廣東技術師範大學
Guangdong Polytechnic Normal University

《系统分析与设计》课程
Instructor: Wen Jianfeng
Email: wjfgdin@qq.com

系统分析与设计

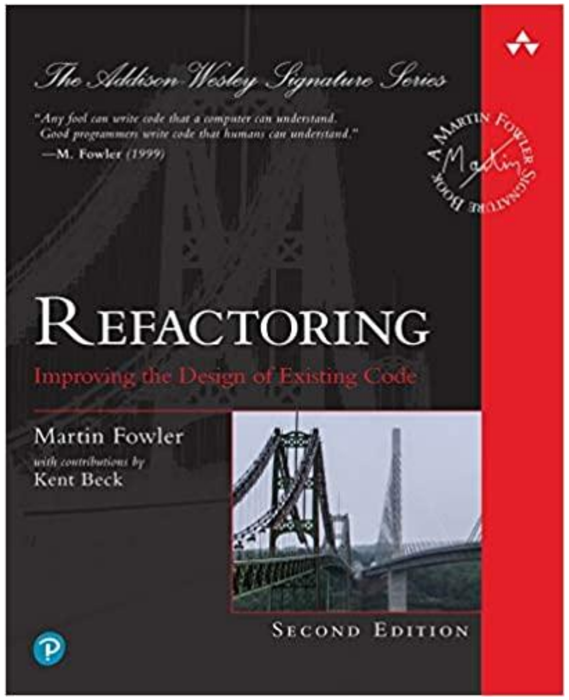
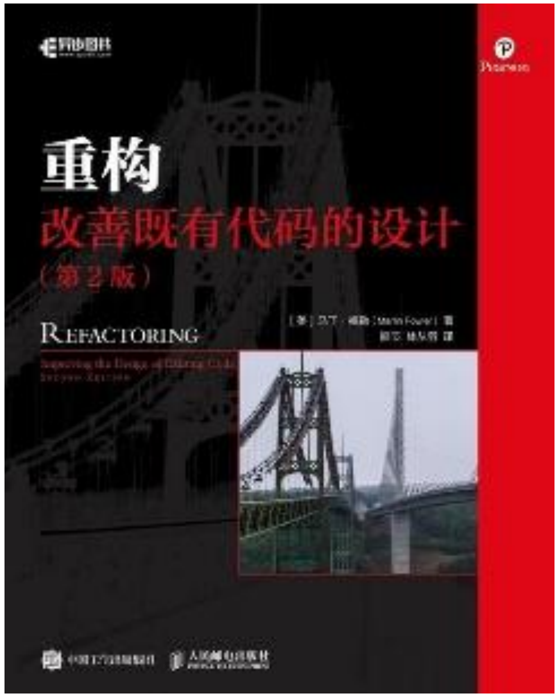
第8讲：代码重构--示例

提纲

- 重构的一个案例
- 重构的原则

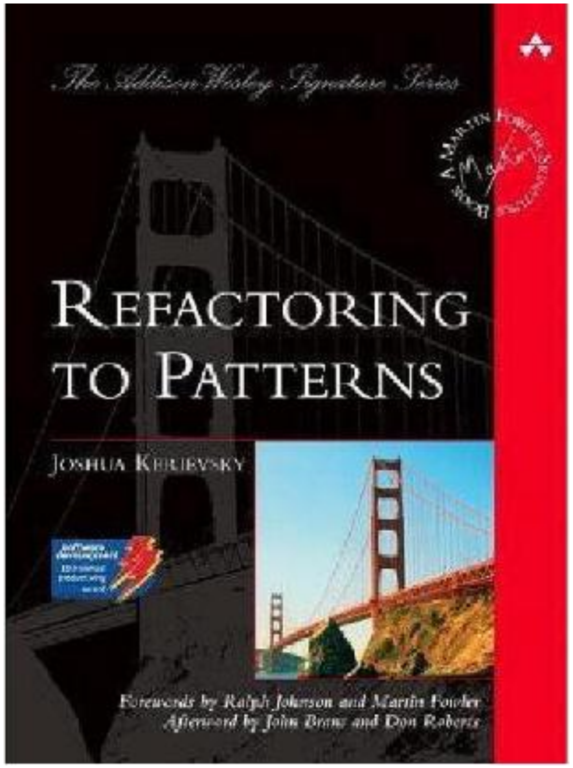
参考书一

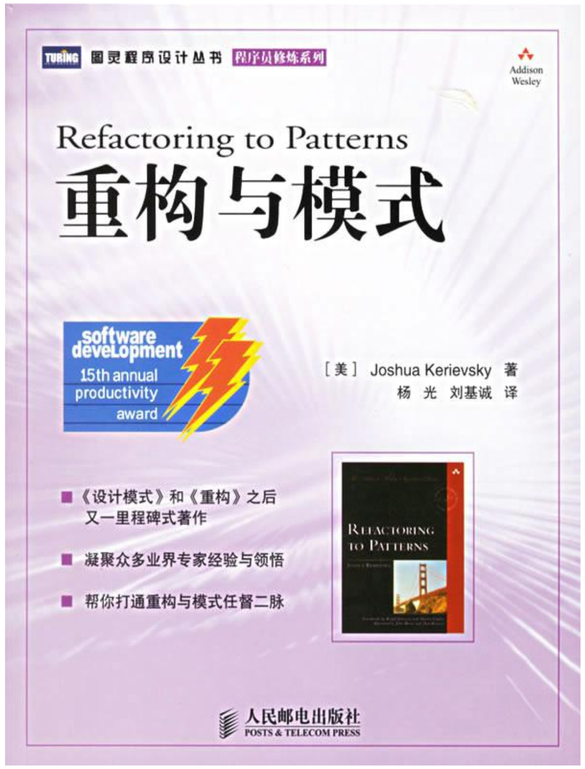
- 书名：Refactoring: Improving the Design of Existing Code (2nd Edition) (2018)
  - 作者：Martin Fowler
  - 出版社：Addision Welsey


