ПАТТЕРН OBSERVER (НАБЛЮДАТЕЛЬ)

* Название и классификация шаблона проектирования

*Наблюдатель* — паттерн поведения объектов, т.е. связан с алгоритмами и распределением обязанностей. Во время выполнения программы запускается ветвистый поток управления, который определяется данными паттернами, причём внимание заостряется на связях между объектами класса, а не на схеме координации непосредственно.

Так, к примеру, в паттернах поведения уровня класса для задания отношений между разными классами применяется *наследование*.

В то же время в паттернах слоя объектов предусматривается т.н. *композиция* (объединение объектов для придания более сложного поведения структуры).

Третья категория шаблонов связана с *инкапсуляцией* поведения внутри объекта и *делегированием* ему *запросов*.

Observer относится ко второму типу паттернов поведения.

* Назначение

Отвечает за обеспечение зависимости типа “один ко многим” между объектами так, чтобы при изменении состояния одного объекта все зависящие от него объекты информировались об этом и автоматически обновлялись. У паттерна Наблюдатель существуют ещё два наименования: *Dependents* и *Publish-Subscribe*.

* Мотивация (сценарий, иллюстрирующий задачу проектирования. Поймём мотивацию -> поймём абстрактное описание паттерна)

Типичный побочный эффект разбиения системы на взаимодействующие классы — необходимость согласования состояния взаимосвязанных объектов. Так электронная таблица и объект-диаграмма не имеют информации друг о друге, следовательно их можно использовать по отдельности. Но их поведение отражает то, что они знают о существовании друг друга. Изменения в таблице влекут за собой коррективы на диаграмме, и наоборот. При такой модели подразумевается, что элементы интерфейса зависят от данных объекта, и из-за этого должны уведомляться о любых изменениях в его состоянии.

Шаблон Наблюдатель дискрибирует, как “налаживаются” такие отношения. Ключевые объекты паттерна: субъект и наблюдатель (их может быть много). Наблюдателям сообщается о правках в состоянии субъекта. После получения сообщения, наблюдатель опрашивает субъекта, чтобы синхронизировать с ним все его данные, параметры и характеристики.

Эта модель отношений называется “издатель-подписчик”, которая исполняется изданием и публикацией субъектом уведомлений и их рассылкой. При этом нет информации, какие объекты классов — подписчики.

* Применимость (основные условия для применения паттерна)

1. В абстракции присутствует два аспекта, зависящих друг от друга. Их инкапсуляция в различные объекты даёт возможность менять и циклически использовать их независимо.
2. При модификации одного объекта необходимо изменить остальные, при этом вы не знаете, сколько объектов нужно редактировать.
3. Один объект обязан оповещать остальных, не задумываясь об уведомляемых объектах. Иными словами, у объектов не должна возникнуть тесная связь.

* Участники

* Subject* — субъект:

• располагает информацией о своих наблюдателях. За субъектом может

«следить» любое число наблюдателей;

• предоставляет интерфейс для присоединения и отделения наблюдателей;

 *Observer* — наблюдатель:

• определяет интерфейс обновления для объектов, которые должны

уведомляться об изменении субъекта;

* ConcreteSubject* — конкретный субъект:

• сохраняет состояние, представляющее интерес для конкретного наблюдателя ConcreteObserver;

• посылает информацию своим наблюдателям, когда происходит изменение;

 *ConcreteObserver* — конкретный наблюдатель:

• хранит ссылку на объект класса ConcreteSubject;

• сохраняет данные, которые должны быть согласованы с данными

субъекта;

• реализует интерфейс обновления, определенный в классе Observer,

чтобы поддерживать согласованность с субъектом.

* Отношения
* Объект **ConcreteSubject** уведомляет своих наблюдателей о любом изменении, которое могло бы привести к рассогласованности состояний наблюдателя и субъекта;
* После получения от конкретного субъекта уведомления об изменении объект **ConcreteObserver** может запросить у субъекта дополнительную информацию, которую использует для того, чтобы оказаться в состоянии, согласованном с состоянием субъекта.

Объект **Observer**, который инициирует запрос на изменение, откладывает свое обновление до получения уведомления от субъекта. Операция **Notify** не всегда вызывается субъектом. Ее может вызвать и наблюдатель, и посторонний объект.

