Fabiana Piesigilli - 3135607

Marcos Farah Nagato - 3121053

MAC 413 – Tópicos Avançados de Programação Orientada a Objetos

Professor Fábio Kon

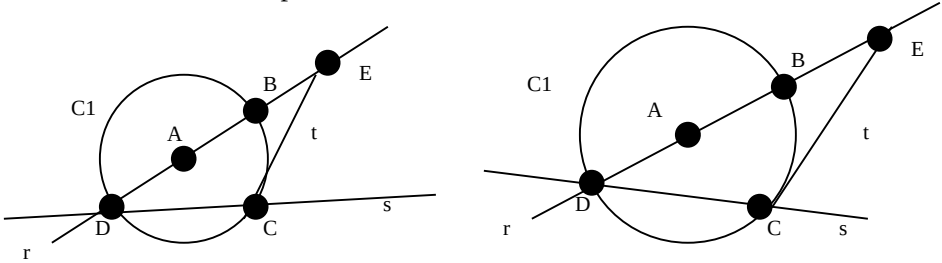
Padrão Reação em Cadeia

1. Objetivo

Diversos tipos de software possuem entidades ou objetos que possuem relações de dependência entre si. Deste modo, quando uma entidade é modificada, todas as entidades que dependem da primeira direta ou indiretamente precisam ser atualizadas. O problema em questão é fazer estas atualizações da melhor maneira possível quando as entidades são do mesmo tipo.

2. Motivação

Num *software* de geometria dinâmica, podem ser realizadas quaisquer construções geométricas a partir de entes elementares tais como pontos, retas, semi-retas, segmentos ou circunferências. Por exemplo: uma reta é definida a partir de dois pontos; uma reta paralela é definida a partir de um ponto e uma reta, etc. Diremos que se um objeto geométrico A é um dos que definem um objeto geométrico B, então B depende de A.
Quando um objeto geométrico qualquer é movido pelo usuário, todos os objetos geométricos que dependem dele devem ser atualizados, para que sejam preservadas as relações e propriedades originais da construção. Por exemplo: dada uma circunferência definida por dois pontos, sendo um deles o centro, e o outro um ponto sobre a própria circunferência, quando o segundo ponto é afastado do primeiro, a circunferência deve automaticamente aumentar de tamanho. Atualizá-la, neste caso, é fácil, mas como fazer quando há um conjunto maior de objetos dependentes entre si, como na figura abaixo, onde não apenas a circunferência aumenta de raio, mas também os segmentos DC, CE e BE têm seu comprimento aumentado?



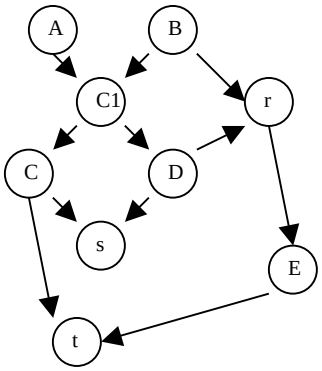
A solução proposta por este padrão consiste na representação do problema em um grafo. Neste grafo os vértices correspondem aos objetos e os arcos são as relações de dependência. Então podem ser usados diversos algoritmos de teoria dos grafos para as atualizações.

3. Aplicabilidade

Quando é preciso atualizar entidades numa cadeia de objetos de mesmo tipo, dependentes entre si .

