Design Patterns

Patterns Comportamentais

Este tipo de *patterns* preocupam-se com os algoritmos e a alocação de responsabilidades aos objectos.

Descrevem sobretudo a comunicação existente entre os objectos. Focam a atenção na maneira como os objectos interactuam entre eles e menos no controlo de fluxo dos dados.

O modo como se distribui aspectos de comportamento entre as classes é obviamente feito através da herança. A este tipo de *patterns* comportamentais chamam-se *patterns comportamentais de classe*. Os *patterns comportamentais de objectos* utilizam a composição ao invés da herança.

Design Patterns - Chain of Responsability

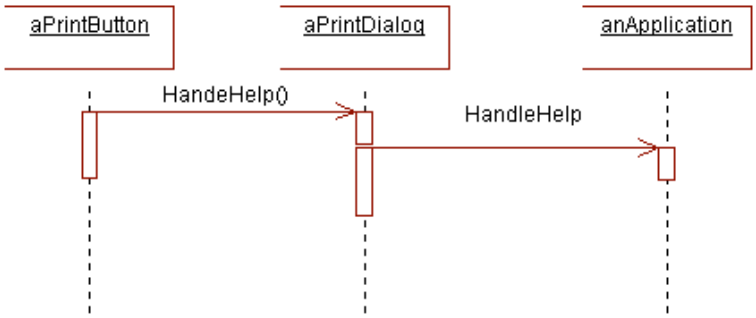
Objectivo: Permite não associar explicitamente o tratamento de uma mensagem a um receptor, mas proporcionar que exista mais do que um candidato a tratar o pedido. Permite criar uma cadeia de objectos através da qual o pedido é passado até que alguém o trate.

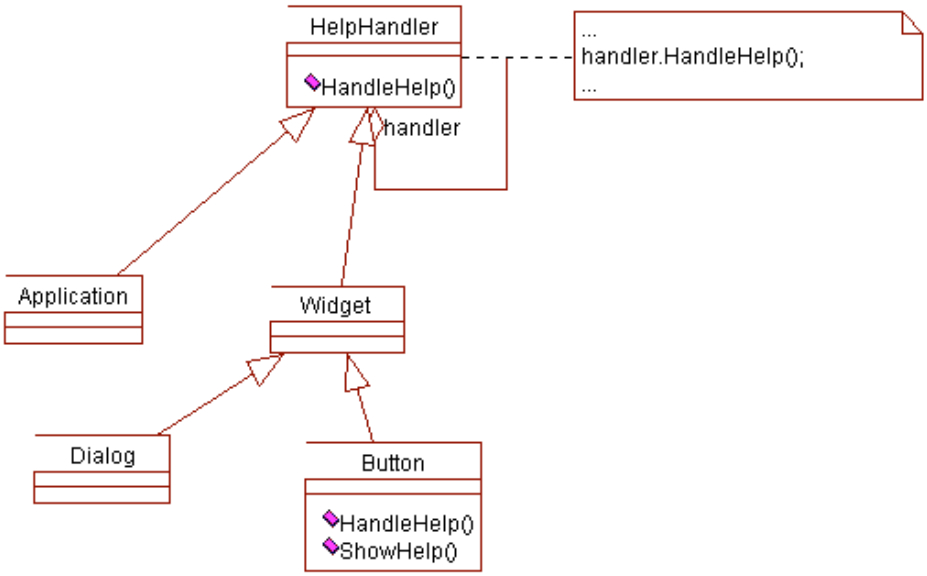
Motivação: Considere-se um interface gráfico em que a ajuda é contextualizada. O utilizador pode obter ajuda sobre qualquer parte da interface apenas por *premir* nela.

O problema advém do facto de que o objecto que fornece a ajuda não é conhecido pelo objecto que inicia o pedido de ajuda.

O *pattern* deve providenciar maneira de este tipo de interacção ser definido e especificado.

