corrente de um sistema de restrições. O memento contém somente aquelas variáveis de restrição que mudaram desde a última solução. Normalmente, apenas um pequeno subconjunto das variáveis do solucionador muda para cada nova solução. Esse subconjunto é suficiente para retornar o solucionador para a solução precedente; a reversão para soluções anteriores exige a restauração de mementos das soluções intermediárias. Por isso, você não pode definir mementos em qualquer ordem; QOCA depende de um mecanismo de história para reverter a soluções anteriores.

Padrões relacionados

Command (222): Comands podem usar mementos para manter estados para operações que normalmente não poderiam ser desfeitas.

Iterator (244): Mementos podem ser usados para iteração, conforme já descrito.

OBSERVER — comportamental de objetos

Intenção

Definir uma dependência um-para-muitos entre objetos, de maneira que quando um objeto muda de estado todos os seus dependentes são notificados e atualizados automaticamente.

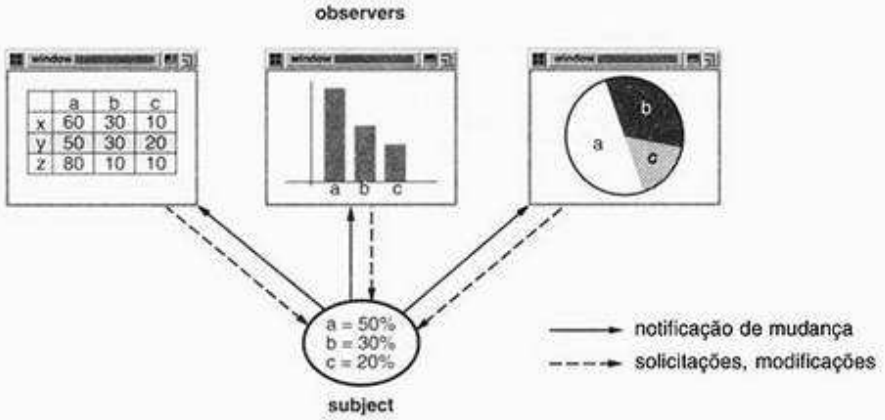
Também conhecido como

Dependents, Publish-Subscribe

Motivação

Um efeito colateral comum resultante do particionamento de um sistema em uma coleção de classes cooperantes é a necessidade de manter a consistência entre objetos relacionados. Você não deseja obter consistência tornando as classes fortemente acopladas, porque isso reduz a sua reusabilidade.

Por exemplo, muitos *toolkits* para construção de interfaces gráficas de usuário separam os aspectos de apresentação da interface do usuário dos dados da aplicação subjacente [KP88,LVC89,P+88, WGM88]. As classes que definem dados da aplicação e da apresentação podem ser reutilizadas independentemente. Elas também podem trabalhar em conjunto. Tanto um objeto planilha como um objeto gráfico de barras podem ilustrar informações do mesmo objeto de aplicação usando diferentes apresentações. A planilha e o gráfico de barras não têm conhecimento um do outro, desta forma permitindo reutilizar somente o objeto de que você necessita. Porém, elas *se comportam* como se conhecessem. Quando o usuário muda a informação na planilha, o gráfico de barras reflete as mudanças imediatamente, e vice-versa.



Esse comportamento implica que a planilha e o gráfico de barras são dependentes do objeto de dados e, portanto, deveriam ser notificados sobre qualquer mudança no seu estado. E não há razão para limitar o número de dependentes a dois objetos; pode haver um número qualquer de diferentes interfaces do usuário para os mesmos dados.

O padrão Observer descreve como estabelecer esses relacionamentos. Os objetos-chave nesse padrão são ***subject*** (assunto) e ***observer*** (observador). Um *subject* pode ter um número qualquer de observadores dependentes. Todos os observadores são notificados quando *o subject* sofre uma mudança de estado. Em resposta, cada observador inquirirá o *subject* para sincronizar o seu estado com o estado do *subject*.

