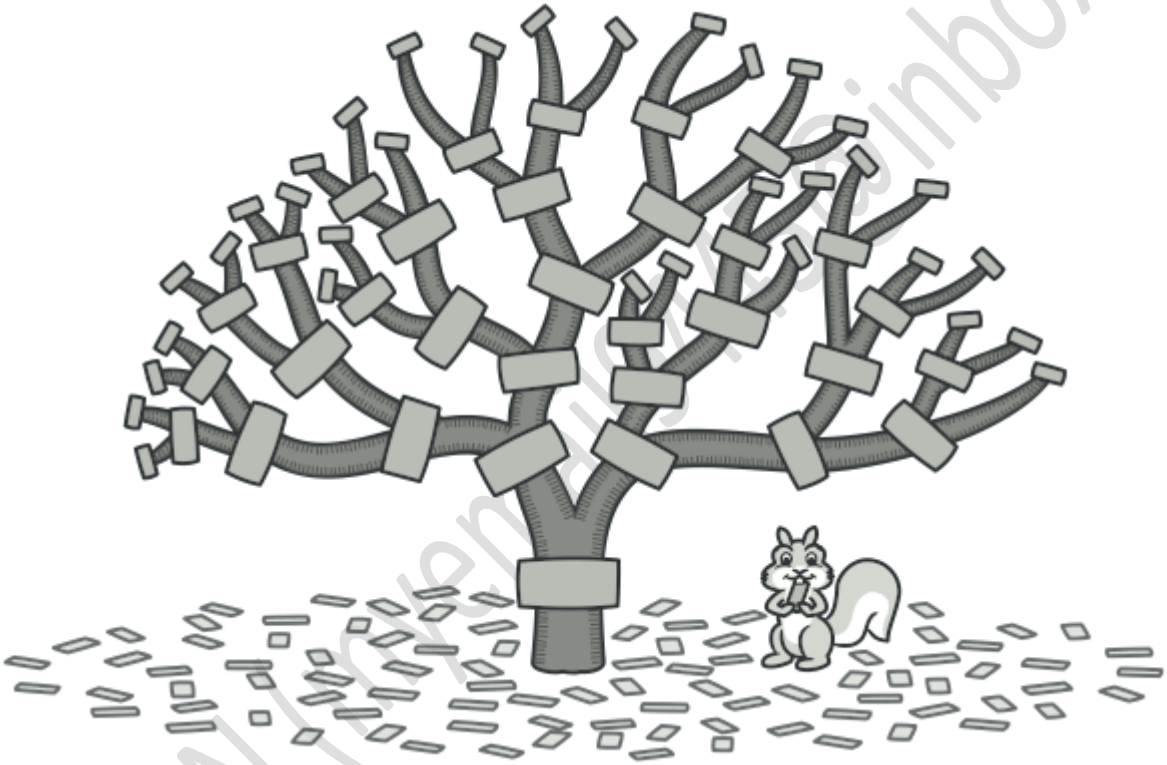
Composite

Object Tree nomi bilan ham ma’lum.

Maqsadi

Composite paterni structural design patterns guruhiga kiradi. Bu patern obyektlarni daraxt ko’rinishida tuzishga imkon berdi. Bu obyektlar daraxtidan xuddi alohida obyektlar ko’rinishida ishlatish mumkin bo’ladi.