Design Patterns - Chain of Responsability



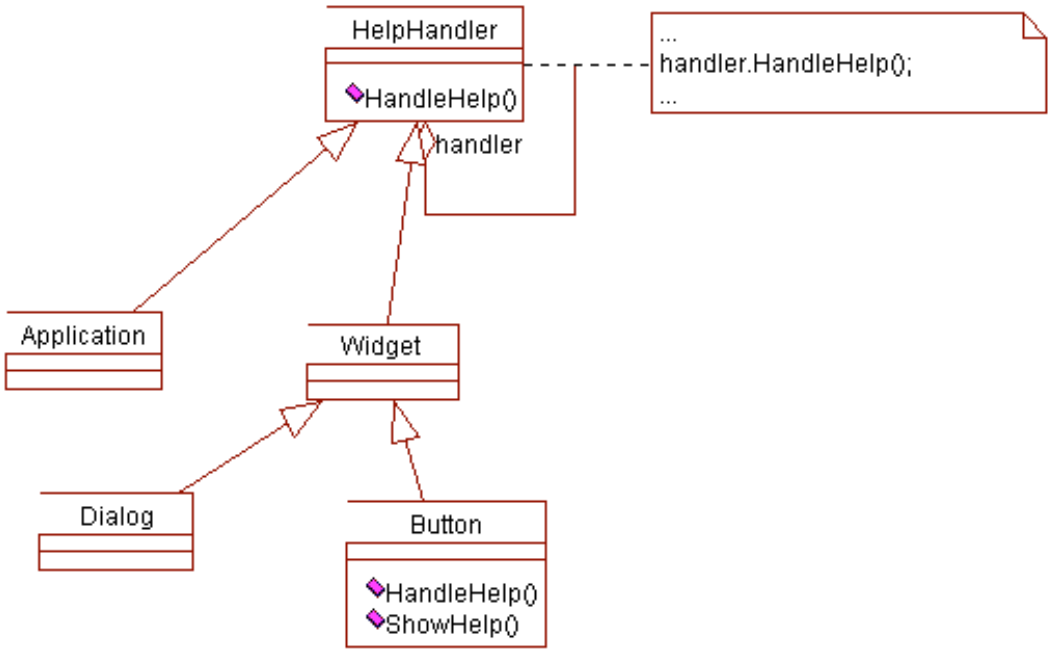


Design Patterns - Chain of Responsability

Aplicação: Usar quando:

- mais do que um objecto pode responder a um pedido e quem responde não é conhecido de antemão
- se quer direccionar um pedido para um grupo de objectos sem especificar quem o recebe e trata
- o conjunto de objectos que pode responder a um pedido pode variar em tempo de execução

Estrutura:



Design Patterns - Chain of Responsability

Consequências:

- o emissor e o receptor de um pedido não se conhecem. Em termos de implementação, cada objecto apenas precisa de conhecer o seu sucessor na cadeia (claro que isto depende da topologia...)
- pode-se acrescentar funcionalidade e especialização a um sistema, adicionando um novo objecto que trata um determinado tipo de problema (existem contudo aqui possíveis problemas de compatibilidade de interfaces...)
- não existe garantia de recepção do pedido

Implementação: Possíveis problemas ao nível de:

- implementação da topologia
- ligação aos objectos *sucessores*
- são objectos *passivos*?? Se não, possibilidade de existência de *deadlock*...

Design Patterns - Command

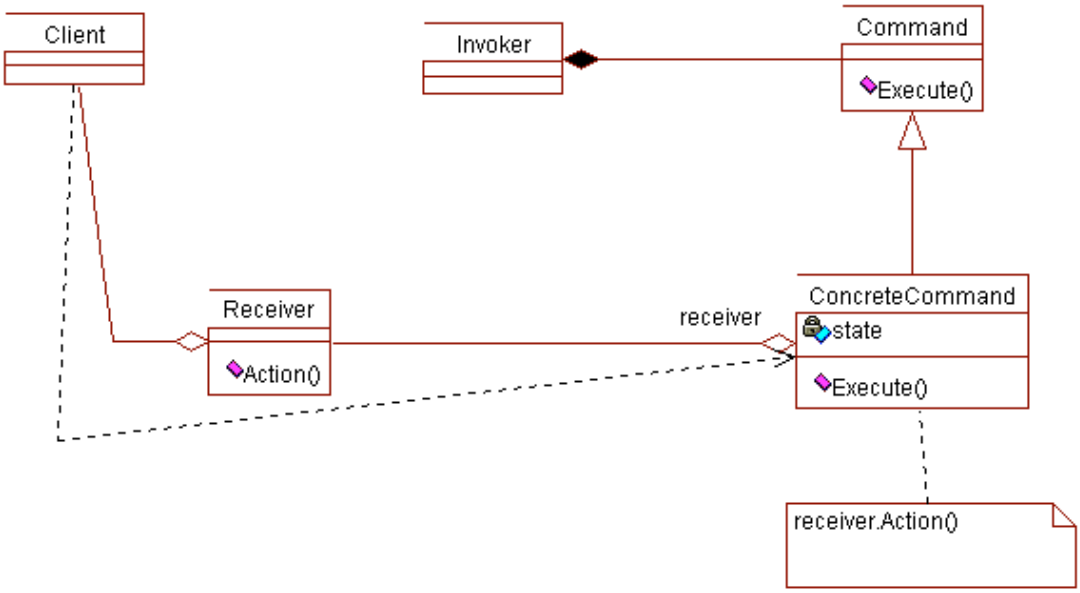
Objectivo: Encapsular um pedido como sendo um objecto, permitindo assim que os clientes possam ser parametrizados por diferentes pedidos.

