Design Patterns and when they are chosen (code)

نمط تصميم Singleton

يسمح النمط الفردي فقط للفئة أو الكائن أن يكون له مثيل واحد ويستخدم متغيرًا شاملاً لتخزين هذا المثال. يمكنك استخدام التحميل الكسول للتأكد من وجود مثيل واحد فقط للفصل لأنه سينشئ الفصل عند الحاجة إليه فقط.

يمنع ذلك مثيلات متعددة من أن تكون نشطة في نفس الوقت مما قد يتسبب في أخطاء غريبة. في معظم الأحيان يتم تنفيذ هذا في المنشئ. الهدف من النمط الفردي هو تنظيم الحالة العالمية للتطبيق.

مثال على المفرد الذي ربما تستخدمه طوال الوقت هو المسجل الخاص بك.

إذا كنت تعمل مع بعض الأطر الأمامية مثل Angular أو React ، فأنت تعرف كل شيء عن مدى صعوبة التعامل مع السجلات القادمة من مكونات متعددة. يعد هذا مثالًا رائعًا على العمل الفردي لأنك لا تريد أبدًا أكثر من مثيل واحد لكائن المسجل ، خاصة إذا كنت تستخدم نوعًا من أدوات تتبع الأخطاء.

```
class FoodLogger {
  constructor() {
    this.foodLog = []
  }

  log(order) {
    this.foodLog.push(order.foodItem)
    // do fancy code to send this log somewhere
  }
}

// this is the singleton
class FoodLoggerSingleton {
```

```
  constructor() {
    if (!FoodLoggerSingleton.instance) {
      FoodLoggerSingleton.instance = new FoodLogger()
    }
  }

  getFoodLoggerInstance() {
    return FoodLoggerSingleton.instance
  }
}

module.exports = FoodLoggerSingleton
```

مثال على الفصل الفردي

الآن لا داعي للقلق بشأن فقد السجلات من حالات متعددة لأن لديك واحدًا فقط في مشروعك. لذلك عندما تريد تسجيل الطعام الذي تم طلبه ، يمكنك استخدام نفس *FoodLogger* مثيل عبر ملفات أو مكونات متعددة.

```
const FoodLogger = require('./FoodLogger')

const foodLogger = new FoodLogger().getFoodLoggerInstance()

class Customer {
  constructor(order) {
    this.price = order.price
    this.food = order.foodItem
    foodLogger.log(order)
  }

  // other cool stuff happening for the customer
}

module.exports = Customer
```

مثال على فئة العملاء باستخدام المفرد

```
const FoodLogger = require('./FoodLogger')

const foodLogger = new FoodLogger().getFoodLoggerInstance()
```

```
class Restaurant {
  constructor(inventory) {
    this.quantity = inventory.count
    this.food = inventory.foodItem
    foodLogger.log(inventory)
  }

  // other cool stuff happening at the restaurant
}

module.exports = Restaurant
```

مثال على فئة المطعم باستخدام نفس الفردي مثل فئة العملاء

مع وجود هذا النمط الفردي في مكانه ، لا داعي للقلق بشأن مجرد الحصول على يمكنك الحصول عليها من أي مكان في قاعدة .السجلات من ملف التطبيق الرئيسي الشفرة الخاصة بك وسوف ينتقلون جميعًا إلى نفس مثيل المسجل بالضبط ، مما .يعني أنه لن يتم فقد أي من سجلاتك بسبب الحالات الجديدة

نمط تصميم الإستراتيجية

إنه في الأساس المكان .if else الاستراتيجية هي النمط مثل نسخة متقدمة من بيان يتم استخدام هذه الواجهة .الذي تنشئ فيه واجهة لطريقة لديك في صنفك الأساسي بعد ذلك للعثور على التطبيق الصحيح لتلك الطريقة التي يجب استخدامها في فئة .سيتم تحديد التنفيذ ، في هذه الحالة ، في وقت التشغيل بناءً على العميل .مشتقة

هذا النمط مفيد بشكل لا يصدق في المواقف التي تطلب فيها طرقًا اختيارية لن تحتاج بعض أمثلة هذه الفئة إلى الأساليب الاختيارية ، وهذا يسبب مشكلة .للفصل يمكنك استخدام واجهات للطرق الاختيارية ، ولكن بعد ذلك .في حلول الوراثة سيتعين عليك كتابة التنفيذ في كل مرة تستخدم فيها هذه الفئة نظرًا لعدم وجود تطبيق .افتراضي

بدلاً من أن يبحث العميل عن .هذا هو المكان الذي ينقذنا فيه نمط الإستراتيجية أحد .تطبيق ، فإنه يفوض إلى واجهة إستراتيجية وتجد الإستراتيجية التنفيذ الصحيح .الاستخدامات الشائعة لهذا هو مع أنظمة معالجة الدفع

يكون لديك عربة تسوق تتيح للعملاء فقط الدفع ببطاقات الائتمان الخاصة *أن* يمكن بهم ، لكنك ستفقد العملاء الذين يرغبون في استخدام طرق دفع أخرى.
يتيح لنا نمط تصميم الإستراتيجية فصل طرق الدفع عن عملية السداد مما يعني أنه يمكننا إضافة أو تحديث الاستراتيجيات دون تغيير أي رمز في عربة التسوق أو عملية الدفع.

فيما يلي مثال على تنفيذ نمط الإستراتيجية باستخدام مثال طريقة الدفع.

```
class PaymentMethodStrategy {

  const customerInfoType = {
    country: string
    emailAddress: string
    name: string
    accountNumber?: number
    address?: string
    cardNumber?: number
    city?: string
    routingNumber?: number
    state?: string
  }

  static BankAccount(customerInfo: customerInfoType) {
    const { name, accountNumber, routingNumber } = customerInfo
    // do stuff to get payment
  }

  static BitCoin(customerInfo: customerInfoType) {
    const { emailAddress, accountNumber } = customerInfo
    // do stuff to get payment
  }

  static CreditCard(customerInfo: customerInfoType) {
    const { name, cardNumber, emailAddress } = customerInfo
    // do stuff to get payment
```

```
  }

  static MailIn(customerInfo: customerInfoType) {
    const { name, address, city, state, country } = customerInfo
    // do stuff to get payment
  }

  static PayPal(customerInfo: customerInfoType) {
    const { emailAddress } = customerInfo
    // do stuff to get payment
  }
}
```

مثال على تنفيذ نمط الاستراتيجية

كل .لتنفيذ إستراتيجية طريقة الدفع الخاصة بنا ، أنشأنا فئة واحدة بطرق ثابتة متعددة ، وهذه المعلمة لها نوع محدد *customerInfo* أسلوب يأخذ نفس المعلمة ، من *customerInfoType* . (مرحبًا بكم جميع مطوري TypeScript! لاحظ ؟؟) *العميل* .أن كل طريقة لها تنفيذها الخاص وتستخدم قيمًا مختلفة من معلومات باستخدام نمط الإستراتيجية ، يمكنك أيضًا تغيير الإستراتيجية المستخدمة في وقت هذا يعني أنك ستتمكن من تغيير الإستراتيجية ، أو تنفيذ الطريقة ، .التشغيل ديناميكيًا .التي يتم استخدامها بناءً على إدخال المستخدم أو البيئة التي يعمل بها التطبيق

بسيط مثل هذا *config.json* يمكنك أيضًا تعيين تنفيذ افتراضي في ملف:

```
{
  "paymentMethod": {
    "strategy": "PayPal"
  }
}
```

