Singleton (130): O ChangeManager pode usar o padrão Singleton para torná-lo único e globalmente acessível.

STATE — comportamental de objetos

Intenção

Permite a um objeto alterar seu comportamento quando o seu estado interno muda. O objeto parecerá ter mudado sua classe.

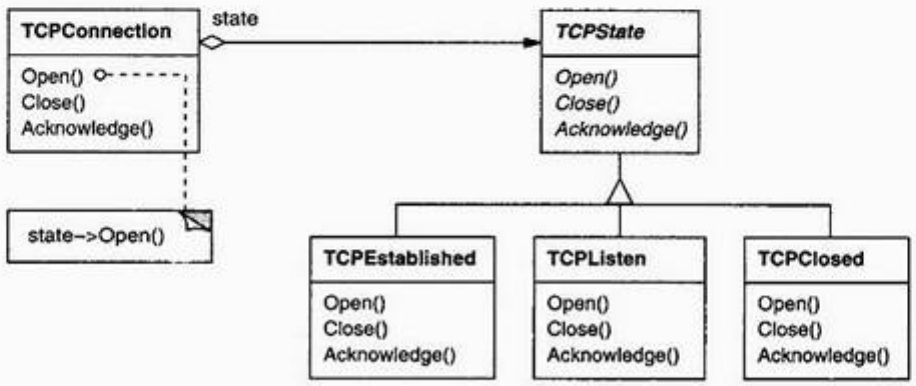
Também conhecido como

Objects for States

Motivação

Considere a classe TCPConnection que representa uma conexão numa rede de comunicações. Um objeto TCPConnection pode estar em diversos estados diferentes: Established (Estabelecida), Listening (Escutando), Closed (Fechada). Quando um objeto TCPConnection recebe solicitações de outros objetos, ele responde de maneira diferente dependendo do seu estado corrente. Por exemplo, o efeito de uma solicitação de Open (Abrir), depende de se a conexão está no seu estado Closed ou no seu estado Established. O padrão State descreve como TCPConnection pode exibir um comportamento diferente em cada estado.

A idéia chave deste padrão é introduzir uma classe abstrata chamada TCPState para representar os estados da conexão na rede. A classe TCPState declara uma interface comum para todas as classes que representam diferentes estados operacionais. As subclasses de TCPState implementam comportamentos específicos ao estado. Por exemplo, as classes TCPEstablished e TCPClosed implementam comportamento específico aos estados Established e Closed de TCPConnection.



A classe TCPConnection mantém um objeto de estado (uma instância da subclasse de TCPstate) que representa o estado corrente na conexão TCP.

Connection delega todas as solicitações específicas de estados para este objeto de estado. TCPConnection usa sua instância da subclasse de TCPState para executar operações específicas ao estado da conexão.

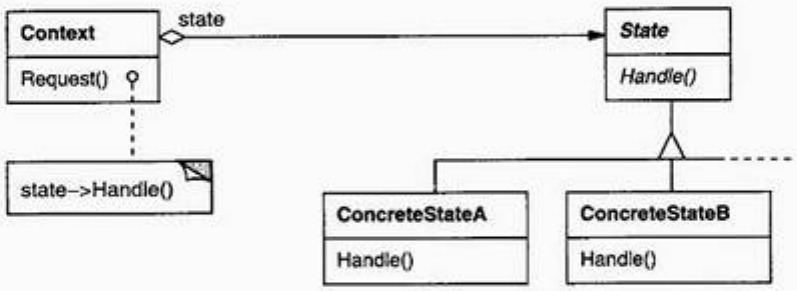
Sempre que a conexão muda de estado, o objeto TCPConnection muda o objeto de estado que ele utiliza. Por exemplo, quando a conexão passa do estado Established para o estado Closed, TCPConnection substituirá sua instância de TCPEstablished por uma instância de TCPClosed.

Aplicabilidade

Use o padrão State em um dos dois casos seguintes:

- o comportamento de um objeto depende do seu estado e ele pode mudar seu comportamento em tempo de execução, dependendo desse estado;
- operações têm comandos condicionais grandes, de várias alternativas, que dependem do estado do objeto. Esse estado é normalmente representado por uma ou mais constantes enumeradas. Freqüentemente, várias operações conterão essa mesma estrutura condicional. O padrão State coloca cada ramo do comando adicional em uma classe separada. Isto lhe permite tratar o estado do objeto como um objeto propriamente dito, que pode variar independentemente de outros objetos.

