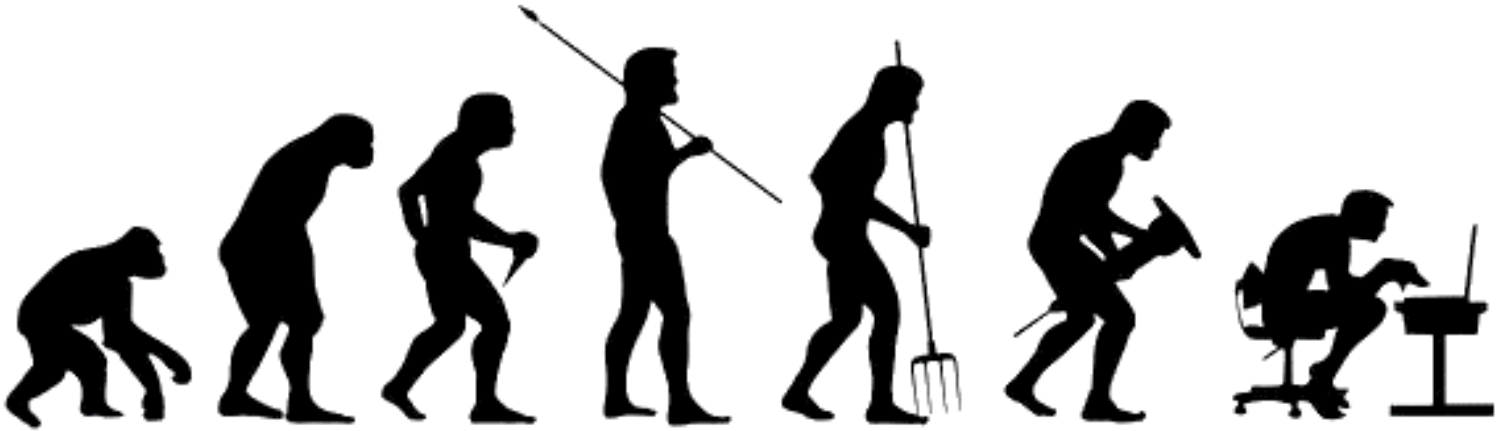

# Информатика

## Паттерны проектирования

© Марченко Антон Александрович 2018 г.
Абрамский Михаил Михайлович

# От двоичных кодов до паттернов проектирования



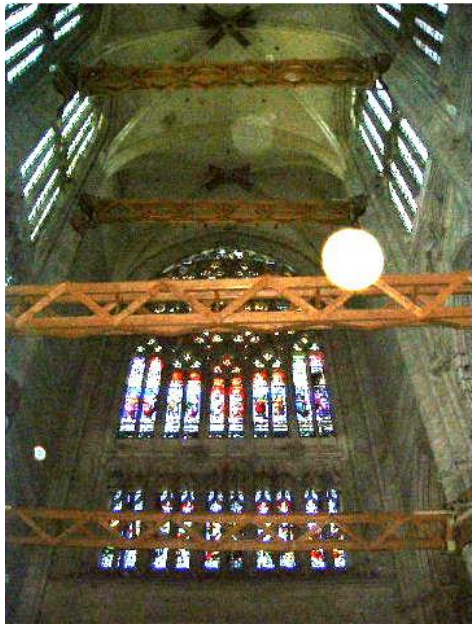
Новый уровень абстракции, существенный скачок в разработке ПО, переход от ремесла к инженерии [Ермаков]

# Решение проблем

- **Паттерны**
- **Проектирования**

"You wanted a banana but what you got was a gorilla holding the banana and the entire jungle."

-- Joe Armstrong



# Что такое паттерн?

- Модель, схема или образ некоторого повторяющегося процесса или явления
- Термин также встречается в психологии, физике, музыке…
  - пример: паттерны поведения – утренние умывания, приветствия, способ отвечать на звонки и переходить проезжую часть

# Паттерн vs шаблон

- Шаблон (template)
  - макет/заготовка
- Паттерн (pattern)
  - выявленная закономерность
  - образец реакции

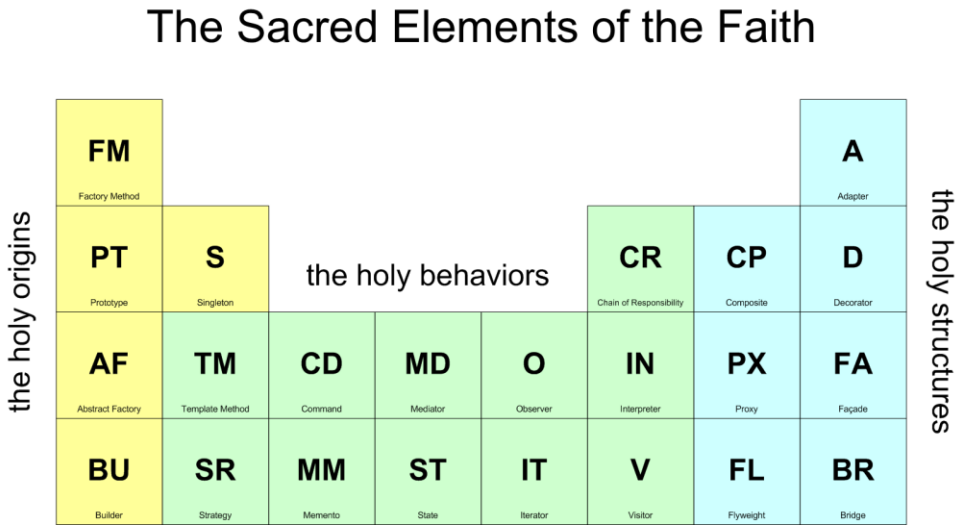
# Дисклеймер

- Паттерны – не серебряная пуля
- Паттерны – не строгие указания
- Должны помогать, а не мешать
- Знать обязательно, применять обосновано

# Паттерны проектирования

## Стадии изучения паттернов:

1. **Отторжение**
   – Что это? Да зачем они?
2. **Заинтересованность**
   – Ух ты! Где бы применить?
3. **Поклонение**
   – Паттерны сила! Везде применять!
4. **Отрезвление**
   – Паттерны – классная штука! Но со временем рефакторинг не выглядит классным
5. **Понимание**
   – Паттерны – хорошо, но своя голова – на порядок лучше!



