Parte IV – Refactoring y diseño simple

Tema 06 – Refactoring – Parte 2

1

Cambiar declaración de función: Motivación (pág. 124)

- Las llamadas a funciones representan el punto de unión entre las distintas partes del programa
- Encontrar buenos nombres para las funciones a veces no es fácil (a veces es necesario escribir un comentario)
- A los parámetros de una función les pasa algo parecido, ya que dictan cómo una función encaja con el "resto de su mundo". Por ejemplo, si tengo una función para dar formato al número de teléfono de una persona, y esa función toma a una persona como su argumento, entonces no puedo usarla para dar formato al número de teléfono de una empresa. Si sustituyo el parámetro persona con el número de teléfono en sí, entonces el código de formato es más útil
- Lo anterior puede ayudar a reducir el acoplamiento
- No hay una respuesta única al caso anterior, ¿y si la lógica evoluciona y necesito algo de la persona para dar el formato al teléfono?

2

2

Cambiar declaración de función: Mecánica (pág. 124)

- Mecánica simple: Si puedo hacer el cambio en la declaración y sus llamadas fácilmente
  - Si se está eliminando un parámetro, asegúrese de que no esté referenciado en el cuerpo de la función
  - Cambia la declaración del método a la declaración deseada
  - Encuentra todas las referencias a la declaración del método anterior, actualícelas a la nueva
  - Prueba
  - Recuerda que puedes revertir los cambios si no se satisfacen las expectativas
- Si la mecánica simple no es suficiente puede ser necesario refactorizar primero el cuerpo de la función y aplicar antes otras técnicas

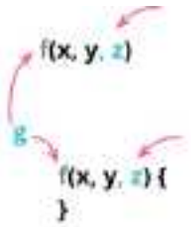
3

Cambiar la declaración de la función (pág. 124). Mecánica para Migraciones.

- Si es necesario, refactoriza el Código de la función para hacer más sencillo el siguiente paso.
- Utiliza *Extraer Función (pág. 106)* sobre el Código de la función para crear la nueva función.
- Si la nueva función tendrá el mismo nombre que la Antigua, asigna a la nueva un nombre temporal para que sea fácil de buscar.
- Si la función extraida necesita parámetros adicionales, utiliza la mecánica simple para añadirlo
- Probar.
- Aplicar *Inline Function (pág. 115)* a la función antigua.
- Si se usó un nombre temporal, Volver a aplicar el método 124 para restaurar el nombre original.
- Probar.

4

Cambiar declaración de función: (pág. 124)




```
function circum(radius) {...}
```

⇩

```
function circumference(radius) {...}
```

5

5

Renombrar variable (pág. 137). Motivación

- Dar los nombres adecuados es la base de la Buena programación.
- Los campos que van más allá de su presencia en una simple función, requieren un cuidado adicional.

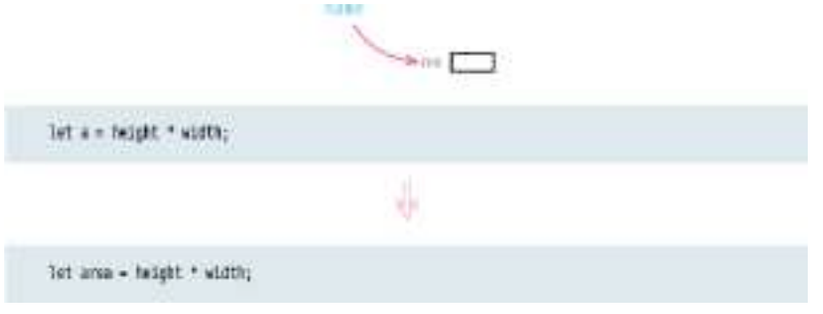
6

Renombrar variable (pág. 137). Mecánica

- Si el nombre de la variable es ampliamente utilizado considerar Encapsular Variable *(pág. 132)*.
- Buscar todas las referencias a la variable y cambiar cada una de ellas.
- Si hay referencias desde otro Código, utiliza una variable pública (esto no puede hacerse con refactoring)
- Si la variable no cambia, se puede copiar a otra con el mismo nombre y entonces cambiar de forma gradual, probando después de cada cambio.
- Probar.

