ЛЕКЦИЯ 13

Паттерны проектирования. Примеры на Go.

Patterns

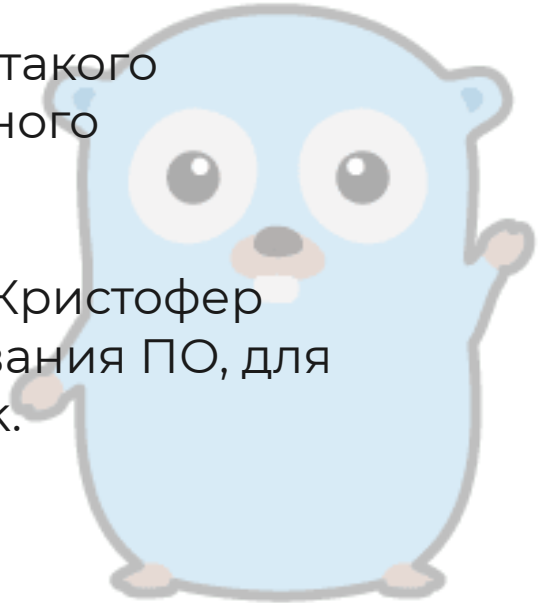
История создания:

В 1970-х архитектор **Кристофер Александр** составил набор шаблонов проектирования в области архитектуры.

Однако в области архитектуры эта идея не получила такого развития как позже в области разработки программного обеспечения.

В 1987 году **Кент Бэк** и **Вард Каннингем** взяли идеи Кристофер Александра и разработали шаблоны для проектирования ПО, для разработки графических оболочек на языке Smalltalk.

Также они написали ряд статей на данную тематику.



Patterns

История создания:

В 1988 году вдохновившись статьями Кента Бэка и Варда Каннингема, **Эрих Гамма** начал писать свою докторскую работу при цюрихском университете.

Закончив докторскую работу, Эрих Гамма переезжает в США, где в сотрудничестве с **Ричардом Хелмом**, **Ральфом Джонсоном** и **Джоном Влиссидсом** публикует книгу **Design Patterns – Elements of Reusable Object-Oriented Software**.

В книге описаны 23 шаблона проектирования, команду авторов этой книги стали называть бандой четырех (gang of four - **GoF**), а сами паттерны "гофовскими”.

Patterns

История создания:

В 1997 году **Крэг Ларман** также опубликовал книгу **Applying UML and Patterns**, в которой перечислил 9 паттернов названными GRASP-паттернами.

G.R.A.S.P. - General Responsibility Assignment Software Patterns.

Общие шаблоны распределения ответственностей - шаблоны, используемые в объектно-ориентированном проектировании для решения общих задач по назначению ответственностей классам и объектам.

Patterns

Как "вывели" паттерны:

Idiom – Напрямую связана с языком программирования

Specific design – Решение частной задачи

Standart design – Дополнительный уровень абстракции

Design pattern – Объектно-ориентированные шаблоны-отношения, взаимодействие и распределение ответственности между классами или объектами для всего класса задач.

Patterns

Пример идиомы: мультиплексор Fan-In на Go

```
func fanIn(input1, input2 <-chan int) <-chan int {
    c := make(chan int)
    go func() {
        for {
            select {
            case s := <-input1:  c <- s
            case s := <-input2:  c <- s
            }
        }
    }()
    return c
}
```

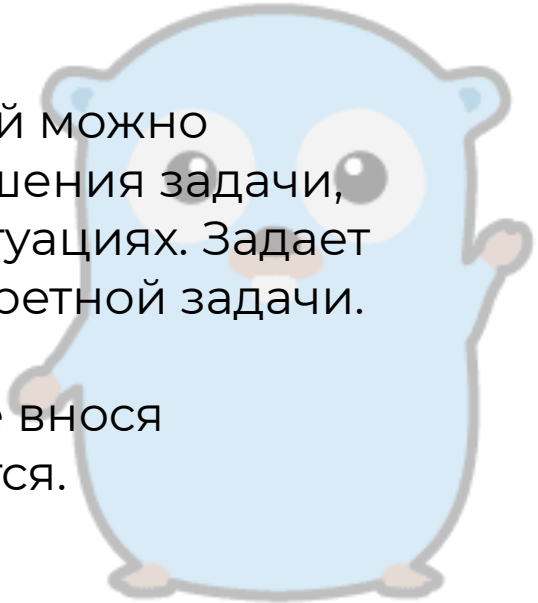
Patterns

Что такое шаблон/паттерн проектирования:

Шаблоны проектирования (паттерны) – это эффективные способы решения характерных задач проектирования, в частности проектирования программного обеспечения.

Паттерн не является законченным образцом, который можно прямо преобразовать в код, скорее это описание решения задачи, чтобы его можно было использовать в различных ситуациях. Задает направление для размышления над решением конкретной задачи.

Следует применять при реальной необходимости, не внося усложнение или избыточность там где это не требуется.



