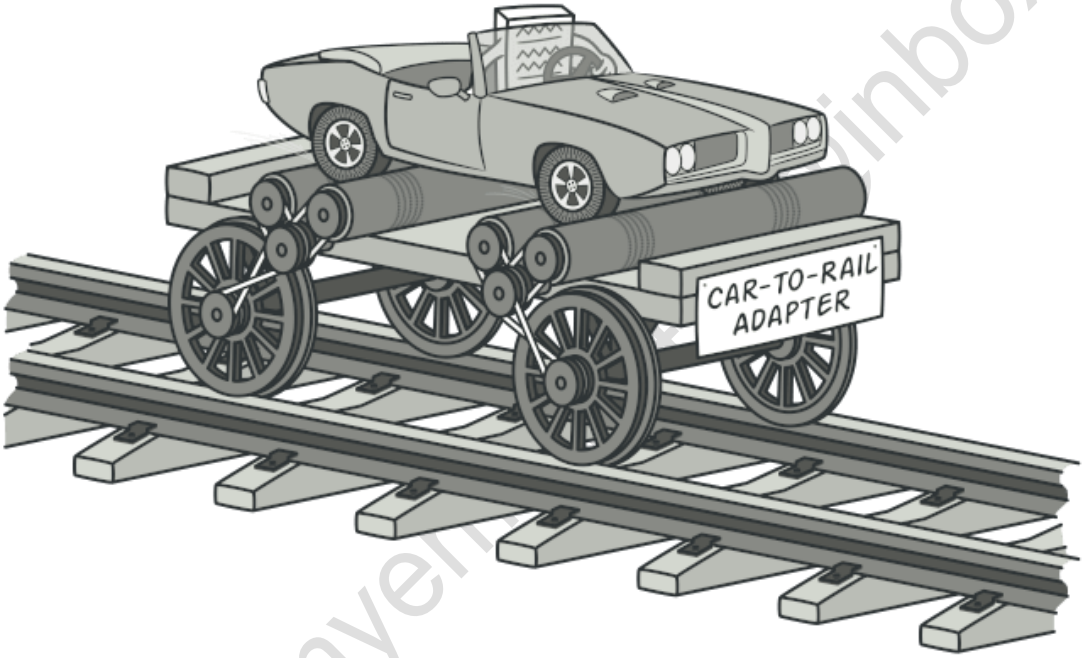
Adapter

Wrapper nomi bilan ham ma’lum.

Maqsadi

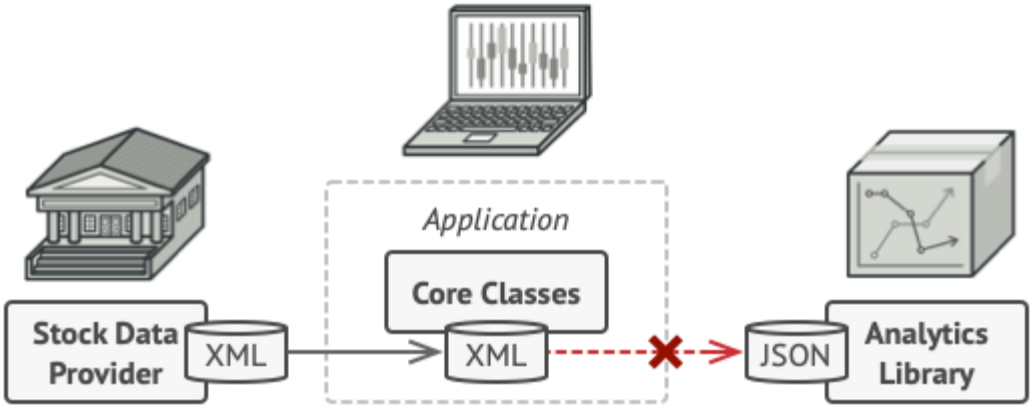
Adapter structural design pattern guruhiga mansub bo’lgan dizayn patern bo’lib, u obyektlarga ularga mos kelmaydigan interfeyslarni o’zaro moslashtirish imkonini beradi.



