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.

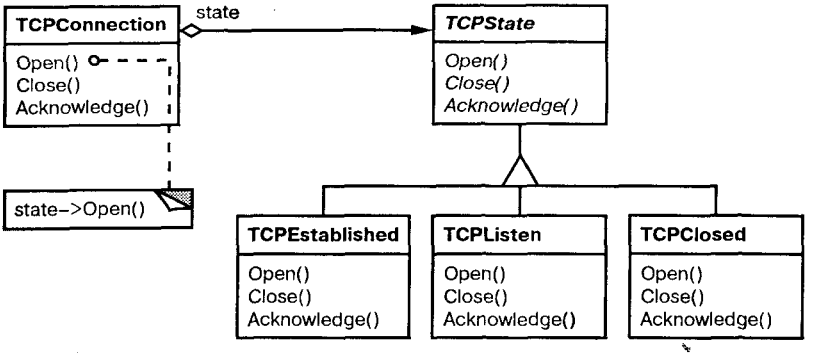
Назначение

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 `TCPConnection` 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.



Класс `TCPConnection` хранит у себя объект состояния (экземпляр некоторого подкласса `TCPState`), представляющий текущее состояние соединения, и делегирует все зависящие от состояния запросы этому объекту. `TCPConnection` использует свой экземпляр подкласса `TCPState` для выполнения операций, свойственных только данному состоянию соединения.

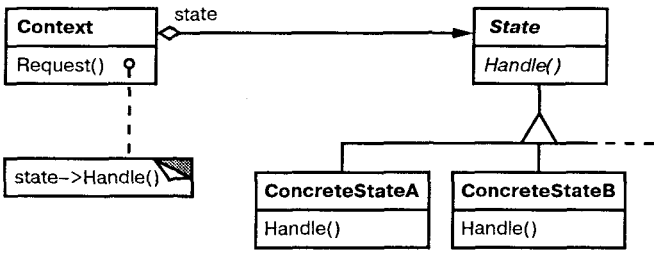
При каждом изменении состояния соединения `TCPConnection` изменяет свой объект-состояние. Например, когда установленное соединение закрывается, `TCPConnection` заменяет экземпляр класса `TCPEstablished` экземпляром `TCPClosed`.

Применимость

Используйте паттерн **состояние** в следующих случаях:

- когда поведение объекта зависит от его состояния и должно изменяться во время выполнения;
- когда в коде операций встречаются состоящие из многих ветвей условные операторы, в которых выбор ветви зависит от состояния. Обычно в таком случае состояние представлено перечисляемыми константами. Часто одна и та же структура условного оператора повторяется в нескольких операциях. Паттерн **состояние** предлагает поместить каждую ветвь в отдельный класс. Это позволяет трактовать состояние объекта как самостоятельный объект, который может изменяться независимо от других.

Структура



Участники

- **Context** (`TCPConnection`) – контекст:
  - определяет интерфейс, представляющий интерес для клиентов;
  - хранит экземпляр подкласса `ConcreteState`, которым определяется текущее состояние;
- **State** (`TCPState`) – состояние:
  - определяет интерфейс для инкапсуляции поведения, ассоциированного с конкретным состоянием контекста `Context`;
- **Подклассы ConcreteState** (`TCPEstablished`, `TCPListen`, `TCPClosed`) – конкретное состояние:
  - каждый подкласс реализует поведение, ассоциированное с некоторым состоянием контекста `Context`.

Отношения

- класс `Context` делегирует зависящие от состояния запросы текущему объекту `ConcreteState`;
- контекст может передать себя в качестве аргумента объекту `State`, который будет обрабатывать запрос. Это дает возможность объекту-состоянию при необходимости получить доступ к контексту;
- `Context` – это основной интерфейс для клиентов. Клиенты могут конфигурировать контекст объектами состояния `State`. Один раз сконфигурировав контекст, клиенты уже не должны напрямую связываться с объектами состояния;
- либо `Context`, либо подклассы `ConcreteState` могут решить, при каких условиях и в каком порядке происходит смена состояний.