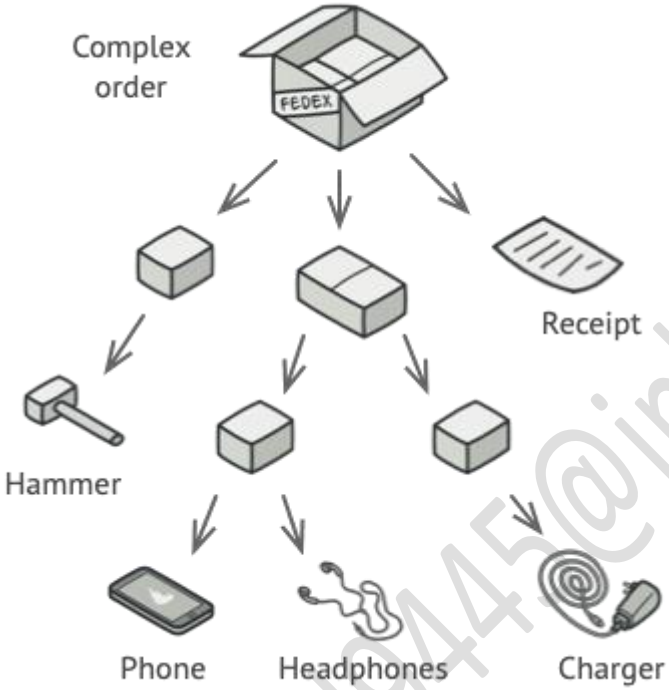


Muammo

Composite paternidan faqatgina dasturingizning asos modeli daraxt ko’rinishida bo’lsa foydalanish mumkin.

Masalan, tasavvur qiling sizda ikki turdagi obyekt bor bo’lsin: *Products* va *Boxes*. *Boxes* o’z ichiga bir qancha *Products* lar va shuningdek bir nechta kichikroq *Boxes* larni oladi. Kichik *Boxes* larning ichida yana *Products* lar va yanayam kichikroq *Boxes* lar bo’lishi mumkin va h.k.

Endi siz shu klasslardan foydalanib buyurtma tizimini yaratmoqchisiz deylik. Buyurtmalar o’z ichiga mahsulotlar va mahsulot solingan qutilarni oladi. Shunda bu mahsulotlarning umumiy narxini qanday hisoblaysiz?



Buyurtma qutiga solingan yoki solinamagan mahsulotlardan tashkil topgan bo’lishi mumkin. Buyurtmaning umumiy ko’rinishi yuqoridan pastga qaragan daraxtga o’xshaydi.

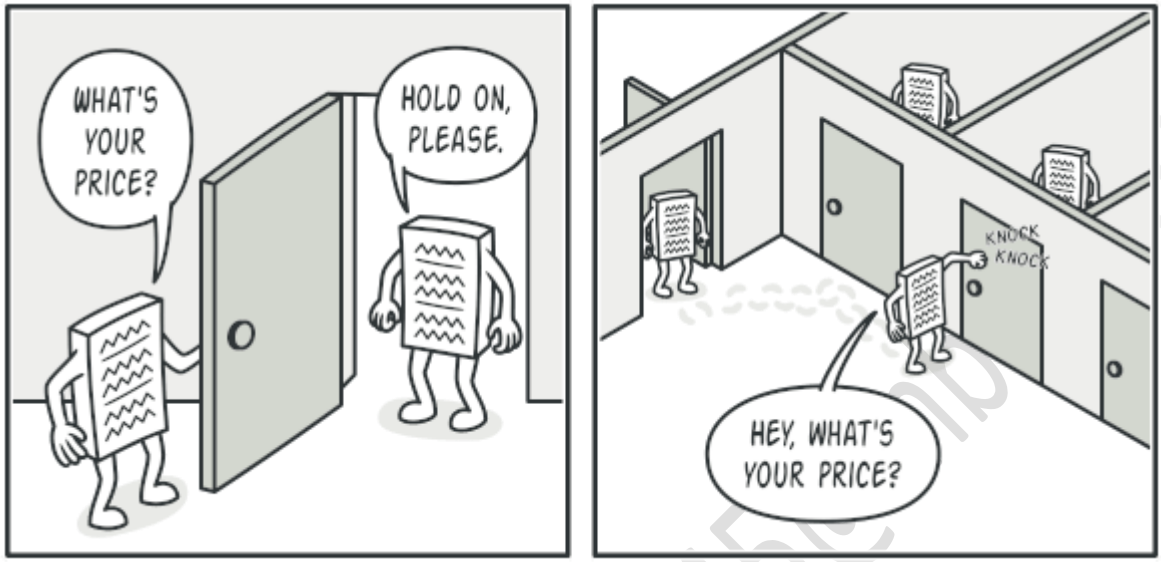
Odatda oddiy dasturchilar to’g’ridan to’g’ri yondashuvni ishlatib ko’radi: barcha qutilarni ochib barcha mahsulotlarni chiqarib oladi va umumiy narxni hisoblaydi. Bu ishni haqiqiy hayotda bajarsa bo’ladi, lekin uni dasturda takrorlanish(loop) operatorini ishlatishdagidek oson emas. Buning uchun ichma-ich joylashgan Products va Boxes klasslarini oldindan bilish kerak bo’ladi. Umumiy narxni hisoblashda to’g’ridan to’g’ri yondashuvni ishlatish yoki juda qiyin yoki umuman imkonsiz bo’ladi.

