

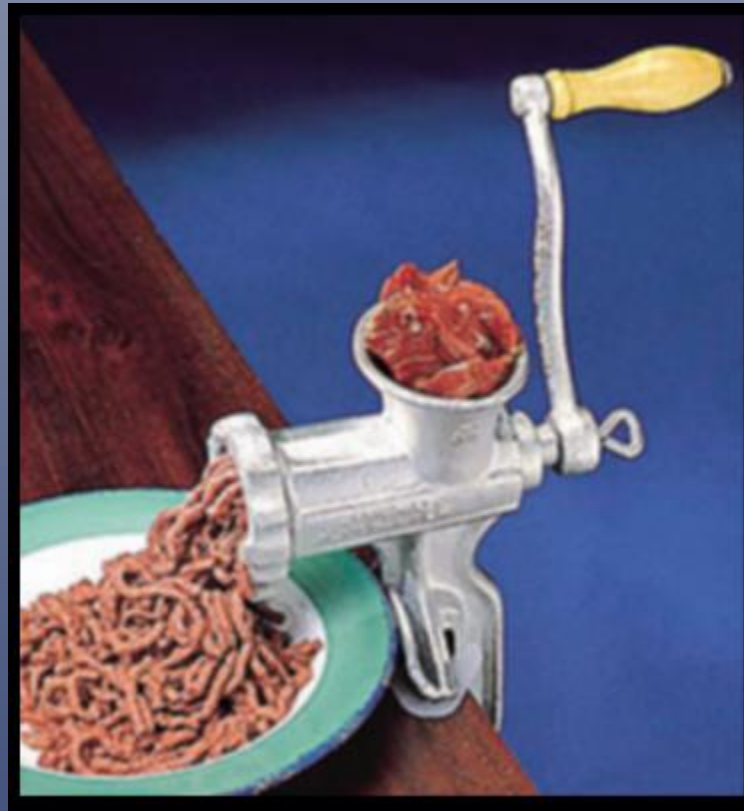
Modelação e Padrões de Desenho

Capítulo 6 Herança e Classes Abstractas (*Refactoring*)

Fernando Miguel Carvalho
DEETC

Instituto Superior de Engenharia de Lisboa,
Centro de Cálculo
mcarvalho@cc.isel.ipl.pt

Refactoring



Desenho de componentes genéricos

Componentes genéricos ⇔ reutilizáveis

- Componentes programáticos (normalmente classes ou *packages*);
- Extensíveis;
- Adaptáveis;
- Reutilizados em diferentes contextos sem alteração do código fonte.

Técnicas para criação de componentes genéricos

- *Refactoring* e generalização (Exemplo: padrão *Iterator*)

Refactoring

Refactoring

➔ Técnica para modificar o código fonte sem modificar o comportamento externo.

Exemplo de técnica usadas:

- Mudar o nome de classes, métodos e campos;
- Extrair métodos, interfaces e classes;
- *Pull up* e *push down* de métodos;
- ...

Usado para melhorar

- A eficiência, a clareza, a extensibilidade, a consistência, ...

Faz parte do ciclo de desenvolvimento de software em metodologias rápidas.

Os programadores alternam entre:

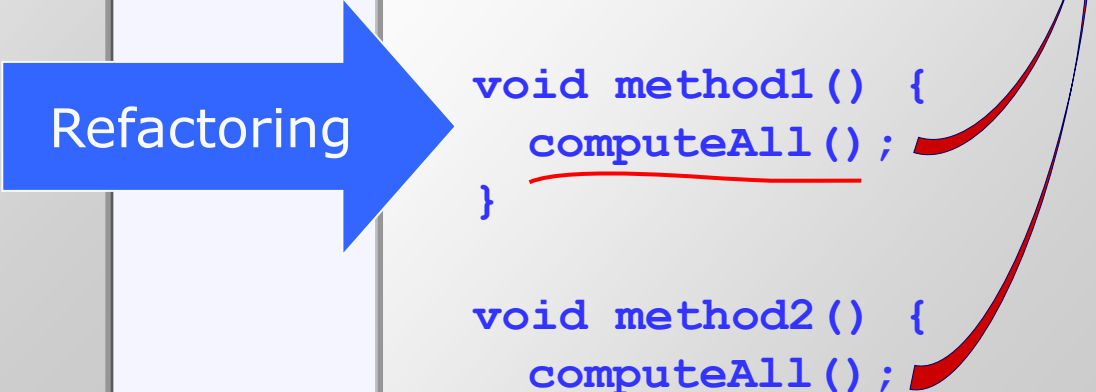
- adicionar novos testes e funcionalidades, e
- o *refactoring* de código para melhorar a consistência e a clareza.