Muammo

Tasavvur qiling siz fond bozorini tahlil qilib boruvchi dastur yaratyapsiz. Dasturingiz fond ma’lumotlarini bir nechta manbaadan XML formatida yuklab oladi va foydalanuvchiga chart yoki diagramma ko’rinishida ko’rsatadi.

Ozroq vaqtdan keyin, ba’zi joylarda, dasturga qo’shimcha ma’lumotlarni o’zi tahlil qilib beradigan kutubxona qo’shishni xohlab qoldingiz deylik. Ammo, shu paytda bitta muammo paydo bo’ladi: siz qo’shmoqchi bo’lgan kutubxona faqat JSON ko’rinishidagi ma’lumot bilan ishlaydi.



Tahlil qiluvchi kutubxonani to'g'ridan to'g'ri foydalana olmaysiz, chunki bu kutubxona sizning dasturingiz ishlatadigan ma'lumot ko'rinishi bilan ishlamaydi.

Kutubxonani XML bilan ishlaydigan qilishingiz mumkin. Biroq, bu kutubxona bilan ishlaydigan ayrim kodlarni buzib qo'yishi ehtimoli bor. Yanayam yomonrog'i, siz kutubxonaning kodini o'zgartira olmasligingiz ham mumkin. Bu esa ushbu yondashuvni ham imkonsiz ekanligini ko'rsatadi.

Yechim

Yechim sifatida, adapter (moslashtirgich) yaratishingiz mumkin. Bu maxsus obyekt bo'lib u o'zimiz ishlatmoqchi bo'lgan biror boshqa obyektning interfeysini o'zgartirib beradi. Shunda boshqa obyekt uni tushunishi mumkin bo'ladi.

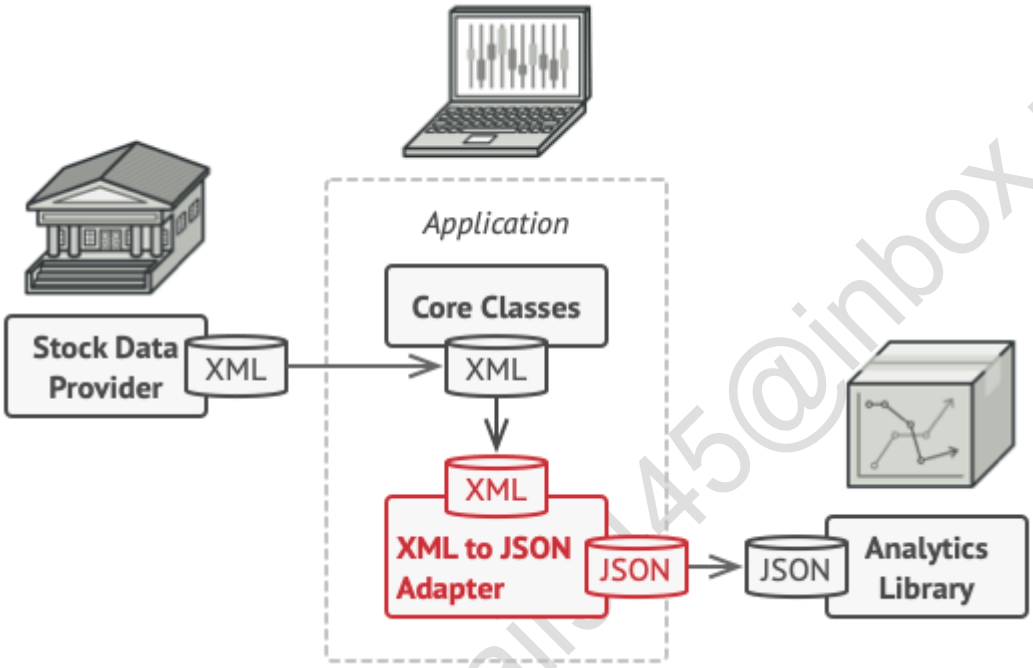
O'zgartirish murakkabligini foydalanuvchidan yashirish maqsadida adapter obyektlardan biriga biriktiriladi. Bunda biriktirilgan obyekt adapter borligini umuman bilmaydi ham. Masalan, metr va kilometrlar bilan ishlaydigan obyektga undagi ma'lumotlarni qadam va milga aylantirib beradigan adapterni biriktirishingiz mumkin.