Patterns

Группы паттернов:

Порождающие паттерны – беспокоятся о гибком создании объектов без внесения в программу лишних зависимостей.

Структурные паттерны – показывают различные способы построения связей между объектами.

Поведенческие паттерны – заботятся об эффективной коммуникации между объектами.

UML

Unified Modeling Language:

UML - это система обозначений, которую можно применять для объектно-ориентированного анализа и проектирования.

Используется для визуализации, спецификации, конструирования и документирования программных систем.

Онлайн-редактор - PlantText.com

UML

Unified Modeling Language:

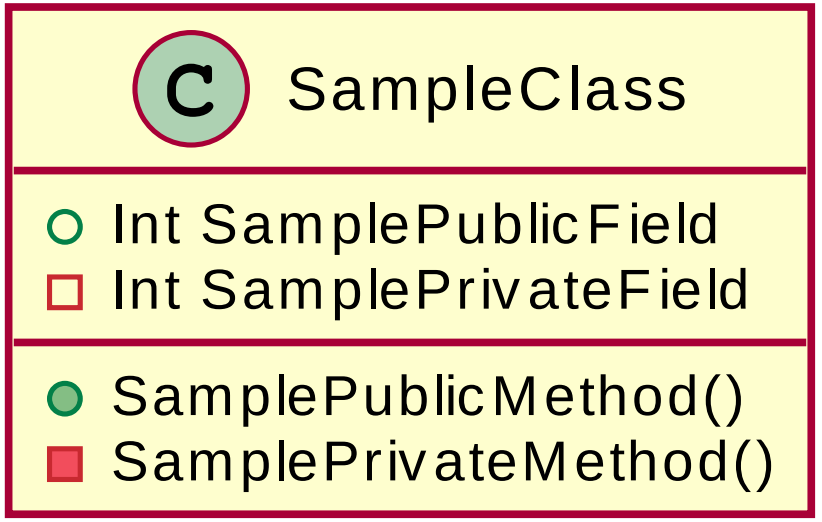
UML - это система обозначений, которую можно применять для объектно-ориентированного анализа и проектирования.

Используется для визуализации, спецификации, конструирования и документирования программных систем.

Онлайн-редактор - PlantText.com

UML

Classes - Class Diagram




```
@startuml

title Classes - Class Diagram

class SampleClass {
  +Int SamplePublicField
  -Int SamplePrivateField
  + SamplePublicMethod()
  - SamplePrivateMethod()
}

@enduml
```

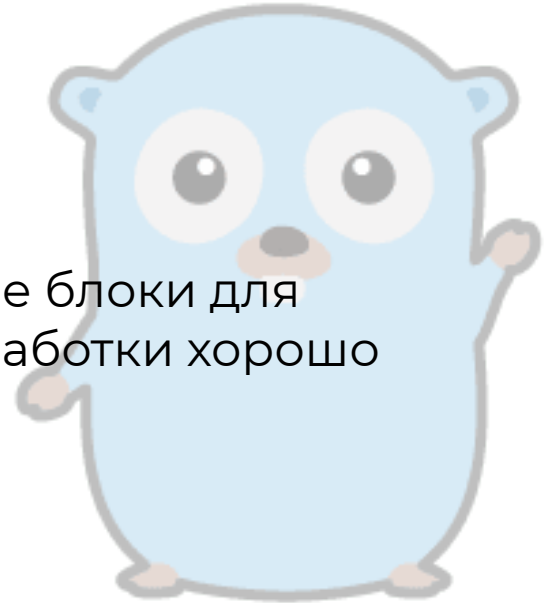
UML

Отношения между классами:

Существует четыре типа связей в UML:

- Зависимость
- Ассоциация
- Обобщение
- Реализация

Эти связи представляют собой базовые строительные блоки для описания отношений в UML, используемые для разработки хорошо согласованных моделей.



UML

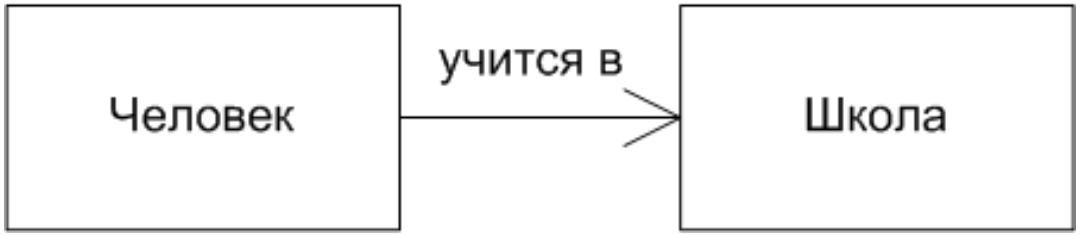
Зависимость – семантически представляет собой связь между двумя элементами модели, в которой изменение одного элемента (независимого) может привести к изменению семантики другого элемента (зависимого).