Результаты

Результаты использования паттерна состояние:

- *локализует зависящее от состояния поведение и делит его на части, соответствующие состояниям.* Паттерн состояние помещает все поведение, ассоциированное с конкретным состоянием, в отдельный объект. Поскольку зависящий от состояния код целиком находится в одном из подклассов класса `State`, то добавлять новые состояния и переходы можно просто путем порождения новых подклассов.

  Вместо этого можно было бы использовать данные-члены для определения внутренних состояний, тогда операции объекта `Context` проверяли бы эти данные. Но в таком случае похожие условные операторы или операторы ветвления были бы разбросаны по всему коду класса `Context`. При этом добавление нового состояния потребовало бы изменения нескольких операций, что затруднило бы сопровождение.

  Паттерн состояние позволяет решить эту проблему, но одновременно порождает другую, поскольку поведение для различных состояний оказывается распределенным между несколькими подклассами `State`. Это увеличивает число классов. Конечно, один класс компактнее, но если состояний

много, то такое распределение эффективнее, так как в противном случае пришлось бы иметь дело с громоздкими условными операторами.

Наличие громоздких условных операторов нежелательно, равно как и наличие длинных процедур. Они слишком монолитны, вот почему модификация и расширение кода становится проблемой. Паттерн **состояние** предлагает более удачный способ структурирования зависящего от состояния кода. Логика, описывающая переходы между состояниями, больше не заключена в монолитные операторы `if` или `switch`, а распределена между подклассами `State`. При инкапсуляции каждого перехода и действия в класс состояние становится полноценным объектом. Это улучшает структуру кода и проясняет его назначение;

- *делает явными переходы между состояниями.* Если объект определяет свое текущее состояние исключительно в терминах внутренних данных, то переходы между состояниями не имеют явного представления; они проявляются лишь как присваивания некоторым переменным. Ввод отдельных объектов для различных состояний делает переходы более явными. Кроме того, объекты `State` могут защитить контекст `Context` от рассогласования внутренних переменных, поскольку переходы с точки зрения контекста – это атомарные действия. Для осуществления перехода надо изменить значение только одной переменной (объектной переменной `State` в классе `Context`), а не нескольких [dCLF93];
- *объекты состояния можно разделять.* Если в объекте состояния `State` отсутствуют переменные экземпляра, то есть представляемое им состояние кодируется исключительно самим типом, то разные контексты могут разделять один и тот же объект `State`. Когда состояния разделяются таким образом, они являются, по сути дела, приспособленцами (см. описание паттерна **приспособленец**), у которых нет внутреннего состояния, а есть только поведение.

Реализация

С паттерном **состояние** связан целый ряд вопросов реализации:

- *что определяет переходы между состояниями.* Паттерн **состояние** ничего не сообщает о том, какой участник определяет критерий перехода между состояниями. Если критерии зафиксированы, то их можно реализовать непосредственно в классе `Context`. Однако в общем случае более гибкий и правильный подход заключается в том, чтобы позволить самим подклассам класса `State` определять следующее состояние и момент перехода. Для этого в класс `Context` надо добавить интерфейс, позволяющий объектам `State` установить состояние контекста.

  Такую децентрализованную логику переходов проще модифицировать и расширять – нужно лишь определить новые подклассы `State`. Недостаток децентрализации в том, что каждый подкласс `State` должен «знать» еще хотя бы об одном подклассе, что вносит реализационные зависимости между подклассами;

