Padrão: Versionamento

Nomes: Flávia Rainone N.USP: 3286141
Stefan Neusatz Guilhen N.USP: 3286332

Intenção

Prover um mecanismo simples para gerenciamento de versões de objetos, de modo a esconder dos clientes os detalhes do processo de versionamento.

Motivação

Muitos sistemas lidam com processos formados por diversas etapas auto−suficientes, que podem ser modificadas independentemente das demais etapas do processo. Em muitos casos, é importante manter um histórico das modificações realizadas.

Por exemplo, em uma companhia especializada em medicina diagnóstica são realizados milhares de exames diariamente. Cada exame é um processo constituído de várias etapas: requisição, coleta de material, execução do teste, análise do resultado e emissão de laudo. Com o passar do tempo, novos métodos para coletar material, executar um teste e analisar um resultado são desenvolvidos e passam a substituir os métodos usados anteriormente.

A solução é guardar todas as informações necessárias à análise organizadas em um histórico. Em sistemas orientados a objetos, podemos ter um objeto responsável por guardar os valores A, B, X e Y da análise, e gerenciar um histórico.

As normas existentes de qualidade total exigem que mudanças como essas sejam devidamente documentadas e recuperáveis. Quando for necessário rever o resultado de um laudo, o médico deve ter acesso às informações corretas. Um exemplo é saber qual foi o método de análise aplicado na hora em que o exame foi feito. Suponha que a análise consiste em adicionar uma quantidade X de um reagente A a uma quantidade Y da solução B, e aplicar a mistura ao material que deve ser examinado. O resultado deverá ser interpretado de acordo com os valores X e Y. As substâncias A e B também podem ser substituídas por composições melhores, à medida que a ciência evolui. Para determinarmos a faixa de valores considerados normais, os valores A, B, X e Y são determinantes. Se um novo método for adotado e os dados do método antigo forem perdidos, é provável que o médico irá obter informações erradas a respeito de um exame realizado anteriormente à mudança.

Generalizando, podemos ter um objeto responsável por cada etapa do processo (requisição, coleta de material, teste, análise do resultado e emissão do laudo). Esse objeto guardaria em seus atributos todo o tipo de informação

necessária para a execução desta etapa. Se alguma modificação for feita nesses atributos, uma nova versão do objeto é criada, e a versão anterior é adicionada a um histórico. Tudo isso deve ser feito de modo que o cliente não tenha acesso à complexidade do gerenciamento de versões. Ao mesmo tempo, o cliente deve ter acesso aos dados de uma certa etapa de acordo com a data de interesse.

Voltando ao exemplo citado, vamos considerar a classe MetodoAnalise, cuja finalidade é fornecer os valores A, B, X e Y corretos, de acordo com determinada data. Cada alteração em um desses novos valores deve, então, criar uma nova versão, sem que a versão anterior seja perdida. Para isso, criaremos o objeto MetodoAnaliseVersao, que efetivamente guarda os valores A, B, X e Y. Cada versão deve ter uma data de início e uma de término. A classe Análise provê uma interface que permite que esses valores sejam lidos e alterados. A cada alteração, um novo objeto MetodoAnaliseVersao é criado, com a data atual como início, e a versão utilizada até o momento é armazenada em uma lista (histórico), com a data atual como término (note que a versão atual não precisa ter uma data de término). Um objeto de versão não deve permitir a alteração de seus valores, já que esses valores não podem ser perdidos. Note que, se houver dados relacionados ao método de análise que não precisem de versionamento, esses devem ser atributos de MétodoAnalise. Ou seja, à classe MetodoAnaliseVersao só cabe armazenar dados que precisam ser recuperados para o entendimento de um processo.

O propósito deste padrão é fornecer um meio transparente para o cliente criar novas versões dos objetos, sem, no entanto, abrir mão da simplicidade. O cliente interage com uma classe que se responsabiliza pela criação e manutenção de suas próprias versões. No exemplo exibido acima, essa classe seria MetodoAnalise, e as versões seriam representadas por objetos da classe MetodoAnaliseVersao. Essa classe contém somente os atributos versionáveis (aqueles que quando modificados forçam a criação de uma nova versão para acomodar as mudanças), definindo uma interface apenas de acesso a esses atributos (métodos `get`). A classe MetodoAnalise, por sua vez, contém apenas os atributos ditos não versionáveis (atributos que podem ser modificados sem a necessidade de versionamento). Sua função é fornecer uma interface para alterar todos os atributos (versionáveis e não−versionáveis) e criar uma nova versão quando atributos versionáveis são modificados.

