей реализации

8. Пул список узлов – как реализуем теперь список наблюдателей, чтобы решить последнюю проблему из предыдущего пункта

9. Оставшаяся проблема, удаление объектов и наблюдателей – проблема при удалении наблюдателя, при удалении объекта, решение проблемы для наблюдателей, решение проблемы для объектов, что добавляется у наблюдателей еще при решении проблемы таким образом

**1. Достижение разблокировано**

Допустим, вам нужно добавить в игру систему достижений. Это будут дюжины значков, которые игрок может заработать "Убив 100 демонических обезьян", "Упав с моста" или "Пройдя уровень, имея на руках только дохлого хорька".

Реализовать такую систему не так уж просто, потому что к разблокированию достижения может вести самое различное поведение. Если вы не будете осторожны, корни вашей системы достижений расползутся во всему остальному коду. Потому что достижение "Упасть с моста" явно будет связано с работой физического движка, но уверены ли вы, что хотите видеть функцию unlockFallOffBridge() прямо в самой гуще алгоритмических вычислений?

Чего мы как обычно хотим, так это того, чтобы весь код, касающийся одного аспекта игры, был максимально сконцентрирован в одном месте. Сложность в том, что на получения достижений влияют самые различные объекты геймплея. Но как этого достичь не внедряя код достижений повсюду?

Здесь то нам и поможет шаблон наблюдатель. Он позволяет коду объявлять что произошло нечто интересное не заботясь о том, кто получит уведомление.

Например, если у вас есть физический код, который занимается симуляцией гравитации и определяет какие тела лежат на плоскости, а какие стремительно несутся к ней в падении. Чтобы реализовать упомянутое достижение "Упасть с моста", вам нужно внедрить сюда код получения достижения. Но в результате мы получим месиво в коде. Вместо этого мы поступим немного иначе:

void Physics::updateEntity(Entity& entity)

{

bool wasOnSurface = entity.isOnSurface();

entity.accelerate(GRAVITY);

entity.update();

if (wasOnSurface && !entity.isOnSurface())

{

notify(entity, EVENT\_START\_FALL);

}

}

Всё, что этот код делает — это говорит "Ну, мне как бы все равно кому это интересно, но этот объект только что упал. Можете делать с этим фактом что хотите."

Система получения достижений регистрирует себя таким образом, чтобы физический код мог посылать сообщения, а система достижения их получала. Затем она может проверить, что упавшим телом был наш несчастный герой, и если до этого он находился на мосту, выдаст значок с достижением. При этом мы увидим как достижение разблокируется с салютом и фанфарами и все это без вмешательства в физический код.

**2. Наблюдатель**

Начнем мы с самого любопытного класса, который хочет знать обо всем интересном, что делает другой объект. Этого можно добиться следующей реализацией:

class Observer

{

public:

virtual ~Observer() {}

virtual void onNotify(const Entity& entity, Event event) = 0;

};

Entity – это сам объект, Event – это enum, содержащий виды событий.

Каждый конкретный класс, реализовывающий это становится наблюдателем. В нашем примере это система достижений, примерно следующего вида:

class Achievements : public Observer

{

public:

virtual void onNotify(const Entity& entity, Event event)

{

switch (event)

{

case EVENT\_ENTITY\_FELL:

if (entity.isHero() && heroIsOnBridge\_)

{

unlock(ACHIEVEMENT\_FELL\_OFF\_BRIDGE);

}

break;

// Handle other events, and update heroIsOnBridge\_...

}

}

private:

void unlock(Achievement achievement)

{

// Unlock if not already unlocked...

}

bool heroIsOnBridge\_;

};

**3. Объект**

Метод уведомления запускается объектом, над которым ведется наблюдение. Банда четырех называет его в своей манере "объектом" (subject). Он делает две вещи. Во- первых, он хранит список наблюдателей, которые терпеливо ожидают интересных им сообщений:

class Subject

{

private:

Observer\* observers\_[MAX\_OBSERVERS];

int numObservers\_;

};

Обратите внимание, что у объекта есть открытое (public) API для изменения этого списка:

class Subject

{

public:

void addObserver(Observer\* observer)

{

// Add to array...

}

void removeObserver(Observer\* observer)

{

// Remove from array...

}

// Other stuff...

};

Таким образом, сторонний код может управлять тем, кто получает уведомления. Объект общается с наблюдателями, но не связан с ними. В нашем примере, ни в одной строчке кода физики достижения не упоминаются. И, тем не менее, система достижений уведомления получает. Это была самая хитрая часть шаблона.

Также крайне важно то, что в объекте хранится список наблюдателей, а не просто один из них. Таким образом, мы добиваемся того, что наблюдатели между собой даже косвенно не связаны. Возьмем, например, звуковой движок, который тоже будет наблюдать за падением и проигрывать соответствующий звук. Если объект будет поддерживать только одного наблюдателя, то для того, чтобы звуковой движок смог в нем зарегистрироваться, ему придется предварительно отменить регистрацию системы достижений.