Adapter nafaqat ma'lumot ko'rinishini o'zgartiradi, balki obyektlarni turli xildagi interfeyslar bilan moslashishiga ham yordam beradi:

1. Adapter mavjud obyektlardan biri ishlatadigan interfeysni oladi
2. Ushbu interfeysdan foydalangan holda, mavjud obyekt bemalol adapterning metodini chaqirishi mumkin.

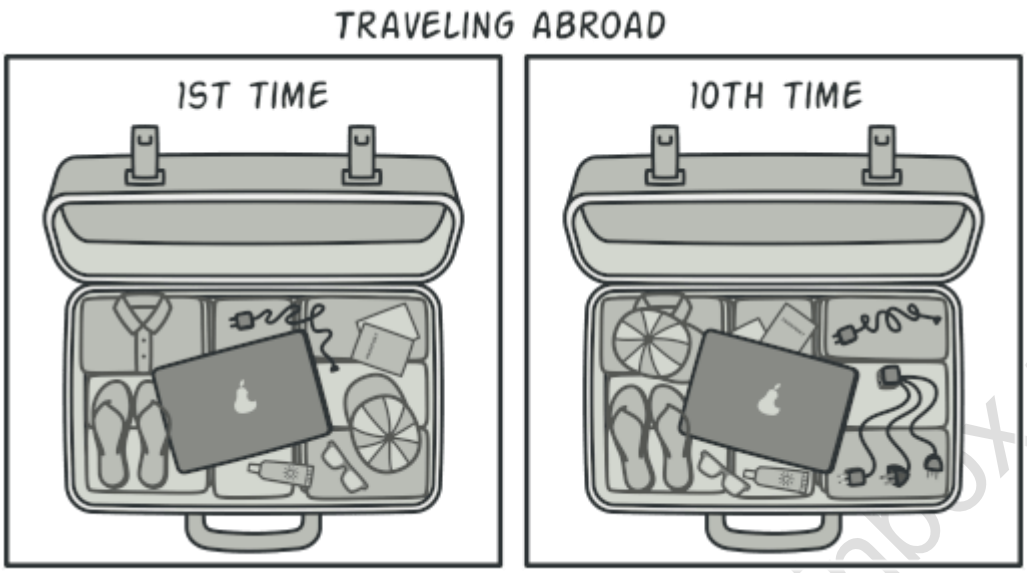
3. Adapter chaqirilganidan so’ng, u so’rovni ikkinchi obyektga xuddi shu obyekt tushunadigan ko’rinishida yuboradi.

Ba’zan, har ikkala tomondagi obyekt bilan ishlay oladigan adapterni ham yaratish mumkin.



Yana o’zimizning fond bozori dasturimizga qaytaylik. Mos kelmaydigan ma’lumot ko’rinishlari muammosini hal qilish uchun kodingiz to’g’ridan to’g’ri ishlaydigan tahlil qiluvchi kutubxonaning har bir klassi uchun XMLdan JSONga o’tkazuvchi adapter yaratasiz. Keyin, yaratgan adapteringiz orqali kutubxonaga bog’lanish uchun kodingizni o’zgartirasiz. Adapter chaqirilganda u kiruvchi XML ma’lumotlarni JSON ko’rinishiga o’tkazadi va bu ma’lumotlarni o’ziga biriktirilgan tahlil qiluvchi obyektga beradi.