# И вообще...

| OO pattern/principle | FP equivalent |
| --- | --- |
| • Single Responsibility Principle | • Functions |
| • Open/Closed principle | • Functions |
| • Dependency Inversion Principle | • Functions, also |
| • Interface Segregation Principle | • Functions |
| • Factory pattern | • You will be assimilated! |
| • Strategy pattern | • Functions again |
| • Decorator pattern | • Functions |
| • Visitor pattern | • Resistance is futile! |

Seriously, FP patterns are different

# Немного истории

- Паттерны проектирования начинались не с программной инженерии и информатики…
- А с архитектуры
  - Той самой, что имеет дело со зданиями и строительством

# Кристофер Александр

- Архитектор Christopher Alexander написал книгу о типовых решениях архитектуры
  - «как сделать комнаты светлыми»

# Определение паттерна

«каждый паттерн описывает некую повторяющуюся проблему и ключ к её разгадке, причем таким образом, что этим ключом можно пользоваться при решении самых разнообразных задач»

Кристофер Александер, 1977

# К разработке ПО

- В 1987 году Кент Бек (XP, TDD) и Уорд Каннингем (wiki) применили подход Кристофера Александра
  - разработали паттерны для разработки графических оболочек на Smalltalk
  - разработали библиотеку, написали пару-тройку статей
  - но их идеи в основном остались внутри компании

# Эрих Гамма

- Прочитал статью Бека и Уорда и загорелся идеей
- В 1988 году начал писать диссертацию
- Обобщил принципы на разработку программ в целом
- Диссертация, впрочем, не нашла широкого распространения

# Тем временем…

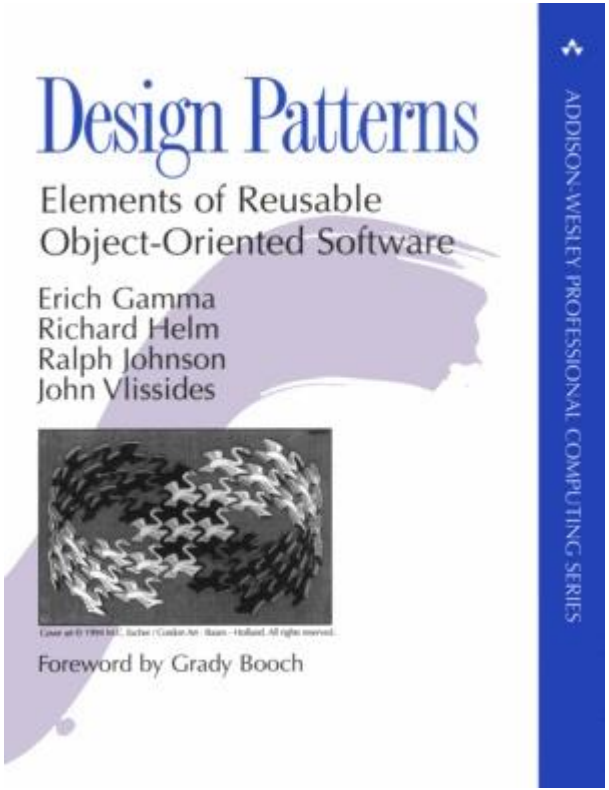
- В 1989-1991 годах Джеймс Коплиен обобщал свои знания в области программирования на C++
- 1991 “Advanced C++ Idioms”
- Идиомы – почти паттерны

# 1991 – 1994 годы

- Гамма закончил работать в Цюрихе, переехал в США
- Вместе в Хелмом, Джонсоном и Влиссидесом опубликовал Design Patterns – Elements of Reusable Object-Oriented Software

# Gang of four Design Patterns

- Книга стала бомбой
  - разорвавшимся снарядом массового поражения
- До сих пор она одна из самых читаемых в сфере



# Более точное определение DP

- ***Паттерны проектирования – эффективные*** *способы решения* ***характерных*** *задач проектирования, в частности* ***проектирования*** *компьютерных программ.*

- Паттерн не является законченным шаблоном проекта, который может быть прямо преобразован в код, скорее это описание или образец для того, как решить задачу, таким образом чтобы это можно было использовать в различных ситуациях.

# От идиом до паттернов

- **Idiom** – для конкретной задачи на конкретном языке
- **Specific Design** – независимо от ЯП
- **Standard Design** – обобщение задачи
- **Design Pattern** – абстрактное решение целого класса задач

# Путь к пониманию паттернов

- 1997 Крэг Ларман. Применение UML и паттернов
- General Responsibility Assignment Software Patterns
- *Вместе с принципами ООП и SOLID помогают вывести GoF паттерны*

# GRASP

1. **Information Expert** – в системе должна аккумулироваться, рассчитываться и т.п. вся необходимая информация. Нужно назначить такую обязанность некоторому классу или нескольким классам.
2. **Creator** – решение проблемы создания экземпляров. Назначить классу B обязанность создавать объекты класса A.
3. **Controller** – решение для обработки входных событий. Делегировать обязанность по обработке контроллеру, который не относится к интерфейсу. Для различных use-case’ов разные контроллеры.

# GRASP

- **Low Coupling** – низкая связанность при создании экземпляра и связывании его с другим классом. Распределение обязанности между объектами. Объекты знают друг о друге как можно меньше.
- **High Cohesion** – сильное функциональное зацепление, сфокусированность обязанностей класса. Отказ от выполнение лишней или разнородной работы.
- **Polymorphism** – обеспечение заменяемость компонент. Распределение обязанностей с помощью полиморфных операций.

# GRASP

- **Pure Fabrication** – «чистая выдумка», делегирование группы обязанностей с сильным зацеплением объекту не существующему в предметной области (синтезировать искуственную сущность)
- **Indirection** – делегировать обеспечение связи между сущностями промежуточному объекту
- **Protected Variation** – обеспечить устойчивую работу системы при возможных изменениях. Локализовать точки возможных изменений и распределить обязанности.

# Польза

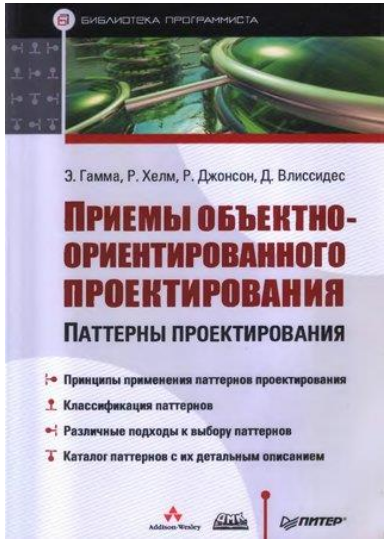
Польза от паттернов:

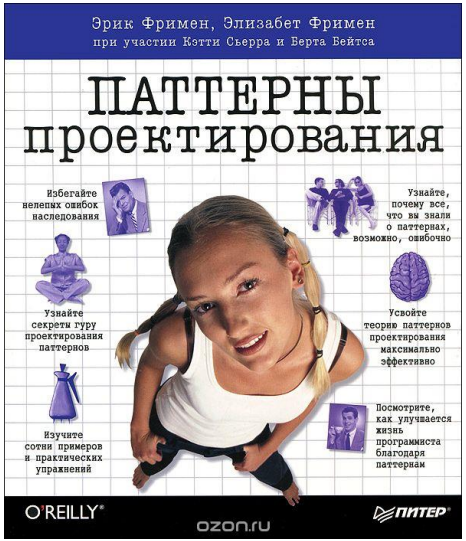
1. описывается решение целого класса абстрактных проблем
2. унифицируется терминология, названия модулей и элементов проекта
3. позволяют применять и использовать удачные решения снова и снова
4. в отличие от идиом, паттерны независимы от применяемого языка программирования