Это значило бы, что две системы начнут между собой взаимодействовать, причем не самым лучшим образом, так как одна будет мешать работе другой. Поддержка списка наблюдателей позволяет каждому наблюдателю действовать независимо от всех остальных. Каждый из них будет работать таким образом, как будто он единственный в мире.

Теперь объекту остается только передать уведомления:

class Subject

{

protected:

void notify(const Entity& entity, Event event)

{

for (int i = 0; i < numObservers\_; i++)

{

observers\_[i]->onNotify(entity, event);

}

}

// Other stuff...

};

**4. Наблюдаемая физика**

Теперь нам осталось еще подцепить сюда физический движок так, чтобы он мог посылать уведомления, а система достижений могла на них подписаться. Будем придерживаться оригинального рецепта из Паттернов проектирования и наследуем объект Subject:

class Physics : public Subject

{

public:

void updateEntity(Entity& entity);

};

Это позволит нам вызвать notify() из Subject . Таким образом, унаследованный класс может вызывать notify() для отправки сообщений, а внешний код — нет. В то же время, addObserver() и removeObserver() публичные. Поэтому любой может наблюдать за подсистемой физики.

Теперь, когда физический движок выполняет нечто стоящее внимания, вызываем notify() как в первом мотивирующем примере. Дальше выполняется обход списка наблюдателей и они уже делают все остальное.



**5. Слишком быстро?**

Не забывайте, что шаблон Наблюдатель работает синхронно. Объект вызывает наблюдателя напрямую, а это значит, что он сам не продолжит работу, пока наблюдатель не вернет свой метод уведомления. Медленный наблюдатель может вообще блокировать выполнение объекта.

Звучит пугающе, но на практике далеко не конец света. Просто следует иметь это в виду. У программистов UI , которые занимается событийным программированием годами, даже есть свой лозунг: "никаких потоков в UI ".

Если вы реагируете на событие синхронно, вам нужно завершить свои дела и передать управление обратно как можно скорее, чтобы не возникла блокировка UI . Если вам нужно выполнить что-то долгое — просто поместите это в другой поток или рабочую очередь.

Нужно быть осторожным, если вы планируете использовать наблюдателей вместе с потоками и настоящими блокировками. Если наблюдатель попытается залочить поток, который уже залочен объектом, мы получим взаимную блокировку (deadlock).

6. «Здесь слишком много динамического выделения памяти»

В примере выше я использовал массив фиксированной длины для того, чтобы максимально упростить код. В реальной реализации, список наблюдателей обычно представляет собой динамическую коллекцию и во время работы увеличивается и уменьшается по мере добавления и удаления наблюдателей. Многих пугает такое обращение с памятью.

Главное, что нужно понять — это то, что выделение памяти происходит только при настройке наблюдателей. **Посылка** уведомлений вообще не требует выделения памяти — это просто вызов метода. Если вы выполните настройку наблюдателей в начале игры и не будете их больше трогать, количество выделений памяти будет минимальным.

**7. Связанные в цепочку наблюдатели**

В коде, который был у нас до сих пор, Subject хранил в себе список на всех Observer , за которыми он наблюдал. У самого класса Observer не было ссылки на этот список. Это чисто виртуальный интерфейс. Интерфейсы предпочтительнее чем конкретные, постоянные классы, так что это не так уж плохо.

Но если мы перенесем часть состояния в сам Observer , мы сможем решить нашу проблему с выделением памяти, передавая объект по цепочке самих наблюдателей. Вместо объекта, хранящего набор указателей, наблюдатели станут узлами связанного списка:



Чтобы это реализовать, мы уберем массив из Subject и заменим его указателем на голову списка наблюдателей:

class Subject

{

Subject()

: head\_(NULL)

{}

// Methods...

private:

Observer\* head\_;

};

Сам Observer мы дополним указателем на следующего наблюдателя в списке:

class Observer

{

friend class Subject;

public:

Observer()

: next\_(NULL)

{}

// Other stuff...

private:

Observer\* next\_;

};

Еще мы сделаем Subject другом. У объекта есть API для добавления и удаления наблюдателей, но теперь мы будем управлять этим списком изнутри самого класса Observer . Самый простой способ позволить ему работать с этим списком — это сделать его другом.

Чтобы зарегистрировать нового наблюдателя, нужно просто добавить его в список. Просто добавим его в начало списка:

void Subject::addObserver(Observer\* observer)

{

observer->next\_ = head\_;

head\_ = observer;

}

Можно выбрать другой вариант и добавить его в конце списка. Такой вариант получается более сложным: Subject должен проходить весь список от начала до конца или хранить еще один указатель tail\_ , который будет указывать на последний элемент.

Добавление в начало списка проще, но имеет один побочный эффект. Когда мы проходим список чтобы отослать уведомления каждому наблюдателю, самые последние из зарегистрированных наблюдателей получают уведомления первыми. Так, что если вы зарегистрировали наблюдателей в последовательности A,B и C, они получат уведомления в последовательности C,B,A.

