Linguagens e Ambientes de Programação (2018/2019)

Teórica 22 (29/mai/2019)

A extensibilidade como ideia orientadora da escrita de programas orientados pelos objetos.

O uso de testes explícitos de tipo compromete a extensibilidade dos programas. A falta de modularidade compromete a extensibilidade dos programas.

Fatorização sim, mas na medida certa.

Extensibilidade e abstração

Um **sistema extensível** é um sistema que se pode fazer crescer, sem que se tenha de se alterar o que já foi escrito.

A extensibilidade obtém-se pela via da **abstração**:

- Código que seja escrito com base em conceitos abstratos consegue lidar com as entidades iniciais descritas no enunciado do problema, mas também (e aqui é que surge a extensibilidade) com entidades a criar no futuro. Claro que isto só funciona se as entidades futuras se enquadrarem nas abstrações inicialmente consideradas.

Apesar de não ser obrigatório, muitas vezes queremos escrever software extensível. As principais razões são as seguintes:

- Os programas ficam organizados em torno de ideias gerais e claras.
- Os programas ficam mais fáceis de atualizar.
- O tempo de vida útil dos programas aumenta.
- Os programas ficam mais fiáveis.
- Poupa-se tempo e dinheiro.
- Os programas ficam mais elegantes e tornam-se fonte de prazer estético (pelo menos para quem os escreve).

Extensibilidade através da fatorização

A fatorização das classes é um **requisito** para obter extensibilidade. A fatorização faz parte do processo de desenho que ajuda a identificar as abstrações naturais dos programas.

Por vezes, basta fatorizar um sistema em torno de abstrações naturais, para ele ficar automaticamente extensível. Isso acontece quando não há iteração entre objetos, ou seja quando todas as operações disponíveis numa classe se aplicam ao objeto **this**, sem envolver outros objetos. Esta é a situação que apareceu nos exemplos finais da aula teórica 19 (exemplo das classes Point1, Point2, Point3 e exemplo das expressões algébrica). Também apareceu nos exercícios da aula prática 10 (aqueles exercícios sobre sucessões matemáticas).

Mas quando as operações envolvem o objeto **this** e outros objetos então a fatorização, só por si, **não é suficiente** para obter extensibilidade. Provavelmente, esta será a primeira vez que você encontra este problema a ser discutido com detalhe.

A evitar: Testes explícitos de tipo

Existe uma questão que, de forma dramática, dá origem a **código não extensível**. Trata-se do uso de **testes para determinar o tipo dum objeto**. Esse teste costuma ser feito de forma direta, usando `instanceof`, mas por vezes também é feito de forma indireta, testando algum atributo do objeto, por exemplo o nome ou a cor.

Tal código nunca pode ser extensível, pois a sua lógica está comprometida com os tipos concretos existentes: ou seja, esse código não conseguirá lidar com novos tipos a criar no futuro. Nessa situação, para estender a lógica a novos tipos, seria necessário reescrever o código para tratar mais casos concretos, o que significa que o código não seria extensível.

A evitar: Quebra de modularidade

Uma classe deve tratar dos seus objetos e não se preocupar com as tarefas de outros objetos. A tarefas dos outros objetos são para implementar nas respetivas classes.

Por outras palavras: cada objeto tratar de si e não se preocupar em fazer as tarefas dos outros objetos.

Técnicas para evitar os testes explícitos de tipo

Geralmente, nas funções que, para além do **this**, recebem outro objeto como argumento, surge a tentação de testar diretamente o tipo do argumento. Isso tem de ser evitado a todo o custo, se tivermos como objetivo a escrita de código extensível.

Vejamos algumas técnicas que permitem evitar os testes explícitos de tipo.

Técnica do envio de mensagem

Em vez de testar diretamente o tipo dum objeto, podemos enviar-lhe uma mensagem a perguntar alguma coisa. Tal código já é extensível pois funciona com quaisquer objetos que suportem uma dada função. Repare, funciona mesmo com objetos de tipos a criar futuramente.

Por exemplo, num jogo baseado numa matriz bidimensional, em que vários monstros perseguem um herói, o que é que um monstro deverá fazer quando se cruza com outra personagem? Imagine que se trata duma função `meets(vizinho)` da classe Monstro.

- **Versão errada** (visão dum mundo fechado) -- O monstro determina, usando a expressão `vizinho instanceof Hero`, se o vizinho é um herói. Se for um herói, então o monstro almoça-o. Se não for um herói, então não faz nada. Este código não é nada extensível pois a sua lógica está dependente das classes concretas iniciais.

- **Versão correta** (visão dum mundo aberto) -- O monstro envia ao vizinho a mensagem `vizinho.comestível()` e depois atua em conformidade com a resposta. Este código já é extensível, pois funciona com qualquer personagem, mesmo com uma personagem a criar futuramente, que saiba responder à pergunta `comestível()`.

Em conclusão, conseguimos escrever código extensível **introduzindo o conceito abstrato** de *comestível*.

Relativamente a detalhes de implementação, tipicamente define-se o método `comestível` na raiz da hierarquia de classes retornando o valor `false`. Desta forma, por omissão, os personagens não são comestíveis. Os personagens que forem comestíveis redefinem o método para retornar `true`.

Técnica dos níveis

Suponha que no jogo existem muitas classes diferentes de personagens, e que entre essas classes de personagens se estabelecem complexas regras de alimentação, de tipo *cadeia alimentar*. Neste caso, uma simples função booleana *comestível* não chega para capturar tal riqueza de relações.

Neste caso, convém associar um *nível alimentar* a cada tipo de personagem e estabelecer a seguinte regra: uma personagem pode comer outra só no caso do nível alimentar da primeira ser superior ao da segunda. Concretamente, um objeto pode comer o seu vizinho se:

```
this.nivelAlimentar() > vizinho.nivelAlimentar()
```

Métodos binários

Chama-se método binário a um método com um argumento, em que se espera que o valor do argumento tenha exatamente o mesmo tipo concreto de `this`. [Paper sobre métodos binários].

Por exemplo, para efeitos de reprodução, um objeto pode precisar de saber se um outro objeto, seu vizinho, é do mesmo tipo. Como fazer isso de forma geral, sem ter de referir o tipo concreto?

Faz-se assim em JavaScript:

```
this.constructor == vizinho.constructor
```

Em Java:

```
getClass() == vizinho.getClass()
```

Em C++:

```
#include <typeinfo>

typeid(*this) == typeid(*vizinho)
```

Exercício sobre métodos binários

Observe o método `equals` das classes `Point1`, `Point2`, `Point3` da aula teórica 19. Esse método não é binário pois permite comparar pontos de diferentes tipos: compara os campos em comum de `this` e `that` e ignora os campos que não sejam comuns.

Agora imagine que você pretende tornar binário o método `equals`, ou seja fazer com que ele só dê `true` no caso de `this` e `that` serem do mesmo tipo concreto e dos campos correspondentes serem iguais. Como fazer?

Tente resolver o problema modificando apenas o método `equals` na classe `Point1`.

Exceção: criação dos dados iniciais

Imagine um jogo em que os personagens iniciais são gerados dinamicamente e colocados em posições aleatórias da matriz bidimensional. Geralmente este código não pode ser extensível porque os personagens são concretos. Se mais tarde resolvermos mudar os personagens iniciais, será necessário voltar ao código de inicialização e rescrevê-lo.

Na inicialização de certos programas, por vezes abre-se uma exceção no requisito da extensibilidade. Quem tem bastante experiência de programação já não pensa duas vezes e está disposto a aceitar alguma falta de extensibilidade no código de inicialização.

Fatorização sim, mas na medida certa

Quando se fatoriza o código, é preciso cuidado para não exagerar a ponto de começar a trocar herança por generalidade. Ao trocar herança por generalidade perdemos extensibilidade, além de que o código começa a ficar complicado e confuso.

Repare que, no limite, a essência que qualquer hierarquia de classes pode ser capturada numa classe única, sem nunca se chegar a usar herança. Essa classe fica muito geral e muito complexa, cheia de variáveis que servem para representar as diferentes propriedades de diversos tipos de entidades ao mesmo tempo. As funções também ficam muito complexas e cheias de ifs porque têm de lidar com muitos casos. No limite, até é possível criar uma única classe, muito geral, complexa e confuso, para representar ao mesmo tempo o Herói e os Monstros, sem nunca se chegar a usar herança.

Esse código não é extensível. Para incorporar um novo tipo de entidade, seria preciso alterar o código para passar a tratar mais um caso, o que prova que não é extensível.

#80