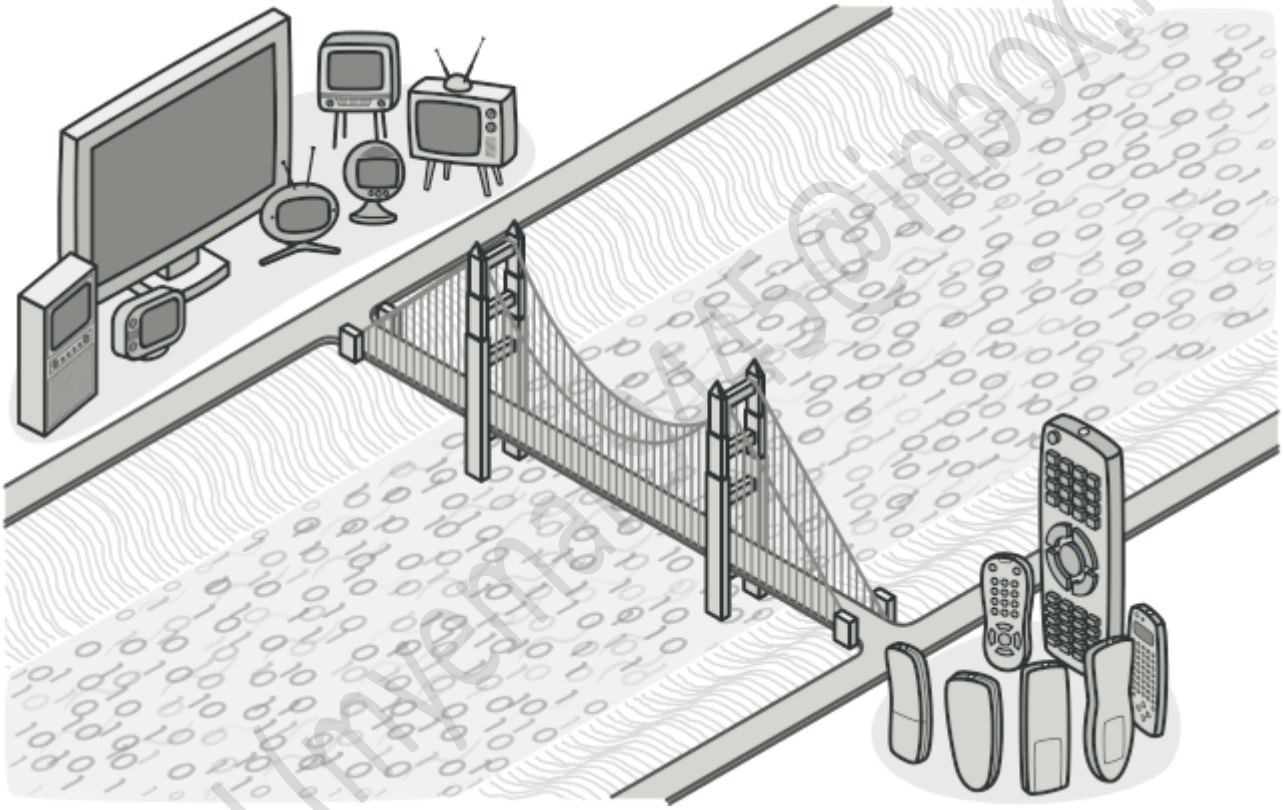
Bridge

Maqsad

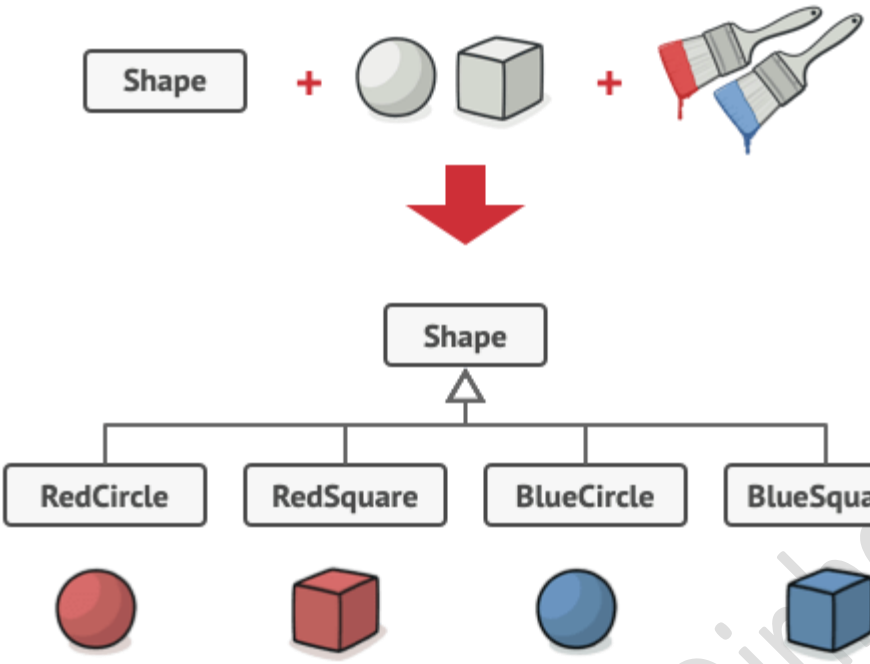
Bridge structural design patternlar turkumiga kiradi. U sizga katta o’lchamli klass yoki o’zaro yaqin bog’langan klasslar to’plamini ikkita alohida iyerarxiyaga – bir biridan mustaqil holda yoziladigan abstraksiya(abstraction) va implementatsiya(implementation)ga ajratib beradi.



