Стратегия (Strategy)

При помощи шаблона Стратегия из клиента выделяется алгоритм, который затем инкапсулируется в типах, реализующих общий интерфейс. Это позволяет клиенту выбирать нужный алгоритм путем инстанциирования объектов необходимых типов. Кроме этого, шаблон допускает изменение набора доступных алгоритмов со временем.

Предположим, что требуется разработать программу, которая показывает календарь. Одно из требований к программе – она должна отображать праздники, отмечаемые различными нациями и религиозными группами. Это требование можно выполнить, помещая логику генерирования для каждого набора праздников в отдельный класс. Основная программа будет выбирать необходимый класс из набора, исходя, например, из действий пользователя (или конфигурационных настроек).

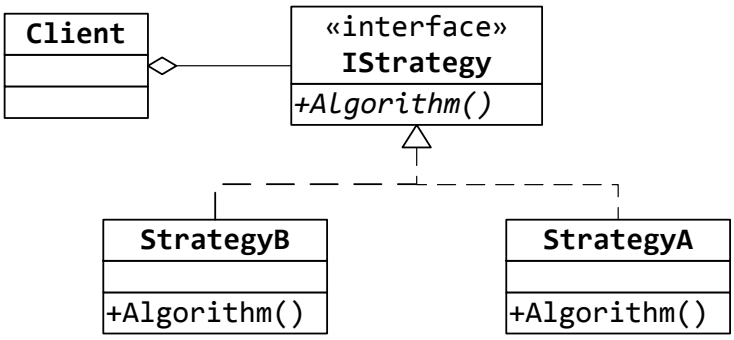

Рис. 15. UML-диаграмма шаблона Стратегия.

Состояние (State)

Шаблон Состояние можно рассматривать как динамическую версию шаблона Стратегия. При использовании шаблона Состояние поведение объекта меняется в зависимости от текущего контекста.

При помощи шаблона Состояние можно эффективно реализовать такую абстракцию как *конечный автомат*. Сделаем небольшое отступление и рассмотрим понятие конечного автомата подробнее. Обычные функции можно рассматривать как некие преобразователь информации. Аргумент функции преобразуется в результат функции, согласно определённому правилу. Функциональные преобразователи обладают важным свойством – их поведение не зависит от предыстории. В реальности, однако, имеется достаточно примеров преобразователей, реакция которых зависит не только от входа в данный момент, но и от того, что было на входе раньше, от *входной истории*. Такие преобразователи называются *автоматами*. Так как количество разных входных историй потенциально бесконечно, на множестве предысторий вводится отношение эквивалентности, и один класс эквивалентности предысторий называется *состоянием автомата*. Состояние автомата меняется только при получении очередного входного сигнала. При этом автомат не только выдаёт информацию на выход как функцию входного сигнала и текущего состояния, но и меняет своё состояние, поскольку входной сигнал изменяет предысторию.

Рассмотрим пример конечного автомата. Опишем поведение родителя, отправившего сына в школу. Сын приносит двойки и пятерки. Отец не хочет хвататься за ремень каждый раз, как только сын получит очередную двойку, и выбирает более тонкую тактику воспитания. Чтобы описать модель поведения отца, используем граф, в котором вершины соответствуют состояниям, а дуга, помеченная x/y, из состояния s в состояние q проводится тогда, когда автомат из состояния s под воздействием входного сигнала x переходит в состояние q с выходной реакцией y. Граф автомата, моделирующего поведение родителя, представлен на рис. 16.

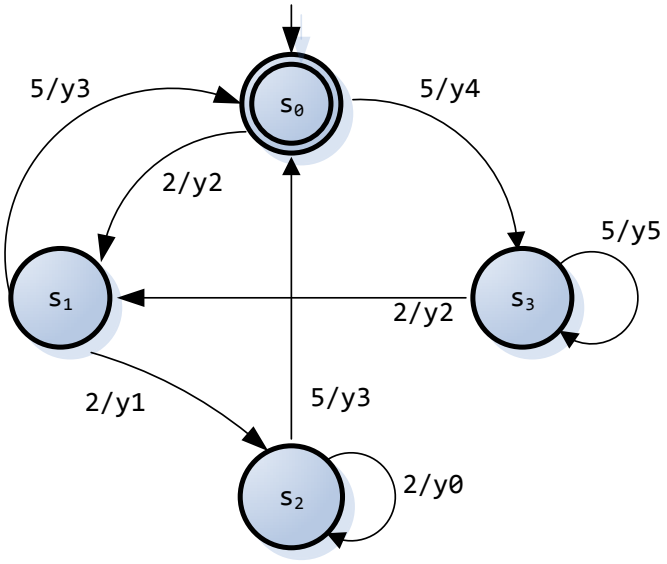

Рис. 16. Автомат, описывающий поведение «умного» отца.