The Addison-Wesley Signature Series
"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."
—M. Fowler (1999)
A Martin Fowler Signature Book
REFACTORING
Improving the Design of Existing Code
Martin Fowler
with contributions by
Kent Beck
Second Edition


参考书二

- 书名：Refactoring to Patterns (2004)
  - 作者：Joshua Kerievsky
  - 出版社：Addision Welsey


The Addison-Wesley Signature Series
A Martin Fowler Signature Book
Refactoring to Patterns
Joshua Kerievsky
Forewords by Ralph Johnson and Martin Fowler
Afterword by John Brant and Don Roberts


TURING 图灵程序设计丛书 程序员修炼系列
Addison Wesley
Refactoring to Patterns
重构与模式
software development 15th annual productivity award
[美] Joshua Kerievsky 著
杨 光 刘基诚 译
《设计模式》和《重构》之后又一里程碑式著作
凝聚众多业界专家经验与领悟
帮你打通重构与模式任督二脉
人民邮电出版社
POSTS & TELECOM PRESS

参考网站

- https://www.refactoring.com/catalog/
- 课堂演示代码
  - https://gitee.com/wenjianfeng/sad-student
    --> refactoring-js

案例
First Example

引入案例

- **案例：**有一个剧团经常要去各种场合表演戏剧。
  - 通常客户（customer）会指定几出剧目，而剧团则根据观众（audience）人数及剧目类型来向客户收费。
  - 该团目前出演两种戏剧：悲剧（tragedy）和喜剧（comedy）。
  - 给客户发出账单时，剧团还会根据到场观众的数量给出“现场观众量积分”（volume credit）优惠，下次客户再请剧团表演时可以使用积分获得折扣。

引入案例

- 剧目的数据存储（plays.json）

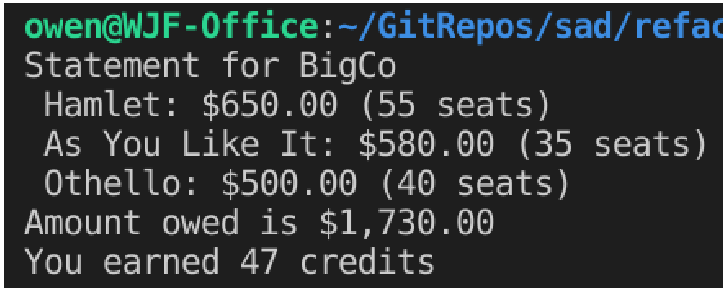
```
{
  "hamlet": {
    "name": "Hamlet",
    "type": "tragedy"
  },
  "as-like": {
    "name": "As You Like It",
    "type": "comedy"
  },
  "othello": {
    "name": "Othello",
    "type": "tragedy"
  }
}
```

引入案例

- 账单的数据存储（invoices.json）

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

```
// statement.js
function statement(invoice, plays) {
    let totalAmount = 0;
    let volumeCredits = 0;
    let result = `Statement for ${invoice.customer}\n`;
    const format = new Intl.NumberFormat(... ...).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":
                ... ...
        }
        volumeCredits += Math.max(perf.audience - 30, 0); // add volume credits
        // add extra credit for every ten comedy attendees
        if ("comedy" === play.type) volumeCredits += Math.floor(perf.audience / 10);
        // 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;
}
```



```
owen@WJF-Office:~/GitRepos/sad/refac
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
```

引入案例

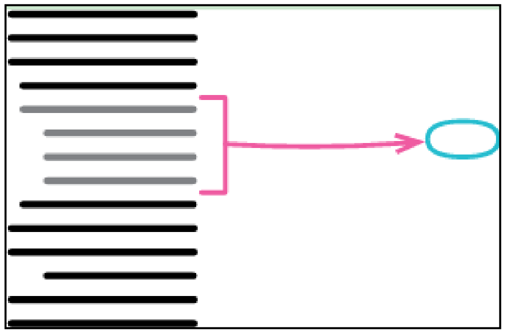
- 初始设计方案有问题吗？
  - 如果用户希望以HTML格式输出详单，咋整？
    - 在每处追加字符串到 result 变量的地方添加分支逻辑
    - 或者复制整个函数为 htmlStatement()，在其中修改输出 HTML 的部分
  - 如果演员们尝试更多的表演类型，计费方式、积分计算规则会发生变化，又咋整？
    - 需要同时改 statement() 函数和 htmlStatement() 函数，且应一致
- 因此，必须对程序代码进行重构！

Extract Function
提炼函数

重构（1）-- Extract Function

- statement()函数主要是计算一场戏剧演出的费用，即 switch 代码块所做的事，可以把这部分代码抽出来作为独立的函数。
  - Extract Function（提炼函数）：将代码中的逻辑块抽出来作为函数（提炼为 amountFor() 函数）

```
function statement(invoice, plays) {
  ... ...
    switch (play.type) {
      case "tragedy":
        thisAmount = 40000;
        if (perf.audience > 30) {
          thisAmount += 1000 * (perf.audience - 30);
        }
        break;
      case "comedy":
        ... ...
    }
}
```