Muammo

Abstraksiya? Implementatsiya? Qo’rqinchli eshitilyaptimi? Tinchlanib olib, oddiy misolni ko’rib chiqing.

Aytaylik sizda geometrik *Shape* klassi va uning ikkita *Circle* va *Square* bola klasslari bor bo’lsin. Siz bu klass iyerarxiyasini ranglar bilan birgalikda qo’llashni xohlayapsiz. Shu sababli, Red va Blue bola klasslarni yaratishni reja qildingiz. Biroq, sizda o’zi ikkita bola klass borligi uchun *BlueCircle* va *RedSquare* kabi to’rtta klasslar kombinatsiyasini yaratishingiz kerak bo’ladi.



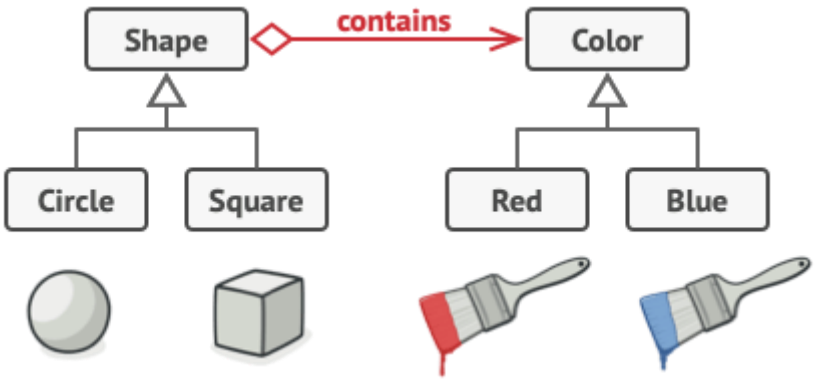
Klass kombinatsiyalari soni geometrik progressiya bilan oshadi

Yangi shakl va ranglarning turi qo’shilishi iyerarxiyani eksponensial tarzda oshiradi. Masalan, piramida shaklini qo’shish uchun siz har bir rang uchun ikkita bola klassni yaratishingiz kerak bo’ladi. Shundan keyin, yana yangi rang qo’shmoqchi bo’lsangiz har bir shakl uchun uchta bola klass qo’shishingizga to’g’ri keladi. Shu tarzda ketganimiz sari, holat yomonlashib borveradi.

Yechim

Bu muammoning paydo bo’lishiga sabab shaklning klassini ikkita mustaqil yo’nalish – shakl va rangda kengaytirib ketishimiz bo’ladi. Bu merosxo’rlikdagi eng ko’p uchraydigan muammolardan hisoblanadi.

Bridge paterni bu muammoni merosxo’rlikdan obyekt kompozitsiyasiga o'tish orqali hal qiladi. Bu o’lchamlardan birini alohida klass iyerarxiyasiga o’tkazishingiz kerakligini anglatadi. Natijada, orginal klasslar o’zining xususiyatlari va holatini bitta klass ichida ega bo’lish o’rniga yangi iyerarxiyadagi obyektga murojaat qiladi.



Klass iyerarxiyasi kattalashib ketmasligini oldini olish uchun uni bir nechta o'zaro bog'langan iyerarxiyalarga ajratib chiqiladi.

Shu yondashuvga amal qilib, rangga bog'liq bo'lgan kodlarni ikkita bola klass – *Red* va *Blue* klasslariga chiqarib olamiz. *Shape* klassi esa rang obyektlarining biriga murojaat qiladi. Natijada, kerakli shakl rangga bog'liq bo'lgan ishlarni o'ziga bog'langan rang obyektiga topshiradi. Mana shu bog'lanish Shape va Color klasslari o'rtasida ko'prik (bridge) vazifasini bajaradi. Keyinchalik yangi rang qo'shmoqchi bo'lsak ham shakl iyerarxiyasini buzishga hojat qolmaydi.

