У интерфейса итерации, основанного на паттерне хранитель, есть два преимущества:

- с одной коллекцией может быть связано несколько активных состояний (то же самое верно и для паттерна итератор);
- не требуется нарушать инкапсуляцию коллекции для поддержки итерации. Хранитель интерпретируется только самой коллекцией, больше никто к нему доступа не имеет. При других подходах приходится нарушать инкапсуляцию, объявляя классы итераторов друзьями классов коллекций (см. описание паттерна итератор). В случае с хранителем ситуация противоположная: класс коллекции Collection является другом класса IteratorState.

В библиотеке QOCA для разрешения ограничений в хранителях содержится информация об изменениях. Клиент может получить хранитель, характеризующий текущее решение системы ограничений. В хранителе находятся только те переменные ограничений, которые были преобразованы со времени последнего решения. Обычно при каждом новом решении изменяется лишь небольшое подмножество переменных Solver. Но этого достаточно, чтобы вернуть Solver к предыдущему решению; для отката к более ранним решениям необходимо иметь все промежуточные хранители. Поэтому передавать хранители в произвольном порядке нельзя. QOCA использует механизм ведения истории для возврата к прежним решениям.

Родственные паттерны

Команда: команды помещают информацию о состоянии, необходимую для отмены выполненных действий, в хранители.

Итератор: хранители можно использовать для итераций, как было показано выше.

Паттерн Observer

Название и классификация паттерна

Наблюдатель - паттерн поведения объектов.

Назначение

Определяет зависимость типа «один ко многим» между объектами таким образом, что при изменении состояния одного объекта все зависящие от него оповещаются об этом и автоматически обновляются.

Известен также под именем

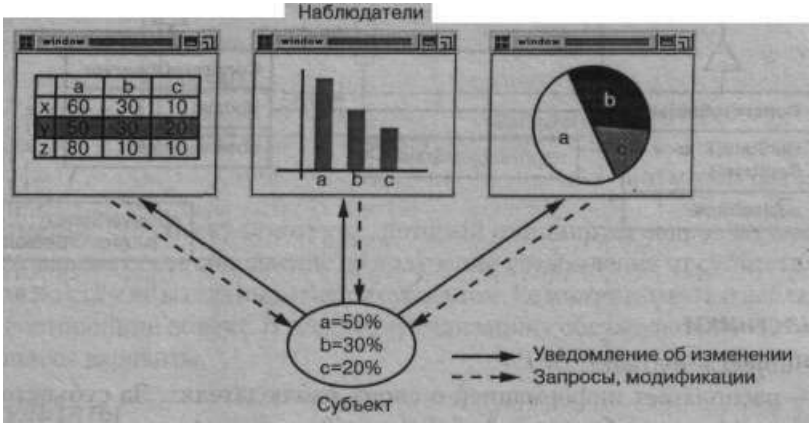
Dependents (подчиненные), Publish-Subscribe (издатель-подписчик).

Мотивация

В результате разбиения системы на множество совместно работающих классов появляется необходимость поддерживать согласованное состояние взаимосвязанных

объектов. Но не хотелось бы, чтобы за согласованность надо было платить жесткой связанностью классов, так как это в некоторой степени уменьшает возможности повторного использования.

Например, во многих библиотеках для построения графических интерфейсов пользователя презентационные аспекты интерфейса отделены от данных приложения [KP88, LVC89, P+88, WGM88]. С классами, описывающими данные и их представление, можно работать автономно. Электронная таблица и диаграмма не имеют информации друг о друге, поэтому вы вправе использовать их по отдельности. Но *ведут* они себя так, как будто «знают» друг о друге. Когда пользователь работает с таблицей, все изменения немедленно отражаются на диаграмме, и наоборот.



При таком поведении подразумевается, что и электронная таблица, и диаграмма зависят от данных объекта и поэтому должны уведомляться о любых изменениях в его состоянии. И нет никаких причин, ограничивающих количество зависимых объектов; для работы с одними и теми же данными может существовать любое число пользовательских интерфейсов.