# Недостатки

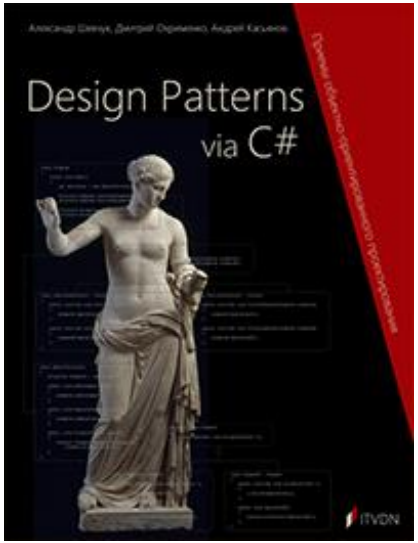
- корпоративное требование соблюдение паттернов (особенно внутрикорпоративных) может консервировать громоздкие и малоэффективные системы при возможном наличии более современного элегантного и эффективного решения
- если паттернов много, то исполнители игнорируют паттерны и всю систему
- слепое применение паттернов замедляет развитие программиста. Нужно мыслить критически, творчески и рефлексировать

# Литература по паттернам







# Видео лекции по паттернам

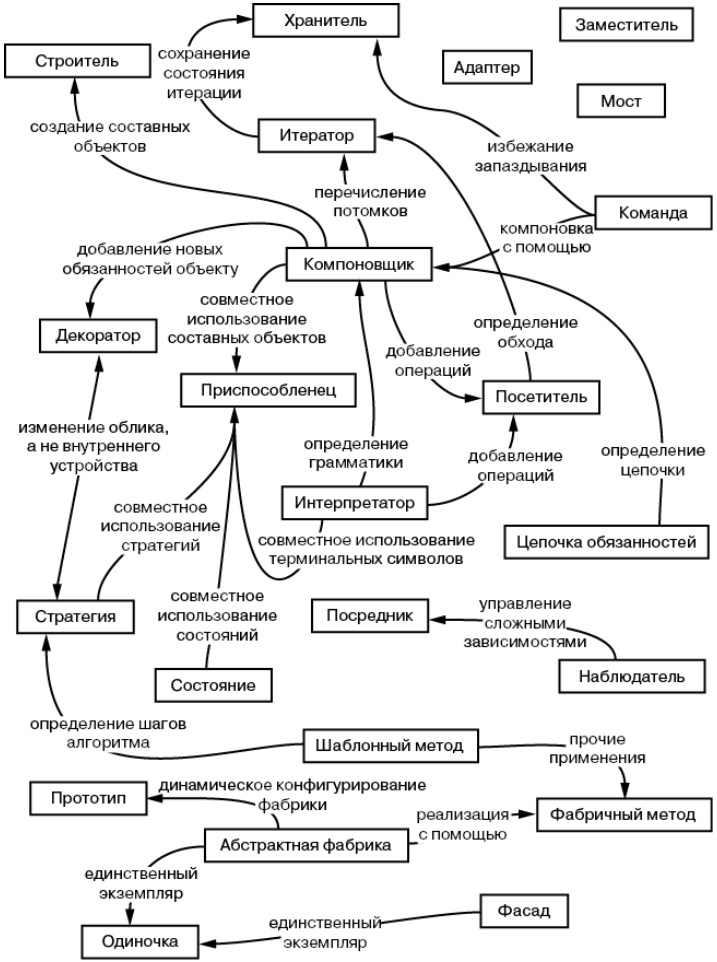
- Сергей Немчинский
  https://www.youtube.com/user/pro100fox2
- Видеокурс ITVDN по паттернам
  https://www.youtube.com/user/CBSystematicsTV
  https://itvdn.com/ru

*но, imho, книжки лучше*

# Статьи по паттернам

- Обзор паттернов проектирования
  http://citforum.ru/SE/project/pattern/
- Конспект лекций Немчинского по паттернам
  http://www.slideshare.net/SergeyNemchinskiy/ss-27467182

# 23 GoF паттерна



# Представление паттерна

- **Название**
  Устойчивые названия улучшают коммуникацию и понимание внутри команды.

- **Проблема**
  Задача, контекст и рамки применения.

- **Решение**
  Пример, не зависящий от языка программирования. Демонстрируется основная идея, абстракции и взаимосвязи, а не детали реализации.

- **Последствия**
  Не бывает идеальных решений. Где-то выигрываем, где-то теряем. Нужно выбирать подходящие решения.

# Типы паттернов

- *Порождающие (Creational)*
  - создание экземпляров
- *Структурные (Structural)*
  - композиция интерфейсов, классов, объектов
- *Поведенческие (Behavioral)*
  - коммуникация между объектами

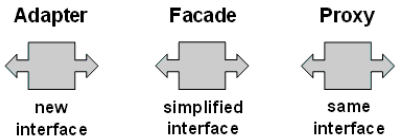
# Типы и уровень применения

| Уровень \ Цель | Порождающие паттерны | Структурные паттерны | Паттерны поведения |
|---|---|---|---|
| **Класс** | Фабричный метод | Адаптер (класса) | Интерпретатор |
| | | | Шаблонный метод |
| **Объект** | Абстрактная фабрика | Адаптер (объекта) | Итератор |
| | Одиночка | Декоратор | Команда |
| | Прототип | Заместитель | Наблюдатель |
| | Строитель | Компоновщик | Посетитель |
| | | Мост | Посредник |
| | | Приспособленец | Состояние |
| | | Фасад | Стратегия |
| | | | Хранитель |
| | | | Цепочка обязанностей |

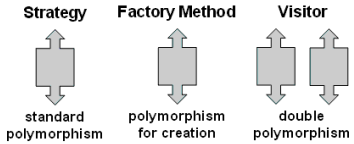
# Группировка по структуре

http://www.vincehuston.org/dp/

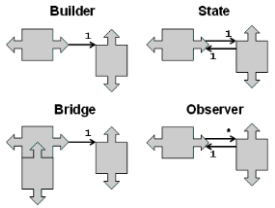
- Оболочки, делегации, а



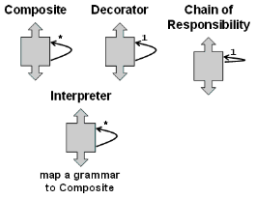

- Иерархии наследования



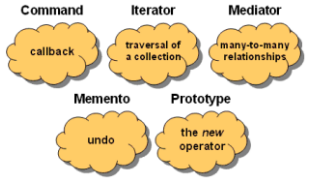

- Оболочка над иерархией



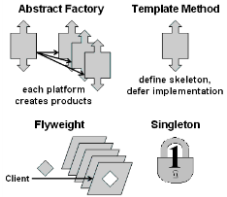

- Рекурсивная композиция




- Отдельный объект




- Разное



# Порождающие паттерны

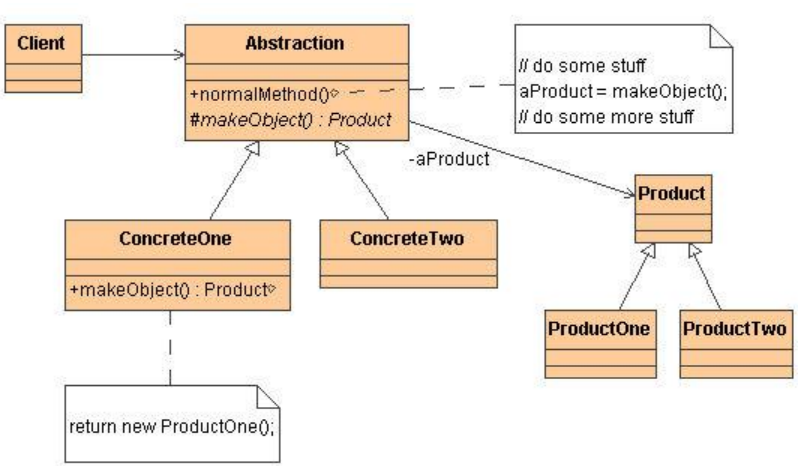
- *Синглтон* (Singleton)
- *Фабричный метод* (Factory method)
- *Абстрактная фабрика* (Abstract factory)
- *Прототип* (Prototype)
- *Строитель* (Builder)