في وقت التشغيل "PayPal" لتعيين التنفيذ الافتراضي لطريقة الدفع إلى config.json

عندما يبدأ العميل في إجراء عملية الدفع على موقع الويب الخاص بك ، فإن طريقة الذي يأتي PayPal الدفع الافتراضية التي يواجهها ستكون تطبيق .يمكن تحديث هذا بسهولة إذا اختار العميل طريقة دفع مختلفة . *config.json* من .الآن سنقوم بإنشاء ملف لعملية الخروج الخاصة بنا

```
const PaymentMethodStrategy = require('./PaymentMethodStrategy')
const config = require('./config')
```

```
class Checkout {
  constructor(strategy='CreditCard') {
    this.strategy = PaymentMethodStrategy[strategy]
  }

  // do some fancy code here and get user input and payment method

  changeStrategy(newStrategy) {
    this.strategy = PaymentMethodStrategy[newStrategy]
  }

  const userInput = {
    name: 'Malcolm',
    cardNumber: 3910000034581941,
    emailAddress: 'mac@gmailer.com',
    country: 'US'
  }

  const selectedStrategy = 'Bitcoin'

  changeStrategy(selectedStrategy)

  postPayment(userInput) {
    this.strategy(userInput)
  }
}

module.exports = new Checkout(config.paymentMethod.strategy)
```

نقوم باستيراد *فئة* Checkout هذه هي المكان الذي يظهر فيه نمط الإستراتيجية. ملفين حتى تتوفر لدينا استراتيجيات طريقة الدفع والاستراتيجية الافتراضية من *التكوين* ملف.

الافتراضية في *للاستراتيجية* ثم نقوم بإنشاء الفئة باستخدام المُنشئ وقيمة احتياطية لمتغير *الإستراتيجية* بعد ذلك نقوم بتعيين قيمة config حالة عدم وجود واحدة في دولة محلي.

القدرة على *Checkout* من الطرق المهمة التي نحتاج إلى تنفيذها في صفنا في قد يغير العميل طريقة الدفع التي يريد استخدامها وستحتاج .تغيير استراتيجية الدفع .طريقة الإستراتيجية*تغيير* هذا هو سبب .إلى أن تكون قادرًا على التعامل مع ذلك
بعد الانتهاء من بعض الترميز الرائع والحصول على جميع المدخلات من أحد العملاء ، يمكنك عندئذٍ تحديث استراتيجية الدفع فورًا بناءً على مدخلاتهم .ديناميكيًا قبل إرسال الدفعة للمعالجة*الإستراتيجية* وتعيين
في مرحلة ما ، قد تحتاج إلى إضافة المزيد من طرق الدفع إلى عربة التسوق الخاصة بك وكل ما عليك فعله هو إضافتها إلى
سيكون متاحًا على الفور في أي مكان يتم فيه . *PaymentMethodStrategy* فئة .استخدام هذا الفصل
يعد نمط تصميم الإستراتيجية نموذجًا قويًا عندما تتعامل مع طرق لها تطبيقات قد تشعر أنك تستخدم واجهة ، لكن ليس عليك كتابة تنفيذ للطريقة في كل .متعددة .يمنحك مرونة أكثر من الواجهات .مرة تسميها في فصل دراسي مختلف

نمط تصميم المراقب

جزء .، فقد استخدمت بالفعل نمط تصميم المراقب MVC إذا سبق لك استخدام نمط موضوعك .النموذج يشبه الموضوع وجزء العرض يشبه مراقب هذا الموضوع ثم لديك مراقبون ، مثل المكونات المختلفة .يحمل جميع البيانات وحالة تلك البيانات ،. سيحصلون على تلك البيانات من الموضوع عند تحديث البيانات

الهدف من نمط تصميم المراقب هو إنشاء علاقة رأس بأطراف بين الموضوع لذلك في أي وقت .وجميع المراقبين الذين ينتظرون البيانات حتى يمكن تحديثها .تتغير حالة الموضوع ، سيتم إخطار جميع المراقبين وتحديثهم على الفور

تتضمن بعض الأمثلة على وقت استخدام هذا النمط: إرسال إشعارات المستخدم .والتحديث والفلاتر والتعامل مع المشتركين

لنفترض أن لديك تطبيق صفحة واحدة يحتوي على ثلاث قوائم منسدلة للميزات هذا شائع في العديد من .تعتمد على اختيار فئة من قائمة منسدلة ذات مستوى أعلى لديك مجموعة من المرشحات على الصفحة .Home Depot مواقع التسوق ، مثل .تعتمد على قيمة مرشح المستوى الأعلى