Abstraksiya va Implementatsiya

GoF kitobida Abstraksiya va Implementatsiya Bridge paterni tavsifining bir qismi sifatida tushuntiriladi. Mening fikrimcha, bu juda ilmiy eshitiladi va paternni aslidagidan ko'ra murakkabroq qilib ko'rsatadi. Shakl va ranglar misolida, GoF kitobida keltirilgan tushunish qiyin bo'lgan so'zlarning ma'nosini ko'rib chiqaylik.

Abstraksiya (yana *interface* deb ham ataladi) – bu biror narsaning yuqori darajadagi boshqaruv qatlami bo'lib hisoblanadi. Bu qatlam o'z-o'zidan biror ish bajarmaydi. U qilinadigan ishni *implementatsiya* qatlami(yana *platform* deb ham ataladi)ga topshiradi.

Shuni yodda tutingki, biz hozir siz ishlatadigan dasturlash tilingizdagi interfeyslar yoki abstract klaslar haqida gapirmayapmiz. Bular bir xil narsalar emas.

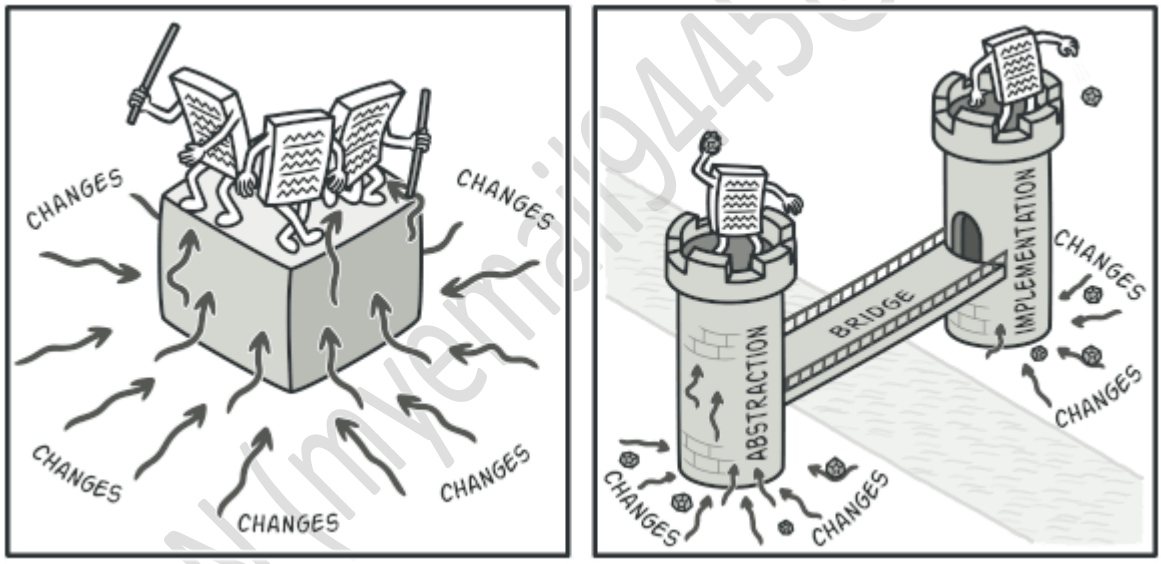
Haqiqiy ilovalar misolida gapiradigan bo'lsak, abstraksiya foydalanuvchi grafik interfeysi(FGI, ing – Graphical User Interface - GUI) orqali berilsa, implementatsiya

foydalanuvchining harakatlariga javob qaytarishda FGI qatlami chaqiradigan operatsion tizim kodi (API) bo'ladi.

Umumiy gapiradigan bo'lsak, siz bunday ilovani ikkita mustaqil yo'nalishda kengaytirishingiz mumkin:

- Turli xildagi FGIlarga ega bo'lib (masalan, doimiy mijozlar va adminlar uchun moslangan FGI).
- Turli xildagi APIlarini yaratib (masalan, ilovani Windows, Linux va macOS da ishlaydigan qilib).

Eng yomon holatda, bu ilova lag'mon solingan kosaga o'xshaydi: yuzlab holatlar turli xildagi FGIlarni turlicha APIlarga bog'lab turadi.

