# Architecture and Design Patterns

Мы работаем по SOLID и TDD, все проекты собираются на CI-сервере, у нас куча тестов и автоматическое развертывание. Мы используем современные технологии, не боимся переделывать и совершенствоваться. Того же самого мы ожидаем от тебя.

# OOD (Object Oriented Design) Principles

**Основные принципы**

1. **Инкапсуляция**

Это способность объекта с помощью модификаторов доступа скрывать свои данные и реализацию от других объектов системы. А также включение в один большой элемент множества более мелких. Модификаторы доступа позволяют обеспечить инкапсуляцию объекта

1. **Наследование**

Это механизм, при котором один или несколько классов можно получить из одного базового класса. Дочерний класс называется подклассом. Он происходит от родительского класса и наследует его свойства и методы. Родительский класс иногда называют суперклассом.

1. **Полиморфизм**

Полиморфизм состоит в том, что родительские методы и свойства могут быть вызваны дочерним классом, а также они могут быть им же переопределены.

1. **Абстрагирование**

Это способ выделить значимые свойства и методы для объекта, и исключить из рассмотрения незначительные

**Проектирование приложения по ООП-принципам**

1. Определяются требования
2. Создается модель предметной области
3. Определяются основные объекты модели (классы)
4. Определяется взаимодействие между объектами (паттерны)
5. Определяются обязанности классов (методы)

# DDD (Domain Design Development) Principles

<https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B1%D0%BB%D0%B5%D0%BC%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5>

Проблемно-ориентированное проектирование – набор принципов и схем для создания изящных систем объектов.

# Software Engineering Principles

## Design Principles

### Ссылки

<http://habrahabr.ru/post/140827/>

<http://javarevisited.blogspot.com/2012/03/10-object-oriented-design-principles.html> - то же самое, но на английском с оригинальными названиями

<http://msdn.microsoft.com/en-us/library/ee658124.aspx> - на английском основные принципы

Описаны в книгах

[Martin Fowler, Patterns of Enterprise Application Architecture](http://www.ozon.ru/context/detail/id/1829406/)  
[Kent Beck, Implementation Patterns](http://www.ozon.ru/context/detail/id/3817229/)  
[Andrew Hunt, David Thomas, The Pragmatic Programmer](http://www.ozon.ru/context/detail/id/1829500/)  
[Robert C. Martin, Clean Code: A Handbook of Agile Software Craftsmanship](http://www.ozon.ru/context/detail/id/4220508/)

Р. Мартин "Быстрая разработка программ" (Agile software development)

### 5 основных принципов SOLID

Одна из главных проблем – борьба с постоянно возрастающей сложностью системы и взаимозависимостью ее компонентов.

SOLID – это аббревиатура 5 основных принципов дизайна классов ООП

<http://ru.wikipedia.org/wiki/SOLID_%28%D0%BE%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5%29>

#### Single responsibility (принцип единственной обязанности)

Каждый класс отвечает за одну обязанность, цель.

Обязанность полностью и целиком инкапсулирована в классе. Все методы класса направлены на обеспечение этой обязанности.

<http://blog.byndyu.ru/2009/10/blog-post.html>

#### Open-closed (принцип открытости/закрытости)

Сущность открыта для расширения, но закрыта для изменения

Вместо дорогостоящей правки исходного кода, что увеличивает вероятность каскада ошибок, используется новый класс-наследник, расширяющий функционал исходного класса. Единственная причина править исходный код – наличие в нем явных ошибок.

<http://blog.byndyu.ru/2009/10/blog-post_14.html>

#### Liskov substitution (принцип подстановки Барбары Лисков)

Замена объектов наследниками без изменения свойств программы. Наследники основаны на том же интерфейсе и имеют непротиворечащий функционал.

Поведение наследуемых классов не должно противоречить поведению базового класса. Функции, использующие базовые классы, могут быть безопасно переведены на наследуемые классы.

<http://blog.byndyu.ru/2009/10/blog-post_29.html>

#### Interface segregation (принцип разделения интерфейса)

Несколько специализированных интерфейсов лучше, чем один универсальный

Клиенты не должны зависеть от методов, которые они не используют. Т.е. слишком универсальный интерфейс придется слишком часто менять. Лучше разделить его на несколько интерфейсов и инкапсулировать изменения.

<http://blog.byndyu.ru/2009/11/blog-post_19.html>

#### Dependency inversion (принцип инверсии зависимостей, не вызывайте нас – мы вас сами вызовем)

<http://blog.byndyu.ru/2009/12/blog-post.html>

Зависимости внутри системы основываются на абстракциях. Модули верхнего уровня не зависят от модулей нижнего уровня. Абстракции не зависят от деталей. Детали зависят от абстракций.

DI – это паттерн и конкретная реализация принципа Dependency Inversion.

### 5 дополнительных принципов

#### KISS

Keep as simple as possible

#### DRY

Do not repeat yourself

#### Encapsulate what changes.

Инкапсулируй то, что изменяется и отдели от неименных частей системы. Раздели приложение на неизменную и изменяющуюся части.

#### Favor composition over Inheritance.

Предпочтите композицию наследованию. <http://www.codeart.ru/2010/08/03/nasledovanie-protiv-kompozicii-chto-predpochtitelnee/>

Наследование жестко привязывает классы-наследники к базовому интерфейсу. Любое изменение базового интерфейса ломает классы-наследники.

Для решения данной проблемы используется паттерн Bridge – вместо 3-уровневого дерева наследования мы можем построить 2 2-уровневых дерева. Элементы деревьев будут порождать необходимые комбинации.

#### Programming for Interface not Implementation

Программируй в соответствии с интерфейсом, а не реализацией (код зависимый от абстракций, а не от конкретных реализаций, принцип инверсии зависимостей)

#### Delegation principle

Принцип минимальной информированности, чем меньше взаимодействующих классов – тем гибче система. Стремись к слабой связанности взаимодействующих объектов. Слабая связанность (low coupling) часто является признаком хорошо структурированной компьютерной системы и признаком хорошего дизайна.

## Principles of Extreme Programming

### DTSTTCPW

Do the simplest thing that could possibly work

Принцип экстремального программирования, созвучный принципу KISS “Keep it simple stupid”.

### YAGNI

**You ain’t gonna need it.**

Перекликается с предыдущим принципом.

Никакая функциональность не должна добавляться без реальной на то необходимости.

# Inversion of Control (IoC, Инверсия управления, Hollywood Principle)

<http://habrahabr.ru/post/244517/> - разобраны все подходы

## Введение в IoC

Важный принцип ООП, используемый для уменьшения зависимости (зацепления) внутри программы. Набор рекомендаций и подходов для написания слабосвязанного кода.

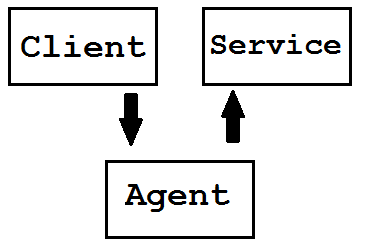
Hollywood Principle – не звоните нам, мы сами вам позвоним.

**Принципы IoC**

1. Абстракции не должны ни от чего зависеть.
2. Модули любого уровня должны зависеть от абстракций.
3. Модули верхнего уровня не должны зависеть от модулей нижнего уровня.

**Методы реализации**

Согласно подходу IoC, клиент, использующий сервер, должен использовать его не напрямую, а через посредника-прослойку-контейнер.



1. Factory Method (Design паттерн)
2. Service Locator (DI паттерн)
3. Dependency Injection (сохранение ссылки на зависимость в свойстве класса)
   1. Constructor Injection (основная зависимость)
   2. Setter Injection (базовый вариант для дополнительных зависимостей)
   3. Property/parameter Injection (самый простой способ, не TRUE OOP)
   4. Interface Injection (продвинутый вариант для дополнительных зависимостей)
4. IoC-Container

**Преимущества IoC**

1. Слабо связанный код легко тестируется и повторно используется.
2. Легкость в расширении кода.

**Недостатки IoC**

1. Логика взаимодействия программы разбросана по отдельным классам и обработчикам.

## Базовый пример без IoC (антипаттерн Control Freak)

Класс, который напрямую контролирует все зависимости.

**Принцип работы**

1. Заказы хранятся в MySQL.
2. За работу с MySQL отвечает класс MySQLOrderStorage, метод load() возвращает информацию о конкретном заказе.
3. За получение информации о заказе отвечает класс модели OrderModel, находящийся в ядре приложения, создающий внутри себя конкретную модель MySQLOrderStorage, получающий от нее заказ, и возвращающий его после обработки клиенту.
4. **Если класс MySQLOrderStorage будет сломан или отсутствует – класс OrderModel не будет работать, и придется разбирать его и лезть внутрь. Эти 2 класса жестко связаны.**

**Плюсы**

1. Все просто и очевидно. С первого взгляда понятно, кто кого вызывает и за что отвечает.

**Минусы**

1. В данном случае мы видим пример хардкода – жесткая зависимость класса OrderModel от класса MySQLOrderStorage и наличия в нем метода MySQLOrderStorage::load($oder\_id)
2. Если заказчик решится отказаться от MySQL в пользу Oracle, придется лезть внутрь класса OrderModel и менять класс.
3. Если заказчик потребует Unit-тесты, то мы не сможем заменить жестко зашитый внутри модели класс MySQLOrderStorage каким-нибудь стаб-объектом заказа. А юнит-тесты не должны лезть в бд.

Т.е. мы не сможем протестировать класс OrderModel отдельно от MySQLOrderStorage из-за их сильной связанности.

**Код**

**namespace** NoIOC {  
*// Получаем конкретный заказ и обрабатываем его.* **class** OrderModel {  
 *// В данном случае мы видим пример хардкода –  
 // жесткая зависимость класса OrderModel от класса MySQLOrderRepository  
 // Если заказчик решится отказаться от MySQL в пользу Oracle,  
 // придется лезть внутрь класса OrderModel и менять класс.  
 // Если заказчик потребует Unit-тесты, то мы не сможем заменить  
 // жестко зашитый внутри модели класс MySQLOrderRepository  
 // каким-нибудь стаб-объектом заказа.* **public function** getOrder($orderID) {  
 $orderRepository = **new** MySQLOrderStorage();  
 $order = $orderRepository->load($orderID);  
 $this->prepareOrder($order);  
 }  
 **private function** prepareOrder($order) {  
 **print 'Order: '** . $order[**'order\_id'**] . **' customer: '** . $order[**'customer'**] . **'<br/>'**;  
 }  
 }  
  
*// Храним данные о заказах в MySQL.* **class** MySQLOrderStorage {  
 **public function** load($orderID) {  
 **return array**(  
 **'order\_id'** => $orderID,  
 **'customer'** => **'Rost'**,  
 );  
 }  
 }  
  
 $model = **new** OrderModel();  
 $model->getOrder(53);  
}

## Factory Method (Design pattern)

**Принцип работы**

Клиент хочет хранить заказы уже в Oracle, но завтра может передумать и переключиться на NoSQL

1. Вместо хардкода MySQLOrderStorage внутри класса OrderModel мы решаем поместить туда высокоуровневую фабрику DbOrderStorageFactory с методом build(), который вернет new MysqlOrderStorage().
2. Однако в данный метод мы всегда можем внести однострочную правку и вернуть уже new OracleOrderStorage(). При этом код внутри самой модели-ядра нам менять не надо.
3. Для того, чтобы гарантировать, что каждое хранилище поддерживает метод load($order\_id), мы создадим интерфейс OrderStorage с данным методом. Каждый тип хранилища должен быть реализован на основе данного интерфейса.

**Плюсы**

1. Гибкость в создании хранилищ.
2. В любой момент можно изменить метод DbOrderStorageFactory::build() и вернуть новый тип хранилища.
3. Поскольку все хранилища реализуют одинаковый интерфейс, мы будем уверены, что метод load($order\_id) существует и будет вызван моделью.
4. Таким образом модель-ядро остается неизменной.
5. Код стал зависеть от более высокоуровневой фабрики, а не от низкоуровневых конкретных хранилищ.

**Минусы**

1. Класс ядра OrderModel все еще зависит от относительно низкоуровневой фабрики, что затрудняет написание юнит-тестов, поскольку требует соединения с БД (уже не обязательно с MySQL, но сути это не меняет)

**Код**

**namespace** FactoryMethodIOC {  
 **class** OrderModel {  
 **public function** getOrder($orderID) {  
 *// Мы отвязались от конкретного конечного класса типа MySQLOrderStorage,  
 // заменив его фабрикой.  
 // В любой момент по желанию заказчика мы можем изменить продукт фабрики с  
 // new MysqlOrderStorage() на new OracleOrderStorage().  
 // Кроме того, мы можем использовать еще более общую фабрику типа OrderRepositoryFactory(),  
 // которая способна вернуть и new FileOrderStorage(), и new RemoteOrderStorage().* $factory = **new** DbOrderStorageFactory();  
 $orderStorage = $factory->build();  
 $order = $orderStorage->load($orderID);  
 $this->prepareOrder($order);  
 }  
 **private function** prepareOrder($order) {  
 **print 'Order: '** . $order[**'order\_id'**] . **' customer: '** . $order[**'customer'**] . **'<br/>'**;  
 }  
 }  
  
 *// Гарантируем всем типам фабрик поддержку метода ::build.* **interface** OrderStorageFactory {  
 **public function** build();  
 }  
  
 **class** DbOrderStorageFactory **implements** OrderStorageFactory {  
 **public function** build() {  
 *// Сюда можно добавить любые типы хранилищ.* **return new** MysqlOrderStorage();  
 *// return new OracleOrderStorage();* }  
 }  
  
 *// Гарантируем всем типам хранилищ поддержку метода ::load, принимающий id заказа.* **interface** OrderStorage {  
 **public function** load($orderID);  
 }  
  
 **class** MysqlOrderStorage **implements** OrderStorage {  
 **public function** load($orderID) {  
 **return array**(  
 **'order\_id'** => $orderID,  
 **'customer'** => **'Rost'**,  
 );  
 }  
 }  
  
 **class** OracleOrderStorage **implements** OrderStorage {  
 **public function** load($orderID) {  
 **return array**(  
 **'order\_id'** => $orderID,  
 **'customer'** => **'Rost'**,  
 );  
 }  
 }  
  
 $model = **new** OrderModel();  
 $model->getOrder(53);  
}

## Service Locator (DI паттерн)

**Принцип работы**

Клиент хочет хранить заказы уже в Oracle, но завтра может передумать и переключиться на NoSQL

1. Отличие от Фабрики в том, что вместо хардкода MySQLOrderStorage внутри класса OrderModel мы решаем поместить туда объект Singleton сервис-локатор с методом getStorage(‘OrderStorage’), который вернет то хранилище, которые ему будет передано в вызове клиента.
2. В клиентском же вызове мы передадим в сервис-локатор нужное хранилище MySQLOrderStorage(), которое всегда можно прямо в клиенте заменить на OracleOrderStorage() и на любой другой.

**Плюсы**

1. Основной плюс (и отличие от Фабрики) в том, что передача нужного хранилища происходит теперь в клиентском коде. Таким образом, при тестировании мы можем заменить реальные объекты типа OrderStorage искусствеными мок-объектами. Или загрузить объект из файла.

**Минусы**

1. Многие считают сервис-локатор (синглтон по сути) антипаттерном, поскольку это глобальный объект, который доступен в любой части кода любому объекту в системе. А потому повышает зависимость.

И это плохо, потому что внутри black box что-то происходит и зависит от неких глобальных данных, в которых что-то может прийти извне. И поэтому мы не контролируем данные объекты полностью.

**Код**

**namespace** ServiceLocatorIOC {  
 **class** OrderModel {  
 **public function** getOrder($orderID) {  
 *// Будет получено то хранилище, которое было загружено в сервис-локатор клиентском коде.* $orderStorage = ServiceLocator::*getInstance*()->getStorage(**'OrderStorage'**);  
 $order = $orderStorage->load($orderID);  
 $this->prepareOrder($order);  
 }  
 **private function** prepareOrder($order) {  
 **print 'Order: '** . $order[**'order\_id'**] . **' customer: '** . $order[**'customer'**] . **'<br/>'**;  
 }  
 }  
  
 *// Основан на паттерне Singlton.* **class** ServiceLocator {  
 **private $storages** = **array**();  
 **private static** *$serviceLocatorInstance* = **NULL**;  
  
 **private function** \_\_construct(){}  
  
 **public static function** getInstance() {  
 **if** (*is\_null*(**self**::*$serviceLocatorInstance*)){  
 **self**::*$serviceLocatorInstance* = **new** ServiceLocator();  
 }  
 **return self**::*$serviceLocatorInstance*;  
 }  
 **public function** loadStorage($name, $service) {  
 $this->**storages**[$name] = $service;  
 }  
 **public function** getStorage($name) {  
 **if** (!**isset**($this->**storages**[$name])){  
 **throw new** \InvalidArgumentException(**'Such storage not found!'**);  
 }  
 **return** $this->**storages**[$name];  
 }  
 }  
  
 *// НИЖЕ ТА ЖЕ ЛОГИКА ЧТО И В ФАБРИКЕ.  
 // Гарантируем всем типам хранилищ поддержку метода ::load, принимающий id заказа.  
 // Логика неизменна, как и в примере с Factory Method.* **interface** OrderStorage {  
 **public function** load($orderID);  
 }  
  
 **class** MysqlOrderStorage **implements** OrderStorage {  
 **public function** load($orderID) {  
 **return array**(  
 **'order\_id'** => $orderID,  
 **'customer'** => **'Rost'**,  
 );  
 }  
 }  
  
 **class** OracleOrderStorage **implements** OrderStorage {  
 **public function** load($orderID) {  
 **return array**(  
 **'order\_id'** => $orderID,  
 **'customer'** => **'Rost'**,  
 );  
 }  
 }  
  
 *// Хранилище передается в клиентском коде и всегда может быть заменено.  
 // Для тестирование сюда всегжа можно передать мок-объект.* ServiceLocator::*getInstance*()->loadStorage(**'OrderStorage'**, **new** MysqlOrderStorage());  
 $model = **new** OrderModel();  
 $model->getOrder(53);  
}

## DI – введение

<http://tutorials.jenkov.com/dependency-injection/index.html> - DI по шагам (англ)

<http://zlob.in/2013/12/chto-takoe-dependency-injection-vnedrenie-zavisimostej/> - DI в PHP

<http://habrahabr.ru/post/166287/> - DI-антипаттерны или как делать не надо

DI – это паттерн, который имеет смысл именно в ООП. Это конкретная реализация принципа D (Dependency Inversion) из SOLID.

В приложении с использованием ООП разработчик оперирует и работает с объектами. Каждый объект нацелен на выполнение определенных функций и не исключено, что внутри него инкапсулируются другие объекты. Получается зависимость одного объекта от другого, в результате которой родительскому объекту предстоит управлять состоянием экземпляров потомков. Шаблон [внедрение зависимости](http://ru.wikipedia.org/wiki/%D0%92%D0%BD%D0%B5%D0%B4%D1%80%D0%B5%D0%BD%D0%B8%D0%B5_%D0%B7%D0%B0%D0%B2%D0%B8%D1%81%D0%B8%D0%BC%D0%BE%D1%81%D1%82%D0%B8) (Dependency Injection, DI) призван избавиться от такой необходимости и предоставить управление зависимостями внешнему коду.

Dependency Injection (DI) — это набор паттернов и принципов разработки програмного обеспечения, которые позволяют писать слабосвязный код.

По сути DI – это предоставление внешнего сервиса какому-то классу путем добавления ссылки на объект сервиса в свойство класса с помощью специального метода (через конструктор или через метод-сеттер). И затем использование свойства класса вместо прямого вызова объекта.

Т.е. вместо простой (и зависимой) схемы, где A внутри создает В и вызывает его метод (и зависит от него)

A принимает B извне как параметр, и сохраняет его. А затем с помощью другого метода вызывает метод B. Таким образом, вместо В всегда можно добавить объект C такого же типа.

Такие классы очень легко поддерживать и расширять, они очень легки в тестировании, поскольку сервис-объект легко может быть подменен мок-объектом.

### DI: constructor injection

Самый простой способ.

В данном случае зависимость внедряется прямо при создании объекта-модели в конструкторе.

**Такой способ является основным при добавлении главных зависимостей, без которых весь класс не будет работать.**

**Код**

**namespace** ConstructorInjectionIOC {  
 **class** OrderModel {  
 **private $storage**;  
  
 *// Зависимость внедряется в виде параметра в конструкторе модели.* **public function** \_\_construct(OrderStorage $storage) {  
 $this->**storage** = $storage;  
 }  
 **public function** getOrder($orderID) {  
 $order = $this->**storage**->load($orderID);  
 $this->prepareOrder($order);  
 }  
 **private function** prepareOrder($order) {  
 **print 'Order: '** . $order[**'order\_id'**] . **' customer: '** . $order[**'customer'**] . **'<br/>'**;  
 }  
 }  
  
 *// НИЖЕ ТА ЖЕ ЛОГИКА ЧТО И В ФАБРИКЕ.  
 // Гарантируем всем типам хранилищ поддержку метода ::load, принимающий id заказа.  
 // Логика неизменна, как и в примере с Factory Method.* **interface** OrderStorage {  
 **public function** load($orderID);  
 }  
  
 **class** MysqlOrderStorage **implements** OrderStorage {  
 **public function** load($orderID) {  
 **return array**(  
 **'order\_id'** => $orderID,  
 **'customer'** => **'Rost'**,  
 );  
 }  
 }  
  
 **class** OracleOrderStorage **implements** OrderStorage {  
 **public function** load($orderID) {  
 **return array**(  
 **'order\_id'** => $orderID,  
 **'customer'** => **'Rost'**,  
 );  
 }  
 }  
  
 $model = **new** OrderModel(**new** MysqlOrderStorage());  
 $model->getOrder(53);  
}

### DI: setter injection

Просто передадим опциональную зависимость в класс через метод-сеттер.

Таким образом, мы всегда можем передать произвольное хранилище либо загрузить его из mock-объекта.

**Такой способ хорош для добавления дополнительных зависимостей.**

Базовый DI-контейнер через сеттер

**class** DI {  
 **protected** $value;  
  
 **function** setValue($value) {  
 $this->storage = $value;  
 }  
 **function** getValue() {  
 **return** $this->value;  
 }  
}

**Код**

**namespace** SetterInjectionIOC {  
 **class** OrderModel {  
 **private $storage**;  
 *// Просто и очевидный способ добавления зависимости:  
 // просто добавляем ссылку на нужную зависимость в свойство данного класса.* **public function** setStorage(OrderStorage $storage) {  
 $this->**storage** = $storage;  
 }

**public function** getOrder($orderID) {  
 $order = $this->**storage**->load($orderID);  
 $this->prepareOrder($order);  
 }  
 **private function** prepareOrder($order) {  
 **print 'Order: '** . $order[**'order\_id'**] . **' customer: '** . $order[**'customer'**] . **'<br/>'**;  
 }  
 }  
  
 *// НИЖЕ ТА ЖЕ ЛОГИКА ЧТО И В ФАБРИКЕ.  
 // Гарантируем всем типам хранилищ поддержку метода ::load, принимающий id заказа.  
 // Логика неизменна, как и в примере с Factory Method.* **interface** OrderStorage {  
 **public function** load($orderID);  
 }  
  
 **class** MysqlOrderStorage **implements** OrderStorage {  
 **public function** load($orderID) {  
 **return array**(  
 **'order\_id'** => $orderID,  
 **'customer'** => **'Rost'**,  
 );  
 }  
 }  
  
 **class** OracleOrderStorage **implements** OrderStorage {  
 **public function** load($orderID) {  
 **return array**(  
 **'order\_id'** => $orderID,  
 **'customer'** => **'Rost'**,  
 );  
 }  
 }  
  
 $model = **new** OrderModel();  
 $model->setStorage(**new** MysqlOrderStorage());  
 $model->getOrder(53);  
}

### DI: property/parameter injection

Максимальное упрощение предыдущего подхода (не TRUE OOP way!!!)

Просто присвоить новое свойство напрямую

$model->value = $value

Это не true-подход, поскольку задание любого свойства должно жестко контролироваться соответствующим методом).

### DI: interface injection

Данный вариант является, по сути, усовершенствованным вариантом Setter Injection.

Создадим интерфейс для модели OrderModel и поместим в него метод setStorage(OrderStorage $storage).

Таким образом, класс OrderModel обязан теперь реализовать в себе метод-сеттер.