قد يبدو رمز القائمة المنسدلة ذات المستوى الأعلى كما يلي:

```
class CategoryDropdown {
  constructor() {
    this.categories = ['appliances', 'doors', 'tools']
    this.subscriber = []
  }

  // pretend there's some fancy code here

  subscribe(observer) {
    this.subscriber.push(observer)
  }

  onChange(selectedCategory) {
    this.subscriber.forEach(observer => observer.update(selectedCategory))
  }
}
```

الموضوع الذي يحدّث المراقبين

ملف *CategoryDropdown* هذا عبارة عن فئة بسيطة مع مُنشئ يقوم بتهيئة خيارات الفئة المتوفرة لدينا في القائمة المنسدلة. هذا هو الملف الذي ستتعامل معه لاسترداد قائمة من النهاية الخلفية أو أي نوع من الفرز تريد القيام به قبل أن يرى المستخدم الخيارات.

هي الطريقة التي سيتلقى بها كل مرشح تم إنشاؤه باستخدام هذه طريقة *الاشتراك* الفئة تحديثات حول حالة المراقب.

هي الطريقة التي نرسل بها إشعارًا إلى جميع المشتركين يفيد طريقة *onChange* بحدوث تغيير في الحالة لدى المراقب الذي يشاهدونه. نحن فقط نمرّر عبر جميع المشتركين وندعو طريقة *المحددة التحديث الخاصة بهم بالفئة* .

قد يبدو رمز المرشحات الأخرى كما يلي:

```
class FilterDropdown {
  constructor(filterType) {
    this.filterType = filterType
    this.items = []
  }
```

```
    // more fancy code here; maybe make that API call to get items list based on filterType

    update(category) {
      fetch('https://example.com')
        .then(res => this.items(res))
    }
}
```

مراقب محتمل للموضوع

يعد ملف *FilterDropdown* هذا فئة بسيطة أخرى تمثل جميع القوائم المنسدلة المحتملة التي قد نستخدمها في الصفحة. ، عندما يتم إنشاء مثيل جديد من هذه الفئة يمكن استخدام هذا لإجراء مكالمات API يجب أن يتم تمريره إلى *filterType*. محددة للحصول على قائمة العناصر.

هي تنفيذ لما يمكنك فعله بالفئة الجديدة بمجرد إرسالها من المراقب *//التحديث* طريقة. الآن سنلقي نظرة على معنى استخدام هذه الملفات بنمط المراقب:

```
const CategoryDropdown = require('./CategoryDropdown')
const FilterDropdown = require('./FilterDropdown')

const categoryDropdown = new CategoryDropdown()

const colorsDropdown = new FilterDropdown('colors')
const priceDropdown = new FilterDropdown('price')
const brandDropdown = new FilterDropdown('brand')

categoryDropdown.subscribe(colorsDropdown)
categoryDropdown.subscribe(priceDropdown)
categoryDropdown.subscribe(brandDropdown)
```

مثال على نمط المراقب في العمل

ما يوضحه هذا الملف هو أن لدينا 3 قوائم منسدلة للمشتركين في القائمة المنسدلة للفئات يمكن ملاحظتها. ثم نقوم بالاشتراك في كل من هذه القوائم المنسدلة. عندما يتم تحديث فئة المراقب ، سيرسل القيمة إلى كل مشترك والذي. للمراقب سيقوم بتحديث القوائم المنسدلة الفردية على الفور

.

نمط تصميم الديكور

يمكن أن يكون لديك فئة أساسية. استخدام نمط تصميم الديكور بسيط إلى حد ما لنفترض الآن أن لديك بأساليب وخصائص موجودة عند إنشاء كائن جديد بالفئة بعض حالات الفئة التي تحتاج إلى طرق أو خصائص لم تأتي من الفئة الأساسية.

يمكنك إضافة تلك الأساليب والخصائص الإضافية إلى الفئة الأساسية ، ولكن هذا قد يفسد مثيلاتك الأخرى. يمكنك أيضًا إنشاء فئات فرعية تحتوي على أساليب وخصائص معينة تحتاجها ولا يمكنك وضعها في صنفك الأساسي.

