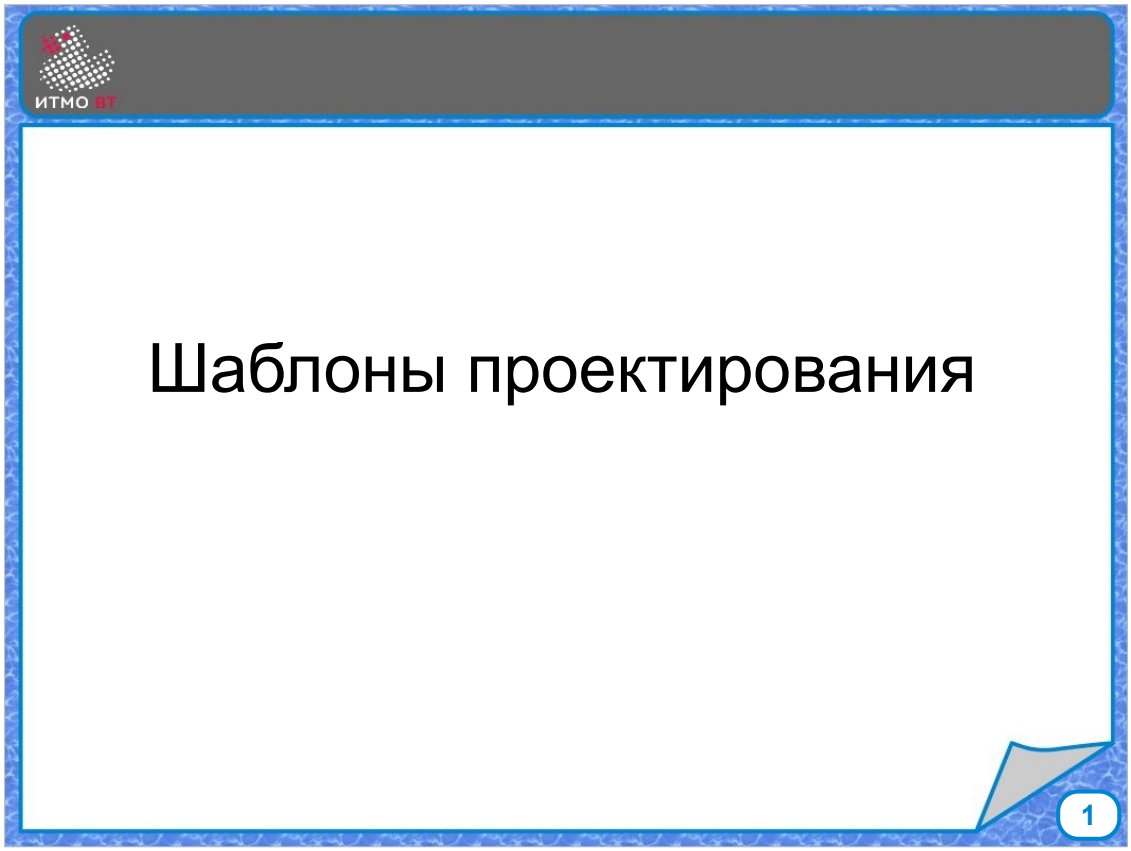
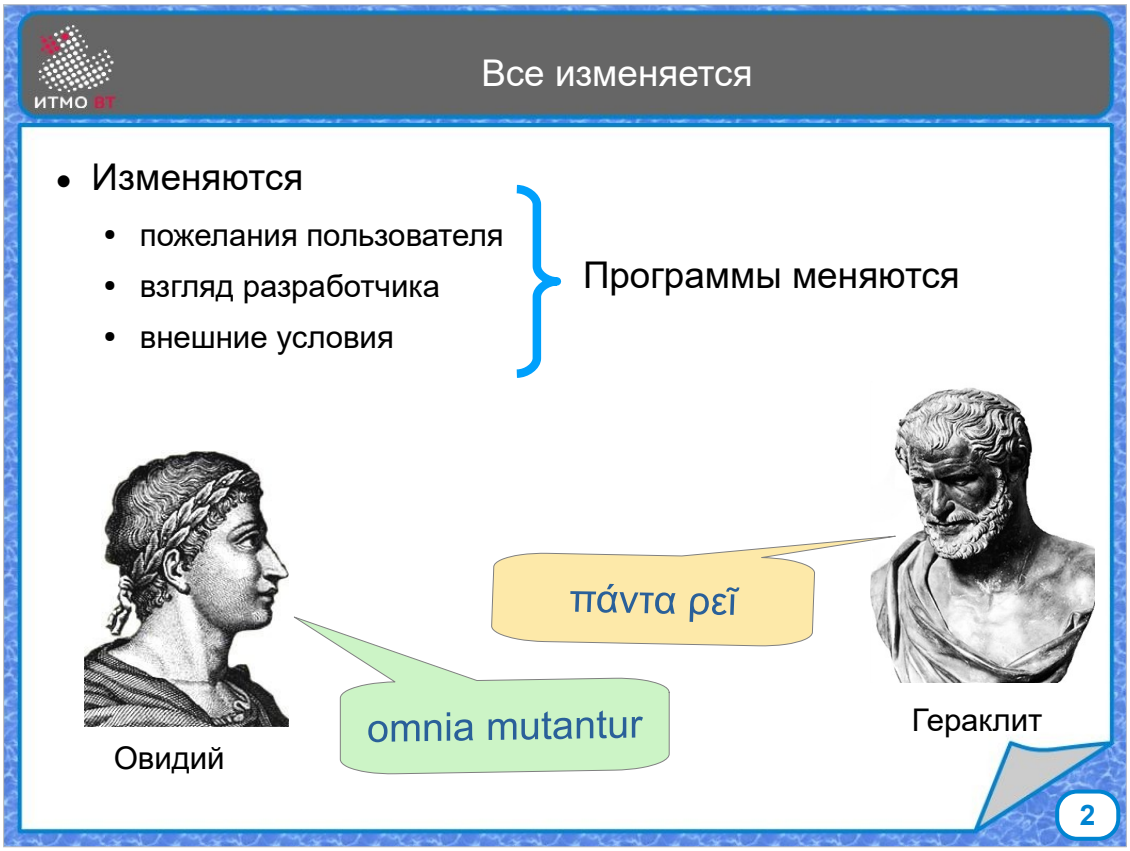
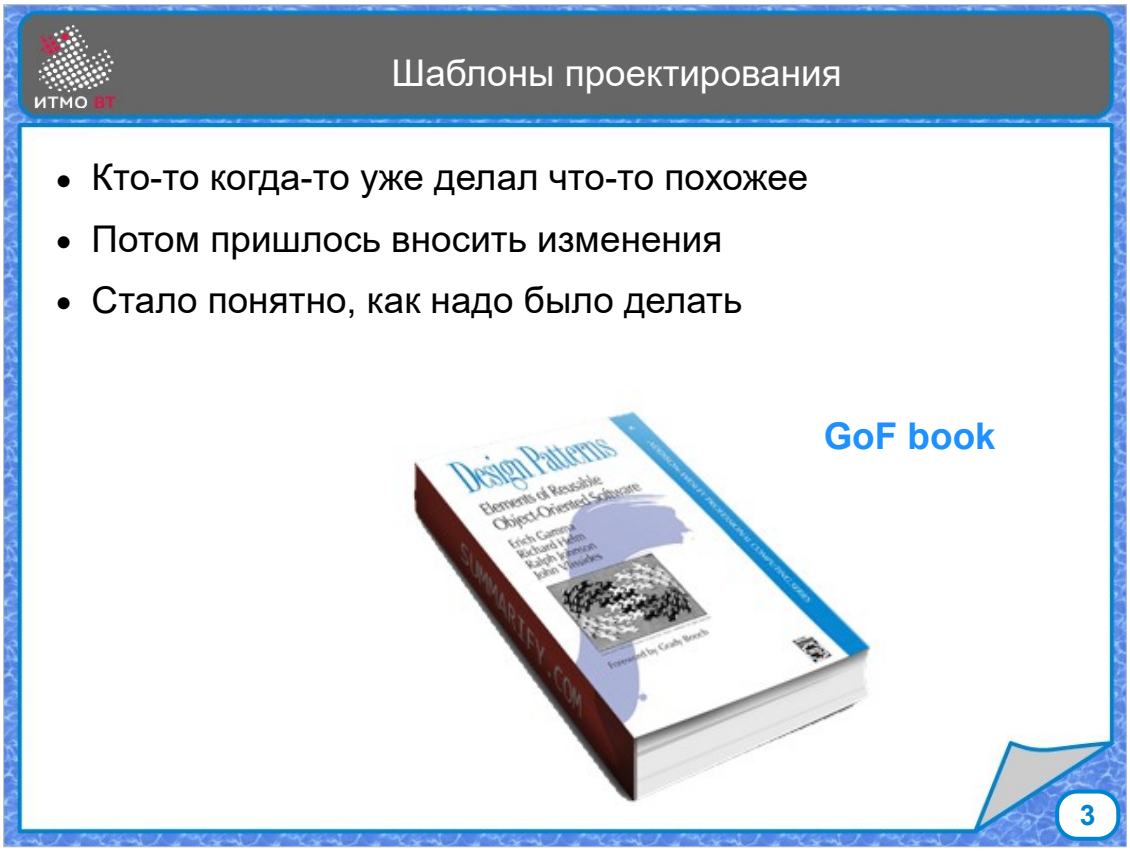