O cliente tem acesso a todos os atributos, *como se eles estivessem na mesma classe*, fazendo com que o efeito de alterações não seja perceptível ao cliente e livrando−o da responsabilidade de lidar com o versionamento de objetos.

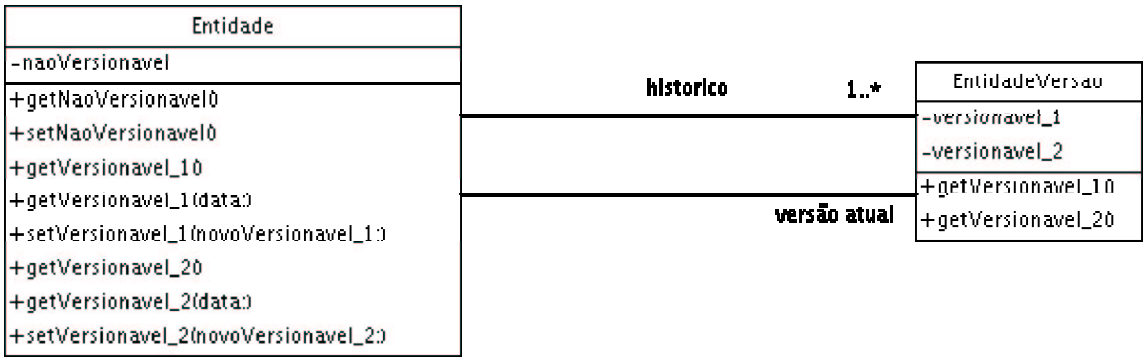
Aplicabilidade

O padrão Versionamento deve ser usado quando:

- Houver necessidade de manter um histórico das mudanças realizadas em alguma parte do sistema, seja por necessidade de documentação ou backup de dados importantes.

- Um processo a ser realizado for composto por diversas etapas auto−suficientes, que possam mudar ao longo do tempo (nesse caso, é necessário conhecer as etapas que foram utilizadas em determinado processo para analisá−lo).

Estrutura



Participantes

- **Entidade:**

  − contém os atributos não versionáveis.
  − fornece uma interface para acesso e alteração dos seus atributos e dos atributos da `EntidadeVersao`, se responsabilizando pela criação de novas versões quando necessário.
  − fornece um meio de encontrar os atributos versionáveis de uma dada versão pela data.

- **EntidadeVersão:**

  − contém os atributos versionáveis.
  − fornece uma interface simples, somente com serviços de acesso aos atributos (métodos `get`).

Colaborações

Clientes acessam apenas a instância da `Entidade`, que representa a versão atual. O cliente pode, então, modificar os valores dos atributos da `Entidade` sem se preocupar em como as modificações irão ser tratadas pelo sistema.

A `Entidade` é responsável por administrar o versionamento, e o faz através de objetos `EntidadeVersao`, que compõem o histórico de versões.

Conseqüências

A aplicação do Versionamento possui como algumas de suas conseqüências:

1. *Encapsulamento do versionamento:* a administração de versões é encapsulada na classe `Entidade`, e não é visível ao cliente. Tudo o que o cliente precisa fazer é acessar dados de acordo com a data de interesse.

2. *Simplicidade:* diminuímos a complexidade do problema ao aplicar essa solução, já que apenas uma classe lida com as versões.

3. *Menor área de superfície:* ao invés de precisar conhecer e gerenciar as várias versões existentes, o sistema só precisa acessar a interface da `Entidade` para obter a funcionalidade necessária.

4. *Excesso de dados:* se `EntidadeVersao` possui muitos atributos, a cada alteração que é feita num atributo todos os outros são copiados para uma nova versão. Isso pode gerar excesso de dados no sistema.

Implementação

Ao implementar o padrão Versionamento, você deve levar em conta os seguintes detalhes de implementação:

1. *Tempo de vida:* as versões têm um tempo de vida, definido por uma data de início e uma data de fim. A versão atual não tem data de fim.

2. *O acesso aos dados versionáveis:* a classe pai deve fornecer um meio de encontrar um valor versionável, através de uma data. Pode ser interessante prover acesso a um valor versionável sem data para indicar o valor em vigência atual. Isso diminui a complexidade do sistema, já que

clientes que precisão sempre do valor atual não terão que fornecer a data.