Aplicação: Necessário quando:

- se pretende parametrizar os objectos pela acção a executar
- especificar, *serializar* e efectuar pedidos em alturas diferentes da execução. Os objectos do tipo command podem ter um tempo de vida independente do objecto que os originou
- é necessário suportar a acção de *"undo"*. Como o comando é encapsulado num objecto, esse objecto pode não ser executado, permitindo recuperar o estado anterior.
- se pretende de alguma forma suportar a noção de transação. Um comando representa de uma forma algo simplista uma transação.

Design Patterns - Command (cont.)

Estrutura:



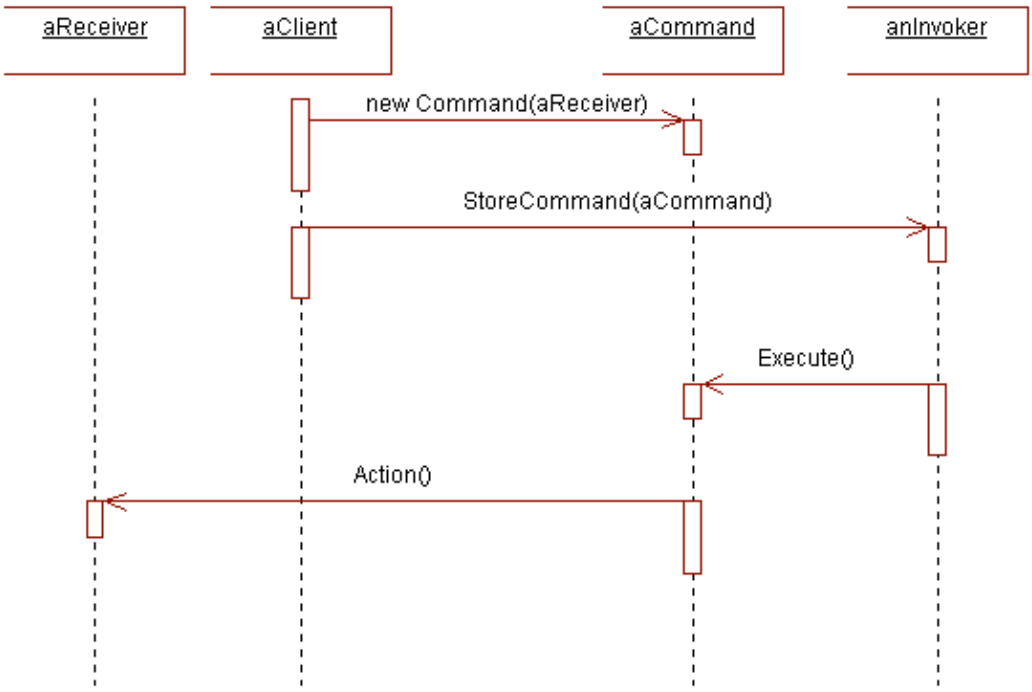
Participantes:

- Command - interface para a execução de uma operação
- ConcreteCommand
  - define uma associação entre o Receiver e uma acção
  - implementa o método de execução, invocando o correspondente método do receptor

Design Patterns - Command (cont.)

- Client - corresponde a quem cria um comando e determina quem é o seu receptor. Normalmente corresponde à aplicação.
- Invoker - despoleta a execução do pedido
- Receiver - sabe como executar as operações associadas à execução de um pedido.

Colaborações:



Design Patterns - Command (cont.)