# Синглтон

- гарантировать создание только одного экземпляра класса и предоставить к нему точку доступа
  - Анти паттерн?
  - Как реализовать потокобезопасный синглтон?
- создать класс с приватным конструктором, ссылкой на экземпляр и публичным методом получения экземпляра
- лучше вместе с отложенной инициализацией

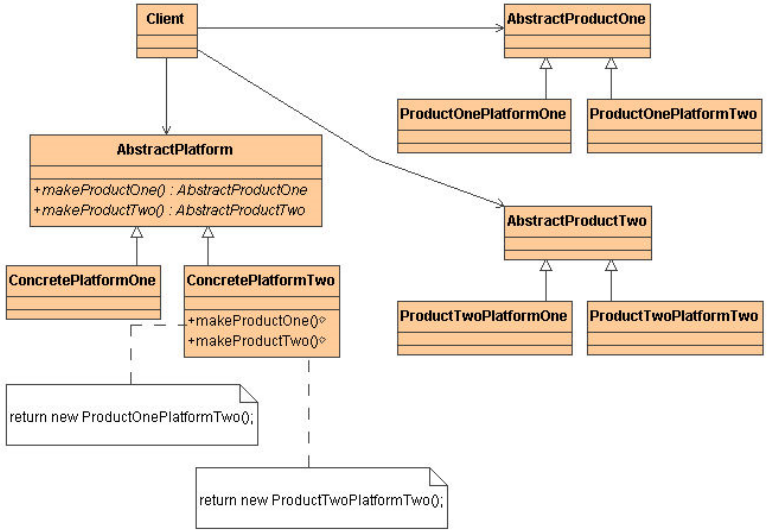
# Фабричный метод

- определить интерфейс для создания объекта, пусть подклассы решают экземпляры какого класса создавать



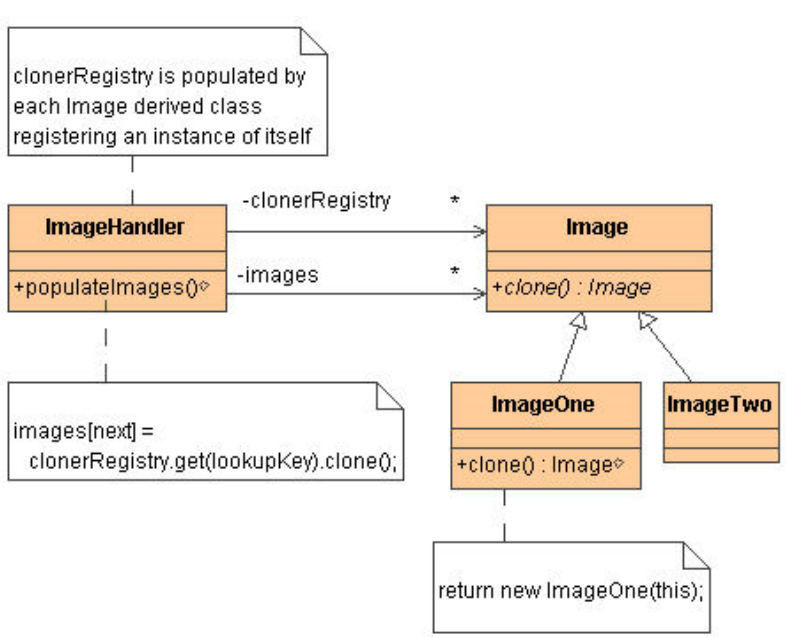
# Абстрактная фабрика

- Для независимости как от процесса создания новых объектов, так и от их типов
  - Игра-стратегия, несколько империй, разные типы войск, разные армии
- Создать семейство логически связанных объектов разных классов (не связанных)
- Сгруппировать семейство объектов с общей целью
- http://java.globinch.com/patterns/design-patterns/factory-design-patterns-and-open-closed principle-ocp-in-solid/
- http://www.dofactory.com/net/abstract-factory-design-pattern



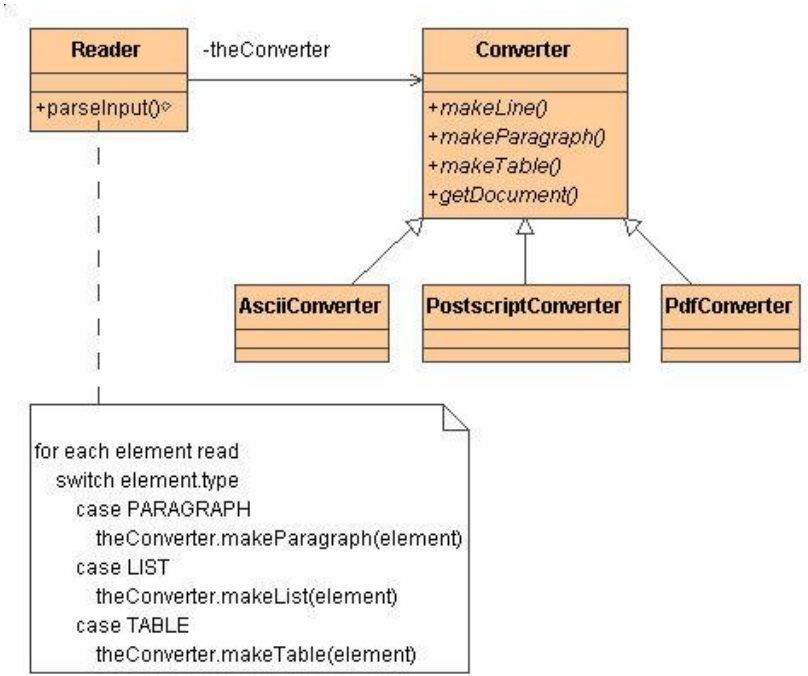
# Прототип

- Система не должна зависеть от создания, компоновки и представления объектов
- Объекты – клоны прототипа
  - как объекты в JS



# Строитель

- Создание сложного объекта не должно зависеть от того, из каких частей он состоит и как они стыкуются между собой
- Логика построения сложная и/или имеет зависимости

