بخش اول

بازآرایی: یه مثال ساده

چگونه باید شروع به صحبت در مورد بازآرایی کد (Refactoring) کنم؟ روش سنتی این است که با معرفی تاریخچه موضوع، اصول کلی و موارد مشابه آغاز کنیم. وقتی کسی این کار را در یک کنفرانس انجام می‌دهد، من کمی خوابم می‌گیرد. ذهنم شروع به پرسه زدن می‌کند و با یک فرآیند پس‌زمینه با اولویت پایین منتظر می‌مانم تا گوینده مثالی بزند.

مثال‌ها مرا بیدار می‌کنند زیرا می‌توانم ببینم چه اتفاقی در حال رخ دادن است. با اصول کلی، بسیار راحت می‌توان تعمیم‌های گسترده‌ای داد و فهمیدن چگونگی اعمال آن‌ها سخت می‌شود. اما یک مثال کمک می‌کند تا همه چیز روشن شود.

پس من می‌خواهم این کتاب را با مثالی از بازآرایی کد شروع کنم. در این مثال توضیح می‌دهم که بازآرایی چگونه کار می‌کند و به شما حس فرآیند بازآرایی را می‌دهم. سپس می‌توانم در فصل بعدی، به شیوه اصولی به معرفی موضوع بپردازم.

با هر مثال مقدماتی، با این حال، یک مشکل پیش می‌آید. اگر یک برنامه بزرگ انتخاب کنم، توصیف آن و چگونگی بازآرایی آن بیش از حد پیچیده خواهد بود که یک خواننده معمولی بتواند با آن کنار بیاید. (من این کار را با کتاب اولم امتحان کردم — و دو مثال را کنار گذاشتم، هرچند که هنوز نسبتاً کوچک بودند اما توصیف آن‌ها بیش از صد صفحه شد.)

اما اگر یک برنامه‌ای انتخاب کنم که به اندازه کافی کوچک باشد تا قابل فهم باشد، بازآرایی کد به نظر نمی‌رسد که ارزش انجام دادن داشته باشد.

بنابراین من در تنگنای کلاسیک کسی قرار دارم که می‌خواهد تکنیک‌هایی را توضیح دهد که برای برنامه‌های دنیای واقعی مفید هستند. صادقانه بگویم، ارزشش را ندارد که همه بازآرایی‌هایی را که می‌خواهم به شما نشان دهم روی این برنامه کوچک انجام دهم. اما اگر کدی که به شما نشان می‌دهم بخشی از یک سیستم بزرگ‌تر باشد، آن‌گاه بازآرایی اهمیت پیدا می‌کند. فقط به مثال من نگاه کنید و آن را در بستر یک سیستم بزرگ‌تر تصور کنید.

نقطه شروع

در نسخه اول این کتاب، برنامه اولیه من یک صورت‌حساب از یک فروشگاه اجاره ویدیو چاپ می‌کرد که اکنون ممکن است بسیاری از شما بپرسید" :فروشگاه اجاره ویدیو چیست؟ "
به جای پاسخ دادن به این سوال، من مثال را به چیزی که هم قدیمی‌تر و هم همچنان رایج است تغییر دادم.

تصور کنید یک شرکت تئاتری وجود دارد که به مکان‌های مختلف برای اجرای نمایش‌ها می‌رود .معمولاً یک مشتری چند نمایش درخواست می‌کند و شرکت بر اساس تعداد مخاطبان و نوع نمایشی که اجرا می‌کنند،
هزینه‌ای از آن‌ها دریافت می‌کند. در حال حاضر، دو نوع نمایش وجود دارد که شرکت اجرا می‌کند: تراژدی‌ها و کمدی‌ها .
علاوه بر ارائه یک صورت‌حساب برای اجرا، شرکت به مشتریان خود "اعتبار حجمی "نیز ارائه می‌دهد که آن‌ها می‌توانند برای تخفیف در اجراهای آینده از آن استفاده کنند —
چیزی مانند یک مکانیزم وفاداری مشتری.

اجراکنندگان داده‌های مربوط به نمایش‌های خود را در یک فایل ساده JSON ذخیره می‌کنند که چیزی شبیه به این است:

plays.json

```
{
  "hamlet": {"name": "Hamlet", "type": "tragedy"},
  "as-like": {"name": "As You Like It", "type": "comedy"},
  "othello": {"name": "Othello", "type": "tragedy"}
}
```

اطلاعات صورت‌حساب‌های آن‌ها نیز در یک فایل JSON ذخیره می‌شود:

invoices.json

```
[
  {
    "customer": "BigCo",
    "performances": [
      {
        "playID": "hamlet",
        "audience": 55
      },
      {
        "playID": "as-like",
        "audience": 35
      },
      {
        "playID": "othello",
        "audience": 40
      }
    ]
  }
]
```

کدی که صورت‌حساب را چاپ می‌کند، یک تابع ساده است:

```
function statement (invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;
  let result = `Statement for ${invoice.customer}\n`;
  const format = new Intl.NumberFormat("en-US", {
    style: "currency",
    currency: "USD",
    minimumFractionDigits: 2
  }).format;

  for (let perf of invoice.performances) {
    const play = plays[perf.playID];
    let thisAmount = 0;

    switch (play.type) {
      case "tragedy":
        thisAmount = 40000;
        if (perf.audience > 30) {
          thisAmount += 1000 * (perf.audience - 30);
        }
        break;
      case "comedy":
        thisAmount = 30000;
        if (perf.audience > 20) {
          thisAmount += 10000 + 500 * (perf.audience - 20);
        }
        thisAmount += 300 * perf.audience;
        break;
      default:
        throw new Error(`unknown type: ${play.type}`);
    }

    // add volume credits
    volumeCredits += Math.max(perf.audience - 30, 0);

    // add extra credit for every ten comedy attendees
    if ("comedy" === play.type) volumeCredits += Math.floor(perf.audience /
5);

    // print line for this order
    result += ` ${play.name}: ${format(thisAmount / 100)} (${perf.audience}
seats)\n`;
    totalAmount += thisAmount;
  }

  result += `Amount owed is ${format(totalAmount / 100)}\n`;
  result += `You earned ${volumeCredits} credits\n`;
  return result;
}
```