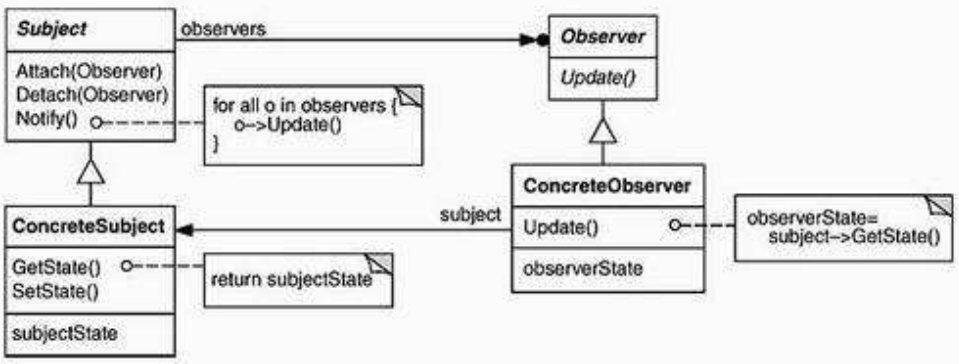
Esse tipo de interação também é conhecido como **publish-subscribe**. O *subject* é o publicador de notificações. Ele envia essas notificações sem ter que saber quem são os seus observadores. Um número qualquer de observadores pode inscrever-se para receber notificações.

Aplicabilidade

Use o padrão Observer em qualquer uma das seguintes situações:

- quando uma abstração tem dois aspectos, um dependente do outro. Encapsulando esses aspectos em objetos separados, permite-se variá-los e reutilizá-los independentemente;
- quando uma mudança em um objeto exige mudanças em outros, e você não sabe quantos objetos necessitam ser mudados;
- quando um objeto deveria ser capaz de notificar outros objetos sem fazer hipóteses, ou usar informações, sobre quem são esses objetos. Em outras palavras, você não quer que esses objetos sejam fortemente acoplados.

Estrutura



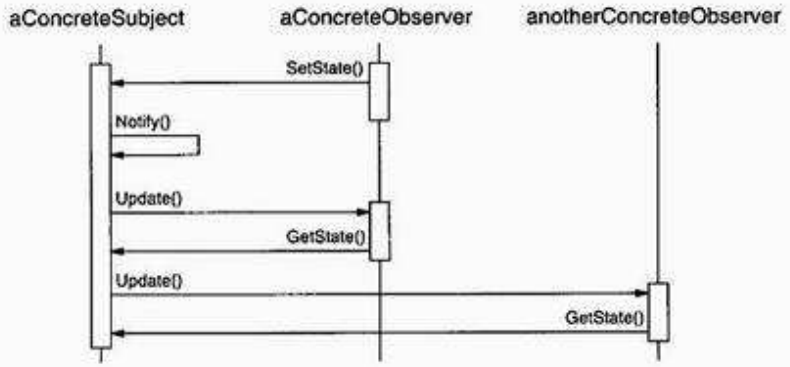
Participantes

- **Subject**
  - conhece os seus observadores. Um número qualquer de objetos Observer pode observar um subject.
  - fornece uma interface para acrescentar e remover objetos, permitindo associar e desassociar objetos observer.
- **Observer**
  - define uma interface de atualização para objetos que deveriam ser notificados sobre mudanças em um Subject.
- **ConcreteSubject**
  - armazena estados de interesse para objetos ConcreteObserver.
  - envia uma notificação para os seus observadores quando seu estado muda.
- **ConcreteObserver**
  - mantém uma referência para um objeto ConcreteSubject.
  - armazena estados que deveriam permanecer consistentes com os do Subject.
  - implementa a interface de atualização de Observer, para manter seu estado consistente com o do subject.

Colaborações

- O ConcreteSubject notifica seus observadores sempre que ocorrer uma mudança que poderia tornar inconsistente o estado deles com o seu próprio.
- Após ter sido informado de uma mudança no subject concreto, um objeto ConcreteObserver poderá consultar o subject para obter informações. O ConcreteObserver usa essa informação para reconciliar o seu estado com o do subject.

  O seguinte diagrama de interação ilustra as colaborações entre um subject e dois observadores:



Note como o objeto Observer que inicia a solicitação de mudança posterga sua atualização até que ele consiga uma notificação do subject. Notify não é sempre chamada pelo subject. Pode ser chamada por um observador ou por um outro tipo de objeto. A seção Implementação discute algumas variações comuns.