Consequências:

- Permite-se a separação entre o objecto que invoca a operação daquele que sabe como executá-la
- Os comandos são objectos de maior grau de importância, na medida que o sistema funciona a partir deles. No entanto a nível de implementação são objectos perfeitamente normais.
- os comandos podem ser compostos, isto é, um comando pode ser uma composição de comando. Para tal efeito pode-se recorrer ao *pattern* Composite.
- é fácil adicionar novos comandos a um sistema, uma vez que nada muda nas classes existentes.

Design Patterns - Iterator

Objectivo: Providencia uma maneira de aceder aos elementos de um objecto composto de modo sequencial sem se saber qual a representação interna.

Motivação: Um objecto agregador, como uma lista, deve permitir que se aceda aos seus elementos sem que seja necessário saber como é que a lista é internamente representada. Também pode ser necessário fazer diversas travessias sobre uma lista, mas não deve ser a lista a fornecer todas estas funcionalidades (mesmo que saiba quais são as necessárias...).

A motivação principal do Iterator é retirar ao agregador estas responsabilidades de acesso e travessia criando para tal um objecto **iterador**.

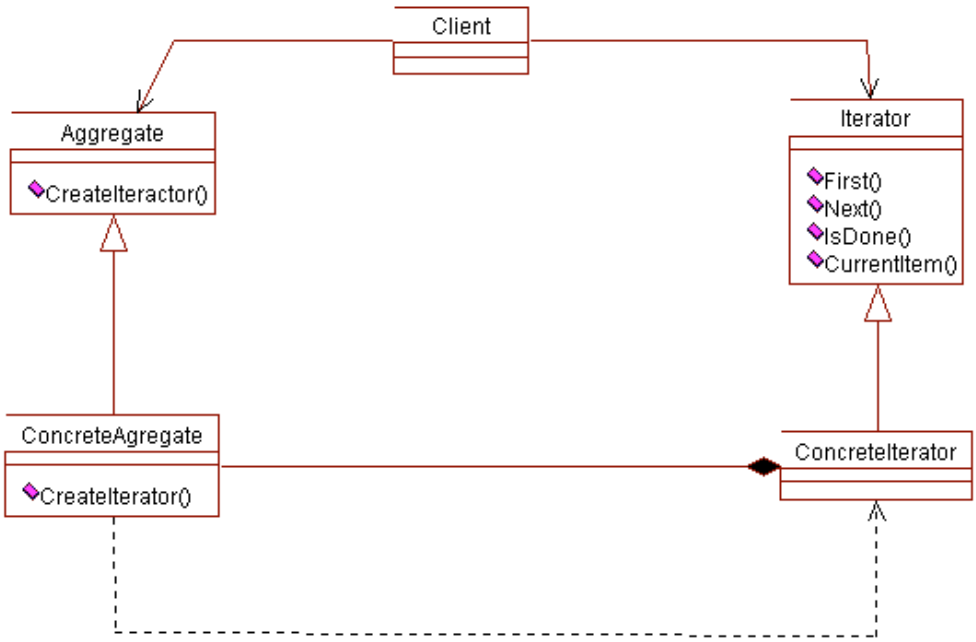
O iterador para uma lista podia ser o seguinte:

Design Patterns - Iterator (cont.)

Aplicação:

- aceder a um objecto agregador sem expôr a sua implementação
- suportar múltiplas travessias de um agregador
- providenciar um interface uniforme para efectuar diferentes travessias em diferentes estruturas agregadoras

Estrutura:



Design Patterns - Iterator (cont.)

Consequências:

- suporta variações nas travessias de um agregador. Para mudar o tipo de travessias basta mudar o iterador.
- a existência de iteradores pode simplificar a estrutura agregadora
- num mesmo agregador podem estar pendentes mais do que uma travessia. Basta para isso que sobre ela estejam a trabalhar mais do que um iterador.

Design Patterns - Observer

Objectivo: Definir dependências do tipo um para muitos, de modo a que quando um objecto mudar os que dele dependem também mudem.

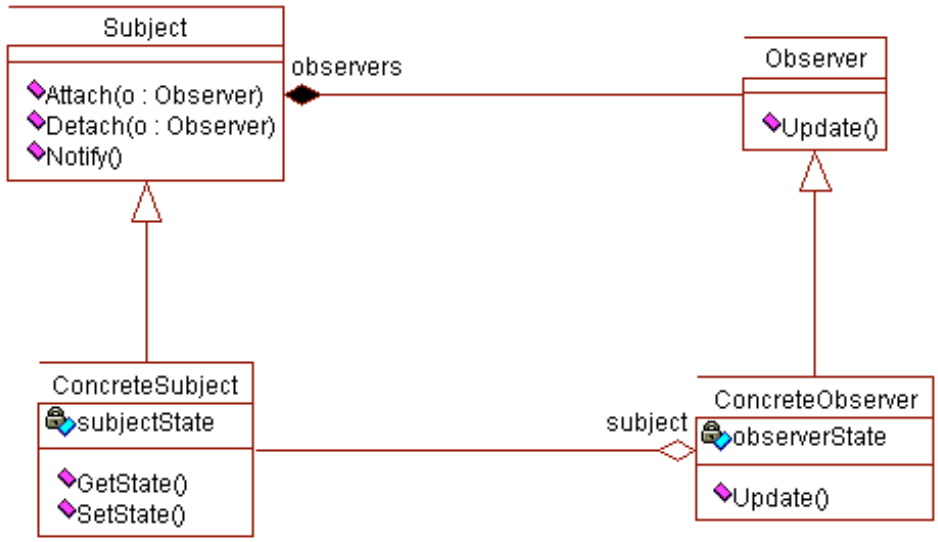
Motivação: Imagine-se por exemplo um sistema que dada uma folha de cálculo, permita ter várias vistas (possivelmente gráficas) sobre essa folha de cálculo. Quando um dos valores de uma nota muda é necessário actualizar as vistas de modo a que reflictam a nova realidade.

Aplicação:

- quando uma abstracção tem duas vistas, uma dependente da outra. Cada uma das vistas pode estar num objecto separado e assim permite-se que continuem a trabalhar em conjunto.
- quando uma mudança num objecto implica mudar o estado de outros, mas não se sabe quais, e quantos, objectos mudam.
- quando se precisa que um objecto avise (notifique) outros sem saber que objectos são e como se comportam.

Design Patterns - Observer (cont.)

Estrutura:

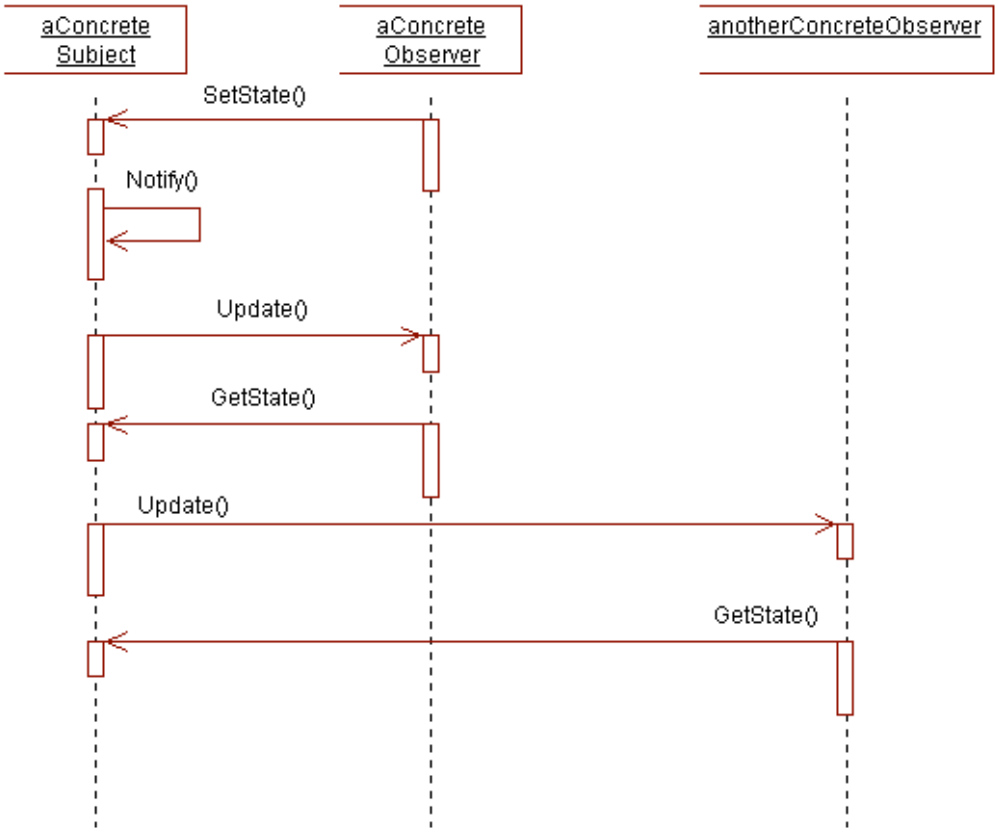


Participantes:

- Subject - sabe quais são os seus observadores
- Observer - define um interface para os objectos de **devem ser** notificados das mudanças em Subject
- ConcreteSubject - envia um notificação para os observadores quando o seu estado muda
- ConcreteObserver - implementa o interface que efectua a actualização de estado quando existe mudança

Design Patterns - Observer (cont.)