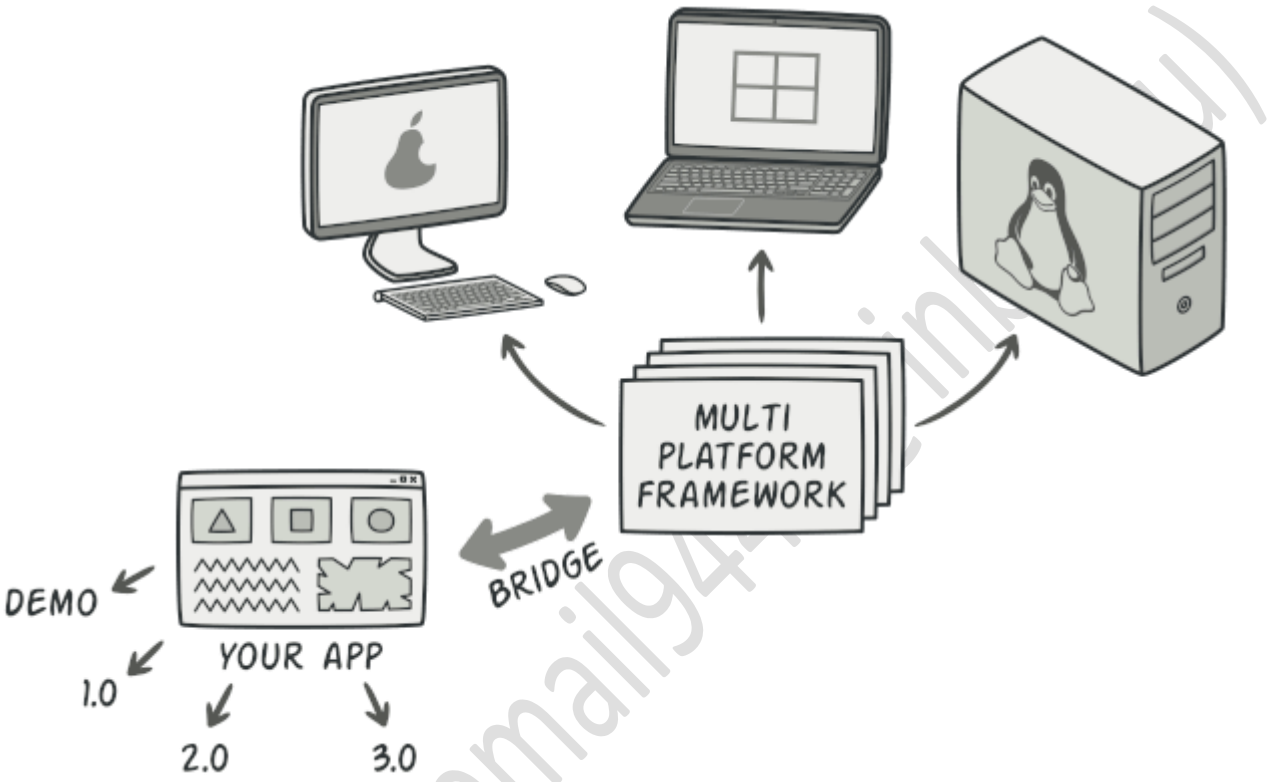


Butun kodni tushunishingiz kerakligi sababli monolitik kodda oddiy o'zgarish qilish ham juda qiyin bo'lib qoladi. O'zgarishlarni kamaytirish uchun ularni kichik modullarga ajratish ancha oson.

Bunday tartibsizlikni to'g'rilash uchun maxsus interfeys-platforma kombinatsiyalariga bog'liq bo'lgan kodlarni alohida klasslarga o'tkazish mumkin. Biroq, tez orada, bunday klasslar juda ko'p ekanligini bilib qolasiz. Yangi FGI qo'shish yoki turlicha APIlarni ishlatish juda ko'plab klasslarni qo'shishni talab qilganligi sababli klass iyerarxiyasi eksponensial tarzda o'sib boradi.

Bu muammoni Bridge paterni orqali yechishga harakat qilib ko’raylik. Ushbu patern klasslarni ikkita iyerarxiyaga ajratishni taklif qiladi:

- Abstraksiya: ilovaning FGI qatlami.
- Implementatsiya: operatsion tizimning APIlari.