* Результаты

Паттерн наблюдатель позволяет изменять субъекты и наблюдатели независимо друг от друга. Субъекты разрешается повторно использовать без участия наблюдателей, и наоборот. Это дает возможность добавлять новы наблюдателей без модификации субъекта или других наблюдателей.

* Основные достоинства и недостатки паттерна Наблюдатель

1(+). *Абстрактная связанность субъекта и наблюдателя*. Субъект имеет информацию лишь о том, что у него есть ряд наблюдателей, каждый из которых подчиняется простому интерфейсу абстрактного класса **Observer**. Субъекту неизвестны конкретные классы наблюдателей. Таким образом, связи между субъектами и наблюдателями носят абстрактный характер и сведены к минимуму.

Так как субъект и наблюдатель не являются тесно связанными, они могут находиться на разных уровнях абстракции системы. Субъект более низкого уровня может уведомлять наблюдателей, находящихся на верхних уровнях, не нарушая иерархии системы. Если бы субъект и наблюдатель представляли собой единое целое, то получающийся объект либо пересекал бы границы уровней (нарушая принцип их формирования), либо должен был находиться на каком-то одном уровне (нарушая абстракцию уровня);

2(+). *Поддержка широковещательных коммуникаций*. В отличие от обычного запроса, для уведомления, посылаемого субъектом, не нужно задавать определенного получателя. Уведомление автоматически поступает всем подписавшимся на него объектам. Субъекта не интересует, сколько существует таких объектов; от него требуется всего лишь уведомить своих наблюдателей. Таким образом, мы можем в любое время добавлять и удалять наблюдателей. Наблюдатель сам решает, обработать полученное уведомление или игнорировать его;

3(-)*. Неожиданные обновления*. Поскольку наблюдатели не располагают информацией друг о друге, им неизвестно и о том, во что обходится изменение субъекта. Безобидная на первый взгляд операция над субъектом может вызвать целый ряд обновлений наблюдателей и зависящих от них объектов. Более того, нечетко определенные или плохо поддерживаемые критерии зависимости могут стать причиной непредвиденных обновлений, отследить которые очень сложно. Проблема усугубляется еще и тем, что простой протокол обновления не содержит никаких сведений о том, что именно изменилось в субъекте. Без дополнительного протокола, который позволяет получить информацию об изменениях, наблюдатели будут вынуждены проделать сложную работу для косвенного получения такой информации.

* Реализации механизма зависимостей

1) *связывание субъектов с наблюдателями*. Этим простейшим способом субъект может отслеживать всех наблюдателей, которым он должен посылать уведомления — то есть хранить на них явные ссылки. В какой-то мере это помогает справиться с минусом паттерна **Observer**. Однако при большом числе субъектов при нескольких наблюдателях это может привести к слишком высоким затратам. Один из возможных компромиссов — экономия памяти за счет времени с использованием ассоциативного массива (например, хеш-таблицы, АВЛ-дерева, дерево Гибаса-Седжвика-Байера (красно-чёрное), Splay-деревья (*оно позволяет находить быстрее те данные, которые использовались недавно*), дерево Фенвика (*позволяет быстро изменять значения в* [*массиве*](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%81%D1%81%D0%B8%D0%B2_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) *и находить некоторые функции от элементов массива*), Декартово дерево (*Типичная для сортирующего дерева операция «разделить по ключу x на „меньше x0“ и „не меньше x0“» работает за O(h), где h — высота дерева. На красно-чёрных деревьях придётся восстанавливать балансировку и окраску узлов*)) для хранения отображения. между субъектами и наблюдателями. Тогда субъект, у которого нет наблюдателей, не будет зря расходовать память. С другой стороны, при таком подходе увеличивается время поиска наблюдателей (которое однако можно сократить применяя бинарный поиск, алгоритм Кнута-Морриса-Пратта или алгоритм BFPRT);

2) *наблюдение более чем за одним субъектом*. Иногда наблюдатель может зависеть более чем от одного субъекта. Например, у электронной таблицы бывает более одного источника данных. В таких случаях необходимо расширить интерфейс Update, чтобы наблюдатель мог узнать, какой субъект прислал уведомление. Субъект может просто передать себя в параметре операции Update, тем самым сообщая наблюдателю, что именно нужно обследовать;

3*) кто инициирует обновление?* Для сохранения согласованности субъект и его наблюдатели полагаются на механизм уведомлений. Но какой именно объект вызывает операцию Notify для инициирования обновления?