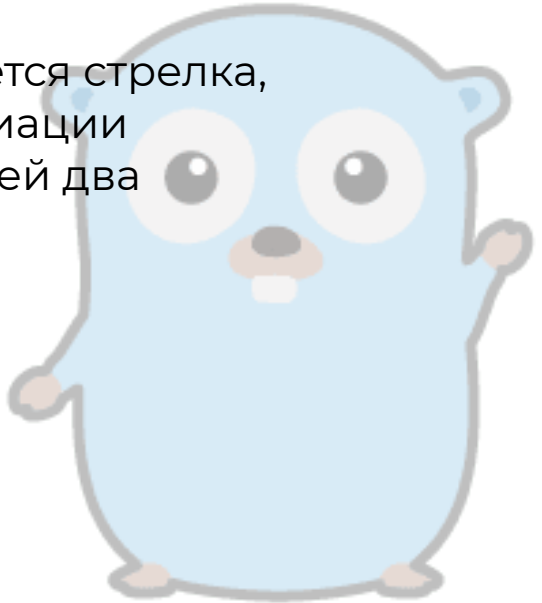
Зависимость – это связь использования, указывающая, что изменение спецификаций одной сущности может повлиять на другие сущности, которые используют ее.

UML

Ассоциация – это структурная связь между элементами модели, которая описывает набор связей, существующих между объектами.
Ассоциация показывает, что объекты одной сущности (класса) связаны с объектами другой сущности таким образом, что можно перемещаться от объектов одного класса к другому.

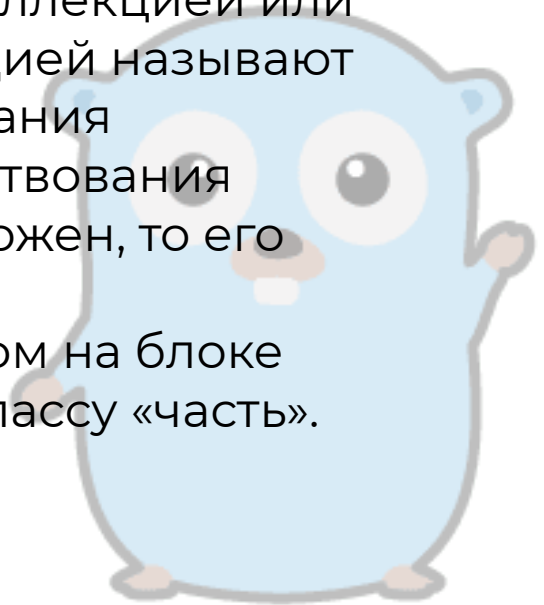
В представлении однонаправленной ассоциации добавляется стрелка, указывающая на направление ассоциации. Двойные ассоциации представляются линией без стрелок на концах, соединяющей два классовых блока.





UML

Агрегация – особая разновидность ассоциации, представляющая структурную связь целого с его частями. Как тип ассоциации, агрегация может быть именованной. Одно отношение агрегации не может включать более двух классов (контейнер и содержимое). Агрегация встречается, когда один класс является коллекцией или контейнером других. Причём, по умолчанию агрегацией называют агрегацию по ссылке, то есть когда время существования содержащихся классов не зависит от времени существования содержащего их класса. Если контейнер будет уничтожен, то его содержимое — нет.
Графически агрегация представляется пустым ромбом на блоке класса «целое», и линией, идущей от этого ромба к классу «часть».



UML

Композиция — более строгий вариант агрегации. Известна также как агрегация по значению.

Композиция – это форма агрегации с четко выраженными отношениями владения и совпадением времени жизни частей и целого.

Композиция имеет жёсткую зависимость времени существования экземпляров класса контейнера и экземпляров содержащихся классов. Если контейнер будет уничтожен, то всё его содержимое будет также уничтожено.

UML

Обобщение – выражает специализацию или наследование, в котором специализированный элемент (потомок) строится по спецификациям обобщенного элемента (родителя). Потомок разделяет структуру и поведение родителя.

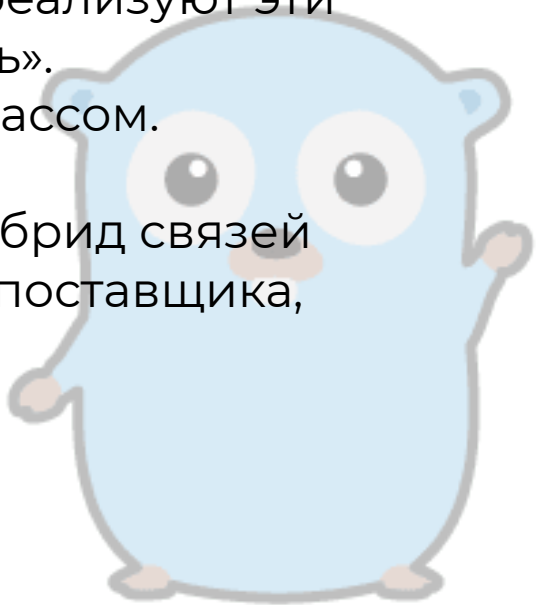
Графически обобщение представлено в виде сплошной линии с пустой стрелкой, указывающей на родителя.