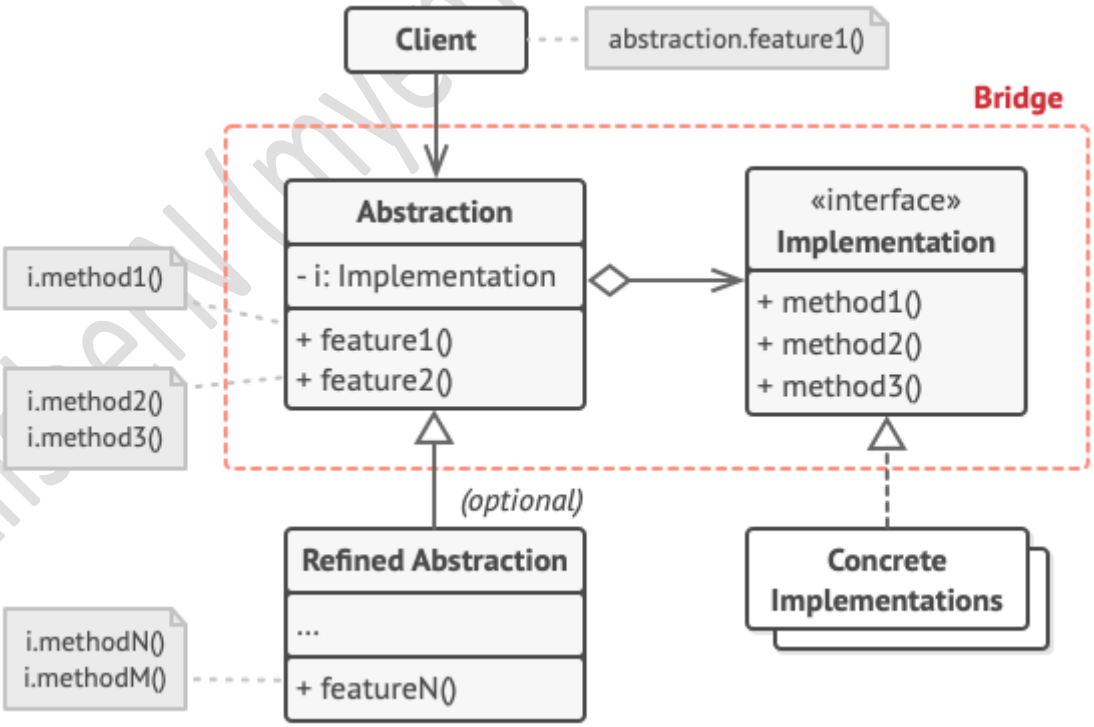
Kross-platformali dastur qurishning yo’llaridan biri

Abstraksiya obyekti qilinadigan ishni bog’langan implementatsiya obyektiga topshirib, ilovaning ko’rinishini boshqaradi. Agar bir xildagi FGIni ham Windows, ham Linuxda ishlashiga imkon bergan holda turli xildagi implementatsiyalar umumiy interfeysni ishlatsa, ular bir birining o’rnini bosa oladi.

Natijada, FGI klasslarni APIga tegishli bo’lgan klasslarni almashtirmay turib o’zgartirish mumkin bo’ladi. Shu bilan birga, dasturni boshqa operatsion tizimda ishlaydigan qilish uchun faqatgina implementatsiya iyerarxiyasiga bola klassni qo’shish yetarli bo’ladi.