Conseqüências

O padrão Observer permite variar subjects e observadores de forma independente. Você pode reutilizar subjects sem reutilizar seus observadores e vice-versa. Ele permite acrescentar observadores sem modificar o subject ou outros observadores. Benefícios adicionais e deficiências do padrão Observer incluem o seguinte:

1. *Acoplamento abstrato entre Subject e Observer.* Tudo que o subject sabe é que ele tem uma lista de observadores, cada um seguindo a interface simples da classe abstrata Observer. O subject não conhece a classe concreta de nenhum observador. Assim, o acoplamento entre o subject e os observadores é abstrato e mínimo.
   Por não serem fortemente acoplados, Subjet e Observer podem pertencer a diferentes camadas de abstração em um sistema. Um subject de nível mais baixo pode comunicar-se com um observador de nível mais alto, desta maneira mantendo intacta as camadas do sistema. Se Subject e Observer forem agrupados, então o objeto resultante deve cobrir duas camadas (e violar a estrutura de camadas), ou ser forçado a residir em uma das camadas (o que pode comprometer a abstração da estrutura de camadas).
2. *Suporte para comunicações do tipo broadcast.* Diferentemente de uma solicitação ordinária, a notificação que um subject envia não precisa especificar seu receptor. A notificação é transmitida automaticamente para todos os objetos interessados que a subscreveram. O subject não se preocupa com quantos objetos interessados existem; sua única responsabilidade é notificar seus observadores. Isso dá a liberdade de acrescentar e remover observadores a qualquer momento. É responsabilidade do observador tratar ou ignorar uma notificação.
3. *Atualizações inesperadas.* Como um observador não tem conhecimento da presença dos outros, eles podem ser cegos para o custo global de mudança do subject. Uma operação aparentemente inócua no subject pode causar uma cascata de atualizações nos observadores e seus objetos dependentes. Além do mais, critérios de dependência que não estão bem-definidos ou mantidos normalmente conduzem a atualizações espúrias que podem ser difíceis de detectar.
   Este problema é agravado pelo fato de que o protocolo simples de atualização não fornece detalhes sobre *o que* mudou no subject. Sem protocolos adicionais para ajudar os observadores a descobrir o que mudou, eles podem ser forçados a trabalhar duro para deduzir as mudanças.

Implementação

Nesta seção são discutidos vários aspectos relacionados com a implementação do mecanismo de dependência.

1. *Mapeando subjects para os seus observadores.* A maneira mais simples para um subject manter o controle e o acompanhamento dos observadores que ele deve notificar é armazenar referências para eles explicitamente no subject. Contudo, tal armazenagem pode ser muito dispendiosa quando existem muitos subjects e poucos observadores. Uma solução é trocar espaço por tempo usando um mecanismo de pesquisa associativo (por exemplo, uma *hash table* – tabela de acesso randômico) para manter o mapeamento subject-para-observador. Assim, um subject sem observadores não tem um custo de memória para esse problema. Por outro lado, esta solução aumenta o custo do acesso aos observadores.
2. *Observando mais do que um subject.* Em algumas situações pode fazer sentido para um observador depender de mais do que um subject. Por exemplo, uma planilha pode depender de uma fonte de dados. É necessário estender a

interface de Update em tais casos para permitir ao observador saber *qual* subject está enviando a notificação. O subject pode, simplesmente, passar a si próprio como um parâmetro para a operação Update, dessa forma permitindo ao observador saber qual subject examinar.

