# Паттерн State (состояние)

## Назначение паттерна State

* Паттерн State позволяет объекту изменять свое поведение в зависимости от внутреннего состояния. Создается впечатление, что объект изменил свой класс.
* Паттерн State является объектно-ориентированной реализацией конечного автомата.

## Решаемая проблема

Поведение объекта зависит от его состояния и должно изменяться во время выполнения программы. Такую схему можно реализовать, применив множество условных операторов: на основе анализа текущего состояния объекта предпринимаются определенные действия. Однако при большом числе состояний условные операторы будут разбросаны по всему коду, и такую программу будет трудно поддерживать.

## Обсуждение паттерна State

Паттерн State решает указанную проблему следующим образом:

* Вводит класс Context, в котором определяется интерфейс для внешнего мира.
* Вводит абстрактный класс State.
* Представляет различные "состояния" конечного автомата в виде подклассов State.
* В классе Context имеется указатель на текущее состояние, который изменяется при изменении состояния конечного автомата.

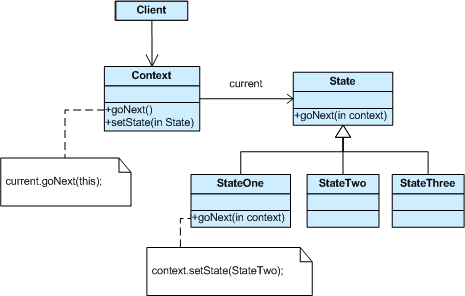
Паттерн State не определяет, где именно определяется условие перехода в новое состояние. Существует два варианта: класс Context или подклассы State. Преимущество последнего варианта заключается в простоте добавления новых производных классов. Недостаток заключается в том, что каждый подкласс State для осуществления перехода в новое состояние должен знать о своих соседях, что вводит зависимости между подклассами.

Существует также альтернативный таблично-ориентированный подход к проектированию конечных автоматов, основанный на использовании таблицы однозначного отображения входных данных на переходы между состояниями. Однако этот подход обладает недостатками: трудно добавить выполнение действий при выполнении переходов. Подход, основанный на использовании паттерна State, для осуществления переходов между состояниями использует код (вместо структур данных), поэтому эти действия легко добавляемы.

## Структура паттерна State

Класс Context определяет внешний интерфейс для клиентов и хранит внутри себя ссылку на текущее состояние объекта State. Интерфейс абстрактного базового класса State повторяет интерфейс Context за исключением одного дополнительного параметра - указателя на экземпляр Context. Производные от State классы определяют поведение, специфичное для конкретного состояния. Класс "обертка" Context делегирует все полученные запросы объекту "текущее состояние", который может использовать полученный дополнительный параметр для доступа к экземпляру Context.

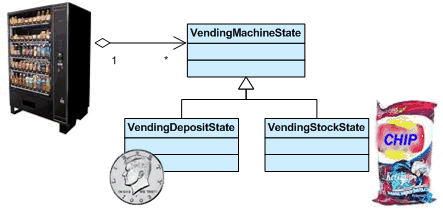
### UML-диаграмма классов паттерна State



## Пример паттерна State

Паттерн State позволяет объекту изменять свое поведение в зависимости от внутреннего состояния. Похожая картина может наблюдаться в работе торгового автомата. Автоматы могут иметь различные состояния в зависимости от наличия товаров, суммы полученных монет, возможности размена денег и т.д. После того как покупатель выбрал и оплатил товар, возможны следующие ситуации (состояния):

* Выдать покупателю товар, выдавать сдачу не требуется.
* Выдать покупателю товар и сдачу.
* Покупатель товар не получит из-за отсутствия достаточной суммы денег.
* Покупатель товар не получит из-за его отсутствия.



## Использование паттерна State

* Определите существующий или создайте новый класс-"обертку" Context, который будет использоваться клиентом в качестве "конечного автомата".
* Создайте базовый класс State, который повторяет интерфейс класса Context. Каждый метод принимает один дополнительный параметр: экземпляр класса Context. Класс State может определять любое полезное поведение "по умолчанию".
* Создайте производные от State классы для всех возможных состояний.
* Класс-"обертка" Context имеет ссылку на объект "текущее состояние".
* Все полученные от клиента запросы класс Context просто делегирует объекту "текущее состояние", при этом в качестве дополнительного параметра передается адрес объекта Context.
* Используя этот адрес, в случае необходимости методы класса State могут изменить "текущее состояние" класса Context.

## Особенности паттерна State

* Объекты класса State часто бывают [одиночками](http://cpp-reference.ru/patterns/creational-patterns/singleton/).
* [Flyweight](http://cpp-reference.ru/patterns/structural-patterns/flyweight/) показывает, как и когда можно разделять объекты State.
* [Паттерн Interpreter](http://cpp-reference.ru/patterns/behavioral-patterns/interpreter/) может использовать State для определения контекстов при синтаксическом разборе.
* Паттерны State и [Bridge](http://cpp-reference.ru/patterns/structural-patterns/bridge/) имеют схожие структуры за исключением того, что Bridge допускает иерархию классов-конвертов (аналогов классов-"оберток"), а State-нет. Эти паттерны имеют схожие структуры, но решают разные задачи: State позволяет объекту изменять свое поведение в зависимости от внутреннего состояния, в то время как Bridge разделяет абстракцию от ее реализации так, что их можно изменять независимо друг от друга.
* Реализация паттерна State основана на паттерне [Strategy](http://cpp-reference.ru/patterns/behavioral-patterns/strategy/). Различия заключаются в их назначении.

## Реализация паттерна State

Рассмотрим пример конечного автомата с двумя возможными состояниями и двумя событиями.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90 | #include <iostream>  using namespace std;  class Machine  {    class State \*current;    public:      Machine();      void setCurrent(State \*s)      {          current = s;      }      void on();      void off();  };    class State  {    public:      virtual void on(Machine \*m)      {          cout << "   already ON\n";      }      virtual void off(Machine \*m)      {          cout << "   already OFF\n";      }  };    void Machine::on()  {    current->on(this);  }    void Machine::off()  {    current->off(this);  }    class ON: public State  {    public:      ON()      {          cout << "   ON-ctor ";      };      ~ON()      {          cout << "   dtor-ON\n";      };      void off(Machine \*m);  };    class OFF: public State  {    public:      OFF()      {          cout << "   OFF-ctor ";      };      ~OFF()      {          cout << "   dtor-OFF\n";      };      void on(Machine \*m)      {          cout << "   going from OFF to ON";          m->setCurrent(new ON());          delete this;      }  };    void ON::off(Machine \*m)  {    cout << "   going from ON to OFF";    m->setCurrent(new OFF());    delete this;  }    Machine::Machine()  {    current = new OFF();    cout << '\n';  }    int main()  {    void(Machine:: \*ptrs[])() =    {      Machine::off, Machine::on    };    Machine fsm;    int num;    while (1)    {      cout << "Enter 0/1: ";      cin >> num;      (fsm. \*ptrs[num])();    }  } |

Вывод программы:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | OFF-ctor  Enter 0/1: 0     already OFF  Enter 0/1: 1     going from OFF to ON   ON-ctor    dtor-OFF  Enter 0/1: 1     already ON  Enter 0/1: 0     going from ON to OFF   OFF-ctor    dtor-ON  Enter 0/1: 1     going from OFF to ON   ON-ctor    dtor-OFF  Enter 0/1: 0     going from ON to OFF   OFF-ctor    dtor-ON  Enter 0/1: 0     already OFF  Enter 0/1: |

Источник: <http://sourcemaking.com/design_patterns/state/>