ИТМО ВТ
Шаблоны проектирования
1



Еще древние греки и римляне говорили, что все изменяется. Если рассматривать область разработки программного обеспечения, то после того, как программа написана и пользователи начали ее использовать, могут поменяться требования пользователей, они могут придумать дополнительные функции, или понять, что текущая реализация их чем-то не устраивает. Также может измениться взгляд разработчика, который видит, как пользователи используют программу, и начинает понимать, что нужно доработать. Могут поменяться внешние условия, появляются новые технологии, стандарты, библиотеки, меняется железо. В общем, программы могут меняться, и с этим надо как-то жить.



Одним из способов справиться с негативными последствиями необходимости внесения изменений является использование шаблонов проектирования. Оказывается, что многие проблемы уже когда-то у кого-то возникали. Также возникали причины для внесения изменений. Потом становились понятны способы, как можно было избежать проблем, если бы изначально была выбрана другая реализация. Соответственно, можно выделить наиболее типичные задачи, описать способ из решения, который в случае необходимости изменения программы не приведет к полному переписыванию кода.
Книга, в которой были описаны 23 шаблона объектно-ориентированного проектирования, стала классикой. Четверых ее авторов стали именовать "бандой четырех" - Gang of four, а саму книгу - GoF Book.

Большинство шаблонов основано на основных концепциях ООП, а также на нескольких базовых принципах, одним из которых является принцип взаимодействия на уровне абстракций. Если клиент хочет обратиться к какому-то сервису, то желательно делать запрос через интерфейс, чтобы не привязываться к какой-то конкретной реализации. Этот интерфейс может реализовываться неким абстрактным классом, определяющим базовое поведение. А дальше уже конкретный сервис может расширять абстрактный класс, обеспечивая нужный клиенту сервис. Какие-то элементы этой схемы могут отсутствовать в некоторых случаях. Но принцип, что взаимодействовать должны абстракции, а не реализации, остается. Примером такой схемы могут быть коллекции - интерфейс Set, классы AbstractSet и HashSet.

Следующий принцип - разумное использование наследования. Если есть базовый класс, и от него наследуется потомок, то потомок реализует все, что может делать базовый класс, и возможно что-то еще. Наследование позволяет использовать динамическое связывание (полиморфизм), что как раз и позволяет проектировать систему на уровне абстракций, а конкретную реализацию подставлять во время исполнения.

Однако, наследование должно использоваться только тогда, когда мы хотим представить статическое отношение вида класс-подкласс. Автомобиль - это подкласс транспортного средства. Вертолет - тоже. Эти отношения постоянны во времени, и именно они должны реализовываться с помощью наследования.

Но есть и другие виды отношений. Например, возникла задача перевезти корову. Можно ли метод перевозки коровы добавить в базовый класс, чтобы все потомки могли им автоматически воспользоваться?
Можно ли сказать, что любое транпортное средство пригодно для перевозки коров?

Перевозка коровы

(С) х/ф «Мимино»

Допустим, что на вертолете корову перевозили, правда это сцены из художественного фильма "Мимино", но все-таки...

С самолетом тоже вопросов не возникает, по крайней мере опять же на примере фильма "Особенности национальной охоты" можно видеть, что корова вполне помещается в бомболюк Ту-22М.

ИТМО ВТ

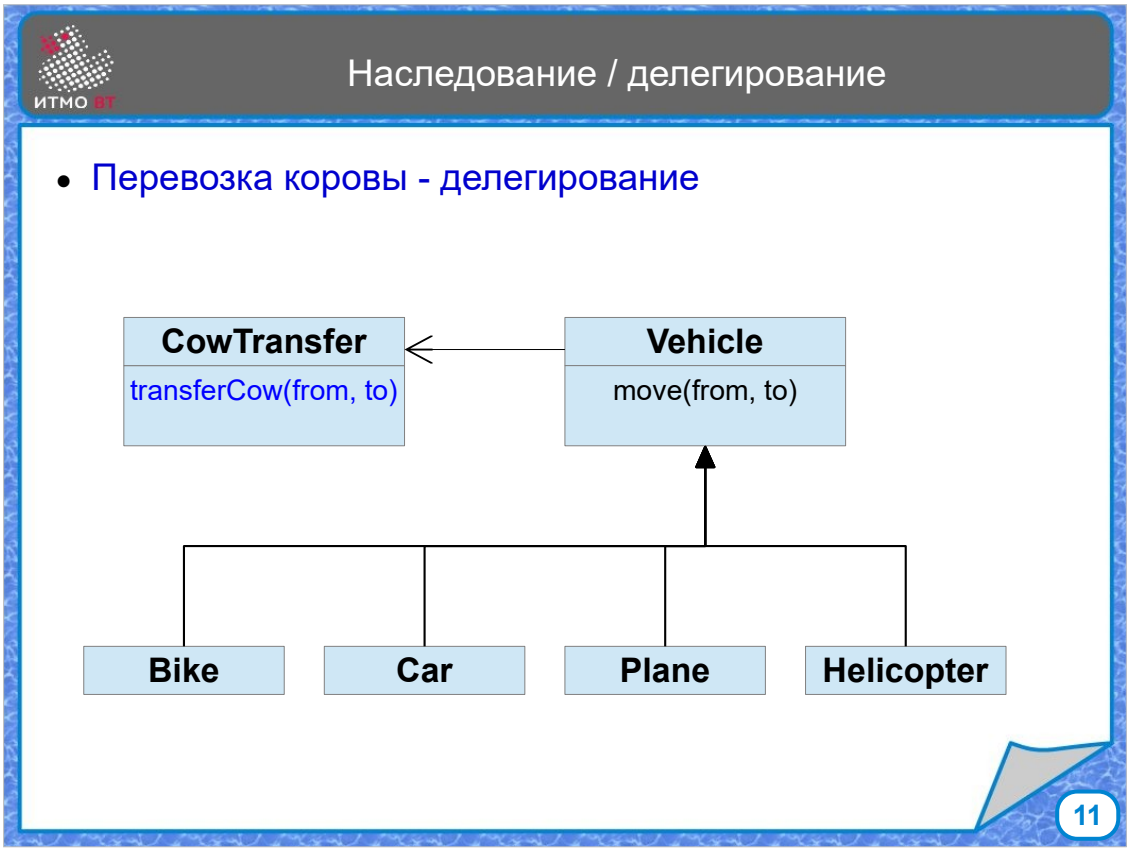
Перевозка коровы

На просторах интернета можно найти фотографии перевозки коров в легковом автомобиле.