重构（1）-- Extract Function

- 这个代码块中涉及3个本地变量：perf, play, thisAmount
  - 前两个在代码块中没有修改，因此可作为参数传入新函数中
  - thisAmount有修改，故可作为新函数的返回值
  - 修改变量名以提高可读性

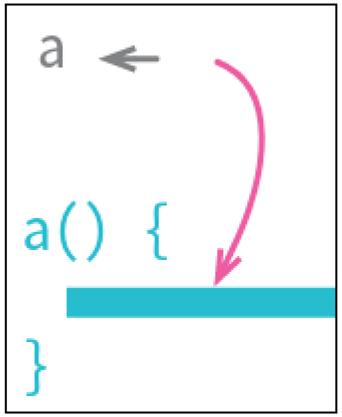
```
function amountFor(aPerformance, play) {
    let result = 0;
    switch (play.type) {
        case "tragedy":
            result = 40000;
            if (aPerformance.audience > 30) {
                result += 1000 * (aPerformance.audience - 30);
            }
            break;
        case "comedy":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

Replace Temp with Query
查询取代临时变量

重构（2） -- Replace Temp with Query

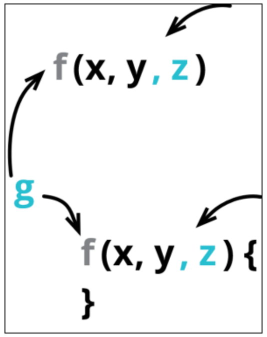
- 在amountFor(aPerformance, play)中的参数 aPerformance 从循环变量中来的，每次循环都会变化，但 play 是由 aPerformance 变量计算得到的。因此，没有必要把它作为参数传入，在 amountFor() 函数中计算它更好，这样可减少局部作用域的临时变量。
  - Replace Temp with Query（查询取代临时变量）
    - 提炼出一个函数 playFor(aPerformance)

```
/* 为了内联 play 变量 */
function playFor(aPerformance) {
    return plays[aPerformance.playID];
}
```



重构（2） -- Replace Temp with Query

- Inline Variable：内联变量
  - 内联 play 变量
    - 用 playFor(aPerformance) 代替
- Change Function Declaration：改变函数声明
  - 移除 amountFor(aPerformance, play) 的参数 play



- 处理完 amountFor 的参数后，回头看它的调用点。它被赋值给一个临时变量 thisAmount，之后就不再被修改，因此又采用内联变量手法内联它。
  - 内联 thisAmount 变量
    - 用 amountFor(perf) 代替

```
/* 为了内联 thisAmount 变量 */
function amountFor(aPerformance~~, play~~) {
    let result = 0;
    switch (~~play~~playFor(aPerformance).type) {
        case "tragedy":
            result = 40000;
            if (aPerformance.audience > 30) {
                result += 1000 * (aPerformance.audience - 30);
            }
            break;
        case "comedy":
            ... ...
        default:
            throw new
             Error(`unknown type ${~~play~~playFor(aPerformance).type}`);
    }
    return result;
}
```

```
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);~~
        // add volume credits
        volumeCredits += Math.max(perf.audience - 30, 0);
        // add extra credit for every ten comedy attendees
        if ("comedy" === playFor(perf).type)
          volumeCredits += Math.floor(perf.audience / 10);
        // print line for this order
        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;
```

重构（2） -- Replace Temp with Query

- 提炼观众量积分的逻辑
  - 类似地，我们来看一下局部变量 volumeCredits。它表示观众量积分的值。它是一个累加变量，每次循环都会更新它的值。
  - 因此，最简单的方式是将整块逻辑提炼到新函数中，然后在新函数中直接返回 volumeCredits。

```
/* 为了内联 volumeCredits 变量 */
function volumeCreditsFor(aPerformance) {
  let result = 0;
  result += Math.max(aPerformance.audience - 30, 0);
  if ("comedy" === playFor(aPerformance).type)
    result +=  Math.floor(aPerformance.audience / 10);
  return result;
}
```

```
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) {
        volumeCredits += volumeCreditsFor(perf);

        ~~volumeCredits += Math.max(perf.audience - 30, 0);~~
        ~~// add extra credit for every ten comedy attendees~~
        ~~if ("comedy" === playFor(perf).type)~~
          ~~volumeCredits += Math.floor(perf.audience / 10);~~

        // print line for this order
        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;
```

重构（2） -- Replace Temp with Query

- 对临时变量 format 的处理
  - format 的作用是格式化货币值的显示形式。
  - 可以将它替换为一个函数。
  - 同时把金额除以100的动作也搬移到函数里

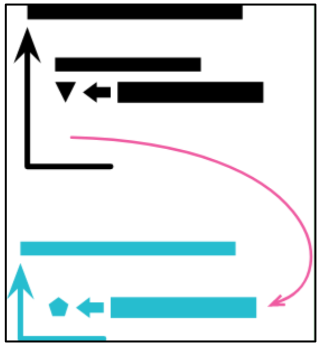
```
/* 将货币值格式化功能抽离出来作为一个函数 */
function usd(aNumber) {
    return new Intl.NumberFormat("en-US",
        {
            style: "currency", currency: "USD",
            minimumFractionDigits: 2
        }).format(aNumber/100);
}
```