3. *Acesso de valores de uma versão:* apenas métodos de leitura − `get`() − devem ser fornecidos pela `EntidadeVersao`. O intuito do padrão é evitar que os dados de uma versão sejam perdidos.

4. *Otimização de armazenamento de dados:* esse padrão pode ser utilizado juntamente com o padrão Observer. Ao adicionarmos observadores à `Entidade`, podemos saber quais deles utilizam uma determinada versão. Desse modo, ao criarmos uma nova versão e adicionarmos a atual ao histórico, é possível verificar se a versão atual foi utilizada por algum cliente. Se ela não tiver sido utilizada, ela pode ser removida sem danos, poupando espaço de armazenamento. Para mantermos a consistência no sistema, a nova versão deve ter como data de criação a data de criação da versão removida. Isso deve ser aplicado nas seguintes situações:
   - o número de observadores não for muito grande;
   - a criação de novas versões for mais freqüente que a sua utilização;
   - o espaço de armazenamento no sistema for crítico.

5. *Criação de versões futuras:* em alguns sistemas, pode ser de interesse a criação de versões que devam ser aplicadas no futuro, a partir de uma data específica. Para isso, basta fornecer uma interface na `Entidade` que permita a criação de versões futuras. Essas devem ser criadas com uma data de início de vigência específica. Tal data deve ser definida como data de término da versão atual. É necessário monitorar o tempo para que a nova versão entre em vigência na data correta. A versão antiga, como sempre, é adicionada ao histórico.

Código−Exemplo

Segue uma implementação em Java do exemplo citado na Motivação:

```
//Classe MetodoAnaliseVersao
public class MetodoAnaliseVersao{

  //Atributos versionÆveis
  private String reagenteA;
  private String reagenteB;
  private double quantidadeA;
  private double quantidadeB;

  //Data de início e fim da versao
  private GregorianCalendar dataInicio;
  private GregorionCalendar dataFim;

  //Construtor
```

```
  public MetodoAnaliseVersao(String A, String B, double valorA,
                             double valorB, GregorianCalendar inicio){

    this.reagenteA = A;
    ...
  }

  //MØtodos de acesso às datas de início e fim
  public GregorianCalendar getDataInicio()
    return this.dataInicio;
  public void setDataInicio(GregorianCalendar data)
    this.dataInicio = data;

  ...

  //Interface de acesso aos atributos versionÆveis
  //Aqui nao existem mØtodos set
  public String getReagenteA()
    return this.reagenteA;

  public String getReagenteB()
    return this.reagenteB;
  public getQuantidadeA()
  ...

}

//Classe MetodoAnalise

public class MetodoAnalise{

  //Atributo nao versionÆvel
  private String nomeDoMetodo;

  //Versoes associadas
  private Collection versoes = new List();
  private MetodoAnaliseVersao versaoAtual;

  //Construtor: recebe todos os parâmetros necessÆrios para criar
  //a primeira versao no sistema
  public MetodoAnalise(String nome, String A, String B,
                           double valorA, double ValorB){

    this.nomeDoMetodo = nome;
    this.versaoAtual = new MetodoAnaliseVersao(A, B ...);
    this.versoes.add(versaoAtual);

  }

  //MØtodos get e set dos atributos nao versionÆveis
  public String getNomeDoMetodo() return this.nomeDoMetodo;
  public void setNomeDoMetodo(String nome)
    this.nomeDoMetodo = nome;


  //MØtodos get e set dos atributos versionÆveis
  public String getReagenteA()
    return this.versaoAtual.getReagenteA();

  public String getReagenteA(GregorianCalendar data)
    return this.findVersao(data).getReagenteA();
```

```
    public void setReagenteA(String newReagente){

      GregorianCalendar data = //data atual
      MetodoAnaliseVersao nova = new MetodoAnaliseVersao(
                  newReagente, versaoAtual.getReagenteB(),
                  versaoAtual.getQuantidadeA(),
                  versaoAtual.getQuantidadeB(), data);
      this.versaoAtual.setDataFim(data);
      this.versoes.add(nova);
      versaoAtual = nova;
    }
    ...

    //Encontra a versªo pela data
    private MetodoAnaliseVersao findVersao(GregorianCalendar data)
    ...
}
```

Usos Conhecidos

Esse padrão é utilizado no sistema Motion, em desenvolvimento para a empresa Diagnósticos da América (líder no ramo de medicina diagnóstica).

Padrões Relacionados

Observer, Proxy.