Refactoring... Na mesma classe

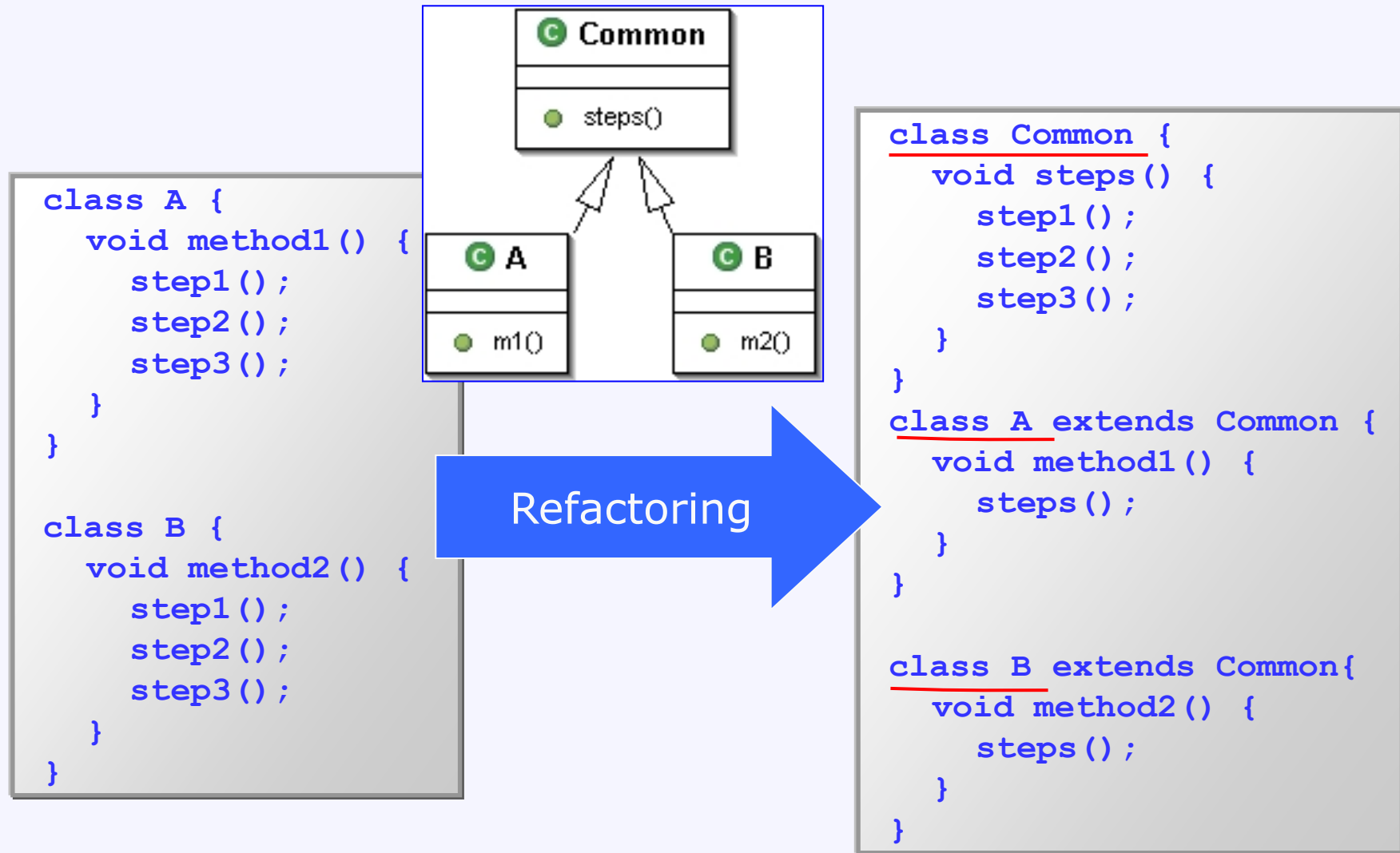
```
class A {  
    void method1() {  
        step1();  
        step2();  
        step3();  
    }  
  
    void method2() {  
        step1();  
        step2();  
        step3();  
    }  
}
```