Паттерн наблюдатель описывает, как устанавливать такие отношения. Ключевыми объектами в нем являются субъект и наблюдатель. У субъекта может быть сколько угодно зависимых от него наблюдателей. Все наблюдатели уведомляются об изменениях в состоянии субъекта. Получив уведомление, наблюдатель опрашивает субъекта, чтобы синхронизировать с ним свое состояние.

Такого рода взаимодействие часто называется отношением издатель-подписчик. Субъект издает или публикует уведомления и рассылает их, даже не имея информации о том, какие объекты являются подписчиками. На получение уведомлений может подписаться неограниченное количество наблюдателей.

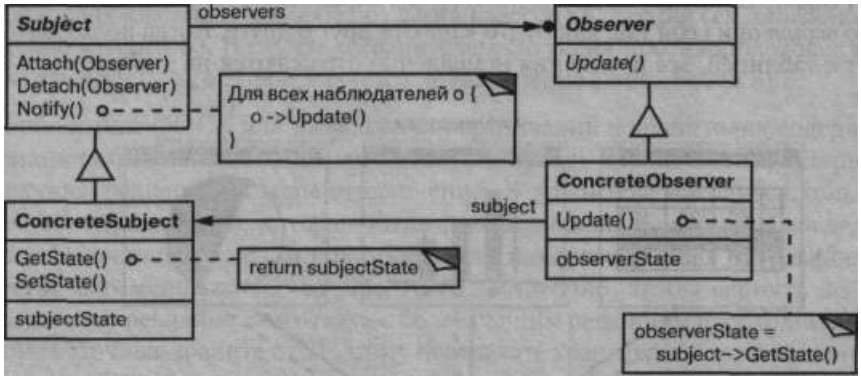
Применимость

Используйте паттерн наблюдатель в следующих ситуациях:

а когда у абстракции есть два аспекта, один из которых зависит от другого. Инкапсуляции этих аспектов в разные объекты позволяют изменять и повторно использовать их независимо;

- когда при модификации одного объекта требуется изменить другие и вы не знаете, сколько именно объектов нужно изменить;
- когда один объект должен оповещать других, не делая предположений об уведомляемых объектах. Другими словами, вы не хотите, чтобы объекты были тесно связаны между собой.

Структура



Участники

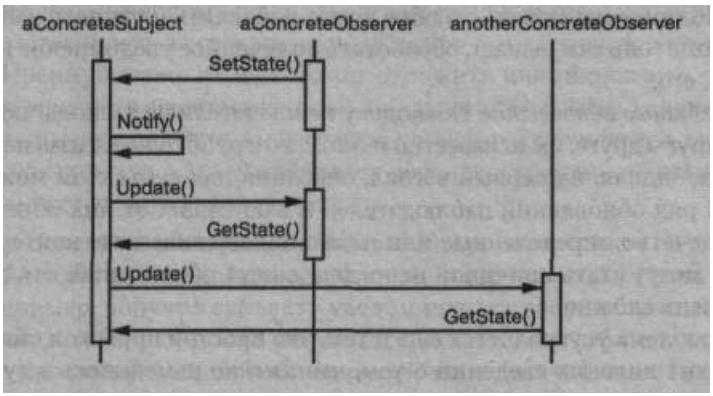
- **Subject** - субъект:
  - располагает информацией о своих наблюдателях. За субъектом может «следить» любое число наблюдателей;
  - предоставляет интерфейс для присоединения и отделения наблюдателей;
- **Observer** - наблюдатель:
  - определяет интерфейс обновления для объектов, которые должны быть уведомлены 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 ConcreteSubj ect;
  - 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.

На диаграмме взаимодействий показаны отношения между субъектом и двумя наблюдателями.



Отметим, что 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. В разделе «Реализация» обсуждаются часто встречающиеся варианты.

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 новых наблюдателей без модификации субъекта или других наблюдателей.

Рассмотрим некоторые достоинства и недостатки паттерна наблюдатель:

а *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.

Поскольку субъект и наблюдатель не являются тесно связанными, то 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 (компрометируя абстракцию уровня);

- *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. Субъекту не нужна информация о количестве таких объектов, от него требуется всего лишь уведомить своих наблюдателей. Поэтому мы можем в любое время добавлять и удалять наблюдателей. Наблюдатель сам решает, обработать полученное уведомление или игнорировать его;
- *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.