```
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) {
        volumeCredits += volumeCreditsFor(perf);

        // print line for this order
        result += ` ${playFor(perf).name}:
          ${usd(amountFor(perf))} (${perf.audience} seats)\n`;
        totalAmount += amountFor(perf);
    }
    result += `Amount owed is ${usd(totalAmount)}\n`;
    result += `You earned ${volumeCredits} credits\n`;
    return result;
}
```

Split Loop
拆分循环

重构（3）-- Split Loop

- 移除观众量积分总和
  - 当前，在同一个 for 循环中，既计算了总金额，又计算了总观众量积分。
  - 可使用拆分循环（Split Loop）将 volumeCredits 的累加过程分离出来。



  - 然后用移动语句（Slide Statements）将变量声明挪动到紧邻循环的位置

```
function statement(invoice, plays) {
    let totalAmount = 0;
    ~~let volumeCredits = 0;~~
    let result = `Statement for ${invoice.customer}\n`;

    for (let perf of invoice.performances) {
        ~~volumeCredits += volumeCreditsFor(perf);~~

        // print line for this order
        result += ` ${playFor(perf).name}:
          ${usd(amountFor(perf))} (${perf.audience} seats)\n`;
        totalAmount += amountFor(perf);
    }

    let volumeCredits = 0;
    for (let perf of invoice.performances) {
      volumeCredits += volumeCreditsFor(perf);
    }

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

重构（3） -- Split Loop

- 对积分计算过程使用提炼函数（Extract Function）的重构手法

```
/* 提炼函数计算观众量积分 */
function totalVolumeCredits() {
    let volumeCredits = 0;
    for (let perf of invoice.performances) {
        volumeCredits += volumeCreditsFor(perf);
    }
    return volumeCredits;
}
```

- 然后内联 volumeCredits 变量，用函数 totalVolumeCredits() 代替

```
function statement(invoice, plays) {
    let totalAmount = 0;
    let result = `Statement for ${invoice.customer}\n`;
    for (let perf of invoice.performances) {
        // print line for this order
        result += ` ${playFor(perf).name}:
          ${usd(amountFor(perf))} (${perf.audience} seats)\n`;
        totalAmount += amountFor(perf);
    }

    ~~let volumeCredits = 0;~~
    ~~for (let perf of invoice.performances) {~~
      ~~volumeCredits += volumeCreditsFor(perf);~~
    ~~}~~

    result += `Amount owed is ${usd(totalAmount)}\n`;
    result += `You earned ${totalVolumeCredits()} credits\n`;
    return result;
}
```

重构（3）-- Split Loop

- 移除总金额 totalAmount
  - 与处理 volumeCredits 的手法一样，对 totalAmount 同样施予如下4个重构步骤：
    - 使用拆分循环（Split Loop）将 totalAmount 的累加过程分离出来。
    - 使用移动语句（Slide Statements）将变量声明挪动到紧邻循环的位置
    - 使用提炼函数（Extract Function）得到计算总金额的函数
    - 内联变量 totalAmount，用函数代替
    - 以及函数名的修改、函数内部的局部变量名的修改
  - 下面直接给出重构后的代码，具体过程就不重复了

```
/* 提炼函数计算观众量积分 */
function totalVolumeCredits() {
    let result = 0;
    for (let perf of invoice.performances) {
        result += volumeCreditsFor(perf);
    }
    return result;
}
```

```
/* 提炼函数计算总金额 */
function totalAmount() {
    let result = 0;
    for (let perf of invoice.performances) {
        result += amountFor(perf);
    }
    return result;
}
```

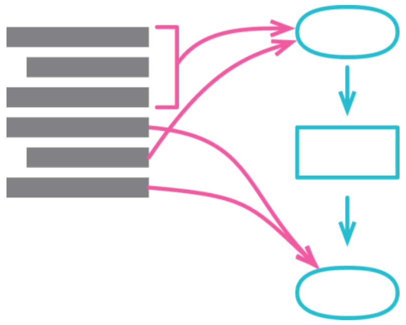
```
function statement(invoice, plays) {
    ~~let totalAmount = 0;~~
    let result = `Statement for ${invoice.customer}\n`;

    for (let perf of invoice.performances) {
        result += ` ${playFor(perf).name}:
          ${usd(amountFor(perf))} (${perf.audience} seats)\n`;
        ~~totalAmount += amountFor(perf);~~
    }

    result += `Amount owed is ${usd(totalAmount())}\n`;
    result += `You earned ${totalVolumeCredits()} credits\n`;
    return result;
}
```

现在代码结构已经好多了。顶层的 statement 函数现在只剩 7 行代码，而且它处理的都是与打印详单相关的逻辑。与计算相关的逻辑从主函数中被移走，改由一组函数来支持。每个单独的计算过程和详单的整体结构，都因此变得更易理解了。

Split Phase
拆分阶段

重构（4）-- Split Phase

- 拆分计算阶段与格式化阶段
  - 现在，可以更多关注要修改的功能部分了，也就是为这张详单提供一个 HTML 版本。
  - 我们希望同样的计算函数可以被文本版详单和 HTML 版详单共用，而不是将它们全复制粘贴到另一个新函数中。
  - 这里采用的重构手法是拆分阶段（Split Phase）
    - 将逻辑分成两部分：一部分计算详单所需的数据，另一部分将数据渲染成文本或HTML。



