

AO Refactoring

- Allo scopo di avere delle versioni (più) modulari di sistemi OO è utile determinare se all'interno di questi vi sono delle funzionalità crosscutting
 - Tali funzionalità crosscutting possono essere estratte ed incapsulate in opportuni aspetti
- Aspect-Oriented (AO) refactoring è il processo che riorganizza un dato codice preservandone il comportamento per isolare elementi crosscutting
- Passi da compiere
 - Scrivere un set di test appropriato per verificare che il refactoring non introduca differenze di comportamento
 - Extract method: liberarsi del codice duplicato creando un metodo apposito (vale per OO refactoring)
 - Extract method call: rimuovere le chiamate di metodo duplicate ed incapsularle in un aspetto
 - Costruire un pointcut per catturare i metodi dove avveniva la chiamata
 - Costruire un advice che implementa la chiamata

Ing. E. Tramontana - AO Refactoring - 6-Dic-06 1

```
public class Account {  
    ...  
    public void withdraw(int amount) {  
        if (checkPermission() && (totDeposit > amount)) {  
            totDeposit -= amount;  
            if (totDeposit <= 20) state = FROZEN;  
        }  
    }  
  
    public void credit(int amount) {  
        if (checkPermission() && (amount > 0)) totDeposit += amount;  
    }  
  
    public void setDeposit(int amount) {  
        if (checkPermission()) totDeposit = amount;  
    }  
  
    public void setSaving() {  
        if (checkPermission()) state = SAVING;  
    }  
  
    public boolean checkPermission() {  
        if ((state != FROZEN) && (state != SAVING)) return true;  
        return false;  
    }  
}
```

Ing. E. Tramontana - AO Refactoring - 6-Dic-06 3

AO Refactoring

- AO refactoring è utile per concern che sono crosscutting rispetto a tante classi
- AO refactoring è utile anche per i concern che sono crosscutting rispetto ai metodi di una singola classe
 - Es. per check pre-condizioni, inizializzazione, etc.
 - L'aspetto ha effetto solo su pochi metodi di una classe
 - I pointcut `within()` e `withincode()` sono spesso utili
 - Coupling: l'aspetto può aver bisogno di accedere ad attributi della classe
 - Ovvero, potremmo non ottenere un lasco accoppiamento tra aspetto e classe
 - Il codice dell'aspetto può essere inserito nello stesso file della classe oppure potrebbe essere un aspetto nested

Ing. E. Tramontana - AO Refactoring - 6-Dic-06 2

```
public class Account {  
    ...  
    public void withdraw(int amount) {  
        if (checkPermission() && (totDeposit > amount)) {  
            totDeposit -= amount;  
            if (totDeposit <= 20) state = FROZEN;  
        }  
    }  
  
    public void credit(int amount) {  
        if (checkPermission() && (amount > 0)) totDeposit += amount;  
    }  
  
    public void setDeposit(int amount) {  
        if (checkPermission()) totDeposit = amount;  
    }  
  
    public void setSaving() {  
        if (checkPermission()) state = SAVING;  
    }  
  
    public boolean checkPermission() {  
        if ((state != FROZEN) && (state != SAVING)) return true;  
        return false;  
    }  
}
```

Ing. E. Tramontana - AO Refactoring - 6-Dic-06 4

AO Refactoring

- Passo 1
 - Introdurre un aspetto static di refactoring nested nella stessa classe
- Passo 2
 - Definire un pointcut che cattura i join point che necessitano refactoring

```
pointcut permission(Account acc) :
    (execution(public void Account.withdraw(int)) ||
     execution(public void Account.credit(int)) ||
     execution(public void Account.setDeposit(int)) ||
     execution(public void Account.setSaving())) &&
    this(acc);
```

Ing. E. Tramontana - AO Refactoring - 6-Dic-06 5

```
public class Account { // senza l'elemento crosscutting
```

```
    ...
    public void withdraw(int amount) {
        if (totDeposit > amount) {
            totDeposit -= amount;
            if (totDeposit <= 20) state = FROZEN;
        }
    }

    public void credit(int amount) {
        if (amount > 0) totDeposit += amount;
    }

    public void setDeposit(int amount) {
        totDeposit = amount;
    }

    public void setSaving() {
        state = SAVING;
    }

    public boolean checkPermission() {
        if ((state != FROZEN) && (state != SAVING)) return true;
        return false;
    }
}
```

Ing. E. Tramontana - AO Refactoring - 6-Dic-06 7

AO Refactoring

- Passo 3
 - Creare un advice di tipo appropriato per il pointcut
- Ambienti per lo sviluppo (come Eclipse) aiutano a capire quali metodi sono catturati in modo corretto
- Passo 4
 - Spostare il codice delle chiamate dalla classe all'advice
- Passo 5 (opzionale)
 - Introdurre un warning per segnalare chiamate ancora interne alla classe

```
declare warning :
    call(public boolean Account.checkPermission()) &&
    !within(CheckAccountPermission) :
    "do not call checkPermission outside CheckAccountPermission";
```

Ing. E. Tramontana - AO Refactoring - 6-Dic-06 6

AO Refactoring

- Passo 6 (opzionale)
 - Ridefinire il pointcut per renderlo più corto e più significativo
 - Potrebbe servire per catturare metodi successivamente inseriti o per continuare a catturare i metodi anche dopo aver modificato nomi, parametri, etc.
- Passo 7 (opzionale)
 - Ri-modificare l'aspetto per poterlo usare anche per altre classi
 - È bene eseguire questo passo solo quando si hanno già altre le classi su cui usarlo

```
pointcut permission(Account acc) :
    execution(public void Account.*(..)) &&
    within(Account) &&
    this(acc);
```

Ing. E. Tramontana - AO Refactoring - 6-Dic-06 8

Aspetto CheckAccount

```
static aspect CheckAccountPermission {  
    pointcut permission(Account acc) :  
        execution(public void Account.*(..))    &&  
        within(Account)                        &&  
        this(acc);  
  
    void around(Account acc) : permission(acc) {  
        if (acc.checkPermission()) proceed(acc);  
    }  
  
    declare warning :  
        call(public boolean Account.checkPermission()) &&  
        !within(CheckAccountPermission) :  
            "do not call checkPermission outside CheckAccountPermission";  
}
```