4. Estrutura

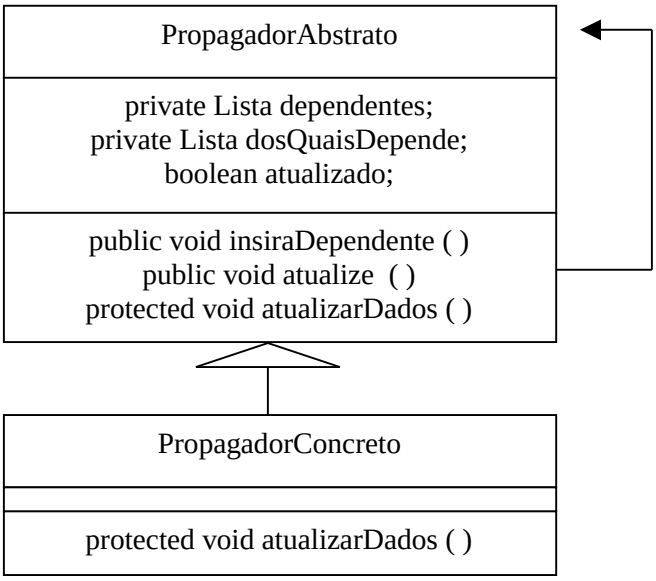
A idéia é que cada objeto tenha uma referência para todos os outros que dependam diretamente dele, de maneira que quando ele for modificado, todos os seus dependentes sejam direta ou indiretamente atualizados. Tomemos como exemplo a figura anterior. Primeiro, é traçada a circunferência C1 a partir dos pontos A e B. Se algum desses pontos são movidos, a circunferência deve ser "avisada", logo, ela será incluída tanto na lista de objetos dependentes de A como de B. Em seguida, são marcados sobre C1 os pontos C e D, que devem ser incluídos na lista da circunferência. Então, são traçadas as retas r, a partir de B e D, e s, a partir de C e D. Finalmente, o ponto E é marcado sobre r, e a reta t é traçada a partir de C e E. Neste ponto teremos um grafo, o qual é mostrado abaixo:



Note que a seta saindo de X para Y indica que Y depende de X.
É usado então algum algoritmo de teoria dos grafos para fazer a "reação em cadeia" de atualizações. Alguns algoritmos serão mostrados na parte de implementação.

5. Participantes

- `PropagadorAbstrato`: classe abstrata que possui uma lista de objetos dependentes; um método abstrato `atualizarDados` que serve para atualizar seus próprios dados; e finalmente, um método `atualize`, que chama `atualize` nos dependentes e `atualizarDados` em si própria.
- `PropagadorConcreto`: subclasse de `PropagadorAbstrato`. Implementa o método `atualizarDados`. Corresponde às classes que implementam os objetos geométricos propriamente ditos (retas, pontos, etc.).



6. Colaborações

As colaborações se dão entre as sub-classes de `PropagadorAbstrato`, através do método atualize.

7. Conseqüências

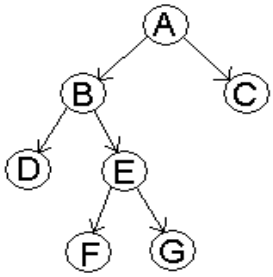
A vantagem do padrão Reação em Cadeia é encapsular atualizações complexas e fazê-las eficientemente (desde que usado o algoritmo c) por usar a melhor ordem possível para isso. Entretanto, o algoritmo de atualizações é, de certa forma, complicado. Além disso, apesar de necessário, é ruim ter-se que restaturar o estado dos objetos para `atualizado = false` toda vez que um processo de atualização termina.

8. Implementação

Nesta seção veremos os algoritmos mais comuns em grafos para realizar a tarefa de atualizações. É importante ressaltar que os grafos são sempre acíclicos no problema que estamos tratando, ou seja, não há nenhum caminho no grafo cujas extremidades correspondem ao mesmo vértice. Se existissem grafos cíclicos aqui, haveria um objeto que pertenceria à sua própria definição.

a) Busca em profundidade:

Consiste em visitar os vértices do grafo indo “sempre o mais longe possível”. Um exemplo explica melhor. Suponhamos que a se um vértice tem mais de um dependente, os dependentes são visitados em ordem alfabética. Assim, o seguinte grafo teria seus vértices atualizados na ordem (A, B, D, E, F, G, C) se o vértice A for atualizado:



Este algoritmo é facilmente implementado usando-se recursão. A função busca_em_profundidade é chamada no vértice raiz.

```
funcao busca_em_profundidade (vértice v) {
   atualiza v
   para cada vértice w dependente de v busca_em_profundidade (w)
}
```

b) Busca em largura:

Ao contrário da busca em profundidade, a busca em largura visita os vértices de um grafo "sempre o mais perto possível" em relação ao ponto de partida. No mesmo grafo acima, a ordem de atualização seria (A, B, C, D, E, F, G).

Este algoritmo é facilmente implementado usando-se uma fila. No começo há apenas um vértice na fila.

```
funcao busca_em_largura () {
   enquanto a fila não está vazia {
         seja  v  o primeiro vértice da fila
         atualiza v
         para cada vértice w dependente de v
                                    insira w no final da fila
        retire v da fila
   }
}
```

c) Busca de Hideo*:

O problema dos dois algoritmos vistos acima é que são ineficientes, pois vértices podem ser atualizados antes de todos os objetos dos quais dependem terem sido eles próprios atualizados, o que resulta em serem, desnecessariamente, atualizados mais de uma vez. Por exemplo no grafo acima se D dependesse de E, ambas buscas teriam que atualizar D duas vezes.
A saída é utilizar um algoritmo que atualize todos os vértices que serão afetados uma só vez, atualizando-os numa ordem correta (pode haver mais de uma). No caso de D depender de E, o ideal seria atualizar E antes de D.