اجرای آن کد با استفاده از فایل‌های داده آزمایشی فوق، نتایج زیر را تولید می‌کند:

```
Statement for BigCo
Hamlet: $650.00 (55 seats)
As You Like It: $580.00 (35 seats)
Othello: $500.00 (40 seats)
Amount owed is $1,730.00
You earned 47 credits
```

نظرات در مورد برنامه ابتدایی

نظر شما در مورد طراحی این برنامه چیست؟ اولین چیزی که می‌گویم این است که به عنوان یک برنامه، قابل قبول است—چون این برنامه خیلی کوتاه است، به هیچ ساختار عمیقی برای قابل درک بودن نیاز ندارد. اما به یاد داشته باشید که من قبلاً به این نکته اشاره کردم که باید مثال‌ها را کوچک نگه‌دارم. تصور کنید این برنامه در مقیاس بزرگ‌تر—شاید صدها خط طولانی—باشد. در این اندازه، یک تابع درون‌خطی به سختی قابل درک است .

با توجه به اینکه برنامه به درستی کار می‌کند، آیا هر گونه نظری درباره ساختار آن فقط یک قضاوت زیباشناختی نیست، و عدم تمایل به کد «زشت»؟ در نهایت، کامپایلر اهمیتی نمی‌دهد که آیا کد زشت است یا تمیز. اما وقتی من سیستم را تغییر می‌دهم، انسانی درگیر است و انسان‌ها اهمیت می‌دهند. یک سیستم به‌طور ضعیف طراحی شده، تغییر دادن آن سخت است—چون فهمیدن اینکه چه چیزی باید تغییر کند و این تغییرات چگونه با کد موجود تعامل خواهند کرد تا رفتار مورد نظر من را به دست آورند، دشوار است. و اگر درک اینکه چه چیزی باید تغییر کند سخت باشد، احتمال زیادی وجود دارد که من اشتباهاتی مرتکب شوم و باگ‌هایی را معرفی کنم .

بنابراین، اگر مجبور به اصلاح برنامه‌ای با صدها خط کد باشم، ترجیح می‌دهم که آن را به مجموعه‌ای از توابع و دیگر عناصر برنامه تقسیم‌بندی کرده باشد که به من اجازه می‌دهد راحت‌تر بفهمم برنامه چه کار می‌کند. اگر برنامه فاقد ساختار باشد، معمولاً برای من راحت‌تر است که ابتدا ساختار را به برنامه اضافه کنم و سپس تغییراتی که نیاز دارم را انجام دهم .

وقتی باید ویژگی جدیدی به یک برنامه اضافه کنید اما کد به صورت راحتی ساختار بندی نشده است، ابتدا برنامه را بازسازی کنید تا اضافه کردن ویژگی آسان‌تر باشد، سپس ویژگی را اضافه کنید .

نظرات در مورد تغییرات در برنامه

در این مورد، من چند تغییر دارم که کاربران می‌خواهند انجام دهند. اول، آنها می‌خواهند یک صورت‌حساب به‌صورت HTMLچاپ شود. این تغییر چه تأثیری خواهد داشت؟ من با اضافه کردن دستور شرطی در اطراف هر بیانیه‌ای که به نتیجه یک رشته اضافه می‌کند، مواجه می‌شوم. این موضوع به پیچیدگی زیادی در تابع می‌افزاید. با چنین وضعیتی، بیشتر افراد ترجیح می‌دهند روش را کپی کرده و آن را تغییر دهند تا HTML تولید کند. به نظر نمی‌رسد که انجام یک کپی کار خیلی سختی باشد، اما این کار مشکلات متعددی را برای آینده ایجاد می‌کند. هرگونه تغییر در منطق محاسبه هزینه، مرا مجبور می‌کند که هر دو روش را به‌روزرسانی کنم—و اطمینان حاصل کنم که آنها به‌طور سازگار به‌روزرسانی شده‌اند. اگر من برنامه‌ای بنویسم که هرگز تغییر نکند، این نوع کپی و چسباندن خوب است. اما اگر این برنامه طولانی‌مدت باشد، duplicationیک تهدید است .

این من را به تغییر دومی می‌رساند. بازیگران به دنبال اجرای انواع بیشتری از نمایش‌ها هستند: آنها امیدوارند تاریخ، روستایی، روستایی–کمدی، تاریخی–روستایی، تاریخی–کمدی–تاریخی–روستایی، صحنه غیرقابل تقسیم و شعر نامحدود را به فهرست خود اضافه کنند. آنها دقیقاً هنوز تصمیم نگرفته‌اند که چه کاری می‌خواهند انجام دهند و چه زمانی. این تغییر بر نحوه محاسبه هزینه‌های نمایش‌هایشان و همچنین نحوه محاسبه اعتبار حجم تأثیر خواهد گذاشت. به عنوان یک توسعه‌دهنده باتجربه می‌توانم مطمئن باشم که هر طرحی که آنها به آن برسند، در عرض شش ماه دوباره تغییر خواهد کرد. بعد از همه، وقتی درخواست‌های ویژگی می‌آیند، آنها به‌عنوان جاسوسان تنها نمی‌آیند، بلکه به‌صورت دسته‌جمعی می‌آیند .

دوباره، آن روش بیانیه جایی است که باید تغییرات برای رسیدگی به تغییرات در طبقه‌بندی و قوانین محاسبه هزینه انجام شود. اما اگر من بیانیه را به htmlStatement کپی کنم، باید اطمینان حاصل کنم که هرگونه تغییر سازگار باشد. بعلاوه، با رشد پیچیدگی قوانین، فهمیدن اینکه کجا باید تغییرات انجام شود و انجام آنها بدون ایجاد اشتباه، دشوارتر خواهد بود .

اجازه دهید تأکید کنم که این تغییرات هستند که نیاز به انجام refactoring را به وجود می‌آورند. اگر کد کار کند و هرگز نیازی به تغییر نداشته باشد، کاملاً خوب است که آن را به حال خود بگذاریم. بهتر است که آن را بهبود دهیم، اما مگر اینکه کسی نیاز به درک آن داشته باشد، هیچ آسیبی واقعی ایجاد نمی‌کند. اما به محض اینکه کسی نیاز داشته باشد بفهمد آن کد چگونه کار می‌کند و با مشکلاتی برای پیگیری آن مواجه شود، پس باید اقدام کنید .