Yechim

Composite paterni Products va Boxes umumiy narxni hisoblaydigan metodni e’lon qiluvchi umumiy interfeys orqali ishlashini taklif qiladi.

Bu metod qanday ishlaydi? Bu metod mahsulot(Product) uchun uning narxini qaytarsa, quti(Box) uchun esa u shu quti ichidagi mahsulotlarning umumiy narxini

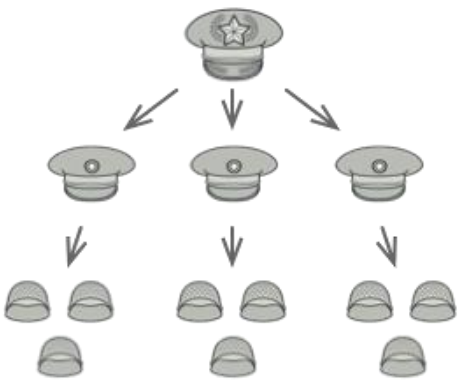
qaytaradi. Agar quti ichida yana kichikroq quti bo’ladigan bo’lsa, bu quti uchun ham umumiy narx olingunicha yuqoridagi amal bajariladi. Yakunida qutining o’zi uchun ham qo’shimcha narx qo’shsa bo’ladi.



Composite paterni obyekt daraxtining barcha komponentlari bo’ylab rekursiv holda kerakli amalni bajarib chiqishga imkon beradi.

Bu yondashuvning eng katta foydasi shundaki sizga daraxtni tashkil qiluvchi obyektlarning interfeys ishlatuvchi klasslari haqida o’ylash kerak bo’lmaydi. Siz obyektingiz oddiy mahsulotmi yoki ichi mahsulot va boshqa qutilarga to’la katta qutimi, bilishingiz shart bo’lmaydi. Ularga umumiy interfeys orqali bir xilda murojaat qilishingiz mumkin. Metodni chaqirgan paytingizda obyektlarning o’zlari so’rovni daraxt bo’ylab pastga uzatishadi.

Hayotiy misol

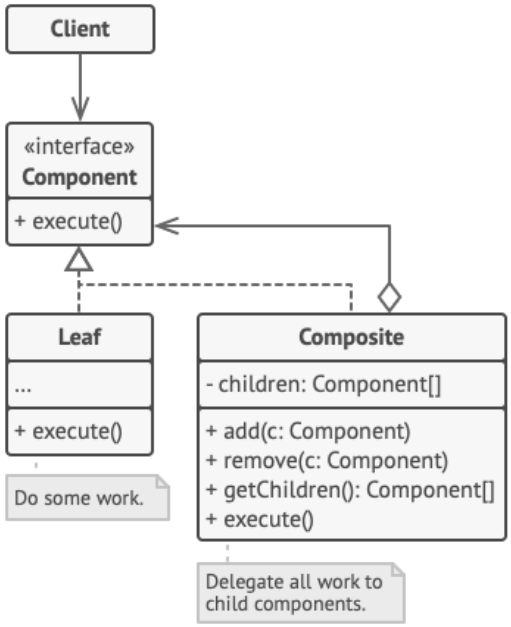


Harbiy tuzilmalardagi misol

Ko’plab davlatlardagi armiyalar iyerarxiya ko’rinishida tuzilgan bo’ladi. Armiya bir qancha bo’linmalardan iborat bo’ladi. Bo’linmalar esa qismlardan iborat bo’ladi. Qismlar esa otryadlarga bo’linuvchi vzvodlardan tashkil topadi. Va nihoyat, otryadlar askarlarning kichik guruhidan iborat bo’ladi. Buyruq iyerarxiyaning yuqorisidan pastga qarab beriladi. Bunda iyerarxiyaning har bir qavatdagi askar o’zi nima qilishi kerakligini yaxshi biladi.