**Такой способ хорош для добавления дополнительных зависимостей.**

**Код**

**namespace** InterfaceInjectionIOC {  
 **interface** OrderStorageInject {  
 **public function** setStorage(OrderStorage $storage);  
 }  
  
 **class** OrderModel **implements** OrderStorageInject {  
 **private $storage**;  
  
 *// Чуть более усложненный способ добавления зависимости,  
 // нежели Setter Injection  
 // Через использование интерфейса насильно принуждаем модель  
 // принять в метод-сеттер нужную зависимость.* **public function** setStorage(OrderStorage $storage) {  
 $this->**storage** = $storage;  
 }  
 **public function** getOrder($orderID) {  
 $order = $this->**storage**->load($orderID);  
 $this->prepareOrder($order);  
 }  
 **private function** prepareOrder($order) {  
 **print 'Order: '** . $order[**'order\_id'**] . **' customer: '** . $order[**'customer'**] . **'<br/>'**;  
 }  
 }  
  
 *// НИЖЕ ТА ЖЕ ЛОГИКА ЧТО И В ФАБРИКЕ.  
 // Гарантируем всем типам хранилищ поддержку метода ::load, принимающий id заказа.  
 // Логика неизменна, как и в примере с Factory Method.* **interface** OrderStorage {  
 **public function** load($orderID);  
 }  
  
 **class** MysqlOrderStorage **implements** OrderStorage {  
 **public function** load($orderID) {  
 **return array**(  
 **'order\_id'** => $orderID,  
 **'customer'** => **'Rost'**,  
 );  
 }  
 }  
  
 **class** OracleOrderStorage **implements** OrderStorage {  
 **public function** load($orderID) {  
 **return array**(  
 **'order\_id'** => $orderID,  
 **'customer'** => **'Rost'**,  
 );  
 }  
 }  
  
 $model = **new** OrderModel();  
 $model->setStorage(**new** MysqlOrderStorage());  
 $model->getOrder(53);  
}

## IoC-Container

### Введение

<http://habrahabr.ru/post/132084/>

<http://habrahabr.ru/post/229879/>

**Популярные standalone-проекты**

<http://pimple.sensiolabs.org/> - от создателя Симфони

<http://php-di.org/>

<http://pressing-matters.io/encase-a-lightweight-ioc-container-for-php/>

**Встроенные контейнеры фреймворков**

Symfony 2 — <http://symfony.com/doc/current/components/dependency_injection/introduction.html>

Laravel — <http://laravel.com/docs/4.2/ioc>

Yii 2 — <http://www.yiiframework.com/doc-2.0/guide-concept-di-container.html>

**IoC-контейнер – это объект (библиотека, фреймворк), реализующий DI и автоматизирующий регистрацию и внедрение новых зависимостей. Популярные решения: Pimple, Symphony DI.**

Выше мы разобрали базовые примеры, где один класс А зависит от другого класса В.

Однако в реальной жизни все сложнее: класс А зависит от класса В, который в свою очередь зависит от классов M и N, а также от класса С, у которого куча своих зависимостей.

Теперь каждый раз, когда нам нужен класс А, нам придется строить и передавать все эти зависимости. И это сложно и порождает кучу копипаста, нарушая принцип DRY.

DI-контейнер инициализирует все переданные ему классы и всегда готов вернуть на любой объект любого класса с любыми зависимостями одной простой командой. Теперь нам не надо строить создание объекта каждый раз с нуля. Причем мы можем объявить класс как синглтон, и возвращать один и тот же объект, либо каждый раз возвращать новые объекты.

### Pimple 3.0

<http://pimple.sensiolabs.org/>

<https://github.com/silexphp/Pimple>

Pimple воплощает в себе принцип DRY – вместо повторной инициализации класса раз за разом, мы храним уже инициализированный класс. Похож по задумке на Service Locator (внешний объект для хранения ссылок на другие объекты) и основан на анонимных функциях и интерфейсе SPL ArrayAccess (поэтому работа с ним напоминает работу с обычным массивом).

Управляет 2 видами данных: параметрами и сервисами.

**Сервис – это объект**, является частью некоей большей системы: соединение с БД, объект-шаблонизатор, объект-логгер. Кроме того, сервисом может быть любой глобальный объект (синглтон).

По умолчанию, Pimple возвращает один и тот же объект для повторного использования. Однако он также имеет метод ->factory() для создания каждый раз нового объекта.

**Параметр – это значение.** Параметр можно сохранить с возможностью перезаписи, либо без такой возможности и использовать как глобальную переменную.

**Логика работы с Pimple (базовый пример)**

Внедряем Pimple-контейнер через конструктор всем классам с зависимостями. Теперь вместо передачи множества отдельных объектов как параметров мы везде будем передавать один контейнер с заключенными в нем объектами.

$container = **new** Container();  
  
$container[**'service'**] = **function**($c) {  
 $twService = **new** TwitterService();  
 $twService->setDB($c[**'db'**]);  
 $twService->setOauth($c[**'oauth'**]);  
 **return** $twService;  
};

**class** SomeClassThatNeedsSession {  
 **private $service**;  
 **public function** \_\_construct(Pimple $container) {  
 $this->**service** = $container[**'service'**];  
 }  
}

Подключаем библиотеку, создаем контейнер, наполняем контейнер (как массив) нужными объектами и значениями, далее по коду используем их.

**namespace** PimpleIOC {  
 **include\_once 'Pimple/Container.php'**;  
 **use** Pimple\Container;

*// Создадим контейнер для хранения объектов.  
// Сервис (объект) в контейнере зашишен от перезаписи.*$container = **new** Container();  
*// Сохраним сервис (объект) в режиме Singleton.  
// Каждый раз вернет один и тот же объект по типу Singleton.*$container[**'MysqlOrderStorage'**] = **function** ($c) {  
 **return new** MysqlOrderStorage();  
};  
*// Сохраним параметр (значение) для использования как глобальная переменная.  
// Мы не можем переписать или переопределить данное значение.*$container[**'MyOrderID'**] = **function** ($c) {  
 **return** 53;  
};  
  
*// Пример использования.*$model = **new** OrderModel($container[**'MysqlOrderStorage'**]);  
$model->getOrder($container[**'MyOrderID'**]);

**Код (см. комменты к коду)**

**namespace** PimpleIOC {  
 **include\_once 'Pimple/Container.php'**;  
 **use** Pimple\Container;  
  
 **class** OrderModel {  
 **private $container**;  
  
 *// Передадим весь контейнер как зависимость.* **public function** \_\_construct(Container $container) {  
 $this->**container** = $container;  
 }  
 **public function** getOrder($orderID) {  
 $order = $this->**container**[**'OrderStorage'**]->load($orderID);  
 $this->prepareOrder($order);  
 }  
 **private function** prepareOrder($order) {  
 **print 'Order: '** . $order[**'order\_id'**] . **' customer: '** . $order[**'customer'**] . **'<br/>'**;  
 }  
 }  
  
 *// НИЖЕ ТА ЖЕ ЛОГИКА ЧТО И В ФАБРИКЕ.  
 // Гарантируем всем типам хранилищ поддержку метода ::load, принимающий id заказа.  
 // Логика неизменна, как и в примере с Factory Method.* **interface** OrderStorage {  
 **public function** load($orderID);  
 }  
  
 **class** MysqlOrderStorage **implements** OrderStorage {  
 **public function** load($orderID) {  
 **return array**(  
 **'order\_id'** => $orderID,  
 **'customer'** => **'Rost'**,  
 );  
 }  
 }  
  
 **class** OracleOrderStorage **implements** OrderStorage {  
 **public function** load($orderID) {  
 **return array**(  
 **'order\_id'** => $orderID,  
 **'customer'** => **'Rost'**,  
 );  
 }  
 }  
  
 *// Создадим контейнер для хранения объектов.  
 // Сервис (объект) в контейнере зашишен от перезаписи.* $container = **new** Container();  
 *// Сохраним сервис (объект) в режиме Singleton.  
 // Каждый раз вернет один и тот же объект по типу Singleton.* $container[**'OrderStorage'**] = **function** ($c) {  
 **return new** MysqlOrderStorage();  
 };  
 *// А вот так с помощью Container::extend() можно добавить функционала объекту  
 // после его создания.* $container->extend(**'OrderStorage'**, **function** ($storage, $c) {  
 **print 'Extended MysqlOrderStorage: '**;  
 **return** $storage;  
 });  
 *// А вот так каждый раз будет создавать новый объект.* $container[**'NewOrderStorage'**] = $container->factory(**function** ($c) {  
 **return new** OracleOrderStorage();  
 });  
 *// Сохраним параметр (значение). Данный параметр мы можем случайно переписать.* $container[**'MyOrderID'**] = 52;  
 *// Сохраним параметр (значение) для использования как глобальную переменную.  
 // Мы не можем переписать или переопределить данное значение.* $container[**'MyOrderID'**] = **function** ($c) {  
 **return** 53;  
 };  
 *// С помощью метода Container::protect() можно сохранить значение с типом callable.* $container[**'custom\_callback\_function'**] = $container->protect(**function** () {  
 *// Здесь какой-нибудь кастомный код.* **return** *rand*();  
 });  
  
 *// Пример использования.* $model = **new** OrderModel($container);  
 $model->getOrder($container[**'MyOrderID'**]);  
}

## DI в Drupal 7

<http://kostya.in.ua/content/patterny-proektirovaniya-v-drupal> - см DI в Drupal

**Принцип DI**

Любой алгоритм не должен лезть вовне за требуемой информацией/параметром. Данная информация должна быть ему передана и внедрена в алгоритм.

**Пример алгоритма с отсутствием DI**

Функция ниже зависит от внешних параметров и поэтому не может быть использована как часть библиотечного кода в другом приложении.

Также ее невозможно протестировать.

function example\_page() {

if (arg(0) == 'node' && is\_numeric(arg(1)) {

$node = node\_load(arg(1));

if (arg(2) == 'example') {

// Do useful stuff here.

}

}

}

1. Функцию example\_page() нельзя повторно использовать для других страниц, поскольку она зависит от внешней переменной $\_GET (т.е. она лезет во внешний контекст)
2. Нельзя использовать юнит-тесты, поскольку опять же данная функция зависит от внешнего контекста.

**Пример примитивной DI**

function example\_menu() {

$items['node/%node/example'] = array(

'page callback' => 'example\_page',

'page arguments' => array(1),

// ...

);

return $items;

}

function example\_page($node) {

print 'Page';

}

1. Функцию example\_page() можно повторно использовать, поскольку зависимость передается в нее в виде аргумента.
2. Можно использовать юнит-тесты, передавая произвольные аргументы в функцию.

# Software Architecture

<https://msdn.microsoft.com/en-us/library/ee658093.aspx> **- software architecture and design – очень толковая статья, определения, принципы и подходы. По хорошему, требует хотя бы пары дней на осмысление.**

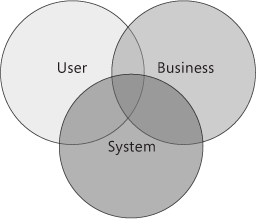
## Введение

Архитектура ПО – это высокоуровневая структура программной системы.

Для описания архитектуры существуют специализированные языки, однако все чаще используется UML.

Архитектура – это стратегия. Включает в себя выбор

* Парадигмы программирования
* Стандартов разработки ПО
* Принципы проектирования
* Основные платформы и компоненты
* Описание высокоуровневых требований



Не архитектура – это тактика. Включает в себя

* Локальное применение шаблонов проектирования
* Рефакторинг
* Выбор конкретных модулей программы или фреймворка
* И т.д.

## Архитектура веб-приложения

Веб-приложение – это клиент-серверное приложение, клиентом выступает браузер, а сервером – веб-сервер, обмен информацией происходит по сети.

**Достоинства**

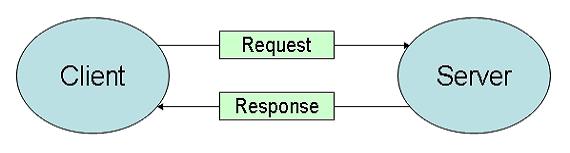
1. Кроссплатформенность, поскольку работа приложения не зависит от ОС клиента.
2. Отсутствие дублирования кода (копий программы) между клиентами.
3. Защищенность данных на сервере.
4. Широкий доступ.

**Недостатки**

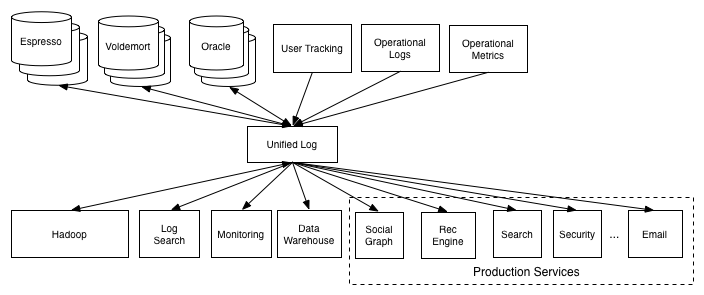
1. Зависимость от работоспособности сети.
2. Медленная работа из-за задержки при передаче данных по сети.
3. Неработоспособность сервера влияет на работу всех клиентов.
4. Высокая стоимость оборудования и обслуживания сервера.
5. Широкий доступ.

**Уровни архитектуры веб-приложения**

1. В классике, архитектура двухуровневая (клиент - сервер)



1. Однако архитектура может быть многоуровневая (клиент - сервера приложений – сервера баз данных – сервера поиска и т.д.)

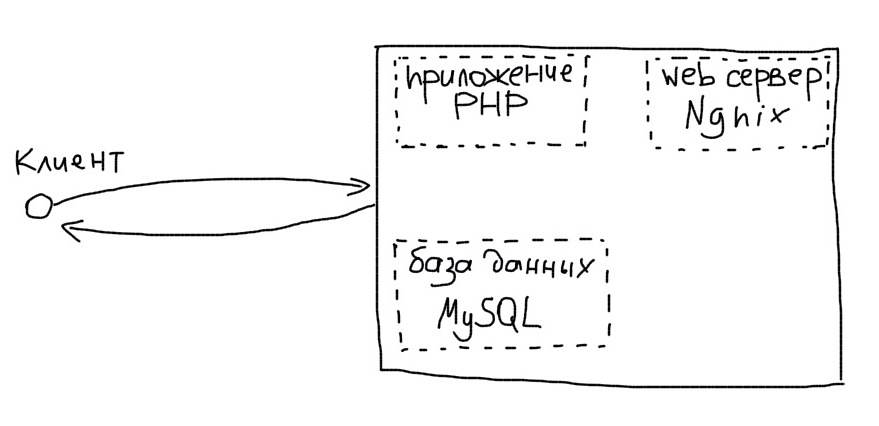


## Highload: архитектура веб-приложения от простой к сложной

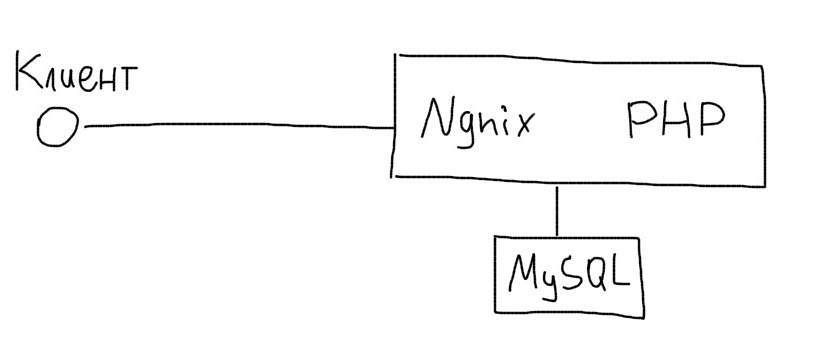
<http://ruhighload.com/post/%D0%90%D1%80%D1%85%D0%B8%D1%82%D0%B5%D0%BA%D1%82%D1%83%D1%80%D0%B0+%D0%B2%D1%8B%D1%81%D0%BE%D0%BA%D0%B8%D1%85+%D0%BD%D0%B0%D0%B3%D1%80%D1%83%D0%B7%D0%BE%D0%BA>

### Базовая архитектура: клиент и один сервер для всего

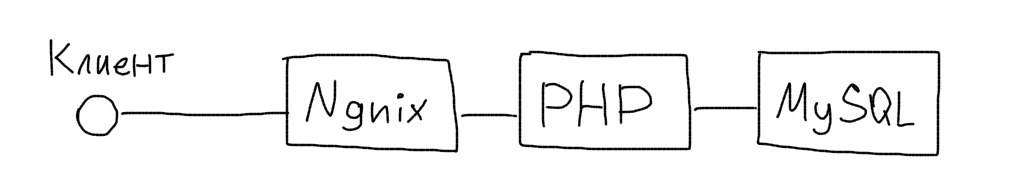
Стартуем проект



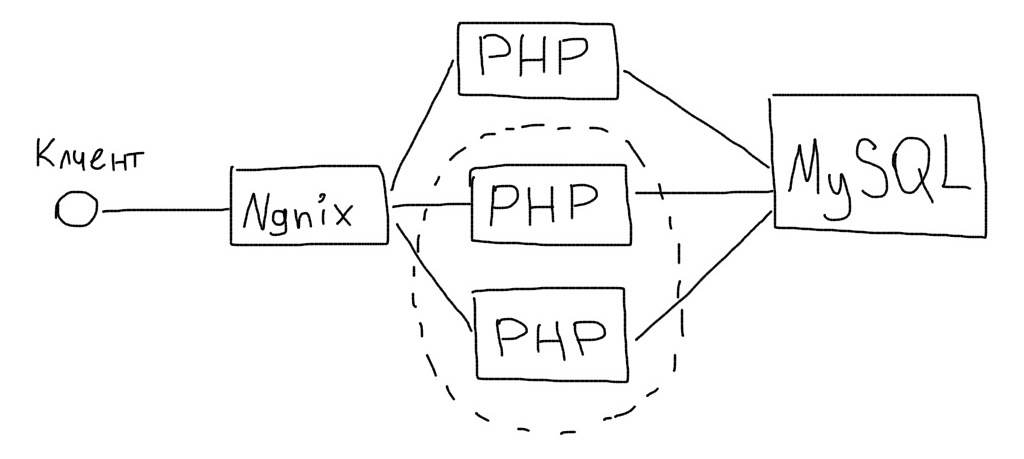
### Переносим БД на отдельный сервер



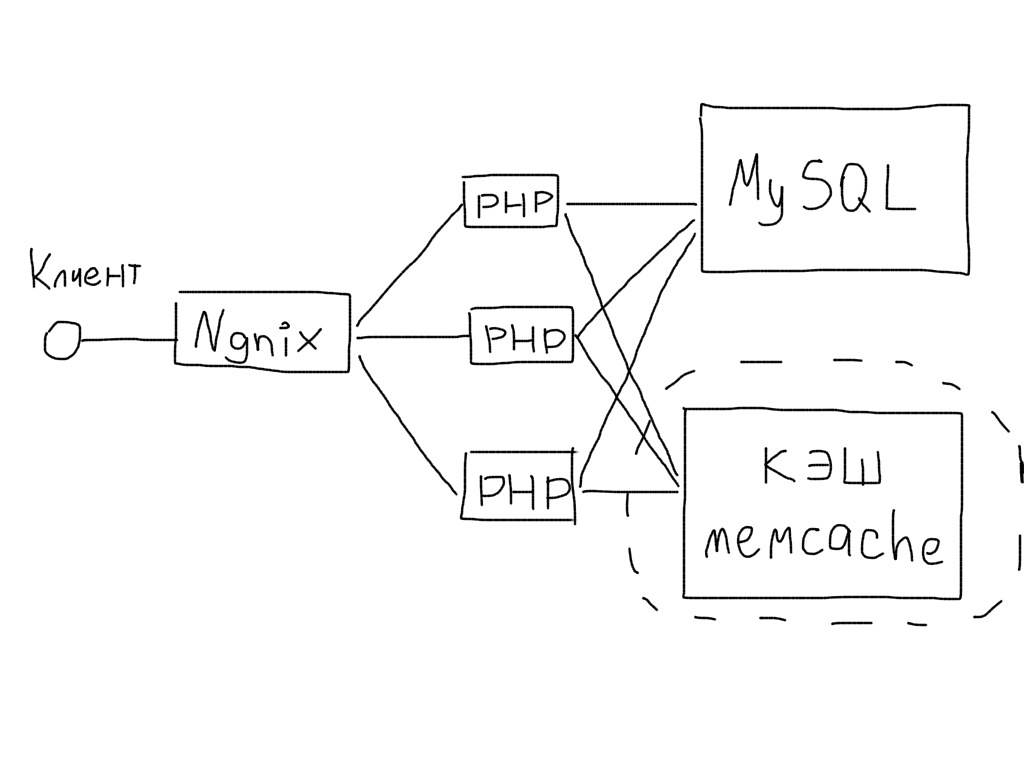
### Переносим Web-сервер на отдельный сервер



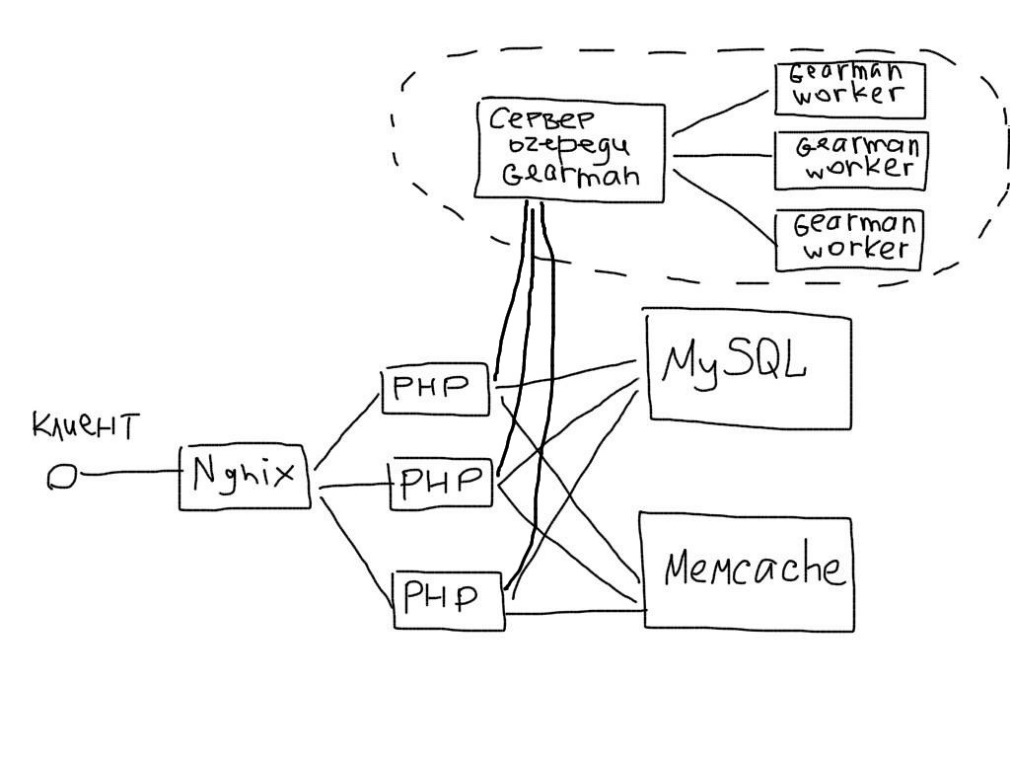
### Создаем копии PHP-приложения на нескольких серверах



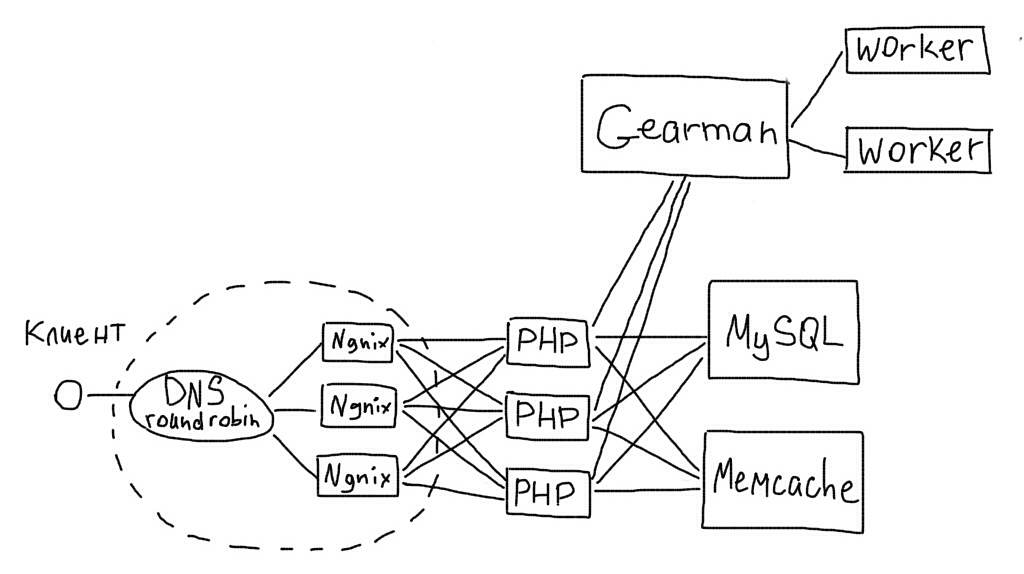
### Подключаем кеширующий сервер/сервера



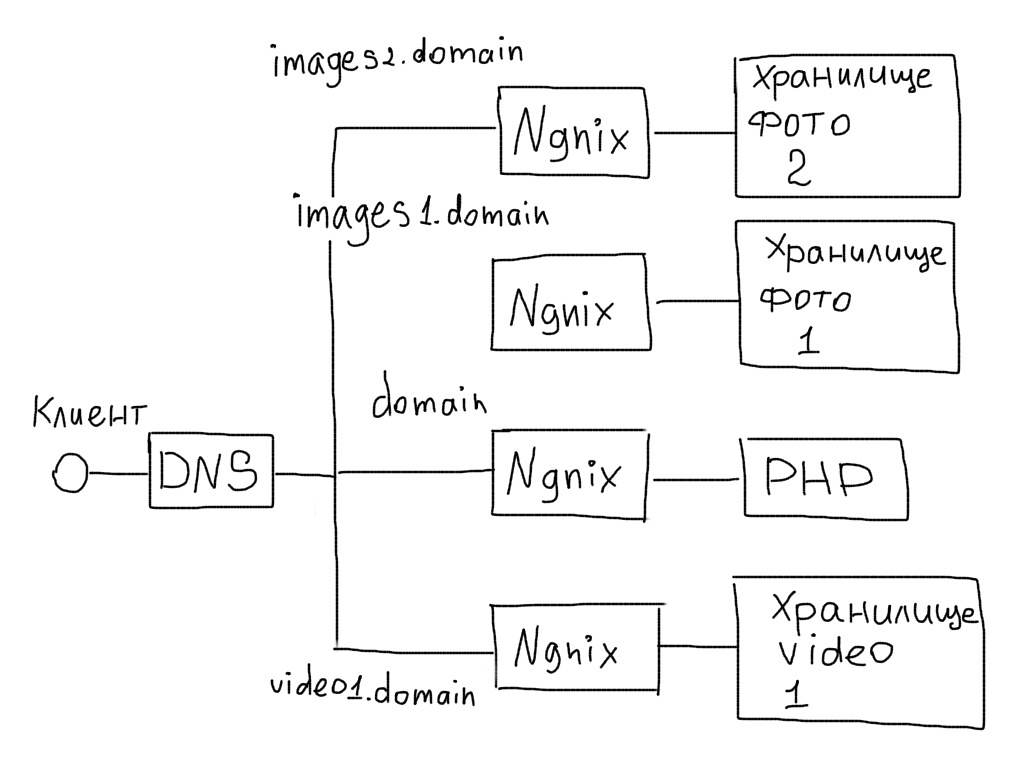
### Выносим обработку тяжелых операция в очереди



### Подключаем несколько серверов с Web-серверами и сервер с балансировщиком DNS между ними



### Выносим файлы на отдельного сервера (CDN)



## MVC паттерн

[**http://habrahabr.ru/post/150267/**](http://habrahabr.ru/post/150267/) **- MVC на примере создания сайта-визитки**

<http://developer.co.ua/posts/view/glava_2_issleduem_kod_symfony> - MVC на примере Symphony

* <http://habrahabr.ru/post/31270/> - PHP MVC – читать обязательно
* <http://habrahabr.ru/post/150267/> - PHP MVC – читать обязательно
* <http://habrahabr.ru/qa/17158/> - PHP MVC – читать обязательно

**Model-View-Controller – схема использования шаблонов проектирования, с помощью которых данные (модель), пользовательский интерфейс (View) и взаимодействие с пользователем (Controller) разделены на 3 отдельных компонента. Таким образом, модификация одного компонента оказывает минимальное воздействие на другой компонент.**

**По данному паттерну строится архитектура большинства современных веб-приложений.**

Model – хранение и обработка данных, бизнес-логика. В теории модель ничего не знает о контроллере. В модели, как правило, содержится основная масса кода приложения:

* Выборка данных из БД
* Обработка выбранных данных
* Реализация бизнес-логики
* Передача конечных данных контролеру для последующей передачи их во View

View – уровень представления данных, графический интерфейс. В теории View ничего не знает о контроллере:

* Общий шаблон для отображения общего каркаса страницы.
* Более детализированные шаблоны для отображения частей страницы.