Hayotiy misol



Chemadonning chet elga sayotdan oldingi va keyingi ko’rinishi

Birinchi marta Qo’shma Shtatlardan Yevropaga sayohat qilganingizda, noutbukingizni zaryadlashda muammoga duch kelasiz. Chunki zaryadlovchi qurilma shnuri standartlari turli davlatlarda turlicha bo’ladi. Shu sababli ham Qo’shma Shtatlarda ishlatadigan zaryadlash qurilmangizni Germaniyada ishlata olmaysiz. Bu muammoni adapter shnuri bilan hal qilishingiz mumkin bo’ladi.

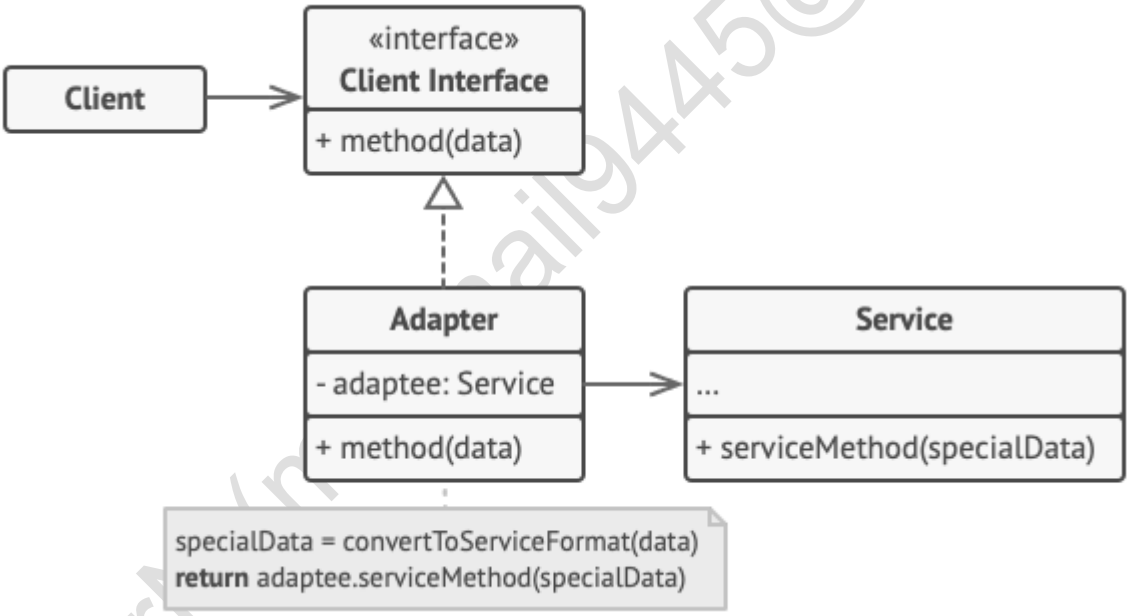
Tuzilishi

Adapter obyekti

Bunday ishlatilish obyekt kompozitsiyasi tamoyilidan foydalanadi: adapter bitta obyektning interfeysini ishlatadi, boshqasini esa o’ziga biriktiradi. Bu barcha mashhur dasturlash tillarida ishlatiladi.

1. Mijoz – bu dasturning biznes logikasini o’z ichiga olgan klass.
2. Mijoz interfeysi boshqa klasslar mijoz kodi bilan o’zaro hamkorlikda ishlaydigan kelishuvni tavsiflaydi.
3. Service klassi ham foydali klass hisoblanadi (odatda, tashqi kutubxonalar yoki qo’llab quvvatlash to’xtatilgan kodlarda). Mijoz bu klassning interfeysi mijozga mos kelmaganligi sababli bu klassdan to’g’ridan to’g’ri foydalana olmaydi.