- *табличная альтернатива.* Том Каргилл (Tom Cargill) в книге *C++ Programming Style* [Car92] описывает другой способ структурирования кода, управляемого сменой состояний. Он использует таблицу для отображения входных данных на переходы между состояниями. С ее помощью можно определить, в какое состояние нужно перейти при поступлении некоторых входных данных. По существу, тем самым мы заменяем условный код (или виртуальные функции, если речь идет о паттерне **состояние**) поиском в таблице.
  Основное преимущество таблиц – в их регулярности: для изменения критериев перехода достаточно модифицировать только данные, а не код. Но есть и недостатки:
  - поиск в таблице часто менее эффективен, чем вызов функции (виртуальной);
  - представление логики переходов в однородном табличном формате делает критерии менее явными и, стало быть, более сложными для понимания;
  - обычно трудно добавить действия, которыми сопровождаются переходы между состояниями. Табличный метод учитывает состояния и переходы между ними, но его необходимо дополнить, чтобы при каждом изменении состоянии можно было выполнять произвольные вычисления.

  Главное различие между конечными автоматами на базе таблиц и паттерном **состояние** можно сформулировать так: паттерн **состояние** моделирует поведение, зависящее от состояния, а табличный метод акцентирует внимание на определении переходов между состояниями;
- *создание и уничтожение объектов состояния.* В процессе разработки обычно приходится выбирать между:
  - созданием объектов состояния, когда в них возникает необходимость, и уничтожением сразу после использования;
  - созданием их заранее и навсегда.

  Первый вариант предпочтителен, когда заранее неизвестно, в какие состояния будет попадать система, и контекст изменяет состояние сравнительно редко. При этом мы не создаем объектов, которые никогда не будут использованы, что существенно, если в объектах состояния хранится много информации. Когда изменения состояния происходят часто, поэтому не хотелось бы уничтожать представляющие их объекты (ибо они могут очень скоро понадобиться вновь), следует воспользоваться вторым подходом. Время на создание объектов затрачивается только один раз, в самом начале, а на уничтожение – не затрачивается вовсе. Правда, этот подход может оказаться неудобным, так как в контексте должны храниться ссылки на все состояния, в которые система теоретически может попасть;
- *использование динамического наследования.* Варьировать поведение по запросу можно, меняя класс объекта во время выполнения, но в большинстве объектно-ориентированных языков это не поддерживается. Исключение составляет `Self` [US87] и другие основанные на делегировании языки, которые предоставляют такой механизм и, следовательно, поддерживают паттерн **состояние** напрямую. Объекты в языке `Self` могут делегировать операции

другим объектам, обеспечивая тем самым некую форму динамического наследования. С изменением целевого объекта делегирования во время выполнения, по существу, изменяется и структура графа наследования. Такой механизм позволяет объектам варьировать поведение путем изменения своего класса.

Пример кода

В следующем примере приведен код на языке C++ с TCP-соединением из раздела «Мотивация». Это упрощенный вариант протокола TCP, в нем, конечно же, представлен не весь протокол и даже не все состояния TCP-соединений.[1]

Прежде всего определим класс `TCPConnection`, который предоставляет интерфейс для передачи данных и обрабатывает запросы на изменение состояния:

```
class TCPOctetStream;
class TCPState;

class TCPConnection {
public:
    TCPConnection();

    void ActiveOpen();
    void PassiveOpen();
    void Close();

    void Send();
    void Acknowledge();
    void Synchronize();

    void ProcessOctet(TCPOctetStream*);
private:
    friend class TCPState;
    void ChangeState(TCPState*);
private:
    TCPState* _state;
};
```

В переменной-члене `_state` класса `TCPConnection` хранится экземпляр класса `TCPState`. Этот класс дублирует интерфейс изменения состояния, определенный в классе `TCPConnection`. Каждая операция `TCPState` принимает экземпляр `TCPConnection` как параметр, тем самым позволяя объекту `TCPState` получить доступ к данным объекта `TCPConnection` и изменить состояние соединения:

```
class TCPState {
public:
    virtual void Transmit(TCPConnection*, TCPOctetStream*);
    virtual void ActiveOpen(TCPConnection*);
    virtual void PassiveOpen(TCPConnection*);
    virtual void Close(TCPConnection*);
```