Estrutura



Participantes

- **Context** (TCPConnection)
  - define a interface de interesse para os clientes.
  - mantém uma instância de uma subclasse ConcreteState que define o estado corrente.
- **State** (TCPState)
  - define uma interface para encapsulamento associado com um determinado estado do Context.
- **ConcreteState subclasses** (TCPEstablished, TCPListen, TCPClosed)
  - cada subclasse implementa um comportamento associado com um estado do Context.

Colaborações

- O Context delega solicitações específicas de estados para o objeto corrente ConcreteState.
- Um contexto pode passar a si próprio como um argumento para o objeto State que trata a solicitação. Isso permite ao objeto State acessar o contexto, se necessário.

- Context é a interface primária para os clientes. Os clientes podem configurar um contexto com objetos State. Uma vez que o contexto está configurado, seus clientes não têm que lidar com os objetos State diretamente.
- Tanto Context quanto as subclasses de ConcreteState podem decidir qual estado sucede outro, e sob quais circunstâncias.

Conseqüências

O padrão State tem as seguintes conseqüências:

1. *Ele confina comportamento específico de estados e particiona o comportamento para estados diferentes.* O padrão State coloca todo o comportamento associado com um estado particular em um objeto. Como todo o código específico a um estado reside numa subclasse de State, novos estados e transições de estado podem ser facilmente adicionados pela definição de novas subclasses.
   Uma alternativa é usar valores de dados para definir estados internos, tendo operações de Context que verificam explicitamente os dados. Mas, em conseqüência, teríamos instruções parecidas, condicionais ou de seleção (Case) espalhadas por toda a implementação de Context. O acréscimo de um novo estado poderia exigir a mudança de várias operações, o que complicaria a manutenção.
   O padrão State evita esse problema, mas introduz um outro, porque o padrão distribui o comportamento para diversos estados entre várias subclasses de State. Isso aumenta o número de classes e é menos compacto do que ter uma única classe. Porém, tal distribuição é, na realidade, boa se existirem muitos estados, pois de outro modo necessitaríamos de grandes comandos condicionais.
   Do mesmo modo que *procedures* longas são indesejáveis, também o são comandos condicionais grandes. Eles são monolíticos e tendem a tornar o código menos explícito, o que, por sua vez torna-os difíceis de modificar e estender. O padrão State oferece uma maneira melhor para estruturar o código específico de estados. A lógica que determina as transições de estado não se localiza em comandos monolíticos `if` ou `switch`, mas é particionada entre as subclasses de State. O encapsulamento de cada transição de estado e ação associada em uma classe eleva a idéia de um estado de execução ao *status* de um objeto propriamente dito. Isso impõe uma estrutura ao código e torna a sua intenção mais clara.
2. *Ele torna explícitas as transições de estado.* Quando um objeto define o seu estado corrente unicamente em termos de valores de dados internos, suas transições de estado não têm representação explícita; elas apenas aparecem como atribuições de valores a algumas variáveis. A introdução de objetos separados, para estados diferentes, torna as transições mais explícitas.
   Os objetos State também podem proteger o Context de estados internos inconsistentes porque, da perspectiva do Context, as transições de estado são atômicas – elas acontecem pela revinculação (*rebinding*) de *uma* variável (a variável que contém o objeto State de Context), e não de várias [dCLF93].
3. *Objetos State podem ser compartilhados.* Se os objetos State não possuem variáveis de instância – ou seja, o estado que elas representam está codificado inteiramente no seu tipo – então contextos podem compartilhar um objeto State. Quando estados são compartilhados dessa maneira, eles são essenci-

almente flyweights (ver Flyweight, 187) sem estado intrínseco, somente comportamento.