4. Adapter ham mijoz, ham servis bilan ishlovchi klass bo’lib u mijoz interfeysini ishlatsa, o’zi servis obyektiga birikadi. Mijoz adapter interfeysi orqali adapterni chaqiradi va o’z navbatida adapter mijozdan kelgan barcha chaqirishlarni o’zi birikkan servisga servis tushunadigan ko’rinishda o’zgartirib yuboradi.
5. Mijoz kodi, toki u mijoz interfeysi orqali adapter bilan ishlamagunicha, concrete (aniq) adapterga biriktirilmaydi. Shu sababli ham, kodni buzmasdan turib yangi turdagi adapterlarni dasturga qo’shishingiz mumkin bo’ladi. Bu interfeys yoki servis klassi o’zi yoki joylashgan joyi o’zgartirilganda foydali bo’ladi. Ya’ni, mijoz kodini o’zgartirmasdan turib yangi adapter klassini yaratishingiz mumkin bo’ladi.

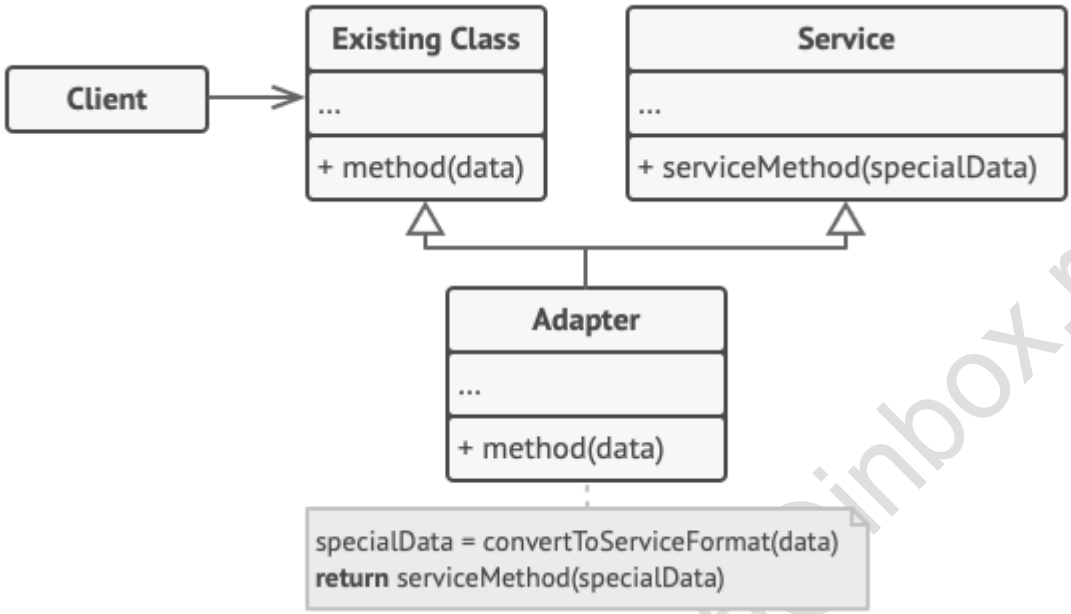


Adapter klassi

Bunday ishlatish merosxo’rlikdan foydalanadi: adapter bir vaqtning o’zida har ikkala obyektning interfeysidan foydalanadi. Shuni yodda tutingki, bunday yondashuv faqatgina C++ ga o’xshagan ko’p merosxo’rlik xususiyatiga ega bo’lgan dasturlash tillarida ishlatilishi mumkin.