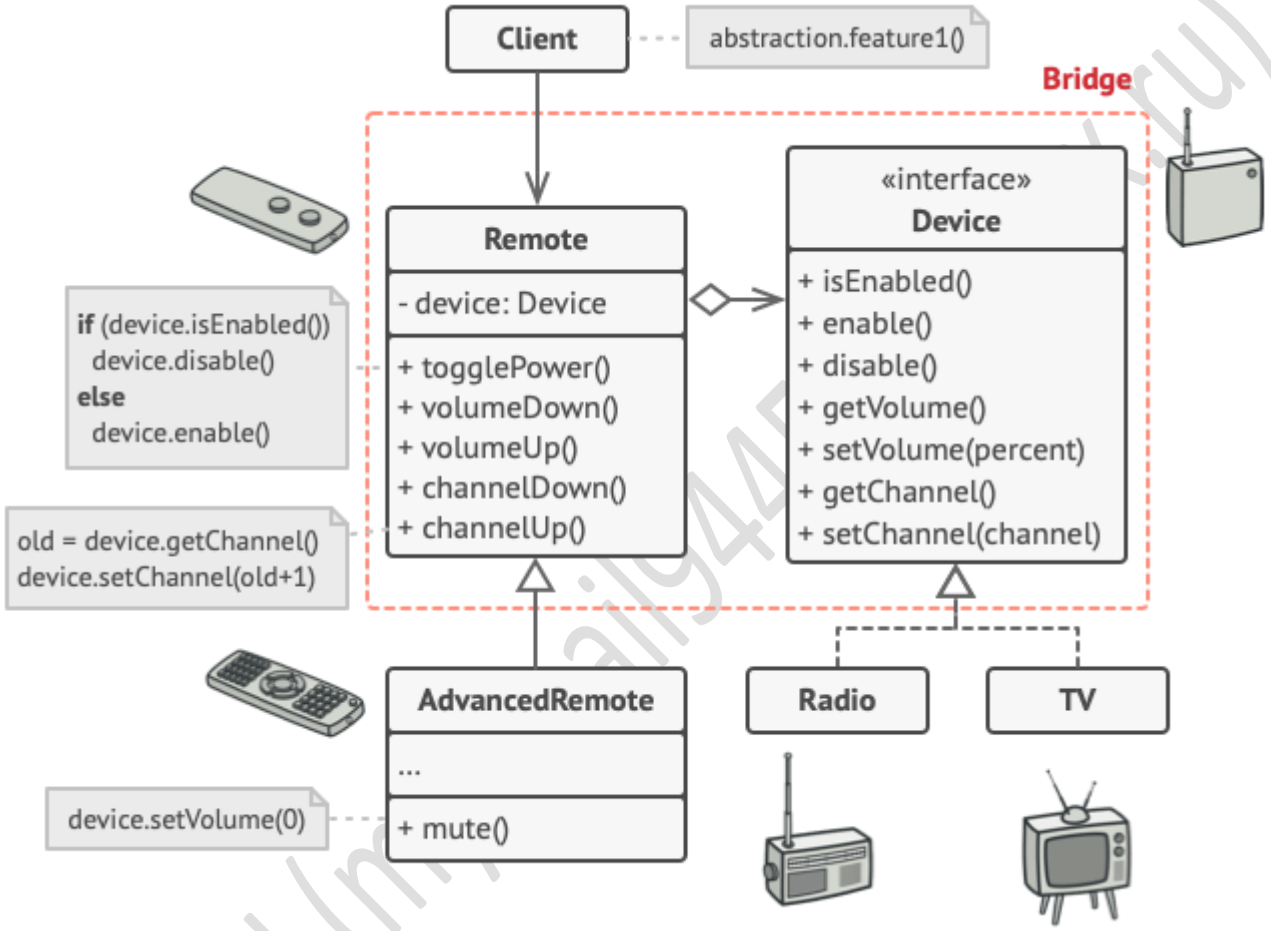
Tuzilishi

1. Abstraksiya (**Abstraction**) yuqori darajali boshqaruvni beradi. U quyi darajada ishlovchi implementatsiya obyektiga suyanadi.
2. Implementatsiya(**Implementation**) barcha interfeys ishlatuvchi implementatsiya(**concrete implementation**)lar uchun umumiy bo'lgan interfeysni e'lon qiladi. Abstraksiya implementatsiya obyekti bilan faqatgina shu yerda e'lon qilingan metodlar orqali bog'lanadi.
3. Interfeys ishlatuvchi implementatsiya(**concrete implementation**)lar platformaga tegishli bo'lgan kodlarni o'z ichiga oladi.
4. Takomillashtirilgan abstraksiya(**refined abstraction**)lar boshqaruv logikasining turli xildagi ko'rinishlarini beradi. O'zlarining ota klasslariga o'xshab ular umumiy impelementatsiya interfeysi orqali turli xildagi impelementatsiyalar bilan ishlaydi.
5. Odatda, Mijoz(**Client**)ni faqat abstraksiya bilan ishlash qiziqtiradi. Biroq, abstraksiya obyekti bilan impelmentatsiya obyektlaridan birini bog'lash mijozning ishi hisoblanadi.



Psevdokod

Quyidagi misol Bridge paterni qurilmalarni ularning pultlari bilan boshqaradigan dasturning monolitik kodini ajratishga qanday qilib yordam berishi tasvirlangan. *Device* klasslari impelementatsiya sifatida ishlasa, *Remote*lar abstraksiya sifatida ishlaydi.



Orginal klass iyerarxiyasi ikkita qismga bo'linadi: qurilmalar va pultlar

Asosiy masofadan boshqarish(pult) klassi ushbu klassni qurilma obyekti bilan bog'lash uchun xususiyat e'lon qiladi. Barcha pultlar qurilmalar bilan umumiy qurilma interfeysi orqali ishlaydi. Bu bir xil pult bilan qurilmaning bir nechta turini qo'llab-quvvatlashga imkon beradi.

Masofadan boshqarish klasslari kodini qurilma klasslaridan mustaqil holda yozishingiz mumkin. Buning uchun yangi pult bola klassini yaratish kerak bo'ladi. Masalan, asosiy pultda faqatgina ikkita tugmasi bo'lishi mumkin, biroq siz

qo’shimcha batareya yoki sensorli ekran kabi yangi xususiyatlarni ham qo’shsangiz bo’ladi.