[1] Пример основан на описании протокола установления TCP-соединений, приведенном в книге Линча и Роуза [LR93].

```
    virtual void Synchronize(TCPConnection*);
    virtual void Acknowledge(TCPConnection*);
    virtual void Send(TCPConnection*);
protected:
    void ChangeState(TCPConnection*, TCPState*);
};
```

TCPConnection делегирует все зависящие от состояния запросы хранимому в _state экземпляру TCPState. Кроме того, в классе TCPConnection существует операция, с помощью которой в эту переменную можно записать указатель на другой объект TCPState. Конструктор класса TCPConnection инициализирует _state указателем на состояние TCPClosed (мы определим его ниже):

```
TCPConnection::TCPConnection () {
    _state = TCPClosed::Instance();
}

void TCPConnection::ChangeState (TCPState* s) {
    _state = s;
}

void TCPConnection::ActiveOpen () {
    _state->ActiveOpen(this);
}

void TCPConnection::PassiveOpen () {
    _state->PassiveOpen(this);
}


void TCPConnection::Close () {
    _state->Close(this);
}

void TCPConnection::Acknowledge () {
    _state->Acknowledge(this);
}

void TCPConnection::Synchronize () {
    _state->Synchronize(this);
}
```

В классе TCPState реализовано поведение по умолчанию для всех делегированных ему запросов. Он может также изменить состояние объекта TCPConnection посредством операции ChangeState. TCPState объявляется другом класса TCPConnection, что дает ему привилегированный доступ к этой операции:

```
void TCPState::Transmit (TCPConnection*, TCPOctetStream*) { }
void TCPState::ActiveOpen (TCPConnection*) { }
void TCPState::PassiveOpen (TCPConnection*) { }
```

```
void TCPState::Close (TCPConnection*) { }
void TCPState::Synchronize (TCPConnection*) { }

void TCPState::ChangeState (TCPConnection* t, TCPState* s) {
    t->ChangeState(s);
}
```

В подклассах TCPState реализовано поведение, зависящее от состояния. Соединение TCP может находиться во многих состояниях: Established (установлено), Listening (прослушивание), Closed (закрыто) и т.д., и для каждого из них есть свой подкласс TCPState. Мы подробно рассмотрим три подкласса – TCPEstablished, TCPListen и TCPClosed:

```
class TCPEstablished : public TCPState {
public:
    static TCPState* Instance();

    virtual void Transmit(TCPConnection*, TCPOctetStream*);
    virtual void Close(TCPConnection*);
};

class TCPListen : public TCPState {
public:
    static TCPState* Instance();

    virtual void Send(TCPConnection*);
    // ...
};

class TCPClosed : public TCPState {
public:
    static TCPState* Instance();

    virtual void ActiveOpen(TCPConnection*);
    virtual void PassiveOpen(TCPConnection*);
    // ...
};
```

В подклассах TCPState нет никакого локального состояния, поэтому их можно разделять, так что потребуется только по одному экземпляру каждого класса. Уникальный экземпляр подкласса TCPState создается обращением к статической операции Instance.[1]

В подклассах TCPState реализовано зависящее от состояния поведение для тех запросов, которые допустимы в этом состоянии:

```
void TCPClosed::ActiveOpen (TCPConnection* t) {
    // послать SYN, получить SYN, ACK и т.д.

    ChangeState(t, TCPEstablished::Instance());
}
```

[1] Таким образом, каждый подкласс TCPState – это **одиночка**.

```
void TCPClosed::PassiveOpen (TCPConnection* t) {
    ChangeState(t, TCPListen::Instance());
}

void TCPEstablished::Close (TCPConnection* t) {
    // послать FIN, получить ACK для FIN

    ChangeState(t, TCPListen::Instance());
}

void TCPEstablished::Transmit (
    TCPConnection* t, TCPOctetStream* o
) {
    t->ProcessOctet(o);
}

void TCPListen::Send (TCPConnection* t) {
    // послать SYN, получить SYN, ACK и т.д.

    ChangeState(t, TCPEstablished::Instance());
}
```

После выполнения специфичных для своего состояния действий эти операции вызывают `ChangeState` для изменения состояния объекта `TCPConnection`. У него нет никакой информации о протоколе TCP. Именно подклассы `TCPState` определяют переходы между состояниями и действия, диктуемые протоколом.