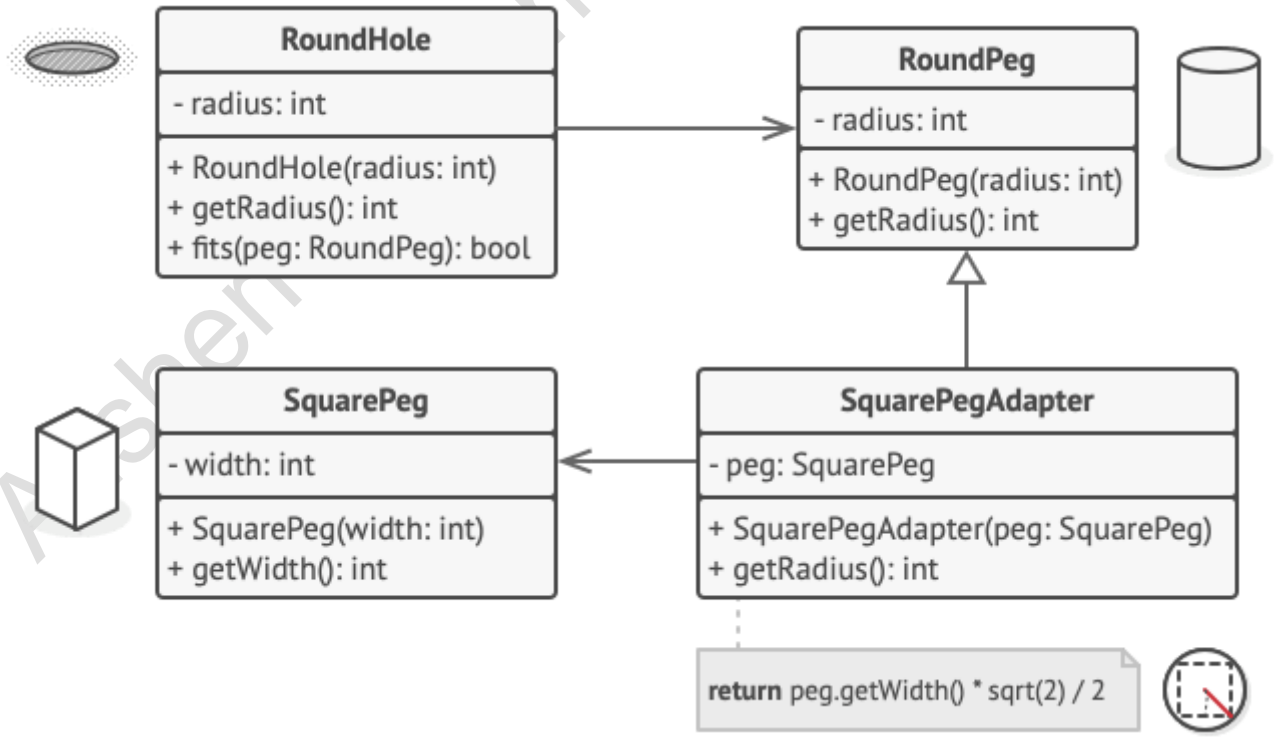
1. Adapter klassi hech qanday obyektni biriktirib olishi shart emas. Chunki, u mijozdan ham, servisdan ham ularning xususiyatlarni meros qilib oladi.

Moslashtirish(adaptation) qayta yozilgan (overridden) metodlarda amalga oshiriladi. Natijaviy adapter esa mijozning klassida ishlatiladi.



Psevdokod

Adapter patern uchun berilgan quyidagi misol kvadratik ustun va aylana tuynuklar orasidagi klassik konflikt (to'qnashuv)ga asoslangan.