Mijoz kodi masofadan boshqarish pultining kerakli turini uning konstruktori orqali ma'lum bir qurilma obyekti bilan bog'laydi.

```
// The "abstraction" defines the interface for the "control"
// part of the two class hierarchies. It maintains a reference
// to an object of the "implementation" hierarchy and delegates
// all of the real work to this object.
class RemoteControl is
    protected field device: Device
    constructor RemoteControl(device: Device) is
        this.device = device
    method togglePower() is
        if (device.isEnabled()) then
            device.disable()
        else
            device.enable()
    method volumeDown() is
        device.setVolume(device.getVolume() - 10)
    method volumeUp() is
        device.setVolume(device.getVolume() + 10)
    method channelDown() is
        device.setChannel(device.getChannel() - 1)
    method channelUp() is
        device.setChannel(device.getChannel() + 1)


// You can extend classes from the abstraction hierarchy
// independently from device classes.
class AdvancedRemoteControl extends RemoteControl is
```

```
    method mute() is
        device.setVolume(0)


// The "implementation" interface declares methods common to all
// concrete implementation classes. It doesn't have to match the
// abstraction's interface. In fact, the two interfaces can be
// entirely different. Typically the implementation interface
// provides only primitive operations, while the abstraction
// defines higher-level operations based on those primitives.
interface Device is
    method isEnabled()
    method enable()
    method disable()
    method getVolume()
    method setVolume(percent)
    method getChannel()
    method setChannel(channel)


// All devices follow the same interface.
class Tv implements Device is
    // ...

class Radio implements Device is
    // ...


// Somewhere in client code.
tv = new Tv()
remote = new RemoteControl(tv)
```

```
remote.togglePower()

radio = new Radio()
remote = new AdvancedRemoteControl(radio)
```

Amaliyotda qo’llanilishi

- **Bridge** paternidan bir xil funksionallikning turli ko’rinishlariga ega bo’lgan monolitik klassni bo’lib qurishda foydalaning (masalan, agar biror klass turli xildagi ma’lumotlar ombori serverlari bilan ishlaydigan bo’lsa).

  Klass qanchalik kattalashib borsa, u bilan ishlash shunchalik qiyinlashib boradi. Funksionallikning bitta ko’rinishini o’zgartirish butun klass bo’ylab o’zgarishlar qilishga olib kelishi mumkin. Natijada, bu xatoliklar paydo bo’lishiga sabab bo’ladi.

  Bridge paterni monolitik klassni turli xildagi klass iyerarxiyalariga bo’lish imkonini beradi. Bunda siz har bir klass iyerarxiyasini mustaqil holda o’zgartirishingiz mumkin bo’ladi. Bu yondashuv kodni kengaytirish va o’zgartirishni soddalashtiradi, hamda kodning buzilish xavfini kamaytiradi.
- **Bridge** paterndan biror klassni turli xildagi mustaqil yo’nalishlarda kengaytirishda foydalaning.
- Implementatsiyalarni ishlash jarayonida almashtirish imkoniyatiga ega bo'lishingiz kerak bo'lsa, Bridge paterndan foydalaning**.**

  Garchi bu ixtiyoriy bo’lsa ham, bridge paterni abstraksiya ichida implementatsiya obyektini almashtirish imkonini beradi. Buni klass xususiyatga yangi qiymat berish kabi oson amalga oshirish mumkin.

  Aytganday, bu oxirgi ma’lumot **Bridge** paternni **Strategy** patern bilan chalkashtirishiga asosiy sabab bo’ladi.

Ishlatilishi

1. Klassladagi ajratilishi kerak bo’lgan qismlarni aniqlab oling. Bu mustaqil qismlar abstraction/platform, domain/infrastructure, front-end/back-end yoki interface/implementation lar bo’lishi mumkin.
2. Mijoz qanaqa operatsiyalarni bajarishini ko’rib oling va ularni asosiy klassda e’lon qiling.
3. Barcha platformalardagi amalga oshirish mumkin bo’lgan operatsiyalarni aniqlang. Umumiy implementatsiya interfeysida abstraksiya kerak bo'lgan operatsiyalarni e'lon qiling.
4. Barcha platformalar uchun interfeys ishlatuvchi implementatsiya klasslarini yarating, ammo ular implementatsiya interfeysini ishlatayotganligiga amin bo’ling.
5. Abstraksiya klassi ichida implementatsiya tipi uchun xususiyat qo’shing. Abstraksiya ishning ko’p qismini o’sha xususiyatga murojaat qiladigan implementatsiya obyektiga topshiradi.
6. Agar sizda yuqori-darajali logikaning ko’plab variantlari bo’ladigan bo’lsa, asosiy abstraksiya klassini kengaytirish orqali har bir variant uchun takommillashtirilgan abstraksiya(refined abstraction)larni yarating.
7. Mijoz kodi implementatsiya obyektini bittasini boshqasi bilan bog’lash uchun abstraksiyaning konstruktoriga berishi kerak. Shundan keyingina, mijoz implementatsiya haqida unutib, faqat abstraksiya obyekti bilan ishlashi mumkin.