Этот автомат имеет четыре состояния $\{s_0, s_1, s_2, s_3\}$ и два входных сигнала - оценки, полученные сыном в школе: $\{2, 5\}$. Начиная с начального состояния s_0 (оно помечено особо), автомат под воздействием входных сигналов переходит из одного состояния в другое и выдает выходные сигналы - реакции на входы. Выходы автомата будем интерпретировать как действия родителя так:

- y_0 – «брать ремень»;
- y_1 – «ругать»;
- y_2 – «успокаивать»;
- y_3 – «надеяться»;
- y_4 – «радоваться»;
- y_5 – «ликовать».

При наивной программной реализации конечного автомата порождаются наборы конструкций `switch-case`, которые, как правило, вложены в друг друга. Использование шаблона Состояние позволяет упростить код. Все состояния конечного автомата описываются отдельными классами, которые обладают набором виртуальных методов, соответствующих входным сигналам. Получение очередного входного сигнала означает вызов метода того объекта, экземпляр которого находится в поле `state`. При этом сам вызываемый метод может поместить в `state` другой объект-состояние (переключить состояние).

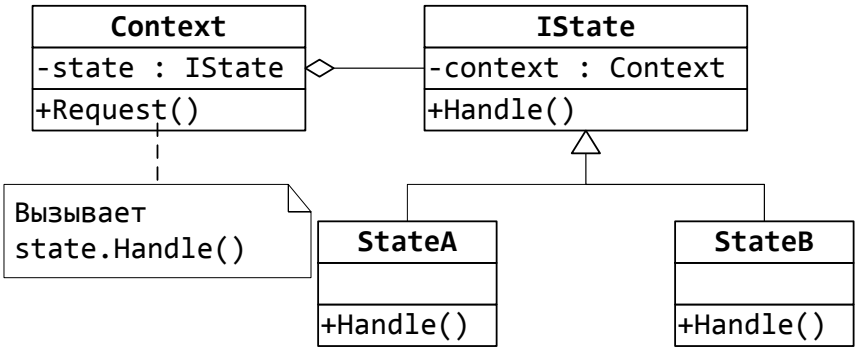

Рис. 17. Дизайн шаблона состояние.

При практической реализации шаблона Состояние отдельные объекты-состояния могут быть оформлены с применением шаблона Одиночка и получать объект-контекст в качестве одно из параметров своих методов.

Конечные автоматы подразделяются на детерминированные и недетерминированные.

Детерминированным конечным автоматом (ДКА) называется такой автомат, в котором для каждой последовательности входных символов существует лишь одно состояние, в которое автомат может перейти из текущего.

Недетерминированный конечный автомат (НКА) является обобщением детерминированного.

Другие способы описания конечного автомата:

Диаграмма состояний (или иногда граф переходов) — графическое представление множества состояний и функции переходов. Представляет собой нагруженный однонаправленный граф, вершины которого — состояния КА, ребра — переходы из одного состояния в другое, а нагрузка — символы, при которых осуществляется данный переход. Если переход из состояния q1 в q2 может быть осуществлен при появлении одного из нескольких символов, то над дугой диаграммы (ветвью графа) должны быть надписаны все они.

Таблица переходов — табличное представление функции δ. Обычно в такой таблице каждой строке соответствует одно состояние, а столбцу — один допустимый входной символ. В ячейке на пересечении строки и столбца записывается действие, которое должен выполнить автомат, если в ситуации, когда он находился в данном состоянии на входе он получил данный символ.

Шаблонный метод (Template method)

Предположим, что программная логика некоего алгоритма представлена в виде набора вызовов методов. При использовании Шаблонного метода создается абстрактный класс, который реализует только часть методов программной логики, оставляя детали реализации остальных методов своим потомкам. Благодаря этому общая структура алгоритма остаётся неизменной, в то время как некоторые конкретные шаги могут изменяться.

Цепочка обязанностей (Chain of responsibility)

Шаблон Цепочка обязанностей работает со списком объектов, называемых *обработчиками*. Каждый из обработчиков имеет естественные ограничения на множество запросов, которые он в состоянии поддержать. Если текущий обработчик не может поддержать запрос, он передает его следующему обработчику в списке. Так продолжается, пока запрос не будет обработан, или пока список не закончится.

Иллюстрацией шаблона Цепочка обязанностей может служить любая организация с иерархической структурой управления. Представим себе банк, в который клиент обращается за кредитом. Если сумма кредита небольшая, запрос удовлетворяет служащий нижнего звена банка. В противном случае запрос отправляется на более высокий уровень иерархии управления.

Дизайн шаблона Цепочка обязанностей достаточно прост. Каждый из обработчиков принадлежит к одному из классов, имеющих общего предка или реализующих общий интерфейс. Кроме метода (или методов) для обработки запроса обработчик имеет поле, указывающее на следующий обработчик в цепочке.