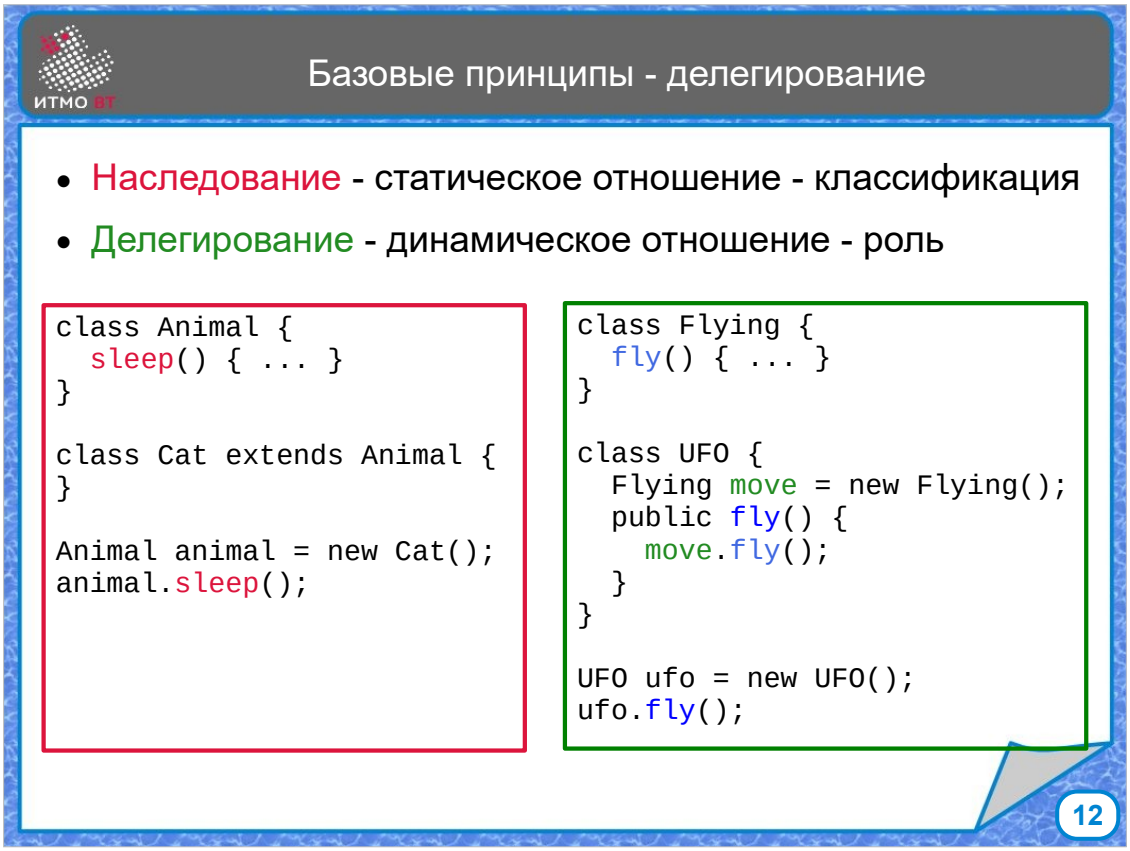
ИТМО ВТ

Перевозка коровы

Ну и перевозить корову на байке тоже можно.



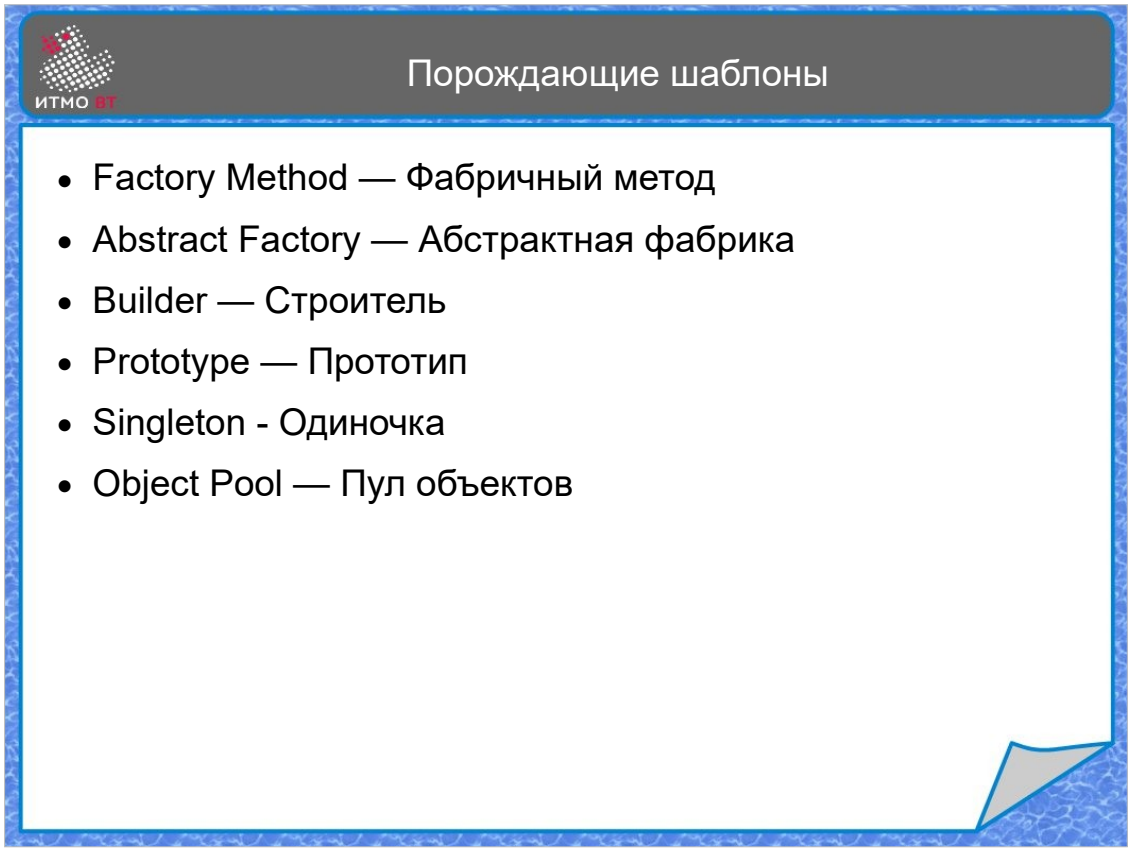
Несмотря на все фото и видеосвидетельства, с точки зрения проектирования, лучше перевозку коровы не делать постоянной особенностью класса транпортное средства. Во всех этих случаях это - временная функция - роль "корововоза". Такие отношения лучше передаются делегированием. Создаем класс, реализующий нужное поведение и позволяем ему выполнить нужное действие.



Соответственно, второй базовый принцип - для получения возможности динамического изменения поведения класса лучше использовать делегирование, оставив наследование для статических отношений, где требуется именно иерархия классов с возможностью применить полиморфизм. Наследник может вообще не изменять методы предка, либо переопределить их. Делегирование обычно реализуется созданием объекта-делегата, и при необходимости вызове методов этого делегата, заставляя делегата выполнять саму работу.

К этим двум базовым принципам можно добавить третий - чтобы решить проблему возможных изменений, нужно отделить то, что изменяется, от того, что остается постоянным. Потом выделить эти изменения в отдельную сущность и инкапсулировать. Применив принципы, можно получить шаблон. Шаблоны проектирования традиционно делятся на порождающие, структурные и поведенческие.

Порождающие шаблоны проектирования

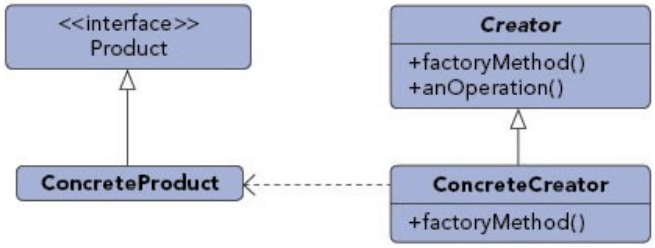


Порождающие шаблоны регулируют создание новых объектов. Это обычно делается ограничением доступа к конструкторам и предоставлением методов для создания объектов.