Afzallik va kamchiliklari

Afzalliklari	Kamchiliklari
Platformadan mustaqil bo’lgan klasslar va dasturlarni yaratishingiz mumkin	Yaxlit klassga paternni qo’llab, kodni murakkablashtirib yuborishingiz mumkin.
Mijoz kodi yuqori darajadagi abstraksiyalar bilan ishlaydi.	

Ochiq/Yopiqlik Tamoyili. Bir biridan mustaqil ravishda yangi abstraksiyalar va implementatsiyalarni e’lon qilishingiz mumkin

Yagona Javobgarlik Tamoyili. Abstraksiyada yuqori darajali logikaga, implementatsiyada esa platforma detallariga diqqat qaratishingiz mumkin.

Boshqa paternlar bilan bog’liqligi

- **Bridge** dasturning qismlarini bir biridan mustaqil holda yaratishga imkon beradi. Boshqa tomondan, **Adapter** umumiy holda dasturdagi o’zaro mos tushmaydigan klasslarni birgalikda ishlashini ta’minlaydi.
- **Bridge**, **State**, **Strategy** (va qaysidir darajada **Adapter**) paternlar juda o’xshash tuzilishga ega. Darhaqiqat, bu barcha paternlar o’z ishini boshqa obyektlarga topshirgan kompozitsiyalarga asoslangan. Biroq, ularning barchasi turlicha muammolarni yechadi. Patern kodingizni muayyan tarzda tuzib beruvchi retsept emas. U boshqa dasturchilarga patern hal qiladigan muammoni yetkazadi.
- **Abstract Factory** paternni **Bridge** bilan birgalikda ishlatishingiz mumkin. Bu juftlik Bridge tomonidan e’lon qilingan ba’zi abstraksiyalar faqat maxsus implementatsiyalar bilan ishlay olganda foydali bo’ladi. Bu holatda, *Abstract Factory* ushbu bog’lanishlarni enkapsulyatsiyalaydi va mijozdan murakkablikni yashiradi.
- **Builder** bilan **Bridge**ni birlashtirishingiz ham mumkin: director klass abstraksiya rolini o’ynasa, turli xil builderlar implementatsiya sifatida ishlaydi.

Namuna kod

```
<?php
```

```php
namespace RefactoringGuru\Bridge\Conceptual;

/**
 * The Abstraction defines the interface for the "control" part of the two class
 * hierarchies. It maintains a reference to an object of the Implementation
 * hierarchy and delegates all of the real work to this object.
 */
class Abstraction
{
    /**
     * @var Implementation
     */
    protected $implementation;

    public function __construct(Implementation $implementation)
    {
        $this->implementation = $implementation;
    }

    public function operation(): string
    {
        return "Abstraction: Base operation with:\n" .
            $this->implementation->operationImplementation();
    }
}

/**
 * You can extend the Abstraction without changing the Implementation classes.
 */
class ExtendedAbstraction extends Abstraction
{
```

```
    public function operation(): string
    {
        return "ExtendedAbstraction: Extended operation with:\n" .
            $this->implementation->operationImplementation();
    }
}

/**
 * The Implementation defines the interface for all implementation classes. It
 * doesn't have to match the Abstraction's interface. In fact, the two
 * interfaces can be entirely different. Typically the Implementation interface
 * provides only primitive operations, while the Abstraction defines higher-
 * level operations based on those primitives.
 */
interface Implementation
{
    public function operationImplementation(): string;
}

/**
 * Each Concrete Implementation corresponds to a specific platform and
 * implements the Implementation interface using that platform's API.
 */
class ConcreteImplementationA implements Implementation
{
    public function operationImplementation(): string
    {
        return "ConcreteImplementationA: Here's the result on the platform A.\n";
    }
}
```

```php
class ConcreteImplementationB implements Implementation
{
    public function operationImplementation(): string
    {
        return "ConcreteImplementationB: Here's the result on the platform B.\n";
    }
}

/**
 * Except for the initialization phase, where an Abstraction object gets linked
 * with a specific Implementation object, the client code should only depend on
 * the Abstraction class. This way the client code can support any abstraction-
 * implementation combination.
 */
function clientCode(Abstraction $abstraction)
{
    // ...

    echo $abstraction->operation();

    // ...
}

/**
 * The client code should be able to work with any pre-configured abstraction-
 * implementation combination.
 */
$implementation = new ConcreteImplementationA();
$abstraction = new Abstraction($implementation);
clientCode($abstraction);
```

```
echo "\n";

$implementation = new ConcreteImplementationB();
$abstraction = new ExtendedAbstraction($implementation);
clientCode($abstraction);
```

```
Abstraction: Base operation with:
ConcreteImplementationA: Here's the result on the platform A.

ExtendedAbstraction: Extended operation with:
ConcreteImplementationB: Here's the result on the platform B.
```