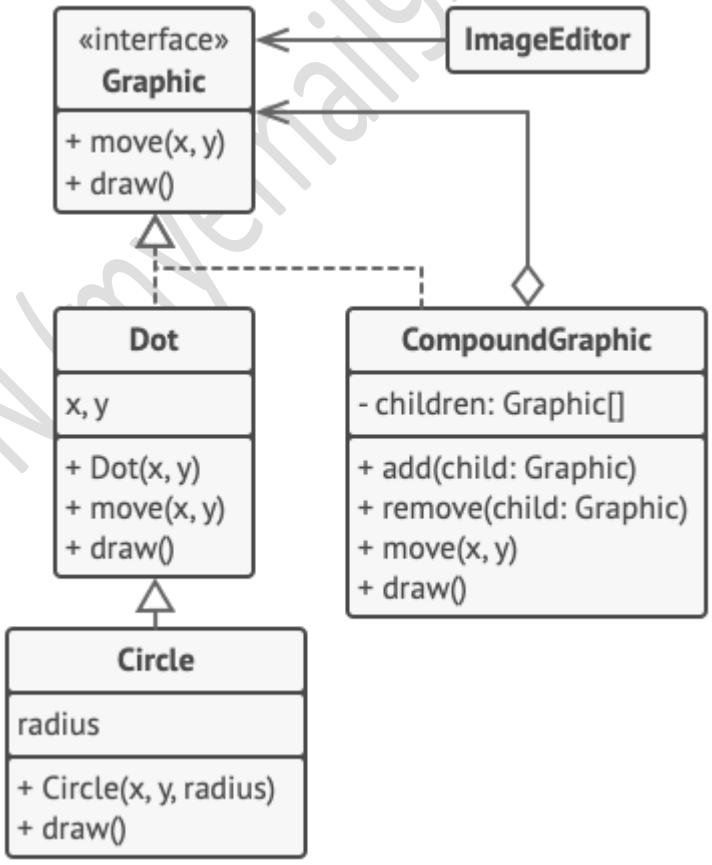
Tuzilishi

1. Component interfeysi daraxtdagi ham oddiy, ham murakkab elementlar uchun umumiy bo’lgan operatsiyalarni tushuntiradi.
2. Leaf daraxtning ost elementlarga ega bo’lmagan oddiy elementi.
   Odatda, haqiqiy ishlarni leaf komponentlari bajaradi. Chunki, ulardan pastda berilgan ishni bajaradigan element mavjud bo’lmaydi.
3. Container (composite sifatida ham ma’lum) – bu ost elementlarga – leaf lar va boshqa containerlarga ega bo’lgan element. Container o’z bolalarining interfeys ishlatadigan klasslarini bilmaydi. U barcha ost elementlari bilan faqatgina component interfeysi orqali ishlaydi.
   So’rovni qabul qilganidan keyin, container ishni o’zining oraliq natijalarni qayta ishlaydigan va yakunda oxirgi natijani mijozga qaytaradigan ost elementlariga topshiradi.
4. Mijoz (Client) barcha elementlar bilan component interfeysi orqali ishlaydi. Natijada, mijoz daraxtdagi oddiy elementlar bilan ham, murakkab elementlar bilan ham bir xilda ishlay oladi.



Psevdokod

Quyidagi misolda, Composite patern yordamida grafik muharrirda geometrik shakllarni yig’ish keltirilgan.



Geometrik shakllar muharririga misol

CompoundGraphic klassi aralash shakllarni o’z ichiga oluvchi har qanday sondagi ost shakllardan tashkil topgan konteyner hisoblanadi. Aralash shaklda oddiy shakldagi bilan bir xildagi metodlar bo’ladi. Biroq, o’zi biror ish qilish o’rniga, aralash shakl o’ziga kelgan so’rovlarni rekursiv tarzda o’zining bolalariga uzatadi va oxirida natijani qabul qilib oladi.