Colaborações:



Consequências:

- os objectos ao serem reutilizados não implicam a reutilização dos seus observadores
- ligação fraca entre o observado e os observadores

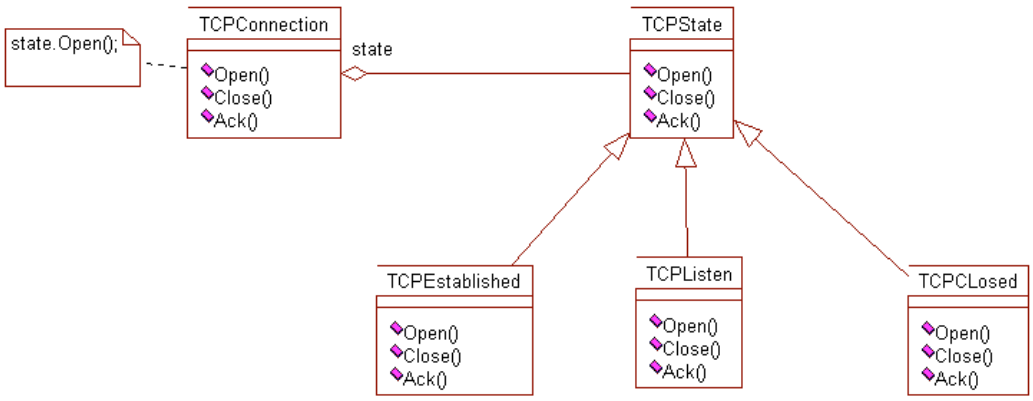
Design Patterns - Observer (cont.)

- suporte à difusão de mensagens. O observado pode interagir com um população de observadores. A comunicação já não é ponto a ponto.
- Propagação de actualizações pode ser eventualmente pesada.

Design Patterns - State

Objectivo: reflectir as situações em que a mudança de estado de um objecto leve também à mudança do seu comportamento.

Motivação: Considere-se que se quer modelar o estado de uma ligação TCP. Como se sabe durante o tempo de vida de uma ligação o protocolo faz com que as entidades tenham estados diferente e respostas diferentes e dependentes do estado.

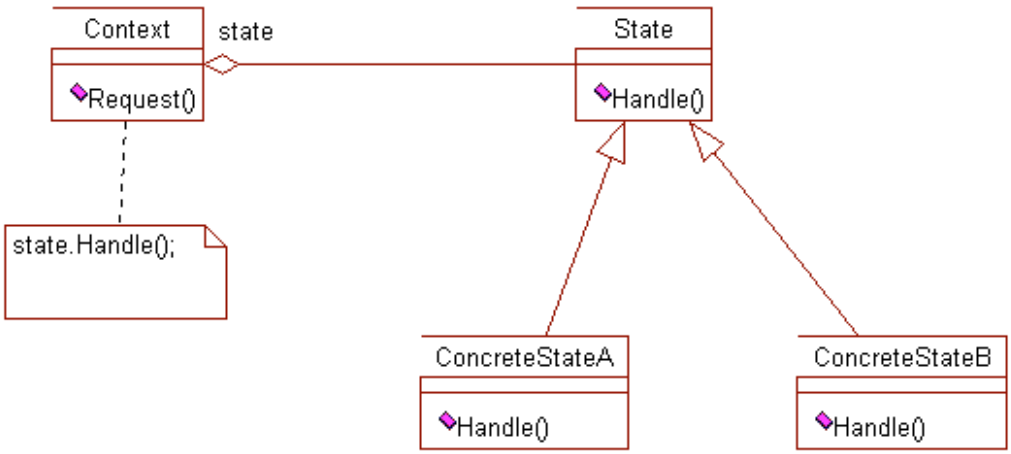


Design Patterns - State (cont.)