Controller – тонкое передающее звено между model и view. Контроллер знает о конкретной модели и конкретной View (аналог паттерна Медиатор). Класс-Контроллер хранит в себе ссылку на нужную модель и нужную View. Контроллер может быть построен по паттернам Front Controller или Page Controller. В методе контроллера buildPage вызывается метод View для темизации страницы, который принимает данные от Модели.

* Обработка пользовательского запроса

Например, для идентичных урлов

<http://www.example.com/contacts.php?action=feedback>

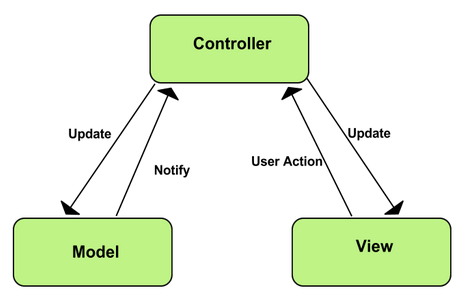
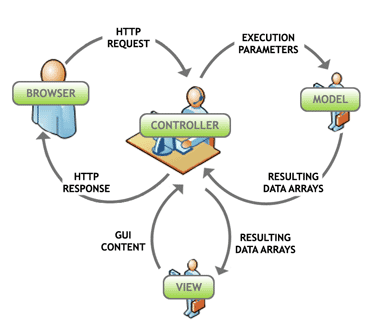
<http://www.example.com/contacts/feedback>

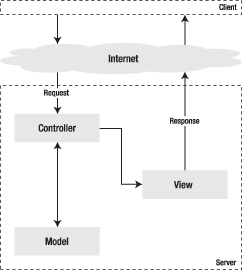
contacts.php – это контроллер, а feedback – метод данного контроллера.

* Передача обработанных параметров для выполнения моделью
* Передача конечных данных (переменных темплейта) во View для отображения (темплейт)

1. получает данные браузерного запроса,
2. обрабатывает их и передает в Model (бд),
3. получает обработанные данные от Model (из бд) в виде массива,
4. передает их во View (темплейт),
5. получает View в виде HTML-кода (пользовательский интерфейс),
6. отправляет пользовательский интерфейс в браузер.

USER -> Validate\_data() -> Prepare\_data() -> Insert\_data() ->DATABASE -> Get\_data() -> Prepare\_data() -> Theme\_data() -> TEMPLATE





### MVC в PHP

**Обязанности**

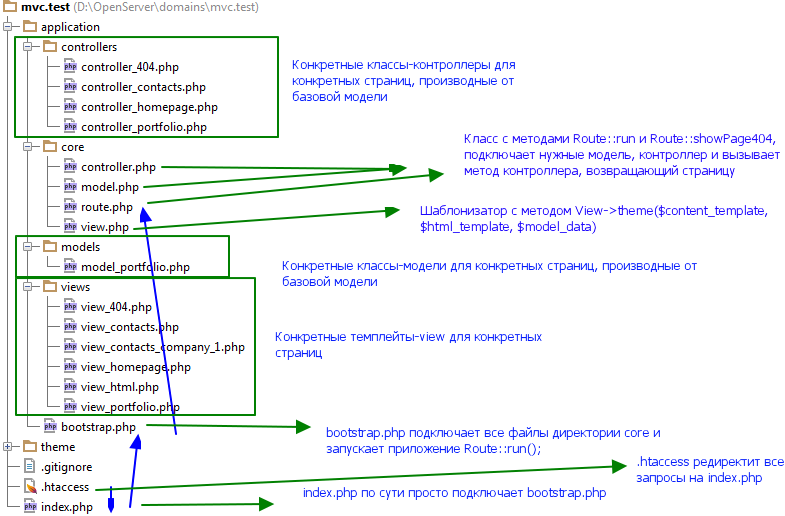
1. PHP-контроллер – класс, обрабатывающий запросы пользователей, корректирующий запросы к модели в зависимости от запроса, подготавливающий данные для передачи представлению
2. PHP-модель – это класс, работающий с БД (запись, чтение, удаление) или сервисами, реализующий бизнес-логику и возвращающий переменные для View-темплейта.
3. PHP-представление – шаблонизатор для вставки уже готовых данных

**Базовая схема**

1. Перенаправляем все запросы один скрипт index.php, получаем урл запроса ($\_SERVER[‘REQUEST\_URI’]).
2. [Page Controller] Извлекаем из урла имя класса-контроллера и метода контроллера, ответственного за генерацию страницы.
3. [Front Controller] Либо передаем весь урл единому контроллеру, который по нему ищет подходящий обработчик.
4. Загружаются файлы, содержащие классы контроллера и модели.
5. Создается экземпляр контроллера (внутри в его конструкторе создается экземпляр подключенной модели и экземпляр View) и у него вызывается метод генерации страницы.
6. Метод контроллера запрашивает данные хранящейся в классе Модели и в виде переменных передает их на темизацию подходящему шаблону View.
7. View рендерит данные пользователю в виде HTML.
8. Если запрошенного класса или метода контроллера не существует, то вызывается Controller\_404.php и отображается страница 404.

**MVC Framework**

1. .htaccess реализует перенаправление всех запросов на файл index.php
2. index.php просто подключает bootstrap.php
   1. bootstrap.php подключает классы
      1. model.php
      2. view.php
      3. controller.php
      4. route.php
   2. и запускает (старт приложения) Route::run();
      1. Внутри метода получаем запрошенный URL: $routes = explode('/', $\_SERVER['REQUEST\_URI']);
      2. разбиваем его на 2 части по слешам: первая часть – имя класса-контроллера, вторая часть – название метода данного контроллера.
      3. По имени класса-контроллера находится и подключается класс модели.
      4. По имени класса-контроллера находится и подключается класс контроллера (класс контроллера знает, какую модель вызывать и какой View подключать).
      5. Вызывается метод класса-контроллера, возвращающий страницу, например:
      6. new Controller\_Contacts->company\_1(); // возвращает страницу /contacts/company\_1
      7. Если не нашлось класса-контроллера или его метода, показывается страница 404.



### HMVC

(Hierarchical MVC)

<http://habrahabr.ru/post/212065/>

<http://habrahabr.ru/post/113117/>

### MVP

### MVVM

## UML (universal modelling language)

<https://ru.wikipedia.org/wiki/UML>

# Design Patterns

**Паттерн проектирования – это общее типовое решение некоторой проблемы. За счет правильного использования паттернов повышается коэффициент использования готовых решений**

## GOF паттерны (23 main patterns)

Epam Training Center!!! – see materials

[**http://habrahabr.ru/post/136766/**](http://habrahabr.ru/post/136766/) **- Паттерны в метафорах!**

<https://ru.wikipedia.org/wiki/Design_Patterns> - PHP все шаблоны проектирования

<http://habrahabr.ru/post/153225/> - как 2 программиста хлеб пекли

<http://habrahabr.ru/post/153845/>

<https://www.youtube.com/playlist?list=PLmqFxxywkatStbd9hdzVOS1hZa9dc56k4> – полный курс шаблонов

<http://dron.by/post/patterny-shablony-proektirovanie-v-php-vvedenie.html> - вроде как самое лучшее описание паттернов

<http://dev.by/blogs/main/myagkoe-vvedenie-v-patterny-proektirovaniya-chast-1>

<http://dev.by/blogs/main/myagkoe-vvedenie-v-patterny-proektirovaniya-chast-2> - очень классно и человеческими словами

<https://ru.wikipedia.org/wiki/Design_Patterns> - Design Patterns: Elements of Reusable Object-Oriented Software (Gang of Four)

<https://ru.wikipedia.org/wiki/%D0%A8%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F> – все известные паттерны

<http://cpp-reference.ru/patterns/creational-patterns/>

<http://faq.linemedia.ru/patternyi-pro/> - паттерны по-русски, плюсы-минусы

<http://citforum.ru/SE/project/pattern/> - Справочник паттернов (GOF, GRASP, Architecture, Enterprize)

<http://sourcemaking.com/design_patterns> - patterns in English, very good!

**23 паттерна описаны в классическом труде 1994 года - Design Patterns: Elements of Reusable Object-Oriented Software (Gang of Four).**

**Остальные (помечены как дополнительные) были открыты после.**

**Паттерны обычно используются в больших системах и описывают целые компоненты/подсистемы.**

### Порождающие паттерны (5 main patterns)

**Основная задача данной группы паттернов – создавать (порождать) объекты.**

1. Позволяют сделать систему (базовые классы) независимой от способа создания, композиции и представления объектов.
2. Используют наследование:
   1. шаблон, порождающий классы, использует наследование, чтобы изменять наследуемый класс
   2. шаблон, порождающий объекты, делегирует наследование другому объекту
3. Порождающие шаблоны инкапсулируют знания о конкретных классах, которые применяются в системе. Они скрывают детали того, как эти классы создаются и стыкуются.

#### Singleton (Синглтон, одиночка)

**Проблема**

1. **Нужно создать и гарантировать существование одного-единственного экземпляра данного класса/объекта и предотвратить/запретить создание его копий.** 
   1. Объект, хранящий настройки приложения
   2. Объект, выполняющий подключение к БД
   3. Объект, выполняющий однократную тяжелую операцию
2. Объект должен быть доступен для любого другого объекта в системе.
3. Не использовать этого глобальную переменную:
   1. поскольку ее можно случайно перезаписать
   2. потому что подрывает инкапсуляцию, класс зависит от контекста, его нельзя извлечь и использовать в другом приложении, не убедившись, что там существует такая же глобальная переменная

**Особенности реализации**

Смысл синглтона в статической переменной-свойстве объекта, в которую однократно записывается результат операции при вызове метода первый раз (например, соединение с БД).

При последующих вызовах в скрипте мы возвращаем данное свойство вместо создания объекта заново. Таким образом, нет повторного выполнения тяжелой операции.

Т.е. нельзя создать экземпляр синглтона. Мы запрещаем создание нового объекта Singleton, делая все методы по созданию приватными (если сделать публичным – теряется весь смысл)

* через new или clone – приватный конструктор
* через \_\_wakeup, \_\_unserialize – данные методы объявлены также приватными

Можно лишь пользоваться статическими методами синглтона, которые используют данное статическое свойство.

**Недостатки**

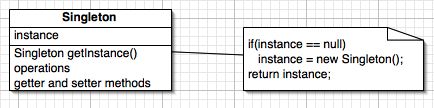
<https://www.youtube.com/watch?v=-AlFSe_K27A>

1. **У синглтона очень много недостатков.**
2. Очень много разработчиков считают синглтон антипаттерном и рекомендуют всячески избегать его. Потому что это все равно по сути глобальная переменная-объект, которую может получить любой объект в системе. И это плохо, потому что внутри black box что-то происходит и зависит от неких глобальных данных (данных, которые могут прийти извне). И поэтому мы не контролируем данные объекты полностью.
3. Нельзя использовать программу с зашитым синглтоном как подпрограмму. Т.е. оформить ее как функцию, пакет или метод и вызывать несколько раз (в цикле). Поскольку она будет зависеть от одного объекта-синглтона, имеющего всегда одни и те же настройки.
4. Очень сложно реализовать наследование синглтона.
5. Синглтон нарушает SRP (Single Responsibility Principle) — класс синглтона, помимо того, чтобы выполнять свои непосредственные обязанности, занимается еще и контролированием количества своих экземпляров.
6. Синглон повышает связность, поскольку не может быть передан как объект (нельзя создать), а используется напрямую.

**Альтернатива синглтону**

1. Просто создать объект (например, хранящий конфигурацию).
2. И передавать его как параметр только тем классам, которым он нужен.
3. А кому не нужен – не передавать.
4. Таким образом, система перестает зависеть от объекта внутри.

**Пример**



**namespace** SingltonPattern {  
 **class** DB {  
 **public static** *$db*;  
  
 *// Запретим создание объекта всеми методами.* **private function** \_\_clone() {}  
 *// Запретим создание объекта всеми методами.* **private function** \_\_wakeup() {}  
 *// Приватный конструктор запрещает создание через new.* **private function** \_\_construct() {  
 $this->**connect** = *mysql\_connect*(**'127.0.0.1'**, **'root'**, **''**);  
 *mysql\_select\_db*(**'mysql'**);  
 }  
  
 **public static function** connect() {  
 **if** (**empty**(**self**::*$db*)) {  
 *// Если первый раз - то вызовется конструктор  
 // и результат запишется в статическую переменную.  
 // Последующие вызовы будем использовать записанный результат  
 // из переменной вместо дорогих операций.* **self**::*$db* = **new self**;  
 }  
 **return self**::*$db*;  
 }  
 *// Статический метод, использующий соединение с БД.* **public static function** query($sql) {  
 $connect = **self**::*connect*();  
 **if** (**isset**($connect->**connect**)) {  
 **return** *mysql\_query*($sql);  
 }  
 **else** {  
 **die**(**'No connection to db server'**);  
 }  
 }  
 }  
}  
  
**namespace** {  
 **use** SingltonPattern\DB;  
  
 *// Здесь в синглтоне сохранено соединение с БД.  
 // Таким образом разные запросы работают с одним и тем же соединением.* $db = DB::*connect*();  
 $result = $db->query(**'SHOW TABLES'**);  
 *var\_dump*(*mysql\_fetch\_array*($result));  
}

#### Factory Method (Virtual Constructor, Фабричный метод)

Класс создает объекты.

**Проблема**

1. **Создание параллельных иерархий наследований классов, связанных общим, открытым к изменениям базовым классом, независимым как от процесса создания и от типов создаваемых объектов**
2. **Реализация IoC (инверсии управления)**
3. Надо создавать объект, но базовому классу неизвестен тип объекта. Таким образом, базовый класс делегирует создание объектов производным классам.

**Особенности реализации**

Базовая фабрика (абстрактный класс) -> конкретная фабрика – класс с методом для генерации объектов-продуктов

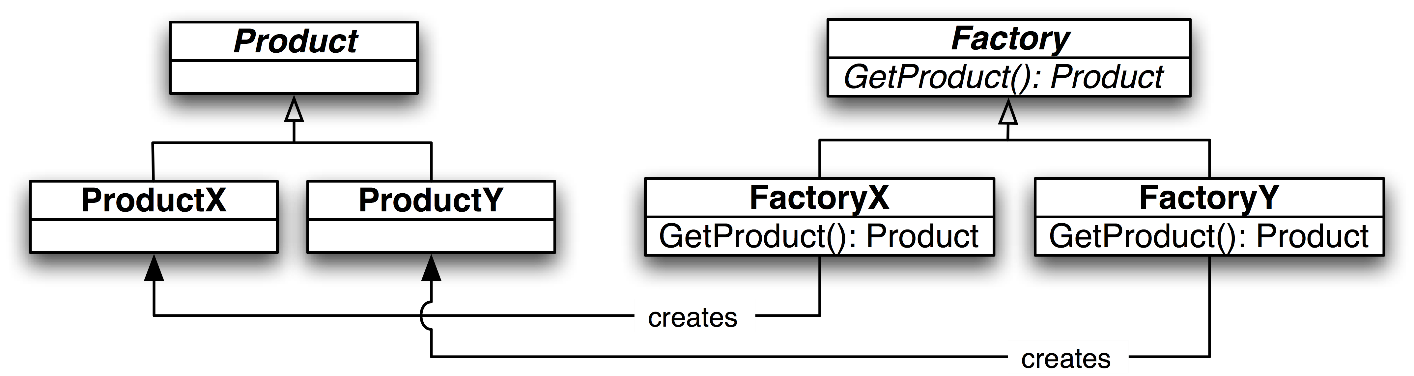
Базовый продукт (абстрактный класс) –> конкретный продукт, генерируемый конкретной фабрикой

Классы-создатели отделены от объектов, которые они будут генерировать

**Недостатки паттерна**

Необходимость создавать наследника для каждого нового типа продукта – как результат, избыточное количество подклассов.

**Пример**



**namespace** FactoryPattern {  
 **interface** Factory {  
 **public function** getProduct();  
 }  
  
 **interface** Water {  
 **public function** GetName();  
 }  
  
 **class** ColaFactory **implements** Factory {  
 **public function** getProduct() {  
 **return new** ColaWater();  
 }  
 }  
  
 **class** ColaWater **implements** Water {  
 **public function** GetName() {  
 **return 'ColaWater<br/>'**;  
 }  
 }  
  
 **class** PepsiFactory **implements** Factory {  
 **public function** getProduct() {  
 **return new** PepsiWater();  
 }  
 }  
  
 **class** PepsiWater **implements** Water {  
 **public function** GetName() {  
 **return 'PepsiWater<br/>'**;  
 }  
 }  
}  
  
**namespace** {  
 **use** FactoryPattern\ColaFactory, FactoryPattern\PepsiFactory;  
  
 *// Использование: на каждый новый тип продукта своя фабрика.  
 // В результате базовая фабрика и базовый продукт отделены от конкретных типов  
 // фабрик и продуктов. В итоге они открыты к изменениям.* $cola\_factory = **new** ColaFactory;  
 $cola = $cola\_factory->getProduct();  
 **print** $cola->GetName();  
  
 $pepsi\_factory = **new** PepsiFactory;  
 $pepsi = $pepsi\_factory->getProduct();  
 **print** $pepsi->GetName();  
}

#### Abstract Factory (Абстрактная Фабрика)

<https://www.youtube.com/watch?v=zQMhKouW-VU>

Интерфейс для создания фабриками систем совместимых объектов.

Дальнейшее развитие идеи паттерна Factory. **Разнотипные фабрики внутри клиента создают различные типы продукты (Factory Method).** Разнотипные продукты взаимодействуют между собой, создавая некий комбинированный продукт, состоящий из взаимосвязанных продуктов.

**Проблема**

1. **Создание клиентом родственных семейств объектов, объединенных базовыми классами. Скрытие от клиента работы с объектами семейств.**
2. Упрощение использования родственных семейств объектов. Есть объект-клиент, который может принимать фабрики различных типов и возвращать конкретные продукты. Система должна быть отделена от деталей реализации
3. Простота работы для программиста – из-за изолирования конкретных классов и скрытия логики фабрик
4. Организация взаимодействие между двумя продуктами одной фабрики

**Недостатки паттерна**

Трудоемкое добавление поддержки нового вида продукта. Поскольку для этого необходимо создать 3 новых класса: фабрика, банка, напиток.

**Особенности реализации**

Это объект Client, который принимает в метод A фабрики разных типов. Фабрики создают товары. В методе B товары разных фабрик взаимодействуют между собой.

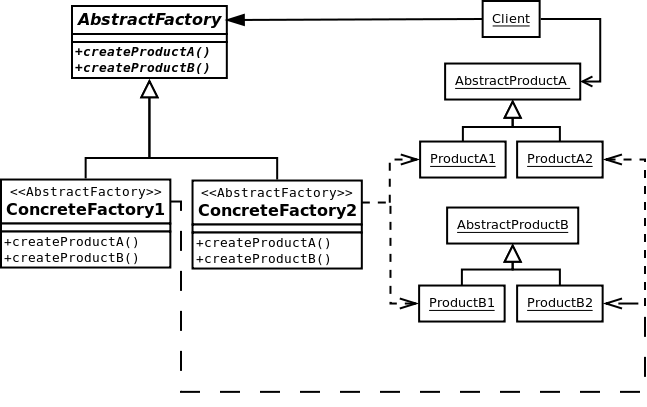
1. Есть базовые абстрактные классы
   1. abstract Factory, который содержит базовые абстрактные характеристики (интерфейс) фабрики
      1. abstract createWater() -> new Water
      2. abstract createBottle() -> new Bottle
   2. abstract Water, интерфейс напитка
   3. abstract Bottle, интерфейс банки
      1. Bottle->getProduct(Water) – разливает воду в банку
   4. **abstract Client(Factory), интерфейс потребителя.** Именно с данным и только с данным объектом будет работать сторонний программист. Задача объекта –
      1. принять любую фабрику в конструктор и там же получить ее продукты, записав в свои свойства
      2. в методе getProduct() совместить ее продукты/свои свойства, выдав итоговый напиток в банке.

Т.е. объекту Client без разницы, какую фабрику получить, он просто в итоге выдает ее продукт.

1. Есть фабрика CocaColaFactory extends Factory, производящая напиток CocaColaWater и разливающий его в банки CocaColaBottle
   1. напиток CocaColaWater extends Water
   2. банка CocaColaBottle extends Bottle
   3. Напиток CocaColaWater и банка CocaColaBottle – семейство продуктов фабрики CocaColaFactory.
2. У фабрики есть станок, создающий напиток -> метод createWater() -> new CocaColaWater
3. У фабрики есть станок, штампующий банки -> метод createBottle() -> new CocaColaBottle
4. Аналогичная фабрика PepsiFactory производит напиток PepsiWater и разливает его в банки PepsiBottle
5. Напиток PepsiWater и банка PepsiBottle – семейство продуктов фабрики PepsiFactory.

! Нелогично разливать CocaColaWater в PepsiBottle – это антипаттерн, поскольку здесь смешиваются различные семейства объектов.

**Пример**



**namespace** AbstractFactoryPatern {  
 **class** Client {  
 **private $water**;  
 **private $bottle**;  
 **public function** \_\_construct(Factory $factory) {  
 $this->**water** = $factory->createWater();  
 $this->**bottle** = $factory->createBottle();  
 }  
 *// абстрагирование вариантов использования.* **public function** getProduct() {  
 **return** $this->**bottle**->waterIntoBottle($this->**water**);  
 }  
 }  
  
 **abstract class** Factory {  
 *// фабричный метод.* **function** createWater() {}  
 *// фабричный метод.* **function** createBottle() {}  
 }  
  
 **abstract class** Water {  
 **public $water\_name** = **'Basic Water'**;  
 }  
  
 **abstract class** Bottle {  
 **public $bottle\_name** = **'Basic Bottle'**;  
 *// взаимодействие двух объектов.* **function** waterIntoBottle(Water $water) {  
 **return** $water->**water\_name** . **' into '** . $this->**bottle\_name**;  
 }  
 }  
  
 **class** ColaFactory **extends** Factory {  
 **function** createWater() {  
 **return new** ColaWater();  
 }  
 **function** createBottle() {  
 **return new** ColaBottle();  
 }  
 }  
  
 **class** ColaWater **extends** Water {  
 **public $water\_name** = **'Cola Water'**;  
 }  
  
 **class** ColaBottle **extends** Bottle {  
 **public $bottle\_name** = **'Cola Bottle'**;  
 }  
}  
  
**namespace** {  
 *// Использование.* $client = **new** AbstractFactoryPatern\Client(**new** AbstractFactoryPatern\ColaFactory);  
 $product = $client->getProduct();  
 **print** $product;  
}

#### Prototype (Прототип)

<https://www.youtube.com/watch?v=hPpe6WrB8Cg>

**Проблема**

Система не должна зависеть от того, как в ней создаются, компонуются и представляются объекты.

**Используется для**

1. Для создания разных видов тяжелых объектов из базовых объектов (прототипов) путем клонирования, вместо создания через new (слишком дорого создавать все заново). А при клонировании мы получаем копию нужного объекта просто и быстро.
2. Позволяет избежать создания нового тяжелого объекта через new.
3. Позволяет избежать параллельных иерархий классов продуктов.