Mijoz kodi barcha shakllar bilan ular uchun umumiy bo’lgan bitta interfeys orqali ishlaydi. Shu sababli ham, mijoz oddiy shakl bilan ishlayaptimi yoki murakkabi bilan, buni bilmaydi. Mijoz juda murakkab obyekt tuzilmalari bilan shu tuzilmani tashkil qiluvchi interfey ishlatadigan klasslar bilan bog’lanmasdan turib ishlay oladi.

```
// The component interface declares common operations for both
// simple and complex objects of a composition.
interface Graphic is
    method move(x, y)
    method draw()

// The leaf class represents end objects of a composition. A
// leaf object can't have any sub-objects. Usually, it's leaf
// objects that do the actual work, while composite objects only
// delegate to their sub-components.
class Dot implements Graphic is
    field x, y

    constructor Dot(x, y) { ... }

    method move(x, y) is
        this.x += x, this.y += y

    method draw() is
        // Draw a dot at X and Y.
```

```
// All component classes can extend other components.
class Circle extends Dot is
    field radius

    constructor Circle(x, y, radius) { ... }

    method draw() is
        // Draw a circle at X and Y with radius R.

// The composite class represents complex components that may
// have children. Composite objects usually delegate the actual
// work to their children and then "sum up" the result.
class CompoundGraphic implements Graphic is
    field children: array of Graphic

    // A composite object can add or remove other components
    // (both simple or complex) to or from its child list.
    method add(child: Graphic) is
        // Add a child to the array of children.

    method remove(child: Graphic) is
        // Remove a child from the array of children.

    method move(x, y) is
        foreach (child in children) do
            child.move(x, y)

    // A composite executes its primary logic in a particular
    // way. It traverses recursively through all its children,
    // collecting and summing up their results. Since the
```

```
    // composite's children pass these calls to their own
    // children and so forth, the whole object tree is traversed
    // as a result.
    method draw() is
        // 1. For each child component:
        //     - Draw the component.
        //     - Update the bounding rectangle.
        // 2. Draw a dashed rectangle using the bounding
        // coordinates.



// The client code works with all the components via their base
// interface. This way the client code can support simple leaf
// components as well as complex composites.
class ImageEditor is
    field all: array of Graphic

    method load() is
        all = new CompoundGraphic()
        all.add(new Dot(1, 2))
        all.add(new Circle(5, 3, 10))
        // ...

    // Combine selected components into one complex composite
    // component.
    method groupSelected(components: array of Graphic) is
        group = new CompoundGraphic()
        foreach (component in components) do
            group.add(component)
            all.remove(component)
        all.add(group)
```

```
// All components will be drawn.
all.draw()
```

Amaliy ishlatilishi

- Composite paterndan daraxt ko’rinishidagi obyekt tuzilmalari bilan ishlashda foydalaning.

  Composite paterni umumiy interfeysga ega bo’lgan ikkita sodda element turlari bilan ta’minlab beradi: sodda leaf (yoki shox) lar va murakkab konteynerlar. Konteyner leaf lardan va boshqa konteynerlardan tuzilgan bo’lishi mumkin. Bu sizga daraxtga o’xshash ichma-ich rekursiv obyekt tuzilmalarini qurish imkonini beradi.
- Mijoz kodi oddiy elementlarga ham, murakkab elementlarga ham bir xilda murojaat qilishi kerak bo’lsa Composite paterndan foydalaning

  Composite patern orqali e’lon qilingan barcha elementlar umumiy interfeysga ega bo’ladi. Bu interfeysdan foydalanish orqali mijoz o’zi ishlayotgan interfeysni ishlatuvchi klasslarni bilishi shart bo’lmaydi.