اولین گام در بازسازی (Refactoring)

هر بار که من بازسازی (refactoring) انجام می‌دهم، اولین گام همیشه یکسان است. من باید اطمینان حاصل کنم که یک مجموعه قوی از تست‌ها برای آن بخش از کد دارم. تست‌ها ضروری هستند زیرا حتی با وجود اینکه من بازسازی‌هایی را دنبال می‌کنم که به گونه‌ای ساختاربندی شده‌اند تا از بیشتر فرصت‌ها برای معرفی باگ جلوگیری کنند، هنوز هم انسان هستم و هنوز اشتباهاتی مرتکب می‌شوم. هرچه برنامه بزرگ‌تر باشد، احتمال بیشتری وجود دارد که تغییرات من باعث شکست ناخواسته‌ای شود—در عصر دیجیتال، نام آسیب‌پذیری نرم‌افزار است .

از آنجایی که بیانیه یک رشته (string) را برمی‌گرداند، من چند فاکتور ایجاد می‌کنم، به هر فاکتور چند اجرا از انواع مختلف نمایش‌ها می‌دهم و رشته‌های بیانیه را تولید می‌کنم. سپس مقایسه‌ای بین رشته جدید و برخی رشته‌های مرجعی که از قبل بررسی کرده‌ام، انجام می‌دهم. من تمام این تست‌ها را با استفاده از یک فریمورک تست راه‌اندازی می‌کنم تا بتوانم با یک کلید ساده در محیط توسعه‌ام آن‌ها را اجرا کنم. تست‌ها تنها چند ثانیه طول می‌کشند و همان‌طور که خواهید دید، من آن‌ها را به‌طور مکرر اجرا می‌کنم .

یک بخش مهم از تست‌ها، نحوه گزارش نتایج آن‌هاست. آن‌ها یا به رنگ سبز در می‌آیند، به این معنی که همه رشته‌ها با رشته‌های مرجع یکسان هستند، یا به رنگ قرمز، که نشان‌دهنده لیست شکست‌هاست—خطوطی که متفاوت به‌نظر می‌رسند. بنابراین، تست‌ها به‌طور خودکار بررسی می‌شوند. ضروری است که تست‌ها خودکار بررسی شوند. اگر این کار را نکنم، مجبور می‌شوم زمان خود را صرف بررسی دستی مقادیر تست با مقادیر روی یک کاغذ دفتری کنم و این کار باعث کاهش سرعت من می‌شود .

فریمورک‌های تست مدرن تمام ویژگی‌های لازم برای نوشتن و اجرای تست‌های خودکار را فراهم می‌کنند .

قبل از اینکه شروع به بازسازی کد کنید، مطمئن شوید که یک مجموعه تست مناسب دارید. این تست‌ها باید خودکار باشند .

در حین بازسازی، به تست‌ها تکیه می‌کنم. آنها را به عنوان یک ابزار تشخیص باگ می‌بینم که مرا در برابر اشتباهات خودم محافظت می‌کند. با نوشتن آنچه می‌خواهم در دو جا، یعنی در کد و در تست، باید در هر دو مکان اشتباه را به طور مداوم انجام دهم تا ابزار تشخیص را فریب دهم. با بررسی دوباره کارم، شانس انجام کار نادرست را کاهش می‌دهم. اگرچه ساخت تست‌ها زمان می‌برد، اما در نهایت با صرف زمان کمتر برای اشکال‌زدایی، آن زمان را با سود قابل توجهی پس می‌گیرم. این قسمت از بازسازی به قدری مهم است که یک فصل کامل به آن اختصاص می‌دهم (ساخت تست‌ها)

تجزیه کردن تابع بیان

وقتی یک تابع طولانی مانند این را بازسازی می‌کنم، به‌طور ذهنی سعی می‌کنم نقاطی را شناسایی کنم که قسمت‌های مختلف رفتار کلی را از هم جدا می‌کنند. اولین بخشی که به چشمم می‌آید، عبارت `switch` در وسط است .

تجزیه کردن تابع بیان

وقتی یک تابع طولانی مانند این را بازسازی می‌کنم، به‌طور ذهنی سعی می‌کنم نقاطی را شناسایی کنم که قسمت‌های مختلف رفتار کلی را از هم جدا می‌کنند. اولین بخشی که به چشمم می‌آید، عبارت `switch`در وسط است.

```
function statement(invoice, plays) {
    let totalAmount = 0;
    let volumeCredits = 0;
    let result = `Statement for ${invoice.customer}\n`;
    const format = new Intl.NumberFormat("en-US",
        { style: "currency", currency: "USD",
        minimumFractionDigits: 2 }).format;

    for (let perf of invoice.performances) {
        const play = plays[perf.playID];
        let thisAmount = 0;

        switch (play.type) {
            case "tragedy":
                thisAmount = 40000;
                if (perf.audience > 30) {
                    thisAmount += 1000 * (perf.audience - 30);
                }
                break;
            case "comedy":
                thisAmount = 30000;
                if (perf.audience > 20) {
                    thisAmount += 10000 + 500 * (perf.audience - 20);
                }
                thisAmount += 300 * perf.audience;
                break;
            default:
```

```
                throw new Error(`unknown type: ${play.type}`);
        }

        // اضافه کردن اعتبار حجم
        volumeCredits += Math.max(perf.audience - 30, 0);
        // اضافه کردن اعتبار اضافی برای هر ده شرکت کننده کمدی
        if ("comedy" === play.type) volumeCredits += Math.floor(perf.audience
/ 5);

        // چاپ خط برای این سفارش
        result += ` ${play.name}: ${format(thisAmount / 100)}
(${perf.audience} seats)\n`;
        totalAmount += thisAmount;
    }

    result += `Amount owed is ${format(totalAmount / 100)}\n`;
    result += `You earned ${volumeCredits} credits\n`;
    return result;
}
```

