Iterator

Mediator

Momento

Observer

State

Iterator паттерн

Назначение паттерна Iterator

- Предоставляет способ последовательного доступа ко всем элементам составного объекта, не раскрывая его внутреннего представления.
- Абстракция в стандартных библиотеках C++ и Java, позволяющая разделить классы коллекций и алгоритмов.
- Придает обходу коллекции "объектно-ориентированный статус".

Решаемая проблема

- Вам необходим механизм "абстрактного" обхода различных структур данных так, что могут определяться алгоритмы, способные взаимодействовать со структурами прозрачно.

- Составной объект, такой как список, должен предоставлять способ доступа к его элементам без раскрытия своей внутренней структуры. Более того, иногда нужно перебирать элементы списка различными способами, в зависимости от конкретной задачи. Но вы, вероятно, не хотите раздувать интерфейс списка операциями для различных обходов, даже если они необходимы. Кроме того, иногда нужно иметь несколько активных обходов одного списка одновременно. Было бы хорошо иметь единый интерфейс для обхода разных типов составных объектов (т.е. полиморфная итерация).

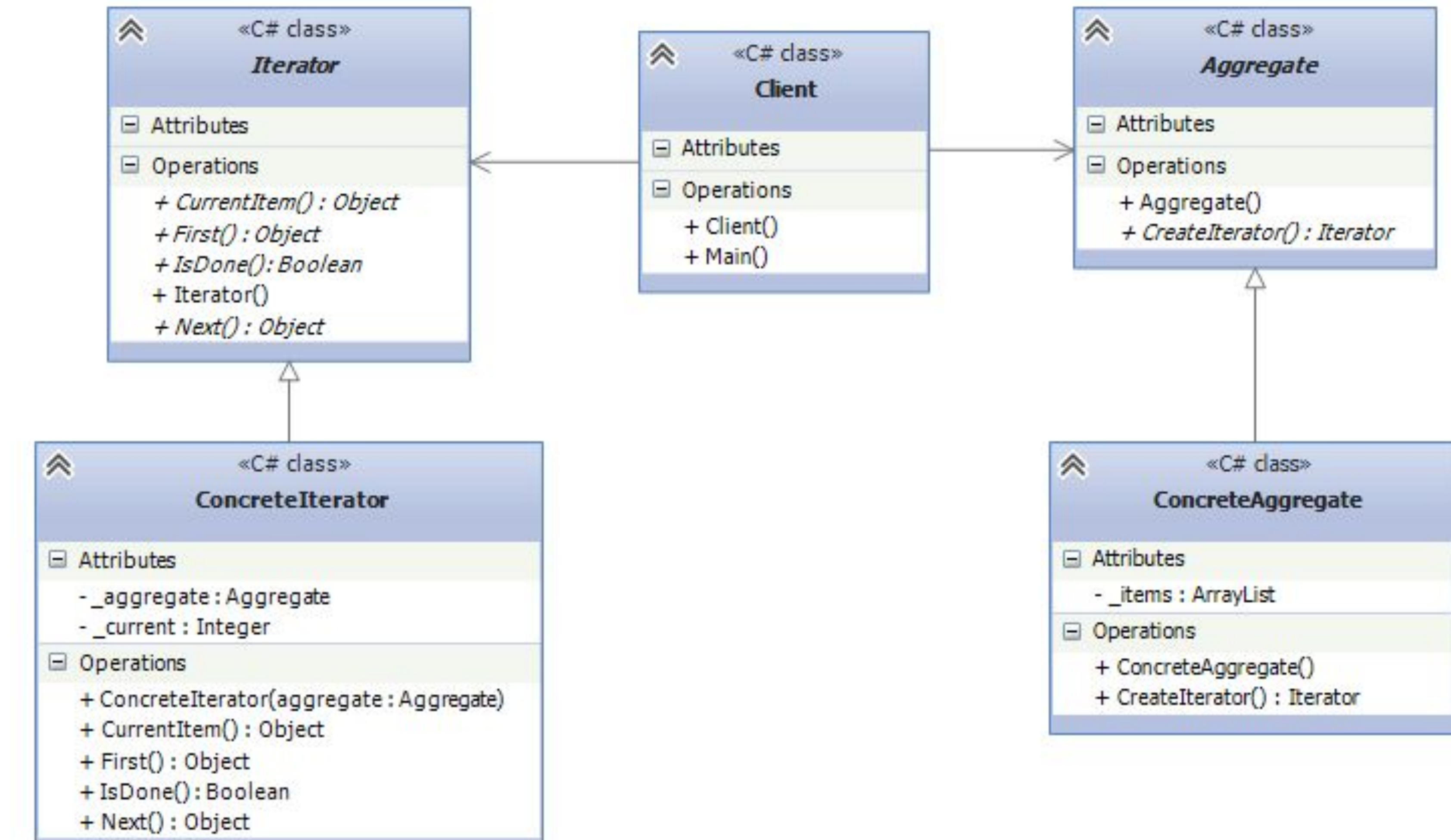
- Ключевая идея состоит в том, чтобы ответственность за доступ и обход переместить из составного объекта на объект Iterator, который будет определять стандартный протокол обхода.
- Абстракция Iterator имеет основополагающее значение для технологии, называемой "обобщенное программирование". Эта технология четко разделяет такие понятия как "алгоритм" и "структура данных".
- Мотивирующие факторы: способствование компонентной разработке, повышение производительности и снижение расходов на управление.

- Если вы хотите одновременно поддерживать четыре вида структур данных (массив, бинарное дерево, связанный список и хэш-таблица) и три алгоритма (сортировка, поиск и слияние), то традиционный подход потребует 12 вариантов конфигураций (четыре раза по три), в то время как обобщенное программирование требует лишь 7 (четыре плюс три).

Когда использовать итераторы?

- Когда необходимо осуществить обход объекта без раскрытия его внутренней структуры
- Когда имеется набор составных объектов, и надо обеспечить единый интерфейс для их перебора
- Когда необходимо предоставить несколько альтернативных вариантов перебора одного и того же объекта

Структура паттерна Iterator



Участники

- Iterator: определяет интерфейс для обхода составных объектов
- Aggregate: определяет интерфейс для создания объекта-итератора
- ConcreteIterator: конкретная реализация итератора для обхода объекта Aggregate.
- ConcreteAggregate: конкретная реализация Aggregate. Хранит элементы, которые надо будет перебирать
- Client: использует объект Aggregate и итератор для его обхода

Теперь рассмотрим конкретный пример. Допустим, у нас есть классы книги и библиотеки:

- class Book
- {
-     public string Name { get; set; }
- }
- class Library
- {
-     private string[] books;
- }

у нас есть класс читателя, который хочет получить информацию о книгах, которые находятся в библиотеке. И для этого надо осуществить перебор объектов с помощью итератора:

```
class Program
{
    static void Main(string[] args)
    {
        Library library = new Library();
        Reader reader = new Reader();
        reader.SeeBooks(library);

        Console.Read();
    }
}

class Reader
```

- Интерфейс IBookIterator представляет итератор наподобие интерфейса IEnumerator. Роль интерфейса составного агрегата представляет тип IBookNumerable. Клиентом здесь является класс Reader, который использует итератор для обхода объекта библиотеки.