**Особенности реализации**

Основан не на наследовании, а на композиции.

Вызывая clone, мы делаем копирование свойств объекта (shallow copy), за исключением свойств, являющихся ссылками на другие переменные. Они так и останутся ссылками. Если необходимо произвести более глубокое копирование, то реализуем в классах свойство \_\_clone

Shallow Copy – это создание нового объекта через неполное копирование свойств старого объекта (свойства-ссылки останутся свойствами-ссылками, кроме того, идентификатор объекта станет другим).

После завершения копирования через clone, если у метода определен метод \_\_clone(), то в нем можно дополнительно изменить свойства объекта.

1. Создадим 3 семейства подклассов с общим интерфейсом Water

Water -> Coke

* Beer

1. Создадим фабрику, которая будет принимать однажды созданные подклассы, клонировать их и возвращать новые подклассы – копии полученных.

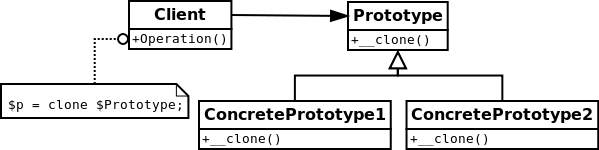
WaterFactory(new Coke, new Beer) -> getCoke() -> new Coke

* + - * -> getBeer() -> new Beer

**Недостатки паттерна**

1. Каждый подкласс прототипа должен реализовать операцию clone и результаты этой операции должны быть предсказуемы.

**Пример**



**namespace** PrototypePattern {  
 **abstract class** Water {}  
  
 **abstract class** Beer **extends** Water {}  
 **class** Budweiser **extends** Beer {}  
 **class** Corona **extends** Beer {}  
  
 **abstract class** Wine **extends** Water {}  
 **class** Martini **extends** Wine {}  
 **class** Champagne **extends** Wine {}  
  
 *// Объект-фабрика, получает уже созданные объекты, клонирует их  
 // и возвращает их копии.* **class** AlcoFactory {  
 **private $beer**;  
 **private $wine**;  
  
 *// Получили кем-то созданные объекты.* **public function** \_\_construct(Beer $beer, Wine $wine) {  
 $this->**beer** = $beer;  
 $this->**wine** = $wine;  
 }  
 *// Сделали копию и вернули ее.* **function** getBeer( ) {  
 **return clone** $this->**beer**;  
 }  
 **function** getWine( ) {  
 **return clone** $this->**wine**;  
 }  
 }  
}  
  
**namespace** {  
 **use** PrototypePattern\AlcoFactory,  
 PrototypePattern\Budweiser,  
 PrototypePattern\Martini;  
  
 *// Использование.* $alcoFactory = **new** AlcoFactory(**new** Budweiser, **new** Martini);  
 $beer = $alcoFactory->getBeer();  
 $wine = $alcoFactory->getWine();  
  
 *var\_dump*($beer);  
 *var\_dump*($wine);  
}

#### Builder (Строитель)

<https://www.youtube.com/watch?v=5YLqUZvpDDA>

Чем отличается от абстрактной фабрики? Тем, что фокусируется именно на процессе постройки объектов с разными свойствами с помощью единого алгоритма.

**Проблема**

Паттерн позволяет создать единый алгоритм конструирования множества объектов одного типа, но обладающих разными свойствами.

**Используется для**

1. Контролирует процесс создания сложного объекта/продукта
   1. Создает пустой продукт
   2. Наполняет его свойствами, поочередно вызывая методы продукта
   3. Возвращает продукт
2. Алгоритм создания сложного объекта не должен зависеть от составляющих частей объекта и от особенностей их стыковки.

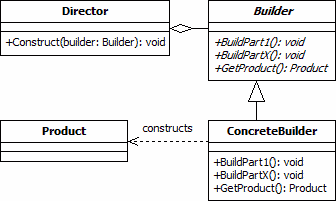
**Особенности реализации**

1. Есть клиент Client, который принимает конкретных строителей Santechnik и Plitochnik (тип Builder).
2. Оба делают продукты Rabota, выполняя один набор операций-методов
   1. –>start() new Rabota()
   2. ->podgotovka($instrument)
   3. ->ustanovka($material)
   4. ->rabotaGotova() return $this->rabota
3. Но входные параметры в эти операции разные
   1. Santechnik
      1. $instrument – ‘otvertka’
      2. $material – ‘unitaz’
   2. Plitochnik
      1. $instrument – ‘molotok’
      2. $material – ‘plitka’
4. Поэтому на выходе объекты одного типа Rabota имеют разные свойства.

**Недостатки паттерна**

1. Класс ConcreteBuilder и создаваемый им продукт жестко связаны между собой (Rabota строителя Santechnik отличается от Rabota строителя Plitochnik). При внесении изменения в продукт придется менять класс ConcreteBuilder. Т.е. скажем так, специализация строителя связана с возвращаемым продуктом.

**Пример**



**namespace** BuilderPattern {  
 */\*\*  
 \* Client - управляющий класс, запускающий строительство.  
 \* Принимает строителя определенной специализации.  
 \* Создает пустой объект продукта.  
 \* Строит объект, наполняя его свойствами.  
 \* Возвращает объект.  
 \*/* **class** Client {  
 **private $builder**;  
  
 *// Приглашаем Строителя.* **public function** setBuilder(Builder $builder) {  
 $this->**builder** = $builder;  
 }  
 *// Разрешаем ему начинать процессс работы.* **public function** launchRabota() {  
 $this->**builder**->start();  
 $this->**builder**->podgotovka();  
 $this->**builder**->ustanovka();  
 }  
 *// Просим его показать работу.* **public function** getRabota() {  
 **return** $this->**builder**->getRabota();  
 }  
 }  
  
 */\*\*  
 \* Builder - абстрактный строитель.  
 \* Стандартный работяга со стандартными методами работы.  
 \*/* **abstract class** Builder {  
 **protected $rabota**;  
  
 **public function** start() {  
 $this->**rabota** = **new** Rabota();  
 }  
 **public function** getRabota() {  
 **return** $this->**rabota**;  
 }  
 **abstract public function** podgotovka();  
 **abstract public function** ustanovka();  
 }  
  
 */\*\*  
 \* Builder - конкретный строитель 1.  
 \* Строит/наполняет полученный пустой объект свойствами.  
 \*/* **class** BuilderSantechnik **extends** Builder {  
 **public function** podgotovka() {  
 $this->**rabota**->setInstrument(**'vantuz'**);  
 }  
 **public function** ustanovka() {  
 $this->**rabota**->setMaterial(**'unitaz'**);  
 }  
 }  
  
 */\*\*  
 \* Builder - конкретный строитель 2.  
 \* Строит/наполняет полученный пустой объект свойствами.  
 \*/* **class** BuilderPlitochnik **extends** Builder {  
 **public function** podgotovka() {  
 $this->**rabota**->setInstrument(**'molotok'**);  
 }  
 **public function** ustanovka() {  
 $this->**rabota**->setMaterial(**'plitka'**);  
 }  
 }  
  
 */\*\*  
 \* Rabota - продукт, который будет возвращать Строитель.  
 \* Для создания продукта необходимо поочередно вызвать его методы.  
 \*/* **class** Rabota {  
 **private $instrument** = **''**;  
 **private $material** = **''**;  
  
 **public function** setInstrument($instrument) {  
 $this->**instrument** = $instrument;  
 }  
 **public function** setMaterial($material) {  
 $this->**material** = $material;  
 }  
 **public function** \_\_toString() {  
 **return** $this->**instrument** . **' : '** . $this->**material** . **'<br />'**;  
 }  
 }  
}  
  
**namespace** {  
 **use** BuilderPattern\Client,  
 BuilderPattern\BuilderSantechnik,  
 BuilderPattern\BuilderPlitochnik;  
  
 *// Инициализация клиента.* $client = **new** Client;  
  
 *// Подготовка (наполнение свойствами).* $client->setBuilder(**new** BuilderSantechnik);  
 *// Постройка продукта.* $client->launchRabota();  
 *// Получение результата.* **print** $client->getRabota();  
  
 $client->setBuilder(**new** BuilderPlitochnik);  
 $client->launchRabota();  
 **print** $client->getRabota();  
}

#### (доп.) Lazy initialization (Ленивая инициализация)

#### (доп.) Multiton (Пул одиночек)

#### (доп.) Object pool (Объектный пул)

#### (доп.) Resource acquisition is initialization (RAII, Получение ресурса есть инициализация)

### Структурные паттерны (7 main patterns)

<http://ru.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%BD%D1%8B%D0%B5_%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD%D1%8B_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F>

**Основная задача данной группы паттернов – создавать из классов и объектов более крупные структуры.**

#### Adapter (Wrapper, Оболочка)

**Проблема**

Есть класс/объект с подходящими по функционалу методами, но неподходящим интерфейсом (методы по-другому названы, принимают иные параметры или вообще отсутствуют). Задача – создать обертку вокруг данного класса и привести внутри нее класс к нужному интерфейсу (в оболочке создать нужные методы, вызывая внутри них методы базового класса). Использовать класс обертки с инкапсулированным внутри исходным объектом.

**Используется для**

**Для преобразования объекта с одним интерфейсом в объект с другим интерфейсом. Таким образом, достигается повторное использование кода.**

1. Создание класса-клиента для работы с БД разных типов. При этом к данным БД уже написаны классы разными авторами. Задача привести работу с ними к одному интерфейсу.
2. Создание класса-клиента для работы

**Особенности реализации**

**Один адаптер на один неправильный класс. Если один адаптер на много неправильных классов, то это уже паттерн фасад.**

Допустим, нашему клиенту A необходим объект B. У нас есть объект, похожий на объект B, который мог бы в целом подойти, однако с несколько он имеет интерфейс, несовместимый с требуемым интерфейсом. Посему мы оборачиваем объект B в объект-оболочку C, которая и приводит объект к нужному интерфейсу (создает нужный метод и вызывает внутри него любой подходящий метод неправильного объекта). И передаем в объект A уже завернутый объект C с объектом B внутри.

ClientDB (Adapter->query) MySQLAdapter implements DBAdapter (->query) MySQLConnect (->request)

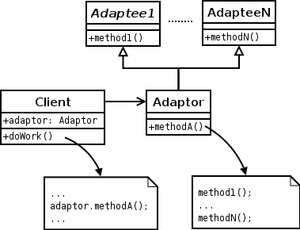
OracleAdapter implements DBAdapter (->query) OracleConnect (->ask)

SybaseAdapter implement DBAdapter (->query) SybaseConnect (->fetch)

**Недостатки**

Адаптация объекта к нужному интерфейсу может оказаться сложной задачей, если требуемые методы не имеют аналогов в адаптируемом классе/объекте.

**Пример**



**namespace** AdapterPattern {  
 *// Клиент, ожидающий продукт с конкретным интерфейсом.* **class** Client {  
 **private $product** = **NULL**;  
 **public function** \_\_construct(CorrectProductInterface $product) {  
 $this->**product** = $product;  
 }  
 **public function** useProduct() {  
 **print** $this->**product**->query();  
 }  
 }  
  
 *// Целевой интерфейс, клиент умеет работать только с ним.* **interface** CorrectProductInterface {  
 **public function** query();  
 }  
  
 *// Стандартный правильный продукт, с которым работает целевой интерфейс.* **class** CorrectProduct **implements** CorrectProductInterface {  
 **public function** query() {  
 **return 'Product<br/>'**;  
 }  
 }  
  
 *// Адаптируемый интерфейс. Клиент с ним не умеет работать, но очень хочет.  
 // Обладает методом ::request, который идентичен требуемому методу ::query.* **interface** IncorrectProductInterface {  
 **public function** request();  
 }  
  
 *// Класс, реализующий адаптирумым интерфейс.* **class** IncorrectProduct **implements** IncorrectProductInterface {  
 **public function** request() {  
 **return 'Product<br/>'**;  
 }  
 }  
  
 **class** ProductAdapter **implements** CorrectProductInterface {  
 *// Здесь храним адаптируемый класс.* **private $adaptee** = **NULL**;  
  
 *// Получили и сохранили адаптируемый класс.* **public function** \_\_construct($incorrectProduct) {  
 $this->**adaptee** = $incorrectProduct;  
 }  
 *// Создали требуемый клиентским интерфейсом метод.  
 // Вызвали внутри него любой подходящий метод полученного объекта.* **public function** query() {  
 **return** $this->**adaptee**->request();  
 }  
 }  
}  
  
**namespace** {  
 **use** AdapterPattern\Client;  
 **use** AdapterPattern\CorrectProduct;  
 **use** AdapterPattern\IncorrectProduct;  
 **use** AdapterPattern\ProductAdapter;  
  
 *// Использовали правильный объект.* $clientA = **new** Client(**new** CorrectProduct());  
 $clientA->useProduct();  
  
 *// Использовали через оболочку неправильный, но очень нужный объект.* $clientB = **new** Client(**new** ProductAdapter(**new** IncorrectProduct()));  
 $clientB->useproduct();  
}

#### Facade (Фасад)

Очень похож на адаптер!

Отличие в том, что адаптер приводит к имеющемуся интерфейсу, а фасад создает новый упрощенный интерфейс, к которому затем приводит.

Цель фасада – **создание более простого интерфейса** и обертывание **множества** классов. Упрощение использования и приведение к единому интерфейсу множество похожих по функционалу объектов.

Цель адаптера – **адаптация одного класса к уже имеющемуся** и используемому интерфейсу.

**Проблема**

Получить упрощенный единый интерфейс (малое количество методов) к схожим объектам с богатым функционалом (большое количество методов).

**Используется для**

1. Создание единой простой точки входа в сложную систему/подсистему.
2. Упрощения работы с множественными интерфейсами за счет создания прослойки, приводящей множественные интерфейсы к единому и к тому же упрощенному. Это позволяет сократить время на изучение функциональности класса.

**Особенности реализации**

Computer startComputer() Memory->load()

CPU->start()

HD->read()

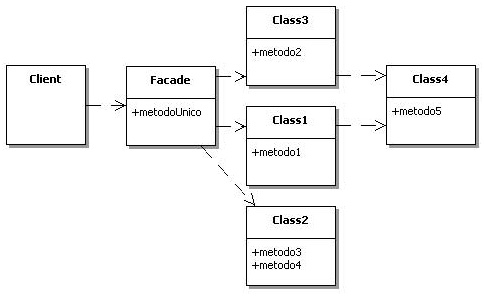
stopComputer() Memory->shutdown()

CPU->stop()

**Недостатки**

Ограничивает имеющиеся возможности объектов.

**Пример**



**namespace** FacadePattern {  
 *// Сложная часть системы 1.* **class** CPU {  
 **public function** start() {  
 **print 'CPU started<br/>'**;  
 }  
 **public function** stop() {  
 **print 'CPU stopped<br/>'**;  
 }  
 }  
  
 *// Сложная часть системы 2.* **class** Memory {  
 **public function** load() {  
 **print 'Memory loaded<br/>'**;  
 }  
 **public function** shutdown() {  
 **print 'Memory shutdowned<br/>'**;  
 }  
 }  
  
 *// Сложная часть системы 3.* **class** HardDrive {  
 **public function** read() {  
 **print 'HD started reading<br/>'**;  
 }  
 }  
  
 *// Фасад - единый простой интерфейс для работы с частями.* **class** Computer {  
 **protected $cpu**;  
 **protected $memory**;  
 **protected $hardDrive**;  
  
 **public function** \_\_construct() {  
 $this->**cpu** = **new** CPU();  
 $this->**memory** = **new** Memory();  
 $this->**hardDrive** = **new** HardDrive();  
 }  
 **public function** startComputer() {  
 $this->**cpu**->start();  
 $this->**hardDrive**->read();  
 $this->**memory**->load();  
 }  
 **public function** stopComputer() {  
 $this->**memory**->shutdown();  
 $this->**cpu**->stop();  
 }  
 }  
}  
  
**namespace** {  
 **use** FacadePattern\Computer;  
  
 *// Facade - это компьютер, состоящий из множества частей с множеством операций.  
 // Но клиенту о них. Его воднует простота. Ему лишь надо включить компьютер,  
 // а затем выключить его после работы.* $facade = **new** Computer;  
 $facade->startComputer();  
 $facade->stopComputer();  
}

#### Decorator (Декоратор)

Т.е. у нас есть объект, который мы оборачиваем и сливаем с другим объектом такого же типа, который внедряется в каждый метод объекта, вызывает его, а потом модифицирует результат вызова.

**Проблема**

Альтернативное решение и замена наследованию для расширения функционала класса.

Позволяет избежать перегруженных методами объектов. Вместо создания нового метода, модифицирует уже существующий метод.

**Используется для**

1. Динамического подключения дополнительного поведения к объекту. Данное поведение может как расширять, так даже и заменять поведение декорируемого объекта.
2. Используем для создания и расширения модульных систем.

**Особенности реализации**

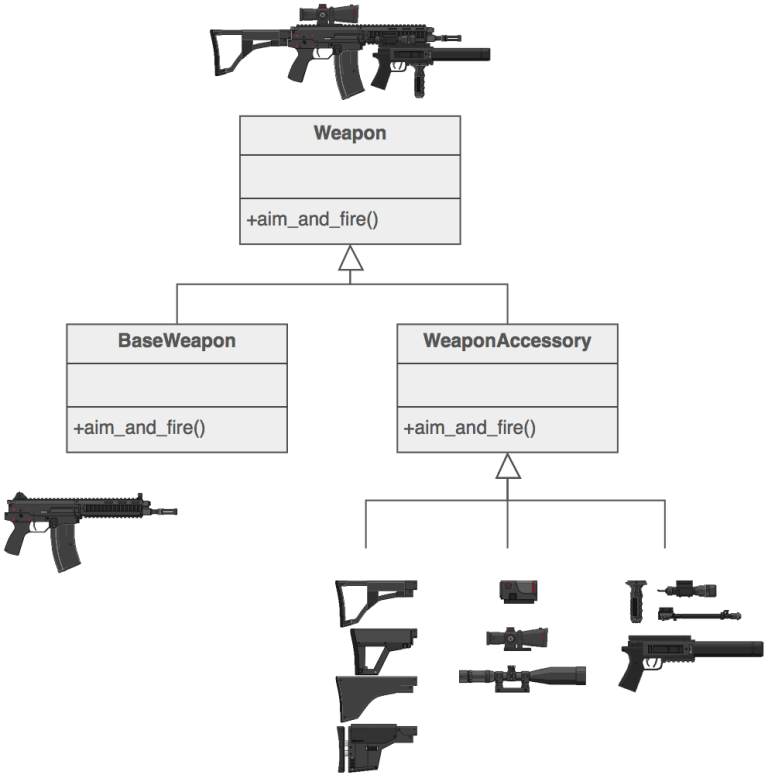
**Декоратор имеет такой же интерфейс, как и декорируемый объект и хранит декорируемый объект в себе. Каждая реализация метода в декораторе вызывает аналогичную функцию в декорируемом объекте и модифицирует ее результат.**

Декоратор предусматривает расширение функциональности объекта без определения подклассов (наследования), а с помощью обертывания базового (декорируемого) объекта другим объектом, реализующим такой же интерфейс.

**Недостатки**

При наличии большого числа декораторов система становится сложна в изучении. Поскольку имеем большое число похожих однотипных объектов, иногда становится непонятно, что из них декорируемые объекты, а что декораторы.

**Пример**



**namespace** DecoratorPattern {  
 *// Интерфейс базового модуля.* **abstract class** AbstractModule {  
 **abstract public function** operation();  
 }  
  
 *// Подмодуль-декоратор к модулю, расширяющий функционал модуля и реализующий тот же интерфейс.* **abstract class** AbstractSubmodule **extends** AbstractModule {  
 **protected $component**;  
  
 **public function** \_\_construct(AbstractModule $component) {  
 $this->**component** = $component;  
 }  
 }  
  
 *// Конкретный модуль на основе базового интерфейса.* **class** Module **extends** AbstractModule {  
 **public function** operation() {  
 **print 'Basic Module functionality<br/>'**;  
 }  
 }  
  
 *// Конкретный сабмодуль на том же интерфейсе, расширяющий модуль.* **class** Submodule **extends** AbstractSubmodule {  
 **public function** operation() {  
 *// Загрузили базовый модуль.  
 // Добавили функциональность.* **print 'Extended Functionality 1<br/>'**;  
 *// Обязательно вызвали функциональность загруженого модуля.* $this->**component**->operation();  
 *// И еще добавили функциональности, модифицируя результат.* **print 'Extended Functionality 2<br/>'**;  
 }  
 }  
}  
  
**namespace** {  
 **use** DecoratorPattern\Submodule, DecoratorPattern\Module;  
  
 *// Здесь несколько неожиданный пример.  
 // Подмодуль именно обертывает базовый функционал модуля,  
 // расширяя его функциональность.* $decoratedComponent = **new** Submodule(**new** Module());  
 $decoratedComponent->operation();  
}

#### Composite (Компоновщик)

**Проблема**

1. Есть базовый объект и составной объект, состоящий из базовых объектов. Необходимо обращаться к составным объектам так же, как базовым.
2. Клиент не должен знать, составной это объект или базовый и из скольких частей он состоит. Он просто должен использовать один и тот же интерфейс.

**Используется для**

1. **Для реализации древовидной структуры.** Например, пользовательского интерфейса: есть листья (кнопки, поля) и композиты (кона с кнопками и полями), которые в свою очередь являются составными частями элемента Окно.
2. **Либо представление документа:** страница состоит из параграфов, состоящих из предложений, состоящих из слов, состоящих из букв.
3. **В Drupal примером композита является Render API, где в итоге собирается объект страницы, состоящий из композитных узлов (блоки, регионы) и узлов-листьев (логотипа, мессаджа в футере и т.д.). Drupal идет по массивы и рекурсивно отрисовывает элементы, используя ‘#theme’-функции каждого узла.**
4. Обеспечения одинаковой работы (создание единого интерфейса) с базовыми классами, составными классами, состоящими из базовых классов и составными классами, состоящими из составных классов.
5. Для создания иерархии классов, которые одновременно могут состоять как из простых, так и из сложных объектов (состоящих из простых) и имеющий единый интерфейс.
6. Для рекурсивного обходы всех узлов

**Особенности реализации**

1. Базовый интерфейс Unit
2. Неделимый объект Soldier extends Unit – называется Leaf (лист, примитив)
3. Составной объект Army (состоит из множества Soldier и Army) extends Unit - называется Composite (Композит)
4. Вызывая метод Army->shoot(), мы вызываем все методы shoot() всех узлов, как листов так и композитов.

Army->shoot TankArmy->shoot Tank1->shoot

Tank2->shoot

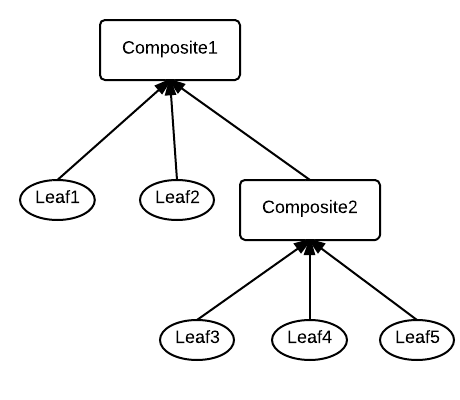
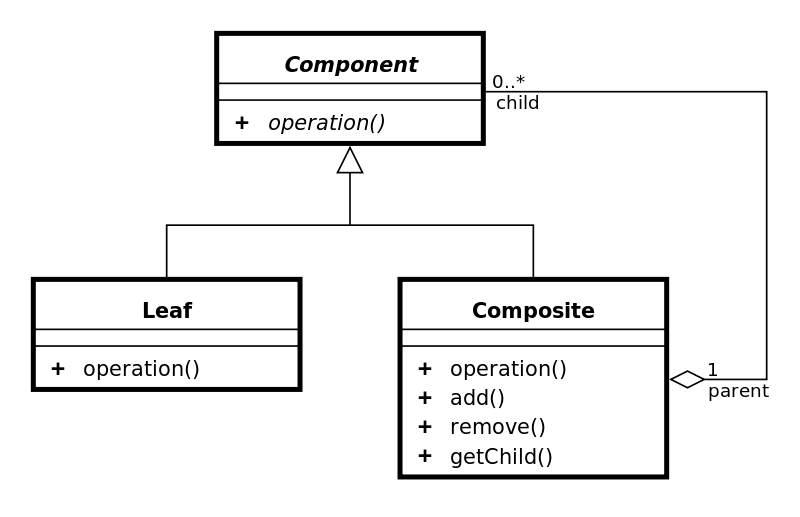
RocketArmy->shoot Rocket1->shoot

Rocket2->shoot

**Недостатки**

1. Неудобно и некрасиво, если надо будет запретить добавление в композитный объект базовых объектов определенного типа. Например, в TankArmy нельзя добавлять юнит Plane или юнит Ship.
2. При создании дерева с большим уровнем иерархии затраты памяти и производительности на обход всего дерева очень велики. И тогда ряд свойств листьев мы можем вынести в паттерн Flyweight.