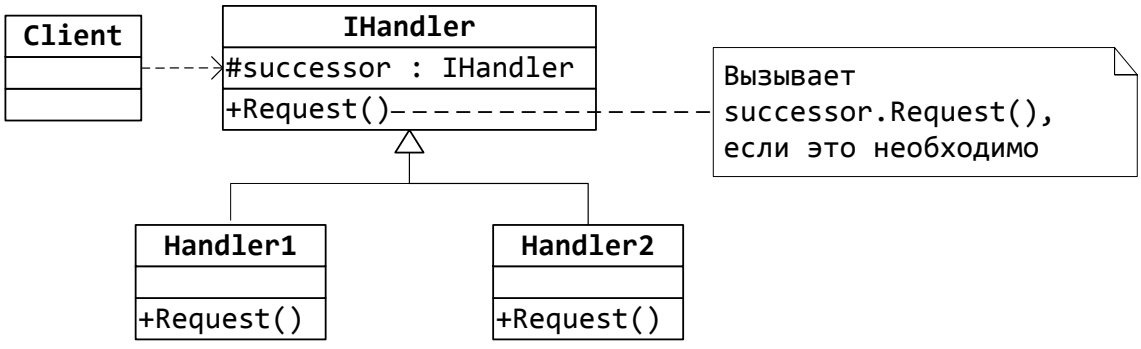

Рис. 18. UML-диаграмма шаблона Цепочка обязанностей.

Итератор (Iterator)

Назначение шаблона Итератор – предоставить последовательный доступ к элементам коллекции без информации о её внутреннем устройстве. Дополнительно шаблон может реализовывать функционал по фильтрации элементов коллекции.

При практической реализации шаблона Итератор на C#, безусловно, используются такие возможности, как перечислители, интерфейсы `IEnumerable` и `IEnumerator`, оператор `yield`.

Посредник (Mediator)

Шаблон Посредник служит для обеспечения коммуникации между объектами. Этот шаблон также инкапсулирует протокол, которому должна удовлетворять процедура коммуникации.

Наблюдатель (Observer)

Шаблон Наблюдатель определяет отношение между объектами таким образом, что когда один из объектов меняет своё состояние, все другие объекты получают об этом уведомление.

Иллюстрацией применения шаблона Наблюдатель служит знакомая по языку C# система событий. Один объект публикует событие, остальные объекты могут подписаться на событие и получать уведомление о его наступлении. Собственно, основное назначение шаблона Наблюдатель – это реализация системы работы с событиями.

Как и в случае с шаблоном Посредник, дизайн шаблона Наблюдатель предполагает наличие двух выделенных классов. Объект класса `Subject` изменяет своё состояния, и именно эти изменения предполагается отслеживать. Объекты (их может быть несколько) класса `Observer` могут подписываться на отслеживание изменений. Класс `Subject` располагает закрытым событием с именем `Notify`. Как только `Subject` изменяет своё состояние, событие активируется. При этом вызывается метод `Update()` подписчиков, которому передаётся состояние объекта `Subject`. Метод `Update()` предварительно регистрируется в `Subject` при помощи операции `Attach()`.

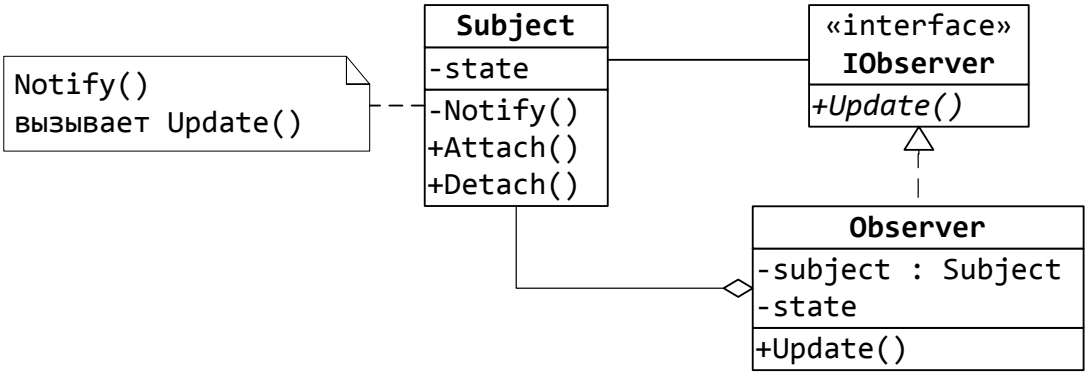

Рис. 22. Дизайн шаблона Наблюдатель.

Хранитель (Memento)

Этот шаблон используется для захвата и сохранения внутреннего состояния объекта с возможностью дальнейшего восстановления состояния. Важной особенностью шаблона Хранитель является то, что он позволяет сохранить внутреннее состояние объекта без нарушения правил инкапсуляции.

Посетитель (Visitor)

Шаблон Посетитель служит для выполнения операций над всеми объектами, объединёнными в некоторую структуру. При этом выполняемые операции не обязательно должны являться частью объектов структуры.

Очень часто в программах встречаются сложные структуры, представляющие собой дерево или граф, состоящий из разнотипных узлов. И, конечно же, при этом имеется необходимость обрабатывать такой граф или дерево. Самое очевидное решение - добавить в базовый класс узла виртуальный метод, перекрываемый в наследниках для выполнения нужного действия и осуществления дальнейшей навигации по структуре.