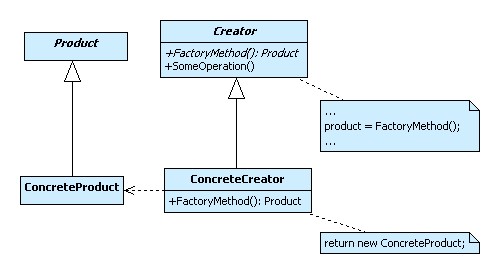
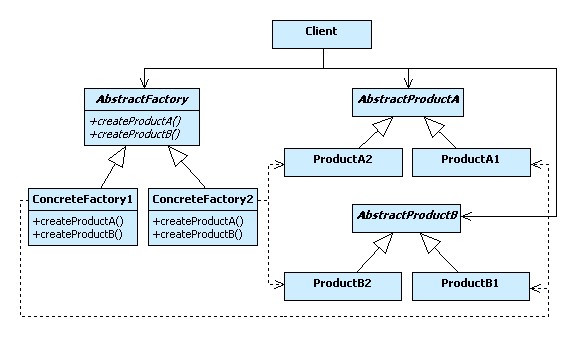
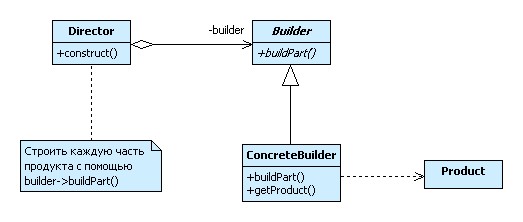
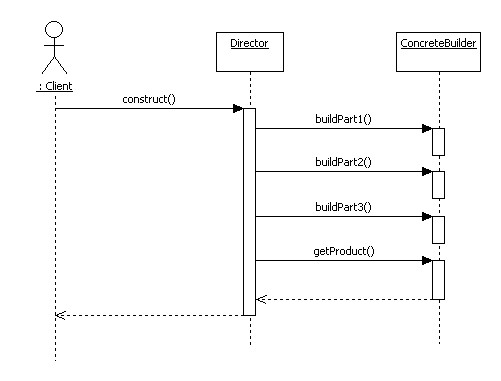
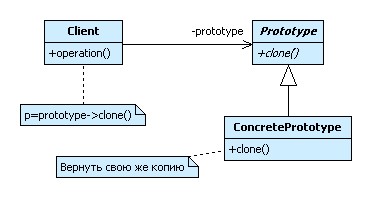
**Creational  
Factory method.** Наследники производят конкретные продукты(Базовая фабрика->Фабрика для продукта 1, Фабрика для продукта 2).

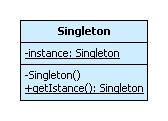
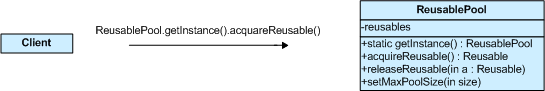


**AbstractFactory.** Базовая содержит группу продуктов. Наследники содержат реализацю группы продуктов(Еда [1е,2е,3е блюдо] -> Русская еда [русское 1е, русское 2е, русское 3е], Французская еда [фр 1е, фр 2е, фр 3е].  


**Builder.** Используется для создания сложных объектов поэтапно(из нескольких частей). Директор определяет общий алгоритм(вызов функций построить часть1, часть2, часть 3, вернуть объект). А конкертная реализация билдера это выполняет(реализация построить часть1,2,3 разная!!! Может быть пустой).  


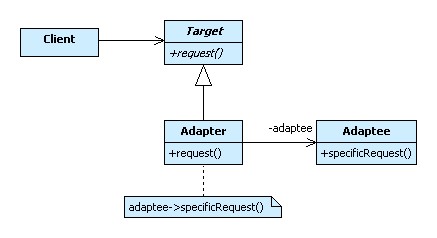


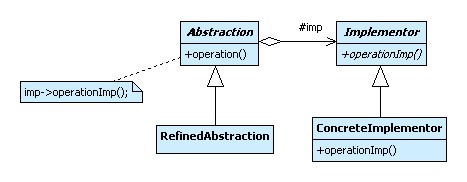
**Prototype.** Для каждого продукта создается экземпляр(прототип), который создает себе подобных методом clone(). Набор прототипов может храниться в регистре или в фабрике.  
  