Возможны два варианта:

• операции класса Subject, изменившие состояние, вызывают Notify для уведомления об этом изменении. Преимущество такого подхода в том, что клиентам не надо помнить о необходимости вызывать операцию Notify субъекта. Недостаток же заключается в том, что при выполнении каждой из нескольких последовательных операций будут проводиться обновления, что может привести к неэффективной работе программы;

• ответственность за своевременный вызов Notify возлагается на клиента. Преимущество: клиент может отложить инициирование обновления до завершения серии изменений, исключив тем самым ненужные промежуточные обновления. Недостаток: у клиентов появляется дополнительная обязанность. Это увеличивает вероятность ошибок, поскольку клиент может забыть вызвать Notify;

4) *висячие ссылки на удаленных субъектов*. Удаление субъекта не должно приводить к появлению висячих ссылок у наблюдателей. Избежать этого можно, например, поручив субъекту уведомлять всех своих наблюдателей о своем удалении, чтобы они могли уничтожить хранимые у себя ссылки. В общем случае простое удаление наблюдателей не годится, так как на них могут ссылаться другие объекты, и под их наблюдением могут находиться другие субъекты;

5) *гарантии целостности состояния субъекта перед отправкой уведомления.* Важно быть уверенным, что перед вызовом операции Notify состояние субъекта непротиворечиво, поскольку в процессе обновления собственного состояния наблюдатели будут опрашивать состояние субъекта. Правило непротиворечивости легко случайно нарушить, если операции одного из подклассов класса Subject вызывают унаследованные операции. Этой ловушки можно избежать, отправляя уведомления из шаблонных методов Subject. Определите примитивную операцию, замещаемую в подклассах, и обратитесь к Notify, используя последнюю операцию в шаблонном методе. В таком случае существует гарантия, что состояние объекта непротиворечиво, если операции Subject замещены в подклассах. Полезно фиксировать те операции, которые инициируют обновления.

6) *предотвращение зависимости протокола обновления от наблюдателя:* модели вытягивания ((pull model), когда субъект не посылает ничего, кроме минимального уведомления, а наблюдатели запрашивают детали позднее) и проталкивания ((push model), когда субъект посылает наблюдателям детальную информацию об изменении независимо от того, нужно ли им это). В реализациях паттерна наблюдатель субъект довольно часто транслирует всем подписчикам дополнительную информацию о характере изменения. Она передается в виде аргумента операции Update, и объем ее меняется в широких диапазонах.

7) *явное определение модификаций, представляющих интерес*. Эффективность обновления можно повысить, предоставив возможность при регистрации наблюдателя указать, какие именно события его интересуют (расширить интерфейс регистрации субъекта). Когда событие происходит, субъект информирует лишь тех наблюдателей, которые про явили к нему интерес.

8) *инкапсуляция сложной семантики обновления*. Если отношения зависимости между субъектами и наблюдателями становятся особенно сложными, то может потребоваться объект, инкапсулирующий эти отношения. Будем называть его ChangeManager (менеджер изменений). Он должен свести к минимуму объем работы, необходимой для того, чтобы наблюдатели смогли отразить изменения субъекта. Например, если некоторая операция влечет за собой изменения в нескольких независимых субъектах, то хотелось бы, чтобы наблюдатели уведомлялись после того, как будут модифицированы все субъекты, дабы не ставить в известность одного и того же наблюдателя несколько раз. У класса ChangeManager есть три обязанности:

• строить отображение между субъектом и его наблюдателями и предоставлять интерфейс для поддержания отображения в актуальном состоянии. Это освобождает субъектов от необходимости хранить ссылки на своих наблюдателей и наоборот;

• определять конкретную стратегию обновления;

• обновлять всех зависимых наблюдателей по запросу от субъекта.