1. Назначение экземпляра класса **ConcreteAggregate** - хранить элементы, которые надо будет перебирать.
2. Назначение экземпляра класса **ConcreteIterator** - реализация итератора для обхода объекта класса **Aggregate**
3. Иначе говоря, итератор у нас реализуется объектом класса **ConcreteIterator**, а не объектом класса**ConcreteAggregate** - который используется только как хранилище объектов для перебора.
4. Объект же класса **ConcreteAggregate** только передается в качестве агрегата в объект класса **ConcreteIterator** как хранилище объектов - и не более того.

Mediator (посредник) паттерн

Назначение паттерна Mediator

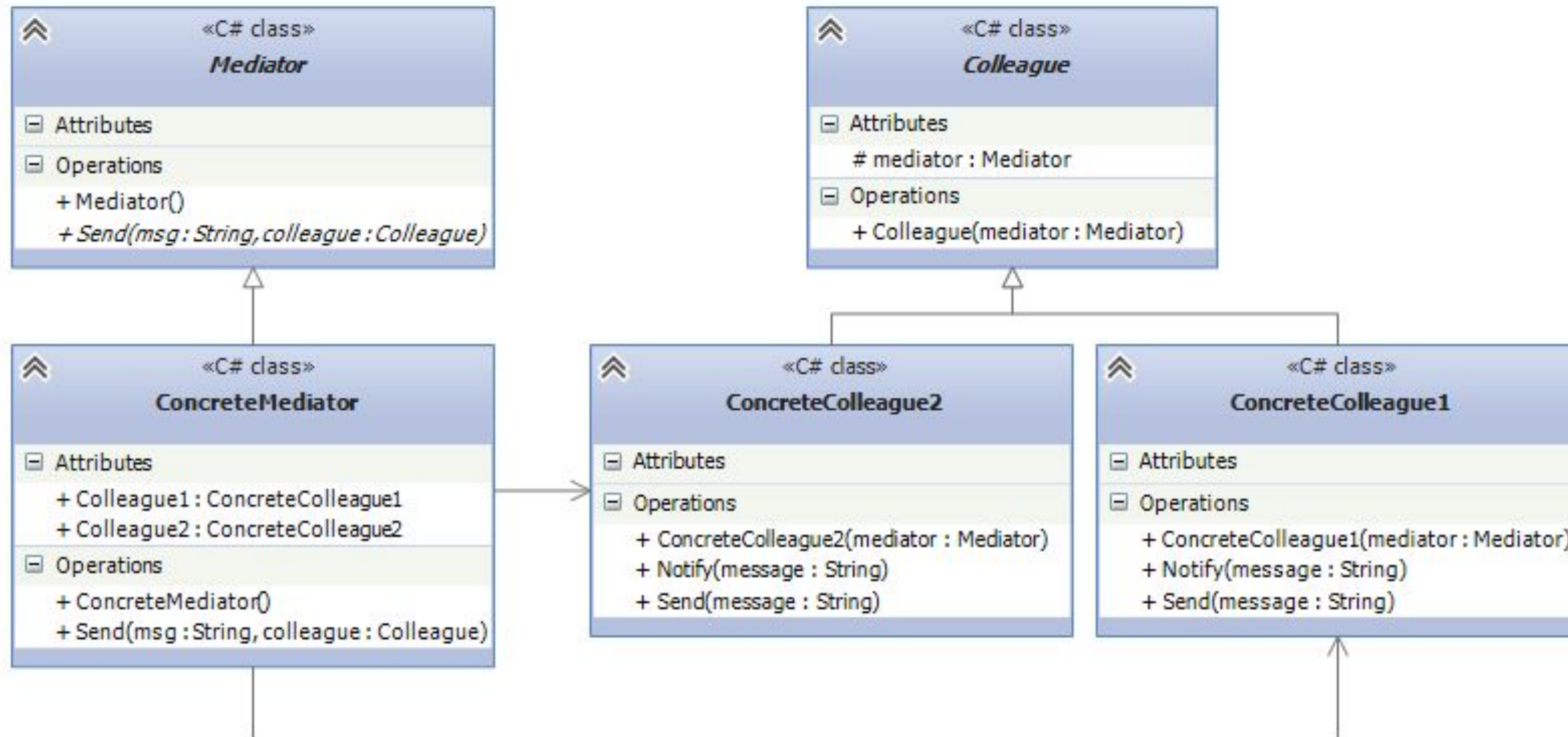
- Mediator делает систему слабо связанной, избавляя объекты от необходимости ссылаться друг на друга, что позволяет изменять взаимодействие между ними независимо.
- Паттерн Mediator вводит посредника для развязывания множества взаимодействующих объектов.
- Заменяет взаимодействие "все со всеми" взаимодействием "один со всеми".

Решаемая проблема

- Мы хотим спроектировать систему с повторно используемыми компонентами, однако существующие связи между этими компонентами можно охарактеризовать феноменом "спагетти-кода".
- Спагетти-код - плохо спроектированная, слабо структурированная, запутанная и трудная для понимания программа. Спагетти-код назван так, потому что ход выполнения программы похож на миску спагетти, то есть извилистый и запутанный.

- В Unix права доступа к системным ресурсам определяются тремя уровнями: владелец, группа и прочие. Группа представляет собой совокупность пользователей, обладающих некоторой функциональной принадлежностью. Каждый пользователь в системе может быть членом одной или нескольких групп, и каждая группа может иметь 0 или более пользователей, назначенных этой группе. Следующий рисунок показывает трех пользователей, являющихся членами всех трех групп.
- Если нам нужно было бы построить программную модель такой системы, то мы могли бы связать каждый объект User с каждым объектом Group, а каждый объект Group - с каждым объектом User. Однако из-за наличия множества взаимосвязей модифицировать поведение такой системы очень непросто, пришлось бы изменять все существующие классы.
- Альтернативный подход - введение "дополнительного уровня косвенности" или построение абстракции из отображения (соответствия) пользователей в группы и групп в пользователей. Такой подход обладает следующими преимуществами: пользователи и группы отделены друг от друга, отображениями легко управлять одновременно и абстракция отображения может быть расширена в будущем путем определения производных классов.
- Разбиение системы на множество объектов в общем случае повышает степень повторного использования, однако множество взаимосвязей между этими объектами, как правило, приводит к обратному эффекту. Чтобы этого не допустить, инкапсулируйте взаимодействия между объектами в объект-посредник. Действуя как центр связи, этот объект-посредник контролирует и координирует взаимодействие группы объектов. При этом объект-посредник делает взаимодействующие объекты слабо связанными, так как им больше не нужно хранить ссылки друг на друга – все взаимодействие идет через этого посредника. Расширить или изменить это взаимодействие можно через его подклассы.
- Паттерн Mediator заменяет взаимодействие "все со всеми" взаимодействием "один со всеми".
- Пример рационального использования паттерна Mediator – моделирование отношений между пользователями и группами операционной системы. Группа может иметь 0 или более пользователей, а пользователь может быть членом 0 или более групп. Паттерн Mediator предусматривает гибкий способ управления пользователями и группами.