Factory Method

- Возможность создавать объекты разных типов с помощью фабрики
- Примеры:

```
Connection conn  = DriverManager.getConnection(args);
Statement stat = conn.createStatement();
```



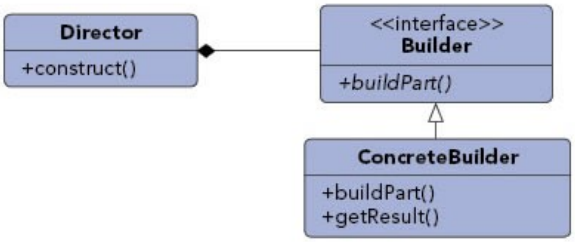
Если взаимодействовать на уровне абстракций, то в какой-то момент все равно понадобится создать конкретный объект. Шаблон Фабричный метод (Factory Method) позволяет отложить решение о конкретном типе создаваемого объекта на этап исполнения. Вместо прямого создания объекта конструктором у нас есть абстракция фабрики, метод которой возвращает абстракцию продукта. При этом метод конкретной фабрики возвращает уже конкретный продукт. Фабричный метод может выдавать либо объекты только одного типа, либо разных в зависимости от параметра. Шаблон Factory Method обеспечивает независимость кода от конкретного типа создаваемых объектов. Вместо прямого вызова конструктора с явно заданным типом мы получаем фабрику, конструирующую объекты нужного нам типа.

Шаблон Абстрактная фабрика (Abstract Factory) делает почти то же самое, но предназначен для генерации объектов не одного типа, а сразу целого семейства связанных типов. Но все равно вместо явного вызова конструктора мы получаем фабрику, которая умеет производить любой объект из заданного набора. Получив конкретную фабрику CocacolaFactory и вызвав метод получения апельсинового напитка мы получим объект типа Fanta, вызвав у той же фабрики метод получения лимонного напитка, получим Sprite. Таким образом, абстрактная фабрика - это фабрика, умеющая создавать семейство продуктов.

Builder

- Создание объектов со сложным набором параметров

```
Builder b = new Builder()
  .setWhere("Moscow")
  .setTransport("train")
  .setDate(1,6,2020);
Product p = b.create();

Director d = new Director();
p = d.makeTrainTrip("Moscow", new Date(1,6,2020));
```

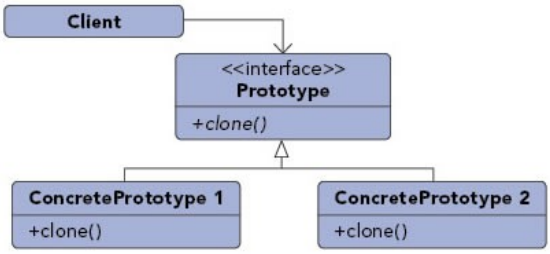


Если создаваемый объект зависит от большого числа параметров, причем некоторые из них могут иметь стандартные значения и тогда их не обязательно указывать, то есть два варианта создания объекта - либо создать объект со значениями по умолчанию, а потом с помощью сеттеров установить его поля, либо написать пару десятков конструкторов на все возможные варианты исходных значений. В первом случае проблема в том, что в процессе вызова сеттеров объект не доделан, во втором проблема в запутанных конструкторах.
Шаблон Строитель (Builder) позволяет задать все параметры в объекте-строителе, а потот одним легким движением руки одномоментно создать нужный объект. Есть вариант шаблона, когда строителем управляет директор с набором методов для создания.

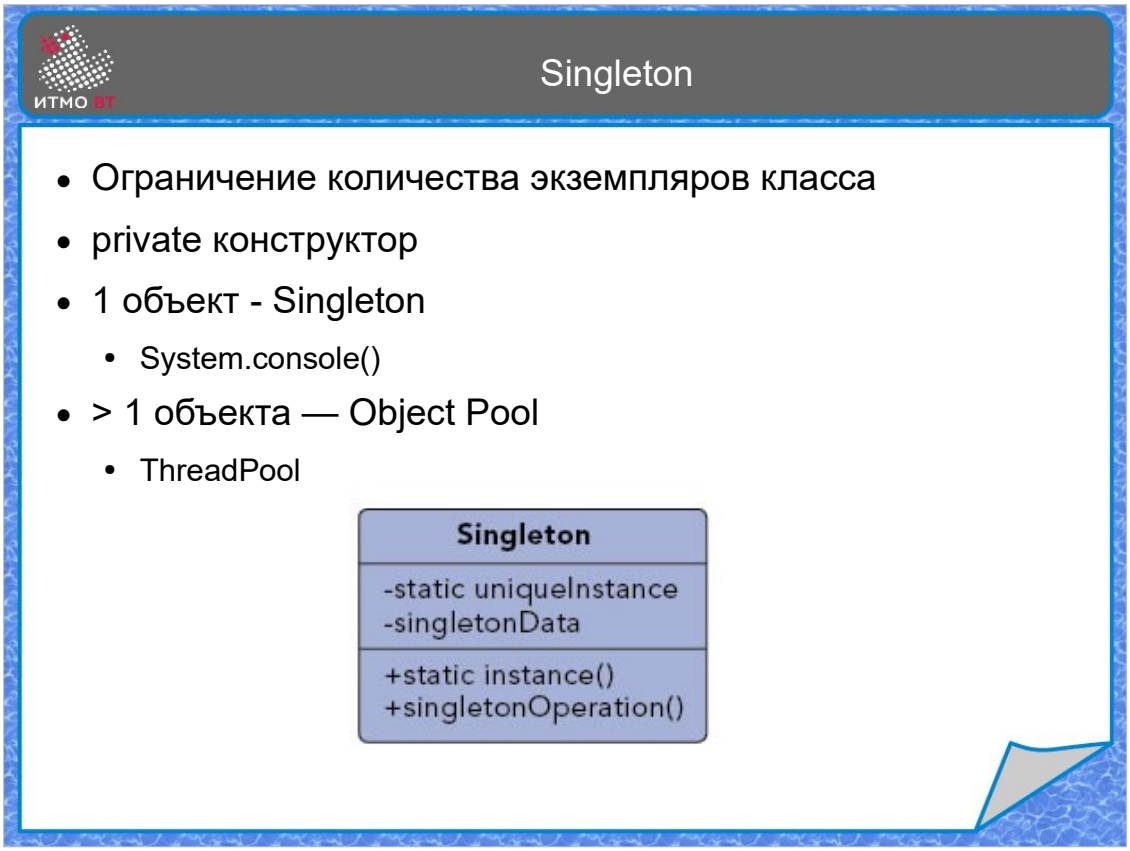
Prototype

- Создание объектов клонированием

```
Cell cell = new HumanCell();
Cell cell2 = cell.clone();
```