Kvadrat ustunlarni aylana tuynuklarga moslashtirish

Bu yerda adapter kvadratik ustun aylana tuynukka mos kelishi uchun uni radiusi kvadrat asosning diametri yarmiga teng bo’lgan aylana asosli ustunga almashtiradi. Shu holatda ustun tuynukka mos keladi (yuqoridagi rasmga qarang).

```
// Say you have two classes with compatible interfaces:
// RoundHole and RoundPeg.
class RoundHole is
    constructor RoundHole(radius) { ... }

    method getRadius() is
        // Return the radius of the hole.

    method fits(peg: RoundPeg) is
        return this.getRadius() >= peg.getRadius()

class RoundPeg is
    constructor RoundPeg(radius) { ... }

    method getRadius() is
        // Return the radius of the peg.


// But there's an incompatible class: SquarePeg.
class SquarePeg is
    constructor SquarePeg(width) { ... }

    method getWidth() is
        // Return the square peg width.
```

```
// An adapter class lets you fit square pegs into round holes.
// It extends the RoundPeg class to let the adapter objects act
// as round pegs.
class SquarePegAdapter extends RoundPeg is
    // In reality, the adapter contains an instance of the
    // SquarePeg class.
    private field peg: SquarePeg

    constructor SquarePegAdapter(peg: SquarePeg) is
        this.peg = peg

    method getRadius() is
        // The adapter pretends that it's a round peg with a
        // radius that could fit the square peg that the adapter
        // actually wraps.
        return peg.getWidth() * Math.sqrt(2) / 2



// Somewhere in client code.
hole = new RoundHole(5)
rpeg = new RoundPeg(5)
hole.fits(rpeg) // true

small_sqpeg = new SquarePeg(5)
large_sqpeg = new SquarePeg(10)
hole.fits(small_sqpeg) // this won't compile (incompatible types)

small_sqpeg_adapter = new SquarePegAdapter(small_sqpeg)
large_sqpeg_adapter = new SquarePegAdapter(large_sqpeg)
hole.fits(small_sqpeg_adapter) // true
hole.fits(large_sqpeg_adapter) // false
```

Amaliy qo’llanilishi

- **Adapter** klassdan siz foydalanmoqchi bo’lgan klassning interfeysi kodning boshqa qismlari bilan mos kelmagan payt foydalaning.

  Adapter paterni sizga kodingiz va tashqi kutubxona klasslari yoki interfeysi mos kelmaydigan boshqa klasslar orasida tarjimon sifatida xizmat qiladigan o’rtada turuvchi klass sifatida xizmat qiladi.
- Bu paterndan ota klassiga qo’shib bo’lmaydigan umumiy vazifani (yoki funksionallikni) bola klasslarga berishda foydalaning.

  Har bir bola klassni kengaytirishingiz va yangi bola klasslarga kerakli funksionalliklarni qo’shishingiz mumkin. Biroq, ushbu yaratilgan barcha yangi klasslarda kodlarni duplikat qilib yozishingizga to’g’ri keladi. Bu esa kodni yomon ko’rinishga olib keladi.

  Eng yaxshi yechim – bu kerakli funksionallikni adapter klassga berish hisoblanadi. Keyin, kerakli imkoniyatlarga ega bo’lgan adapterga obyektlar birlashtiriladi. Shunda bu imkoniyatlarni dinamik olish mumkin bo’ladi. Bu ish uchun, mo’ljallangan klasslar umumiy interfeysga ega bo’lishi va adapterlar shu interfeyslarni ishlatishi kerak. Bu yondashuv Decorator paterniga juda o’xshaydi.

Ishlatilishi

1. Sizda interfeyslari o’zaro mos kelmaydigan ikkita klass bo’lishi kerak:
   a. O’zgartirib bo’lmaydigan servis klassi (bu ko’pincha tashqi kutubxona, support qilinmay qo’yilgan kod yoki boshqa kodlarga bog’liq bo’lgan kod bo’lishi mumkin)
   b. Servis klassdan foydalanuvchi bir yoki bir nechta mijoz klasslari.
2. Mijoz interfeysini e’lon qilib, servis bilan qanday qilib ular bog’lanishlari kerakligi tushuntiriladi.
3. Adapter klassi yaratiladi va u mijoz interfeysini ishlatadigan qilinadi. Boshlanishiga barcha metodlarni bo’sh qoldirib ketiladi.

4. Servis obyektiga murojaat qilishni saqlash uchun adapter klassiga xususiyat (property) qo’shiladi. Umumiy holatda, bu xususiyat konstruktorda initsializatsiya qilinadi. Ammo, ba’zan adapterning biror metodini chaqirganda servis klassni adapterga berish ancha qulay bo’lib hisoblanadi.
5. Bitta bittalab adapter klassida mijoz interfeysidagi barcha metodlar ishga tushiriladi. Adapter faqat interfeys yoki ma’lumot ko’rinishini o’zgartirish bilan shug’ullanib qolgan ishlarni servis obyektiga topshirishi kerak.
6. Mijoz adapterni mijoz interfeysi orqali ishlatishi kerak. Bu sizga mijoz kodiga ta’sir qilmasdan adapterlarni o’zgartirish yoki kengaytirish imkonini beradi.