Структура паттерна Mediator



Участники

- **Mediator**: представляет интерфейс для взаимодействия с объектами Colleague
- **Colleague**: представляет интерфейс для взаимодействия с объектом Mediator
- **ConcreteColleague1** и **ConcreteColleague2**: конкретные классы коллег, которые обмениваются друг с другом через объект Mediator
- **ConcreteMediator**: конкретный посредник, реализующий интерфейс типа Mediator

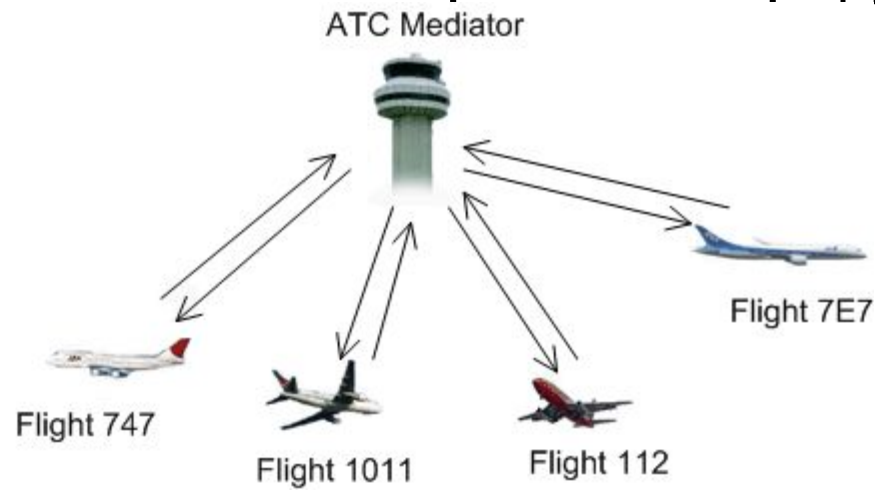
Когда используется паттерн Посредник?

- Когда имеется множество взаимосвязаных объектов, связи между которыми сложны и запутаны.
- Когда необходимо повторно использовать объект, однако повторное использование затруднено в силу сильных связей с другими объектами.

- **Пример паттерна Mediator**

Паттерн Mediator определяет объект, управляющий набором взаимодействующих объектов. Слабая связанность достигается благодаря тому, что вместо непосредственного взаимодействия друг с другом коллеги общаются через объект-посредник.

Башня управления полетами в аэропорту хорошо демонстрирует этот паттерн. Пилоты взлетающих или идущих на посадку самолетов в районе аэропорта общаются с башней вместо непосредственного общения друг с другом. Башня определяет, кто и в каком порядке будет садиться или взлетать. Важно отметить, что башня контролирует самолеты только в районе аэродрома, а не на протяжении всего полета.



Пример

Система создания программных продуктов включает ряд акторов: заказчики, программисты, тестировщики и так далее.

Но нередко все эти акторы взаимодействуют между собой не непосредственно, а опосредованно через менеджера проектов. То есть менеджер проектов выполняет роль посредника. В этом случае процесс взаимодействия между объектами мы могли бы описать так:

- class Program
- {
-     static void Main(string[] args)
-     {
-         ManagerMediator mediator = new ManagerMediator();
-         Colleague customer = new CustomerColleague(mediator);
-         Colleague programmer = new ProgrammerColleague(mediator);
-         Colleague tester = new TesterColleague(mediator);
-         mediator.Customer = customer;
-         mediator.Programmer = programmer;
-         mediator.Tester = tester;
-         customer.Send("Есть заказ, надо сделать программу");
-         programmer.Send("Программа готова, надо протестировать");
-         tester.Send("Программа протестирована и готова к продаже");
-
-         Console.Read();
-     }

Класс менеджера - ManagerMediator в методе Send() проверяет, от кого пришло сообщение, и в зависимости от отправителя перенаправляет его другому объекту с помощью методов Notify(), определенных в классе Colleague.

- Консольный вывод программы:

- Сообщение программисту: Есть заказ, надо сделать программу
- Сообщение тестеру: Программа готова, надо протестировать
- Сообщение заказчику: Программа протестирована и готова к продаже

В итоге применение паттерна Посредник дает нам следующие преимущества:

- Устраняется сильная связанность между объектами Colleague
- Упрощается взаимодействие между объектами: вместо связей по типу "все-ко-всем" применяется связь "один-ко-всем"
- Взаимодействие между объектами абстрагируется и выносится в отдельный интерфейс
- Централизуется управления отношениями между объектами

Использование паттерна Mediator

- Определите совокупность взаимодействующих объектов, связанность между которыми нужно уменьшить.
- Инкапсулируйте все взаимодействия в абстракцию нового класса.
- Создайте экземпляр этого нового класса. Объекты-коллеги для взаимодействия друг с другом используют только этот объект.
- Найдите правильный баланс между принципом слабой связанности и принципом распределения ответственности.
- Будьте внимательны и не создавайте объект-"контроллер" вместо объекта-посредника.

Особенности паттерна Mediator

- Паттерны Chain of Responsibility, Command, Mediator и Observer показывают, как можно разделить отправителей и получателей запросов с учетом их особенностей. Chain of Responsibility передает запрос отправителя по цепочке потенциальных получателей. Command номинально определяет связь - "оправитель-получатель" с помощью подкласса. В Mediator отправитель и получатель ссылаются друг на друга косвенно, через объект-посредник. В паттерне Observer связь между отправителем и получателем слабее, при этом число получателей может конфигурироваться во время выполнения.
- Mediator и Observer являются конкурирующими паттернами. Если Observer распределяет взаимодействие с помощью объектов "наблюдатель" и "субъект", то Mediator использует объект-посредник для инкапсуляции взаимодействия между другими объектами. Мы обнаружили, что легче сделать повторно используемыми Наблюдателей и Субъектов, чем Посредников.
- С другой стороны, Mediator может использовать Observer для динамической регистрации коллег и их взаимодействия с посредником.
- Mediator похож Facade в том, что он абстрагирует функциональность существующих классов. Mediator абстрагирует/централизует взаимодействие между объектами-коллегами, добавляет новую функциональность и известен всем объектам-коллегам (то есть определяет двунаправленный протокол взаимодействия). Facade, наоборот, определяет более простой интерфейс к подсистеме, не добавляя новой функциональности, и неизвестен классам подсистемы (то есть имеет однонаправленный протокол взаимодействия, то есть запросы отправляются в подсистему, но не наоборот).