توضیحات

در این تابع، ابتدا مقادیر اولیه برای `totalAmount` **و** `volumeCredits` **تعریف می‌شود و سپس برای هر اجرای نمایشی در فاکتور، نوع نمایش بررسی می‌شود. بسته به نوع نمایش** `tragedy` **یا** `comedy`**، مبلغ مربوطه محاسبه و اعتبار حجم اضافه می‌شود. در نهایت، نتیجه نهایی شامل مبلغ کل و اعتبارهای کسب‌شده برگردانده می‌شود.**

تجزیه و تحلیل تابع

وقتی به این بخش نگاه می‌کنم، به این نتیجه می‌رسم که در حال محاسبه هزینه برای یک اجرا هستم. این نتیجه‌گیری یک بینش درباره کد است. اما همان‌طور که وارد کانینگام می‌گوید، این درک در ذهن من است—که شکلی به شدت ناپایدار از ذخیره‌سازی محسوب می‌شود. من نیاز دارم این درک را با انتقال آن از ذهن به خود کد، پایدار کنم. به این ترتیب، اگر بعداً به آن برگردم، کد به من می‌گوید که چه کاری انجام می‌دهد—دیگر نیازی نیست دوباره آن را کشف کنم.
روش قرار دادن این درک در کد، تبدیل این بخش از کد به یک تابع مستقل است و نامگذاری آن بر اساس کاری که انجام می‌دهد—چیزی مانند `amountFor(aPerformance)`**.**

وقتی می‌خواهم یک بخش از کد را به چنین تابعی تبدیل کنم، یک رویه برای انجام آن دارم که شانس اشتباه کردنم را به حداقل می‌رساند. من این رویه را یادداشت کرده‌ام و برای آسان‌تر کردن ارجاع، آن را «استخراج تابع» نامیده‌ام) ۱۰۶.(

اول، باید در این بخش به دنبال هر متغیری بگردم که پس از استخراج کد به تابع خود، دیگر در دسترس نخواهد بود. در این مورد، سه متغیر وجود دارد `perf`**،** `play` **و** `thisAmount`**. دو مورد اول توسط کد استخراج‌شده استفاده**

می‌شوند، اما تغییر نمی‌کنند، بنابراین می‌توانم آنها را به عنوان پارامتر به تابع بفرستم. متغیرهای تغییر یافته به مراقبت بیشتری نیاز دارند. در اینجا تنها یک مورد وجود دارد، بنابراین می‌توانم آن را برگردانم. همچنین می‌توانم مقداردهی اولیه آن را داخل کد استخراج‌شده قرار دهم.

تمام این موارد به این شکل به دست می‌آید:

```
function amountFor(perf, play) {
    let thisAmount = 0;
    switch (play.type) {
        case "tragedy":
            thisAmount = 40000;
            if (perf.audience > 30) {
                thisAmount += 1000 * (perf.audience - 30);
            }
            break;
        case "comedy":
            thisAmount = 30000;
            if (perf.audience > 20) {
                thisAmount += 10000 + 500 * (perf.audience - 20);
            }
            thisAmount += 300 * perf.audience;
            break;
        default:
            throw new Error(`unknown type: ${play.type}`);
    }
    return thisAmount;
}
```

توضیحات

در این تابع جدید، محاسبه هزینه برای یک اجرا بر اساس نوع نمایش انجام می‌شود. متغیر `thisAmount`مقدار هزینه را نگه می‌دارد و پس از محاسبه، به عنوان خروجی تابع برگردانده می‌شود. این کار به من این امکان را می‌دهد که در صورت نیاز، از این تابع در بخش‌های دیگر کد استفاده کنم و همچنین خوانایی و نگهداری کد را بهبود بخشم.

توضیحات مربوط به کد و روند بازسازی

وقتی از عنوانی مانند "function someName..." با کج‌نویسی برای برخی کدها استفاده می‌کنم، این بدان معناست که کدهای زیر در دامنه تابع، فایل یا کلاسی که در عنوان ذکر شده است، قرار دارد. معمولاً کدهای دیگری در این دامنه وجود دارند که آنها را نشان نمی‌دهم، زیرا در حال حاضر در موردشان بحث نمی‌کنم.

کد تابع اصلی `statement`اکنون این تابع را برای پر کردن `thisAmount`فراخوانی می‌کند:

```
function statement(invoice, plays) {
    let totalAmount = 0;
    let volumeCredits = 0;
    let result = `Statement for ${invoice.customer}\n`;
    const format = new Intl.NumberFormat("en-US",
        { style: "currency", currency: "USD",
        minimumFractionDigits: 2 }).format;

    for (let perf of invoice.performances) {
        const play = plays[perf.playID];
        let thisAmount = amountFor(perf, play);

        // اضافه کردن اعتبار حجم
        volumeCredits += Math.max(perf.audience - 30, 0);
        // اضافه کردن اعتبار اضافی برای هر ده شرکت کننده کمدی
        if ("comedy" === play.type) volumeCredits += Math.floor(perf.audience
/ 5);

        // چاپ خط برای این سفارش
        result += ` ${play.name}: ${format(thisAmount / 100)}
(${perf.audience} seats)\n`;
        totalAmount += thisAmount;
    }

    result += `Amount owed is ${format(totalAmount / 100)}\n`;
    result += `You earned ${volumeCredits} credits\n`;
    return result;
}
```

عادت‌های مهم در بازسازی

پس از اینکه این تغییر را اعمال کردم، بلافاصله کامپایل و تست می‌کنم تا ببینم آیا چیزی خراب شده است یا نه. این یک عادت مهم است که پس از هر بازسازی، هرچند ساده، تست کنم. اشتباهات راحت رخ می‌دهند—حداقل، من آنها را راحت می‌بینم. تست کردن پس از هر تغییر به این معناست که وقتی اشتباهی مرتکب شوم، تنها با یک تغییر کوچک سر و کار دارم تا بتوانم خطا را شناسایی کنم که این کار پیدا کردن و اصلاح آن را بسیار ساده‌تر می‌کند.