Afzallik va kamchiliklari

Afzalliklari	Kamchiliklari
Yagona Javobgarlik Tamoyili. Dasturning biznes logikasidan ajratish mumkin.	Yangi interfeys va klasslar to’plami interfeys qo’shilishi sababli kodning umumiy murakkabligi oshadi.
Ochiq/Yopiqlik Tamoyili. Mijozning kodini buzmay turib dasturga yangi turdagi adapterlarni qo’shish mumkin.	

Boshqa paternlar bilan bog’liqiligi

- **Bridge** dasturning qismlarini bir biridan mustaqil holda yaratishga imkon beradi. Boshqa tomondan, **Adapter** umumiy holda dasturdagi o’zaro mos tushmaydigan klasslarni o’zaro ishlashini ta’minlaydi.
- **Adapter** mavjud obyektning interfeysini o’zgartirsa, **Decorator** obyektning interfeysini o’zgartirmasdan obyektning o’zini kuchaytiradi. Shu bilan birga, *Decorator Adapter*ni ishlatganda amalga oshirib bo’lmaydigan rekursiv kompozitsiyani qo’llab quvvatlaydi.

- **Adapter** o’ziga biriktirilgan obyektni turlicha interfeys bilan ta’minlab beradi. **Proxy** ham obyektni xuddi shunday interfeys bilan ta’minlaydi. **Decorator** esa kuchaytirilgan interfeys bilan ta’minlaydi.
- **Facade** mavjud obyektlar uchun yangi interfeys e’lon qilsa, **Adapter** mavjud interfeysni ishlatib bo’ladigan holatga keltirib beradi. *Adapter* odatda bitta obyektga biriksa, *Facade* obyektlarning butun bir tizimi bilan ishlaydi.
- **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 yana boshqa dasturchilarga patern hal qiladigan muammoni yetkazadi.

Namuna kod

```
namespace RefactoringGuru\Adapter\Conceptual;

/**
 * The Target defines the domain-specific interface used by the client code.
 */
class Target
{
    public function request(): string
    {
        return "Target: The default target's behavior.";
    }
}

/**
 * The Adaptee contains some useful behavior, but its interface is incompatible
 * with the existing client code. The Adaptee needs some adaptation before the
```

```
 * client code can use it.
 */
class Adaptee
{
    public function specificRequest(): string
    {
        return ".eetpadA eht fo roivaheb laicepS";
    }
}

/**
 * The Adapter makes the Adaptee's interface compatible with the Target's
 * interface.
 */
class Adapter extends Target
{
    private $adaptee;

    public function __construct(Adaptee $adaptee)
    {
        $this->adaptee = $adaptee;
    }

    public function request(): string
    {
        return "Adapter: (TRANSLATED) " . strrev($this->adaptee->specificRequest());
    }
}

/**
```

```
 * The client code supports all classes that follow the Target interface.
 */
function clientCode(Target $target)
{
    echo $target->request();
}

echo "Client: I can work just fine with the Target objects:\n";
$target = new Target();
clientCode($target);
echo "\n\n";

$adaptee = new Adaptee();
echo "Client: The Adaptee class has a weird interface. See, I don't understand it:\n";
echo "Adaptee: " . $adaptee->specificRequest();
echo "\n\n";

echo "Client: But I can work with it via the Adapter:\n";
$adapter = new Adapter($adaptee);
clientCode($adapter);
```

```
Client: I can work just fine with the Target objects:
Target: The default target's behavior.

Client: The Adaptee class has a weird interface. See, I don't understand it:
Adaptee: .eetpadA eht fo roivaheb laicepS

Client: But I can work with it via the Adapter:
Adapter: (TRANSLATED) Special behavior of the Adaptee
```