  Эта проблема усугубляется еще и тем, что простой протокол обновления не содержит никаких сведений о том, что *именно* изменилось в субъекте. Без дополнительного протокола, помогающего выяснить характер изменений, наблюдатели будут вынуждены проделать сложную работу для косвенного получения такой информации.

Реализация

В этом разделе обсуждаются вопросы, относящиеся к реализации механизма зависимостей:

- *отображение субъектов на наблюдателей.* С помощью этого простейшего способа субъект может отследить всех наблюдателей, которым он должен посылать уведомления, то есть хранить на них явные ссылки. Однако при наличии большого числа субъектов и всего нескольких наблюдателей это может оказаться накладно. Один из возможных компромиссов в пользу экономии памяти за счет времени состоит в том, чтобы использовать ассоциативный массив (например, хэш-таблицу) 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;
- *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 качестве параметра операции Update, тем самым сообщая наблюдателю, что именно нужно обследовать;
- *кто инициирует обновление.* Чтобы сохранить согласованность, субъект и его наблюдатели полагаются на механизм уведомлений. Но какой именно объект вызывает операцию-Notify для инициирования обновления? Есть два варианта:

- операции класса Subj ect, изменившие состояние, вызывают Noti fy 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 следующем: при выполнении каждой из нескольких последовательных операций будут производиться обновления, что может стать причиной неэффективной работы 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;

а *висячие ссылки на удаленные субъекты.* Удаление субъекта не должно приводить к появлению висячих ссылок у наблюдателей. Избежать этого можно, например, поручив субъекту уведомлять все свои наблюдатели 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;

а *гарантии непротиворечивости 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 очень легко нарушить, если операции одного из подклассов класса Subject вызывают унаследованные операции. Например, в следующем фрагменте уведомление отправляется, когда состояние субъекта противоречиво:

```
void MySubject::0peration (int newValue) {
    BaseClassSubject::Operation(newValue);
        // отправить уведомление

    _my!nstVar += newValue;
        // обновить состояние подкласса (слишком поздно!)
}
```

Избежать этой ловушки можно, отправляя уведомления из шаблонных методов (см. описание паттерна шаблонный метод) абстрактного класса 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:

```
void Text::Cut (TextRange r) {
    ReplaceRange(r); // переопределена в подклассах
    Notify();
}
```

Кстати, всегда желательно фиксировать, какие операции класса Subject инициируют обновления;

- *как избежать зависимости протокола обновления от наблюдателя: модели вытягивания и проталкивания.* 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.

  На одном полюсе находится так называемая *модель проталкивания* (push model), когда субъект посылает наблюдателям детальную информацию об изменении независимо от того, нужно ли им это. На другом - *модель вытягивания* (pull model), когда субъект не посылает ничего, кроме минимального уведомления, а наблюдатели запрашивают детали позднее.

  В модели вытягивания подчеркивается неинформированность субъекта о своих наблюдателях, а в модели проталкивания предполагается, что субъект владеет определенной информацией о потребностях наблюдателей. В случае применения модели проталкивания степень повторного их использования может снизиться, так как классы Subj ect предполагают о классах Observer, которые не всегда могут быть верны. С другой стороны, модель вытягивания может оказаться неэффективной, ибо наблюдателям без помощи субъекта необходимо выяснять, что изменилось;

- *явное специфицирование представляющих интерес модификаций* Эффективность обновления можно повысить, расширив интерфейс регистрации субъекта, то есть предоставив возможность при регистрации наблюдателя указать, какие события его интересуют. Когда событие происходит, субъект информирует лишь тех наблюдателей, которые проявили к нему интерес. Чтобы получать конкретное событие, наблюдатели присоединяются к своим субъектам следующим образом:

```
void Subject::Attach(Observer*, Aspects interest);
```

  где `interest` определяет представляющее интерес событие. В момент посылки уведомления субъект передает своим наблюдателям изменившийся аспект в виде параметра операции Update. Например:

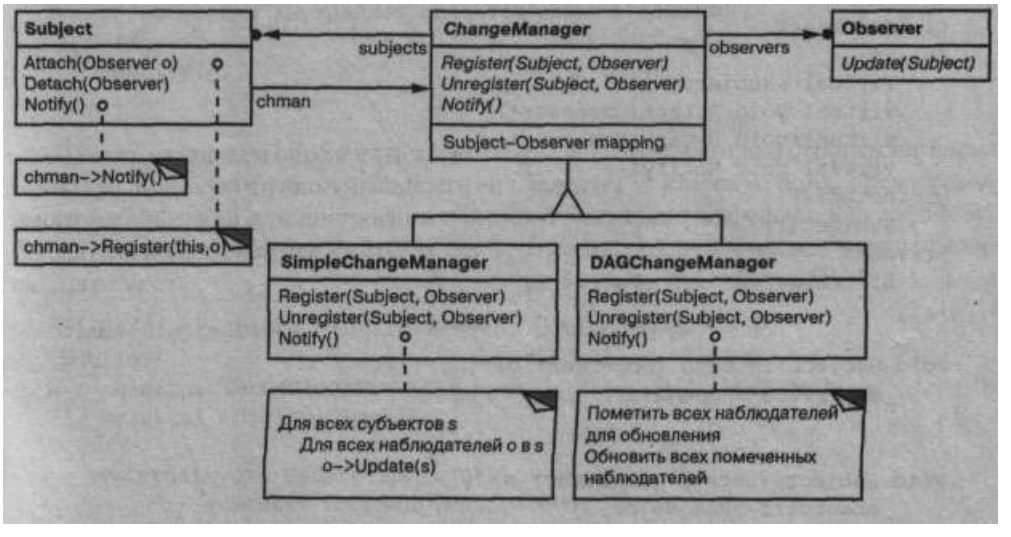
```
void Observer::Update(Subject*, Aspects interest);
```

- *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 служит для минимизации объема 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.

На следующей диаграмме представлена простая реализация паттерна наблюдатель с использованием менеджера изменений ChangeManager. Имеется два специализированных менеджера. SimplechangeManager всегда обновляет всех наблюдателей каждого субъекта, а DAGChangeManager обрабатывает направленные ациклические графы зависимостей между субъектами и их наблюдателями. Когда наблюдатель должен «присматривать» за несколькими субъектами, предпочтительнее использовать DAGChangeManager. В этом случае изменение сразу двух или более субъектов может привести к избыточным обновлениям. Объект DAGChangeManager гарантирует, что наблюдатель в любом случае получит только одно уведомление. Если обновление одного и того же наблюдателя допускается несколько раз подряд, то вполне достаточно объекта SimplechangeManager.



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

а *комбинирование классов Subject и Observer,* В библиотеках классов, которые написаны на языках, не поддерживающих множественного наследования (например, на Smalltalk), обычно не определяются отдельные классы Subject и Observer. Их интерфейсы комбинируются в одном классе. Это позволяет определить объект, выступающий в роли одновременно субъекта

и наблюдателя, без множественного наследования. Так, в Smalltalk интерфейсы Subj ect и Observer определены в корневом классе Obj ect и потому доступны вообще всем классам.

Пример кода

Интерфейс наблюдателя определен в абстрактном классе Observer:

```
class Subject;

class Observer {
public:
    virtual -Observer();
    virtual void Update(Subject* theChangedSubject) = 0;
protected:
    Observer(),-
};
```

При такой реализации поддерживается несколько субъектов для одного наблюдателя. Передача субъекта параметром операции Update позволяет наблюдателю определить, какой из наблюдаемых им субъектов изменился.

Таким же образом в абстрактном классе Subject определен интерфейс субъекта:

```
class Subject {
public:
    virtual -Subject()
    virtual void Attach(Observer*);
    virtual void Detach(Observer*);
    virtual void Notify();
protected:
    Subject();
private:
    List<0bserver*>  *_observers;
};

void Subject::Attach (Observer* o) {
    _observers->Append(o);
}

void Subject::Detach (Observer* o) {
    _observers->Remove(o);
}

void Subject::Notify () {
    ListIterator<0bserver*> i(„observers);

    for (i.First(); !i.IsDone(); i.NextO) {
        i.CurrentItemf)->Update(this);
    }
}
```