Implementação

O padrão State dá origem a diversas questões de implementação:

1. *Quem define as transições de estado?* O padrão State não especifica qual participante define os critérios para transições de estado. Se os critérios são fixos, então podem ser implementados inteiramente no Context. Contudo, é geralmente mais flexível e adequado permitir que as subclasses de State especifiquem elas mesmas o seu estado sucessor e quando efetuar a transição. Isto exige o acréscimo de uma interface para Context que permite aos objetos State definirem explicitamente o estado corrente de Context.
   A forma de descentralizar a lógica de transição torna mais fácil modificar ou estender a lógica pela definição de novas subclasses State. Uma desvantagem da descentralização é que uma subclasse State terá conhecimento de pelo menos uma outra, o que introduz dependências de implementação entre subclasses.
2. *Uma alternativa baseada no uso de tabelas.* Em C++ Programming Style [Car92], Cargill descreve uma outra maneira de impor estrutura a um código orientado por estados: ele usa tabelas para mapear entradas para transições de estado. Para cada estado, uma tabela mapeia cada entrada possível para um estado sucessor. Com efeito, essa solução converte o código condicional (e, no caso de padrão State, funções virtuais) em uma pesquisa de tabela.
   A principal vantagem do uso de tabelas é sua regularidade: você pode mudar os critérios de transição através da modificação de dados em vez da modificação do código de programas. Contudo, aqui apresentamos algumas desvantagens:

   - Uma pesquisa de tabela freqüentemente é menos eficiente do que a chamada (virtual) de uma função.
   - A colocação da lógica de transição num formato uniforme, tabular, torna os critérios de transição menos explícitos e, portanto, mais difíceis de compreender.
   - É normalmente difícil acrescentar ações para acompanhar as transições de estado. A solução baseada em tabelas captura os estados e suas transições, porém, ela deve ser aumentada para executar computações arbitrárias em cada transição.

   A diferença-chave entre máquinas de estado guiadas por tabelas e o padrão State pode ser resumida desta maneira: o padrão State modela o comportamento específico de estados, enquanto que a abordagem voltada para tabelas se concentra na definição das transições de estado.
3. *Criando e destruindo objetos State.* Duas alternativas comuns de implementação que valem a pena considerar são: (1) criar objetos State apenas quando são necessários e destruí-los logo que se tornem desnecessários e (2) criá-los antecipadamente e nunca destruí-los.
   A primeira escolha é preferível quando os estados que serão atingidos não são conhecidos em tempo de execução *e* os contextos mudam de estado com pouca freqüência. Esta solução evita a criação de objetos que não serão usados, o que

é importante se os objetos State armazenarem grande quantidade de informação. A segunda solução é melhor quando as mudanças de estado ocorrem rapidamente, sendo que nesse caso você deseja evitar a destruição de estados porque serão necessários novamente em breve. Os custos de instanciação acontecem somente uma vez antes do processo, e não existe nenhum custo de destruição. Entretanto, esta solução pode ser inconveniente porque o Context deve manter referências para todos os estados que possam ser alcançados.

4. *Usando herança dinâmica.* A mudança de comportamento para uma determinada solicitação poderia ser obtida pela mudança da classe do objeto em tempo de execução, porém, isso não é possível na maioria das linguagens de programação orientadas a objetos. As exceções incluem Self [US87] e outras linguagens baseadas em delegação que oferecem tal mecanismo e, portanto, suportam diretamente o padrão State. Objetos em Self podem delegar operações para outros objetos para obter uma forma de herança dinâmica. A mudança do alvo da delegação, em tempo de execução, efetivamente muda a estrutura de herança. Esse mecanismo permite aos objetos mudarem o seu comportamento e corresponde a mudar as suas classes.

Exemplo de código

O exemplo a seguir fornece o código C++ para o exemplo de conexões TCP descrito na seção Motivação. Este exemplo é uma versão simplificada do protocolo TCP; ele não descreve o protocolo completo ou todos os estados das conexões TCP[8].