重构（4） -- Split Phase

- 先把打印详单代码的代码即 statement 函数的全部内容抽取为一个新的函数，命名为 renderPlainText

```
function statement(invoice, plays) {
    return renderPlainText(invoice, plays);
}
function renderPlainText(invoice, plays) {
    let result = `Statement for ${invoice.customer}\n`;
    for (let perf of invoice.performances) {
        result += ` ${playFor(perf).name}:
          ${usd(amountFor(perf))} (${perf.audience} seats)\n`;
    }
    result += `Amount owed is ${usd(totalAmount())}\n`;
    result += `You earned ${totalVolumeCredits()} credits\n`;
    return result;

    function totalAmount() {...}
    function totalVolumeCredits() {...}
    function usd(aNumber) {...}
    ... ...
}
```

重构（4）-- Split Phase

- 接着创建一个对象，作为在两个阶段间传递的中转数据结构

```
function statement(invoice, plays) {
    const statementData = {};
    return renderPlainText(statementData, invoice, plays);
}

function renderPlainText(data, invoice, plays) {
    ... ...
}
```

- 然后检查一下 renderPlainText 用到的其他参数，将它们挪到这个中转数据结构里，这样所有计算代码都可以被挪到 statement 函数中，让 renderPlainText 只操作通过 data 参数传进来的数据。
  - 如：customer 字段、performances 字段挪到 statementData 对象中，这样 invoice 参数便可移除了

```
function statement(invoice, plays) {
    const statementData = {};
    statementData.customer = invoice.customer;
    statementData.performances = invoice.performances;
    return renderPlainText(statementData, ~~invoice,~~ plays);
}

function renderPlainText(data, ~~invoice,~~ plays) {
    let result = `Statement for ${data.customer}\n`;
    for (let perf of data.performances) {
    ... ...
    }

    function totalAmount() {
        let result = 0;
        for (let perf of data.performances) {
            result += amountFor(perf);
        }
        return result;
    }

    ... ...
}
```

重构（4）-- Split Phase

- 将“剧目名称”信息也从中转数据中获得。为此，需要使用 play 中的数据填充 performance 对象（因“剧目名称”信息在 play 对象中）

```
function statement(invoice, plays) {
    const statementData = {};
    statementData.customer = invoice.customer;
    // 用map()让performances数组中的每个元素都调用enrichPerformance函数得到新值
    statementData.performances = invoice.performances.map(enrichPerformance);
    return renderPlainText(statementData, plays);

    function enrichPerformance(aPerformance) {
        const result = Object.assign({}, aPerformance); //得到aPerformance副本
        return result;
    }
}
```

- 在 enrichPerformance 函数中为返回的对象副本添加 play 字段

```
function enrichPerformance(aPerformance) {
    const result = Object.assign({}, aPerformance);
    result.play = playFor(result);
    return result;
```

重构（4）-- Split Phase

- 然后用搬移函数（Move Function）手法替换 renderPlainText 中对 playFor 函数的所有引用点，让它们使用新数据

```
function renderPlainText(data~~, plays~~) {
    let result = `Statement for ${data.customer}\n`;
    for (let perf of data.performances) {
        result += ` ${perf.play.name}:
          ${usd(amountFor(perf))} (${perf.audience} seats)\n`;
    }
    ... ...
    function amountFor(aPerformance) {
        ... ...
        switch (aPerformance.play.type)
        ... ...
    }
    function volumeCreditsFor(aPerformance) {
        ... ...
        if ("comedy" === aPerformance.play.type)
        ... ...
    }
    ... ...
}
```

重构（4） -- Split Phase

- 类似地，用前面搬移 playFor 函数的手法搬移 amountFor 函数

```
function enrichPerformance(aPerformance) {
    const result = Object.assign({}, aPerformance);
    result.play = playFor(result);
    result.amount = amountFor(result);
    return result;
}
```

```
function renderPlainText(data) {
    let result = `Statement for ${data.customer}\n`;
    for (let perf of data.performances) {
        result += ` ${perf.play.name}:
          ${usd(perf.amount)} (${perf.audience} seats)\n`;
    }
    ... ...
    function totalAmount() {
        ... ...
        for (let perf of data.performances) {
            result += perf.amount;
        } ... ...
```

重构（4）-- Split Phase

- 类似地，搬移观众量积分计算的函数 totalVolumeCredits

```
function enrichPerformance(aPerformance) {
    const result = Object.assign({}, aPerformance);
    result.play = playFor(result);
    result.amount = amountFor(result);
    result.volumeCredits = volumeCreditsFor(result);
    return result;
}
```

```
function renderPlainText(data) {
    ... ...

    function totalVolumeCredits() {
        let result = 0;
        for (let perf of data.performances) {
            result += perf.volumeCredits;
        }
        return result;
    }
    ......
```