ClockTimer - это конкретный субъект, который следит за временем суток. Он извещает наблюдателей каждую секунду. Класс ClockTimer предоставляет интерфейс для получения отдельных компонентов времени: часа, минуты, секунды и т.д.:

```
class ClockTimer : public Subject {
public:
    ClockTimer();

    virtual int GetfHour();
    virtual int GetMinute();
    virtual int GetSecond(') ;

    void Tick();
};
```

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:

```
void ClockTimer::Tick () {
    // обновить внутреннее представление о времени
    // ...
    Notify();
}
```

Теперь мы можем определить класс DigitalClock, который отображает время. Свою графическую функциональность он наследует от класса Widget, предоставляемого библиотекой для построения пользовательских интерфейсов. Интерфейс наблюдателя подмешивается к интерфейсу DigitalClock путем наследования от класса Observer:

```
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 изображения часов
private:
    ClockTimer* _subject;
};

DigitalClock::DigitalClock (ClockTimer* s) {
```

```
    _subject = s;
    _subject->Attach(this);
}

DigitalClock::~DigitalClock () {
    _subject->Detach(this);
}
```

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

```
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();
    // и т.д.

    // нарисовать цифровые часы
}
```

Аналогично можно определить класс AnalogClock:

```
class AnalogClock : public Widget, public Observer {
public:
    AnalogClock(ClockTimer*);
    virtual void Update(Subject*);
    virtual void Draw();
    // ...
};
```

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

```
ClockTimer* timer = new- ClockTimer;
AnalogClock* analogclock = new AnalogClock ( timer ) ;
DigitalClock* digitalClock = new DigitalClock(timer) ;
```

При каждом срабатывании таймера timer оба экземпляра часов обновляются и перерисовывают себя.

Известные применения

Первый и, возможно, самый известный пример паттерна наблюдатель появился в схеме модель/вид/контроллер (МУС) языка Smalltalk, которая представляет собой каркас для построения пользовательских интерфейсов в среде

Smalltalk [KP88]. Класс Model в MVC - это субъект, а View - базовый-класс для наблюдателей. В языках Smalltalk, ET++ [WGM88] и библиотеке классов THINK [Sym93b] предлагается общий механизм зависимостей, в котором интерфейсы субъекта и наблюдателя помещены в класс, являющийся общим родителем всех остальных системных классов.

Среди других библиотек для построения интерфейсов пользователя, в которых используется паттерн наблюдатель, стоит упомянуть Interviews [LVC89], Andrew Toolkit [P+88] и Unidraw [VL90]. В Interviews явно определены классы Observer и Observable (для субъектов). В библиотеке Andrew они называются *видом* (view) и *объектом данных* (data object) соответственно. Unidraw делит объекты графического редактора на части View (для наблюдателей) и Subject.

Родственные паттерны

Посредник: класс ChangeManager действует как посредник между субъектами и наблюдателями, инкапсулируя сложную семантику обновления.

Одиночка: класс ChangeManager может воспользоваться паттерном одиночка, чтобы гарантировать уникальность и глобальную доступность менеджера изменений.

Паттерн State

Название и классификация паттерна

Состояние - паттерн поведения объектов.

Назначение

Позволяет объекту варьировать свое поведение в зависимости от внутреннего состояния. Извне создается впечатление, что изменился класс объекта.

Мотивация

Рассмотрим класс TCPConnection, с помощью которого представлено сетевое соединение. Объект этого класса может находиться в одном из нескольких состояний: Established (установлено), Listening (прослушивание), Closed (закрыто). Когда объект TCPConnection получает запросы от других объектов, то в зависимости от текущего состояния он отвечает по-разному. Например, ответ на запрос Open (открыть) зависит от того, находится ли соединение в состоянии Closed или Established. Паттерн состояние описывает, каким образом объект TCPConnect ion может вести себя по-разному, находясь в различных состояниях.

Основная идея этого паттерна заключается в том, чтобы ввести абстрактный класс TCPState для представления различных состояний соединения. Этот класс объявляет интерфейс, общий для всех классов, описывающих различные рабочие состояния. В подклассах TCPState реализовано поведение, специфичное для конкретного состояния. Например, в классах TCPEstablished и TCPClosed реализовано поведение, характерное для состояний Established и Closed соответственно.