**Пример**



**namespace** CompositeTemplate {  
 *// Базовый интерфейс для объектов: как базового, так и составного.* **abstract class** Unit {  
 **protected $name**;  
  
 **public function** \_\_construct($name) {  
 $this->**name** = $name;  
 }  
 **public abstract function** add(Unit $c);  
 **public abstract function** remove(Unit $c);  
 **public abstract function** shoot();  
 }  
  
 *// Базовый объект. По логике программы не может быть составным.* **class** Soldier **extends** Unit {  
 **public function** shoot() {  
 **print** $this->**name** . **' shoots<br/>'**;  
 }  
 *// Заглушим для несоставного объекта метод add().* **public function** add(Unit $c) {  
 **print**(**'It is not a combined object.'**);  
 }  
 *// Заглушим для несоставного объекта метод remove().* **public function** remove(Unit $c) {  
 **print**(**'It is not a combined object.'**);  
 }  
 }  
  
 *// Составной объект-композит.* **class** Army **extends** Unit {  
 **private $children** = **array**();  
 *// Вот здесь соль.  
 // Т.е. композитный объект использует вызовы аналогичных методов  
 // своих составных частей.* **public function** shoot() {  
 **foreach**($this->**children as** $child) {  
 $child->shoot();  
 }  
 }  
 *// Реализуем метод add().* **public function** add(Unit $component) {  
 $this->**children**[$component->**name**] = $component;  
 }  
 *// Реализуем метод remove().* **public function** remove(Unit $component) {  
 **unset**($this->**children**[$component->**name**]);  
 }  
 }  
}  
  
**namespace** {  
 **use** CompositeTemplate\Army;  
 **use** CompositeTemplate\Soldier;  
  
 *// Глобальный узел.* $global\_army = **new** Army(**'Army'**);  
  
 *// Составной подузел-1 глобального узла.* $tank\_army = **new** Army(**'TankArmy'**);  
 $tank\_army->add(**new** Soldier(**'Tank 1'**));  
 $tank\_army->add(**new** Soldier(**'Tank 2'**));  
  
 *// Составной подузел-2 глобального узла.* $rocket\_army = **new** Army(**'RocketArmy'**);  
 $rocket\_army->add(**new** Soldier(**'Rocket 1'**));  
 $rocket\_army->add(**new** Soldier(**'Rocket 2'**));  
  
 *// Добавляем составные подузлы в глобальный узел.* $global\_army->add($tank\_army);  
 $global\_army->add($rocket\_army);  
  
 *// Добавляем базовые узлы в глобальный узел.* $global\_army->add(**new** Soldier(**'Soldier 1'**));  
 $global\_army->add(**new** Soldier(**'Soldier 2'**));  
 $one\_more\_soldier = **new** Soldier(**'Soldier 3'**);  
 $global\_army->add($one\_more\_soldier);  
 $global\_army->remove($one\_more\_soldier);  
  
 *// Выводим весь объект рекурсивно.* $global\_army->shoot();  
}

#### Flyweight (Приспособленец)

<http://habrahabr.ru/post/88393/>

**Дополняет шаблон Factory Method и шаблон Композит.**

Отличие от Фабрики в том, что Приспособленец не просто создает новый объект, а прежде всего, имеет пул таких объектов, откуда может быстро вернуть уже готовый объект, вместо медленного создания нового. Профит – скорость работы и экономия ресурсов.

**Проблема**

1. **Ускоряет генерацию большого количества мелких объектов ограниченного количества подтипов за счет повторного использования объектов из пула, тем самым экономя ресурсы приложения.**
2. При реализации дерева объектов с глубокой иерархией на самом нижнем уровне огромное количество примитивов-листьев будет однотипными (или будут иметь некоторое ограниченное количество типов). Поэтому нет большого смысла хранить эти объекты полностью (со всеми повторяющимися свойствами), потому что это ведет к большому перерасходу памяти.

Мы будем хранить только тип листа, а отдельный объект у нас будет

* 1. принимать тип
  2. искать у себя в пуле объект такого типа
  3. **если такого объекта еще нет - генерировать его и помещать в пул**
  4. возвращать объект

1. Таким образом, объект создается только один раз. А при дальнейших запросах очень быстро возвращается.

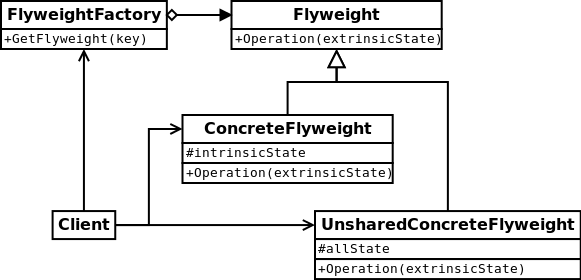
**Используется для**

Используется для ускорения работы и экономии ресурсов при генерации фабрикой большого количества схожих объектов (примитивов-листьев у дерева).

**Недостатки**

Если мелкие объекты имеют много разных типов, то Приспособленец не ускорит работу, поскольку вместо возвращения объекта из пула, они лишь будут накапливаться в нем.

**Пример**



**namespace** FlyweightTemplate {  
 *// Flyweight-фабрика по созданию объектов.  
 // Отличается от обычной фабрики наличием пула.* **class** SoldierFactory {  
 *// Пул приспособленцев. При запросе нового экземпляра, фабрика ищет в пуле.  
 // И только если его в пуле нет, фабрика создает новый.* **private $soldiers** = **array**();  
 *// Здесь вся соль и отличие от фабрики.  
 // Фабрика просто создает новый объект,  
 // а Приспособленец прежде всего ищет объект у себя в пуле.*

*// Это называется 'ленивой инициализацией' -  
 // когда мы инициализируем объект всего один раз, а потом используем его.*

**public function** getSoldier($type) {  
 **if** (!*array\_key\_exists*($type, $this->**soldiers**)) {  
 **switch** ($type) {  
 **case 'A'**:  
 $this->**soldiers**[$type] = **new** SoldierA();  
 **break**;  
 **case 'B'**:  
 $this->**soldiers**[$type] = **new** SoldierB();  
 **break**;  
 }  
 }  
 **return** $this->**soldiers**[$type];  
 }  
 }  
  
 *// Basic Flyweight.* **abstract class** Soldier {  
 **protected $type**;  
 **protected $weapon**;  
 **protected $armor**;  
 **protected $strength**;  
  
 **public abstract function** shoot($strength);  
 }  
  
 *// ConcreteFlyweight A.* **class** SoldierA **extends** Soldier {  
 **public function** \_\_construct() {  
 $this->**type** = **'A'**;  
 $this->**weapon** = **'TommyGun'**;  
 $this->**armor** = **'Light Armor'**;  
 }  
 **public function** shoot($strength) {  
 $this->**strength** = $strength;  
 **print**(**'Soldier '** . $this->**type** . **' -> shoots: '** . $this->**strength** . **'<br/>'**);  
 }  
 }  
  
 *// ConcreteFlyweight B.* **class** SoldierB **extends** Soldier {  
 **public function** \_\_construct() {  
 $this->**type** = **'B'**;  
 $this->**weapon** = **'RocketGun'**;  
 $this->**armor** = **'Heavy Armor'**;  
 }  
 **public function** shoot($strength) {  
 $this->**strength** = $strength;  
 **print**(**'Soldier '** . $this->**type** . **' -> shoots: '** . $this->**strength** . **'<br/>'**);  
 }  
 }  
}  
  
**namespace** {  
 **use** FlyweightTemplate\SoldierFactory;  
  
 $soldier\_factory = **new** SoldierFactory();  
 $soldier\_types = **array**(**'A'**, **'B'**, **'B'**, **'A'**);  
  
 *// На каждый полученный тип - объект данного типа.* **foreach** ($soldier\_types **as** $type) {  
 *// Flyweight-фабрика возвращает солдата.  
 // Причем нпервые 2 раза она создаст объекты, положит их в пул, и вернет их.  
 // А вторые 2 раза она быстро вернет объекты из пула, экономя ресурсы.* $soldier = $soldier\_factory->getSoldier($type);  
 *// Солдат выполняет действие.* $soldier->shoot(*mt\_rand*(1, 10));  
 }  
}

#### Proxy (Заместитель, Суррогат)

<https://ru.wikipedia.org/wiki/Proxy_%28%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F%29>

<https://www.youtube.com/watch?v=HVGzZ3sBOy4> – 1 часть

<https://www.youtube.com/watch?v=q-an2KfTT8o> – 2 часть

**Это объект такого же типа (интерфейса), который контролирует доступ к другому объекту, перехватывая все вызовы. Т.е. контейнер по сути, заместитель-оболочка вокруг оригинального продукта. Используется для замещения оригинального объекта (допустим, переписаны некоторые методы).**

Работа с прокси ничем не отличается от работы с оригинальным объектом, поскольку прокси реализует тот же интерфейс. Поэтому никто на стороне клиента не знает, что мы используем прокси вокруг реального объекта. Клиент думает, что он работает с реальным объектом.

Proxy предоставляет тот же интерфейс, существенно изменяя существующие методы.

Decorator предоставляет тот же интерфейс, однако расширяет существующие методы на базе функционала базовых методов.

Decorator и Proxy имеют разные цели, но схожие структуры. Оба вводят дополнительный уровень косвенности: их реализации хранят ссылку на объект, на который они отправляют запросы.

Adapter предоставляет своему объекту другой интерфейс, приводя объект к другому интерфейсу.

**Проблема**

1. Необходимо проконтролировать доступ к объекту (особенно к тяжелому). Проверить, имеет ли пользователь доступ к тому или иному методу.
2. Необходимо более гибко работать с тяжелыми объектами.

**Используется для**

1. Наиболее частое применение – для ленивой загрузки (lazy load). Тяжелые объекты не всегда разумно загружать в момент инициализации. Намного лучше загрузить их во время вызова конкретного метода (поскольку есть вероятность, что вызывать метод может не понадобиться).
2. Клиентская программа обращается к нашей библиотеке (на сервере) с запросом за объектом, передавая определённые параметры.
3. Наша библиотека должна вернуть объект клиенту. Однако нам в будущем может прийти задачи типа
   1. Отдавать объект клиенту только с определенного IP
   2. Отдавать только на каждый n-запрос
   3. Проверять определенный параметр-хеш (права) перед отдачей
   4. и т.д.
4. При этом библиотека вендорская – т.е. ничего внутри нее менять нельзя.
5. Поэтому любой клиентский запрос приходит объекту Прокси с тем же интерфейсом. Прокси делает все нужные проверки, и возвращает оригинальный объект, или возвращает Прокси-объект с измененными свойствами, или не возвращает никаких объектов.
6. Может использоваться для работы с тяжелыми объектами.
7. Для обеспечения доступа к объекту.

**Разновидности паттерна Прокси**

**!!! У данного паттерна около 10 разновидностей. Подробнее см. ссылку википедии.**

1. Виртуальный proxy является заместителем объектов, создание которых обходится дорого. Реальный объект создается только при первом запросе/доступе клиента к объекту. Т.е. используется пулл объектов или ленивая инициализация.
2. Защитный proxy контролирует доступ к основному объекту. «Суррогатный» объект предоставляет доступ к реальному объекту, только если вызывающий объект имеет соответствующие права
3. Интеллектуальный proxy выполняет дополнительные действия при доступе к объекту

**Особенности реализации**

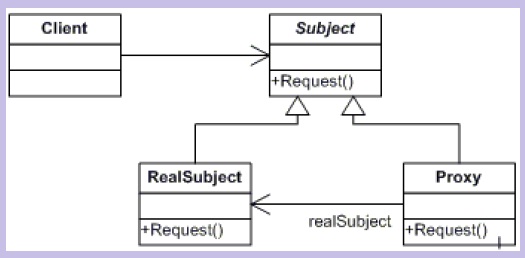
1. Это оболочка над реальным объектом. Прокси-объект делегирует вызов к другому объекту и подвешивается на действия.
2. Есть оригинал Human (субъект, real subject) и его заместитель Robot (proxy). **Интерфейс субъекта и суррогата совпадают.** Т.е. субъект может использоваться вместо суррогата и наоборот, вместо субъекта может использоваться прокси.

Human сидит внутри и управляет Robot-ом в бою. Robot выполняет боевые действия (бежать, стрелять). Однако Human также может выйти из робота и выполнить те же действия (одинаковый интерфейс).

**Недостатки**

1. Повышенное время и ресурсы при обращении к реальному объекту, поскольку запрос проходит через прослойку Прокси (запрос проксируется).

**Пример**



**namespace** ProxyPattern {  
 *// Объект-клиент, который на самом деле не знает,  
 // с каким объектом (оригинальным либо прокси) он будет работать.  
 // Поскольку объекты реализуют один интерфейс, то клиенту все равно,  
 // какой объект ему передадут.* **class** Client {  
 **public function** getFastRequest(Request $request, $x, $y) {  
 **print** $request->fastRequest($x, $y) . **'<br/>'**;  
 }  
 **public function** getHeavyRequest(Request $request, $x, $y) {  
 **print** $request->heavyRequest($x, $y) . **'<br/>'**;  
 }  
 **public function** getSecureRequest(Request $request, $hash, $x, $y) {  
 **print** $request->secureRequest($hash, $x, $y) . **'<br/>'**;  
 }  
 }  
  
 *// Общий интерфейс для оригинального объекта и прокси.* **interface** Request {  
 **function** fastRequest($x, $y);  
 **function** heavyRequest($x, $y);  
 **function** secureRequest($hash, $x, $y);  
 }  
  
 *// Оригинальный очень тяжелый объект, его мы не хотим создавать без причины.  
 // Допустим, соединение с БД либо с удаленным сервером.* **class** OriginalRequest **implements** Request {  
 *// Очень тяжелый конструктор выполняется при создании объекта.* **public function** \_\_construct() {  
 **print** (**'... heavy operation ... wait ... '**);  
 *sleep*(3);  
 }  
 **public function** fastRequest($x, $y) {  
 **return 'fast response (from original): '** . ($x + $y - 5);  
 }  
 **public function** heavyRequest($x, $y){  
 **return 'slow response (from original): '** . ($x \* $y \* 100);  
 }  
 **public function** secureRequest($hash, $x, $y) {  
 **if** (!**empty**($hash)) {  
 **return 'secure response (from original): '** . ($x \* $y);  
 }  
 **else** {  
 **return 'no access (from original)'**;  
 }  
 }  
 }  
  
 *// Proxy - замещающий обьект с тем же интерфейсом.  
 // В данном объекте мы перепишем ряд методов.  
 //  
 // Нам не нравится, что для создания объекта, объект выполняет очень тяжелый конструктор.  
 // Даже для вызова самого незначительного метода.  
 // Для этого мы используем ленивую загрузку (lazy initialization) -  
 // в конструкторе не будем ничего загружать, а вот в методе, который требует данных конструктора,  
 // создадим оригинальный объект и используем его данные.  
 //  
 // Также некоторые методы мы хотим перезаписать полностью.  
 //  
 // А к некоторым методам сильнее ограничить доступ.* **class** ProxyRequest **implements** Request {  
 **protected $request**;  
  
 **public function** \_\_construct() {  
 $this->**request** = **NULL**;  
 }  
 *// Быстрая операция - не требует создания реального объекта.  
 // Не будем вызывать оригинальный объект и сделаем ее еще быстрее.* **public function** fastRequest($x, $y) {  
 **return 'super fast response (from proxy): '** . ($x + $y - $y);  
 }  
 *// Медленная операция - требует создания реального объекта.  
 // Поэтому адресуем ее оригинальному объекту, создав данный объект  
 // и вызвав его аналогичный метод.* **public function** heavyRequest($x, $y) {  
 **if** ($this->**request** == **NULL**) {  
 $this->**request** = **new** OriginalRequest();  
 }  
  
 **return** $this->**request**->heavyRequest($x, $y);  
 }  
 *// Защищенная медленная операция.  
 // Защитим ее еще сильнее.* **public function** secureRequest($hash, $x, $y) {  
 **if** ($this->**request** == **NULL**) {  
 $this->**request** = **new** OriginalRequest();  
 }  
  
 **if** ($hash == **'correct\_password'**) {  
 **return** $this->**request**->secureRequest($hash, $x, $y);  
 }  
 **else** {  
 **return 'no access (from proxy)'**;  
 }  
 }  
 }  
}  
  
**namespace** {  
 **use** ProxyPattern\Client;  
 **use** ProxyPattern\OriginalRequest;  
 **use** ProxyPattern\ProxyRequest;  
  
 *// Мы обращаемся к свойствам реального объекта именно через объект-посредник.  
 // Который в свою очередь контролирует доступ.* $client = **new** Client();  
  
 *// Можем возвращать клиенту оригинальный объект.* $client->getFastRequest(**new** OriginalRequest, 5, 10);  
 $client->getHeavyRequest(**new** OriginalRequest, 5, 10);  
 $client->getSecureRequest(**new** OriginalRequest, **'any\_pass'**, 5, 10);  
  
 *// Либо можем возвращать клиенту Прокси-объект,  
 // который вызовет в своем методе внутри соответствующий метод  
 // оригинального объекта, соответственно его обработав.  
 // Также прокси может просто заоверрайдить один из методов целиком.* $client->getFastRequest(**new** ProxyRequest, 5, 10);  
 $client->getHeavyRequest(**new** ProxyRequest, 5, 10);  
 $client->getSecureRequest(**new** ProxyRequest, **'any\_pass'**, 5, 10);  
}

#### Bridge (Мост)

<http://cpp-reference.ru/patterns/structural-patterns/bridge/> - хорошо объяснено

<http://javatutor.net/books/tip/connecting_different_types> (секция Bridge)

**Проблема**

1. Мы хотим создать объект, имеющий комбинированное значение свойств любого подкласса иерархии А (передний план) и любого подкласса иерархии B (задний план)
2. Мы не хотим для этого использовать наследование, так как возможных комбинаций может быть слишком много. Т.е. нашу иерархию слишком сложно было бы поддерживать для получения объекта с определенной комбинацией свойств, она бы слишком разрасталась по мере усложнения приложения.

**Т.е. использование данного паттерна сокращает количество подклассов в системе и скрывает реализацию от клиента. Т.е. нет эффекта взрывного роста подклассов в системе.**

1. **Кроме того, мы можем легко добавлять необходимое количество объектов переднего и заднего плана и гибко комбинировать их.**

**Используется для**

1. Частей программы, где может измениться как базовый класс переднего плана, так и базовый класс заднего плана (использующийся передним планом).
2. Для замены связей через наследование более гибкой структурой.

**Вариант с наследованием – необходимо поддерживать 3 уровень с множеством комбинаций.**

**Введение нового объекта 2-го уровня заставляет создавать кучу зависимых наследников-комбинаций 3-го уровня.**

Car PrototypeCar PrototypeCarMinivanType

PrototypeCarCrossoverType

PrototypeCarPickupType

ProductionCar ProductionCarMinivanType

ProductionCarCrossoverType

PrototypeCarPickupType

**Вариант Bridge – только 2 уровень, которые принимает любой тип.**

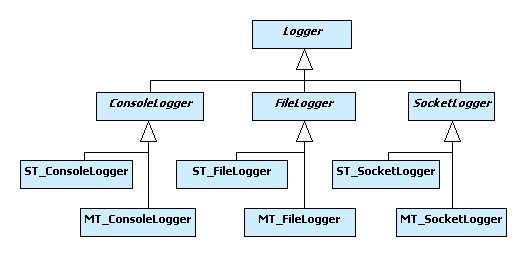
**Введение нового объекта 2-го уровня не зависит от наследников.**

Car PrototypeCar(MinivanType || CrossoverType || PickupType)

ProductionCar(MinivanType || CrossoverType || PickupType)

**Особенности реализации**

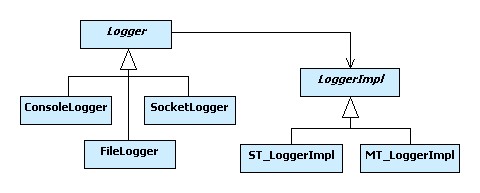
Классический подход на наследовании: куча недостатков – для получения комбинации надо вырастить огромное 3-уровневое дерево. В случае необходимости изменения класса logger необходимо будет изменить всех потомков. Потому что они зависимы от класса Logger.



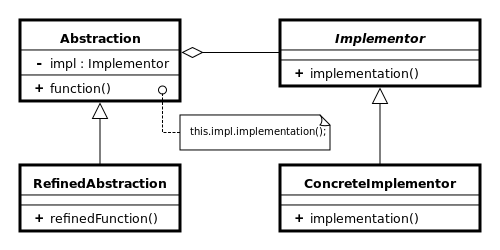
Применение паттерна Мост: 2 двухуровневых дерева, при этом для получения комбинации A + B достаточно подклассу A передать объект B

В данном случае внутри класса logger есть ссылка на объект loggerImplementation.

Таким образом мы можем изменить класс logger, но при этом не надо будет менять объект loggerImpolementation. Потому что здесь мы комбинируем 2 независимые иерархии.



**Пример**



**namespace** BridgePattern {  
 *// Иерархия 1. Стадия готовности автомобиля к выпуску.* **abstract class** Car {  
 *// Сюда мы загрузили из конструктора объект из иерархии B.* **protected $type**;  
  
 **public function** \_\_construct(Type $type) {  
 $this->**type** = $type;  
 }  
 **abstract public function** drive();  
 }  
  
 *// Подкласс иерархии 1. Помним, что каждый конкретный подкласс  
 // содержит внутри себя ссылку на конкретный подкласс иерархии 2.  
 // Таким образом данный подкласс 1 комбинирует свои свойства с подклассом 2.* **class** PrototypeCar **extends** Car {  
 **private $state** = **'prototype'**;  
 **private $speed** = **'slow'**;

*// Вот здесь мы как раз и взаимодействуем с сохраненным объектом $type,  
 // получая его свойства и методы.*  
 **public function** drive() {  
 **print** $this->**state** . **' '** . $this->**type**->getTypeName() . **' drives '** . $this->**speed** . **'<br />'**;  
 }  
 }  
  
 *// Подкласс иерархии 1.* **class** ProductionCar **extends** Car {  
 **private $state** = **'production'**;  
 **private $speed** = **'fast'**;  
  
 **public function** drive() {  
 **print** $this->**state** . **' '** . $this->**type**->getTypeName() . **' drives '** . $this->**speed** . **'<br />'**;  
 }  
 }  
  
 *// Иерархия 2. Тип готовящегося авто.  
 // Таким образом мы можем иметь до 4-х комбинаций.* **abstract class** Type {  
 **abstract function** getTypeName();  
 }  
  
 **class** MinivanType **extends** Type {  
 **private $typeName** = **'minivan'**;  
  
 **public function** getTypeName() {  
 **return** $this->**typeName**;  
 }  
 }  
  
 **class** CrossoverType **extends** Type {  
 **private $typeName** = **'crossover'**;  
  
 **public function** getTypeName() {  
 **return** $this->**typeName**;  
 }  
 }  
}  
  
**namespace** {  
 **use** BridgePattern\PrototypeCar;  
 **use** BridgePattern\ProductionCar;  
 **use** BridgePattern\CrossoverType;  
 **use** BridgePattern\MinivanType;  
  
 *// Как видим, паттерн Мост позволяет нам гибко комбинировать объекты.  
 // Т.е. мы могли бы создать с помощью наследования все возможные комбинации.  
 // Но вместо этого мы имеем 2 иерархии, которые компонуются друг с другом.* $prototypeMinivan = **new** PrototypeCar(**new** MinivanType());  
 $prototypeMinivan->drive();  
  
 $productionCrossover = **new** ProductionCar(**new** CrossoverType());  
 $productionCrossover->drive();  
}

### Поведенческие паттерны (11 main patterns)

<http://www.pcmag.ru/solutions/detail.php?ID=34633>

**Основная задача данной группы паттернов – обеспечить взаимодействие объектов и классов.**

Есть поведенческие паттерны уровня классов (используется наследование) и уровня объектов (используется композиция).

#### Observer (Наблюдатель, Подчиненные, Dependents, Publisher-Subscriber, Listener)

**Observer и Visitor – наиболее употребляемые и важные паттерны в Drupal.**

Реализует зависимость типа “Один ко многим”.

В PHP осуществляется встроенная поддержка этого шаблона через входящее в поставку

расширение SPL (Standard PHP Library):

SplObserver - интерфейс для Observer (наблюдателя),

SplSubject - интерфейс Observable (наблюдаемого),

SplObjectStorage - вспомогательный класс (обеспечивает улучшенное сохранение и удаление

объектов, в частности, реализованы методы attach() и detach()).

**Проблема**

При изменении состояния объекта уведомить и обновить все зависящие от него объекты. Количество зависимых объектов может быть динамическим.

**Используется для**

Суть событийного программирования. Что-то в системе случается, и в ответ выполняется набор некоторых операций, которые могут быть динамически добавлены в систему.

View в MVC, изменяется модель (Model) – автоматически изменяется интерфейс (View).

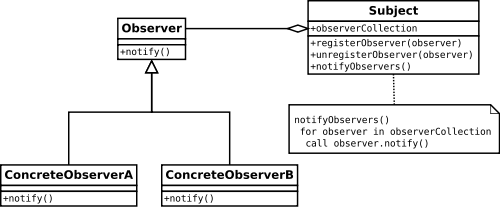
Систем, где существует объект, рассылающий сообщения получателям. Количество получателей может динамически меняться, но не менее одного. Отправителя сообщений в целом не интересует, что получатели будут делать с полученной информацией, ему просто надо переслать ее им.

**Особенности реализации**

Подписчики new Listener() при регистрации добавляют себя Event::getInstance()->registerListener($this) в каталог-пул email-рассылки Event.

Когда рассылка рассылается Event::getInstance()->fireEvent(), она в цикле вызывает внутри каждого подписавшегося реакцию Listener()->notify(). Таким образом, каждая разосланная рассылка вызовет реакцию подписчиков.