Refactoring

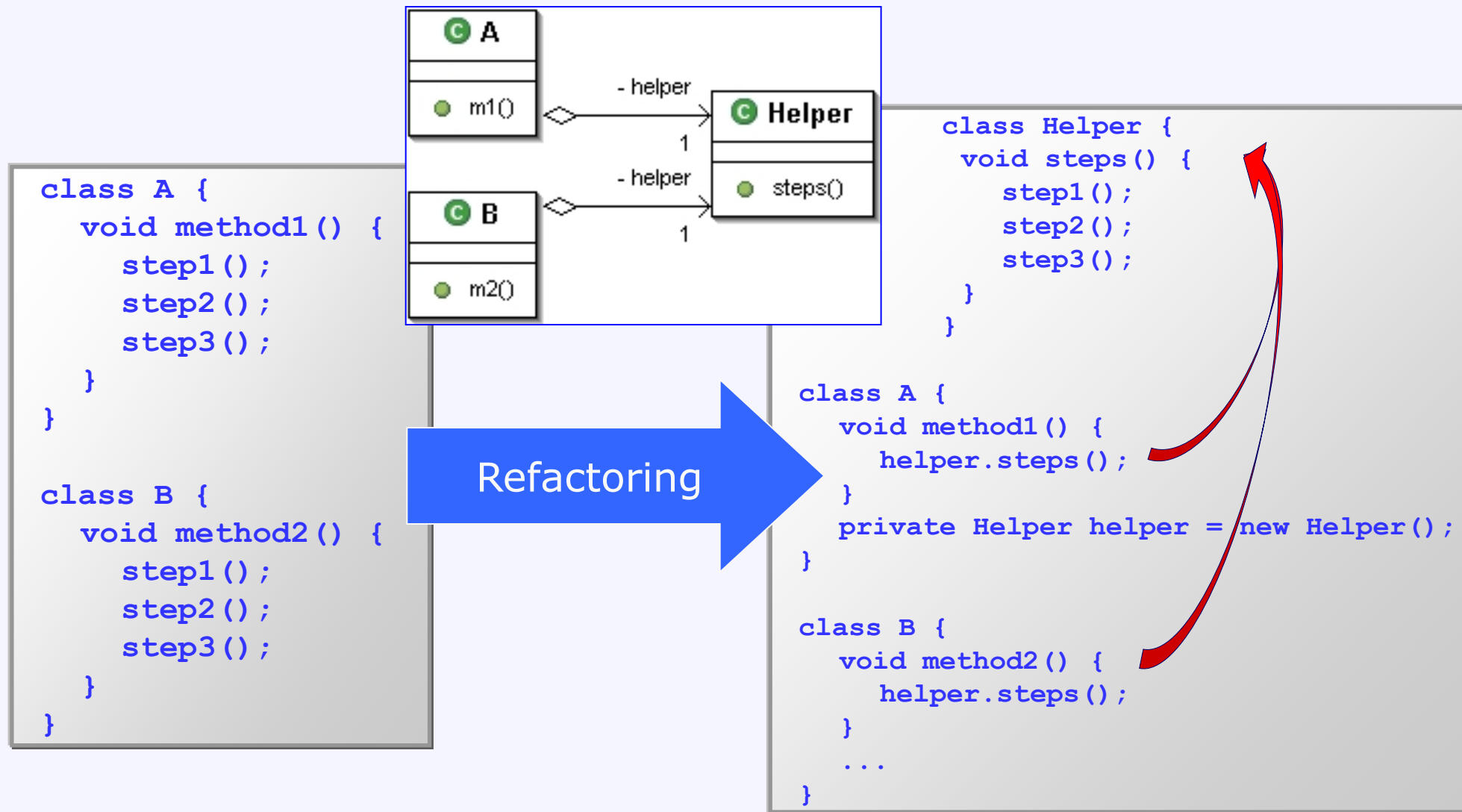
```
class RefactoredA {  
    void computeAll() {  
        step1();  
        step2();  
        step3();  
    }  
  
    void method1() {  
        computeAll();  
    }  
  
    void method2() {  
        computeAll();  
    }  
}
```



Refactoring... Em classe diferentes – Herança



Refactoring... Em classe diferentes – Delegação



Refactoring...

Identificação de segmentos de código idênticos.

Consiste na execução das seguintes tarefas:

1. Identificação de segmentos de código que implementam a mesma lógica (muitas vezes com código repetido) em diversos locais;
2. Captar essa lógica num componente genérico definido uma única vez
(na mesma classe, num classe base ou numa associada);
3. Reestruturar o programa de modo a que todas as ocorrências do código repetido seja substituído por chamadas ao componente genérico.

ATENÇÃO:

O *Refactoring* só deve ser aplicado a código que replica a mesma lógica.

→ Nem todo o código de aspecto idêntico implementa a mesma lógica!!!