В теории вас вообще не должна интересовать последовательность. При использовании наблюдателей хорошим тоном считается написание такого кода, который не зависит от очередности уведомлений. А вот если очередность обработки имеет значение, это значит, что между двумя наблюдателями существует связь, которая может больно вам аукнуться.

Давайте посмотрим как выглядит удаление:

void Subject::removeObserver(Observer\* observer)

{

if (head\_ == observer)

{

head\_ = observer->next\_;

observer->next\_ = NULL;

return;

}

Observer\* current = head\_;

while (current != NULL)

{

if (current->next\_ == observer)

{

current->next\_ = observer->next\_;

observer->next\_ = NULL;

return;

}

current = current->next\_;

}

}

Единственное что осталось реализовать — это уведомление. Это также просто как перемещение по списку:

void Subject::notify(const Entity& entity, Event event)

{

Observer\* observer = head\_;

while (observer != NULL)

{

observer->onNotify(entity, event);

observer = observer->next\_;

}

}

Здесь мы проходимся по всему списку и уведомляем каждый наблюдатель в нем. Таким образом мы добиваемся одинакового приоритета для всех наблюдателей и их независимость друг от друга.

Мы можем изменить этот механизм таким образом, чтобы когда наблюдатель получал уведомление, он возвращал флаг, означающий что объект должен продолжать прохождение по списку наблюдателей или остановиться.

Не так уж плохо, верно? Объекту может принадлежать любое количество наблюдателей без малейшего динамического использования памяти. Регистрировать и отменять регистрацию также быстро и просто, как и при работе с обычным массивом. Придется пожертвовать только одним маленьким удобством.

Так как мы используем самого наблюдателя в качестве узла списка, мы предполагаем, что он сам и есть часть списка наблюдателей. Другими словами, наблюдатель может наблюдать только за одним объектом в каждый момент времени. В более традиционной реализации, когда каждый объект хранит свой собственный список, каждый наблюдатель может находиться сразу в нескольких списках одновременно.

Вам придется смириться с этим ограничением. Я нахожу более естественной ситуацию, когда у объекта есть много наблюдателей, а не наоборот. Если для вас это проблема — существует более сложное решение, которое также не использует динамическое выделение памяти. Оно слишком большое, чтобы целиком поместиться в эту главу, но я покажу вам небольшой набросок, а в остальном вы разберетесь сами.

**8. Пул списков узлов**

Как и раньше, каждый объект содержит связанный список наблюдателей. Однако сами эти списки не будут состоять из объектов наблюдателей. Вместо этого они сами будут представлять из себя маленький "узел списка", в котором есть указатель на наблюдатель и указатель на следующий узел в списке.



Так как несколько узлов могут указывать на один и тот же наблюдатель, это означает, что один и тот же наблюдатель может одновременно находиться в списках нескольких объектов. И, таким образом, к нам вернулась способность наблюдать за несколькими объектами одновременно.

**9. Оставшаяся проблема, удаление объектов и наблюдателей**

Основной код, который мы с вами рассмотрели хорош, но что насчет сопроводительного кода: что происходит, когда мы удаляем объект или наблюдателя? Если просто применить delete к любому из наблюдателей, на него останется ссылка в объекте. Теперь это опасный указатель на удаленный объект. И если этот объект попытается послать уведомление... вот почему некоторые люди начинают ненавидеть C++ .

Уничтожать объект проще потому, что в большинстве наших реализаций у нас нет ссылок на него. Но даже в этом случае отправка объекта в мусор может доставить определенные проблемы. Наблюдатели могут ожидать получения уведомлений в будущем и никак не смогут узнать, что этого больше не произойдет. И теперь они вовсе не наблюдатели, как они сами о себе думают.

Справиться с этим можно несколькими способами. Проще всего решить проблему напрямую. Пусть сам наблюдатель и разбирается с тем, чтобы при удалении отменить повсюду свою регистрацию. Чаще всего наблюдатель обладает информацией о том, за какими объектами он наблюдает, и остается только добавить вызов removeObserver() в деструктор.

Если у вас нет желания оставлять наблюдателей висеть в системе, когда объекты превратятся в призраков, это можно легко исправить. Для этого можно заставить каждый объект перед своим уничтожением послать всем финальное уведомление — "последний вздох". Таким образом, все наблюдатели его получат и смогут предпринять соответствующие действия.

Самый правильный ответ состоит в том, чтобы заставить наблюдателей самостоятельно отменять свою регистрацию в объектах при своем удалении. Если вы реализуете эту логику единожды в базовом классе, все остальные, кто будет ее использовать, уже не должны будут заботиться об этом самостоятельно. Сложность при этом конечно увеличивается. Получается, что каждый наблюдатель должен хранить список объектов, за которыми он следит. Получится двухсторонняя связь с помощью указателей.