این اساس روند بازسازی است: تغییرات کوچک و تست کردن پس از هر تغییر. اگر سعی کنم کارهای زیادی انجام دهم، مرتکب اشتباه شدن من را مجبور می‌کند تا در یک موقعیت دشوار اشکال‌زدایی قرار بگیرم که ممکن است زمان زیادی بگیرد. تغییرات کوچک و ایجاد یک حلقه بازخورد تنگ، کلید جلوگیری از این آشفتگی هستند.

من در اینجا از واژه "کامپایل" استفاده می‌کنم تا به هر کاری که لازم است انجام شود تا جاوااسکریپت قابل اجرا باشد، اشاره کنم. از آنجایی که جاوااسکریپت به‌طور مستقیم قابل اجرا است، ممکن است هیچ‌چیزی لازم نباشد، اما در موارد دیگر ممکن است به معنای انتقال کد به یک دایرکتوری خروجی و/یا استفاده از پردازشگری مانند Babel باشد.

توضیحات مربوط به بازسازی و مراحل آن

بازسازی برنامه‌ها در مراحل کوچک انجام می‌شود، بنابراین اگر اشتباهی رخ دهد، آسان است که محل باگ را پیدا کنیم. از آنجا که این کد جاوااسکریپت است، می‌توانم تابع amountFor را به یک تابع تو در تو از تابع statement استخراج کنم. این کار مفید است زیرا به این معناست که نیازی به ارسال داده‌هایی که در دامنه تابع والد قرار دارند به تابع جدید نیست. در این مورد تفاوتی ایجاد نمی‌کند، اما این یک مشکل کمتر برای مدیریت است.

در این مورد، تست‌ها موفقیت‌آمیز بودند، بنابراین مرحله بعدی من این است که تغییرات را به سیستم کنترل نسخه محلی خود ثبت کنم. من از یک سیستم کنترل نسخه، مانند گیت یا مرکیوریال، استفاده می‌کنم که به من اجازه می‌دهد تا کامیت‌های خصوصی انجام دهم. بعد از هر بازسازی موفق، کامیت می‌کنم تا بتوانم به راحتی به یک وضعیت کاری برگردم، اگر بعدها اشتباهی مرتکب شوم. سپس تغییرات را به کامیت‌های بزرگ‌تر جمع می‌کنم قبل از اینکه تغییرات را به یک مخزن مشترک ارسال کنم.

استخراج تابع (Extract Function) یک بازسازی رایج است که می‌توان آن را اتوماسیون کرد. اگر در حال برنامه‌نویسی به زبان جاوا بودم، به طور غریزی به کلیدهای میان‌بر IDE خود برای انجام این بازسازی دست می‌یافتم. در حال حاضر، به هنگام نوشتن این متن، هیچ پشتیبانی قوی برای این بازسازی در ابزارهای جاوااسکریپت وجود ندارد، بنابراین باید این کار را به صورت دستی انجام دهم. این کار دشوار نیست، هرچند باید با متغیرهای محلی احتیاط کنم.

پس از اینکه از استخراج تابع استفاده کردم، نگاهی به آنچه استخراج کرده‌ام می‌اندازم تا ببینم آیا کارهای سریع و آسانی برای روشن‌تر کردن تابع استخراج‌شده وجود دارد. اولین کاری که انجام می‌دهم تغییر نام برخی از متغیرها برای روشن‌تر شدن آنهاست، به عنوان مثال تغییر thisAmount به result.

نتیجه‌گیری

این مراحل نشان‌دهنده یک روند منظم و تدریجی برای بهبود کیفیت کد و کاهش احتمال بروز خطاها هستند. با استفاده از تکنیک‌های بازسازی مانند استخراج تابع و مدیریت تغییرات در سیستم کنترل نسخه، می‌توان به کدی شفاف‌تر و قابل نگهداری‌تر دست یافت.

توضیحات مربوط به تابع amountFor

```
function amountFor(perf, play) {
    let result = 0;
    switch (play.type) {
        case "tragedy":
            result = 40000;
            if (perf.audience > 30) {
                result += 1000 * (perf.audience - 30);
            }
            break;
        case "comedy":
            result = 30000;
            if (perf.audience > 20) {
                result += 10000 + 500 * (perf.audience - 20);
            }
            result += 300 * perf.audience;
            break;
        default:
            throw new Error(`unknown type: ${play.type}`);
    }
    return result;
}
```

استانداردهای کدنویسی

به استاندارد کدنویسی من، همیشه نام مقدار برگشتی از یک تابع را «نتیجه (result)» می‌گذارم. به این ترتیب، همیشه می‌دانم که نقش آن چیست.

مراحل بعدی

بار دیگر کامپایل می‌کنم، تست می‌کنم و سپس تغییرات را کامیت می‌کنم. سپس به آرگومان اول می‌پردازم.

نتیجه‌گیری

با رعایت استانداردهای نام‌گذاری و مدیریت دقیق مراحل کدنویسی، می‌توان به کدی خواناتر و قابل نگهداری‌تر دست یافت. این رویکرد به بهبود کیفیت کد کمک کرده و فرایند اشکال‌زدایی را ساده‌تر می‌کند.

توضیحات مربوط به تابع amountFor

```
function amountFor(aPerformance, play) {
    let result = 0;
    switch (play.type) {
        case "tragedy":
            result = 40000;
            if (aPerformance.audience > 30) {
                result += 1000 * (aPerformance.audience - 30);
            }
            break;
        case "comedy":
            result = 30000;
            if (aPerformance.audience > 20) {
                result += 10000 + 500 * (aPerformance.audience - 20);
            }
            result += 300 * aPerformance.audience;
            break;
        default:
            throw new Error(`unknown type: ${play.type}`);
    }
    return result;
}
```

سبک کدنویسی

بار دیگر این کد مطابق با سبک کدنویسی من نوشته شده است. با توجه به اینکه زبان‌هایی مانند جاوااسکریپت دینامیک تایپ هستند، مفید است که نوع متغیرها را دنبال کنیم—از این رو، نام پیش‌فرض من برای یک پارامتر شامل نام نوع آن است. من از یک حرف نکره (indefinite article) **با آن استفاده می‌کنم مگر اینکه اطلاعات خاصی در مورد نقش متغیر در نام گنجانده شود.**

این رسم را از کنت بک یاد گرفتم و همچنان آن را مفید می‌دانم.