**Пример**



**namespace** ObserverPattern {  
 **interface** Observer {  
 **function** notify($event);  
 }  
  
 *// Подписчик на событие.* **class** Listener **implements** Observer {  
 **private $name** = **''**;  
  
 *// При создании слушатель сразу же добавляет себя в список наблюдателей внутри события.* **public function** \_\_construct($name) {  
 $this->**name** = $name;  
 Event::*getInstance*()->registerListener($this);  
 }  
 *// При срабатывании события Event::fireEvent, событием будет вызван данный метод  
 // для каждого подписчика из пула.* **public function** notify($event) {  
 **print** $this->**name** . **' received update!<br/>'**;  
 }  
 }  
  
 *// Класс Синглтон, обладающий методов вызова события,  
 // Синглтон здесь используется только для того, чтобы сохранить подписчиков в пул.* **class** Event {  
 **static private** *$instance* = **NULL**;  
 *// Пул зарегистрированных подписчиков. При возникновении события  
 // пройдемся в цикле по данному массиву и разошлем уведомления.* **private $listeners** = **array**();  
 *// Просто некий произвольный параметр, который показывает что что-то изменилось.* **private $parameter**;  
  
 *// По аналогиии с Синглтоном, закроем возможность создать объект события.  
 // В этом нет смысла, поскольку мы собираемся лишь вызвать Event::fireEvent.* **private function** \_\_construct() {}  
 **private function** \_\_clone() {}  
 *// Введем метод Синглтона, чтобы не иметь возможности создавать метод объекта,  
 // но иметь возможность вызывать обычные методы.* **static public function** getInstance() {  
 **if** (**self**::*$instance* == **NULL**) {  
 **self**::*$instance* = **new** Event();  
 }  
 **return self**::*$instance*;  
 }  
 *// Регистрирует слушателя, добавляя его в пул слушателей внутри события.* **public function** registerListener(Observer $listener) {  
 $this->**listeners**[] = $listener;  
 }  
 *// КЛЮЧЕВОЙ МЕТОД. Вызывает событие, уведомляя при этом о нем всех подписчиков из пула.* **public function** fireEvent($parameter) {  
 $this->**parameter** = $parameter;  
 *// Уведомляет слушателей из пула.* **foreach**($this->**listeners as** $listener) {  
 $listener->notify($this);  
 }  
 }  
 }  
}  
  
**namespace** {  
 **use** ObserverPattern\Listener, ObserverPattern\Event;  
  
 *// Подписчики. Если произойдет событие, они отреагируют на него, поскольку подписаны на него.  
 // Как видим, никаких данных не передается внутрь подписчиков,  
 // поскольку при создании подписчика в конструкторе он добавляет сам себя в пулл слушателей соьытия.* $listener\_1 = **new** Listener(**'Listener 1'**);  
 $listener\_2 = **new** Listener(**'Listener 2'**);  
  
 *// Произошло событие. Если подписчики подписаны на событие данного типа, они отреагируют.* Event::*getInstance*()->fireEvent(15070);  
}

#### Visitor (Посетитель)

**Observer и Visitor – наиболее употребляемые и важные паттерны в Drupal.**

**Проблема**

Добавить в класс новые методы/операции, не изменяя при этом класс.

Необходимо выполнить однотипную операцию/алгоритм над объектами разных типов, но при этом не трогать код объектов.

Реализация добавления нового функционала при отсутствии в PHP возможности множественного наследования.

**Используется для**

Для создания модульных систем типа Drupal, когда каждая часть ядра предусматривает опциональное включение нового модуля, который изменит поведение данной части ядра.

**Особенности реализации**

Есть больной new SinglePatient(‘Max’), которого требуется вылечить.

Пациент открывает дверь врачу, ложится на кушетку и дверь закрывается (далее никто не знает, что происходит).

$patient->acceptDoctor(new DoctorDentist);

За закрытыми дверями внутри метода больного ->acceptDoctor Доктор вызывает свой метод

$doctor->visitSinglePatient($this); // в котором обновляет состояние пациента

$patient->set($patient->get() . ' [cured by dentist]');

Далее мы просто убеждаемся, что у больного изменилось состояние:

var\_dump($patient->get()); // Max [cured by dentist]

**По сути, объект принимает к себе посетителя, и посетитель внутри black box своим методом меняет свойство объекта! Т.е. после приема любого посетителя свойства/состояние объекта изменяется.**

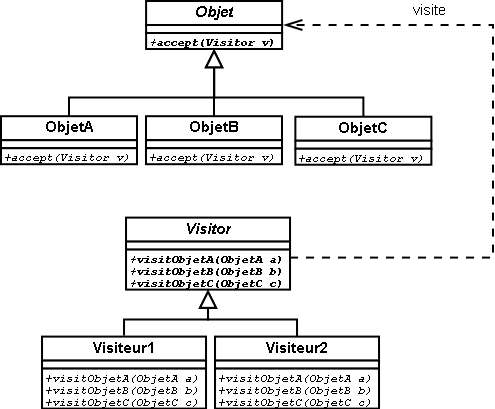
Подвиды паттерна:

1. Single serving visitor – самый популярный вариант. К одному больному приходит врач, делает свою работу и уходит навсегда.
2. Hierarchical visitor – К нескольким больным приходит врач, обходит их всех и делает свою работу.

**Недостатки**

Трудоемко добавление нового класса-реципиента, поскольку нужно добавлять новый метод для поддержки возможности визита нового класса-реципиента базовому посетителю и всем его потомкам.

**Пример**



**namespace** VisitorPattern {  
 *// Пациент. Объект, который не изменяется сам.  
 // Единственное, что он делает - он принимает посетителя,  
 // который потом изменяет его свойства.* **abstract class** Patient {  
 **private $name**;  
  
 **public function** \_\_construct($name) {  
 $this->set($name);  
 }  
 **public function** set($name) {  
 $this->**name** = $name;  
 }  
 **public function** get() {  
 **return** $this->**name**;  
 }  
 **public abstract function** acceptDoctor(Doctor $doctor);  
 }  
  
 *// Конкретный пациент 1. На примере единственного пациента.* **class** SinglePatient **extends** Patient {  
 *// Принимает посетителя, который его потом изменяет.* **public function** acceptDoctor(Doctor $doctor) {  
 $doctor->visitSinglePatient($this);  
 }  
 }  
  
 *// Конкретный пациент 2. На примере группы пациентов.* **class** GroupPatients **extends** Patient {  
 *// Группа принимает посетителя, который потом изменяет каждого члена группы.* **public function** acceptDoctor(Doctor $doctor) {  
 $doctor->visitGroupPatients($this);  
 }  
 }  
  
 *// Доктор-посетитель.* **interface** Doctor {  
 *// Для визита одного посетителя и группы посетителей внедрим 2 метода.* **public function** visitSinglePatient(SinglePatient $patient);  
 **public function** visitGroupPatients(GroupPatients $patients);  
 }  
  
 *// Доктор-стоматолог. Приходит как к одному пациенту, так и в больницу к группе.  
 // Изменяет состояние каждого больного.* **class** DoctorDentist **implements** Doctor {  
 *// Метод изменения состояния одного больного.* **public function** visitSinglePatient(SinglePatient $patient) {  
 $patient->set($patient->get() . **' [cured by dentist]<br/>'**);  
 }  
 *// Метод изменения состояния нескольких больных.* **public function** visitGroupPatients(GroupPatients $patients) {  
 $updatedPatients = **array**();  
  
 **foreach** ($patients->get() **as** $patientNumber => $name) {  
 $updatedPatients[$patientNumber] = $name . **' [cured by dentist]<br/>'**;  
 }  
  
 $patients->set($updatedPatients);  
 }  
 }  
}  
  
**namespace** {  
 **use** VisitorPattern\SinglePatient,  
 VisitorPattern\GroupPatients,  
 VisitorPattern\DoctorDentist;  
  
 $patient = **new** SinglePatient(**'Kate'**);  
 $patients = **new** GroupPatients(**array**(**'Bill'**, **'John'**, **'Maggy'**));  
  
 *// Принимает одиночного посетителя SinglePatient.* $patient->acceptDoctor(**new** DoctorDentist);  
 $patients->acceptDoctor(**new** DoctorDentist);  
  
 *var\_dump*($patient->get());  
 *var\_dump*($patients->get());  
}

#### Mediator (Посредник)

Соответствует GRASP-паттерну Indirection (посредник).

**Проблема**

Обеспечить взаимодействие объектов друг с другом, формируя при этом как можно меньшую связанность.

**Используется для**

Медиатор - это интерфейс взаимодействия между объектами.

Избавляет объекты от необходимости явно ссылаться друг на друга.

Централизует взаимодействие между объектами, инкапсулируя взаимодействие в себе.

Таким образом, объекты не зависят друг от друга, а посылают сообщения медиатору и принимают сообщения от него.

**Особенности реализации**

**Пример сильной связанности**

Buyer->send($message) -> DrugDealer

DrugDealer->send($message) -> Buyer

**Пример слабой связанности**

Buyer->send($message) -> Mediator->intercept($message, $from) -> DrugDealer->receive()

DrugDealer->send($message) -> Mediator->intercept($message, $from) -> Buyer->receive()

Здесь мы видим четкие параллели между покупателем и драгдилером:

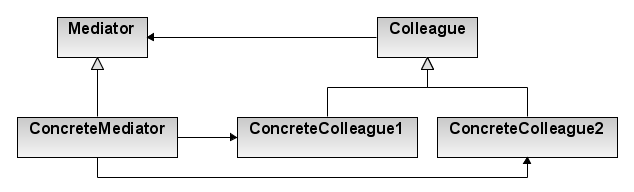
Покупатель отправляет сообщение посреднику, который сам определяет, какому дилеру отправить сообщение.

Драгдилер отвечает также посреднику, который сам определяет, какому покупателю доставить сообщение.

При этом, что бы не случилось с дилером, покупатель останется защищен, поскольку он ничего не знает о дилере.

1. Создается пустой класс-посредник с методом send().
2. **Создаются объекты А и В, получая ссылку на посредник в конструкторе. Чтобы знать, кому посылать сообщение.** У каждого объекта есть метод notify(), который возвращает сообщение.
3. **В свою очередь, посредник также получает ссылки на объекты А и В** с помощью методов setA(A $a) и setB(B $b). **Чтобы знать, кому пересылать ответ.**
4. Объект А (или В) отправляет сообщение методом send($message), внутри которого вызывается метод медиатора intercept($from, $message)
5. Посредник получает сообщение, определяет, что оно от Объекта А, самостоятельно определяет получателя (объект В), вызывая там соответствующий метод receive($from, $message), который обрабатывает и печатает сообщение.

**Пример**



**namespace** MediatorPattern {  
 *// Абстрактный класс посредник.* **abstract class** AbstractMediator {  
 */\*\*  
 \* Перехват сообщения, определение отправителя и отправка указанному получателю.  
 \*/* **public abstract function** intercept(AbstractCompany $from, $message);  
 }  
  
 *// Абстрактный класс основного объекта.* **abstract class** AbstractCompany {  
 **protected $mediator**;  
  
 **public function** \_\_construct(Mediator $mediator) {  
 $this->**mediator** = $mediator;  
 }  
 */\*\*  
 \* Отправка сообщения посреднику.  
 \*/* **public function** send($message){  
 $this->**mediator**->intercept($this, $message);  
 }  
 */\*\*  
 \* Получения сообщения от посредника.  
 \*/* **public abstract function** receive($from, $message);  
 }  
  
 *// Конкретный посредник.* **class** Mediator **extends** AbstractMediator {  
 **private $CompanyA**;  
 **private $CompanyB**;  
  
 **public function** setCompanyA(CompanyA $CompanyA) {  
 $this->**CompanyA** = $CompanyA;  
 }  
 **public function** setCompanyB(CompanyB $CompanyB) {  
 $this->**CompanyB** = $CompanyB;  
 }  
 *// Посредник перехватывает сообщение, определяет, что оно от Компании А,  
 // следовательно пересылает его компании В, вызывая в ней метод receive.  
 // Передает ей сообщение и объект, от которого пришло сообщение.* **public function** intercept(AbstractCompany $from, $message) {  
 **if** ($from == $this->**CompanyA**) {  
 $this->**CompanyB**->receive($from, $message);  
 }  
 **else** {  
 $this->**CompanyA**->receive($from, $message);  
 }  
 }  
 }  
  
 *// Конкретный объект А.* **class** CompanyA **extends** AbstractCompany {  
 **public $name** = **'CompanyA'**;  
 **public function** receive($from, $message) {  
 **echo** *sprintf*(**'%s receives a message from %s: %s<br/>'**, $this->**name**, $from->**name**, $message);  
 }  
 }  
  
 *// Конкретный объект В.* **class** CompanyB **extends** AbstractCompany {  
 **public $name** = **'CompanyB'**;  
 **public function** receive($from, $message) {  
 **echo** *sprintf*(**'%s receives a message from %s: %s<br/>'**, $this->**name**, $from->**name**, $message);  
 }  
 }  
}  
  
**namespace** {  
 **use** MediatorPattern\Mediator, MediatorPattern\CompanyA, MediatorPattern\CompanyB;  
 *// Создали посредник.* $mediator = **new** Mediator();  
 *// Создали 2 основных объекта: каждый объект должен знать о посреднике.  
 // Связали их с посредником: внутри каждого объекта ссылка на посредник.* $companyA = **new** CompanyA($mediator);  
 $companyB = **new** CompanyB($mediator);  
 *// Связали посредник с каждым из объектов: в посреднике ссылка на каждый из объектов.* $mediator->setCompanyA($companyA);  
 $mediator->setCompanyB($companyB);  
 *// Отправили сообщение посреднику, который далее перехватит сообщение и отправит его.* $companyA->send(**'CompanyB, would you like our offer?'**);  
 $companyB->send(**'CompanyA, we like you offer!'**);  
}

#### Template Method (Шаблонный метод)

**Проблема**

Определить основу алгоритма, позволить наследникам переопределить некоторые части алгоритма, защитив от изменений основную суть/структуру алгоритма.

**Используется для**

Для постройки модульной системы, где объект собирается из составных частей, причем основной алгоритм процесса сборки защищен от изменения, а вот отдельные составные части ядра могут быть изменены перед сборкой.

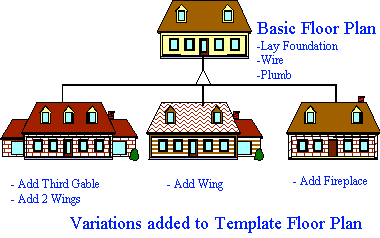
**Особенности реализации**

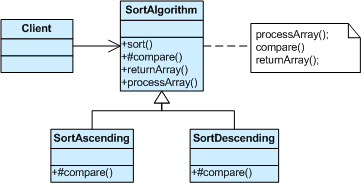
Типовой проект дома abstract House подразумевает 3 операции:

1. prepareRoof($roof) – можно переопределить подготовку и внешний вид **типовой** крыши
2. prepareWalls($walls) – можно переопределить подготовку и внешний вид **типовых** стен
3. final buildHouse() – данный метод является template method, осуществляет сборку и должен быть защищен от изменений. Однако внутри него вызываются методы выше, таким образом, есть ограниченная возможность изменить отдельные части дома.

На базе данного проекта были построены BambooHouse и IceHouse, отличающиеся видом крыши и стен. Но поскольку сам механизм сборки переопределить нельзя, то оба дома собираются по одной схеме – вначале стены, потом крыша.

**Пример**





**namespace** TemplateMethodPattern {  
 *// Строим дом.  
 // Базовый объект, основной алгоритм постройки которого защищен от изменений.  
 // Изменять позволено лишь детали объекта.* **abstract class** House {  
 *// Это пример template метода, который запрещает перезапись.  
 // В PHP это достигается путем использования ключевого слова final.  
 // Т.е. template method - это некий совершенный метод-алгоритм, осуществляющий всю сборку задуманного,  
 // функционал которого строго определен разработчиком и который невозможно перезаписать.* **public final function** buildHouse($houseParts) {  
 $preparedRoof = $this->prepareRoof($houseParts->getRoof());  
 $preparedWalls = $this->prepareWalls($houseParts->getWalls());  
 **print** $preparedRoof . **' on '** . $preparedWalls . **'<br/>'**;  
 }  
 *// А это один из вспомогательных методов, влияющий на детали алгоритма,  
 // который можно переопределить.* **abstract function** prepareRoof($title);  
 *// Еще один малозначительный метод, влияющий на детали алгоритма,  
 // который можно переопределить.* **abstract function** prepareWalls($author);  
 }  
  
 *// Конкретный бамбуковый дом, который мы стоим на базе проекта  
 // и по принципам базового Дома.  
 // Поскольку нам разрешено изменять крышу и стены, воспользуемся этим.* **class** BambooHouse **extends** House {  
 **function** prepareRoof($roof) {  
 **return** *str\_replace*(**' '**, **' bamboo '**, $roof);  
 }  
 **function** prepareWalls($walls) {  
 **return** *str\_replace*(**' '**, **' bamboo '**, $walls);  
 }  
 }  
  
 *// Конкретный ледяной дом на базе базового Дома.  
 // Также изменим крышу и стены.* **class** IceHouse **extends** House {  
 **function** prepareRoof($roof) {  
 **return** *str\_replace*(**' '**, **' ice '**, $roof);  
 }  
 **function** prepareWalls($walls) {  
 **return** *str\_replace*(**' '**, **' ice '**, $walls);  
 }  
 }  
  
 *// Базовые части дома.* **class** HouseParts {  
 **private $roof**;  
 **private $walls**;  
  
 **function** \_\_construct($roof, $walls) {  
 $this->**roof** = $roof;  
 $this->**walls** = $walls;  
 }  
 **function** getRoof() {  
 **return** $this->**roof**;  
 }  
 **function** getWalls() {  
 **return** $this->**walls**;  
 }  
 }  
}  
  
**namespace** {  
 **use** TemplateMethodPattern\HouseParts, TemplateMethodPattern\BambooHouse, TemplateMethodPattern\IceHouse;  
  
 *// Поскольку все нашет дома построены на базе одного базового проекта,  
 // составные части для него мы также используем одни и те же.* $houseParts = **new** HouseParts(**'basic roof'**, **'basic walls'**);  
  
 $bambooHouse = **new** BambooHouse();  
 *// Основной алгоритм постройки дома мы изменить не можем, однако детали будут отличаться.* $bambooHouse->buildHouse($houseParts);  
  
 $iceHouse = **new** IceHouse();  
 *// Основной алгоритм постройки дома мы изменить не можем, однако детали будут отличаться.* $iceHouse->buildHouse($houseParts);  
}

#### State (Состояние, Objects for States)

Объектно-ориентированная реализация конечного автомата (что это такое - хз)

Паттерны State и Bridge имеют много общего, однако Bridge допускает иерархию классов-оберток, а State – нет. Эти паттерны имеют схожие структуры, но решают разные задачи: State изменяет поведение объекта вследствие смены внутреннего состояния, а Bridge разделяет абстракцию от реализации так, что их можно менять независимо друг от друга.

**Проблема**

Во время выполнения программы объект изменяет свое состояние, при этом за сменой состояния должна произойти смена поведения.

**Используется для**

1. Это своего рода альтернатива использованию кучи условных операторов в коде. Т.е. здесь есть объект, у которого при вызове одного и того же метода каждый раз щелкает переключатель – и меняется состояние и вследствие этого меняется реакция.
2. Можем использовать для системы логирования изменения состояния объекта – при изменении состояния будет создаваться запись в лог с соответствующим типом, при критичных изменениях будет дополнительно отправляться email ответственному лицу.

**Особенности реализации**

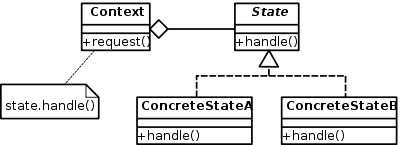
1. Мальчик Boy может иметь несколько настроений (флаги настроений определены в константах в объекте Мальчика: Boy::MOOD\_LAZY, Boy::MOOD\_FUNNY, Boy::MOOD\_ENERGETIC).
2. При создании Мальчика в конструкторе создается объект некого начального настроения и сохраняется в свойстве $mood мальчика.
3. Каждый объект конкретного Настроения также сохраняет в себе ссылку на Мальчика для того, чтобы поменять настроение потом (переключить) и сохранить измененное настроение мальчику.
4. Каждый объект Настроения хранит в себе реакцию на данное настроение Mood::reactionOnMood().
5. Мальчик несколько раз действует (Boy->act()) под воздействием настроения, вызывая реакцию на данное настроение ($this->mood->reactOnMood())
6. Метод реакции на настроение после отработки переключает настроение на следующее Настроение ($this->boy->setMood(Boy:NEXT\_MOOD)). Следовательно, при следующем действии (Boy->act()) его действие-реакция будет уже другим.

**Недостатки**

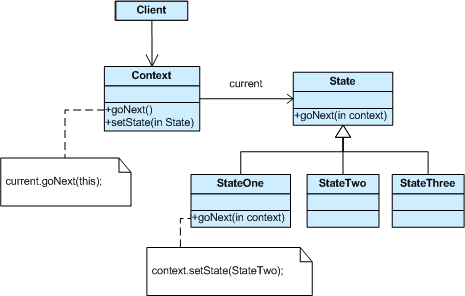
1. Каждый класс конкретного состояния должен знать о хотя бы одном другом классе состояния, чтобы осуществить переключение на него. Таким образом, классы становятся связанными.
2. Предопределена очерёдность вызова состояний.

**Пример**

Базовый вариант



Чуть более сложный вариант с введением объекта-клиента, который создает (порождает) объект, изменяющий состояние (Boy)



**namespace** StatePattern {  
 */\*\*  
 \* Класс 'Мальчик' с несколькими внутренними состояниями.  
 \*/* **class** Boy {  
 *// Здесь хранится текущее настроение,  
 // которое также имеет определенную реакцию на него.* **public $mood**;  
  
 **const *MOOD\_LAZY*** = **'lazy'**;  
 **const *MOOD\_FUNNY*** = **'funny'**;  
 **const *MOOD\_ENERGETIC*** = **'energetic'**;  
  
 *// Задаем дефолтное значение для настроение.* **public function** \_\_construct() {  
 $this->setMood(Boy::***MOOD\_LAZY***);  
 }  
 *// КЛЮЧЕВОЙ!!! метод-сеттер текущего состояния, принимая название настроения,  
 // создает данное настроение вместе с реакцией на него и сохраняет его у мальчика.* **public function** setMood($mood\_name) {  
 **if** ($mood\_name == Boy::***MOOD\_LAZY***) {  
 $this->**mood** = **new** MoodLazy($mood\_name, $this);  
 }  
 **elseif** ($mood\_name == Boy::***MOOD\_FUNNY***) {  
 $this->**mood** = **new** MoodFunny($mood\_name, $this);  
 }  
 **else** {  
 $this->**mood** = **new** MoodEnergetic($mood\_name, $this);  
 }  
 }  
 *// Реакция мальчика под действием конкретного настроения  
 // зависит от объекта настроения и поэтому хранится в нем.  
 // Отреагировав на одно настроение, наше настроение меняется.  
 // Соответственно, следующая реакция будет другой.* **public function** act() {  
 $this->**mood**->reactOnMood();  
 }  
 }  
  
 */\*\*  
 \* Общий интерфейс настроения.  
 \*/* **class** Mood {  
 **protected $mood\_name**;  
 **protected $boy**;  
  
 **public function** \_\_construct($mood\_name, $boy) {  
 $this->**mood\_name** = $mood\_name;  
 $this->**boy** = $boy;  
 }  
 **public function** reactOnMood() {  
 **print "Reaction on standard mood<br/>"**;  
 }  
 }  
  
 */\*\*  
 \* Конкретное настроение. При этом реакция на настроение зависит от него и хранится в нем.  
 \* Т.е. получая данное настроение, мальчик получает и реакцию на него.  
 \* Далее мы можем опционально изменить настроение мальчика,  
 \* чтобы следующая реакция на настроение была другой.  
 \*/* **class** MoodLazy **extends** Mood {  
 **public function** reactOnMood() {  
 **print "Reaction on** $this->**mood\_name mood<br/>"**;  
 *// переключаем состояние* $this->**boy**->setMood(Boy::***MOOD\_FUNNY***);  
 }  
 }  
  
 **class** MoodFunny **extends** Mood {  
 **public function** reactOnMood() {  
 **print "Reaction on** $this->**mood\_name mood<br/>"**;  
 *// переключаем состояние* $this->**boy**->setMood(Boy::***MOOD\_ENERGETIC***);  
 }  
 }  
  
 **class** MoodEnergetic **extends** Mood {  
 **public function** reactOnMood() {  
 **print "Reaction on** $this->**mood\_name mood<br/>"**;  
 *// переключаем состояние* $this->**boy**->setMood(Boy::***MOOD\_LAZY***);  
 }  
 }  
}  
  
**namespace** {  
 **use** StatePattern\Boy;  
  
 $boy = **new** Boy();

*// Помним, что каждая новая реакция мальчика  
 // изменяет его настроение на другое (свойство state).* $boy->act();  
 $boy->act();  
 $boy->act();  
}

#### Memento (Token, Хранитель, Лексема)

**Проблема**

Иметь возможность сохранить на некоторое время, а затем восстановить состояние (свойства) объекта. При сохранении состояния (свойства) не нарушить инкапсуляцию, чтобы свойства не достались врагу.

**Используется для**

Система хранения измененных настроек, а также фиксации и сброса к предыдущим/дефолтных настройкам (отмена или откат). При этом информация о сохраненных настройках открыта только самому объекту.

**Особенности реализации**

Клиент Оригинатор имеет 100$ в бумажнике (состояние). Он кладет их в банковскую ячейку Memento и передает ячейку на хранение банку Кипер. При этом банк не видит, что сохранено в закрытой ячейке (инкапсуляция). В какой-то момент клиент Оригинатор продает все и тратит все свои деньги – 0$ (состояние объекта изменилось).