هذا هو. أي من هذين النهجين سوف يحل مشكلتك ، لكنهما صعبان وغير فعالين المكان الذي يتدخل فيه نمط الزخرفة. بدلاً من جعل قاعدة التعليمات البرمجية الخاصة بك قبيحة فقط لإضافة بعض الأشياء إلى مثيل الكائن ، يمكنك التعامل مع تلك الأشياء المحددة مباشرة إلى المثيل.

لذلك إذا كنت بحاجة إلى إضافة خاصية جديدة تحمل سعر كائن ما ، فيمكنك استخدام نمط الزخرفة لإضافته مباشرةً إلى مثيل الكائن المحدد ولن يؤثر على أي حالات أخرى لكائن الفئة هذا.

إذن ربما تكون قد واجهت نمط هل سبق لك أن طلبت الطعام عبر الإنترنت؟ إذا كنت تحصل على شطيرة وترغب في إضافة طبقة خاصة ، فإن موقع الديكور. الويب لا يضيف تلك الإضافات إلى كل مثيل يحاول المستخدمون الحاليون طلبها.

فيما يلي مثال لفئة العملاء:

```
class Customer {
  constructor(balance=20) {
    this.balance = balance
    this.foodItems = []
  }

  buy(food) {
    if (food.price) < this.balance {
```

```
        console.log('you should get it')
        this.balance -= food.price
        this.foodItems.push(food)
      }
      else {
        console.log('maybe you should get something else')
      }
    }
}

module.exports = Customer
```

مثال على فئة العملاء

وإليك مثال لفصل شطيرة:

```
class Sandwich {
  constructor(type, price) {
    this.type = type
    this.price = price
  }

  order() {
    console.log(`You ordered a ${this.type} sandwich for $ ${this.price}.`)
  }
}

class DeluxeSandwich {
  constructor(baseSandwich) {
    this.type = `Deluxe ${baseSandwich.type}`
    this.price = baseSandwich.price + 1.75
  }
}

class ExquisiteSandwich {
  constructor(baseSandwich) {
    this.type = `Exquisite ${baseSandwich.type}`
    this.price = baseSandwich.price + 10.75
```

```
  }

  order() {
    console.log(`You ordered an ${this.type} sandwich. It's got everything you need to be happy for days.`)
  }
}

module.exports = { Sandwich, DeluxeSandwich, ExquisiteSandwich }
```

مثال على فئة شطيرة

لدينا فئة .فئة الشطيرة هذه هي المكان الذي يتم فيه استخدام نمط الديكور
قد يرغب .تحدد القواعد لما يحدث عند طلب شطيرة عادية *الساندويتش* أساسية
.العملاء في ترقية السندويشات وهذا يعني فقط تغيير المكون والسعر
لقد أردت فقط إضافة الوظيفة لزيادة السعر وتحديث نوع الساندويتش الخاص
على الرغم من أنك قد تحتاج إلى . دون تغيير طريقة طلبها *DeluxeSandwich* بـ
لأن هناك تغييرًا جذريًا في جودة *ExquisiteSandwich* طريقة ترتيب مختلفة لـ
.المكونات
يتيح لك نمط الديكور تغيير الفئة الأساسية ديناميكيًا دون التأثير عليها أو أي فئات
لا داعي للقلق بشأن تنفيذ الوظائف التي لا تعرفها ، مثل الواجهات ، ولا .أخرى
.يتعين عليك تضمين خصائص لن تستخدمها في كل فصل دراسي

الآن إذا ذهبنا إلى مثال حيث يتم إنشاء مثيل لهذه الفئة كما لو كان العميل يقدم طلب
.شطيرة

```
const { Sandwich, DeluxeSandwich, ExquisiteSandwich } = require('./Sandwich')
const Customer = require('./Customer')

const cust1 = new Customer(57)

const turkeySandwich = new Sandwich('Turkey', 6.49)
const bltSandwich = new Sandwich('BLT', 7.55)

const deluxeBltSandwich = new DeluxeSandwich(bltSandwich)
const exquisiteTurkeySandwich = new ExquisiteSandwich(turkeySandwich)
```

```
cust1.buy(turkeySandwich)
cust1.buy(bltSandwich)
```