ChangeManager — это пример паттерна посредник (Mediator). В общем случае есть только один объект ChangeManager, известный всем участникам. Поэтому полезен будет также и паттерн одиночка (Singletone);

* Пример кода

I.Интерфейс наблюдателя определен в абстрактном классе Observer. Такая реализация поддерживает несколько субъектов для одного наблюдателя. Передача субъекта в параметре операции Update позволяет наблюдателю определить, какой из наблюдаемых им субъектов изменился.

|  |
| --- |
| class Subject;  class Observer {  public:  virtual ~ Observer();  virtual void Update(Subject\* theChangedSubject) = 0;  protected:  Observer();  }; |

II. Аналогичным образом в абстрактном классе Subject определен интерфейс субъекта (\*)

III. ClockTimer — это конкретный субъект, который следит за временем суток. Он оповещает наблюдателей каждую секунду. Класс ClockTimer предоставляет интерфейс для получения отдельных компонентов времени: часа, минуты, секунды и т. д. (\*\*):

IV. Операция Tick вызывается через одинаковые интервалы внутренним таймером. Тем самым обеспечивается правильный отсчет времени. При этом обновляется внутреннее состояние объекта ClockTimer и вызывается операция Notify для извещения наблюдателей об изменении (\*\*):

V. Теперь можно определить класс DigitalClock для вывода времени. Свою графическую функциональность он наследует от класса Widget, предоставляемого библиотекой для построения пользовательских интерфейсов. Интерфейс наблюдателя примешивается к интерфейсу DigitalClock путем наследования от класса Observer (\*\*\*):

VI. Прежде чем начнется рисование часов посредством операции Update, будет проверено, что уведомление получено именно от объекта таймера.

Аналогичным образом определяется класс AnalogClock.

Следующий код создает объекты классов AnalogClock и DigitalClock, которые всегда показывают одно и то же время (\*\*\*\*):

При каждом срабатывании таймера timer оба экземпляра часов обновляются

и перерисовывают себя.

|  |  |  |  |
| --- | --- | --- | --- |
| (\*)class Subject {  public:  virtual~Subject();  virtual void Attach(Observer\*);  virtual void Detach(Observer\*);  virtual void Notify();  protected:  Subject();  private:  List<Observer\*> \*\_observers;  }; | (\*)void Subject::Attach (Observer\* o) {  \_observers->Append(o);  }  void Subject::Detach (Observer\* o) {  \_observers->Remove(o);  }  void Subject::Notify () {  ListIterator<Observer\*> i(\_observers);  for (i.First(); !i.IsDone(); i.Next()) {  i.CurrentItem()->Update(this);  }  } | (\*\*)class ClockTimer : public Subject {  public:  ClockTimer();  virtual int GetHour();  virtual int GetMinute();  virtual int GetSecond();  void Tick();  };  void ClockTimer::Tick () {  // Обновить внутреннее представление времени  // ...  Notify();  } | (\*\*\*)class DigitalClock: public Widget, public Observer {  public:  DigitalClock(ClockTimer\*);  virtual ~DigitalClock();  virtual void Update(Subject\*);  // Замещает операцию класса Observer  virtual void Draw();  // Замещает операцию класса Widget;  // определяет способ отображения часов  private:  ClockTimer\* \_subject;  };  DigitalClock::DigitalClock (ClockTimer\* s) {  \_subject = s;  \_subject->Attach(this);  }  DigitalClock:: DigitalClock () {  \_subject->Detach(this);  } |

|  |  |
| --- | --- |
| (\*\*\*\*) void DigitalClock::Update (Subject\* theChangedSubject) {  if (theChangedSubject == \_subject) {  Draw();  }  }  void DigitalClock::Draw () {  // Получить новые значения от субъекта  int hour = \_subject->GetHour();  int minute = \_subject->GetMinute();  // etc.  // Нарисовать цифровые часы  } | class AnalogClock : public Widget, public Observer {  public:  AnalogClock(ClockTimer\*);  virtual void Update(Subject\*);  virtual void Draw();  // ...  };  ClockTimer\* timer = new ClockTimer;  AnalogClock\* analogClock = new AnalogClock(timer);  DigitalClock\* digitalClock = new DigitalClock(timer); |