Momento паттерн

Назначение паттерна Memento

- Не нарушая инкапсуляции, паттерн Memento получает и сохраняет за пределами объекта его внутреннее состояние так, чтобы позже можно было восстановить объект в таком же состоянии.
- Является средством для инкапсуляции "контрольных точек" программы.
- Паттерн Memento придает операциям "Отмена" (undo) или "Откат" (rollback) статус "полноценного объекта".

Решаемая проблема

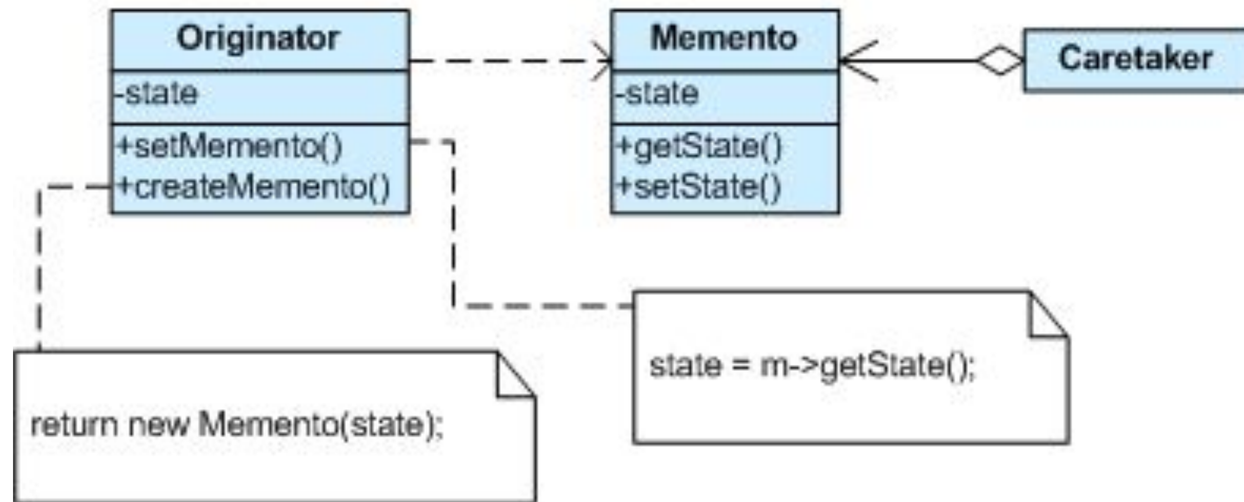
- Вам нужно восстановить объект обратно в прежнее состояние (те есть выполнить операции "Отмена" или "Откат").

- Клиент запрашивает Memento (хранителя) у исходного объекта, когда ему необходимо сохранить состояние исходного объекта (установить контрольную точку). Исходный объект инициализирует Memento своим текущим состоянием. Клиент является "посыльным" за Memento, но только исходный объект может сохранять и извлекать информацию из Memento (Memento является "непрозрачным" для клиентов и других объектов). Если клиенту в дальнейшем нужно "откатить" состояние исходного объекта, он передает Memento обратно в исходный объект для его восстановления.
- Реализовать возможность выполнения неограниченного числа операций "Отмена" (undo) и "Повтор" (redo) можно с помощью стека объектов Command и стека объектов Memento.

Паттерн проектирования Memento определяет трех различных участников:

- **Originator (хозяин)** - объект, умеющий создавать хранителя, а также знающий, как восстановить свое внутреннее состояние из хранителя.
- **Caretaker (смотритель)** - объект, который знает, почему и когда хозяин должен сохранять и восстанавливать себя.
- **Memento (хранитель)** - "ящик на замке", который пишется и читается хозяином и за которым присматривает смотритель.

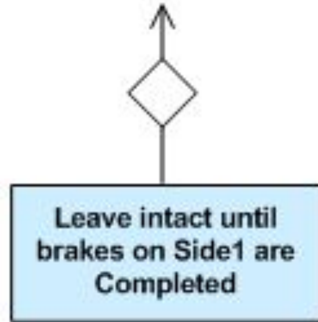
Структура паттерна Memento



Пример паттерна Memento

- Паттерн Memento фиксирует и сохраняет за пределами объекта его внутреннее состояние так, чтобы позже этот объект можно было бы восстановить в таком же состоянии.
- Этот паттерн часто используется механиками-любителями для ремонта барабанных тормозов на своих автомобилях.
- Барабаны удаляются с обеих сторон, чтобы сделать видимыми правые и левые тормоза. При этом разбирается только одна сторона, другая же служит напоминанием (Memento) о том, как части тормозной системы тормозной системы собраны вместе. Только после того, как завершена работа с одной стороны, разбирается другая сторона. При этом в качестве Memento выступает уже первая сторона.

return(brakeReference)



Использование паттерна Memento

- Определите роли "смотрителя" и "хозяина".
- Создайте класс Memento и объявите хозяина другом.
- Смотритель знает, когда создавать "контрольную точку" хозяина.
- Хозяин создает хранителя Memento и копирует свое состояние в этот Memento.
- Смотритель сохраняет хранителя Memento (но смотритель не может заглянуть в Memento).
- Смотритель знает, когда нужно "откатить" хозяина.
- Хозяин восстанавливает себя, используя сохраненное в Memento состояние.

Особенности паттерна Memento

- Паттерны Command и Memento определяют объекты "волшебная палочка", которые передаются от одного владельца к другому и используются позднее. В Command такой "волшебной палочкой" является запрос; в Memento - внутреннее состояние объекта в некоторый момент времени. Полиморфизм важен для Command, но не важен для Memento потому, что интерфейс Memento настолько "узкий", что его можно передавать как значение.
- Command может использовать Memento для сохранения состояния, необходимого для выполнения отмены действий.
- Memento часто используется совместно с Iterator. Iterator может использовать Memento для сохранения состояния итерации.

Когда использовать Memento?