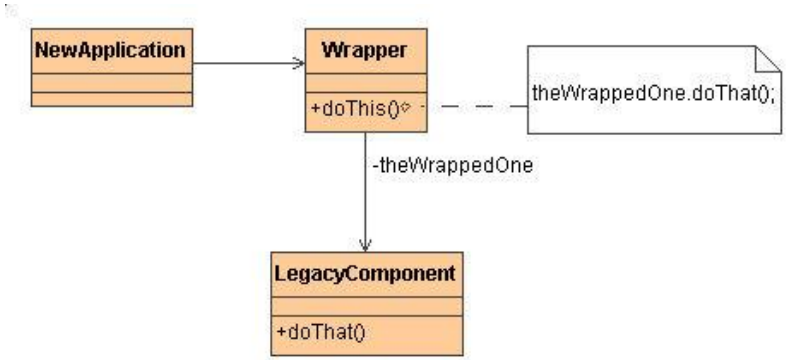


# Структурные паттерны

- *Адаптер* (Adapter)
- *Фасад* (Façade)
- *Декоратор* (Decorator)
- *Заместитель* (Proxy)
- *Приспособленец* (Flyweight)
- *Мост* (Bridge)
- *Компоновщик* (Composite)

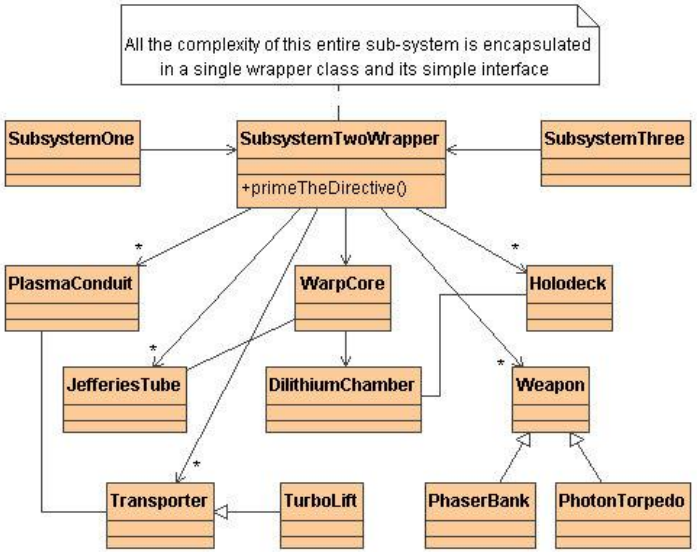
# Адаптер

- Обеспечить взаимодействие несовместимых интерфейсов.
- Единый интерфейс для компонент с разными интерфейсами
- Адаптер – «переходник»
- *Не изменяет поведения*



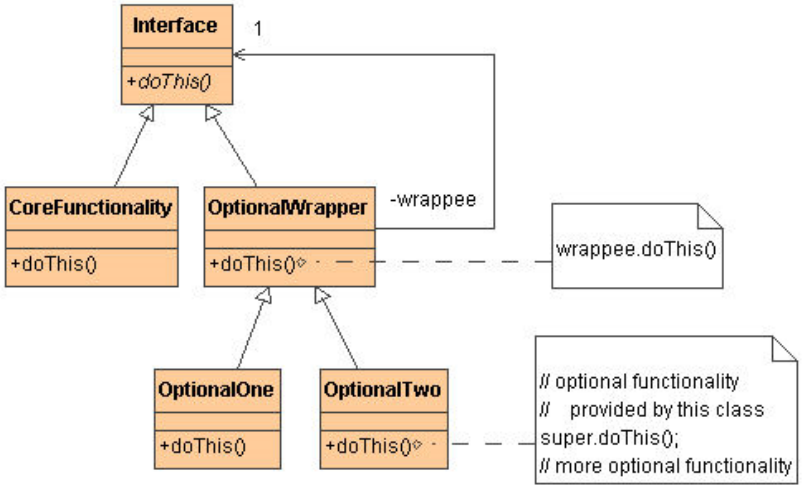

# Фасад

- Упрощение доступа к сложным системам, повышение уровня абстракции. Связь с изменчивой под-системой.
- Упрощает интерфейс, не изменяет функционал




# Декоратор

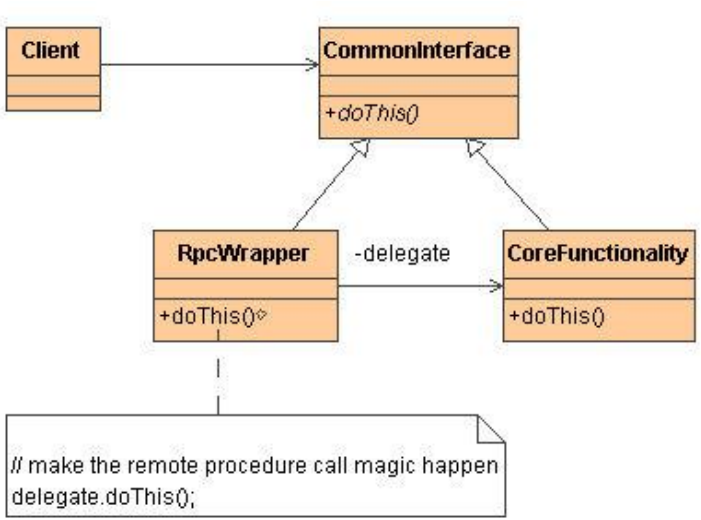
- Добавить функционал объекту, а не классу.
- Можно декорировать цепочкой.



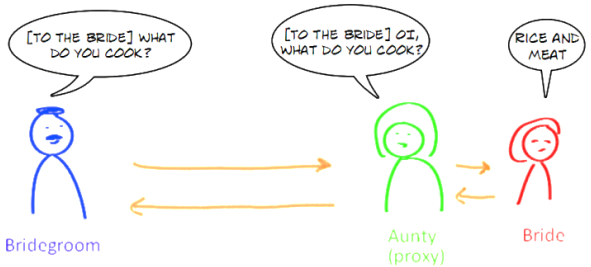

- Пример: разновидности пиццы с разным набором ингредиентов и разной стоимостью

# Заместитель

- Управление доступом к объекту, предотвращение прямого доступа к объекту, создание громоздских объектов.