Известные применения

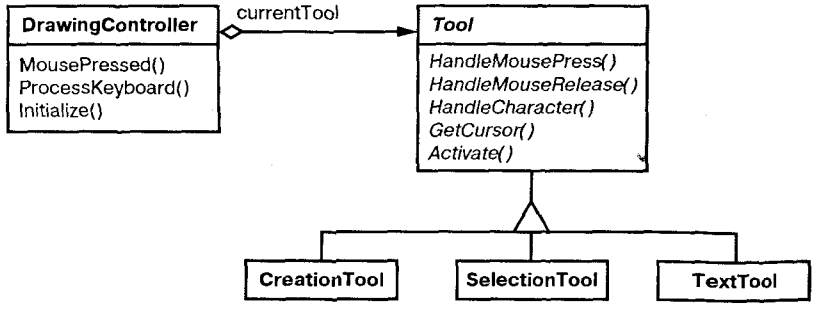
Ральф Джонсон и Джонатан Цвейг [JZ91] характеризуют паттерн состояние и описывают его применительно к протоколу TCP.

Наиболее популярные интерактивные программы рисования предоставляют «инструменты» для выполнения операций прямым манипулированием. Например, инструмент для рисования линий позволяет пользователю щелкнуть в произвольной точке мышью, а затем, перемещая мышь, провести из этой точки линию. Инструмент для выбора позволяет выбирать некоторые фигуры. Обычно все имеющиеся инструменты размещаются в палитре. Работа пользователя заключается в том, чтобы выбрать и применить инструмент, но на самом деле поведение редактора варьируется при смене инструмента: посредством инструмента для рисования мы создаем фигуры, при помощи инструмента выбора – выбираем их и т.д. Чтобы отразить зависимость поведения редактора от текущего инструмента, можно воспользоваться паттерном состояние.

Можно определить абстрактный класс `Tool`, подклассы которого реализуют зависящее от инструмента поведение. Графический редактор хранит ссылку на текущий объект `Tool` и делегирует ему поступающие запросы. При выборе инструмента редактор использует другой объект, что приводит к изменению поведения.

Данная техника используется в каркасах графических редакторов HotDraw [Joh92] и Unidraw [VL90]. Она позволяет клиентам легко определять новые виды

инструментов. В HotDraw класс `DrawingController` переадресует запросы текущему объекту `Tool`. В Unidraw соответствующие классы называются `Viewer` и `Tool`. На приведенной ниже диаграмме классов схематично представлены интерфейсы классов `Tool` и `DrawingController`.



Описанный Джеймсом Коплиеном [Cop92] прием конверт-письмо (Envelope-Letter) также относится к паттерну состояние. Техника конверт-письмо – это способ изменить класс объекта во время выполнения. Паттерн состояние является частным случаем, в нем акцент делается на работу с объектами, поведение которых зависит от состояния.

Родственные паттерны

Паттерн приспособленец подсказывает, как и когда можно разделять объекты класса `State`.

Объекты класса `State` часто бывают одиночками.

Паттерн Strategy

Название и классификация паттерна

Стратегия – паттерн поведения объектов.

Назначение

Определяет семейство алгоритмов, инкапсулирует каждый из них и делает их взаимозаменяемыми. Стратегия позволяет изменять алгоритмы независимо от клиентов, которые ими пользуются.

Известен также под именем

Policy (политика).

Мотивация

Существует много алгоритмов для разбиения текста на строки. Жестко «зашивать» все подобные алгоритмы в классы, которые в них нуждаются, нежелательно по нескольким причинам: