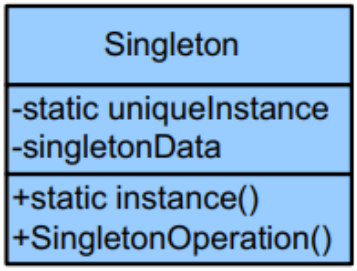
Одиночка
Singleton

Тип: Порождающий

Что это:

Гарантирует, что класс имеет только один экземпляр и предоставляет глобальную точку доступа к нему.




```
//Singleton

//  {}=={} false

// let instance

class Counter {
  constructor() {
    if (typeof Counter.instance == 'object') {
      return Counter.instance
    }

    this.count = 0
    Counter.instance = this
    return this
  }

  getCounter() {
    return this.count
  }

  increaseCount() {
    return this.count++
  }
}

const myCount1 = new Counter()
const myCount2 = new Counter()

myCount1.increaseCount()
myCount1.increaseCount()
myCount2.increaseCount()

console.log(myCount1.getCounter()) //3
console.log(myCount2.getCounter()) //3
```

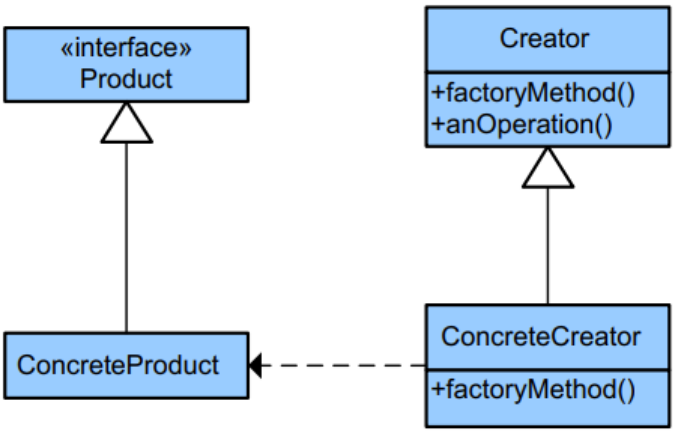
Фабричный метод

Factory method

Тип: Порождающий

Что это:

Определяет интерфейс для создания объекта, но позволяет подклассам решать, какой класс инстанцировать. Позволяет делегировать создание объекта подклассам.




```js
class Bmv {
  constructor(model, price, maxSpeed) {
    this.model = model
    this.price = price
    this.maxSpeed = maxSpeed
  }
}

class BmvFactory {
  create(type) {
    if (type === 'X5') return new Bmv(type, 10800, 300)
    if (type === 'X6') return new Bmv(type, 11100, 320)
  }
}

const factory = new BmvFactory()

const x5 = factory.create('X5')
const x6 = factory.create('X6')

console.log(x5)

// {
//     "model": "X5",
//     "price": 10800,
//     "maxSpeed": 300
// }

console.log(x6)

// {
//     "model": "X6",
//     "price": 11100,
//     "maxSpeed": 320
// }
```

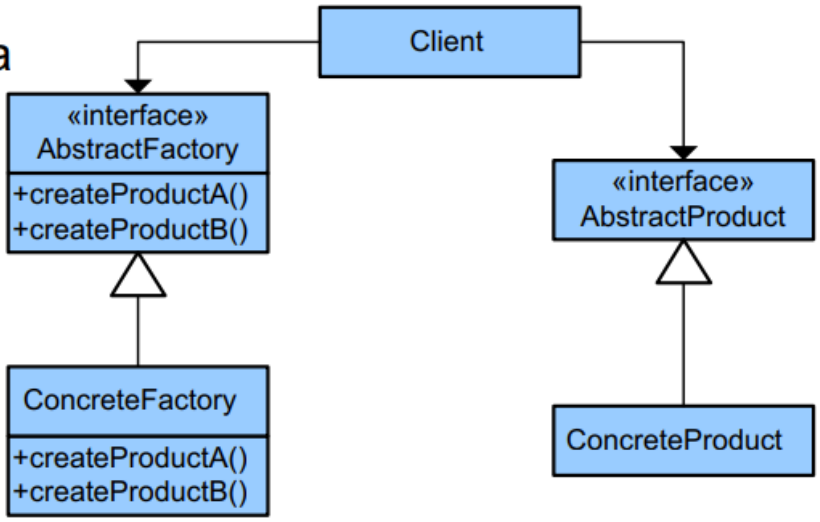
Абстрактная фабрика

Abstract factory

Тип: Порождающий

Что это:

Предоставляет интерфейс для создания групп связанных или зависимых объектов, не указывая их конкретный класс.



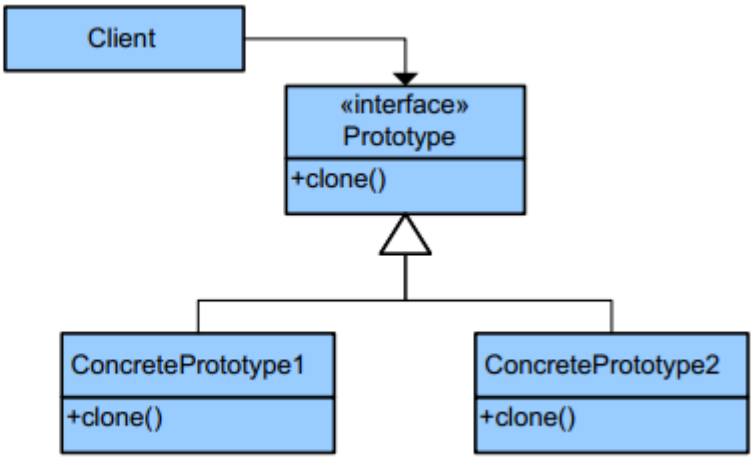

```
// Abstract factory

function bmwProducer(kind) {
  return kind === 'sport' ? sportCarFactory : familyCarFactory
}

function sportCarFactory() {
  return new Z4()
}
function familyCarFactory() {
  return new I3()
}

class Z4 {
  info() {
    return 'Z4 is a Sport car!'
  }
}
class I3 {
  info() {
    return 'I3 is a Family car!'
  }
}

const produce = bmwProducer('sport')

const myCar = new produce()

console.log(myCar.info()) // Z4 is a Sport car!
```

Прототип
Prototype

Тип: Порождающий

Что это:

Определяет несколько видов объектов, чтобы при создании использовать объект-прототип и создаёт новые объекты, копируя прототип.