Padrão *Template Method*



Refactoring... Código semelhante intercalado

```
class A {  
    void method(...) {  
        <common code segment1>  
        <specific code A>  
        <common code segment2>  
    }  
}  
  
class B {  
    void method(...) {  
        <common code segment1>  
        <specific code B>  
        <common code segment2>  
    }  
}
```

Refactoring

```
class Common {  
    void comonCode1() {  
        <common code segment1>  
    }  
    void comonCode2() {  
        <common code segment2>  
    }  
}  
  
class A extends Common {  
    void method(...) {  
        comonCode1();  
        <specific code A>  
        comonCode2();  
    }  
}  
  
class B extends Common{  
    void method(...) {  
        comonCode1();  
        <specific code B>  
        comonCode2();  
    }  
}
```

Quebra da lógica do fluxo, não garantindo a ordem da chamada aos métodos comuns.

Refactoring... Com um método *placeholder* de código

```
class A {  
    void method(...) {  
        <common code segment1>  
        <specific code A>  
        <common code segment2>  
    }  
}  
  
class B {  
    void method(...) {  
        <common code segment1>  
        <specific code B>  
        <common code segment2>  
    }  
}
```

Refactoring

```
class Common {  
    void comonCode1() {  
        <common code segment1>  
        contextSpecificCode();  
        <common code segment2>  
    }  
    void contextSpecificCode(){...}  
}  
  
class A extends Common {  
    void contextSpecificCode(...) {  
        <specific code A>  
    }  
}  
  
class B extends Common{  
    void contextSpecificCode(...) {  
        <specific code B>  
    }  
}
```

Exemplo

Javax.swing.JComponent:

- Grande parte do trabalho de desenho de um componente *swing* está definido no método `paint(Graphics g);`
- No entanto, os componentes implementados pelos programadores (*custom components*) não devem redefinir este método para adaptar o seu desenho.
- Tal como descrito na *framework swing* este método delega o trabalho de desenho em três métodos `protected: paintComponent, paintBorder, e paintChildren.`

➔ A má utilização e redefinição directa do método `paint`, pode ter consequências inesperadas.

Característica:

- ➔ O método `paintComponent` funciona como um *placeholder* onde a classe derivada coloca o código específico de desenho do conteúdo do componente.

Padrão *Template Method* - Participantes

Nome do Participante	Descrição
GenericClass (JComponent)	<ul style="list-style-type: none">Define os métodos abstractos (<i>hook</i> ou <i>primitive</i> ou <i>placeholder</i>) que as subclasses redefinem para implementar as partes variáveis de um algoritmo;implementa o método template que define o esqueleto do algoritmo recorrendo aos métodos abstractos
ConcreteClass (JButton, JLabel, etc)	Implementa os métodos abstractos com as partes <u>variáveis</u> do algoritmo definido no método template

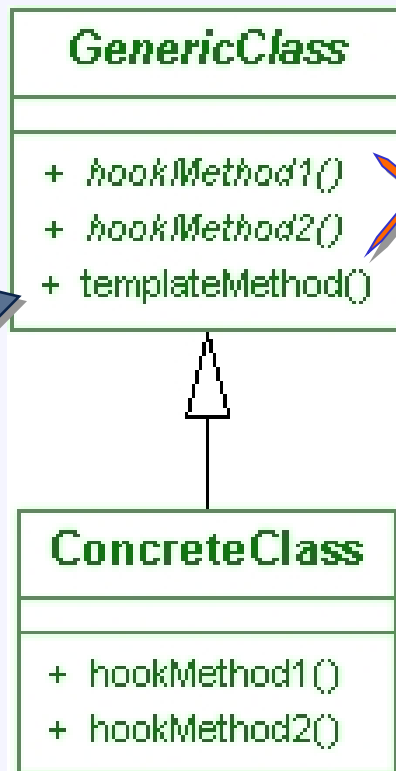
Padrão *Template Method*

A classe `JComponent` segue o padrão *template method* na forma como deve ser especificado o desenho do seu conteúdo.

Tem intercalado no seu código a chamada aos métodos gancho:

```
...  
hookMethod1()  
...  
hookMethod2()  
...
```

Exemplo:
método `paint`



Métodos *placeholder* sem implementação também são designados por métodos gancho (*hook*).
Exemplo:
`paintComponent`

Padrão *Template Method*

Característica	Descrição
Nome	<i>Template Method</i>
Categoria	Comportamento - Classe
Objectivo	Definir o esqueleto de um algoritmo num método, permitindo que as subclasses redefinam alguns dos passos.
Aplicabilidade	<ul style="list-style-type: none">• Para implementar a parte invariante do algoritmo uma só vez e permitir que as subclasses implementem o comportamento variável• Em <i>refactoring</i> para colocar o comportamento comum na superclasse e evitando o código repetido nas subclasses.
Nome alternativo	