نتیجه‌گیری

این رویکرد به خوانایی و فهم بهتر کد کمک می‌کند و همچنین به برنامه‌نویس این امکان را می‌دهد که هنگام کار با پارامترهای مختلف، نوع آنها را به راحتی شناسایی کند. با پیروی از این کنوانسیون، کدهایی شفاف‌تر و قابل نگهداری‌تر تولید می‌شود.

توضیحات مربوط به کدنویسی و بازسازی

هر کسی می‌تواند کدی بنویسد که یک کامپیوتر بتواند آن را درک کند. اما برنامه‌نویسان خوب کدی می‌نویسند که انسان‌ها بتوانند آن را درک کنند.

ارزش تغییر نام‌ها

آیا تغییر نام‌ها ارزش تلاش را دارد؟ مطمئناً. کد خوب باید به وضوح آنچه را که انجام می‌دهد، ارتباط برقرار کند و نام‌های متغیرها کلیدی برای نوشتن کد واضح هستند. هرگز از تغییر نام‌ها برای بهبود وضوح نترسید. با ابزارهای خوب جستجو و جایگزینی، معمولاً دشوار نیست؛ تست کردن و نوع‌دهی استاتیک در زبان‌هایی که از آن پشتیبانی می‌کنند، هر گونه وقوعاتی را که از قلم انداخته‌اید، روشن می‌کند. و با ابزارهای خودکار بازسازی، تغییر نام حتی توابع پرکاربرد نیز بسیار ساده است.

حذف متغیر play

آیتم بعدی که باید برای تغییر نام در نظر بگیرم، پارامتر playاست، اما من سرنوشت متفاوتی برای آن دارم.

وقتی به پارامترهای تابع amountForفکر می‌کنم، به منبع آن‌ها نگاه می‌کنم aPerformance .از متغیر حلقه می‌آید و به طور طبیعی در هر تکرار حلقه تغییر می‌کند. اما playاز عملکرد محاسبه می‌شود، بنابراین نیازی به ارسال آن به عنوان یک پارامتر نیست—می‌توانم آن را در داخل تابع amountForدوباره محاسبه کنم. وقتی که یک تابع طولانی را شکسته‌ام، دوست دارم متغیرهایی مانند playرا حذف کنم، زیرا متغیرهای موقتی نام‌های محلی زیادی ایجاد می‌کنند که استخراج‌ها را پیچیده می‌کند.

بازسازی با استفاده از "Replace Temp with Query"

بازسازی که در اینجا استفاده می‌کنم، جایگزینی متغیر موقت با پرس‌وجو (Replace Temp with Query)است.

ابتدا سمت راست انتساب را به یک تابع جدید استخراج می‌کنم.

نتیجه‌گیری

این رویکرد به بهبود کیفیت کد کمک می‌کند و می‌تواند در درک بهتر منطق کد مؤثر باشد. با حذف متغیرهای موقت و ساده‌سازی کد، می‌توان به کدی شفاف‌تر و قابل نگهداری‌تر دست یافت.

وضیحات مربوط به تابع playFor

```
function playFor(aPerformance) {
    return plays[aPerformance.playID];
}
```

نحوه استفاده در تابع statement

```
function statement(invoice, plays) {
    let totalAmount = 0;
    let volumeCredits = 0;
    let result = `Statement for ${invoice.customer}\n`;
    const format = new Intl.NumberFormat("en-US", {
        style: "currency",
        currency: "USD",
        minimumFractionDigits: 2
    }).format;

    for (let perf of invoice.performances) {
        const play = playFor(perf);
        let thisAmount = amountFor(perf, play);

        // اضافه کردن اعتبار حجم
        volumeCredits += Math.max(perf.audience - 30, 0);

        // اضافه کردن اعتبار اضافی برای هر ده نفر در کمدی
        if ("comedy" === play.type) {
            volumeCredits += Math.floor(perf.audience / 5);
        }

        // چاپ خط برای این سفارش
        result += ` ${play.name}: ${format(thisAmount / 100)}
(${perf.audience} seats)`;
        totalAmount += thisAmount;
    }

    result += `Amount owed is ${format(totalAmount / 100)}\n`;
    result += `You earned ${volumeCredits} credits\n`;
    return result;
}
```

مراحل بعدی

پس از افزودن تابع `playFor` **و استفاده از آن در تابع** `statement`**، مراحل زیر را انجام می‌دهم:**

1. **کامپایل و تست :این مرحله برای اطمینان از عدم وجود خطاهای جدید و عملکرد صحیح کد انجام می‌شود.**
2. **کامیت :پس از اطمینان از درست بودن تغییرات، آن‌ها را در سیستم کنترل نسخه کامیت می‌کنم.**

3. **استفاده از Inline Variable: به منظور ساده‌تر کردن و بهبود خوانایی کد، از تکنیک Inline Variable استفاده می‌کنم.**

نتیجه‌گیری

ایجاد تابع `playFor` **باعث می‌شود که کد تمیزتر و خواناتر شود و مسئولیت محاسبه متغیر** `play` **را به طور مشخص‌تری تفکیک کند. این کار نه تنها کد را سازماندهی می‌کند بلکه نگهداری و توسعه آن را نیز ساده‌تر می‌کند.**

توضیحات مربوط به تابع `statement`

```
function statement(invoice, plays) {
    let totalAmount = 0;
    let volumeCredits = 0;
    let result = `Statement for ${invoice.customer}\n`;
    const format = new Intl.NumberFormat("en-US", {
        style: "currency",
        currency: "USD",
        minimumFractionDigits: 2
    }).format;

    for (let perf of invoice.performances) {
        const play = playFor(perf);
        let thisAmount = amountFor(perf, playFor(perf)); // استفاده از
playFor

        // اضافه کردن اعتبار حجم
        volumeCredits += Math.max(perf.audience - 30, 0);

        // اضافه کردن اعتبار اضافی برای هر ده نفر در کمدی
        if ("comedy" === playFor(perf).type) {
            volumeCredits += Math.floor(perf.audience / 5);
        }

        // چاپ خط برای این سفارش
        result += ` ${playFor(perf).name}: ${format(thisAmount / 100)}
(${perf.audience} seats)`;
        totalAmount += thisAmount;
    }

    result += `Amount owed is ${format(totalAmount / 100)}\n`;
    result += `You earned ${volumeCredits} credits\n`;
    return result;
}
```

مراحل بعدی

پس از اجرای تغییرات بالا، مراحل زیر را دنبال می‌کنم:

1. کامپایل و تست :این مرحله برای اطمینان از عملکرد صحیح کد و عدم وجود خطاهای جدید انجام می‌شود.
2. کامیت :پس از اطمینان از درست بودن تغییرات، آن‌ها را در سیستم کنترل نسخه کامیت می‌کنم.
3. استفاده از :**Change Function Declaration** با استفاده از این تکنیک، می‌توانم پارامتر `play`را از تابع `amountFor`حذف کنم.