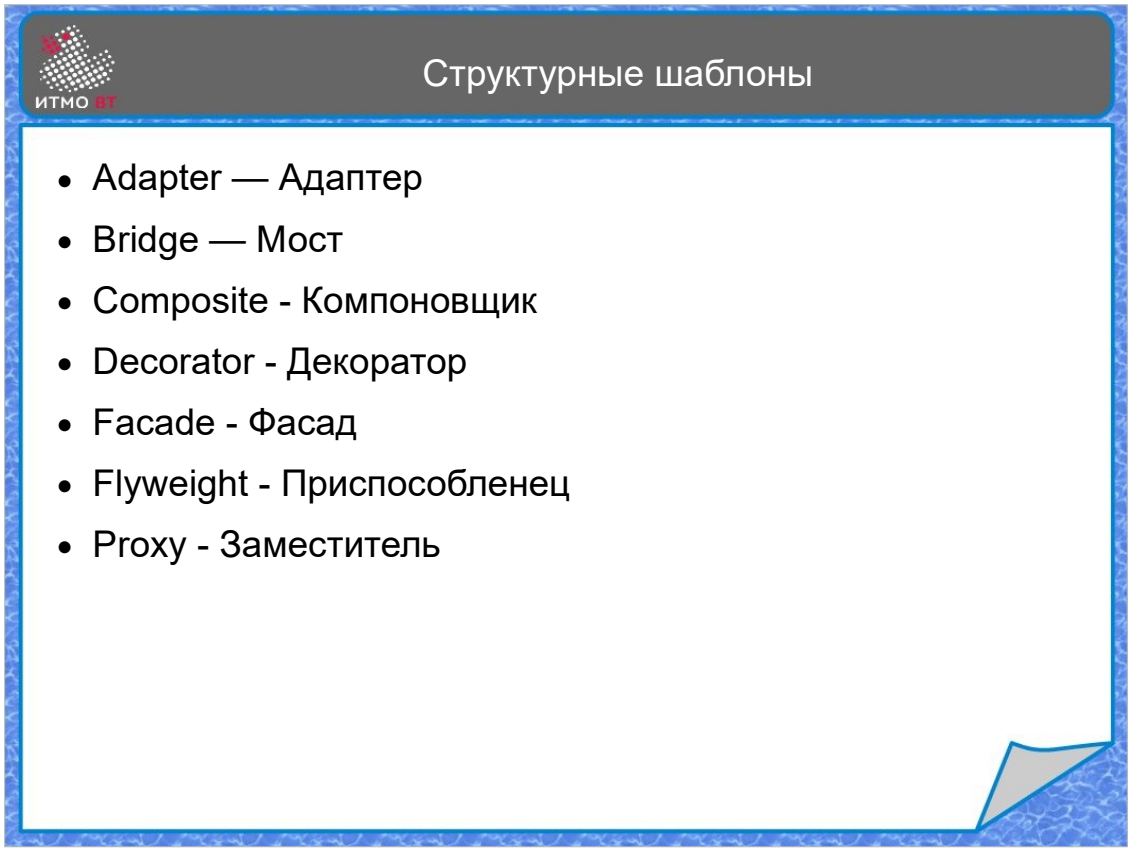


Другой способ создания сложных объектов - это клонирование. Можно заранее создать разные варианты, а потом в нужный момент создать копию и возвратить ее. Такой способ реализуется шаблоном Прототип (Prototype).



Шаблон Одиночка (Singleton) - это по факту тоже фабрика, у которой установлено ограничение на количество создаваемых объектов - он должен быть только один. После создания этого объекта, фабрика возвращает всегда один и тот же объект, не создавая новые.
Вариантом одиночки является пул объектов (Object Pool), только тут количество объектов больше одного.

Структурные шаблоны проектирования



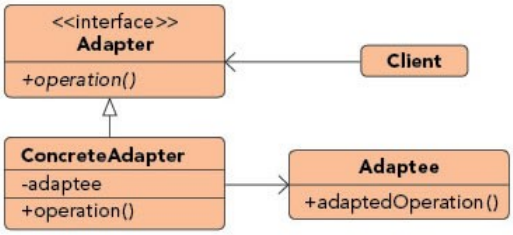
Структурные шаблоны обычно решают задачу совместного использования классов, многие из них являются частными случаями делегирования.

Шаблон Заместитель (Proxy) представляет собой случай, когда класс-делегат имеет тот же интерфейс, что и делегирующий класс. Методы те же, но вместо прямых вызовов используется прокси. Прокси может применяться в следующих случаях:
1) реальный объект долго создается, либо требует значительных ресурсов на создание, поэтому его создание откладывается до момента первого обращения
2) связь с реальным объектом может нарушаться, тогда прокси может кэшировать вызовы, либо подставлять ответ
3) прокси может перехватывать вызов методов и производить дополнительные действия (например, записывать лог), хотя такое поведение уже ближе к декоратору

Adapter

- Делегирование с заменой интерфейса

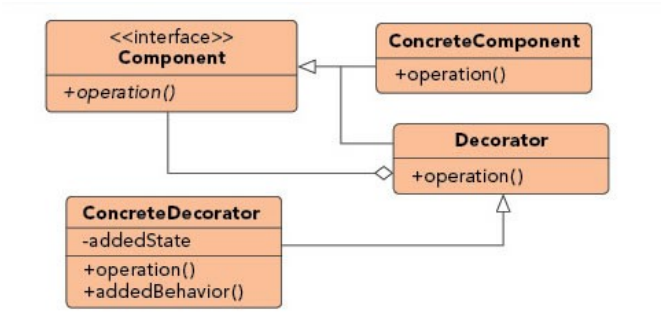
```
class EUPlug {
  connect(EUSocket s) { s.getEUPower(); }
}
interface UKSocket { getUKPower() }

class Adapter implements EUSocket {
  UKSocket uks;
  getEUPower() {
    return uks.getUKPower();
  }
}
```



Шаблон Адаптер (Adapter) применяется, когда есть уже готовый клиентский класс и готовая библиотека, предоставляющая нужную клиенту функциональность, но имеющая другой интерфейс, которым не может пользоваться клиентский класс. В этом случае, чтобы не переписывать код клиента и код библиотеки, можно написать класс-адаптер, предоставляющий клиенту нужные методы, которые внутри адаптера преобразуются в вызовы методов библиотеки.

Decorator

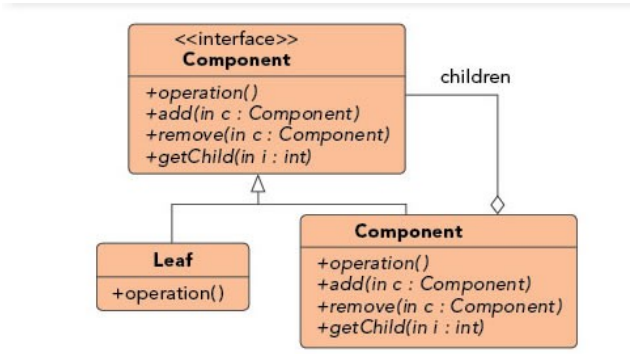
- Делегирование с расширением функциональности
- Блины с добавками в Теремке
- InputStream
  - BufferedInputStream, LineNumberInputStream, ...