O algoritmo consiste em manter um subconjunto S dos vértices do grafo. Inicialmente ele é composto por todos os vértices que podem ser alcançados a partir do vértice inicial, ou seja, o território deste vértice. Este subconjunto S inicial corresponde a todos os vértices que devem ser atualizados. Para obtê-lo, uma opção é ter um campo booleano em cada vértice que indica que

*Ricardo Hideo Sahara – autor do algoritmo

ele pertence a S. Daí, Aplica-se uma busca em largura ou profundidade a partir do vértice inicial marcando cada vértice alcançado.

Deve ser mantido também um conjunto P de todos os vértices que não dependem de nenhum outro vértice em S. Inicialmente só o vértice que desencadeia a reação está em P. A cada iteração do algoritmo, qualquer vértice de P pode ser atualizado.

A iteração do algoritmo corresponde então a:

```
escolher um vértice v qualquer de P
atualizar v
remover v de P e de S
inserir em P vértices que dependem de v e não dependem de
                                mais nenhum outro vértice em S
```

Quando o conjunto S estiver vazio, o algoritmo termina. A última linha da iteração pode ser implementada com cada filho v' de v no grafo consultando todos os objetos em S dos quais depende, e verificando se estes já foram atualizados. Em caso afirmativo, v' pode então ser alterado. Veja um exemplo de implementação deste algoritmo no código de amostra.

9. Código de Amostra

No exemplo a seguir, a classe `PropagadorAbstrato` é chamada de `ObjetoGeometrico`, `atualizarDados` corresponde a `recalculeCoordenadas`, e a lista ligada é atravessada através de um *iterator*. O método `recalculeCoordenadas` calcula a nova posição do objeto geométrico em questão a partir dos objetos modificados dos quais ele depende. Exemplos de `PropagadorConcreto` poderiam ser classes como Ponto, Reta, etc.

```
public abstract class ObjetoGeometrico {

     boolean atualizado = false;
     private Lista dosQuaisDepende;
     private Lista dependentes;

     public void atualize () {

          if (atualizado) return;

          Enumeration e = dosQuaisDepende.elements();
          while (e.hasMoreElements) {
              if (!((ObjetoGeometrico)e.nextElement()).atualizado)
                return;
          }

          atualizado = true;
          recalculeCoordenadas();


          e = dependentes.elements();
```

```
            while (e.hasMoreElements) {
                ((ObjetoGeometrico)e.nextElement()).atualize();
            }
        }

        public void insiraDependente (ObjetoGeometrico o) {
            dependentes.insira (o);
        }
}
```

É importante que o método que desencadeou todo o processo – por exemplo, um que move um ponto – torne a atribuir `false` a todos os objetos, para que o algorimo funcione posteriormente.

10. Usos Conhecidos

O padrão Reação em Cadeia é utilizado para propagar mudanças na posição de objetos geométricos na área de desenho no *software* de Geometria Dinâmica iGeom (http://www.matematica.br).

11. Padrões Relacionados

Iterator

Usado para percorrer a lista de dependentes.

Chain of Responsibility

A diferença entre o *Chain of Responsability* e o Reação em Cadeia é que no primeiro, os eventos podem ser passados adiante numa cadeia linear ou processados, interrompendo-na, enquanto no segundo, os eventos ou modificações são processados e passados adiante num grafo.

Observer

O padrão *Observer* é parecido com o Reação em Cadeia. A principal diferença entre eles é que no Observer a cadeia de atualizações é apenas uma árvore de dois níveis. Num grafo, os objetos da classe *Subject* seriam pais, e os da *Observer*, filhos.

Outra diferença é que o Reação em Cadeia simplifica o *design* pois pressupõe-se que as entidades possuem o mesmo tipo, coisa que não acontece no Observer. Além disso, no Reação em Cadeia, uma vez que um dependente é inscrito para receber os eventos de outro objeto, o primeiro só pode sair da lista de dependentes do segundo ao deixar de existir.