Refactoring Techniques

A focus on methods

Composing methods

Gran parte del refactoring è dedicato a comporre correttamente i metodi. Nella maggior parte dei casi, i metodi eccessivamente lunghi sono la fonte di molti dei problemi che potremmo riscontrare nella codebase. Le variazioni del codice all'interno di questi metodi nascondono la logica di esecuzione e rendono il metodo estremamente difficile da comprendere, e persino più difficile da modificare.

Le tecniche di refactoring in questo gruppo semplificano i metodi, rimuovono la duplicazione del codice e pongono le basi per future migliorie.

Extract Method

```
void printOwing() {

  printBanner();

  // Print details.

  System.out.println("name: " + name);

  System.out.println("amount: " +
getOutstanding());

  }
```

```
 void printOwing() {
  printBanner();
  printDetails(getOutstanding());
}

void printDetails(double outstanding) {
  System.out.println("name: " + name);
  System.out.println("amount: " +
outstanding);
}
```

Problema: c’è una parte di codice che può essere raggruppata

Benefici: codice più leggibile, meno duplicazione, isolamento di parti di codice indipendenti

Inline Method

```
void printOwing() {

  class PizzaDelivery {
    // ...
    int getRating() {
      return moreThanFiveLateDeliveries()
  ? 2 : 1;
    }
    boolean moreThanFiveLateDeliveries()
  {
      return numberOfLateDeliveries > 5;
    }
  }
```

```
class PizzaDelivery {
  // ...
  int getRating() {
    return numberOfLateDeliveries > 5 ? 2 :
1;
  }
}
```

Problema: la logica dentro un metodo è più esplicativa della chiamata al metodo stesso

Benefici: codice più leggibile riducendo il cluttering di metodi inutili

Extract Variable

```
renderBanner(): void {

  if ((platform.toUpperCase().indexOf("MAC") >
-1) &&

       (browser.toUpperCase().indexOf("IE") > -
1) &&

        wasInitialized() && resize > 0 )

  {

    // do something

  }

  }
```

```
renderBanner(): void {
  const isMacOs =
platform.toUpperCase().indexOf("MAC") > -1;
  const isIE =
browser.toUpperCase().indexOf("IE") > -1;
  const wasResized = resize > 0;

  if (isMacOs && isIE && wasInitialized() &&
wasResized) {
    // do something
  }
}
```

Problema: espressioni difficili di leggere

Benefici: codice più leggibile

DRAWBACKS!

Inline Temp

```
hasDiscount(order: Order): boolean {

  let basePrice: number =
order.basePrice();

  return basePrice > 1000;

  }
```

```
hasDiscount(order: Order): boolean {
  return order.basePrice() > 1000;
}
```

Problema: c’è una variabile a cui viene assegnato il valore di un calcolo semplice e null’altro

Benefici: codice più leggibile

Replace Temp With Query

```
calculateTotal(): number {
  let basePrice = quantity * itemPrice;
  if (basePrice > 1000) {
    return basePrice * 0.95;
  }
  else {
    return basePrice * 0.98;
  }
 }
```

```
calculateTotal(): number {
  if (basePrice() > 1000) {
    return basePrice() * 0.95;
  }
  else {
    return basePrice() * 0.98;
  }
}
basePrice(): number {
  return quantity * itemPrice;
}
```

Problema: viene salvato il risultato di un calcolo all'interno di una variabile affinché sia utilizzato successivamente

Benefici: codice più leggibile, meno codice (potenzialmente)

DRAWBACKS!

Split Temporary Variable

```
let temp = 2 * (height + width);

console.log(temp);

temp = height * width;

  console.log(temp);
```

```
const perimeter = 2 * (height + width);
console.log(perimeter);
const area = height * width;
console.log(area);
```

Problema: c’è una variabile il cui valore viene aggiornato durante l’esecuzione del metodo per tenere traccia di valori intermedi

Benefici: SRP, codice più leggibile, pone le basi per applicare un Extract Method successivamente

Remove Assignments to parameters

```
discount(inputVal: number, quantity:
number): number {

  if (quantity > 50) {

    inputVal -= 2;

  }

  // ...

 }
```

```
discount(inputVal: number, quantity: number):
number {
  let result = inputVal;
  if (quantity > 50) {
    result -= 2;
  }
  // ...
}
```

Problema: simile a Split Temporary Variable

Benefici: SRP, codice più leggibile e facili da mantenere

Replace method to method object

```
class Order {
  // ...
  price(): number {
    let primaryBasePrice;
    let secondaryBasePrice;
    let tertiaryBasePrice;
    // Perform long computation.
  }
}
```

Problema: ci sono metodi talmente lunghi e le cui variabili sono talmente legate l'una all'altra che diventa difficile applicare l'Extract Method

```
class Order {
  // ...
  price(): number {
    return new
PriceCalculator(this).compute();
  }
}

class PriceCalculator {
  private _primaryBasePrice: number;
  private _secondaryBasePrice: number;
  private _tertiaryBasePrice: number;

  constructor(order: Order) {
    // Copy relevant information from the
    // order object.
  }

  compute(): number {
    // Perform long computation.
  }
}
```

Substitute Algorithm

```
foundPerson(people: string[]): string{
  for (let person of people) {
    if (person.equals("Don")){
      return "Don";
    }
    if (person.equals("John")){
      return "John";
    }
    if (person.equals("Kent")){
      return "Kent";
    }
  }
  return "";
  }
```

```
foundPerson(people: string[]): string{
  let candidates = ["Don", "John", "Kent"];
  for (let person of people) {
    if (candidates.includes(person)) {
      return person;
    }
  }
  return "";
}
```

Problema: rimpiazzare un algoritmo attualmente implementato

Talvolta il refactoring graduale non sempre è la soluzione più efficiente, magari in seguito a un drastico cambiamento nei requirements del software

Simplifying Method Calls

Queste tecniche rendono le chiamate dei metodi più semplici e più comprensibili. Questo, a sua volta, semplifica le interazioni tra le classi.

Rename Method - Add Param - Remove Param

Rename Method - il nome del metodo non è esplicativo

getsnm() –> getSecondName()

Add Param - il metodo non possiede tutti i dati di cui necessita

getContact() –> getContact(date)

Remove Param - il metodo possiede degi parametri che non vengono mai utilizzati

getContact(date) –> getContact()

Separate Query From modifier

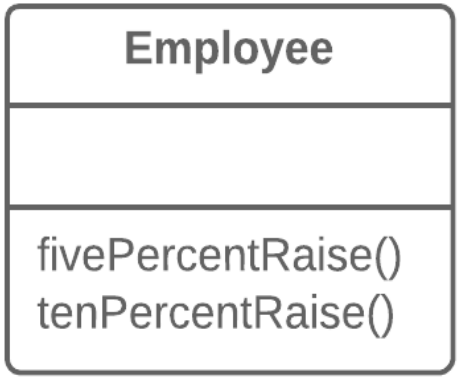
```
ottieniIlTotaleEPreparaPerNuovaTransazione(
)
```

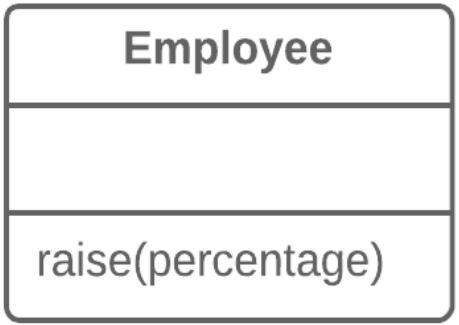
```
ottieniIlTotale()
preparaPerNuovaTransazione()
```

Problema: una funzione ti restituisce un valore ma modifica anche lo stato della classe

Benefici: *Command and Query Responsibility Segregation*, possibilità di chiamare la query ovunque sia necessario senza paura di modificare lo stato

Parametrize Methods





Problema: metodi simili con stessa logica

Drawbacks: rischio di esagerare, creando una logica tropppo complessa dove sarebbero bastati semplici metodi separati

Replace parameter with explicit methods

```
setValue(name: string, value: number): void {
  if (name.equals("height")) {
    height = value;
    return;
  }
  if (name.equals("width")) {
    width = value;
    return;
  }

  }
```

```
setHeight(arg: number): void {
  height = arg;
}
setWidth(arg: number): number {
  width = arg;
  }
```

Problema: un metodo è diviso in più parti che vengono eseguite a seconda del valore del parametro

Benefits: leggibilità

`startEngine()` vs `setValue("engineEnabled", true)`.