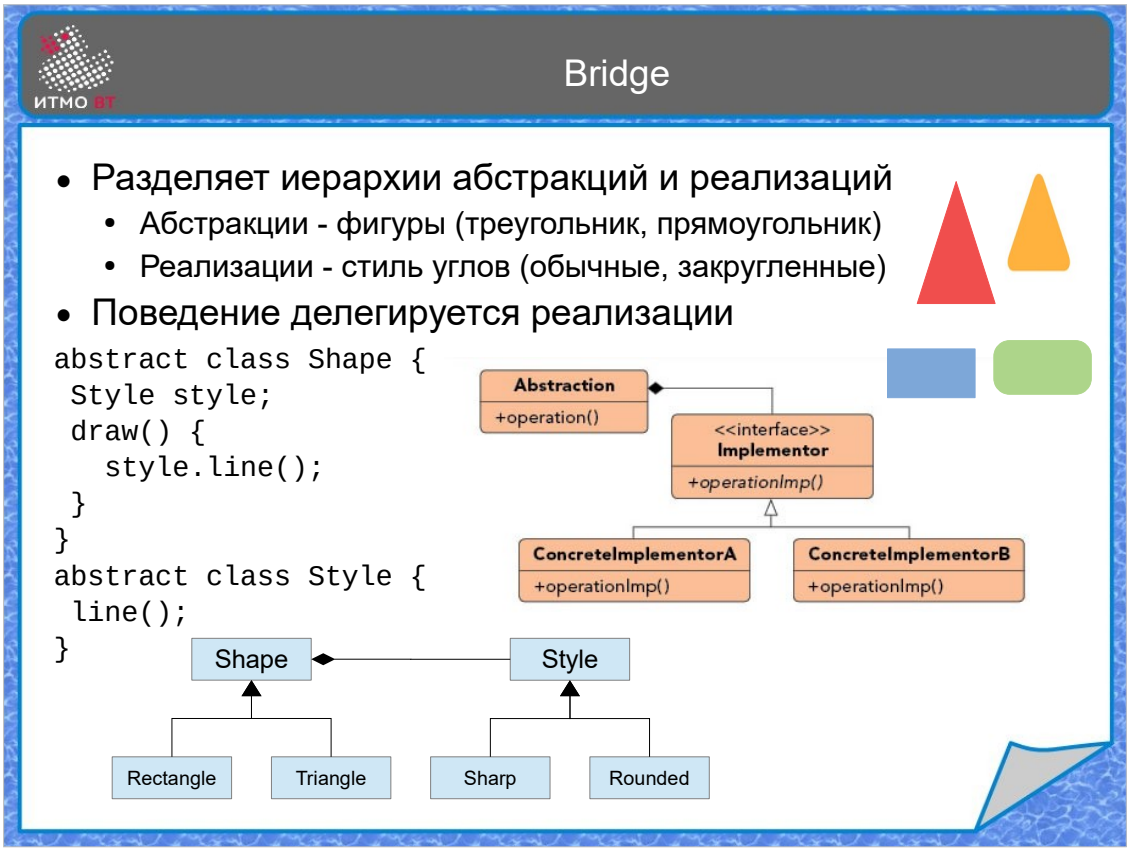
Шаблон Декоратор (Decorator) также использует тот же самый интерфейс для делегата и для делегирующего класса, при этом добавляя функциональность. Декоратор позволяет динамически добавлять классам новые функции, не переписывая при этом код.

Composite

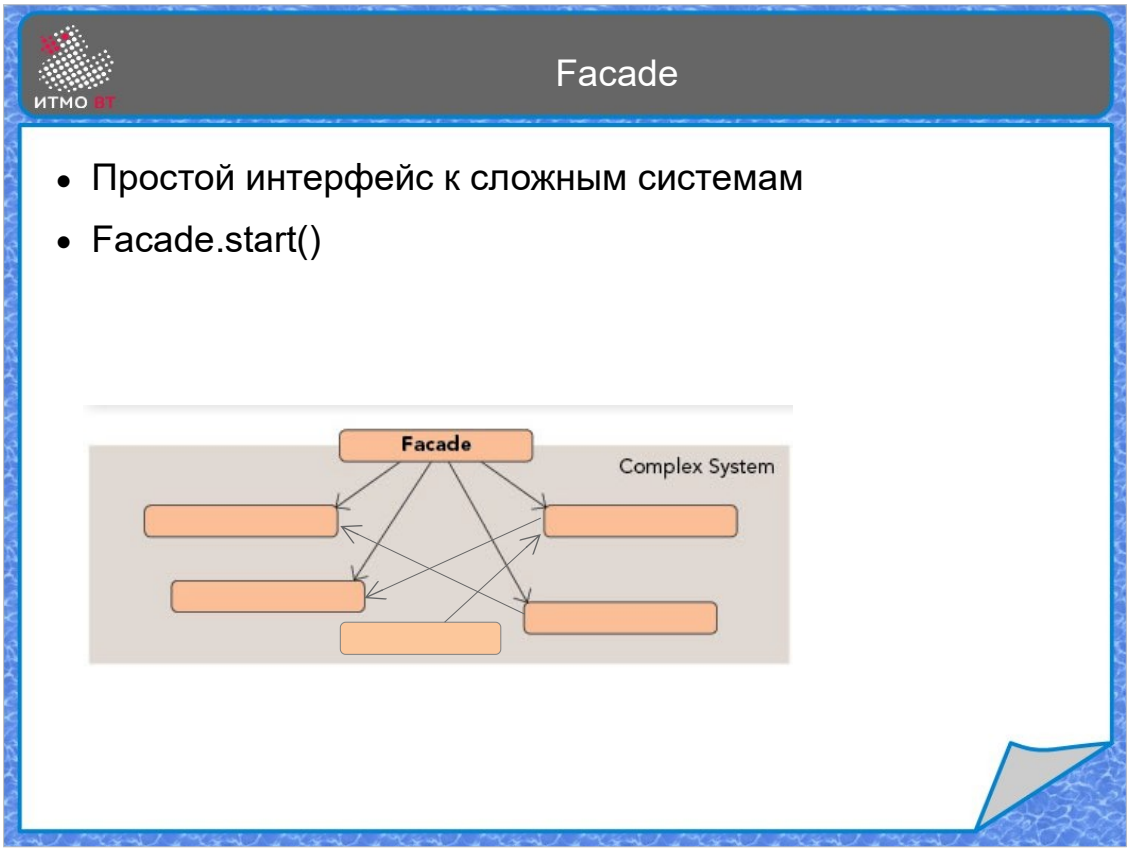
- Иерархические структуры
- GUI: контейнер — это компонент, на котором расположены другие компоненты



Шаблон Компоновщик (Composite) используется для представления иерархических структур. Он объединяет понятия коллекции и ее элемента, предоставляя им единый интерфейс, при этом компонент сам является коллекцией компонентов.



Шаблон Мост (Bridge) используется, чтобы раздельно можно было управлять иерархией абстракций и иерархией реализаций. Например, определим иерархию абстракций фигур разной формы - прямоугольники, треугольники и т.д. Все они реализуют интерфейс Shape. При этом отрисовка каждой фигуры может быть выполнена в разном стиле - с острыми углами, с закругленными углами. Рисовать каждую такую фигуру нужно по-своему. Чтобы не размножать методы рисования каждой фигуры в своем стиле, можно добавить фигуре абстракцию стиля. Конкретная реализация стиля знает, как рисовать линию в этом стиле, а метод рисования фигуры нужной формы знает, какие линии надо нарисовать для данной формы, но само рисование этих линий делегирует объекту стиля, который рисует эту линию в нужном стиле.



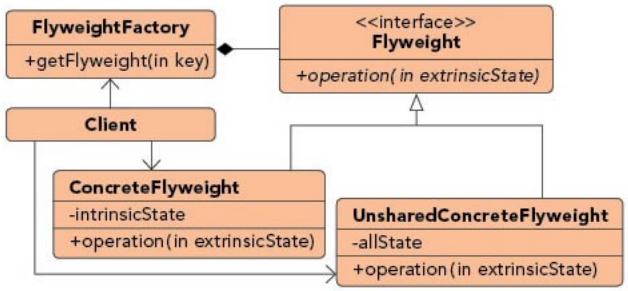
Шаблон Фасад (Facade) нужен, когда имеется сложная система объектов, но для упрощения пользования этой системой желательно предоставить клиенту простой интерфейс типа "включить-выключить", а все необходимые действия с системой выполнит фасад, вызвав методы объектов системы в нужной последовательности.