Он приходит в банк и запрашивает назад свою закрытую ячейку Memento. Далее он из нее забирает 100$, восстанавливая состояние бумажника к исходному уровню.

Оригинатор изменяет свое свойство ->setState(A)

Далее Оригинатор вызывает метод ->saveState(B), инкапсулируя свойство в объекте Memento.

После объект-Оригинатор опять изменяет свое свойство ->setState(B), после чего решает откатиться до предыдущей версии

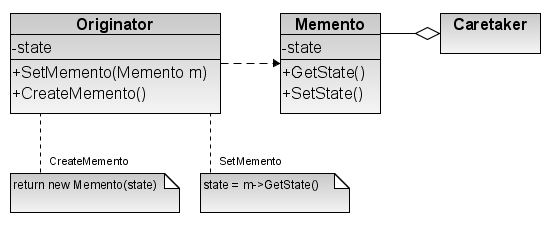
Оригинатор запрашивает методом ->restoreState($memento) и восстанавливает предыдущее значение свойства.

1. Класс Originator – класс, состояние которого необходимо сохранить.
   1. setState($state) – сеттер состояния
   2. getState() – геттер состояния
   3. saveState() – сохраняет состояние в объекте Memento и возвращает Memento объекту-хранителю Keeper
   4. restoreState($memento) – получает объект Memento, хранящий состояние, от Хранителя, и восстанавливает из него состояние (свойства) объекта.
2. Класс Memento – хранитель состояния.
   1. \_\_construct($state) – сохраняет состояние
   2. getState() – возвращает сохраненное состояние

**Недостатки**

1. Если будем хранить большое количество данных в Memento, может быть проблема с ресурсами.

**Пример**



**namespace** MementoPattern {  
 *// Класс, поддерживаюший сохранение состояний внутреннего состояния.  
 // Т.е. именно соостояние данного класса мы будем сохранять, а потом восстанавливать.* **class** Originator {  
 **private $state**;  
  
 *// Задали некое произвольное значение свойству (обычный сеттер).* **public function** setState($state) {  
 $this->**state** = $state;  
 **print** *sprintf*(**"State setted %s"**, $this->**state**) . **'<br/>'**;  
 }  
 *// Вернули значение свойства (обычный геттер).* **public function** getState() {  
 **print** $this->**state** . **'<br/>'**;  
 }  
 *// Позволяет зафиксировать состояние свойства в объекте Memento.* **public function** saveState() {  
 **return new** Memento($this->**state**);  
 }  
 *// Позволяет откатить состояние свойства из сохраненного в Мементо значения.* **public function** restoreState(Memento $memento) {  
 **print** *sprintf*(**"Restoring state..."**) . **'<br/>'**;  
 $this->**state** = $memento->getState();  
 }  
 }  
  
 *// Хранитель состояния. Сюда мы запишем состояние объекта Originator.* **class** Memento {  
 **private $state**;  
  
 **public function** \_\_construct($state) {  
 $this->**state** = $state;  
 }  
 **public function** getState() {  
 **return** $this->**state**;  
 }  
 }  
}  
  
**namespace** {  
 **use** MementoPattern\Originator;  
  
 *// Создали главный объект, состояние которого будем хранить.* $originator = **new** Originator();  
 *// Задали некое исходное состояние объекту.* $originator->setState(**"On"**);  
 *// Зафиксировали состояние в Memento.* $memento = $originator->saveState();  
 *// Опять изменили состоние у объекта.* $originator->setState(**"Off"**);  
 *// Откатили к предыдущему состоянию, запросив данные из Memento.* $originator->restoreState($memento);  
}

#### Strategy (Стратегия)

**Похоже, реализация в коде не совсем соответствует UML-схеме. Поискать и разобрать еще один пример.**

Достаточно простой паттерн.

Имеет отношение к паттерну Flyweight: Strategy objects often make good flyweights

**Проблема**

Выбирать по типу входных данных/по типу клиентов алгоритм (стратегии) из множества алгоритмов, который нужно применить. При этом выбор алгоритма отделен от конкретных реализаций алгоритма (Protected Variations).

Если данные или клиенты не меняются, то в данном паттерне нет смысла, так как его цель – именно выбрать правильный алгоритм на основании меняющихся входных параметров.

**Используется для**

Позволяет менять выбранный алгоритм (стратегию) независимо от объектов-клиентов, которые его используют.

Например, система расчета стоимости товара на основании сезонных скидок, скидок постоянным клиентам.

**Особенности реализации**

Зашедший в магазин клиент Client видит базовую цену товара, а также получает скидку new PremiumCustomerDiscount. Базовая цена и скидка сохраняются в памяти у клиента.

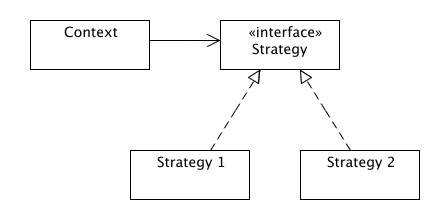
**Тип получаемой скидки зависит от неких внешних параметров** (количества звезд на небе, настроения продавца либо внешнего вида самого клиента). New Client(new PremiumCustomerDiscount, 100$)

Внутри каждой скидки есть метод createPrice($price), который из базовой цены вычисляет цену со скидкой.

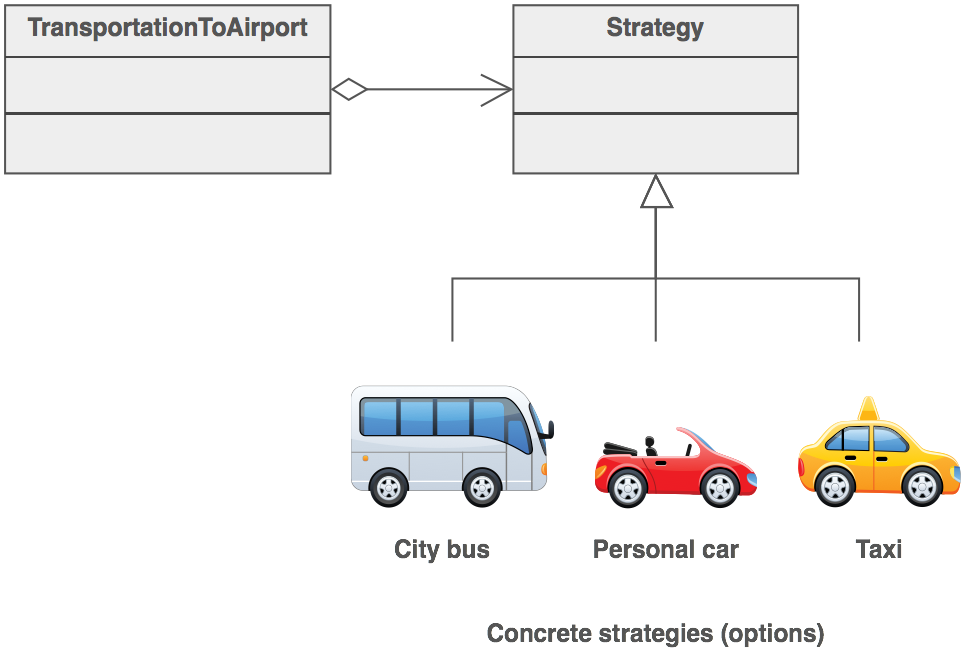
В итоге клиент получает конечную цену Client->calculatePrice(), внутри которого вызывается метод скидки $this->discount->createPrice($this->price)

**Пример**

Базовый пример



Чуть более понятный пример



**namespace** StrategyPattern {  
 *// Базовый интерфейс, описывающий семество/типы стратегий/скидок.* **interface** Discount {  
 **function** createPrice($price);  
 }  
  
 *// Тип скидки 1.* **class** SeasonDiscount **implements** Discount {  
 **function** createPrice($price) {  
 **return** $price - 10;  
 }  
 }  
  
 *// Тип скидки 2.* **class** PremiumCustomerDiscount **implements** Discount {  
 **function** createPrice($price) {  
 **return** $price - 20;  
 }  
 }  
  
 *// Класс-клиент, который имеет базовую цену на товар,  
 // получает разные типы скидок на основании неких внешних параметров.* **class** Client {  
 *// Здесь сохраним базовую цену на товар.* **private $price**;  
 *// Здесь сохраним скидку.* **private $discount**;  
  
 **function** \_\_construct(Discount $discount, $price) {  
 $this->**discount** = $discount;  
 $this->**price** = $price;  
 }  
 *// Ключевой метод расчета цены - объект скидки принимает цену и вычисляет итоговое значение.* **function** calculatePrice() {  
 **print** $this->**discount**->createPrice($this->**price**);  
 }  
 }  
}  
  
**namespace** {  
 **use** StrategyPattern\Client, StrategyPattern\SeasonDiscount, StrategyPattern\PremiumCustomerDiscount;  
  
 $price = 100;  
 $customer\_type = **'A'**;  
  
 *// Исходя из внешних факторов меняются входные параметры: тип клиента в данном случае.  
 // Исходя из этого меняется тип скидки.  
 // Соответственно, меняется и итоговая цена на товар для клиента.* **if** ($customer\_type == **'A'**) {  
 $client = **new** Client(**new** SeasonDiscount, $price);  
 }  
 **else** {  
 $client = **new** Client(**new** PremiumCustomerDiscount, $price);  
 }  
  
 $client->calculatePrice();  
}

#### Chain of Responsibility (Event Loop, Цепочка ответственности)

**Непростой для понимания метод, поскольку используется рекурсия.**

**Проблема**

1. Организовать в системе уровни ответственности.
2. При этом ослабить связь между отправителем и получателем запроса.

**Используется для**

1. Система событий в JS, где событие всплывает через обработчиков, некоторые из которых реагируют на событие. Данная система называется Event Loop.
2. Системы логирования.

**Особенности реализации**

Это набор обработчиков, по очереди получающих запрос и решающих, что делать с ним дальше: обработать (а потом передать дальше или не передавать) либо передать дальше без обработки.

Пишем систему логирования.

3 уровня ошибок: error, warning, notice, у каждой свой лог.

Если возникла ошибка типа error – она попадает в 3 лога. Warning – в 2 лога. Notice – в 1 лог.

Базовый класс Handler определяет 3 типа ошибок (в константах).

Кроме того, он определяет рекурсивный метод processMessageRecursive, определяющий логику отправки сообщения всем обработчикам, и вспомогательный метод addNextHandler, добавляющий на хранение ссылку на следующего обработчика в цепочке.

Производные от Handler объекты-обработчики HandlerError, HandlerWarning, HandlerNotice логируют сообщение в своем формате в свой лог ->logMessage($message).

Класс-клиент Client в конструкторе строит и сохраняет цепочку вызовов объектов-обработчиков:

object(ChainOfResponsibilityPattern\HandlerError)[2]

protected 'priority' => int 3

protected 'next\_handler' =>

object(ChainOfResponsibilityPattern\HandlerWarning)[3]

protected 'priority' => int 2

protected 'next\_handler' =>

object(ChainOfResponsibilityPattern\HandlerNotice)[4]

protected 'priority' => int 1

protected 'next\_handler' => null

Далее он запускает ее для полученного сообщения.

$client = new Client();

$client->runChain ('the system is broken.', Handler::ERROR);

Внутри данного метода вызывается рекурсивный обход цепочки, таким образом, каждый объект может обработать либо пропустить сообщение.

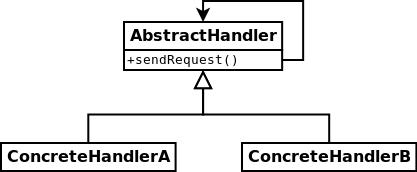
$this->chain->processMessageRecursive($message, $priority);

**Недостатки**

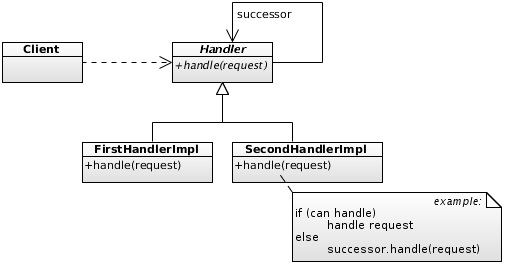
Если следующий обработчик не нашелся, сообщение может быть потеряно. Данному моменту надо уделить повышенное внимание.

**Пример**

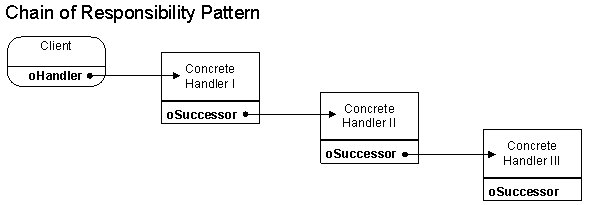
Упрощенная базовая схема



Классическая и не очень понятная схема



Более понятная схема



**namespace** ChainOfResponsibilityPattern {  
 *// Обработчик ошибок.* **abstract class** Handler {  
 **const *ERROR*** = 3;  
 **const *WARNING*** = 2;  
 **const *NOTICE*** = 1;  
  
 *// Приоритет ошибки.* **protected $priority**;  
 *// Следующий элемент в цепочке обязанностей.* **protected $next\_handler**;  
  
 **public function** \_\_construct($priority) {  
 $this->**priority** = $priority;  
 }  
 *// Сохраняет в цепочке и возвращает следующий обработчик.* **public function** addNextHandler(Handler $next\_handler) {  
 $this->**next\_handler** = $next\_handler;  
 **return** $next\_handler;  
 }  
 *// Ключевой рекурсивный метод обработки сообщения путем перебора обработчиков.* **public function** processMessageRecursive($message, $priority) {  
 *// если условие удовлетворено,  
 // то сообщение обрабатывается данным обработчиком  
 // и далее пересылается следующему обработчику.* **if** ($priority >= $this->**priority**) {  
 $this->logMessage($message);  
 }  
 *// Если же условие не удовлетворено, то сообщение сразу пересылается дальше.* **if** ($this->**next\_handler**) {  
 $this->**next\_handler**->processMessageRecursive($message, $priority);  
 }  
 }  
 *// Функция логирования сообщения: каждый объект создаст свою.* **protected abstract function** logMessage($message);  
 }  
  
 *// Класс конкретного обработчика: содержит значение приоритетности ошибки,  
 // на которую он будет реагировать, и определяет свой формат сообщения,* **class** HandlerError **extends** Handler {  
 **public function** \_\_construct() {  
 $this->**priority** = **self**::***ERROR***;  
 }  
 **protected function** logMessage($message) {  
 **print** *sprintf*(**"[error]: %s<br/>"**, $message);  
 }  
 }  
  
 *// Класс конкретного обработчика.* **class** HandlerWarning **extends** Handler {  
 **public function** \_\_construct() {  
 $this->**priority** = **self**::***WARNING***;  
 }  
 **protected function** logMessage($message) {  
 **print** *sprintf*(**"[warning]: %s<br/>"**, $message);  
 }  
 }  
  
 *// Класс конкретного обработчика.* **class** HandlerNotice **extends** Handler {  
 **public function** \_\_construct() {  
 $this->**priority** = **self**::***NOTICE***;  
 }  
 **protected function** logMessage($message) {  
 **print** *sprintf*(**"[notice]: %s<br/>"**, $message);  
 }  
 }  
  
 *// Класс-клиент, строит, хранит и использует цепочку обработчиков.* **class** Client {  
 **protected $chain**;  
  
 *// Строим и сохраняем цепочку обязанностей,  
 // задание последовательно обрабатывается каждым объектом.* **public function** \_\_construct() {  
 *// Создаем первый обработчик.* $handler1 = **new** HandlerError;  
 *// Добавляем в первый обработчик второй обработчик.* $handler2 = $handler1->addNextHandler(**new** HandlerWarning);  
 *// Добавляем во второй обработчик третий обработчик.* $handler2->addNextHandler(**new** HandlerNotice);  
 *// Передаем цепочку: первый обработчик + второй + третий на хранение.* $this->**chain** = $handler1;  
 }  
 *// Запускает цепочку обработчиков для конкретного сообщения.  
 // Каждый обработчик получит сообщение и решит, обрабатывать либо передать его дальше.* **public function** runChain($message, $priority) {  
 $this->**chain**->processMessageRecursive($message, $priority);  
 **print '-------------------------<br/>'**;  
 }  
 }  
}  
  
**namespace** {  
 **use** ChainOfResponsibilityPattern\Client, ChainOfResponsibilityPattern\Handler;  
  
 *// При создании клиента в конструкторе строится цепочка.* $client = **new** Client();  
  
 *// 3 примера обработки сообщения в зависимости от уровня приоритетности:  
 // Высокоприоритетная ошибка типа Handler::ERROR пройдет через все 3 обработчика.* $client->runChain(**'error has occured, the system is entirely broken.'**, Handler::***ERROR***);  
 *// Среднеприоритетная ошибка типа Handler::WARNING пройдет через 2 обработчика.* $client->runChain(**'warning has occured, the system still works.'**, Handler::***WARNING***);  
 *// Низкоприоритетная ошибка типа Handler::WARNING пройдет только через 1 обработчик.* $client->runChain(**'notice has occured, just pay attention.'**, Handler::***NOTICE***);  
}

#### Iterator (Cursor, Итератор)

**Создает единый способ обхода различных типов составных объектов.**

**Кроме того позволяет создать разные способы обхода объектов (например, игнорируя определенные свойства и т.д.).**

**Проблема**

Необходимо получить последовательный доступ (пролистать, проитерировать, перебрать элементы) к данным объекта-коллекции (дерево, связанный список, хэш-таблица, массив).

При этом внутренняя структура коллекции не должна быть раскрыта. Т.е. клиент не знает о ее структуре, а имеет доступ лишь к методам итератора по ней.

Для этого перебор должен выполняться не коллекцией, а отдельным объектом-итератором (упрощает интерфейс коллекции и является более удачным разделением обязанностей).

**Используется для**

Обхода сложных структур, созданных, например, паттерном Composite.

**Особенности реализации**

Правильно реализованный итератор ничего не знает о типе данных, по которому проходят итерации.

Объект-коллекция при создании в конструкторе наполняется элементами.

Также она создает и возвращает клиенту объект-итератор по коллекции, хранящий в себе элементы коллекции.

Объект-итератор представляет собой свойства: набор элементов коллекции и счетчик текущей позиции. А также методы для итерации по коллекции (issetNext(), next())

Объект-Клиент получает в конструктор коллекцию.

Далее получает у коллекции итератор $this->collection->createIterator()

И с помощью методов итератора (Iterator::issetNext(), Iterator::next()) итерирует по коллекции.

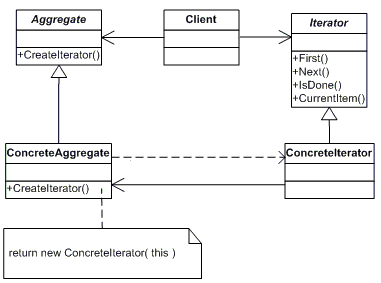
while ($iterator->issetNext()) {

$item = $iterator->next();

print $item->name . ' ' . $item->price . '<br/>';

}

**Пример**



**namespace** IteratorPattern {  
 *// Наличие общего интерфейса удобно для клиента, поскольку клиент отделяется от реализации коллекции объектов.  
 // Collection содержит коллекцию объектов и реализует метод, который возвращает итератор для этой коллекции.* **interface** Collection {  
 *// Метод создает экземпляр Concrete Iterator, с помощью которого можно проитерировать коллекцию.* **public function** createIterator();  
 }  
  
 *// Интерфейс Iterator должен быть реализован всеми итераторами.* **interface** Iterator {  
 *// @return boolean проверяет, существует ли следующий элемент в коллекции.* **public function** issetNext();  
 *// @return mixed возвращает следующий элемент массива.* **public function** *next*();  
 }  
  
 */\*\*  
 \* Объект-коллекция, содержит элементы.  
 \*/* **class** CollectionA **implements** Collection {  
 **public $items** = **array**();  
 *// Здесь задаются элементы.* **public function** \_\_construct() {  
 $this->**items** = **array**(  
 **new** Item(**'Banana'**, **'5$'**),  
 **new** Item(**'Orange'**, **'10$'**),  
 **new** Item(**'Grape'**, **'15$'**),  
 );  
 }  
 *// Метод, в котором возвращает конкретный объект-итератор для данной коллекции.* **public function** createIterator() {  
 **return new** IteratorA($this->**items**);  
 }  
 }  
  
 *// Объект-итератор для данной коллекции.* **class** IteratorA **implements** Iterator {  
 *// Collection elements.* **protected $items** = **array**();  
 *// Current position.* **public $position** = 0;  
  
 **public function** \_\_construct($items) {  
 $this->**items** = $items;  
 }  
 **public function** issetNext() {  
 **if** ($this->**position** >= *count*($this->**items**) || *count*($this->**items**) == 0) {  
 **return FALSE**;  
 }  
 **else** {  
 **return TRUE**;  
 }  
 }  
 **public function** *next*() {  
 $item = $this->**items**[$this->**position**];  
 $this->**position**++;  
 **return** $item;  
 }  
 }  
  
 *// Класс-клиент, получает коллекцию и имеет метод для запуска итератора во ней.* **class** Client {  
 **public $collection**;  
  
 **public function** \_\_construct(Collection $collection) {  
 $this->**collection** = $collection;  
 }  
 **public function** iterate() {  
 $iterator = $this->**collection**->createIterator();  
  
 **while** ($iterator->issetNext()) {  
 $item = $iterator->next();  
 **print** $item->**name** . **' '** . $item->**price** . **'<br/>'**;  
 }  
 }  
 }  
  
 */\*\*  
 \* Вспомогательный класс, хранящий информацию об элементе объекта-коллекции.  
 \*/* **class** Item {  
 **public $price**;  
 **public $name**;  
  
 **public function** \_\_construct($name, $price) {  
 $this->**name** = $name;  
 $this->**price** = $price;  
 }  
 }  
}  
  
**namespace** {  
 **use** IteratorPattern\Client, IteratorPattern\CollectionA;  
  
 $client = **new** Client(**new** CollectionA());  
 $client->iterate();  
}

#### Command (Команда, Action/Действие, Transaction/Транзакция)

<http://habrahabr.ru/post/114455/>

**Команда – это инкапсуляция запроса в виде объекта.**

**Проблема**

Создать программу, в которой класс-отправитель Client отправляет команды-объекты классу-получателю Calculator.

Команда будет представлять собой объект с данными и операцию, которую необходимо осуществить с данными.

Команды с данными сохраняются для истории в пуле клиента.

Клиент может откатить команды на указанное количество шагов назад, при откате каждая команда выполняет обратное действие.

**Используется для**

Используется сервисами для выполнения и последующего логирования команды и ее параметров.

Либо для записи выполненной команды с возможностью отката ее назад.

**Особенности реализации**

Клиент выполняет ряд арифметических операций (команд). Каждая операция с параметрами записывается в объекте-клиенте. Клиент может откатить указанное число операций.

1. Client – класс клиента, вычисляющий результаты арифметических операций.

Хранит в себе данные

* 1. $calculator – объект калькулятора-исполнителя, который выполняет указанную команду
  2. $commands – массив команд с данными по каждой команде (оператор, операнд) для возможности отката
  3. $counter – счетчик количества выполненных команд

Выполняет действия

* 1. Calculate($operator, $operand) - выполняет команду и записывает ее в массив выполненных команд для возможности отката.
  2. Undo($number\_of\_commands) - отменяет указанное количество команд, используя информацию из массива выполненных команд.