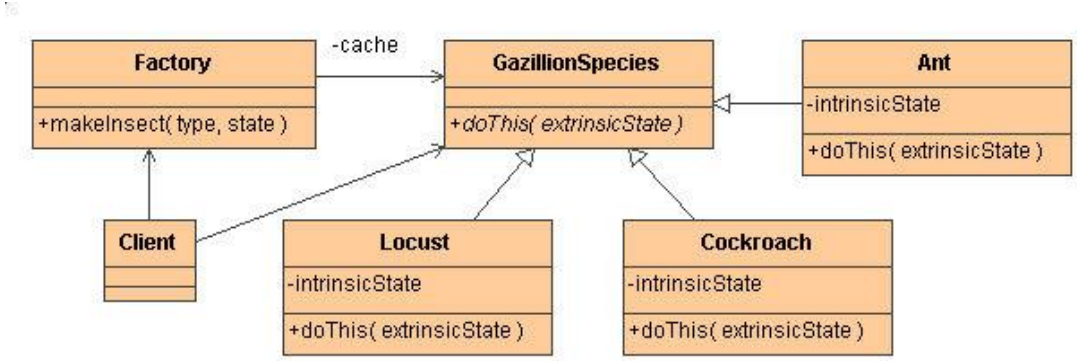



- Не изменяет реализацию и интерфейс
- см. виртуальный заместитель, защитник, кеширующий



# Приспособленец

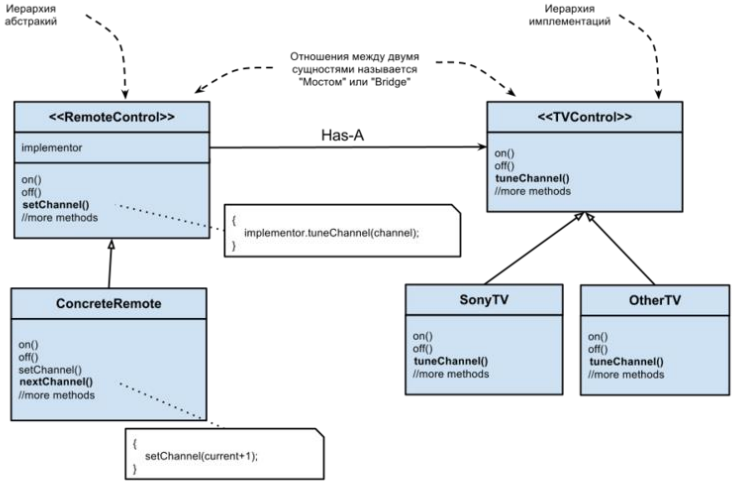
- Поддержка множества мелких объектов.




- Пример: структуры данных для графического представления символов в текстовом редакторе
- https://tamasgyorfi.net/2016/05/30/design-patterns-in-the-real-world-flyweight/

# Мост

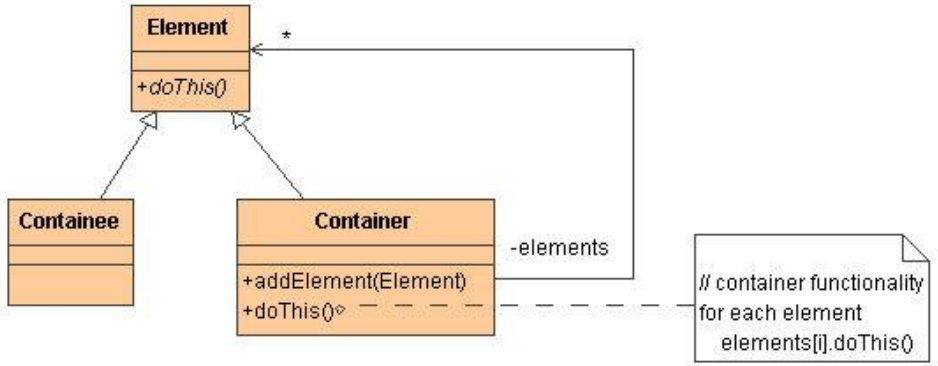
- Отделить абстракцию от реализации, чтобы и то и другое можно было изменять независимо.
- Поместить абстракцию и реализацию в отдельные иерархии




- Пример: графические фигуры с отрисовкой. Можно добавлять фигуры и способы отрисовки.
- https://habrahabr.ru/post/138585/

# Компоновщик

- Обрабатывать группу или композицию структур объектов одновременно.
- Объединить объекты в древовидную структуру как «часть-целое»

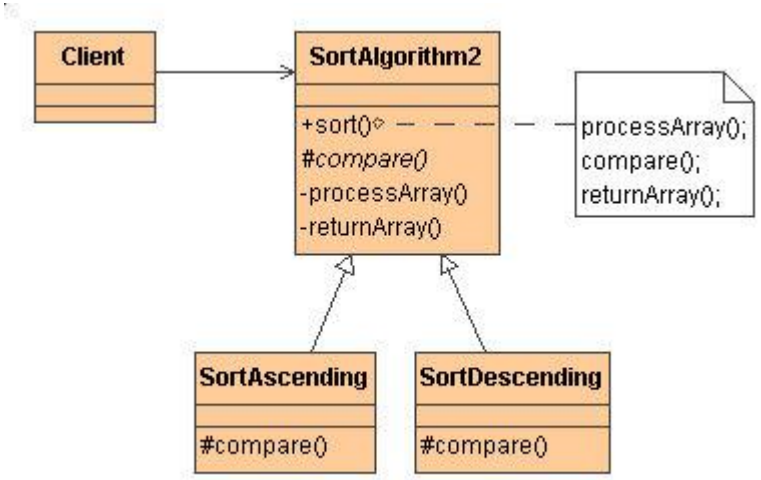



# Поведенческие паттерны

- *Шаблонный метод* (Template method)
- *Стратегия* (Strategy)
- *Итератор* (Iterator)
- *Команда* (Command)
- *Посредник* (Mediator)
- *Наблюдатель* (Observer)
- *Цепочка обязанностей* (Chain of responsibility)
- *Хранитель* (Memento)
- *Состояние* (State)
- *Посетитель* (Visitor)
- *Интерпретатор* (Interpreter)

# Шаблонный метод

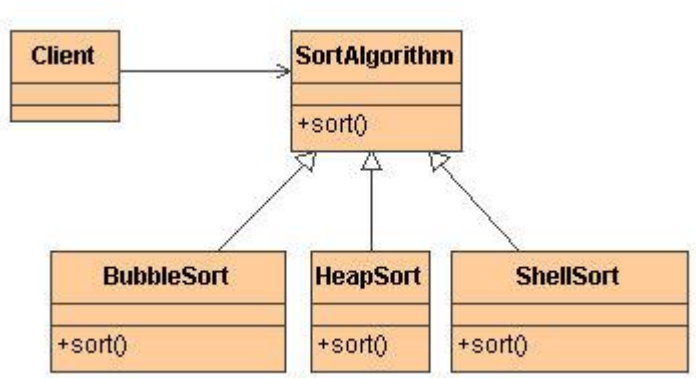
- Дать возможность переопределять шаги в алгоритме у подклассов, не изменяя общей структуры алгоритма




- Абстрактный класс алгоритма с реализацией виртуальных методов в конкретных классах

# Стратегия

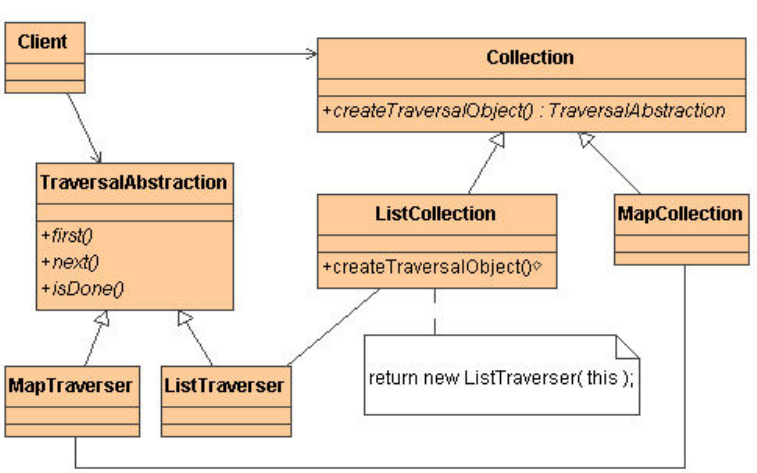
- Изменяемые, но надёжные алгоритмы или стратегии.