Flyweight

- Представление большого множества объектов одним
- Для экономии памяти

~~PhoneNumber number = new PhoneNumber("+79119876543");~~

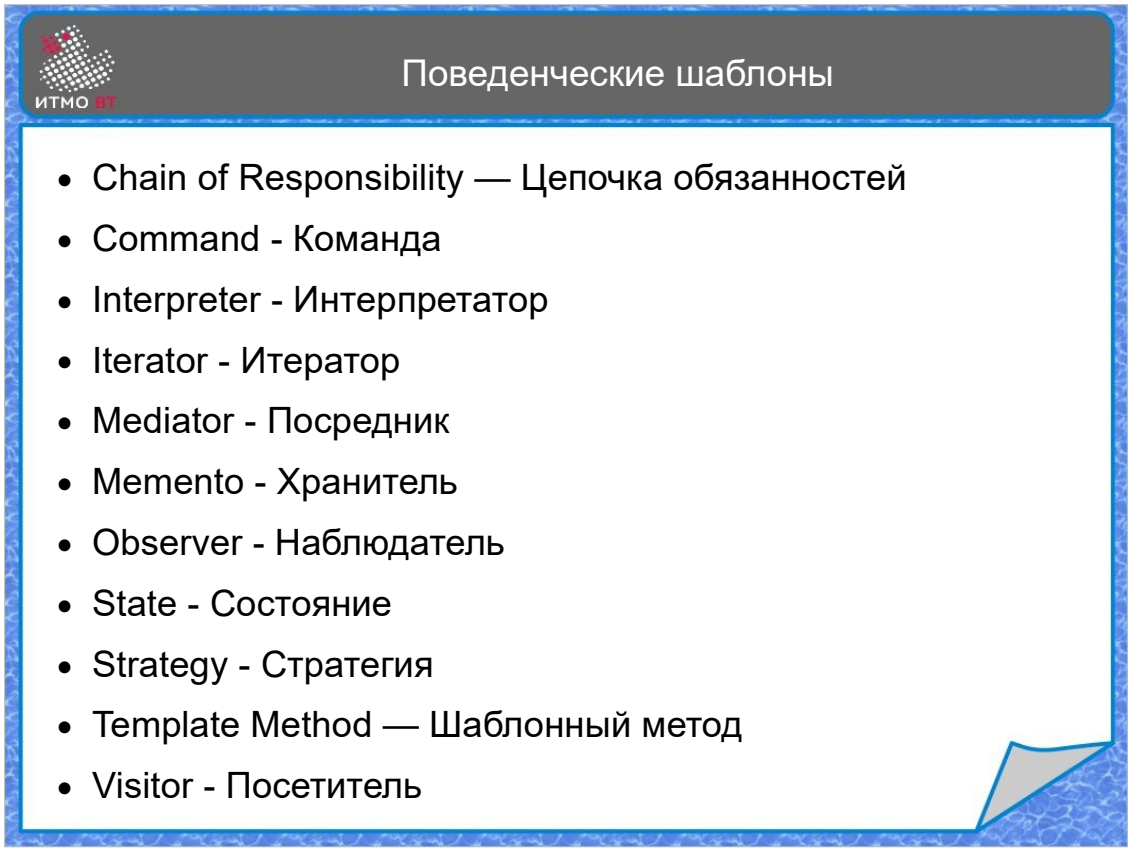
```
Phone.call("+79119876543");
```



Шаблон Приспособленец (Flyweight) предназначен для случая, когда имеется огромное количество однотипных объектов, и для их хранения требуется большое количество памяти. В этом случае можно создавать один экземпляр для множества представляемых объектов, передавая уникальные характеристики как параметры, чтобы не тратить память на их хранение.

ИТМО ВТ

Поведенческие шаблоны проектирования

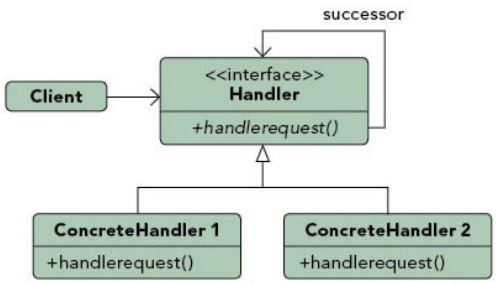


Поведенческие шаблоны реализуют различные варианты поведения. Обычно они позволяют динамически управлять ответственностью.

Chain of Responsibility

- Передача запроса по цепочке обработчиков
- Звонок в тех. поддержку

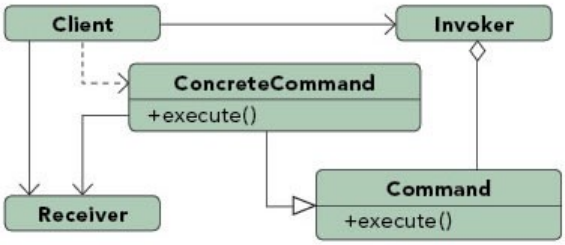
  1) взять кредит — нажмите 1, застряла карта — нажмите 2
  2) нажмите кнопку "сброс" на банкомате 2 раза
  3) стойте рядом, сейчас приедем



Шаблон Цепочка ответственности (Chain of Responsibility) позволяет клиенту не думать о том, какой из объектов выполнит его запрос. Запрос передается по цепочке исполнителей, и в итоге его выполнит тот исполнитель, который решил, что именно он способен выполнить необходимое действие.

Command

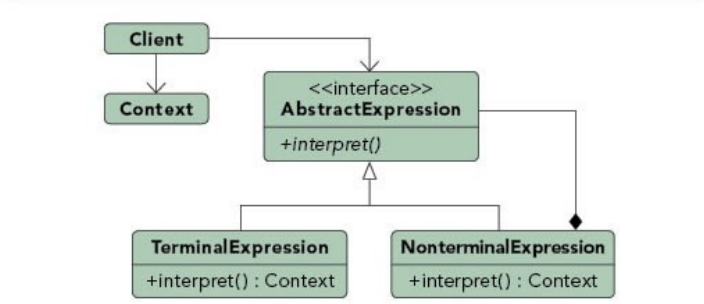
- Управление командами
- Разделение вызова и исполнения команд
- Можно организовать очередь команд и макрокоманды



Шаблон Команда (Command) позволяет разделить вызов и исполнение команды. Для вызываемого объекта все команды - экземпляры интерфейса Command, каждая отдельная команда знает, какой метод какого исполнителя нужно вызвать. Соответственно, теперь можно управлять командами единым образом, создавать очереди команд и макрокоманды (команды, состоящие из команд)

Interpreter

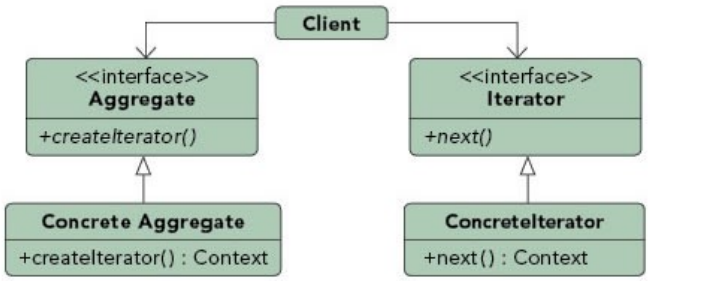
- Позволяет управлять поведением с помощью простого языка
  - Регулярные выражения
  - Форматирование строк