7

Renombrar variable (pág. 137).



8

Renombrar campo (pág. 244). Motivación

- Los nombres de los campos en los registros son de especial importancia para la comprensión del código.
- Es importante mantener claras estas estructuras.
- Puedes necesitar renombrar un campo en una estructura de registro, pero este concepto también aplica a clases.

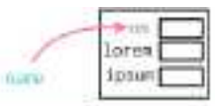
9

Renombrar campo (pág. 244). Mecánica.

- Si el registro tiene un alcance limitado, renombrar todos los accesos al campo y sus pruebas. No es necesario continuar con el resto de la mecánica.
- Si el registro no está encapsulado, aplicar Encapsular Registro (pág. 162).
- Renombrar el campo privado dentro del objeto y ajustar el método interno para ajustarse a la nueva declaración
- Probarlo.
- Si el constructor usa el nombre, aplicar Cambiar la declaración de la función *(pág. 124)* para renombrarlo.
- Aplicar Renombrar Función *(pág. 124)* para los ancestors.

10

Renombrar campo (pág. 244). Mecánica.




```
class Organization {
  get name() {...}
}
```

```
class Organization {
  get title() {...}
}
```


11

Extraer función: Motivación (pág. 106)

- Extraer un fragmento de código y crear su propia función con un nombre de acuerdo con su propósito.
- Es una de las técnicas más comunes (aplicable a método en OO)
- Observo un fragmento de código, entiendo lo que hace y lo extraigo en una función con nombre representativo
- ¿Cuál es el límite? Más allá de la extensión del código y de la reutilización del mismo estaría la capacidad para comprender de manera sencilla qué hace. Si no se sabe lo puedes extraer y ponerle nombre.
- Como norma es mejor no hacer funciones demasiado largas desde el principio y así evitar la refactorización por este motivo

12

Extraer función: Mecánica (pág. 106)

- Crea una nueva función y nómbrala por lo que hace, no por cómo lo hace
- La facilidad para nombrar la nueva función es una prueba de la necesidad de extraerla
- Si es posible anídala
- Sé cuidadoso con las variables y su ámbito: ¿Qué ha de pasarse como parámetro?
- Si la extracción obliga a usar demasiadas variables locales, quizá haya que desistir
- Reemplaza el código extraído con la llamada a la función y prueba

13

13

Extraer función (pág. 106)

14

14

Desplazar sentencias (pág. 223). Motivación

- El código se entiende mejor cuando los elementos que están relacionados aparecen juntos.
- Si varias líneas de código procesan un mismo registro, es mejor que estén juntas.
- Por ejemplo mucha gente prefiere poner juntas todas las declaraciones de variable al principio del método.

15

Desplazar sentencias (pág. 223). Mecánica

- Identifica la posición a la que moverás el Código. Examina las sentencias entre el origen y el destino para ver si hay alguna interferencia. Si hay alguna interferencia, no realices la acción.
- Un fragmento no se puede mover más atrás del lugar en el que cualquier otro elemento al que referencia haya sido declarado.
- Un elemento no pude ser movido hacia adelante más allá de cualquier otro elemento que lo referencie.
- Un fragmento que modifique un elemento no puede ser desplazado más allá de cualquier otro elemento que referencie al elemento modificado.
- Corta el fragmento de código y pégalo en la posición destino.
- Probar todo.

16

Desplazar sentencias (pág. 223)

```
const pricingPlan = retrievePricingPlan();
const order = retrieveOrder();
let charge;
const chargePerUnit = pricingPlan.unit;
```



```
const pricingPlan = retrievePricingPlan();
const chargePerUnit = pricingPlan.unit;
const order = retrieveOrder();
let charge;
```


17

Enviar el método a la superclase (pág. 350). Motivación

- La forma más sencilla de utilizarlo es cuando hay varios métodos con el mismo contenido (pág. copy-paste)
- Cuando el método utiliza propiedades de la subclase, aplicar Enviar la propiedad a la superclase (pág. 353).