مرحله اول: استفاده از تابع جدید در `amountFor`

ابتدا، درون تابع `amountFor`، تابع `playFor`را فراخوانی می‌کنم تا دیگر نیازی به ارسال `play`به عنوان پارامتر نباشد. این کار به وضوح و خوانایی کد کمک می‌کند و همچنین می‌تواند پیچیدگی‌های اضافی را کاهش دهد.

مرحله دوم: حذف پارامتر `play`

حالا با استفاده از **Change Function Declaration (124)**، می‌توانم پارامتر `play`را از تابع `amountFor`حذف کنم.

نتیجه‌گیری

این بازسازی‌ها به بهبود کیفیت کد و کاهش پیچیدگی کمک می‌کنند. با استفاده از توابع کوچک و واضح، کد خواناتر و نگهداری آن آسان‌تر می‌شود. با ادامه این روند، می‌توانم اطمینان حاصل کنم که کد به خوبی ساختاربندی شده و قابلیت فهم آن برای سایر برنامه‌نویسان افزایش می‌یابد.

به‌روزرسانی تابع `amountFor`

تابع `amountFor`را به گونه‌ای تغییر می‌دهیم که دیگر نیازی به پارامتر `play`نداشته باشد. به این ترتیب، می‌توانیم تنها از تابع `playFor`استفاده کنیم تا نوع بازی را محاسبه کنیم. در اینجا تغییرات مربوطه آمده است:

```
function amountFor(aPerformance) {
    let result = 0;
    const play = playFor(aPerformance); // دریافت نوع بازی

    switch (play.type) {
```

```
        case "tragedy":
            result = 40000;
            if (aPerformance.audience > 30) {
                result += 1000 * (aPerformance.audience - 30);
            }
            break;
        case "comedy":
            result = 30000;
            if (aPerformance.audience > 20) {
                result += 10000 + 500 * (aPerformance.audience - 20);
            }
            result += 300 * aPerformance.audience;
            break;
        default:
            throw new Error(`unknown type: ${play.type}`);
    }
    return result;
}
```

مراحل بعدی

1. **کامپایل و تست :پس از تغییرات، کد را کامپایل و تست می‌کنم تا اطمینان حاصل شود که هیچ مشکلی وجود ندارد.**
2. **کامیت :پس از اطمینان از عملکرد صحیح کد، تغییرات را در سیستم کنترل نسخه کامیت می‌کنم.**

نتیجه‌گیری

این تغییرات نه تنها کد را ساده‌تر می‌کنند بلکه آن را قابل خواندن‌تر و نگهداری را آسان‌تر می‌سازند. با استفاده از توابعی که مسئولیت مشخصی دارند، به وضوح وظایف هر بخش از کد را مشخص کرده‌ایم و به این ترتیب کیفیت کد بهبود یافته است.

به نظر می‌رسد در کد تابع `statement` **چند خطای تایپی و دستوری وجود دارد. بیایید این کد را اصلاح کنیم تا همه چیز به درستی کار کند.**

نسخه اصلاح شده تابع `statement`

```
function statement(invoice, plays) {
    let totalAmount = 0;
    let volumeCredits = 0;
    let result = `Statement for ${invoice.customer}\n`;
    const format = new Intl.NumberFormat("en-US", {
        style: "currency",
        currency: "USD",
        minimumFractionDigits: 2
    }).format;
```

```
    for (let perf of invoice.performances) {
        let thisAmount = amountFor(perf); // استفاده از تنها پارامتر
aPerformance
        // افزودن امتیاز حجم
        volumeCredits += Math.max(perf.audience - 30, 0);
        // افزودن اعتبار اضافی برای هر ده نفر تماشاچی کمدی
        if ("comedy" === playFor(perf).type) {
            volumeCredits += Math.floor(perf.audience / 5);
        }
        // چاپ خط مربوط به این سفارش
        result += ` ${playFor(perf).name}: ${format(thisAmount / 100)}
(${perf.audience} seats)\n`;
        totalAmount += thisAmount;
    }

    result += `Amount owed is ${format(totalAmount / 100)}\n`;
    result += `You earned ${volumeCredits} credits\n`;
    return result;
}
```

نکات اصلاح شده

1. **اصلاح خطای تایپی:**
   - perf.audien **به** perf.audience **تغییر داده شد.**
   - **علامت -به درستی به جای (که یک کاراکتر غیرقابل مشاهده است) در محاسبات استفاده شده است.**
2. **کاهش پارامترهای تابع:**
   - **از** amountFor(perf) **به**amountFor(perf, playFor(perf)) **تغییر داده شده است**

     زیرا با توجه به تغییرات قبلی، تابع amountFor**دیگر به پارامتر** play**نیاز ندارد.**
3. **فرمت خروجی:**
   - **متن خروجی به درستی فرمت شده و در انتهای هر خط با** \n**پایان می‌یابد.**

نتیجه‌گیری

با این اصلاحات، تابع statement**به درستی اجرا خواهد شد و می‌تواند اطلاعات مربوط به صورتحساب را به طور صحیح نمایش دهد. در نهایت، حتماً کد را کامپایل و تست کنید تا از عملکرد صحیح آن اطمینان حاصل کنید.**

را نشان می‌دهد که هزینه یک اجرا را بر اساس نوع نمایش محاسبه می‌کند. amountForکدی که ارائه کرده‌اید تابع همچنین اشاره‌ای به نگرانی‌های مربوط به عملکرد پس از تغییرات انجام شده دارد. بیایید کد را بررسی کنیم و مطمئن شویم .که همه چیز به درستی کار می‌کند