UML

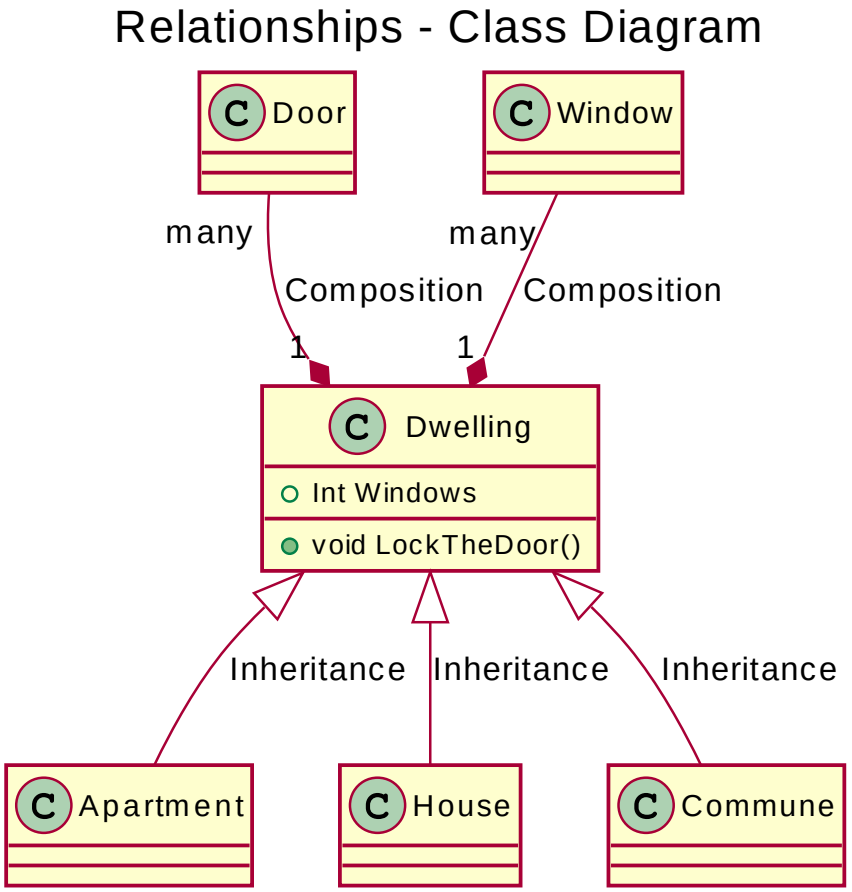
Реализация – это семантическая связь между классами, когда один из них (поставщик) определяет соглашение, которого второй (клиент) обязан придерживаться.

Это связи между интерфейсами и классами, которые реализуют эти интерфейсы. Это, своего рода, отношение «целое-часть». Поставщик, как правило, представлен абстрактным классом.

В графическом исполнении связь реализации – это гибрид связей обобщения и зависимости: треугольник указывает на поставщика, а второй конец пунктирной линии – на клиента.

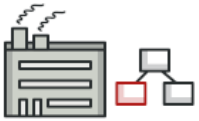


UML


Relationships - Class Diagram
Door
Window
many
many
Composition
Composition
1
1
Dwelling
Int Windows
void LockTheDoor()
Inheritance
Inheritance
Inheritance
Apartment
House
Commune

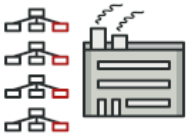
GoF patterns

Порождающие паттерны:



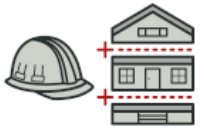
Фабричный метод
Factory Method

Определяет общий интерфейс для создания объектов в суперклассе, позволяя подклассам изменять тип создаваемых объектов.



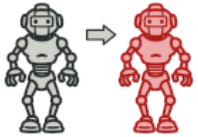
Абстрактная фабрика
Abstract Factory

Позволяет создавать семейства связанных объектов, не привязываясь к конкретным классам создаваемых объектов.



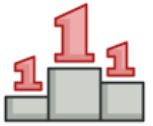
Строитель
Builder

Позволяет создавать сложные объекты пошагово. Строитель даёт возможность использовать один и тот же код строительства для получения разных представлений объектов.



Прототип
Prototype

Позволяет копировать объекты, не вдаваясь в подробности их реализации.

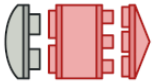


Одиночка
Singleton

Гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа.

GoF patterns

Структурные паттерны:



Адаптер
Adapter

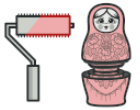
Позволяет объектам с несовместимыми интерфейсами работать вместе.

Мост
Bridge

Разделяет один или несколько классов на две отдельные иерархии — абстракцию и реализацию, позволяя изменять их независимо друг от друга.

Компоновщик
Composite

Позволяет сгруппировать множество объектов в древовидную структуру, а затем работать с ней так, как будто это единичный объект.

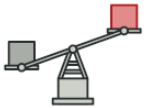


Декоратор
Decorator

Позволяет динамически добавлять объектам новую функциональность, оборачивая их в полезные «обёртки».

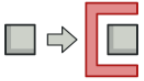
Фасад
Facade

Предоставляет простой интерфейс к сложной системе классов, библиотеке или фреймворку.



Легковес
Flyweight

Позволяет вместить бóльшее количество объектов в отведённую оперативную память. Легковес экономит память, разделяя общее состояние объектов между собой, вместо хранения одинаковых данных в каждом объекте.



Заместитель
Proxy

Позволяет подставлять вместо реальных объектов специальные объекты-заменители. Эти объекты перехватывают вызовы к оригинальному объекту, позволяя сделать что-то *до* или *после* передачи вызова оригиналу.

GoF patterns

Поведенческие паттерны:

Цепочка обязанностей
Chain of Responsibility

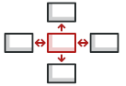
Позволяет передавать запросы последовательно по цепочке обработчиков. Каждый последующий обработчик решает, может ли он обработать запрос сам и стоит ли передавать запрос дальше по цепи.

Команда
Command

превращает запросы в объекты, позволяя передавать их как аргументы при вызове методов, ставить запросы в очередь, логировать их, а также поддерживать отмену операций.

Итератор
Iterator

Даёт возможность последовательно обходить элементы составных объектов, не раскрывая их внутреннего представления.

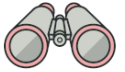


Посредник
Mediator

Позволяет уменьшить связанность множества классов между собой, благодаря перемещению этих связей в один класс-посредник.

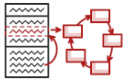
Снимок
Memento

Позволяет сохранять и восстанавливать прошлые состояния объектов, не раскрывая подробностей их реализации.



Наблюдатель
Observer

Создаёт механизм подписки, позволяющий одним объектам следить и реагировать на события, происходящие в других объектах.



Состояние
State

Позволяет объектам менять поведение в зависимости от своего состояния. Извне создаётся впечатление, что изменился класс объекта.

Стратегия
Strategy

Определяет семейство схожих алгоритмов и помещает каждый из них в собственный класс, после чего алгоритмы можно взаимозаменять прямо во время исполнения программы.

Шаблонный метод
Template Method

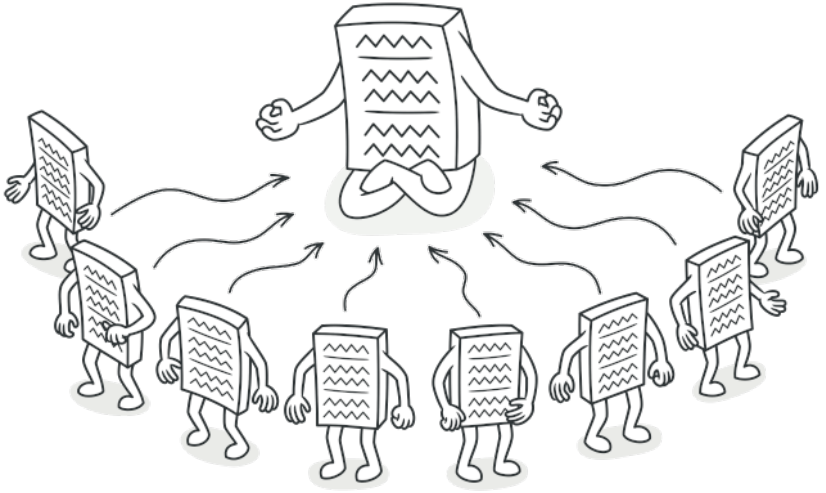
Определяет скелет алгоритма, перекладывая ответственность за некоторые его шаги на подклассы. Паттерн позволяет подклассам переопределять шаги алгоритма, не меняя его общей структуры.

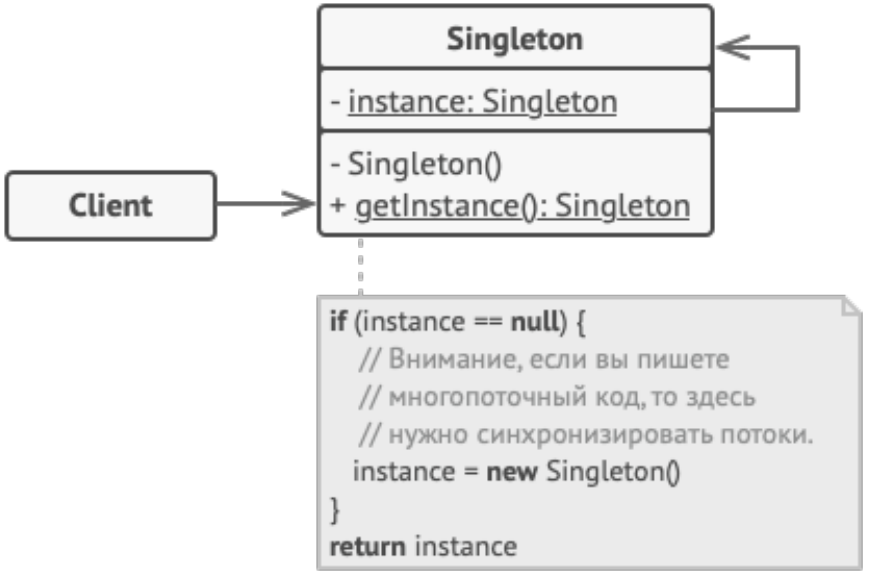
Посетитель
Visitor

Позволяет добавлять в программу новые операции, не изменяя классы объектов, над которыми эти операции могут выполняться.

GoF: Singleton

Одиночка — это порождающий паттерн проектирования, который гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа.





GoF: Singleton

1. Гарантирует наличие единственного экземпляра класса.

Чаще всего это полезно для доступа к какому-то общему ресурсу, например, базе данных.

Представьте, что вы создали объект, а через некоторое время пробуете создать ещё один. В этом случае хотелось бы получить старый объект, вместо создания нового.

Такое поведение невозможно реализовать с помощью обычного конструктора, так как конструктор класса всегда возвращает новый объект.

GoF: Singleton

2. Предоставляет глобальную точку доступа.

Это не просто глобальная переменная, через которую можно достучаться к определённому объекту. Глобальные переменные не защищены от записи, поэтому любой код может подменять их значения без вашего ведома.

Но есть и другой нюанс. Неплохо бы хранить в одном месте и код, который решает проблему №1, а также иметь к нему простой и доступный интерфейс.

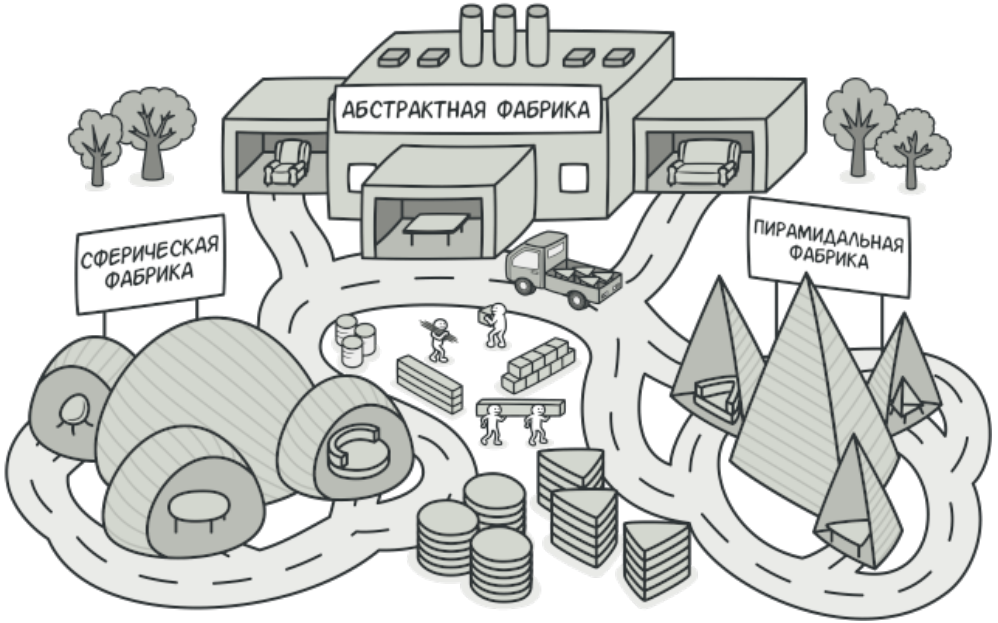
GoF: Singleton

```
type singleton struct {
    count int
    sync.RWMutex
}

func (s *singleton) AddOne() {
    s.Lock()
    defer s.Unlock()
    s.count++
}

func (s *singleton) GetCount()int
{
    s.RLock()
    defer s.RUnlock()
    return s.count
}
```

```
var instance singleton

func GetInstance() *singleton {
    return &instance
}
```

GoF: Abstract Factory

Абстрактная фабрика — это порождающий паттерн проектирования, который позволяет создавать семейства связанных объектов, не привязываясь к конкретным классам создаваемых объектов.



GoF: Abstract Factory

Представьте, что вы пишете симулятор мебельного магазина. Ваш код содержит:

Семейство зависимых продуктов. Скажем, Кресло + Диван + Столик.

Несколько вариаций этого семейства.
Например, продукты Кресло, Диван и Столик представлены
в трёх разных стилях: Ар-деко, Викторианском и Модерне.

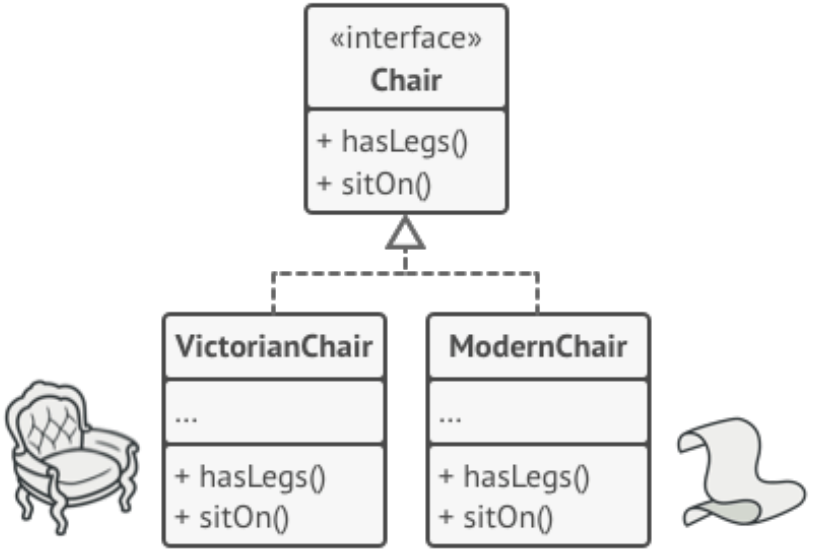
Вам нужен такой способ создавать объекты продуктов, чтобы они сочетались с другими продуктами того же семейства. Это важно, так как клиенты расстраиваются, если получают несочетающуюся мебель.

Кроме того, вы не хотите вносить изменения в существующий код при добавлении новых продуктов или семейств в программу. Поставщики часто обновляют свои каталоги, и вы бы не хотели менять уже написанный код каждый раз при получении новых моделей мебели.

GoF: Abstract Factory

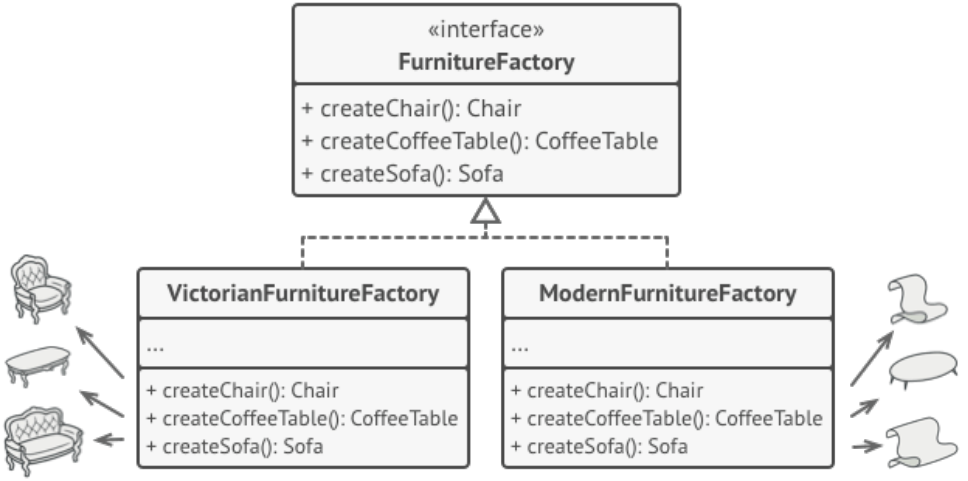
GoF: Abstract Factory

Для начала паттерн Абстрактная фабрика предлагает выделить общие интерфейсы для отдельных продуктов, составляющих семейства. Так, все вариации кресел получат общий интерфейс Кресло, все диваны реализуют интерфейс Диван и так далее.



GoF: Abstract Factory

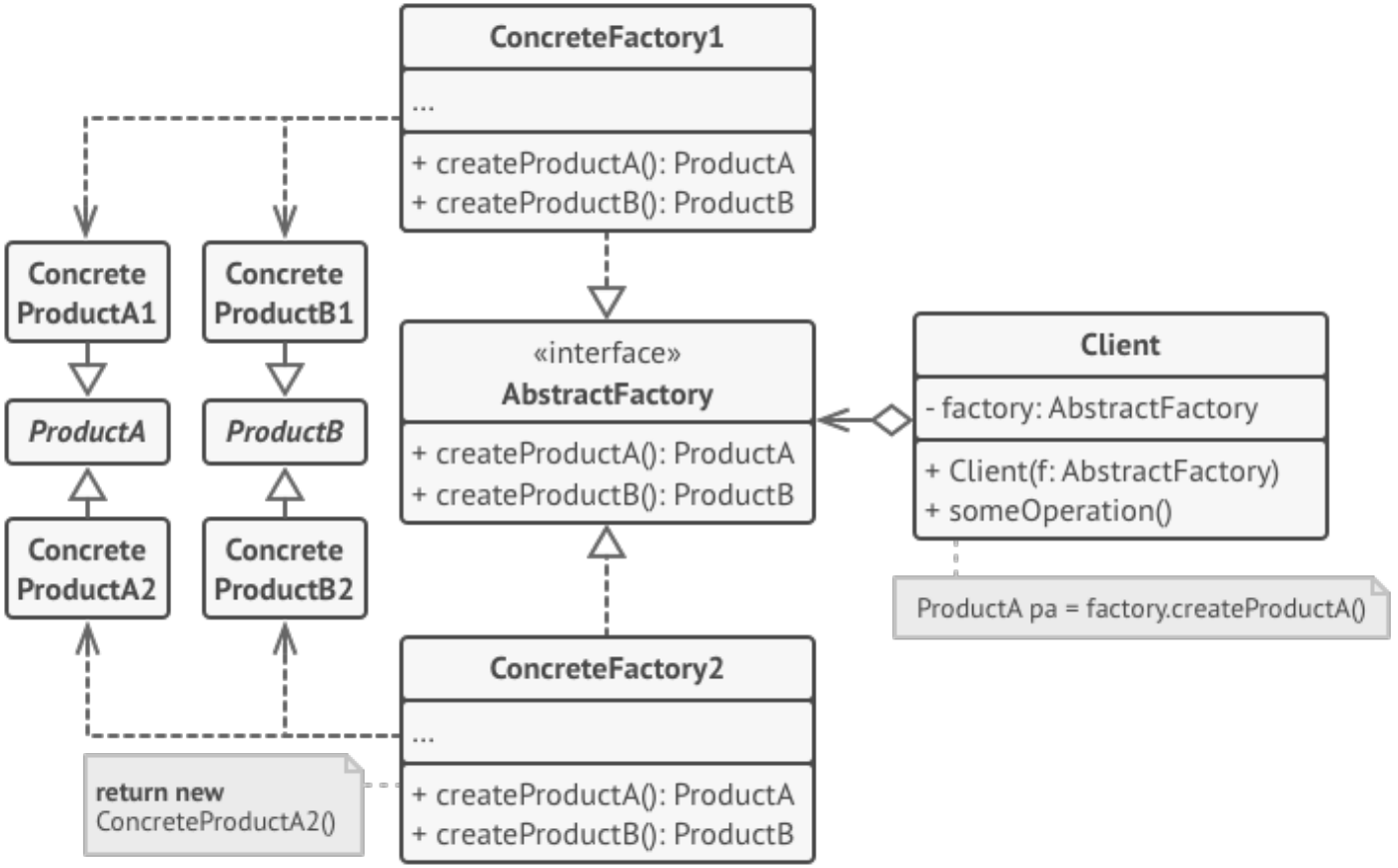
Далее вы создаёте абстрактную фабрику — общий интерфейс, который содержит методы создания всех продуктов семейства (например, создатьКресло, создатьДиван и создатьСтолик). Эти операции должны возвращать абстрактные типы продуктов, представленные интерфейсами, которые мы выделили ранее — Кресла, Диваны и Столики.



GoF: Abstract Factory

Клиентский код должен работать как с фабриками, так и с продуктами только через их общие интерфейсы. Это позволит подавать в ваши классы любой тип фабрики и производить любые продукты, ничего не ломая. Для клиентского кода должно быть безразлично, с какой фабрикой работать.

GoF: Abstract Factory



GoF: Singleton

```go
// Package abstract_factory is an example of the Abstract Factory Pattern.
package abstract_factory



// AbstractFactory provides an interface for creating families
// of related objects.
type AbstractFactory interface {
        CreateWater(volume float64) AbstractWater
        CreateBottle(volume float64) AbstractBottle
}

// AbstractWater provides a water interface.
type AbstractWater interface {
        GetVolume() float64
}

// AbstractBottle provides a bottle interface.
type AbstractBottle interface {
        PourWater(water AbstractWater) // Bottle interacts with a water.
        GetBottleVolume() float64
        GetWaterVolume() float64
}



// CocaColaFactory implements AbstractFactory interface.
type CocaColaFactory struct {
}

// NewCocaColaFactory is the CocaColaFactory constructor.
func NewCocaColaFactory() AbstractFactory {
        return &CocaColaFactory{}
}
```

```go
// CreateWater implementation.
func (f *CocaColaFactory) CreateWater(volume float64) AbstractWater {
        return &CocaColaWater{volume: volume}
}

// CreateBottle implementation.
func (f *CocaColaFactory) CreateBottle(volume float64) AbstractBottle {
        return &CocaColaBottle{volume: volume}
}

// CocaColaWater implements AbstractWater.
type CocaColaWater struct {
        volume float64 // Volume of drink.
}

// GetVolume returns volume of drink.
func (w *CocaColaWater) GetVolume() float64 {
        return w.volume
}

// CocaColaBottle implements AbstractBottle.
type CocaColaBottle struct {
        water  AbstractWater // Bottle must contain a drink.
        volume float64       // Volume of bottle.
}

// PourWater pours water into a bottle.
func (b *CocaColaBottle) PourWater(water AbstractWater) {
        b.water = water
}

// GetBottleVolume returns volume of bottle.
func (b *CocaColaBottle) GetBottleVolume() float64 {
        return b.volume
}

// GetWaterVolume returns volume of water.
func (b *CocaColaBottle) GetWaterVolume() float64 {
        return b.water.GetVolume()
}
```