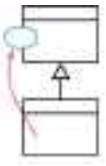
18

Enviar el método a la superclase (pág. 350). Mecánica.

- Revisar los métodos para asegurarse de que son iguales.
- Comprobar que todas las llamadas al método y las referencias a las propiedades dentro del método se refieren a elementos que pueden ser llamados también desde la superclase.
- Si los métodos tienen nombres diferentes, utilice Cambiar la declaración de la función (pág. 124) para que todos tengan el mismo nombre que se utilizará en la superclase.
- Crear el nuevo método en la superclase. Copiar el cuerpo del método dentro.
- Realizar las comprobaciones.
- Borrar el método de una sublcase.
- Probar.
- Repetir el borrado para cada subclase hasta terminar.

19

Enviar el método a la supeclase (pág. 350).



20

Dividir en Fases (pág. 154). Motivación

- Cuando el código tiene que procesar dos cosas diferentes, es interesante dividirlo.
- Así podemos tratar cada problema por separado.
- Una de las formas más limpias de dividirlo en dos fases secuenciales.
- Es común hacerlo en grandes desarrollos.

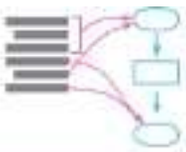
21

Dividir en Fases (pág. 154). Motivación

- Extraer el código de la segunda fase en su propia función.
- Probarlo.
- Introducir una estructura de datos intermedia como parámetro adicional de la función extraída.
- Probar.
- Examinar cada parámetro de la segunda fase extraída. Si se usa en la primera fase, llevarlo a la estructura de datos intermedia.
- Probar después de cada movimiento.
- A veces un parámetro no debe ser utilizado en la segunda fase. En ese caso usar use *Move Statements to Callers (pág. 217)*.
- Aplicar Extraer Función (pág. 106) en la primera fase del código retornando la estructura de datos intermedia.

22

Dividir en Fases (pág. 154).



```
const orderData = orderString.split(/\s+/);
const productPrice = priceList[orderData[0].split("-")[1]];
const orderPrice = parseInt(orderData[1]) * productPrice;
```

```
const orderRecord = parseOrder(order);
const orderPrice = price(orderRecord, priceList);

function parseOrder(aString) {
  const values = aString.split(/\s+/);
  return ({
    product: values[0].split("-")[1],
    quantity: parseInt(values[1]),
  });
}
function price(order, priceList) {
  return order.quantity * priceList[order.product];
}
```


Mover Función (pág. 198).
Motivación

- Asegurar que todos los elementos relacionados están agrupados juntos y los enlaces entre ellos son fáciles de localizar y comprender.
- Por ejemplo cuando un método referencia más elementos de otras regiones de código de los que referencia en la suya propia.

Mover Función (pág. 198). Mecánica

- Examinar todos los elementos relacionados con la función elegida para mover. Considerar si deben ser movidos también.
- Comprobar si la función elegida es un método polimórfico.
- Copiar la función al contexto destino y ajustarla para que encaje bien.
- Realizar un análisis estático.
- Imaginar como debe llamarse a la función desde el lugar en el que se encontraba anteriormente.
- Convertir la función origen en una función delegada.
- Probar.