重构（4）-- Split Phase

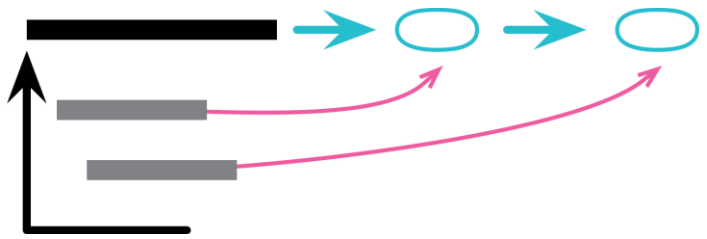
- 最后，将两个计算总数的函数搬移到 statement 函数

```
function statement(invoice, plays) {
    const statementData = {};
    statementData.customer = invoice.customer;
    statementData.performances = invoice.performances.map(enrichPerformance);
    statementData.totalAmount = totalAmount(statementData);
    statementData.totalVolumeCredits = totalVolumeCredits(statementData);
    return renderPlainText(statementData);

    function totalAmount(data) { ... ... }
    function totalVolumeCredits(data) { ... ... }
}
```

```
function renderPlainText(data) {
    let result = `Statement for ${data.customer}\n`;
    for (let perf of data.performances) {
      ... ...
    }
    result += `Amount owed is ${usd(data.totalAmount)}\n`;
    result += `You earned ${data.totalVolumeCredits} credits\n`;
    return result;
```

重构（5）-- Split Phase（续）

- 函数搬移完成后，totalAmount 和 totalVolumeCredits 函数中的循环语句可以使用管道取代循环（Replace Loop with Pipeline）的手法进行重构



```
function totalAmount(data) {
    // 数组对象的reduce方法根据累加器函数将数组元素从左至右累加为一个值
    return data.performances.reduce((total, p) => total + p.amount, 0);
}

function totalVolumeCredits(data) {
    return data.performances.reduce((total, p) =>
                                    total + p.volumeCredits, 0);
}
```

重构（5）-- Split Phase（续）

- 现在可以把第一阶段的代码提炼到一个独立的函数里了，并把它搬移到另一个文件 createStatement.js

```
// createStatement.js
exports.createStatementData = function createStatementData(invoice, plays) {
    const result = {};
    result.customer = invoice.customer;
    result.performances = invoice.performances.map(enrichPerformance);
    result.totalAmount = totalAmount(result);
    result.totalVolumeCredits = totalVolumeCredits(result);
    return result;

    function enrichPerformance(aPerformance) {...}
    function playFor(aPerformance) {...}
    function amountFor(aPerformance) {...}
    function volumeCreditsFor(aPerformance) {...}
    function totalAmount(data) {...}
    function totalVolumeCredits(data) {...}
}
```

```
// statement.js
const CreateStatementData = require('./createStatementData.js');
function statement(invoice, plays) {
    return renderPlainText(CreateStatementData.createStatementData(invoice,
                                                                   plays));
}

function renderPlainText(data) {
    ... ...
}

function usd(aNumber) {
    ... ...
}
```

重构（5）-- Split Phase（续）

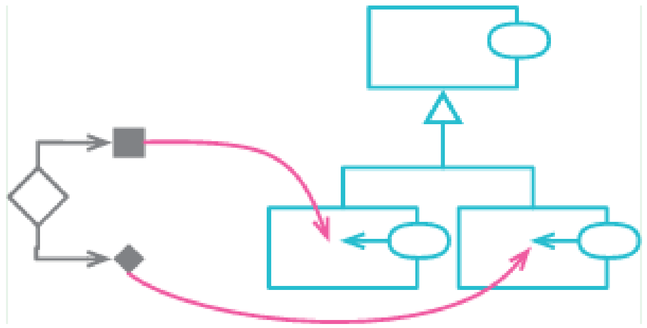
- 重构至此，便完成了阶段的划分，即把计算阶段和格式化阶段分离开来了，它们之间通过一个数据结构作为桥梁
  - 现在，编写HTML版本的对账单就很容易了

```
// statement.js
const CreateStatementData = require('./createStatementData.js’);
... ...
function htmlStatement(invoice, plays) {
    return renderHtml(CreateStatementData.createStatementData(invoice,
                                                                 plays));
}
function renderHtml(data) {
    let result = `<h1>Statement for ${data.customer}</h1>\n`;
    result += "<table>\n";
    result += "<tr><th>play</th><th>seats</th><th>cost</th></tr>";
    ... ...
}
... ...
```

Replace Conditional with Polymorphism
多态取代条件表达式

重构（6） -- Replace Conditional with Polymorphism

- 考虑这个需求：支持更多类型的戏剧，以及支持它们各自的价格计算和观众量积分计算
  - 即：戏剧的类型具有多态性
  - 按目前的代码，可以在计算函数 amountFor 里面添加分支逻辑，这违背了开闭原则
  - 可以使用多态取代条件表达式（Replace Conditional with Polymorphism）的重构手法



重构（6） -- Replace Conditional with Polymorphism

- enrichPerformance 函数用每场演出的数据来填充中转数据结构，它直接调用了计算价格和观众量积分的函数。现在创建一个类 PerformanceCalculator，通过这个类来调用这些函数

```
function enrichPerformance(aPerformance) {
    const calculator = new PerformanceCalculator(aPerformance);
    const result = Object.assign({}, aPerformance);
    result.play = playFor(result);
    result.amount = amountFor(result);
    result.volumeCredits = volumeCreditsFor(result);
    return result;
}
```

```
class PerformanceCalculator {
    constructor(aPerformance) {
        this.performance = aPerformance;
    }
}
```