- Когда нужно охранить его состояние объекта для возможного последующего восстановления
- Когда сохранение состояния должно проходить без нарушения принципа инкапсуляции

Участники

- **Memento**: хранитель, который сохраняет состояние объекта Originator и предоставляет полный доступ только этому объекту Originator
- **Originator**: создает объект хранителя для сохранения своего состояния
- **Caretaker**: выполняет только функцию хранения объекта Memento, в то же время у него нет полного доступа к хранителю и никаких других операций над хранителем, кроме собственно сохранения, он не производит

пример:

- нам надо сохранять состояние игрового персонажа в игре:

- class Program
- {
-     static void Main(string[] args)
-     {
-         Hero hero = new Hero();
-         hero.Shoot(); // делаем выстрел, осталось 9 патронов
-         GameHistory game = new GameHistory();
-
-         game.History.Push(hero.SaveState()); // сохраняем игру
-
-         hero.Shoot(); //делаем выстрел, осталось 8 патронов
-
-         hero.RestoreState(game.History.Pop());
-
-         hero.Shoot(); //делаем выстрел, осталось 8 патронов
-
-         Console.Read();
-     }

Консольный вывод программы:

- Производим выстрел. Осталось 9 патронов
- Сохранение игры. Параметры: 9 патронов, 5 жизней
- Производим выстрел. Осталось 8 патронов
- Восстановление игры. Параметры: 9 патронов, 5 жизней
- Производим выстрел. Осталось 8 патронов

- Здесь в роли Originator выступает класс Hero, состояние которого описывается количество патронов и жизней. Для хранения состояния игрового персонажа предназначен класс HeroMemento. С помощью метода SaveState() объект Hero может сохранить свое состояние в HeroMemento, а с помощью метода RestoreState() - восстановить.

- Для хранения состояний предназначен класс GameHistory, причем все состояния хранятся в стеке, что позволяет с легкостью извлекать последнее сохраненное состояние.

Использование паттерна Memento дает нам следующие преимущества:

- Уменьшение связанности системы
- Сохранение инкапсуляции информации
- Определение простого интерфейса для сохранения и восстановления состояния
- В то же время мы можем столкнуться с недостатками, в частности, если требуется сохранение большого объема информации, то возрастут издержки на хранение всего объема состояния.

Observer(наблюдатель) паттерн

Назначение паттерна Observer

- Паттерн Observer определяет зависимость "один-ко-многим" между объектами так, что при изменении состояния одного объекта все зависящие от него объекты уведомляются и обновляются автоматически.
- Паттерн Observer инкапсулирует главный (независимый) компонент в абстракцию Subject и изменяемые (зависимые) компоненты в иерархию Observer.
- Паттерн Observer находит широкое применение в системах пользовательского интерфейса, в которых данные и их представления ("виды") отделены друг от друга. При изменении данных должны быть изменены все представления этих данных (например, в виде таблицы, графика и диаграммы).

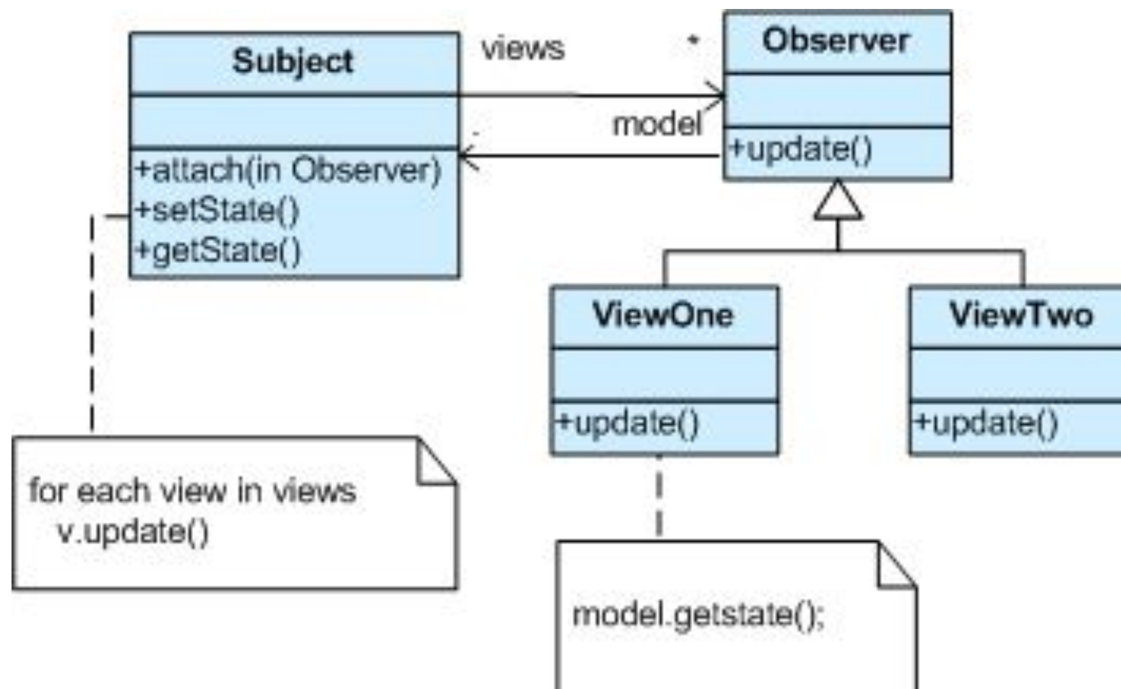
Решаемая проблема

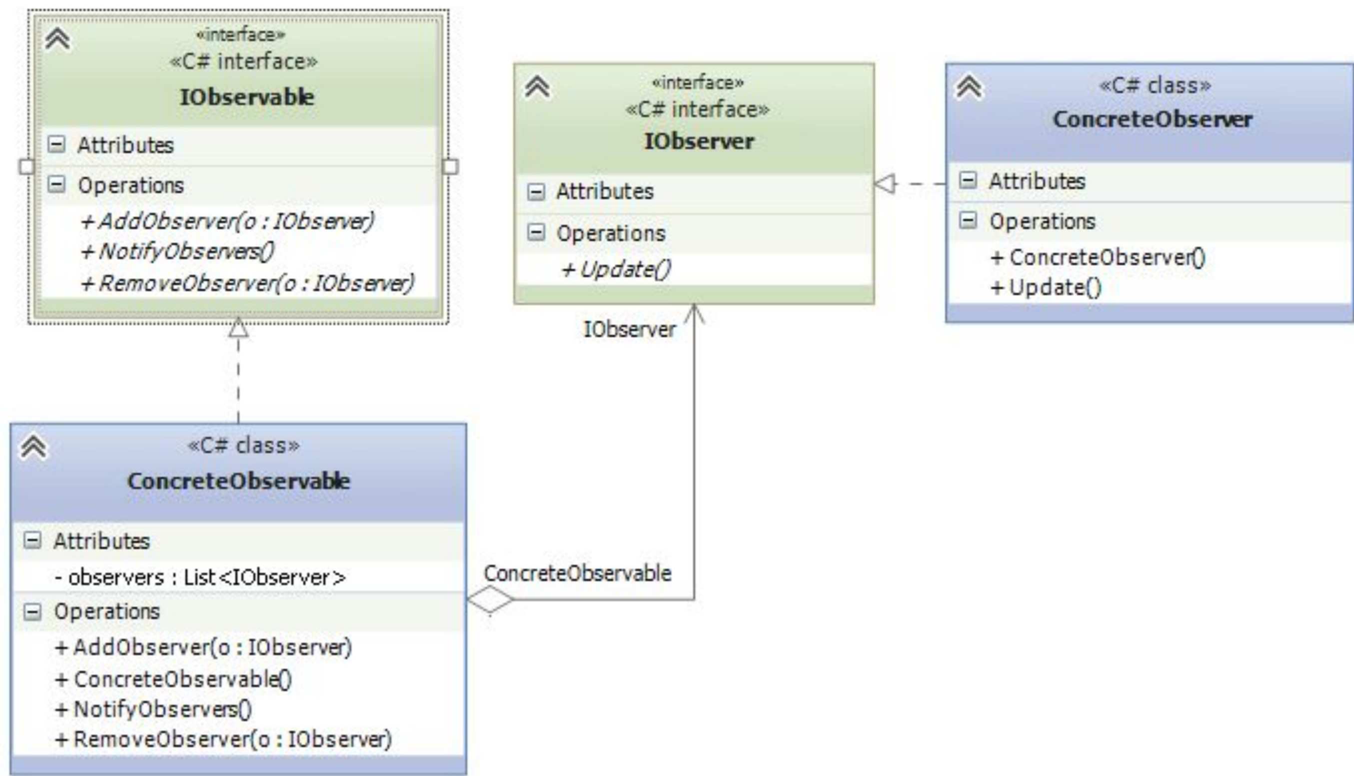
- Имеется система, состоящая из множества взаимодействующих классов. При этом взаимодействующие объекты должны находиться в согласованных состояниях. Вы хотите избежать монолитности (неразделимости)такой системы, сделав классы слабо связанными (или повторно используемыми).