نسخه بهبود یافته تابع amountFor

```
function amountFor(aPerformance) {
    let result = 0;
    switch (playFor(aPerformance).type) {
        case "tragedy":
            result = 40000;
            if (aPerformance.audience > 30) {
                result += 1000 * (aPerformance.audience - 30);
            }
            break;
        case "comedy":
            result = 30000;
            if (aPerformance.audience > 20) {
                result += 10000 + 500 * (aPerformance.audience - 20);
            }
            result += 300 * aPerformance.audience;
            break;
        default:
            throw new Error(`unknown type: ${playFor(aPerformance).type}`);
    }
    return result;
}
```

نکات کلیدی

1. **استفاده از `playFor` در هر بار فراخوانی:**
   - **در این کد، `playFor(aPerformance)` برای دریافت نوع نمایش مورد استفاده قرار می‌گیرد. این فراخوانی در هر حالت `switch` انجام می‌شود که به این معنی است که می‌تواند در هر تکرار حلقه سه بار فراخوانی شود. در حالی که این به طور تئوری می‌تواند بر عملکرد تأثیر بگذارد، در عمل این تأثیر جزئی است، به ویژه با توجه به مزایای بهبود خوانایی و نگهداری کد.**
2. **بهبود خوانایی و نگهداری کد:**
   - **انجام این تغییرات به کاهش پیچیدگی کد و آسان‌تر کردن تغییرات بعدی کمک می‌کند. کد بهتر ساختاربندی شده و واضح‌تر می‌شود که می‌تواند به برنامه‌نویسان دیگر کمک کند تا کد را راحت‌تر درک کنند.**
3. **نگرانی‌های عملکرد:**
   - **در مورد نگرانی‌ها درباره تأثیر عملکرد، مهم است که توجه داشته باشیم که بهبود خوانایی و نگهداری کد معمولاً باید در اولویت قرار گیرد. اگر بعداً مشخص شود که عملکرد یک مسئله است، می‌توانیم به راحتی نقاطی را که نیاز به بهینه‌سازی دارند، شناسایی کنیم.**

نتیجه‌گیری

پس از انجام این تغییرات، مطمئن شوید که کد را کامپایل کرده و تست کنید تا از کارکرد صحیح آن اطمینان حاصل شود. همچنین، کد را در سیستم کنترل نسخه خود کامیت کنید تا از تغییرات ذخیره‌شده و تاریخچه کار خود محافظت کنید. این روند به شما کمک می‌کند تا همیشه بتوانید به یک وضعیت کاری برگردید.

مزیت بزرگ حذف متغیرهای محلی این است که انجام استخراج‌ها را بسیار آسان‌تر می‌کند، زیرا دامنه محلی کمتری برای مدیریت وجود دارد. در واقع، معمولاً قبل از هر گونه استخراج، متغیرهای محلی را حذف می‌کنم.

حال که به آرگومان‌های amountFor **پایان داده‌ام، به جایی که فراخوانی می‌شود نگاه می‌کنم. این متغیر برای تنظیم یک متغیر موقتی که دیگر به‌روز نمی‌شود استفاده می‌شود، بنابراین من از** Inline Variable **(۱۲۳) استفاده می‌کنم.**

```
function statement(invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;
  let result = `Statement for ${invoice.customer}\n`;
  const format = new Intl.NumberFormat("en-US", {
    style: "currency",
    currency: "USD",
    minimumFractionDigits: 2
  }).format;

  for (let perf of invoice.performances) {
    // اضافه کردن امتیازات ویژه
    volumeCredits += Math.max(perf.audience - 30, 0);

    // اضافه کردن امتیاز اضافی برای هر ده نفر تماشاگر کمدی
    if ("comedy" === playFor(perf).type) {
      volumeCredits += Math.floor(perf.audience / 5);
```

```
    }

    // چاپ خط مربوط به این سفارش
    result += ` ${playFor(perf).name}: ${format(amountFor(perf)
/ 100)} (${perf.audience} seats)\n`;
    totalAmount += amountFor(perf);
  }

  result += `Amount owed is ${format(totalAmount / 100)}\n`;
  result += `You earned ${volumeCredits} credits\n`;
  return result;
}
```

استخراج اعتبار حجمی(Volume Credits)

اینجا وضعیت فعلی بدنه تابع statement **است:**

```
function statement(invoice, plays) {
  let totalAmount = 0; // مقدار کل
  let volumeCredits = 0; // اعتبار حجمی
  let result = `Statement for ${invoice.customer}\n`; // نتیجه خروجی
  const format = new Intl.NumberFormat("en-US", {
    style: "currency",
    currency: "USD",
    minimumFractionDigits: 2
  }).format; // فرمتساز برای نمایش مقدار پول

  for (let perf of invoice.performances) {
    // اضافه کردن اعتبار حجمی
    volumeCredits += Math.max(perf.audience - 30, 0); // محاسبه اعتبار حجمی

    // اضافه کردن اعتبار اضافی برای هر ده نفر در اجرای کمدی
    if ("comedy" === playFor(perf).type)
      volumeCredits += Math.floor(perf.audience / 10);

    // چاپ خط مربوط به این سفارش
    result += ` ${playFor(perf).name}: ${format(amountFor(perf) / 100)}
(${perf.audience} seats)\n`;
    totalAmount += amountFor(perf); // اضافه کردن به مقدار کل
  }

  result += `Amount owed is ${format(totalAmount / 100)}\n`; // مقدار بدهی
```

```
  result += `You earned ${volumeCredits} credits\n`; // نمایش تعداد اعتبارها
  return result;
}
```

توضیح:

در این کد، تابع `statement` **برای ایجاد یک صورت‌حساب برای مشتری بر اساس اجرای‌های نمایشی استفاده می‌شود. اعتبار حجمی با در نظر گرفتن تعداد تماشاگران بیش از ۳۰ نفر محاسبه می‌شود و برای نمایش‌های کمدی اعتبار اضافی برای هر ۱۰ نفر تماشاگر تعلق می‌گیرد.**