**Singleton.** Есть класс со статической функцией (получить объект), и сам статический объект. Функция вызывает приватный конструктор если объект еще не был создан.

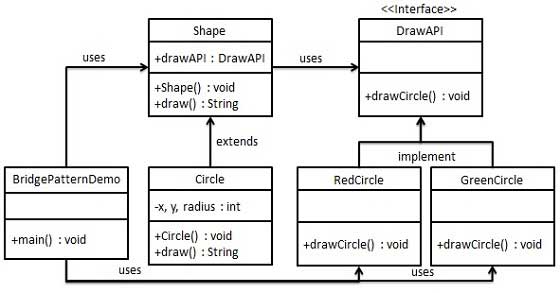
  
**Object Pool.** Когда создание экземпляров некоторого класса требует больших затрат. Обычно одиночка. Использует пул объектов, если нет свободных то может создать новый(иногда).  


**Structural**

**Adapter.** Конвертирует интерфейс класса в другой интерфейс, ожидаемый клиентом.

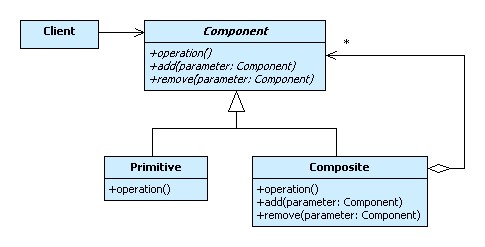
  
**Bridge.** Абстрактный интерфейс содержит указатель на конкретную реализацию, метод которой он и вызывает.



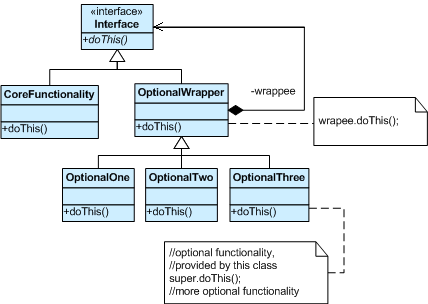


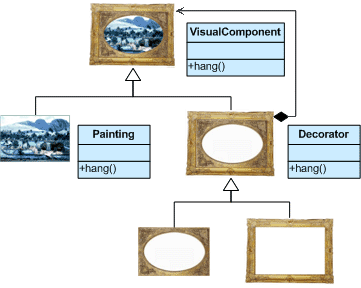
**Composite.** Необходимо объединять группы схожих объектов и управлять ими.

Объекты могут быть как примитивными (элементарными), так и составными. Код клиента работает с примитивными и составными объектами единообразно(например одна и та же функция у простых и сложных объектов).

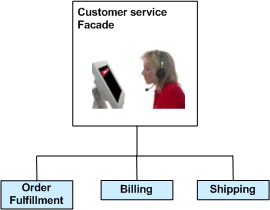


**Decorator.** Декораторы являются гибкой альтернативой порождению подклассов для расширения функциональности. Рекурсивно декорирует основной объект.Есть базовый интерфейс(минимальный функционал). От него наследуется основная реализация и Wrapper(Decorator). От декоратора тоже есть реализации. Они содержат ссылку на базовый интерфейс(рекурсивно добирается до основной реализации вызывая метод) и добавляют к ней еще функионал.





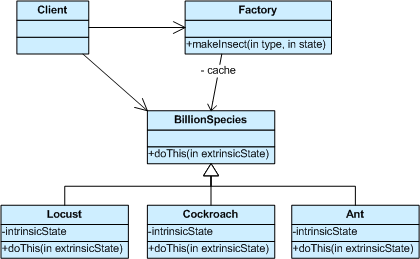
**Façade.** Создание более простого интерфейса. Скрывает сложность системы от пользователя. Делегирование к системе.