- Паттерн Observer определяет объект Subject, хранящий данные (модель), а всю функциональность "представлений" делегирует слабосвязанным отдельным объектам Observer. При создании наблюдатели Observer регистрируются у объекта Subject. Когда объект Subject изменяется, он извещает об этом всех зарегистрированных наблюдателей. После этого каждый обозреватель запрашивает у объекта Subject ту часть состояния, которая необходима для отображения данных.
- Такая схема позволяет динамически настраивать количество и "типы" представлений объектов.
- Описанный выше протокол взаимодействия соответствует модели вытягивания (pull), когда субъект информирует наблюдателей о своем изменении, и каждый наблюдатель ответственен за "вытягивание" у Subject нужных ему данных. Существует также модель проталкивания, когда субъект Subject посылает ("проталкивает") наблюдателям детальную информацию о своем изменении.

Структура паттерна Observer

- Subject представляет главную (независимую) абстракцию. Observer представляет изменяемую (зависимую) абстракцию. Субъект извещает наблюдателей о своем изменении, на что каждый наблюдатель может запросить состояние субъекта.




«interface»
«C# interface»
IObservable
Attributes
Operations
+AddObserver(o : IObserver)
+NotifyObservers()
+RemoveObserver(o : IObserver)
«interface»
«C# interface»
IObserver
Attributes
Operations
+Update()
«C# class»
ConcreteObserver
Attributes
Operations
+ConcreteObserver()
+Update()
IObserver
«C# class»
ConcreteObservable
Attributes
- observers : List<IObserver>
Operations
+AddObserver(o : IObserver)
+ConcreteObservable()
+NotifyObservers()
+RemoveObserver(o : IObserver)
ConcreteObservable

Участники

- IObservable: представляет наблюдаемый объект. Определяет три метода: AddObserver() (для добавления наблюдателя), RemoveObserver() (удаление набюдателя) и NotifyObservers() (уведомление наблюдателей)

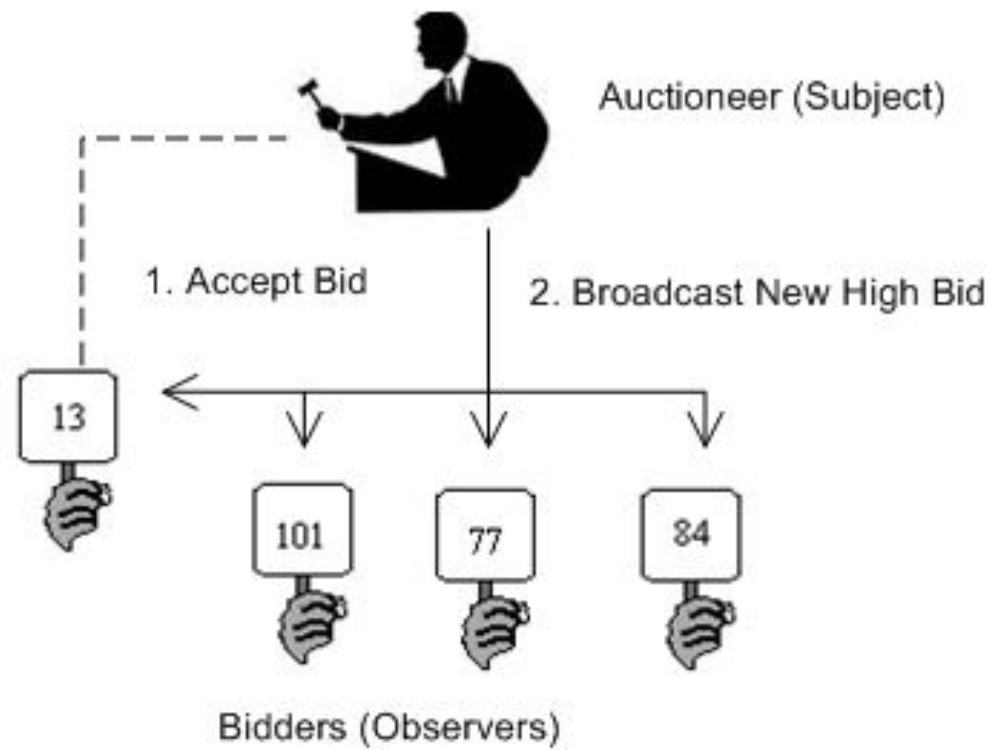
- ConcreteObservable: конкретная реализация интерфейса IObservable. Определяет коллекцию объектов наблюдателей.

- IObserver: представляет наблюдателя, который подписывается на все уведомления наблюдаемого объекта. Определяет метод Update(), который вызывается наблюдаемым объектом для уведомления наблюдателя.

- ConcreteObserver: конкретная реализация интерфейса IObserver.

Пример паттерна Observer

- Паттерн Observer определяет зависимость "один-ко-многим" между объектами так, что при изменении состояния одного объекта все зависящие от него объекты уведомляются и обновляются автоматически.
- Некоторые аукционы демонстрируют этот паттерн. Каждый участник имеет карточку с цифрами, которую он использует для обозначения предлагаемой цены (ставки). Ведущий аукциона (Subject) начинает торги и наблюдает, когда кто-нибудь поднимает карточку, предлагая новую более высокую цену. Ведущий принимает заявку, о чем тут же извещает всех участников аукциона (Observers).