Preserve Whole Object

```
let low = daysTempRange.getLow();
let high = daysTempRange.getHigh();
let withinPlan = plan.withinRange(low,
high);
```

```
let withinPlan =
plan.withinRange(daysTempRange);
```

Problema: diversi parametri di un oggetto vengono passati come parametri a un metodo

Benefits: leggibilità

Drawbacks: il metodo potrebbe diventare meno flessibile

Replace parameters with Method Call

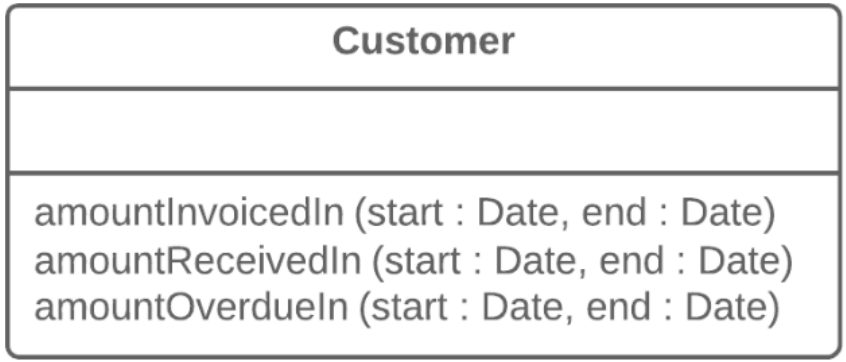
```
let basePrice = quantity * itemPrice;
const seasonDiscount =
this.getSeasonalDiscount();
const fees = this.getFees();
  const finalPrice =
  discountedPrice(basePrice,
  seasonDiscount, fees);
```

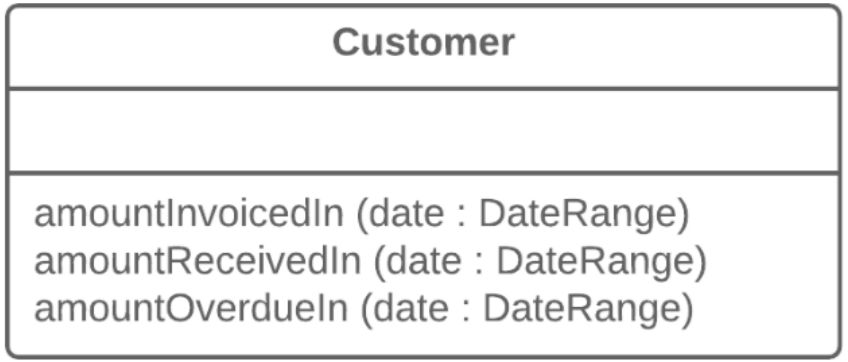
```
let basePrice = quantity * itemPrice;
let finalPrice = discountedPrice(basePrice);
```

Problema: ottenere dei dati per poi passarli come parametri, quando il metodo potrebbe ottenerli in autonomia

Benefits: liberarsi di params e variabili inutili

Introduce parameter object





Problema: gruppi di parametri ripetuti

Benefits: riduce code duplication, migliora leggibilità

Drawbacks: rischio di creazione di **DataClass**

Hide Method

Problema: un metodo non viene utilizzato esternamente alla classe

Soluzione: rendere il metodo privato o protected

Benefits: rende l’evoluzione del codice più facile
Rendendo i metodi privati tali, garantisce una comprensione più rapida di cosa rappresenta l’interfaccia pubblica e cosa rappresenta metodi di utilities

Replace constructor with factory method

```
class Employee {
  constructor(type: number) {
    this.type = type;
  }
  // ...
  }
```

```
class Employee {
  static create(type: number): Employee {
    let employee = new Employee(type);
    // Do some heavy lifting.
    return employee;
  }
  // ...
  }
```

Problema: il costruttore è troppo complesso e contiene molta logica che va oltre il semplice assegnare valori ai fields interni alla classe

Benefits: un factory method può avere un nome più chiaro ed esplicativo della logica che contiene; può avere qualsiasi tipo di return data, al contrario di un costruttore che ritorna sempre una nuova istanza della classe

Replace error code with exception

```
withdraw(amount: number): number {
  if (amount > _balance) {
    return -1;
  }
  else {
    balance -= amount;
    return 0;
  }
 }
```

```
withdraw(amount: number): void {
  if (amount > _balance) {
    throw new Error();
  }
  balance -= amount;
 }
```

Problema: il metodo ritorna un valore che dovrebbe rappresentare il fatto che qualcosa è andato storto durante l'esecuzione

Benefits: meno condizionali, consente una migliore gestione dell'errore

Replace exception with tests

```
getValueForPeriod(periodNumber: number):
number {
  try {
    return values[periodNumber];
  } catch (ArrayIndexOutOfBoundsException e) {
    return 0;
  }
 }
```

```
getValueForPeriod(periodNumber: number):
number {
  if (periodNumber >= values.length) {
    return 0;
  }
  return values[periodNumber];
 }
```

Problema: vengono utilizzate exceptions quando un semplice edge case avrebbe gestito più rapidamente e facilmente la cosa

Benefits: evita il throw di un errore laddove non strettamente necessario