Primeiramente, definiremos a classe `TCPConnection`, a qual fornece uma interface para transmissão de dados e trata solicitações para mudanças de estado.

```
class TCPOctetStream;
class TCPState;

class TCPConnection {
public:
    TCPConnection();

    void ActiveOpen();
    void PassiveOpen();
    void Close();
    void Send();
    void Acknowledge();
    void Synchronize();

    void ProcessOctet(TCPOctetStream*);
private:
    friend class TCPState;
    void ChangeState(TCPState*);
private:
    TCPState* _state;
};
```

`TCPConnection` mantém uma instância da classe `TCPState` na variável membro `_state`. A classe `TCPState` duplica a interface de mudança de estados de `TCPConnection`. Cada operação `TCPState` recebe uma instância de `TCPConnection` como um parâmetro, permitindo a `TCPState` acessar dados de `TCPConnection` e mudar o estado da conexão.

```
class TCPState {
public:
    virtual void Transmit(TCPConnection*, TCPOctetStream*);
    virtual void ActiveOpen(TCPConnection*);
    virtual void PassiveOpen(TCPConnection*);
    virtual void Close(TCPConnection*);
    virtual void Synchronize(TCPConnection*);
    virtual void Acknowledge(TCPConnection*);
    virtual void Send(TCPConnection*);
protected:
    void ChangeState(TCPConnection*, TCPState*);
};
```

`TCPConnection` delega todas as solicitações específicas de estado para sua instância `_state` de `TCPState`. O `TCPConnection` também fornece uma operação para mudar essa variável para um novo `TCPState`. O constructor de `TCPConnection` inicia o objeto no estado `TCPClosed` (definido mais tarde).

```
TCPConnection::TCPConnection () {
    _state = TCPClosed::Instance();
}

void TCPConnection::ChangeState (TCPState* s) {
    _state = s;
}

void TCPConnection::ActiveOpen () {
    _state->ActiveOpen(this);
}

void TCPConnection::PassiveOpen () {
    _state->PassiveOpen(this);
}

void TCPConnection::Close () {
    _state->Close(this);
}

void TCPConnection::Acknowledge () {
    _state->Acknowledge(this);
}

void TCPConnection::Synchronize () {
    _state->Synchronize(this);
}
```

`TCPState` implementa o comportamento-padrão para todas as solicitações delegadas a ela. Ela pode também mudar o estado de uma `TCPConnection` com a operação `ChangeState`. `TCPState` é declarado como um *friend* de `TCPConnection` para ter acesso privilegiado a essa operação.

```
void TCPState::Transmit (TCPConnection*, TCPOctetStream*) { }
void TCPState::ActiveOpen (TCPConnection*) { }
void TCPState::PassiveOpen (TCPConnection*) { }
void TCPState::Close (TCPConnection*) { }
void TCPState::Synchronize (TCPConnection*) { }

void TCPState::ChangeState (TCPConnection* t, TCPState* s) {
    t->ChangeState(s);
}
```

As subclasses de `TCPState` implementam o comportamento específico de estado. Uma conexão TCP pode estar em muitos estados: Established, Listening, Closed, etc., e existe uma subclasse de `TCPState` para cada estado. Discutiremos três subclasses em detalhe: `TCPEstablished`, `TCPListen` e `TCPClosed`.

```
class TCPEstablished : public TCPState {
public:
    static TCPState* Instance();

    virtual void Transmit(TCPConnection*, TCPOctetStream*);
    virtual void Close(TCPConnection*);
};

class TCPListen : public TCPState {
public:
    static TCPState* Instance();

    virtual void Send(TCPConnection*);
    // ...
};

class TCPClosed : public TCPState {
public:
    static TCPState* Instance();

    virtual void ActiveOpen(TCPConnection*);
    virtual void PassiveOpen(TCPConnection*);
    // ...
};
```

As subclasses de `TCPState` não mantêm estados locais, de modo que possam ser compartilhadas e apenas uma instância de cada seja requerida. A única instância de cada subclasse `TCPState` é obtida pela operação estática `Instance`[9].

Cada subclasse `TCPState` implementa um comportamento específico do estado para cada solicitação válida no estado:

```
void TCPClosed::ActiveOpen (TCPConnection* t) {
    // envia SYN, recebe SYN, ACK, etc.

    ChangeState(t, TCPEstablished::Instance());
}

void TCPClosed::PassiveOpen (TCPConnection* t) {
    ChangeState(t, TCPListen::Instance());
}

void TCPEstablished::Close (TCPConnection* t) {
    // envia FIN, recebe ACK de FIN

    ChangeState(t, TCPListen::Instance());
}

void TCPEstablished::Transmit (
    TCPConnection* t, TCPOctetStream* o
) {
    t->ProcessOctet(o);
}

void TCPListen::Send (TCPConnection* t) {
    // envia SYN, recebe SYN, ACK, etc.

    ChangeState(t, TCPEstablished::Instance());
}
```

Após executar o trabalho específico do estado, essas operações chamam a operação `ChangeState` para mudar o estado de `TCPConnection`. O `TCPConnection` em si não sabe nada sobre o protocolo de conexão TCP; são as subclasses `TCPState` que definem cada transição de estado e ação em TCP.