25

Mover Función (pág. 198)

```
class Account {
  get overdraftCharge() {...}
```



```
class AccountType {
  get overdraftCharge() {...}
```


26

Reemplazar condicional por polimorfismo: Motivación (pág. 272)

- Posible lógica condicional compleja
- ¿Se puede formar un conjunto de tipos con comportamiento o respuesta diferente? Elimina posible duplicación de la lógica creando clases para cada uno de los casos y aprovechando el polimorfismo para resaltar el comportamiento específico de cada tipo
- ¿Hay un caso base con posibles variantes? Se puede poner la lógica en general en la superclase y especificarla con subclases
- ¿Siempre tengo que usar polimorfismo? No, solo si ayuda a mejorar una lógica compleja

27

27

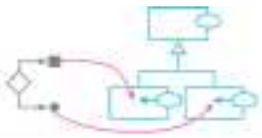
Reemplazar condicional por polimorfismo: Mecánica (pág. 272)

- Si las clases no existen hay que crearlas
- Utilice la función en la llamada
- Traslada la lógica condicional a la superclase
- Si la lógica condicional no es una función autónoma, usa la extracción de método (*106*) para hacerlo
- Elige una de las subclases. Crea un método de subclase que invalide el método de declaración condicional. Copia el cuerpo de ese tramo de la declaración condicional en el método de subclase y ajústalo
- Repite estos pasos para cada rama del condicional.
- Deja un caso predeterminado para el método de la superclase. O, si la superclase debe ser abstracta, declara ese método como abstracto o lanza un error para demostrar que debe ser responsabilidad de una subclase.

28

28

Reemplazar condicional por polimorfismo (pág. 272)



```
switch (bird.type) {
  case 'EuropeanSwallow':
    return "average";
  case 'AfricanSwallow':
    return (bird.numberOfCoconuts > 2) ? "tired" : "average";
  case 'NorwegianBlueParrot':
    return (bird.voltage > 100) ? "scorched" : "beautiful";
  default:
    return "unknown";
}
```

```
class EuropeanSwallow {
  get plumage() {
    return "average";
  }
class AfricanSwallow {
  get plumage() {
    return (this.numberOfCoconuts > 2) ? "tired" : "average";
  }
class NorwegianBlueParrot {
  get plumage() {
    return (this.voltage > 100) ? "scorched" : "beautiful";
  }
```

29

29

Extraer clase: Motivación (pág. 182)

- Las clases van creciendo, inevitablemente: una responsabilidad por aquí, una dato más por allí y nos decimos ¡No merece la pena una clase solo por esto! Pero se termina perdiendo claridad y no queda más remedio, hay que dividirla
- Suele ser bueno identificar datos y métodos muy relacionados o subconjuntos de datos que cambian a la vez o son muy dependientes entre sí. Suele ser bueno preguntarse ¿qué pasaría si elimino este dato? ¿o este método? ¿Qué otros datos y/o métodos estarían de más?
- Muchas veces hay atributos que no siempre se usan o no se usan para cualquiera de los objetos de la clase

30

30

Extraer clase: Mecánica (pág. 182)

- Decide cómo dividir las responsabilidades de la clase: Qué tienen que hacer y qué tiene que conocer
- Crea una nueva clase secundaria para expresar las responsabilidades divididas
- Si las responsabilidades de la clase padre original ya no coinciden con su nombre, cambia el nombre del padre
- Crea una instancia de la clase secundaria al construir el padre y agrega un enlace de padre a hijo
- Usa la función de movimiento (198) para mover métodos al nuevo hijo. Comienza con métodos de nivel inferior (los que se llaman en lugar de llamar). Prueba después de cada movimiento
- Revisa las interfaces de ambas clases, elimina los métodos innecesarios, cambia los nombres para adaptarse mejor a las nuevas circunstancias

31

31

Extraer clase (pág. 182)

```
class Person {
  get officeAreaCode() {return this._officeAreaCode;}
  get officeNumber()   {return this._officeNumber;}
```



```
class Person {
  get officeAreaCode() {return this._telephoneNumber.areaCode;}
  get officeNumber()   {return this._telephoneNumber.number;}
}
class TelephoneNumber {
  get areaCode() {return this._areaCode;}
  get number()   {return this._number;}
}
```

32

32

Extraer superclase: Motivación (pág. 375)