Ishlatilishi

1. Dasturingizning asos modeli daraxt ko’rinishida ekanligiga amin bo’ling. Bu tuzilishni oddiy elementlar va konteynerlarga bo’lib chiqishga harakat qiling. Yana, konteyner ham oddiy elementlar va boshqa konteynerlarni o’z ichiga olishi kerakligini esdan chiqarmaslik kerak.
2. Sodda va murakkab konteynerlar uchun metodlar ro’yxatiga ega bo’lgan komponent interfeysini e’lon qiling.
3. Sodda elementlarni beruvchi leaf klassini yarating. Dasturda ko’plab turli xildagi leaf klasslar bo’lishi kerak.
4. Murakkab elementlarni beruvchi container klassni yarating. Bu klassda ost elementlarni bog’lash uchun array tipidagi xususiyat e’lon qiling. E’lon

qilingan xususiyat leaf larni ham, containerlarni ham saqlab bilishi kerak bo'ladi. Shu sababli uni componentning interfeys tipi bilan e'lon qilinishi kerak. Component interfeysi metodlarini klassda ishlatish payti containerning ishning ko'p qismini ost elementlarga topshirishi mumkinligini esdan chiqarmaslik kerak.

5. Va nihoyat, containerda bola elementlarni qo'shish va olib tashlash uchun metodlarni e'lon qiling.

   Ushbu operatsiyalar component interfeysida e'lon qilishinishini esingizda saqlang. Bu Interfeysni Ajratish Tamoyiliga zid keladi. Chunki metodlar leaf klasslarda bo'sh qolib ketadi. Biroq, mijoz hattoki obyektlar daraxtini qurishda ham barcha elementlarga bir xilda murojaat qiladi.

Afzallik va kamchiliklari

Afzalliklari	Kamchiliklari
Murakkab daraxt tuzilmalari bilan ancha qulay holatda ishlashingiz mumkin. Bunda polimorfizm va rekursiyadan foydalansangiz ishingiz osonlashadi. Ochiq/Yopiqlilik Tamoyili. Yangi element turlarini obyet daraxti bilan ishlovchi mavjud kodni buzmasdan turib dasturingizga qo'shishingiz mumkin.	Funksionalligi juda ham farq qiladigan klasslar uchun umumiy interfeys yaratish qiyin bo'lishi mumkin. Aniq holatlarda ortiqcha component interfeyslarini yaratishga to'g'ri keladi. Bu esa kodni tushunishni qiyinlashtirib yuboradi.

Boshqa paternlar bilan bog'liqligi

- Murakkab **Composite** daraxtlarni qurishda **Builder** paterndan foydalanish mumkin. Chunki daraxtning qurilish qadamlarini rekursiv holda ishlashi uchun dasturlash mumkin bo'ladi.
- **Chain of Responsibility** (javobgarlik zanjiri) paterni ko'pincha **Composite** bilan birga qo'shib ishlatiladi. Bu holatda, leaf komponenti so'rovni qabul qilib olgan paytda, so'rovni barcha ota komponentlarning zanjiri bo'ylab obyekt daraxti ildizidan pastga qarab uzatadi.

- **Iterator** paternidan **Composite** daraxtining elementlari bo'ylab yurib chiqishda ishlatish mumkin.
- Tezkor xotira (RAM)ni biroz bo'lsada tejash maqsadida **Composite** daraxtining umumiy leaf tugunlarini **Flyweight** sifatida ishlatish mumkin.
- **Composite** va **Decorator** paternlar cheklanmagan sondagi obyektlarni tashkil etishda rekursiv kompozitsiyaga asoslanganligi sababli ikkalasi ham umumiy tuzilish diagrammasiga ega bo'ladi.

  Decorator *Composite*ga o'xshab ketadi, lekin un faqat bitta bola komponentiga ega bo'ladi. Shu bilan birga yana bitta muhim farqi – *Decorator* o'zi birikkan obyektga qo'shimcha javobgarlikni yuklaydi, *Composite* esa bola klasslari qaytargan natijalarni birlashtirib oladi.
  Bundan tashqari *Composite* daraxtidagi biror obyektning xarakteri (behavior)ni kengaytiradi.
- **Composite** va **Decorator** paternlarni ishlatishni qiyinlashtiradigan dizaynlar ko'pincha **Prototype**dan foydalanib muammosini hal qilishi mumkin. Bu patern murakkab tuzilmalarni noldan qayta qurib chiqmasdan ularning nusxasini olish imkonini beradi.