3. *Quem dispara a atualização?* O subject e seus observadores dependem do mecanismo de notificação para permanecerem consistentes. Mas qual objeto na realidade chama Notify para disparar a atualização? Aqui estão duas opções:
   (a) ter operações de estabelecimento de estados no Subject que chame Notify após elas mudarem o estado do subject. A vantagem dessa solução é que os clientes não têm que lembrar de chamar Notify no subject. A desvantagem é que diversas operações consecutivas causarão diversas atualizações consecutivas, o que pode ser ineficiente.
   (b) tornar os clientes responsáveis por chamar Notify no momento correto. Aqui, a vantagem é que o cliente pode esperar para disparar a atualização até que seja concluída uma série de mudanças de estado, desta forma evitando atualizações intermediárias desnecessárias. A desvantagem é que os clientes têm uma responsabilidade adicional, de disparar a atualização. Isto torna a ocorrência de erros mais provável, uma vez que os clientes podem esquecer de chamar Notify.
4. *Referências ao vazio (dangling references) para subjects deletados.* A remoção de um subject não deve produzir *referências ao vazio* nos seus observadores. Uma forma de evitar referências ao vazio é fazer com que o subject notifique os seus observadores quando é deletado, de modo que possam restabelecer suas referências. Em geral, simplesmente deletar os observadores não é uma opção, porque outros objetos podem referenciá-los ou porque eles também podem estar observando outros subjects.
5. *Garantindo que o estado do Subject é autoconsistente antes da emissão da notificação.* É importante se assegurar de que o estado do Subject é autoconsistente antes de invocar Notify, porque os observadores consultam o subject sobre o seu estado corrente no curso da atualização de seus próprios estados.
   Esta regra de autoconsistência é fácil de violar não-intencionalmente quando operações nas subclasses de Subject chamam operações herdadas.
   Por exemplo, a notificação na seqüência de código seguinte é disparada quando o subject está em um estado inconsistente:

```
void MySubject::Operation (int newValue) {
    BaseClassSubject::Operation(newValue);
        // dispara a notificação

    _myInstVar += newValue;
        // atualiza o estado da subclasse (tarde demais!)
}
```

   Podemos evitar essa armadilha enviando notificações por métodos-template (Template Method, 301) em classes Subject abstratas. Defina uma operação primitiva para ser substituída pelas subclasses e torne Notify a última operação do método-template, o que garantirá que o objeto é autoconsistente quando as subclasses substituem operações de Subject.
   A propósito, é sempre uma boa idéia documentar quais operações de Subject disparam notificações.

```
void Text::Cut (TextRange r) {
     ReplaceRange(r);        // redefinido nas subclasses
     Notify();
}
```

6. *Evitando protocolos de atualização específicos dos observadores: os modelos push e pull.* A implementação do padrão Observer freqüentemente tem o subject emitindo (*broadscast*) informações adicionais sobre a modificação. O subject passa essa informação como argumento para Update. A quantidade de informação pode variar bastante.
   Num extremo, que chamaremos de **push model (modelo de empurrar informação),** o subject manda aos observadores informações detalhadas sobre a mudança, quer eles queiram ou não. No outro extremo, está o **pull model (modelo de puxar informação)**; o subject não envia nada além da menor notificação possível, e posteriormente os observadores solicitam detalhes.
   O pull model enfatiza o desconhecimento dos subjects sobre seus observadores, enquanto que o push model assume que os subjects sabem algo das necessidades de seus observadores. O push model pode tornar os observadores menos reutilizáveis porque as classes Subject fazem hipóteses sobre as classes Observer, que podem não ser sempre verdadeiras. Por outro lado, o pull model pode ser ineficiente porque as classes Observer devem verificar o que mudou sem a ajuda do Subject.
7. *Especificando explicitamente as modificações de interesse.* Você pode melhorar a eficiência do processo de atualização estendendo a interface de inscrição no subject para permitir aos observadores se registrarem somente para eventos específicos de seu interesse. Quando tal evento ocorre, o subject informa somente os observadores que se registraram. Uma maneira de suportar isso usa a noção de **aspectos** para objetos Subject. Para registrar o seu interesse em determinados eventos, os observadores são associados aos seus subjects usando

   ```
   void Subject::Attach(Observer*, Aspect& interest);
   ```

   onde `interest` especifica o evento de interesse. Em tempo de notificação, o subject fornece o aspecto mudado para os seus observadores, como um parâmetro para a operação Update. Por exemplo:

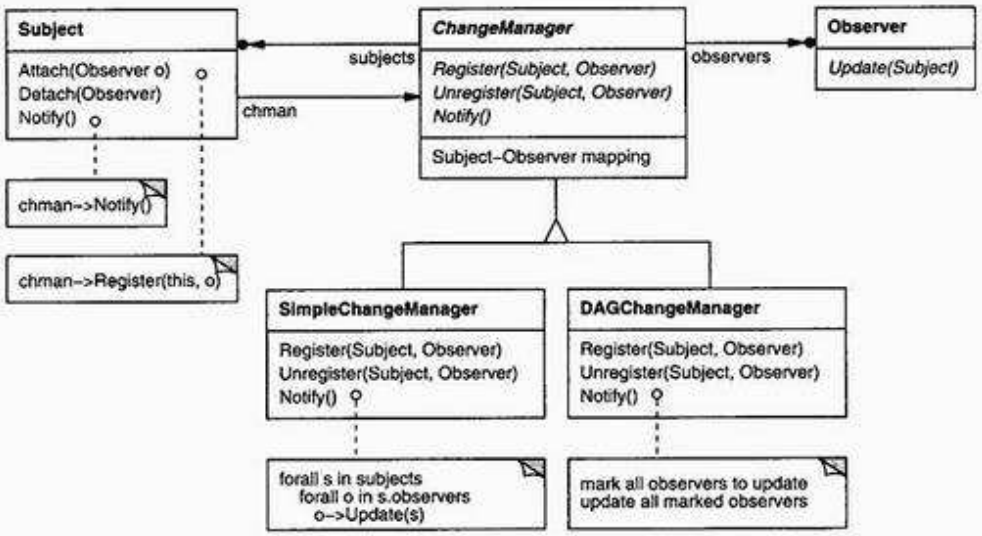
   ```
   void Observer::Update(Subject*, Aspect& interest);
   ```

8. *Encapsulando a semântica de atualizações complexas.* Quando o relacionamento de dependência entre subjects e observadores é particularmente complexo pode ser necessário um objeto que mantenha esses relacionamentos. Chamamos tal objeto de **ChangeManager** (Administrador de Mudanças). Sua finalidade é minimizar o trabalho necessário para fazer com que os observadores reflitam uma mudança no seu subject. Por exemplo, se uma operação envolve mudanças em vários subjects interdependentes, você pode ter que assegurar que seus observadores sejam notificados somente após *todos* subjects terem sido modificados para evitar de notificar os observadores mais de uma vez.

O ChangeManager tem três responsabilidades:

(a) ele mapeia um subject aos seus observadores e fornece uma interface para manter esse mapeamento. Isto elimina a necessidade dos subjects manterem referências para os seus observadores, e vice-versa;
(b) ele define uma estratégia de atualização específica;
(c) ele atualiza todos os observadores dependentes, por solicitação de um subject.

O diagrama a seguir ilustra uma implementação simples, baseada no uso de um ChangeManager, do padrão Observer. Existem dois ChangeManagers especializados. O SimpleChangeManager é ingênuo no sentido de que sempre atualiza todos os observadores de cada subject. Por outro lado, o DAGChangeManager trata grafos acíclicos direcionados de dependências entre subjects e seus observadores. Um DAGChangeManager é preferível a um SimpleChangeManager quando um observador observa mais do que um subject. Nesse caso, uma mudança em dois ou mais subjects pode causar atualizações redundantes. O DAGChangeManager assegura que o observador receba somente uma atualização. Quando múltiplas atualizações não são um problema, SimpleChangeManager funciona bem.



O ChangeManager é uma instância do padrão Mediator (257). Em geral, existe somente um ChangeManager e ele é conhecido globalmente. Aqui, o padrão Singleton (130) seria útil.

9. *Combinando as classes Subject e Observer.* As bibliotecas de classes escritas em linguagens que não têm herança múltipla (como Smalltalk) geralmente não definem classes Subject e Observer separadas, mas combinam suas interfaces em uma única classe. Isso permite definir um objeto que funciona tanto como um subject como um observe, sem usar herança múltipla. Por exemplo, em Smalltalk, as interfaces de Subject e de Observer estão definidas na classe raiz Object, tornando-as disponíveis para todas as classes.