重构（6） -- Replace Conditional with Polymorphism

- 使用改变函数声明的手法将 performance 的 play 字段传给 PerformanceCalculator，方便数据集中在一起

```
function enrichPerformance(aPerformance) {
    const calculator = new PerformanceCalculator(aPerformance);
    const result = Object.assign({}, aPerformance);
    result.play = ~~playFor(result)~~ calculator.play;
    result.amount = amountFor(result);
    result.volumeCredits = volumeCreditsFor(result);
    return result;
}
```

```
class PerformanceCalculator {
    constructor(aPerformance, aPlay) {
        this.performance = aPerformance;
        this.play = aPlay;
    }
}
```

重构（6） -- Replace Conditional with Polymorphism

- 将函数 amountFor 搬移至类 PerformanceCalculator

```
class PerformanceCalculator {
    constructor(aPerformance, aPlay) {
        this.performance = aPerformance;
        this.play = aPlay;
    }
    get amount() {
        let result = 0;
        switch (~~playFor(aPerformance).type~~ this.play.type) {
            case "tragedy":
              result = 40000;
              if (~~aPerformance.audience~~ this.performance.audience > 30) {
                    result += 1000 * (this.performance.audience - 30);
                }
                break;
            case "comedy":
              ... ...
        }
        return result;
    }
}
```

重构（6） -- Replace Conditional with Polymorphism

- 类似地，将计算观众量积分的函数 volumeCreditsFor 也搬移至类中

```
class PerformanceCalculator {
    constructor(aPerformance, aPlay) {
        this.performance = aPerformance;
        this.play = aPlay;
    }

    get amount() { ... ...}

    get volumeCredits() {
        let result = 0;
        result += Math.max(this.performance.audience - 30, 0);
        if ("comedy" === this.play.type) result +=
                                 Math.floor(this.performance.audience / 5);
        return result;
    }
}
```

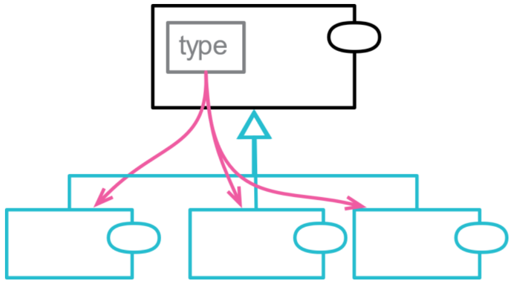
重构（6） -- Replace Conditional with Polymorphism

- 相应将函数 enrichPerformance 中原调用 amountFor 和 volumeCreditsFor 的地方改为用对象的方法代替

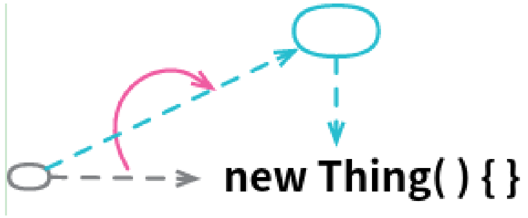
```
function enrichPerformance(aPerformance) {
    const calculator =
        new PerformanceCalculator(aPerformance, playFor(aPerformance));
    const result = Object.assign({}, aPerformance);
    result.play = calculator.play;
    result.amount = ~~amountFor(result)~~ calculator.amount;
    result.volumeCredits = ~~volumeCreditsFor(result)~~ calculator.volumeCredits;
    return result;
}
```

重构（6） -- Replace Conditional with Polymorphism

- 搬移函数完成后，便可开始着手实现多态化了
  - 应用以子类取代类型码（Replace Type Code with Subclasses）引入子类，弃用类型代码（play.type）。也即是为演出计算器类 PerformanceCalculator 引入子类



  - 为了在 createStatementData 中正确获取子类，需要使用工厂函数取代构造函数（Replace Constructor with Factory Function）



```
function enrichPerformance(aPerformance) {
    ~~const calculator =~~
~~        new PerformanceCalculator(aPerformance, playFor(aPerformance));~~
    const calculator =
        createPerformanceCalculator(aPerformance, playFor(aPerformance));
    const result = Object.assign({}, aPerformance);
    result.play = calculator.play;
    result.amount = calculator.amount;
    result.volumeCredits = calculator.volumeCredits;
    return result;
}
```

```
function createPerformanceCalculator(aPerformance, aPlay) {
  switch(aPlay.type) {
    case "tragedy": return new TragedyCalculator(aPerformance, aPlay);
    case "comedy" : return new ComedyCalculator(aPerformance, aPlay);
    default:
      throw new Error(`unknown type: ${aPlay.type}`);
  }
}

class TragedyCalculator extends PerformanceCalculator {
}

class ComedyCalculator extends PerformanceCalculator {
}
```

重构（6） -- Replace Conditional with Polymorphism

- 准备好实现多态的类结构后,我就可以继续使用以多态取代条件表达式（Replace Conditional with Polymorphism）手法了
  - 搬移悲剧的价格计算逻辑

```
class TragedyCalculator extends PerformanceCalculator {
    get amount() {
      let result = 40000;
      if (this.performance.audience > 30) {
        result += 1000 * (this.performance.audience - 30);
      }
      return result;
    }
}
```

重构（6） -- Replace Conditional with Polymorphism

- 搬移喜剧的价格计算逻辑

```
class ComedyCalculator extends PerformanceCalculator {
    get amount() {
      let result = 30000;
      if (this.performance.audience > 20) {
        result += 10000 + 500 * (this.performance.audience - 20);
      }
      result += 300 * this.performance.audience;
      return result;
    }
}
```