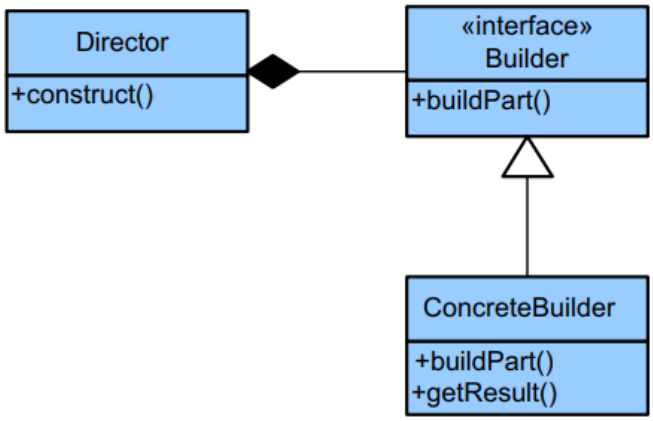

```
// Prototype

class TeslaCar {
  constructor(model, price, interior, autopilot) {
    this.model = model
    this.price = price
    this.interior = interior
    this.autopilot = autopilot
  }

  produce() {
    return new TeslaCar(this.model, this.price, this.interior, this.autopilot)
  }
}

const prototypeCar = new TeslaCar('S', 80000, 'black', false)

const car1 = prototypeCar.produce()
const car2 = prototypeCar.produce()
const car3 = prototypeCar.produce()

car1.interior = 'white'
car1.autopilot = true

console.log(car1)
// {
//     "model": "S",
//     "price": 80000,
//     "interior": "white",
//     "autopilot": true
// }
console.log(car2)
// {
//     "model": "S",
//     "price": 80000,
//     "interior": "black",
//     "autopilot": false
// }
console.log(car3)
```

Строитель
Builder

Тип: Порождающий

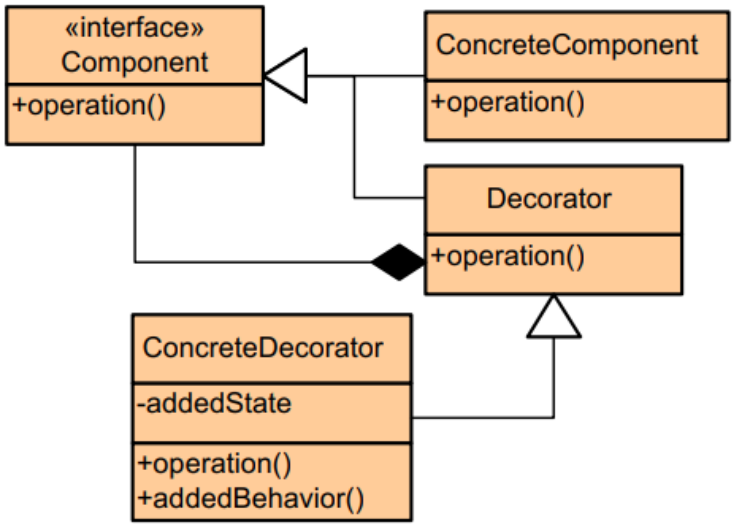
Что это:

Разделяет создание сложного объекта и инициализацию его состояния так, что одинаковый процесс построения может создать объекты с разным состоянием.




```
// Builder

class Car {
  constructor() {
    this.autoPilot = false
    this.parktronic = false
    this.signaling = false
  }
}

class CarBuilder {
  constructor() {
    this.car = new Car()
  }

  addAutoPilot(autoPilot) {
    this.car.autoPilot = autoPilot
    return this
  }

  addParktronic(parktronic) {
    this.car.parktronic = parktronic
    return this
  }

  addSignaling(signaling) {
    this.car.signaling = signaling
    return this
  }

  updateEngine(engine) {
    this.car.engine = engine
    return this
  }

  build() {
    return this.car
  }
}

const myCar = new CarBuilder()
  .addAutoPilot(true)
  .addParktronic(true)
  .updateEngine('V8')
  .build()

console.log(myCar)
```

```
// {
//     "autoPilot": true,
//     "parktronic": true,
//     "signaling": false,
//     "engine": "V8"
// }
```



Декоратор
Decorator

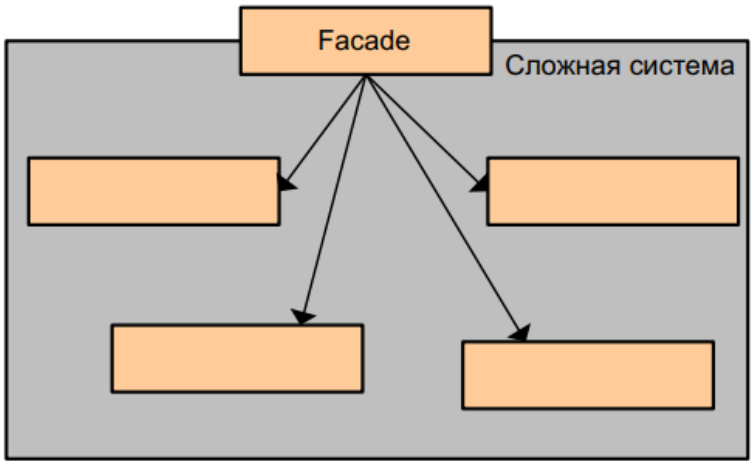
Тип: Структурный

Что это:

Динамически предоставляет объекту дополнительные возможности. Представляет собой гибкую альтернативу наследованию для расширения функциональности.

```
// Decorator

class Car {
  constructor() {
    this.price = 10000
    this.model = 'Car'
  }

  getPrice() {
    return this.price
  }

  getDescription() {
    return this.model
  }
}

class Tesla extends Car {
  constructor() {
    super()
    this.price = 25000
    this.model = 'Tesla'
  }
}

class AutoPilot {
  constructor(car) {
    this.car = car
  }

  getPrice() {
    return this.car.getPrice() + 5000
  }

  getDescription() {
    return `${this.car.getDescription()} with autopilot`
  }
}
```

```
class Parktronic {
  constructor(car) {
    this.car = car
  }

  getPrice() {
    return this.car.getPrice() + 3000
  }

  getDescription() {
    return `${this.car.getDescription()} with parktronic`
  }
}

let tesla = new Tesla()

tesla = new AutoPilot(tesla)
tesla = new Parktronic(tesla)

console.log(tesla.getPrice(), tesla.getDescription())
//33000 Tesla with autopilot with parktronic

let newTesla = new Tesla()

newTesla = new AutoPilot(newTesla)

console.log(newTesla.getPrice(), newTesla.getDescription())
//30000 Tesla with autopilot
```



Фасад
Facade

Тип: Структурный
Что это:
Предоставляет единый интерфейс к группе интерфейсов подсистемы. Определяет высокоуровневый интерфейс, делая подсистему проще для использования.

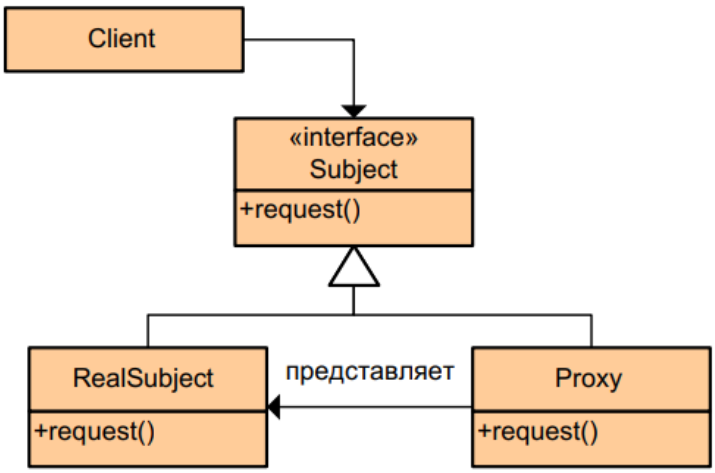
```
/*
Body set!
Dismantle Engine!
Engine set!
Update interior!
Interior added!
Added exterior!
Wheels!!!
Added electronics!
Car painted!
Dismantle Engine!
Engine set!
Update interior!
Interior added!
*/
```

```
// Facade

class Conveyor {
  setBody() {
    console.log('Body set!')
  }

  getEngine() {
    console.log('Dismantle Engine!')
  }

  setEngine() {
    console.log('Engine set!')
  }

  getInterior() {
    console.log('Update interior!')
  }

  setInterior() {
    console.log('Interior added!')
  }

  setExterior() {
    console.log('Added exterior!')
  }

  setWheels() {
    console.log('Wheels!!!')
  }

  addElectronic() {
    console.log('Added electronics!')
  }
  print() {
    console.log('Car painted!')
  }
}
```

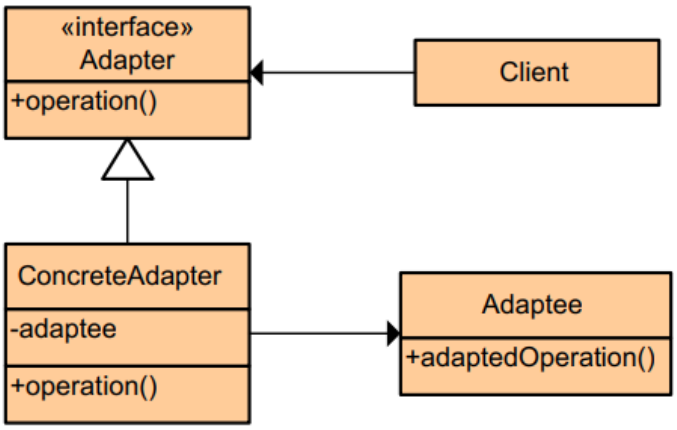
```
class ConveyorFacade {
  constructor(car) {
    this.car = car
  }

  assembleCar() {
    this.car.setBody()
    this.car.getEngine()
    this.car.setEngine()
    this.car.getInterior()
    this.car.setInterior()
    this.car.setExterior()
    this.car.setWheels()
    this.car.addElectronic()
    this.car.print()
  }

  changeEngine() {
    this.car.getEngine()
    this.car.setEngine()
  }
  changeInterior() {
    this.car.getInterior()
    this.car.setInterior()
  }
}

const conveyor = new ConveyorFacade(new Conveyor())
let car = conveyor.assembleCar()
car = conveyor.changeEngine()
car = conveyor.changeInterior()

console.log(car)
```

Прокси
Proxy

Тип: Структурный
Что это:
Предоставляет замену другого объекта для контроля доступа к нему.



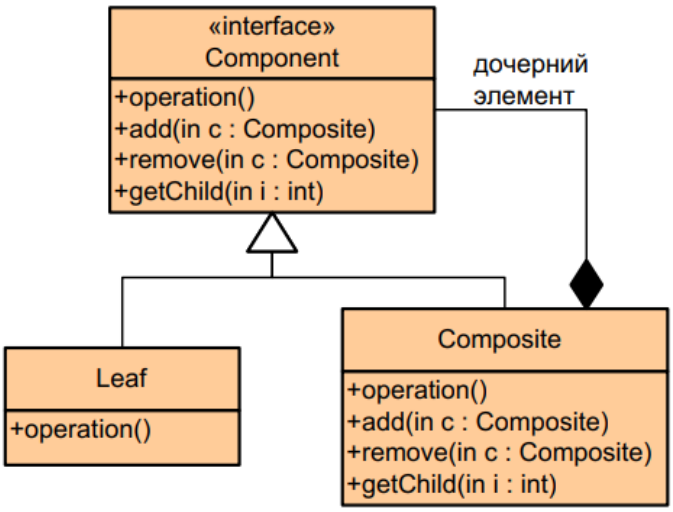

```
// Proxy

class CarAccess {
  open() {
    console.log('Opening car door')
  }

  close() {
    console.log('Closing the car door')
  }
}

class SecuritySystem {
  constructor(door) {
    this.door = door
  }

  open(password) {
    if (this.authenticate(password)) return this.door.open()
    console.log('Access denied!')
  }

  authenticate(password) {
    return password === 'Ilon'
  }

  close() {
    this.door.close()
  }
}

const door = new SecuritySystem(new CarAccess())

door.open('Jack') //Access denied!
door.open('Ilon') //Opening car door
door.close() //Closing the car door
```



Адаптер
Adapter

Тип: Структурный
Что это:
Конвертирует интерфейс класса в другой интерфейс, ожидаемый клиентом. Позволяет классам с разными интерфейсами работать вместе.

```
//Adapter

class Engine2 {
  simpleInterface() {
    console.log('Engine 2.0 - tr-tr')
  }
}

class EngineV8 {
  complicatedInterface() {
    console.log('Engine V8! - wroom!')
  }
}

class EngineV8Adapter {
  constructor(engine) {
    this.engine = engine
  }

  simpleInterface() {
    this.engine.complicatedInterface()
  }
}

class Auto {
  startEngine(engine) {
    engine.simpleInterface()
  }
}

const myCar = new Auto()
const oldEngine = new Engine2()

myCar.startEngine(oldEngine) //Engine 2.0 - tr-tr

const newCar = new Auto()
const engineAdapter = new EngineV8Adapter(new EngineV8())

newCar.startEngine(engineAdapter) //Engine V8! - wroom!

const errCar = new Auto()
const engineV8 = new EngineV8()

errCar.startEngine(engineV8) //Erro
```



Компоновщик
Composite

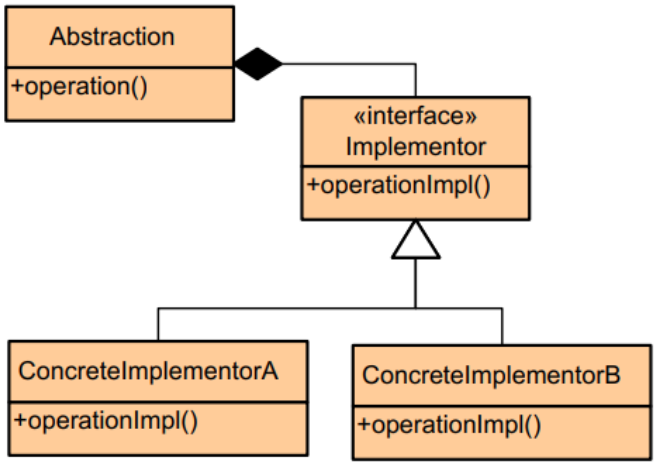
Тип: Структурный

Что это:

Компонует объекты в древовидную структуру, представляя их в виде иерархии. Позволяет клиенту одинаково обращаться как к отдельному объекту, так и к целому поддереву.

```
//Composite

class Equipment {
  getPrice() {
    return this.price || 0
  }

  getName() {
    return this.name
  }

  setName(name) {
    this.name = name
  }

  setPrice(price) {
    this.price = price
  }
}

class Engine extends Equipment {
  constructor() {
    super()
    this.setName('Engine')
    this.setPrice(800)
  }
}

class Body extends Equipment {
  constructor() {
    super()
    this.setName('Body')
    this.setPrice(3000)
  }
}

class Tools extends Equipment {
  constructor() {
    super()
    this.setName('Tools')
    this.setPrice(4000)
  }
}
```

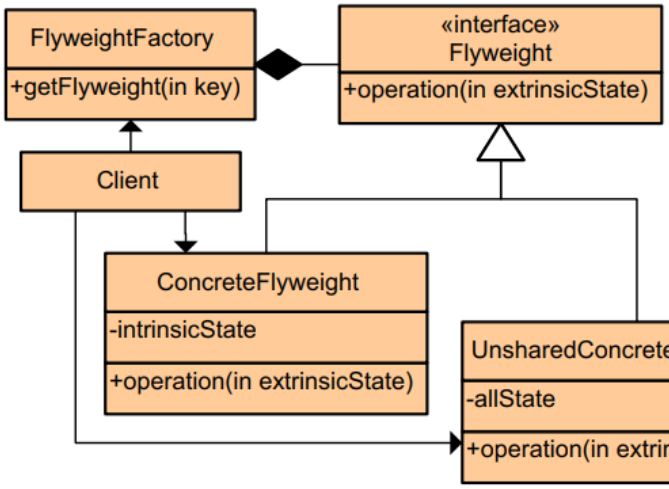
```
class Composite extends Equipment {
  constructor() {
    super()
    this.equipments = []
  }

  add(equipments) {
    this.equipments.push(equipments)
  }

  getPrice() {
    return this.equipments
      .map((equipment) => equipment.getPrice())
      .reduce((a, b) => a + b)
  }
}

class Car extends Composite {
  constructor() {
    super()
    this.setName('Audi')
  }
}

const myCar = new Car()

myCar.add(new Engine())
myCar.add(new Body())
myCar.add(new Tools())

console.log(`${myCar.getName()} price is ${myCar.getPrice()}$`)
// Audi price is 7800$
```



Мост
Bridge

Тип: Структурный
Что это:
Разделяет абстракцию и реализацию так, чтобы они могли изменяться независимо.

```
//Bridge

class Model {
  constructor(color) {
    this.color = color
  }
}

class Color {
  constructor(type) {
    this.type = type
  }

  get() {
    return this.type
  }
}

class BlackColor extends Color {
  constructor() {
    super('dark-black')
  }
}

class SiblingColor extends Color {
  constructor() {
    super('Silbermatallic')
  }
}

class Audi extends Model {
  constructor(color) {
    super(color)
  }

  paint() {
    return `Auto: Audi, Color: ${this.color.get()}`
  }
}
class Bmw extends Model {
  constructor(color) {
    super(color)
  }

  paint() {
    return `Auto: Bmw, Color: ${this.color.get()}`
  }
}

const blackBmw = new Bmw(new BlackColor())

console.log(blackBmw.paint())
//Auto: Bmw, Color: dark-black
```



Приспособленец

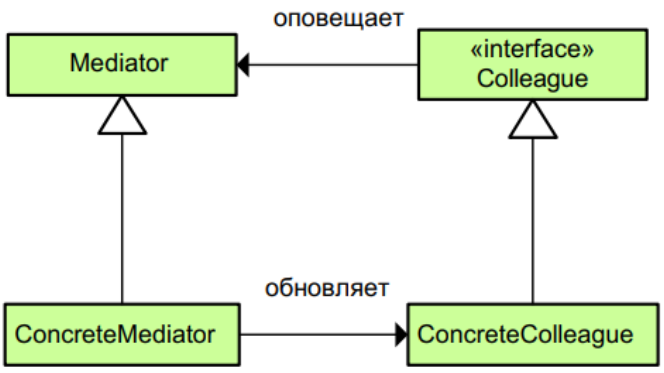
Flyweight

Тип: Структурный

Что это:

Благодаря совместному использованию, поддерживает эффективную работу с большим количеством объектов.

```
//Flyweight

class Auto {
  constructor(model) {
    this.model = model
  }
}

class AutoFactory {
  constructor(name) {
    this.models = {}
  }
  create(name) {
    let model = this.models[name]
    if (model) return model
    console.count('model')

    this.models[name] = new Auto(name)

    return this.models[name]
  }

  getModels() {
    console.log(this.models)
    console.table(this.models)
  }
}

const factory = new AutoFactory()

const bmw = factory.create('BMW')
const audi = factory.create('Audi')
const tesla = factory.create('Tesla')
const blackTesla = factory.create('Tesla')
/*
model: 1
model: 2
model: 3
*/
factory.getModels()

/*
{
    "BMW": {"model": "BMW" },
    "Audi": {"model": "Audi" },
    "Tesla": { "model": "Tesla" }
}
*/
```



Посредник
Mediator

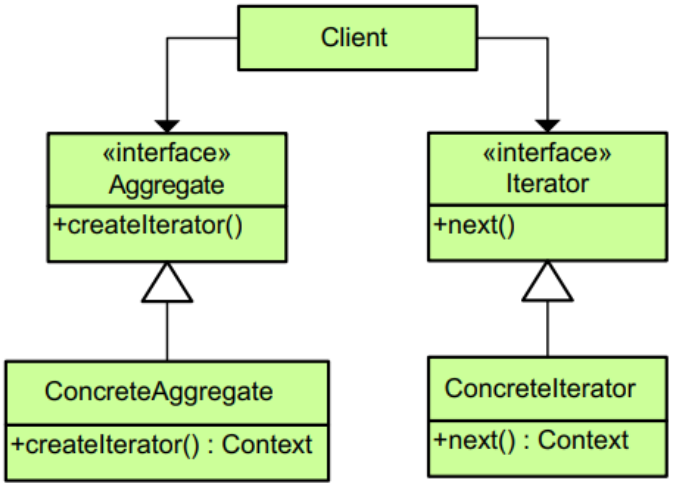
Тип: Поведенческий

Что это:

Определяет объект, инкапсулирующий способ взаимодействия объектов. Обеспечивает слабую связь, избавляя объекты от необходимости прямо ссылать друг на друга и даёт возможность независимо изменять их взаимодействие.

```
//Mediator

class OfficialDealer {
  constructor() {
    this.customers = []
  }

  orderAuto(customer, auto, info) {
    const name = customer.getName()

    console.log(`Order name ${name}. Order auto is ${auto}`)
    console.log(`Additional info: ${info}`)
    this.addCustomerList(name)
  }

  addCustomerList(name) {
    this.customers.push(name)
  }

  getCustomerList() {
    return this.customers
  }
}

class Customer {
  constructor(name, dealerMediator) {
    this.name = name
    this.dealerMediator = dealerMediator
  }

  getName() {
    return this.name
  }

  makeOrder(auto, info) {
    this.dealerMediator.orderAuto(this, auto, info)
  }
}
```

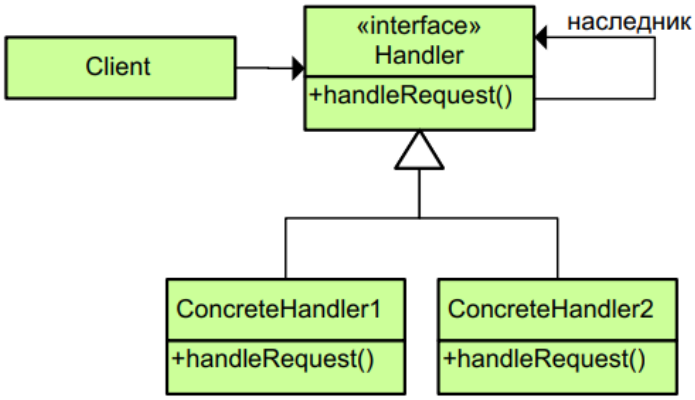
```
const mediator = new OfficialDealer()

const yauhen = new Customer('Yauhen', mediator)

const valera = new Customer('Valera', mediator)

yauhen.makeOrder('Tesla', 'With autopilot!')
//Order name Yauhen. Order auto is Tesla
//Additional info: With autopilot!

valera.makeOrder('Tesla', 'With parktronik!')
// Order name Valera. Order auto is Tesla
// Additional info: With parktronik!

console.log(mediator.getCustomerList())
// [    "Yauhen",    "Valera"]
```



Итератор
Iterator

Тип: Поведенческий
Что это:
Предоставляет способ последовательного доступа к элементам множества, независимо от его внутреннего устройства.

```
// Iterator

class Iterator {
  constructor(el) {
    this.index = 0
    this.elements = el
  }

  next() {
    return this.elements[this.index++]
  }

  hasNext() {
    return this.index < this.elements.length
  }
}

const collection = new Iterator(['Audi', 'BMW', 'Tesla', 'Mercedes'])

while (collection.hasNext()) {
  console.log(collection.next())
}

// 'Audi', 'BMW', 'Tesla', 'Mercedes
```



Цепочка обязанностей

Chain of responsibility

Тип: Поведенческий

Что это:

Избегает связывания отправителя запроса с его получателем, давая возможность обработать запрос более чем одному объекту. Связывает объекты-получатели и передаёт запрос по цепочке пока объект не обработает его.

```
//Chain of responsibility

class Account {
  pay(orderPrice) {
    if (this.canPay(orderPrice)) {
      console.log(`Paid ${orderPrice} using ${this.name}`)
    } else if (this.incomer) {
      console.log(`Cannot pay using ${this.name}`)
      this.incomer.pay(orderPrice)
    } else {
      console.log('Unfortunately, not enough money')
    }
  }
  canPay(amount) {
    return this.balance ⩾ amount
  }
  setNext(account) {
    this.incomer = account
  }
  show() {
    console.log(this)
  }
}

class Master extends Account {
  constructor(balance) {
    super()
    this.name = 'Master Card'
    this.balance = balance
  }
}
class Paypal extends Account {
  constructor(balance) {
    super()
    this.name = 'Paypal'
    this.balance = balance
  }
}
class Qiwi extends Account {
  constructor(balance) {
    super()
    this.name = 'Qiwi'
    this.balance = balance
  }
}
```

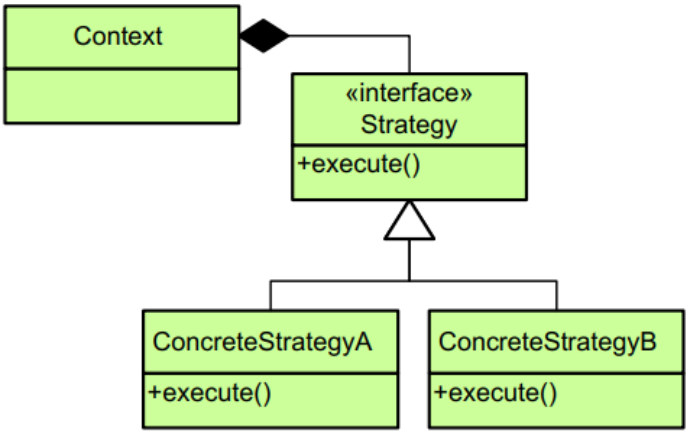
```
const master = new Master(100)
const paypal = new Paypal(200)
const qiwi = new Qiwi(500)

master.setNext(paypal)
paypal.setNext(qiwi)

master.pay(438)
/*
Cannot pay using Master Card
Cannot pay using Paypal
Paid 438 using Qiwi
*/

master.show()
/*
{
    "name": "Master Card",
    "balance": 100,
    "incomer": {
        "name": "Paypal",
        "balance": 200,
        "incomer": {
            "name": "Qiwi",
            "balance": 500
        }
    }
}
*/
```

Стратегия
Strategy

Тип: Поведенческий

Что это:

Определяет группу алгоритмов, инкапсулирует их и делает взаимозаменяемыми. Позволяет изменять алгоритм независимо от клиентов, его использующих.



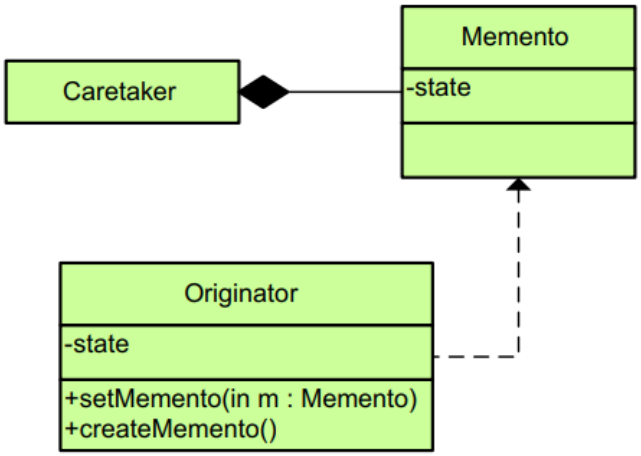

```
//Strategy

function baseStrategy(amount) {
  return amount
}

function premiumStrategy(amount) {
  return amount * 0.85
}

function platinumStrategy(amount) {
  return amount * 0.65
}

class AutoCart {
  constructor(discount) {
    this.discount = discount
    this.amount = 0
  }

  checkout() {
    return this.discount(this.amount)
  }

  setAmount(amount) {
    this.amount = amount
  }
}

const baseCustomer = new AutoCart(baseStrategy)
const premiumCustomer = new AutoCart(premiumStrategy)
const platinumCustomer = new AutoCart(platinumStrategy)

baseCustomer.setAmount(50000)
console.log(baseCustomer.checkout()) // 50000

premiumCustomer.setAmount(50000)
console.log(premiumCustomer.checkout()) // 42500

platinumCustomer.setAmount(50000)
console.log(platinumCustomer.checkout()) // 32500
```

Хранитель
Memento

Тип: Поведенческий

Что это:

Не нарушая инкапсуляцию, определяет и сохраняет внутреннее состояние объекта и позволяет позже восстановить объект в этом состоянии.



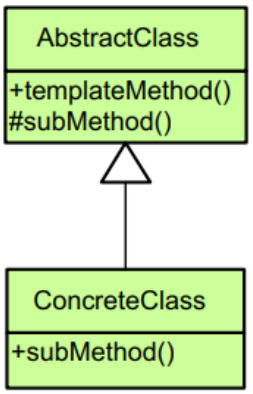

```
//Memento

class Memento {
  constructor(value) {
    this.value = value
  }
}

const creator = {
  save: (val) => new Memento(val),
  restore: (memento) => memento.value,
}

class Caretaker {
  constructor() {
    this.values = []
  }

  addMemento(memento) {
    this.values.push(memento)
  }

  getMemento(index) {
    return this.values[index]
  }
}

const careTaker = new Caretaker()

careTaker.addMemento(creator.save('hello'))
careTaker.addMemento(creator.save('hello world'))
careTaker.addMemento(creator.save('hello world !!!'))

console.log(creator.restore(careTaker.getMemento(1))) //hello world
```

Шаблонный метод
Template method

Тип: Поведенческий

Что это:

Определяет алгоритм, некоторые этапы которого делегируются подклассам. Позволяет подклассам переопределить эти этапы, не меняя структуру алгоритма.




```
//Template method

class Builder {
  build() {
    this.addEngine()
    this.installChassis()
    this.addElectronic()
    this.collectAccessories()
  }
}

class TeslaBuild extends Builder {
  addEngine() {
    console.log('Add Engine')
  }
  installChassis() {
    console.log('Install Tesla chassis')
  }
  addElectronic() {
    console.log('Add special electric')
  }
  collectAccessories() {
    console.log('Collect Accessories')
  }
}

class BmwBuild extends Builder {
  addEngine() {
    console.log('Add Bmw Engine')
  }
  installChassis() {
    console.log('Install Bmw chassis')
  }
  addElectronic() {
    console.log('Add special electric')
  }
  collectAccessories() {
    console.log('Collect Accessories')
  }
}
```

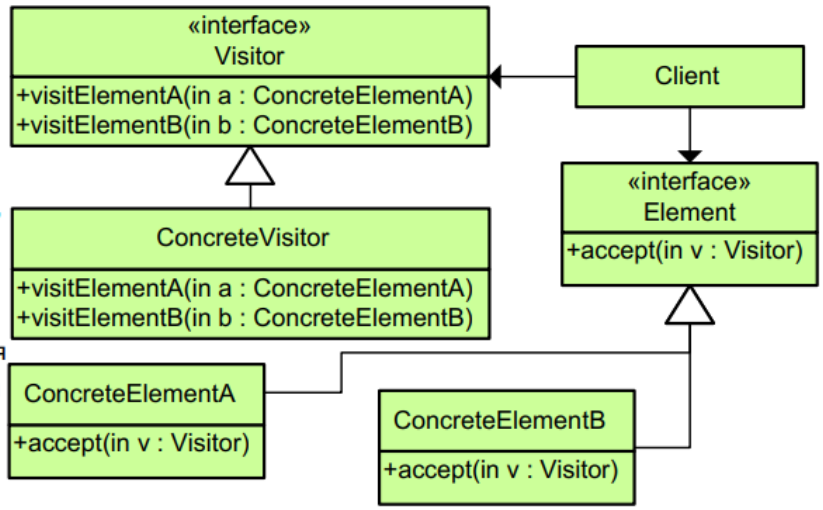
```
const teslaBuilder = new TeslaBuild()
const bmwBuilder = new BmwBuild()
teslaBuilder.build()

/*
Add Engine
Install Tesla chassis
Add special electric
Collect Accessories
*/

bmwBuilder.build()

/*
Add Bmw Engine
Install Bmw chassis
Add special electric
Collect Accessories
*/
```

Посетитель
Visitor

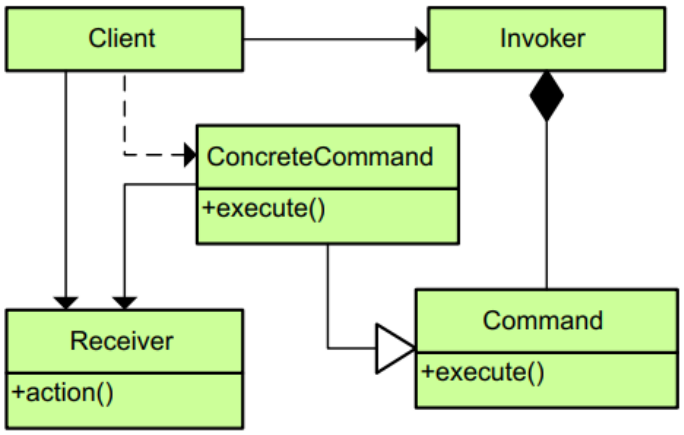
Тип: Поведенческий

Что это:

Представляет собой операцию, которая будет выполнена над объектами группы классов. Даёт возможность определить новую операцию без изменения кода классов, над которами эта операция проводится.




```
//Visitor

class Auto {
  accent(visitor) {
    visitor(this)
  }
}

class Tesla extends Auto {
  info() {
    return 'It is a Tesla car!'
  }
}

class Audi extends Auto {
  info() {
    return 'It is a Audi car!'
  }
}

function exportVisitor(auto) {
  if (auto instanceof Tesla)
    auto.export = console.log(`Export data: ${auto.info()}`)

  if (auto instanceof Audi)
    auto.export = console.log(`Export data: ${auto.info()}`)
}

const tesla = new Tesla()
const audi = new Audi()

tesla.accent(exportVisitor) // Export data: It is a Tesla car!
audi.accent(exportVisitor) //Export data: It is a Audi car!
```



Команда
Command

Тип: Поведенческий

Что это:

Инкапсулирует запрос в виде объекта, позволяя передавать их клиентам в качестве параметров, ставить в очередь, логировать а также поддерживает отмену операций.

```
//Command

class Driver {
  constructor(command) {
    this.command = command
  }

  execude() {
    this.command.execude()
  }
}

class Engine {
  constructor() {
    this.state = false
  }

  on() {
    this.state = true
  }

  off() {
    this.state = false
  }
}

class OnStartCommand {
  constructor(engine) {
    this.engine = engine
  }

  execude() {
    this.engine.on()
  }
}

class OnSwitchOffCommand {
  constructor(engine) {
    this.engine = engine
  }

  execude() {
    this.engine.off()
  }
}
```

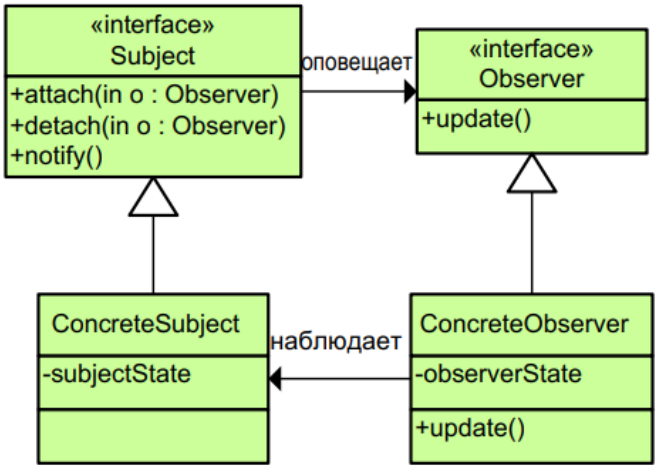
```
const engine = new Engine()

console.log(engine) //{ "state": false}

const onStartCommand = new OnStartCommand(engine)
const drive = new Driver(onStartCommand)

drive.execude()

console.log(engine) //{ "state": true}
```

Наблюдатель
Observer

Тип: Поведенческий

Что это:

Определяет зависимость “один ко многим” между объектами так, что когда один объект меняет своё состояние, все зависимые объекты оповещаются и обновляются автоматически.



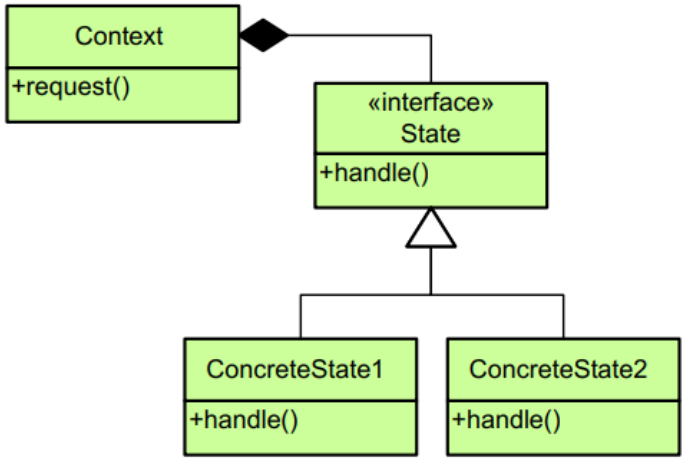

```
//Observer

class AutoMews {
  constructor() {
    this.news = ''
    this.actions = []
  }

  setNews(text) {
    this.news = text
    this.notifyAll()
  }

  notifyAll() {
    return this.actions.forEach((subs) => subs.inform(this))
  }

  register(observer) {
    this.actions.push(observer)
  }

  unregister(observer) {
    this.actions = this.actions.filter((el) => !(el instanceof observer))
  }
}

class Jack {
  inform(message) {
    console.log(`Jack has been informed about: ${message.news}`)
  }
}

class Max {
  inform(message) {
    console.log(`Max has been informed about: ${message.news}`)
  }
}

const autoNews = new AutoMews()

autoNews.register(new Jack())
autoNews.register(new Max())

autoNews.setNews('New Tesla price is 40 000')
// Jack has been informed about: New Tesla price is 40 000
// Max has been informed about: New Tesla price is 40 00
```

Состояние
State

Тип: Поведенческий

Что это:

Позволяет объекту изменять своё поведение в зависимости от внутреннего состояния.




```
//State

class OrderStatus {
  constructor(name, nextStatus) {
    this.name = name
    this.nextStatus = nextStatus
  }

  next() {
    return new this.nextStatus()
  }
}

class WaitingForPayment extends OrderStatus {
  constructor() {
    super('waitingForPayment', Shipping)
  }
}

class Shipping extends OrderStatus {
  constructor() {
    super('shipping', Delivered)
  }
}

class Delivered extends OrderStatus {
  constructor() {
    super('delivered', Delivered)
  }
}

class Order {
  constructor() {
    this.state = new WaitingForPayment()
  }

  nextState() {
    this.state = this.state.next()
  }

  cancelOrder() {
    this.state.name === 'waitingForPayment'
      ? console.log('Order is canceled')
      : console.log('Order can not be canceled')
  }
}

const myOrder = new Order()
console.log(myOrder.state.name) //waitingForPayment
myOrder.cancelOrder() //Order is canceled

myOrder.nextState()
console.log(myOrder.state.name) //shipping

myOrder.nextState()
console.log(myOrder.state.name) //delivered
myOrder.cancelOrder() //Order can not be canceled
```