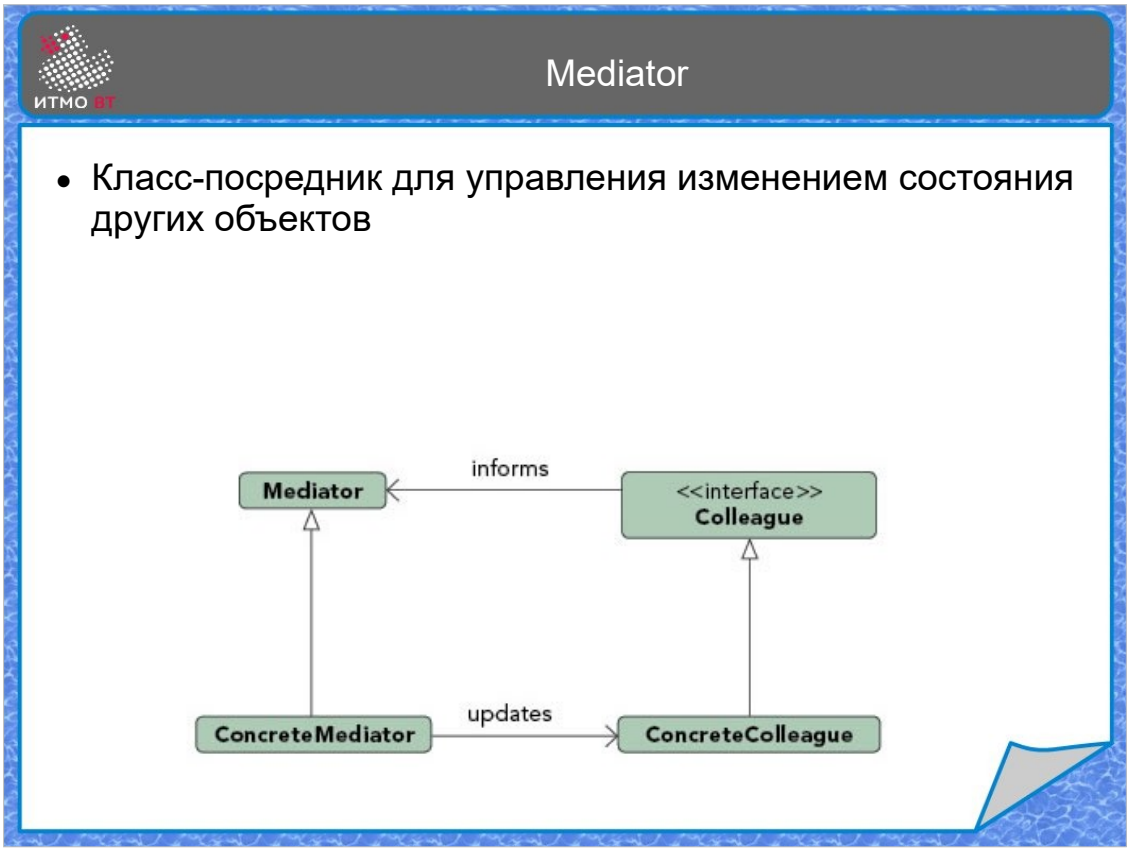
Шаблон Интерпретатор (Interpreter) позволяет управлять поведением с помощью простого языка. Грамматика языка задается выражениями, которыми могут быть терминальными (представлящие собой элементарные понятия) и нетерминальными (составленными из элементарных понятий). Для каждого элементарного и составного понятия создается отдельный класс, метод interpret() которого выполняет нужное действие. Примерами могут быть регулярные выражения и форматирование строк методом printf()

Iterator

- Последовательный доступ к элементам коллекции
- Экскурсия
  - Реальный гид
  - Аудиогид
  - Путеводитель



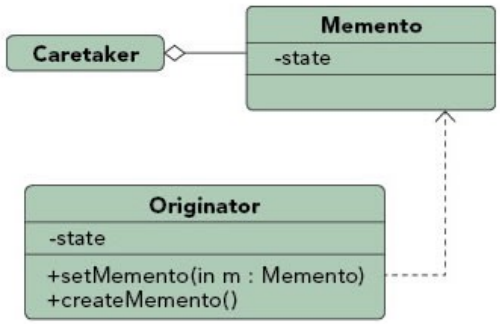
Шаблон Итератор (Iterator) позволяет обеспечить перебор элементов коллекции. Обычно сама коллекция предоставляет итератор для обхода содержащихся в ней элементов. Клиент не знает ничего о типе коллекции, он пользуется итератором, чтобы получить ее элементы.



Шаблон Посредник (Mediator) нужен, когда имеется сложная структура объектов, которым нужно как-то взаимодействовать между собой. Чтобы при добавлении нового класса, который должен взаимодействовать со всеми другими, не пришлось вносить изменения в код остальных классов, создаем посредника и заставляем все общение производить только через него.

Memento

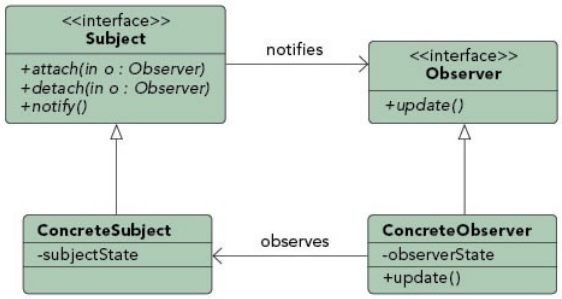
- Хранение и восстановление состояния объекта
- UNDO



Чтобы иметь возможность сохранять состояние объектов, не нарушая инкапсуляцию, и не предоставляя доступ к закрытым элементам класса, можно использовать шаблон Хранитель (Memento). Класс может создать Memento, и восстановить свое состояние из него. А единый интерфейс Memento позволяет управлять сохраненными состояниями единым образом.

Observer

- Оповещение объектов об изменении состояния
- Обработка событий:
  - Источник — наблюдаемый
  - Обработчик - наблюдатель

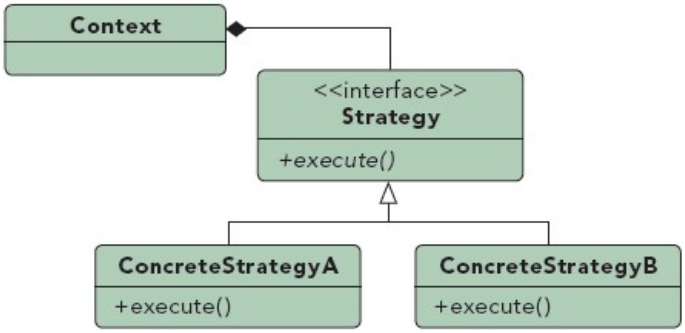


Шаблон Наблюдатель (Observer) позволяет реализовать так называемое событийное программирование. Объект-наблюдатель может подписаться на оповещения об изменениях состояния наблюдаемого объекта. Каждый раз, когда происходит обновление состояния, наблюдаемый объект вызывает метод оповещения у всех, кто за ним наблюдает.

Шаблон Состояние (State) позволяет реализовать конечный автомат, то есть автомат, имеющий конечное количество состояний, переход из которых в другие состояния задается таблицей переходов. Каждое состояние задается отдельным конкретным классом, в котором реализуется логика перехода из этого состояния в другие. Ссылка на текущее состояние при переходе меняется, отражая изменение состояния системы.

Strategy

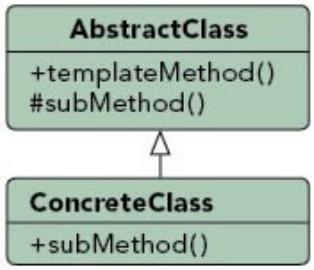
- Выбор одного из алгоритмов, реализованных в классе
  - Как добраться до аэропорта
    - Автобус
    - Такси
    - Пешком



Шаблон Стратегия (Strategy) позволяет подменять алгоритм действий. Каждый алгоритм реализуется отдельным классом-стратегией, который можно выбрать динамически.

Template Method

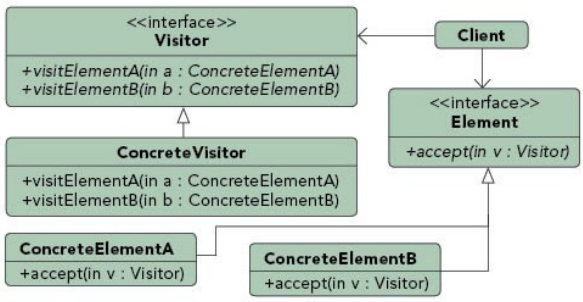
- Позволяет реализовать часть поведения в базовом классе, остальное реализуется в подклассах
- Покемоны
- Move.applyOppDamage()



Шаблонный метод (Template Method) является основой фреймворков. В нем реализуется обычно закрытый метод, в котором прописан некий алгоритм или последовательность действий. Методы, которые вызываются внутри шаблонного метода - protected, то есть могут быть переопределены в подклассах, тем самым обеспечивая разную реализацию действий, но все они будет подчиняться единому шаблону.

Visitor

- Позволяет сгруппировать операции, выполняемые над структурой объектов
- Применяется если структура элементов более стабильна, чем набор операций



Шаблон Посетитель (Visitor) применяется в случаях, когда есть заданный стабильный набор объектов, при этом возникает необходимость выполнять над ними разный набор операций. Тогда каждый объект должен иметь метод accept(Visitor), а классы, реализующие интерфейс Visitor, реализуют методы для посещения каждого элемента.