- Замена стратегии во время исполнения
- Агрегация стратегии у клиента

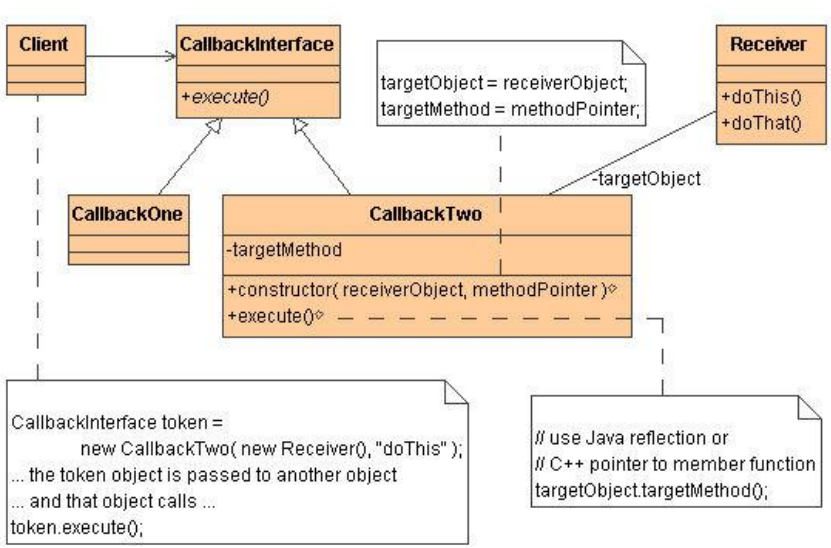
# Итератор

- Доступ к составным частям сложного объекта, не раскрывая внутренней структуры элементов, с возможностью перебирать элементы по-разному в зависимости от задачи.



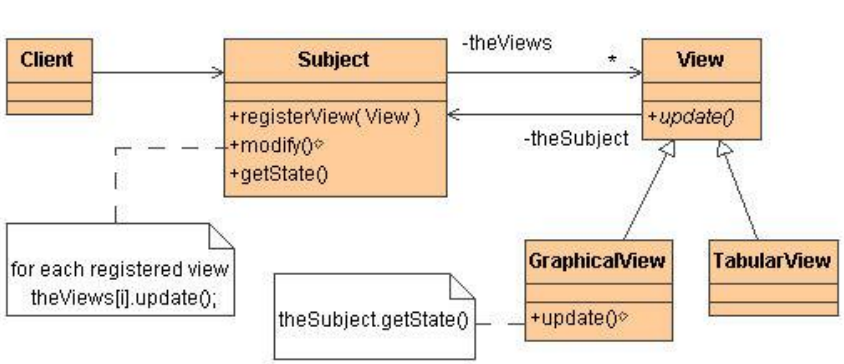
# Команда

- Послать запрос, не зная какая операция запрошена и кто получатель.
- Callback
- Пример: телевизор, пульт, команда управления телевизором



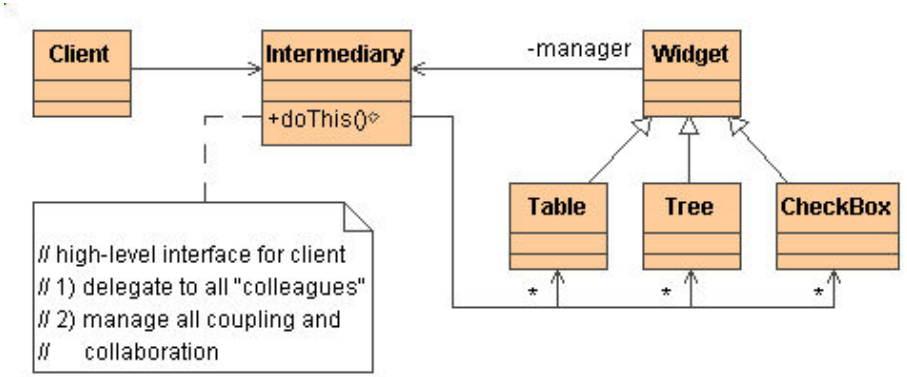
# Наблюдатель

- Подписка объекта на изменения состояний объектов при низком связывании.
- Делегаты, события
- https://docs.microsoft.com/en-us/dotnet/standard/events/observer-design-pattern



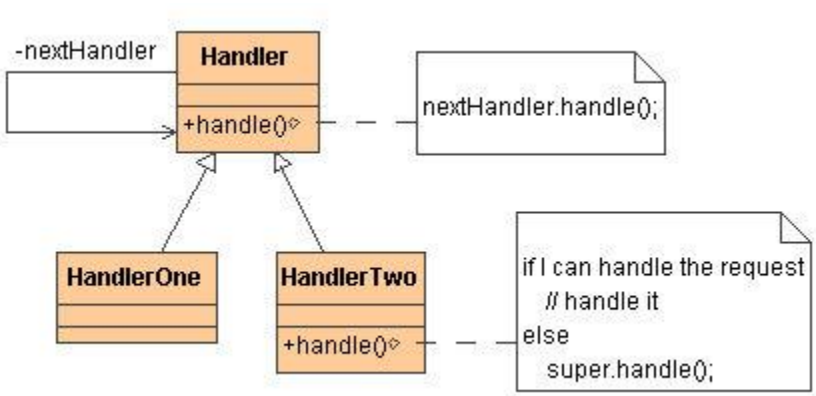
# Посредник

- Взаимодействие множества объектов при слабой связанности и явных ссылок.
- Может реализовываться наблюдателем
- Пример: лампа и кнопка через контроллер
- Повышает непрозрачность
- https://github.com/jbogard/MediatR



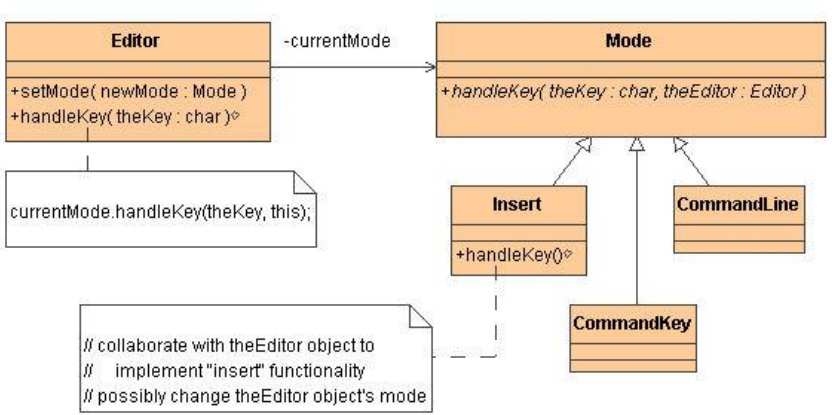
# Цепочка обязанностей

- Обработка запросов несколькими объектами, которые выбирают, либо обработать запрос, либо передать его дальше по цепочке.