**Flyweight. П**аттерн Flyweight показывает, как эффективно разделять множество мелких объектов. Ключевая концепция - различие между внутренним и внешним состояниями.

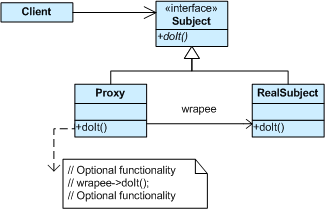
Внутреннее состояние состоит из информации, которая не зависит от контекста и может разделяться (например, имя иконки, ее ширина и высота). Оно хранится в приспособленце (то есть в классе Icon).

Внешнее состояние не может разделяться, оно зависит от контекста и изменяется вместе с ним (например, координаты верхнего левого угла для каждого экземпляра иконки). Внешнее состояние хранится или вычисляется клиентом и передается приспособленцу при вызове операций. Клиенты не должны создавать экземпляры приспособленцев напрямую, а получать их исключительно из объектаFlyweightFactory для правильного разделения.

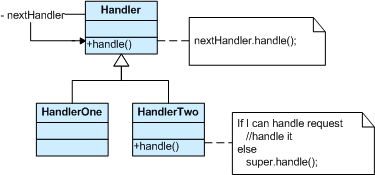


**Proxy.** Предоставляет интерфейс идентичный интерфейсу объекта. Существует четыре ситуации, когда можно использовать паттерн Proxy:

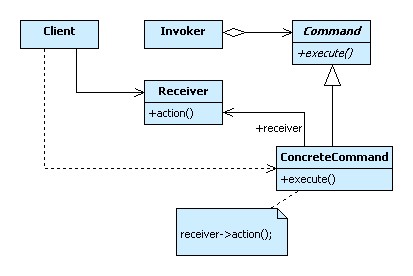
* Виртуальный proxy является заместителем объектов, создание которых обходится дорого. Реальный объект создается только при первом запросе/доступе клиента к объекту.
* Удаленный proxy предоставляет локального представителя для объекта, который находится в другом адресном пространстве ("заглушки" в RPC и CORBA).
* Защитный proxy контролирует доступ к основному объекту. "Суррогатный" объект предоставляет доступ к реальному объекту, только вызывающий объект имеет соответствующие права.
* Интеллектуальный proxy выполняет дополнительные действия при доступе к объекту.

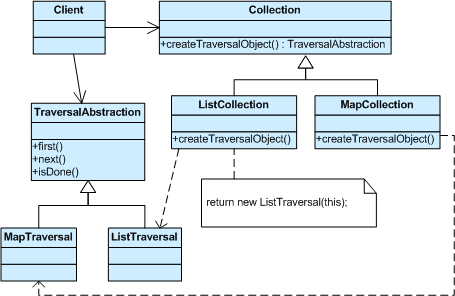


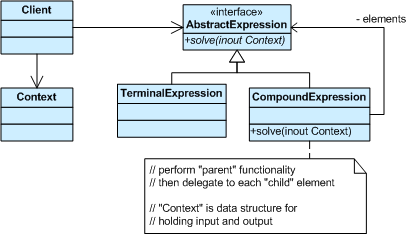
**Behavioral**

**Chain of responsibility.** Есть сообщение которое надо обработать, и несколько обработчиков(точно неизвестно кто должен обрабатывать). Все получатели связываются в рекурсивный список. Может обрабатываться несколькими обработчиками.  


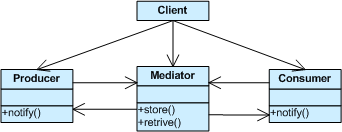
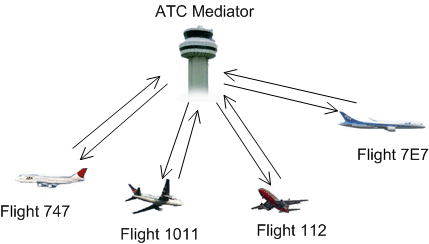
**Command.** Команда становится объектом(вместо вызова функции). Система управляется событиями. Придает системе **гибкость**, отделяя инициатора запроса от его получателя.