Namuna kod

```
<?php

namespace RefactoringGuru\Composite\Conceptual;

/**
 * The base Component class declares common operations for both simple and
 * complex objects of a composition.
 */
abstract class Component
{
    /**
```

```
     * @var Component
     */
    protected $parent;

    /**
     * Optionally, the base Component can declare an interface for setting and
     * accessing a parent of the component in a tree structure. It can also
     * provide some default implementation for these methods.
     */
    public function setParent(Component $parent)
    {
        $this->parent = $parent;
    }

    public function getParent(): Component
    {
        return $this->parent;
    }

    /**
     * In some cases, it would be beneficial to define the child-management
     * operations right in the base Component class. This way, you won't need to
     * expose any concrete component classes to the client code, even during the
     * object tree assembly. The downside is that these methods will be empty
     * for the leaf-level components.
     */
    public function add(Component $component): void { }

    public function remove(Component $component): void { }

    /**
```

```
     * You can provide a method that lets the client code figure out whether a
     * component can bear children.
     */
    public function isComposite(): bool
    {
        return false;
    }

    /**
     * The base Component may implement some default behavior or leave it to
     * concrete classes (by declaring the method containing the behavior as
     * "abstract").
     */
    abstract public function operation(): string;
}

/**
 * The Leaf class represents the end objects of a composition. A leaf can't have
 * any children.
 *
 * Usually, it's the Leaf objects that do the actual work, whereas Composite
 * objects only delegate to their sub-components.
 */
class Leaf extends Component
{
    public function operation(): string
    {
        return "Leaf";
    }
}
```

```php
/**
 * The Composite class represents the complex components that may have children.
 * Usually, the Composite objects delegate the actual work to their children and
 * then "sum-up" the result.
 */
class Composite extends Component
{
    /**
     * @var \SplObjectStorage
     */
    protected $children;

    public function __construct()
    {
        $this->children = new \SplObjectStorage();
    }

    /**
     * A composite object can add or remove other components (both simple or
     * complex) to or from its child list.
     */
    public function add(Component $component): void
    {
        $this->children->attach($component);
        $component->setParent($this);
    }

    public function remove(Component $component): void
    {
        $this->children->detach($component);
```

```
        $component->setParent(null);
    }

    public function isComposite(): bool
    {
        return true;
    }

    /**
     * The Composite executes its primary logic in a particular way. It
     * traverses recursively through all its children, collecting and summing
     * their results. Since the composite's children pass these calls to their
     * children and so forth, the whole object tree is traversed as a result.
     */
    public function operation(): string
    {
        $results = [];
        foreach ($this->children as $child) {
            $results[] = $child->operation();
        }

        return "Branch(" . implode("+", $results) . ")";
    }
}

/**
 * The client code works with all of the components via the base interface.
 */
function clientCode(Component $component)
{
    // ...
```

```
    echo "RESULT: " . $component->operation();

    // ...
}

/**
 * This way the client code can support the simple leaf components...
 */
$simple = new Leaf();
echo "Client: I've got a simple component:\n";
clientCode($simple);
echo "\n\n";

/**
 * ...as well as the complex composites.
 */
$tree = new Composite();
$branch1 = new Composite();
$branch1->add(new Leaf());
$branch1->add(new Leaf());
$branch2 = new Composite();
$branch2->add(new Leaf());
$tree->add($branch1);
$tree->add($branch2);
echo "Client: Now I've got a composite tree:\n";
clientCode($tree);
echo "\n\n";

/**
 * Thanks to the fact that the child-management operations are declared in the
```

```
 * base Component class, the client code can work with any component, simple or
 * complex, without depending on their concrete classes.
 */
function clientCode2(Component $component1, Component $component2)
{
    // ...

    if ($component1->isComposite()) {
        $component1->add($component2);
    }
    echo "RESULT: " . $component1->operation();

    // ...
}

echo "Client: I don't need to check the components classes even when managing the tree:\n";
clientCode2($tree, $simple);
```

```
Client: I get a simple component:
RESULT: Leaf

Client: Now I get a composite tree:
RESULT: Branch(Branch(Leaf+Leaf)+Branch(Leaf))

Client: I don't need to check the components classes even when managing the tree::
RESULT: Branch(Branch(Leaf+Leaf)+Branch(Leaf)+Leaf)
```