Auctioneer (Subject)
1. Accept Bid
2. Broadcast New High Bid
13
101
77
84
Bidders (Observers)

Использование паттерна Observer

- Проведите различия между основной (или независимой) и дополнительной (или зависимой) функциональностями.
- Смоделируйте "независимую" функциональность с помощью абстракции "субъект".
- Смоделируйте "зависимую" функциональность с помощью иерархии "наблюдатель".
- Класс Subject связан только с базовым классом Observer.
- Клиент настраивает количество и типы наблюдателей.
- Наблюдатели регистрируются у субъекта.
- Субъект извещает всех зарегистрированных наблюдателей.
- Субъект может "протолкнуть" информацию в наблюдателей, или наблюдатели могут "вытянуть" необходимую им информацию от объекта Subject.

Особенности паттерна Observer

- Паттерны Chain of Responsibility, Command, Mediator и Observer показывают, как можно разделить отправителей и получателей запросов с учетом своих особенностей. Chain of Responsibility передает запрос отправителя по цепочке потенциальных получателей. Command определяет связь - "оправитель-получатель" с помощью подкласса. В Mediator отправитель и получатель ссылаются друг на друга косвенно, через объект-посредник. В паттерне Observer связь между отправителем и получателем получается слабой, при этом число получателей может конфигурироваться во время выполнения.
- Mediator и Observer являются конкурирующими паттернами. Если Observer распределяет взаимодействие с помощью объектов "наблюдатель" и "субъект", то Mediator использует объект-посредник для инкапсуляции взаимодействия между другими объектами. Мы обнаружили, что легче сделать повторно используемыми Наблюдателей и Субъектов, чем Посредников.
- Mediator может использовать Observer для динамической регистрации коллег и их взаимодействия с посредником.

- Допустим, у нас есть биржа, где проходят торги, и есть брокеры и банки, которые следят за поступающей информацией и в зависимости от поступившей информации производят определенные действия:

```
class Program
{
    static void Main(string[] args)
    {
        Stock stock = new Stock();
        Bank bank = new Bank("ЮнитБанк", stock);
        Broker broker = new Broker("Иван Иваныч", stock);
        // имитация торгов
        stock.Market();
        // брокер прекращает наблюдать за торгами
        broker.StopTrade();
        // имитация торгов
        stock.Market();

        Console.Read();
    }
}
```

- Здесь наблюдаемый объект представлен интерфейсом IObservable, а наблюдатель - интерфейсом IObserver. Реализацией интерфейса IObservable является класс Stock, который символизирует валютную биржу. В этом классе определен метод Market(), который имитирует торги и инкапсулирует всю информацию о валютных курсах в объекте StockInfo. После проведения торгов производится уведомление всех наблюдателей.

- Реализациями интерфейса IObserver являются классы Broker, представляющий брокера, и Bank, представляющий банк. При этом метод Update() интерфейса IObserver принимает в качестве параметра некоторый объект. Реализация этого метода подразумевает получение через данный параметр объекта StockInfo с текущей информацией о торгах и произведение некоторых действий: покупка или продажа долларов и евро. Дело в том, что часто необходимо информировать наблюдателя об изменении состояния наблюдаемого объекта. В данном случае состояние заключено в объекте StockInfo. И одним из вариантом информирования наблюдателя о состоянии является push-модель, при которой наблюдаемый объект передает (иначе говоря толкает - push) данные о своем состоянии, то есть передаем в виде параметра метода Update().

- Альтернативой push-модели является pull-модель, когда наблюдатель вытягивает (pull) из наблюдаемого объекта данные о состоянии с помощью дополнительных методов.
- Также в классе брокера определен дополнительный метод StopTrade(), с помощью которого брокер может отписаться от уведомлений биржи и перестать быть наблюдателем.

State (состояние) паттерн

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

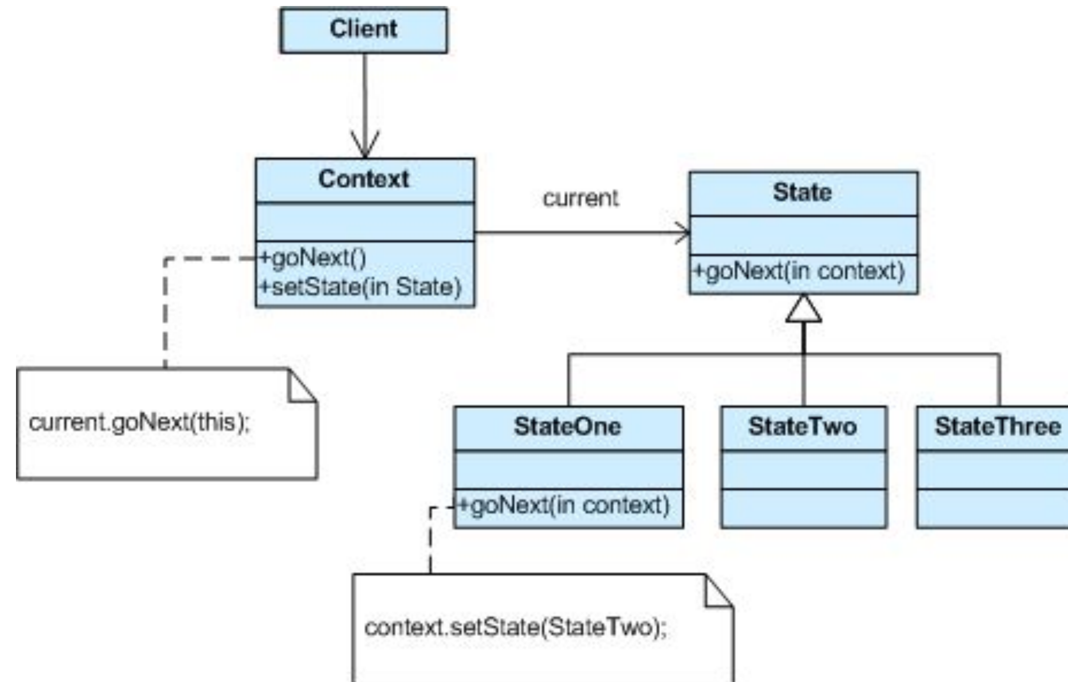
- Поведение объекта зависит от его состояния и должно изменяться во время выполнения программы. Такую схему можно реализовать, применив множество условных операторов: на основе анализа текущего состояния объекта предпринимаются определенные действия.
- Однако при большом числе состояний условные операторы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

Паттерн State решает указанную проблему следующим образом:

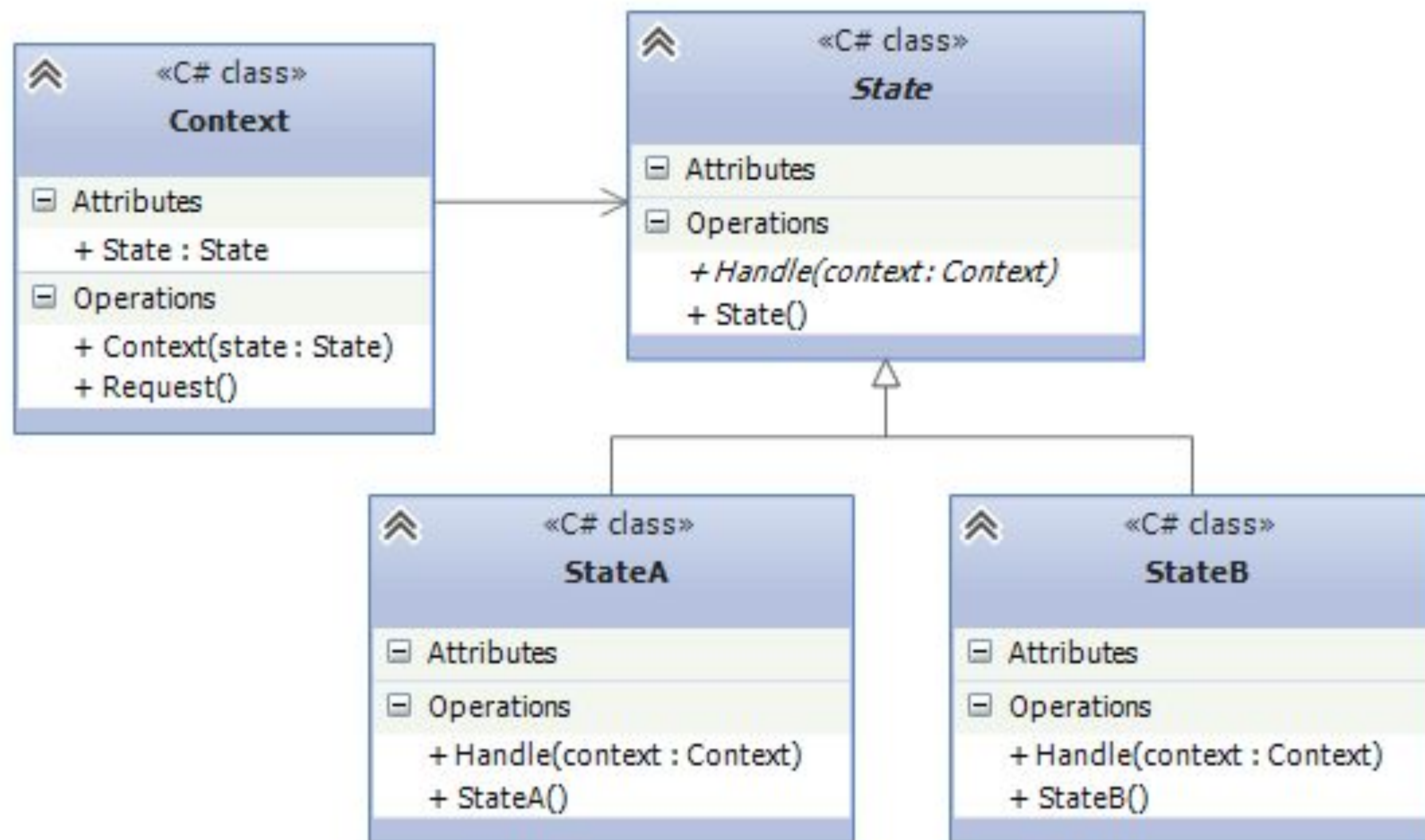
- Вводит класс Context, в котором определяется интерфейс для внешнего мира.
- Вводит абстрактный класс State.
- Представляет различные "состояния" конечного автомата в виде подклассов State.
- В классе Context имеется указатель на текущее состояние, который изменяется при изменении состояния конечного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

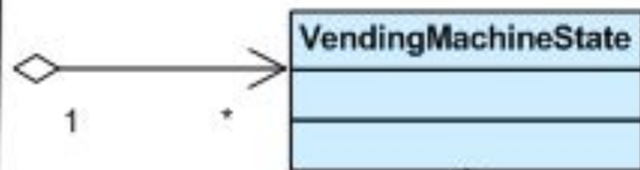

«C# class»
Context
Attributes
+ State : State
Operations
+ Context(state : State)
+ Request()
«C# class»
State
Attributes
Operations
+ Handle(context : Context)
+ State()
«C# class»
StateA
Attributes
Operations
+ Handle(context : Context)
+ StateA()
«C# class»
StateB
Attributes
Operations
+ Handle(context : Context)
+ StateB()

Участники паттерна

- **State**: определяет интерфейс состояния
- Классы **StateA** и **StateB** - конкретные реализации состояний
- **Context**: представляет объект, поведение которого должно динамически изменяться в соответствии с состоянием. Выполнение же конкретных действий делегируется объекту состояния

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.



1

*

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.
- Flyweight показывает, как и когда можно разделять объекты State.
- Паттерн Interpreter может использовать State для определения контекстов при синтаксическом разборе.
- Паттерны State и 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. Различия заключаются в их назначении.

- Например, вода может находиться в ряде состояний: твердое, жидкое, парообразное. Допустим, нам надо определить класс Вода, у которого бы имелись методы для нагревания и заморозки воды. Без использования паттерна Состояние мы могли бы написать следующую программу:

```
class Program
{
    static void Main(string[] args)
    {
        Water water = new Water(WaterState.LIQUID);
        water.Heat();
        water.Frost();
        water.Frost();

        Console.Read();
    }
}
enum WaterState
{
    SOLID,
    LIQUID,
    GAS
}
class Water
{
    public WaterState State { get; set; }

    public Water(WaterState ws)
    {
        State = ws;
```

- Вода имеет три состояния, и в каждом методе нам надо смотреть на текущее состояние, чтобы произвести действия. В итоге с трех состояний уже получается нагромождение условных конструкций. Да и самим методов в классе Вода может также быть множество, где также надо будет действовать в зависимости от состояния. Поэтому, чтобы сделать программу более гибкой, в данном случае мы можем применить паттерн Состояние:

```
• class Program
• {
•     static void Main(string[] args)
•     {
•         Water water = new Water(new LiquidWaterState());
•         water.Heat();
•         water.Frost();
•         water.Frost();
•
•         Console.Read();
•     }
• }
•  class Water
•     {
•         public IWaterState State { get; set; }
•
```

Таким образом, реализация паттерна Состояние позволяет вынести поведение, зависящее от текущего состояния объекта, в отдельные классы, и избежать перегруженности методов объекта условными конструкциями, как if..else или switch.

Кроме того, при необходимости мы можем ввести в систему новые классы состояний, а имеющиеся классы состояний использовать в других объектах.

Лабораторная работа №8

(дедлайн 13.05)

1. UML диаграммы:
   - Iterator
   - Mediator
   - Momento
   - Observer
   - State
2. Релиз примера