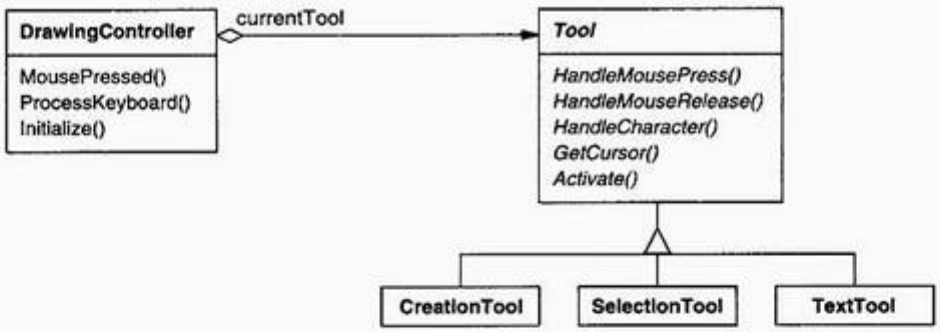
Usos conhecidos

Johnson e Zweig [JZ91] caracterizam o padrão State e sua aplicação aos protocolos de conexão TCP.

A maioria dos programas populares para fazer desenho fornece "ferramentas" para executar operações por manipulação direta. Por exemplo, uma ferramenta de desenho de linhas permite ao usuário "clicar" e arrastar para criar uma nova linha. Uma ferramenta de seleção permite ao usuário selecionar formas. Existe usualmente uma *pallete* de tais ferramentas para serem escolhidas. O usuário vê essa atividade como escolher uma ferramenta e utilizá-la, mas na realidade o comportamento do editor muda com a ferramenta corrente: quando uma ferramenta de desenho está ativa, criamos formas; quando a ferramenta de seleção está ativa, nós selecionamos formas; e assim por diante. Podemos usar o padrão State para mudar o comportamento do editor dependendo da ferramenta corrente.

Podemos definir uma classe abstrata Tool, a partir da qual definiremos subclasses que implementarão comportamentos específicos de ferramentas. O editor de desenho mantém um objeto Tool corrente e delega solicitações para ele. Ele substitui esse objeto quando o usuário escolhe uma nova ferramenta, fazendo com que o comportamento do editor de desenhos mude de acordo. Essa técnica é usada nos *frameworks* para editores de desenho HotDraw [Joh92] e Unidraw [VL90]. Ela permite aos clientes definir facilmente novos tipos de ferramentas. No HotDraw, a classe DrawingController

repassa as solicitações para o objeto Tool corrente. O seguinte diagrama de classe esboça as interfaces de Tool e DrawingController:



O termo Envelope-Letter (técnica, estilo) apresentado por Coplien [Cop92] está relacionado com State.

Envelope-Letter é uma técnica para mudar a classe de um objeto em tempo de execução. O padrão State é mais específico, focalizando sobre como lidar com um objeto cujo comportamento depende do seu estado.

Padrões relacionados

O padrão Flyweight (187) explica quando e como objetos State podem ser compartilhados.

Objetos State são freqüentemente Singletons (130).

STRATEGY — comportamental de objetos

Intenção

Definir uma família de algoritmos, encapsular cada uma delas e torná-las intercambiáveis. Strategy permite que o algoritmo varie independentemente dos clientes que o utilizam.

Também conhecido como

Policy

Motivação

Existem muitos algoritmos para quebrar um *stream* de texto em linhas. Codificar de maneira fixa e rígida tais algoritmos nas classes que os utilizam não é desejável, por várias razões:

- clientes que necessitam de quebras de linhas se tornam mais complexos se incluirem o código de quebra de linhas. Isso os torna maiores e mais difíceis de manter, especialmente se suportam múltiplos algoritmos de quebra de linhas;