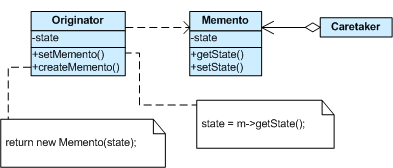


**Iterator.** Создает объект, предоставляющий последовательный доступ к элементам коллекции, не зная внутреннюю структуру коллекции и позволяя создавать несколько итераторов на одной коллекции.  


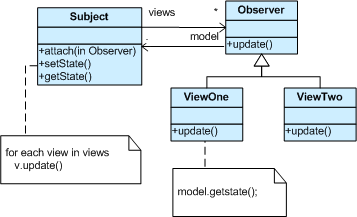
**Interpreter.**  Пусть в некоторой, хорошо определенной области периодически случается некоторая проблема. Если эта область может быть описана некоторым “языком“, то проблема может быть легко решена с помощью “интерпретирующей машины“. Строится дерево с терминальными и нетерминальными выражениями и рекурсивно обрабатывается.  


**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. Медиатор рулит процессом, может добавлять функциональности системе. Вместо вызова одним компонентом кучи других, он просто дергает медиатора, который сам знает кого дернуть.

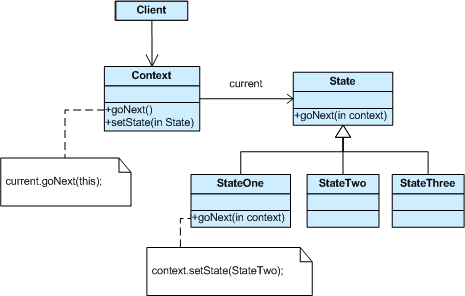
  
  
**Memento.** Есть объект сохраняющий текущее состояние другого(для отката). Есть опекун который хранит мементо. Когда надо опекун сохраняет мементо, и когда надо восстанавливает его в исходный объект.



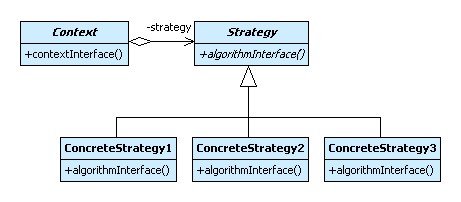
**Observer**. Есть объект у которого происходят изменения. Есть объекты которые зависят от изменений основного объекта. Как только основной меняется все узнают об этом.(издатель-подписчики). Используется в MVC(model-views)



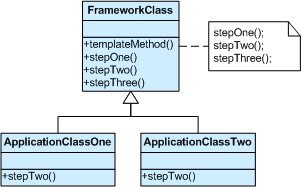
**State.** Реализация конечного автомата. Аналог большого числа if. Есть класс контекст с интерфейсом = функционалом +ссылка на текущее состояние(интерфейс с таким же функционалом). И есть много разных состояний. Когда нужно меняет реализацию у класса контекста(ссылку ставят новую реализацию). И поведение меняется. Определяет **КТО** решать задачу.



**Strategy.** Задачу можно решить несколькими алгоритмами. У них одинаковый интерфейс. Создаем класс обертку, который содержит интерфейс решения задачи. При создании обертки указываем реализацию алгоритма. Определяет **КАК** решать задачу.



**Template Method.** Используется почти во всех фреймворках. Есть базовый класс, который определяет неизменные методы, общий алгоритм их выполнения и методы, которые наследники могут изменять.



**Visitor.** Есть иерархия и набор методов, которые бы хотелось к ним применить, но не добавлять их внутрь. Для этого создается дополнительная иерархия, где подклассы будут символизировать операции, и должны принимать все возможные типы элементов 1й иерархии. При использовании элемент просто вызывает метод accept() в который передается опредленный подкласс визитера. Этот метод внутри вызывает метод visit(), и передает туда ссылку на себя. Таким образом визитер определяет конкретный класс элемента, а класс визитера определяет операцию которую необходимо выполнить. Используется в STL c++.