- 搬移了悲喜剧计算逻辑后的父类

```
class PerformanceCalculator {
    constructor(aPerformance, aPlay) {... ...}
    get amount() {
        throw new Error('subclass responsibility');
    }
    get volumeCredits() {... ...}
}
```

重构（6） -- Replace Conditional with Polymorphism

- 接下来要替换的条件表达式是观众量积分的计算
  - 观察发现大多数剧类在计算积分时都会检查观众数是否达到30，仅一小部分剧类有所不同。因此，将更为通用的逻辑放到父类作为默认条件，出现不同时按需覆盖它

```
class PerformanceCalculator {
    constructor(aPerformance, aPlay) {... ...}
    get amount() {...}
    get volumeCredits() {
        return Math.max(this.performance.audience - 30, 0);
    }
}

class ComedyCalculator extends PerformanceCalculator {
    get amount() {...}
    get volumeCredits() {
        return super.volumeCredits +
                         Math.floor(this.performance.audience / 10);
    }
}
```

重构示例的小结

- 重构示例的小结
  - 通过这个简单的例子，同学们应该对“重构怎么做”有一点感觉了
    - 示例中使用了多种重构手法，如：提炼函数、内联变量、搬移函数、以多态取代条件表达式等
  - 示例中的重构有3个较为重要的节点：
    - 将原函数分解成一组嵌套的函数
    - 应用拆分阶段手法分离计算逻辑与输出格式化逻辑
    - 为类引入多态性来处理计算逻辑
  - 重构早期的主要动力是尝试理解代码如何工作，找到一些感觉后，再通过重构将这些感觉从脑海里搬回到代码中
  - 好代码的检验标准是：人们是否能轻而易举地修改它
  - 重构的节奏：测试→小修改→测试→小修改→测试…

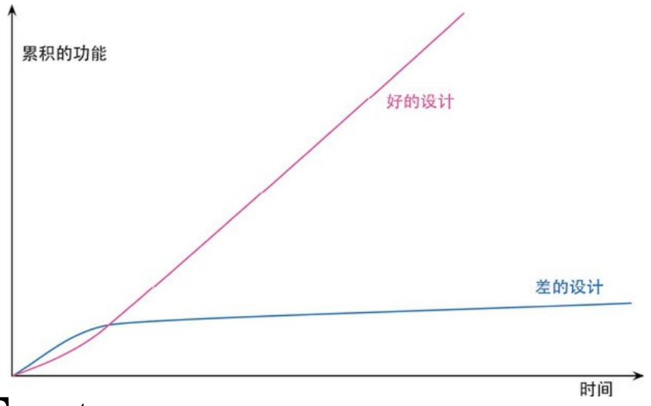
重构的原则
Principles in Refactoring

什么是重构？（What Refactor）

- 重构的定义
  - 重构（名词）：对软件内部结构的一种调整，目的是在不改变软件可观察行为的前提下，提高其可理解性，降低其修改成本
  - 重构（动词）：使用一系列重构手法，在不改变软件可观察行为的前提下，调整其结构

为什么要重构？(Why Refactor?)

- Refactoring Improves the Design of Software
  重构改进软件的设计
- Refactoring Makes Software Easier to Understand
  重构使软件更易于理解
- Refactoring Helps You Find Bugs
  重构有助于找出Bugs
- Refactoring Helps You Program Faster
  重构有助于提高编程速度



何时进行重构？(When Refactor?)

- 何时进行重构
  - 预备性重构：让添加新功能更容易
  - 帮助理解的重构：使代码更易懂
  - 捡垃圾式重构
  - 有计划的重构和见机行事的重构
  - 长期重构
  - 评审代码时重构

- 何时不该重构
  - 有些情形重写可能比重构更合适
  - 临近deadline时，应避免重构

重构的难题 (Problems with Refactoring)

- 延缓新功能开发
  - 尽管重构的目的是加快开发速度，但是仍旧很多人认为花在重构的时间是在拖慢新功能的开发进度。这种看法仍然很普遍，这可能是导致人们没有充分重构的最大阻力所在
- 代码所有权
  - 很多重构手法不仅会影响一个模块内部，还会影响该模块与系统其他部分的关系。代码所有权的边界会妨碍重构，因为一旦修改就一定会破坏使用者的程序

重构的难题 (Problems with Refactoring)

- 分支
  - 分支合并本来就是一个复杂的问题，随着Feature分支存在的时间加长，合并的难度会指数上升
- 测试
  - 绝大多数情况下，如果想要重构，得先有可以自测试的代码
- 遗留代码
  - 遗留代码往往很复杂，测试又不足，而且最关键的是代码是别人写的

重构和设计 (Refactoring and Design)

- 重构有个特殊的角色：作为设计的补充

- 有一种观点是用重构代替预先的设计，即：不用作任何设计，先写出代码实现，再进行重构
  - 但这不是最有效的途径

- 重构改变了预先设计的角色，即：仍作预先设计，只要得到足够合理的解决方案就可以，将来随着理解的加深，再进行重构
  - 简化设计，避免初始设计过于复杂
  - 重构使设计简单而不至于丧失灵活性

重构和性能 (Refactoring and Performance)

- 有时为了使软件易于理解而进行重构，可能导致程序的运行性能受到影响
  - 重构节省出来的开发时间可用于性能优化
  - 重构后的软件使性能优化更容易
  - 重构好的软件使性能分析基于更加合适的粒度

Thank you!