- Si observo que hay dos clases que hacen cosas similares puedo aprovechar el mecanismo de la herencia para agrupar sus similitudes en una superclase
- En ocasiones la herencia se presenta durante el desarrollo, en la evolución de un programa
- Obviamente esta técnica es una alternativa a la extracción de clase
- Existe la posibilidad complementaria de reemplazar la clase con subdelegado

33

33

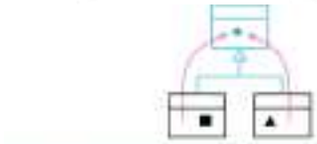
Extraer superclase: Mecánica (pág. 375)

- Crear una superclase vacía
- Crea las subclases procedentes de las clases originales
- Haz pruebas
- Mueve los elementos necesarios a la superclase
- Examina los métodos de las subclases para ver si hay partes comunes que se puedan extraer
- Modifica las llamadas a las clases originales para adaptarlas al nuevo interfaz
- Clientes de las clases originales: Considere ajustarlos para usar la interfaz de superclase

34

34

Extraer superclase (pág. 375)



```
class Department {
  get totalAnnualCost() {...}
  get name() {...}
  get headCount() {...}
}

class Employee {
  get annualCost() {...}
  get name() {...}
  get id() {...}
}
```



```
class Party {
  get name() {...}
  get annualCost() {...}
}

class Department extends Party {
  get annualCost() {...}
  get headCount() {...}
}

class Employee extends Party {
  get annualCost() {...}
  get id() {...}
}
```

35

35

Reemplazar código de tipo con subclases: Motivación (pág. 362)

- Para presentar diferentes tipos de algo similar podemos utilizar enumerados, por ejemplo
- En algunas situaciones no es suficiente con lo anterior y es necesario crear subclases, lo que me permite utilizar el polimorfismo si es necesario reemplazar lógica condicional o cuando tengo funciones que invocan un comportamiento diferente despendiendo del valor del código de tipo
- En otros casos tengo campos o métodos que solo son válidos para valores particulares de un código de tipo, como una cuota de ventas que solo se aplica al código de tipo "vendedor".

36

36

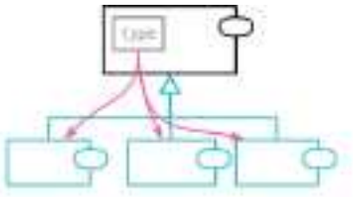
Reemplazar código de tipo con subclases: Mecánica (pág. 362)

- Encapsula el campo de código de tipo
- Elige un valor de código de tipo y crea una subclase para ese código de tipo. Reemplaza el 'getter' de código de tipo para devolver el valor de código de tipo literal
- Crea la lógica del selector para asignar desde el parámetro de código de tipo a la nueva subclase
- Ten cuidado con los constructores y haz pruebas
- Repite lo anterior para cada valor de código y no olvides probar cada vez

37

37

Reemplazar código de tipo con subclases (pág. 362)




```
function createEmployee(name, type) {
  return new Employee(name, type);
}
```

```
function createEmployee(name, type) {
  switch (type) {
    case "engineer": return new Engineer(name);
    case "salesman": return new Salesman(name);
    case "manager":  return new Manager (name);
  }
```

38

38

Introducir aserción: Motivación (pág. 302)

- Permiten que un supuesto se convierta en algo explícito
- Una aserción es una declaración condicional que se supone que siempre es cierta
- Deben escribirse de modo que el programa funcione igual si se quitan todas
- A veces se usan para detectar errores y/o depurar

39

39

Introducir aserción: Mecánica (pág. 302)

- Cuando se observe que se supone que una condición es verdadera, introduce una aserción para establecerla
- Dado que las aserciones no deberían afectar la ejecución de un sistema, agregar una siempre preserva el comportamiento

40

40

Introducir aserción (pág. 302)

```
if (this.discountRate)
  base = base - (this.discountRate * base);
```



```
assert(this.discountRate >= 0);
if (this.discountRate)
  base = base - (this.discountRate * base);
```

41