Exemplo de código

Uma classe abstrata define a interface de Observer:

Esta implementação suporta múltiplos subjects para cada observador. O subject passado para a operação Update permite ao observador determinar qual subject mudou quando ele observa mais de um.

```
class Subject;

class Observer {
public:
    virtual ~Observer();
    virtual void Update(Subject* theChangedSubject) = 0;
protected:
    Observer();
};
```

De maneira semelhante, uma classe abstrata define a interface de Subject:

```
class Subject {
public:
    virtual ~Subject();

    virtual void Attach(Observer*);
    virtual void Detach(Observer*);
    virtual void Notify();
protected:
    Subject();
private:
    List<Observer*> *_observers;
};

void Subject::Attach (Observer* o) {
    _observers->Append(o);
}

void Subject::Detach (Observer* o) {
    _observers->Remove(o);
}

void Subject::Notify () {
    ListIterator<Observer*> i(_observers);

    for (i.First(); !i.IsDone(); i.Next()) {
        i.CurrentItem()->Update(this);
    }
}
```

`ClockTimer` é um subject concreto para armazenar e manter a hora do dia. Ele notifica seus observadores a cada segundo. `ClockTimer` fornece a interface para a recuperação de unidades de tempo individuais, como hora, minuto e segundo.

```
class ClockTimer : public Subject {
public:
    ClockTimer();

    virtual int GetHour();
    virtual int GetMinute();
    virtual int GetSecond();

    void Tick();
};
```

A operação `Tick` é chamada por um relógio interno, a intervalos regulares, para fornecer uma base de tempo precisa. `Tick` atualiza o estado interno de `ClockTimer` e chama `Notify` para informar os observadores sobre a mudança:

```
void ClockTimer::Tick () {
    // atualiza o estado interno de manutenção do tempo
    // ...
    Notify();
}
```

Agora queremos definir uma classe `DigitalClock` que exibe a hora. Ela herda a sua funcionalidade gráfica de uma classe `Widget` fornecida por um *toolkit* para interfaces de usuário. A interface de Observer é misturada com a interface de `DigitalClock` por herança de `Observer`.

```
class DigitalClock: public Widget, public Observer {
public:
    DigitalClock(ClockTimer*);
    virtual ~DigitalClock();

    virtual void Update(Subject*);
        // redefine a operação de Observer

    virtual void Draw();
        // redefine a operação de Widget
        // define como desenhar o relógio digital
private:
    ClockTimer* _subject;
};

DigitalClock::DigitalClock (ClockTimer* s) {
    _subject = s;
    _subject->Attach(this);
}

DigitalClock::~DigitalClock () {
    _subject->Detach(this);
}
```

Antes que a operação `Update` mude o mostrador do relógio, ela faz a verificação para se assegurar de que o subject notificador é o subject do relógio:

```
void DigitalClock::Update (Subject* theChangedSubject) {
    if (theChangedSubject == _subject) {
        Draw();
    }
}

void DigitalClock::Draw () {
    // obtém os novos valores do subject

    int hour = _subject->GetHour();
    int minute = _subject->GetMinute();
    // etc.

    // desenha o relógio digital
}
```

Da mesma maneira pode ser definida uma classe `AnalogClock`.

```
class AnalogClock : public Widget, public Observer {
public:
    AnalogClock(ClockTimer*);
    virtual void Update(Subject*);
    virtual void Draw();
    // ...
};
```

O código a seguir cria uma `AnalogClock` e uma `DigitalClock` que sempre mostra o mesmo tempo:

```
ClockTimer* timer = new ClockTimer;
AnalogClock* analogClock = new AnalogClock(timer);
DigitalClock* digitalClock = new DigitalClock(timer);
```

Sempre que o `timer` pulsar e notificar, os dois relógios serão atualizados e reexibirão a si próprios adequadamente.

Usos conhecidos

O primeiro e talvez mais conhecido exemplo do padrão Observer aparece no Model/View/Controller (MVC), da Smalltalk, o *framework* para a interface do usuário no ambiente Smalltalk [KP88]. A classe Model, do MVC, exerce o papel do Subject, enquanto View é a classe base para observadores. Smalltalk, ET++ [WGM88] e a biblioteca de classes THINK [Sym93b] fornecem um mecanismo geral de dependência colocando interfaces para Subject e Observer na classe-mãe para todas as outras classes do sistema.

Outros *toolkits* para interfaces de usuário que empregam esse padrão são: InterViews [LVC89], Andrew Toolkit [P+88] e Unidraw [VL90]. O InterViews define classes Observer e Observable (para subjects) explicitamente. Andrew as chama "view" e "data object", respectivamente. O Unidraw separa objetos do editor gráfico em View (para observadores) e Subject.

Padrões relacionados

Mediator (257): encapsulando a semântica de atualizações complexas, o ChangeManager atua como um mediador entre subjects e observadores.