1. Calculator –класс-получатель и исполнитель команды (оператор, операнд)

Хранит в себе данные

* 1. $result – результат выполнения операции (0 по умолчанию, пока ничего не выполнялось)

Выполняет действия

* 1. Operation($operator, $operand) – арифметическая операция, возвращается $result

1. CalculatorCommand extends Command – класс команды

Хранит в себе данные

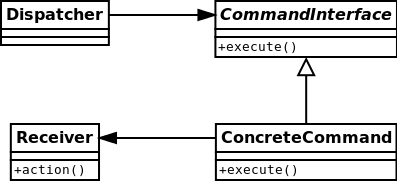
* 1. $calculator – объект калькулятора, который выполняет указанную операцию
  2. $operator – арифметический оператор
  3. $operand – арифметический операнд

Выполняет действия

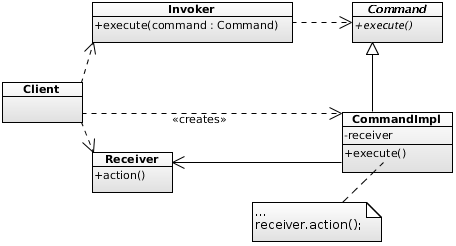
1. Execute() – вычисляет команду
2. Revert() – откатывает команду
3. GetBackwordOperator($operator) – метод-хелпер, возвращает оператор, обратный выполненному, для отката изменений

**Пример**

Упрощенный вариант



Классический вариант



**namespace** CommandPattern {  
 */\*\*  
 \* Абстрактый класс "команды"  
 \*/* **abstract class** Command {  
 **public abstract function** Execute();  
 **public abstract function** Revert();  
 }  
  
 */\*\*  
 \* Класс конкретной "команды"  
 \*/* **class** CalculatorCommand **extends** Command {  
 **public $calculator**;  
 **public $operator**;  
 **public $operand**;  
  
 **public function** \_\_construct($calculator, $operator, $operand) {  
 $this->**calculator** = $calculator;  
 $this->**operator** = $operator;  
 $this->**operand** = $operand;  
 }  
 *// Выполним команду.* **public function** Execute() {  
 $this->**calculator**->Operation($this->**operator**, $this->**operand**);  
 }  
 *// Откатим команду.* **public function** Revert() {  
 $this->**calculator**->Operation($this->GetBackwordOperator($this->**operator**), $this->**operand**);  
 }  
 *// Возвращает оператор, противоположный вызванному для последующего отката действия.* **private function** GetBackwordOperator($operator) {  
 **switch**($operator) {  
 **case '+'**:  
 $backwordOperator = **'-'**;  
 **break**;  
 **case '-'**:  
 $backwordOperator = **'+'**;  
 **break**;  
 **case '\*'**:  
 $backwordOperator = **'/'**;  
 **break**;  
 **case '/'**:  
 $backwordOperator = **'\*'**;  
 **break**;  
 **default** :  
 $backwordOperator = **' '**;  
 **break**;  
 }  
 **return** $backwordOperator;  
 }  
 }  
  
 */\*\*  
 \* Калькулятор - получатель и исполнитель "команд"  
 \*/* **class** Calculator {  
 *// Текущий результат выполнения команд. Здесь результат накапливается.* **private $result** = 0;  
 *// Метод выполняет вычисление, принимая тип оператора и аргумент.* **public function** Operation($operator, $operand) {  
 *// Сохраним исходное значение до выполнения операции в переменную.* $initial\_value = $this->**result**;  
 *// выбрать оператора для вычисления результата* **switch**($operator) {  
 **case '+'**:  
 $this->**result** += $operand;  
 **break**;  
 **case '-'**:  
 $this->**result** -= $operand;  
 **break**;  
 **case '\*'**:  
 $this->**result** \*= $operand;  
 **break**;  
 **case '/'**:  
 $this->**result** /= $operand;  
 **break**;  
 }  
 **print**(**"Current result =** $this->**result (after execution** $initial\_value $operator $operand**)<br/>"**);  
 }  
 }  
  
 */\*\*  
 \* Класс, вызывающий команды  
 \*/* **class** Client {  
 *// Этот класс будет получать команды на исполнение* **private $calculator**;  
 *// Массив операций* **private $commands** = **array**();  
 *// Cчетчик количества выполненных команд.* **private $counter** = 0;  
  
 **public function** \_\_construct() {  
 $this->**calculator** = **new** Calculator();  
 }  
 *// Выполняет команду и записывает ее в массив выполненных команд для возможности отката.* **public function** Calculate($operator, $operand) {  
 *// Создаем команду операции и выполняем её* $command = **new** CalculatorCommand($this->**calculator**, $operator, $operand);  
 $command->Execute();  
 *// Добавляем операцию к массиву операций и увеличиваем счетчик для понимания, сколько команд уже выполнено.* $this->**commands**[] = $command;  
 $this->**counter**++;  
 }  
 *// Отменяет указанное количество команд, используя информацию из массива выполненных команд.  
 // @param int $number\_of\_commands количество отменяемых операций, начиная с последней* **public function** Undo($number\_of\_commands) {  
 **print**(**"---- Undo** $number\_of\_commands **operations<br/>"**);  
 *// Делаем отмену операций* **for** ($i = 0; $i < $number\_of\_commands; $i++) {  
 **if** ($this->**counter** > 0) {  
 $this->**commands**[--$this->**counter**]->Revert();  
 }  
 }  
 }  
 }  
}  
  
**namespace** {  
 **use** CommandPattern\Client;  
  
 $client = **new** Client();  
  
 *// Произвольные команды* $client->Calculate(**'+'**, 100);  
 $client->Calculate(**'-'**, 50);  
 $client->Calculate(**'\*'**, 10);  
 $client->Calculate(**'/'**, 2);  
  
 *// Отменяем 2 последние команды* $client->Undo(2);  
}

#### Interpreter (Интерпретатор, Small Language)

<http://habrahabr.ru/post/136371/>

Данный шаблон – самый сложный и самый мощный, поскольку позволяет расширить синтаксис языка программирования для решения текущей задачи.

**Проблема**

Имеется часто встречающаяся задача, состоящая из подзадач. Основная задача (комбинация подзадач) подвержена изменениям. Подзадачи неизменны. Каждую подзадачу можно оформить как команду, а для интерпретации команды написать интерпретатор. Таким образом, мы получим более высокоуровневый язык программирования на базе текущего языка.

**Используется для**

1. Создания интерпретатора своего языка.
2. Пример: Ruby on Rails таким образом расширила Ruby.

Достоинством является возможность расширения синтаксиса и поддержки новых команд путем наследования.

**Особенности реализации**

Клиент строит предложение в виде синтаксического дерева.

Каждый узел дерева поддерживает метод Interpret.

Клиент вызывает метод interpret() для самого верхнего узла, который запускает вызовы Interpret по всем остальным узлам.

Создаем интерпретатор языка для базовых арифметических операций (число, переменная, суммирование, вычитание, умножение, деление)

Базовая операция имеет метод execute(), которая возвращает результат операции.

Операция (выражение) может быть неделимой (конечной, terminal): число, переменная.

->execute() для неделимой операции возвращает само число либо численное значение переменной.

И делимой, состоящей из неделимых операций (неконечная, nonterminal).

->execute() для делимой операции вызывает аналогичный метод ->execute() каждой своей части (рекурсивный вызов метода) и суммирует (вычитает и т.д.) части.

В итоге вначале мы составляем выражение:

$expression = new Divide(new Multiply(new Integer(10), new Sum(new Variable('a'), new Integer(3))), new Variable('b'));

Далее мы вызываем для него метод execute(), передав ему массив значений переменных. Метод рекурсивно обработает все операции и вернет результат.

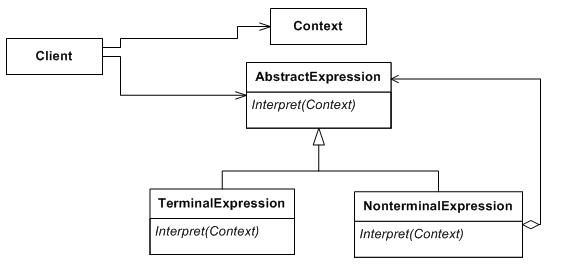
$result = $expression->execute(array('a' => 7, 'b' => 10));

**Недостатки**

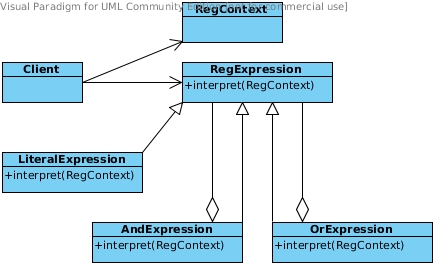
Затруднительно сопровождение грамматики с большим числом правил. Т.е. данный шаблон применим для относительно простых языков. Для создания языков со сложной грамматикой есть другие решения.

**Пример**

Классический малопонятный пример



Чуть более понятная схема



**namespace** InterpreterPattern {  
 */\*\*  
 \* Базовый интерфейс выражения.  
 \*/* **interface** MathExpression {  
 *// Переводит высокоуровневое выражение в базовый низкоуровневый язык,  
 // в данном случае PHP, и вычисляет его.* **public function** execute(**array** $values);  
 }  
  
 */\*\*  
 \* Конечное неделимое выражение простое фиксированное значение-литерал (цифра).  
 \* (terminal expression).  
 \*/* **class** Integer **implements** MathExpression {  
 **private $integer**;  
  
 **public function** \_\_construct($integer) {  
 **if** (*is\_int*($integer)) {  
 $this->**integer** = $integer;  
 }  
 **else** {  
 **throw new** \Exception(**'Only integer is acceptable!'**);  
 }  
 }  
 *// В кнкретном данном случае метод является заглушкой и просто возвращает сохраненный integer.  
 // Потому что во всех остальный случая мы работаем с массивом.* **public function** execute(**array** $integer) {  
 **return** $this->**integer**;  
 }  
 }  
  
 */\*\*  
 \* Конечное неделимое выражение-переменная.  
 \* (terminal expression).  
 \*/* **class** Variable **implements** MathExpression {  
 **private $variable**;  
  
 *// Сохраняет имя переменной.* **public function** \_\_construct($variable) {  
 $this->**variable** = $variable;  
 }  
 *// При вычислении принимает ассоциативный массив переменных.  
 // Возвращает из полученного массива (здесь может быть много переменных)  
 // значение элемента с ключом, соответствующим сохраненному имени переменной.* **public function** execute(**array** $values) {  
 **return isset**($values[$this->**variable**]) ? $values[$this->**variable**] : 0;  
 }  
 }  
  
 */\*\*  
 \* Неконечное делимое выражение, представляющее сумму двух конечных выражений.  
 \* (nonterminal expression).  
 \*/* **class** Sum **implements** MathExpression {  
 **private $a**;  
 **private $b**;  
  
 **public function** \_\_construct(MathExpression $a, MathExpression $b) {  
 $this->**a** = $a;  
 $this->**b** = $b;  
 }  
 *// В неконечных выражениях происходит рекурсивный вызов execute,  
 // таким образом данный метод будет выполняться до тех пор, пока  
 // он не достигнет наконец конечное выражение.  
 // В конечном выражении он трансформируется в конкретное значение.  
 // Таким образом, все выражение в итоге будет трансформировано  
 // в выражение с конечными значениями.* **public function** execute(**array** $values) {  
 **return** $this->**a**->execute($values) + $this->**b**->execute($values);  
 }  
 }  
  
 */\*\*  
 \* Неконечное делимое выражение, представляющее разность двух конечных выражений.  
 \* (nonterminal expression).  
 \*/* **class** Subtract **implements** MathExpression {  
 **private $a**;  
 **private $b**;  
  
 **public function** \_\_construct(MathExpression $a, MathExpression $b) {  
 $this->**a** = $a;  
 $this->**b** = $b;  
 }  
 *// Рекурсивный вызов execute.* **public function** execute(**array** $values) {  
 **return** $this->**a**->execute($values) - $this->**b**->execute($values);  
 }  
 }  
  
 */\*\*  
 \* Неконечное делимое выражение, представляющее произведение двух конечных выражений.  
 \* (nonterminal expression).  
 \*/* **class** Multiply **implements** MathExpression {  
 **private $a**;  
 **private $b**;  
  
 **public function** \_\_construct(MathExpression $a, MathExpression $b) {  
 $this->**a** = $a;  
 $this->**b** = $b;  
 }  
 *// Рекурсивный вызов execute.* **public function** execute(**array** $values) {  
 **return** $this->**a**->execute($values) \* $this->**b**->execute($values);  
 }  
 }  
  
 */\*\*  
 \* Неконечное делимое выражение, представляющее произведение двух конечных выражений.  
 \* (nonterminal expression).  
 \*/* **class** Divide **implements** MathExpression {  
 **private $a**;  
 **private $b**;  
  
 **public function** \_\_construct(MathExpression $a, MathExpression $b) {  
 $this->**a** = $a;  
 $this->**b** = $b;  
 }  
 *// Рекурсивный вызов execute.* **public function** execute(**array** $values) {  
 **return** $this->**a**->execute($values) / $this->**b**->execute($values);  
 }  
 }  
}  
  
**namespace** {  
 **use** InterpreterPattern\Integer,  
 InterpreterPattern\Variable,  
 InterpreterPattern\Sum,  
 InterpreterPattern\Subtract,  
 InterpreterPattern\Multiply,  
 InterpreterPattern\Divide;  
  
 *// Составляем выражение: expression = 10(a + 3) / b* $expression = **new** Divide(**new** Multiply(**new** Integer(10), **new** Sum(**new** Variable(**'a'**), **new** Integer(3))), **new** Variable(**'b'**));  
 *// Вычисляем выражение путем рекурсивного вызова execute для каждого узла выражения.  
 // При этом каждое неконечное выражение делегирует данный метод далее для своих узлов,  
 // конечное выражение-литерал просто вернет свое значение,  
 // а конечное выражение-переменная найдет и верент свое значение из переданного массива.* $result = $expression->execute(**array**(**'a'** => 7, **'b'** => 10));  
  
 **print** $result;  
}

#### (доп.) Null Object

#### (доп.) Servant (Слуга)

#### (доп.) Specification

#### (доп.) Simple Policy

#### (доп.) Event listener

#### (доп.) Single-serving visitor

#### (доп.) Hierarchical visitor

## GRASP паттерны (10 patterns)

<https://ru.wikipedia.org/wiki/GRASP>

<http://habrahabr.ru/post/92570/>

<http://www.slideshare.net/alexandernemanov/grasp-33677750>

**General Responsibility Assignment Software Patterns – паттерны для решения общих задач по назначению обязанностей классам и объектам**

Известно 10 паттернов. 9 паттернов изначально описаны в книге Крейга Лармана «Применение UML и шаблонов проектирования». 10 паттерн Don’t talk to strangers добавлен позднее.

### Information Expert (Информационный эксперт)

**Обязанность (метод) должна быть назначена объекту, который владеет максимумом необходимой информации для выполнения данной обязанности (метода). Самый важный шаблон.**

Повышает уровень инкапсуляции, уменьшает зависимости (зацепление), код можно использовать повторно, код читабелен и понятен.

Класс Продукт знает больше всего о продукте (свойства) и поэтому именно он обладает методами setProductPrice, setProductTitle и т.д.

### Creator (Создатель)

**Решает, кто должен создавать объекты и создает объекты. Класс A создает объекты класса B.**

**Уменьшает связность, класс создается внутри класса-создателя.**

Реализация – порождающий паттерн Абстрактная Фабрика (клиент создает объекты).

Альтернатива – паттерн Фабрика (класс создает объекты)

Класс A должен создавать объект B, если выполняется одно из условий:

1. **Класс A обладает данными инициализации для объектов B.**
2. Класс A содержит или агрегирует объекты B.
3. Класс A записывает экземпляры объектов B.
4. Класс A активно использует объекты B.

### Controller (Контроллер)

**Отвечает за обработку входных системных событий (сигналов).**

Смысл контроллера – получить некое событие и перенаправить работу с ним соответствующему классу.

Контроллер получает событие и реагирует на него, в зависимости от события, делегируя действие далее соответствующим классам.

Важно не перегружать контроллер дополнительным функционалом.

#### (подтип) Front Controller (Контролер Входа, Единая Точка Входа)

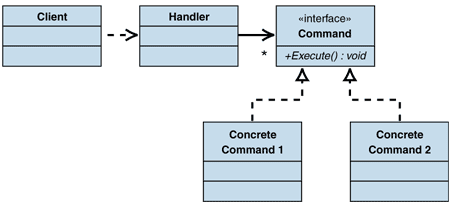
**Один контроллер обрабатывает все запросы к сайту.**

Т.е. паттерн описывает объект обработчик, через который проходят все запросы. После обработки запроса контроллер обращается к конкретному объекту для обработки конкретного типа запроса.

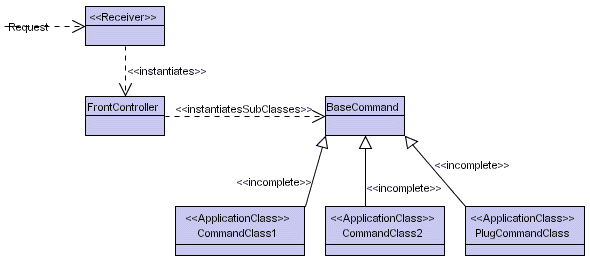
**Реализация в PHP**

1. Перенаправление всех запросов с помощью .htaccess на index.php
2. Далее единый контроллер в зависимости от типа запроса делегирует обработку запроса нужному объекту.

Базовая схема



Приближенная к реальности схема



#### (подтип) Page Controller (Контроллер страницы)

**Один контроллер на одну страницу сайта.**

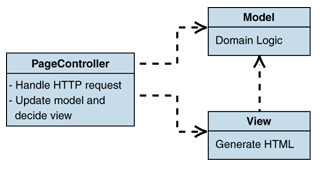
Так реализован PHPbb 3 – каждый урл/запрос обрабатывается отдельным скриптом (viewform.php, viewtopic.php etc) либо отдельным объектом.

По такому же принципу строятся статические сайты, когда за каждую страницу отвечает свой HTML файл.

Описание из книги: обрабатывает HTTP GET и POST, выбирает необходимые модель и вид.

Основной недостаток: дублирование кода и сложность внесения изменений при большом количестве страниц.

Основное достоинство при правильном проектировании: скорость работы и простота нахождения контроллера.



### Low Coupling (Низкая связанность, Слабая зависимость)

**Слабая зависимость данного класса от других классов или их методов.**

Проектируя классы, следует распределять их обязанности так, чтобы было как меньше входных параметров в виде объектов. Концепция black box.

1. В результате код меньше зависит от контекста, является модульным и может быть повторно использован.
2. Функционал класса/метода можно изучить без необходимости изучения дополнительных классов.
3. В результате классы легче изменять.

### High Cohesion (Высокое зацепление, single responsibility)

**Концепция Unix – делаем что-то одно, но делаем это очень хорошо. Обязанности класса хорошо согласованы между собой, и класс не выполняет огромных объемов работы.**

Если функционал растет, то класс разбивается на несколько классов/подклассов.

Если больше 20-ти методов в классе, разбиваем класс.

Если же связанность низка, класс выполняет слишком много обязанностей, то

1. трудность понимания.
2. сложность при повторном использовании.
3. сложность поддержки
4. частая необходимость вносить изменения

### Polymorphism (Полиморфизм)

**Единообразная обработка данных разных типов. Одна реализация с обобщенным интерфейсом.**

Интерфейс определяет единые базовые операции для всех типов. Реализация базируется на интерфейсе и определяет операции для конкретного типа.

Систему становится легко расширять.

### Pure Fabrication (Искусственная сущность, Чистая выдумка, Искусственный класс)

**Использование некоего придуманного объекта, не существующего в реальности, для обеспечения слабой зависимости и сильной связанности.**

Например, у нас есть класс A, который становится перегруженным функционалом и зависимостями.

Решением может быть создание искусственного класса B и вынесение ряда методов туда.

Пример.

Перегруженный методами класс Product, который помимо работы с продуктом должен сохранить его в базу. Данный функционал выносим в отдельный класс ProductDB. Данный класс не отражает реальный объект, однако позволяет разгрузить основной класс Product и уменьшить зависимости.

Не злоупотреблять, поскольку иначе система становится сложной для понимания.

### Indirection (Посредник, перенаправление)

**Обеспечить отсутствие прямого связывания, за счет присвоения обязанности по обеспечению связи между основными изменяемыми объектами промежуточным объектам-посредникам.**

Паттерн реализует Mediator из GOF.

Таким образом, есть основной объект А и основной объект Z. Мы создаем промежуточный объект M, в котором происходит взаимодействие A и B. За счет использования посредника объекты A и B остаются независимыми друг от друга – изменяя класс A нам не нужно изменять класс B.

Пример.

MVC – объект-контроллер ослабляет связь между данными и представлением, инкапсулируя в себе функционал, отвечающий за поведение системы.

### Protected Variations (Устойчивый к изменениям, Сокрытие реализации)

**Спроектировать систему так, чтобы изменения в одних объектах не влияли на другие объекты и не требовали изменений в других объектах.**

Паттерн дополняет паттерн Indirection.

Выносим все взаимодействие между объектами А и B в отдельный интерфейс. Класс-посредник С реализует данный интерфейс. Если изменился класс А или В – изменяем интерфейс и класс С.

### Don’t talk to strangers (Law of Demeter, Не разговаривайте с незнакомцами)

<http://serg-dobrinin.narod.ru/tutorial/oop.htm#_Toc182894190>

<http://citforum.ru/SE/project/pattern/p_1.shtml#3.2.5>

Предназначен, как и все остальные, для низкой связности и высокого зацепления.

Паттерн регламентирует, что метод нашего класса должен посылать сообщения только прямым объектам и никому более:

1. Текущему объекту
2. Устанавливать в виде свойства текущего объекта
3. Методам текущего объекта в виде параметра
4. Объекту, созданному внутри данного метода

Все остальные объекты – непрямые. Им посылать сообщения нельзя, так как это сразу создает зависимости.

## Enterprise Integration Patterns (65 patterns)

<http://en.wikipedia.org/wiki/Enterprise_Integration_Patterns>

<http://citforum.ru/SE/project/pattern/> - Паттерны интеграции корпоративных информационных систем

Это 65 паттернов из книги “Enterprise Integration Patterns” 2003 г.

Продвинутый уровень для архитекторов.

## System Architecture Patterns (26 patterns)

<http://citforum.ru/SE/project/pattern/> - Архитектурные системные паттерны

## Antipatterns

<https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D1%82%D0%B8%D0%BF%D0%B0%D1%82%D1%82%D0%B5%D1%80%D0%BD>

<http://lurkmore.to/%D0%90%D0%BD%D1%82%D0%B8-%D0%BF%D0%B0%D1%82%D1%82%D0%B5%D1%80%D0%BD>

**Это наиболее часто внедряемые ошибочные решения. Это то, как поступать не надо, но как все равно все поступают.**

Их разделяют на ООП-антипаттерны, антипаттерны программирования, методологии и еще много всяких других антипаттернов. Ниже этого разделения нет, просто приведены наиболее частые антипаттерны, которые вообще имеют отношение к разработке.

1. **Hard code** — про него ещё говорят «прибито гвоздями». Код сильно привязан к окружению, за счет чего не может быть переиспользован. Изменить его или отвязать от окружения требует больших усилий.
2. **Soft code** — почти то же, что мягкий стул, или, техническим языком, код, конфигурируемый настолько гибко (и запутано), что лучше уж это был бы hard code. Избыточная конфигурация всего и вся.
3. **Magic numbers** — то, на чём нередко строится hard code. Всяческие числовые значения в программе без пояснения их физического (или любого иного) смысла.
4. **Magic strings** — как магические числа, но только это строки.
5. **Accidental complexity (ненужная сложность, индусский код, KISS)** — делать сложно то, что можно сделать просто. Как правило, программисту хочется побыстрее применить всё, что он изучил (и воон ту новую библиотеку для обмена данными по протоколу XXX), а ещё заполучить по результатам проекта сразу много новых строчек в резюме. Если это желание побеждает доводы рассудка, то в проекте появляется accidental complexity.
6. **Code spaghetti** — это тот самый код, который вам однажды дают сопровождать и после пяти минут причёсывания вставших дыбом волос вы молча закрываете редактор и открываете браузер на поисках вакансии в более другой фирме, где не практикуют такого жестокого обращения с программистами. Характеризуется тем, что в одном куске кода описывается куча перемешанных до полного непонимания сущностей, которые по-хорошему необходимо разделять. Например, HTML теги в PHP коде вместо выноса всей верстки в шаблоны.
7. **Copy and Paste programming** - копирование (и лёгкая модификация) существующего кода вместо создания общих решений
8. **God Object** — небольшая часть кода, где сконцентрировано всё. Буквально всё. Возможно, там даже прячется немаленьких размеров галактика. Весь прочий код программы — исключительно декорация вокруг Божественного Объекта.
9. **Detonator** — очень распространён, но редко обнаруживаем. Наглядный пример: использование в вычислениях с датами поля из двух цифр. Бомба заложена, и детонатор рано или поздно сработает
10. **Absolver (не виноватая я**) —код, написанный бывшими сотрудниками компании. В таком коде заключено столько старых проблем, что теперешние сотрудники могут защитить свои наработки от обвинений, утверждая, что именно чужой код — причина всех возникающих ошибок. Также известен под именем Это-Не-Моя-Правка.
11. **Stake (исторический вклад)** — паттерн имеет место в программе, написанной сотрудником, который впоследствии получил продвижение по службе. Несмотря на изобилие ошибок в программе, вклад этого сотрудника слишком велик, чтобы позволить кому-либо начать переписывать код, поскольку он является апофеозом технических достижений этого товарища.
12. **Private Ryan (спасти рядового Райана)** — паттерн, в котором доступ к полю получить в принципе можно, но для этого надо снаряжать экспедицию, сопряженную со множеством опасностей.
13. **Mutilation (членовредительство)** - излишнее "затачивание" объекта под определенную очень узкую задачу таким образом, что он не способен будет работать с никакими иными, пусть и очень схожими задачами.
14. **Инверсия абстракции (Abstraction inversion)** - сокрытие части функциональности от внешнего использования, в надежде на то, что никто не будет его использовать
15. **Big Ball of Mud** (большая куча грязи) – система с нераспознаваемой структурой.

## Drupal 7 Patterns

<http://kostya.in.ua/content/patterny-proektirovaniya-v-drupal>

Drupal 7 использует идею, стоящую за конкретным паттерном, а не его ООП реализацию.

1. Front Controller – index.php – т.е. одна функция, отвечающая за обработку всех запросов
2. Observer - что-то в системе случается, и в ответ выполняется набор некоторых операций, которые могут быть динамически добавлены в систему (методы-наблюдатели – это по сути хуки: hook\_node\_load(), hook\_user\_login() etc)
3. Visitor – к базовому модулю подключается дополнительный модуль, который изменяет некоторые данные базового модуля. Каждая часть ядра предусматривает опциональное включение нового модуля, который изменит поведение данной части. Модулей может быть много.
4. Factory – слой базы данных, где метод типа db\_insert() возвращает объект DatabaseConnection()
5. Composite - Render API, где в итоге собирается объект страницы, состоящий из композитных узлов (блоки, регионы) и узлов-листьев (логотипа, мессаджа в футере и т.д.). Drupal идет по массивы и рекурсивно отрисовывает элементы, используя ‘#theme’-функции каждого узла.

# Refactoring

<https://ru.wikipedia.org/wiki/%D0%A0%D0%B5%D1%84%D0%B0%D0%BA%D1%82%D0%BE%D1%80%D0%B8%D0%BD%D0%B3>

Золотой принцип рефакторинга – если нашли тормозящую функцию, то не правим ее (потому что она может вызываться еще в миллионе мест и иметь кучу зависимостей), а делаем на ее основе свою функцию и заменяем тормозящую функцию нашей функцией.