# Состояние

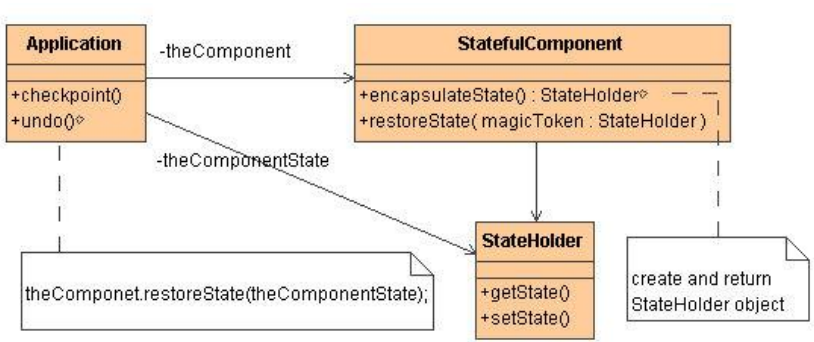
- Изменять поведение в зависимости от внутреннего состояния объекта.




- Конечный автомат (Finite State Machine)
- https://habrahabr.ru/post/341134/

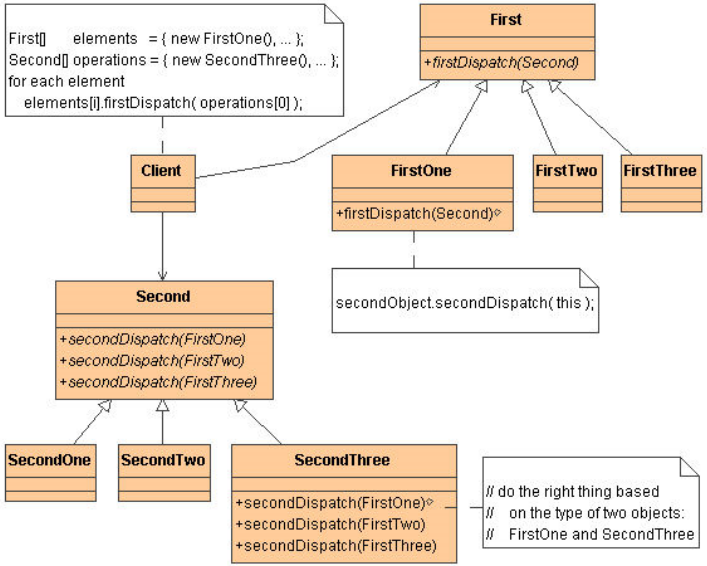
# Хранитель

- Фиксация состояний для реализации механизма отката к прошлому состоянию
- «Бэкап состояния системы»



# Посетитель

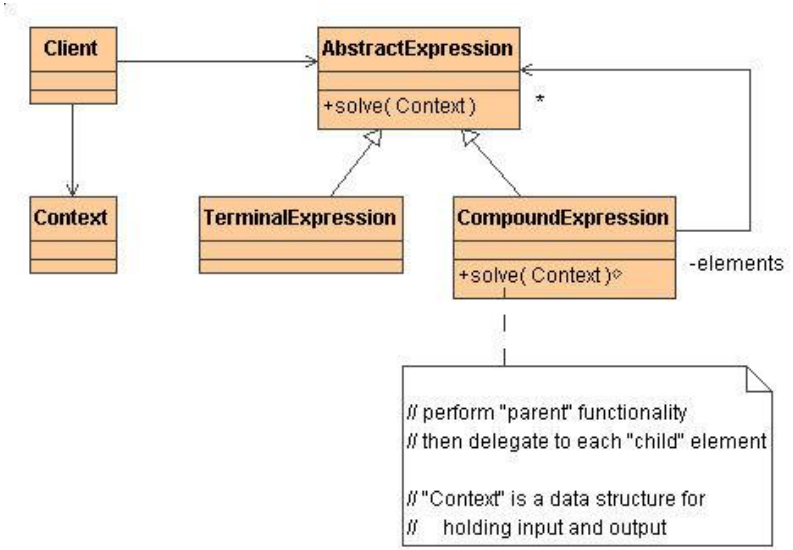
- Выполнять операции над каждым объектом некоторой структуры. Определять новую операцию, не изменяя классы объектов-частей.




- Хорошо подходит для разбора древовидных структур

# Интерпретатор

- Решать часто встречающуюся, подверженную изменениям задачу.
- Создаётся интерпретатор, решающий задачу
- Свободная монада
- https://www.youtube.com/watch?v=hmX2s3pe_qk



# Антипаттерны

anti-patterns, pitfalls

- Наиболее часто внедряемые неудачные решения проблем
- Изучаются для избегания повторения проблем в будущем

# Общие анти-паттерны

- *Копи-паста* (Copy&Paste programming)
- *Золотой молот* (Golden Hammer)
- *Фактор невероятности* (Improbability factor)
- *Преждевременная и микро оптимизация* (Premature and micro optimization)
- *Изобретение колеса* (иногда квадратного) (Reinventing the wheel)

# Антипаттерны программирования

- *Ненужная сложность* (Accidental complexity)
- *Лодочный якорь* (Boat anchor)
- *Действие на расстоянии* (Action at a distance)
- *Накопить и запустить* (Accumulate and fire)
- *Таинственный код* (Cryptic code)
- *Хардкод* (Hard code)
- *Магические числа* (Magic numbers)
- *Спагетти-код* (Spaghetti code)
- *Мыльный пузырь* (Soap bubble)

https://ru.wikipedia.org/wiki/Антипаттерн

# Антипаттерны ООП

- *Базовый класс-утилита* (BaseBean)
- *Вызов предка* (CallSuper)
- *Божественный объект* (God object)
- *Объектная клоака* (Object cesspool)
- *Полтергейст* (Poltergeist)
- *Одиночество* (Singletonitis)

# Подводим итоги

- **Польза**
  - паттерны описывают решение целого класса проблем
  - унифицируется терминология, именование
  - удачные решения применяются вновь
  - независимость от языка программирования
- **Недостатки**
  - могут привести к созданию ненужных абстракций
  - большое количество использованных паттернов приводит к игнорированию их и всех системы
  - бездумное применение паттернов тормозит профессиональный рост программиста

# Итоги

- Паттерны проектирования – эффективные способы решения характерных задач проектирования ПО
- Паттерны – скачок разработки ПО от ремесла к инженерии
- Паттерны – не законченный образец проекта, а лишь способ решения, «повод подумать»
- Паттерны – не серебряная пуля
- Паттерны – хорошо, а своя голова и здравый смысл – намного лучше


Вопросы?

*e-mail:* marchenko@it.kfu.ru