Aplicação:

- o comportamento depende do estado e não é viável ter estruturas condicionais
- cada uma das possíveis partes condicionais redefine uma quantidade de coisas, como constantes, variáveis, etc. que não é de todo possível pôr dentro de aninhamentos condicionais.

Estrutura:



Design Patterns - State (cont.)

Participantes:

- Context - define a interface para os clientes e tem uma instância que define qual o comportamento actual
- State - encapsula o comportamento associado a um determinado estado
- ConcreteState - implementação real do comportamento

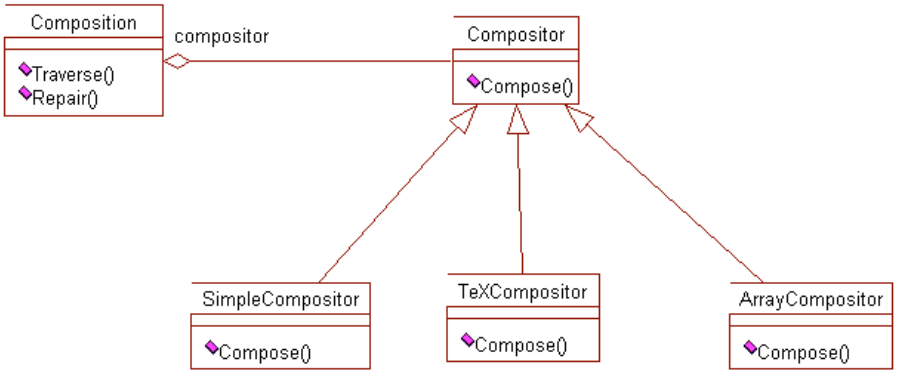
Consequências:

- novas alterações de estado são inseríveis criando uma nova subclasse
- permite visualizar no diagram de classes os estados possíveis que uma entidade tem. Permite-se assim uma visão sobre a parte comportamental, e de máquina de estados, de algumas das entidades.

Design Patterns - Strategy

Objectivo: Definir uma família de algoritmos, encapsular cada um deles tornando-os assim reutilizáveis em mais do que uma situação. Permite assim que clientes diferentes possam usar algoritmos (estratégias) diferentes.

Motivação: Suponha-se o exemplo de construção de um editor de texto, em que existe a necessidade de separar uma stream de texto por linhas. Claro que a separação das linhas depende da aplicação cliente. Esta pode precisar de marcas, de separação física, etc. Torna-se também difícil alterar novos algoritmos de partição de linhas quando estes estão embebidos na aplicação cliente.

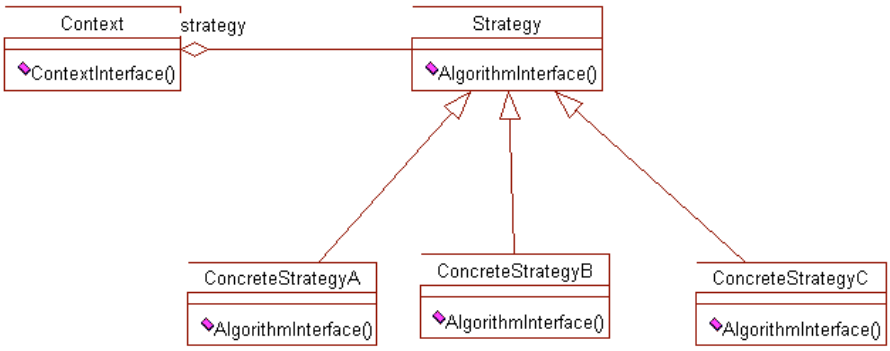


Design Patterns - Strategy (cont.)

Aplicação:

- muitas classes que estão relacionadas são diferentes no comportamento. Pode-se desta maneira fazer com que as classes sejam parametrizadas pelo seu comportamento.
- quando se necessita de variações de um algoritmo
- um algoritmo usa dados que não devem ser conhecidos do cliente
- uma classe pode exibir muitos comportamentos

Estrutura:



Design Patterns - Strategy (cont.)

Participantes:

- Strategy - declara um interface comum para todos os algoritmos suportados.
- ConcreteStrategy - implementa o algoritmo cuja interface respeita Strategy
- Context - é configurada para utilizar um algoritmo concreto e mantém uma referência para o algoritmo.

Consequências:

- permite detectar funcionalidades semelhantes. Favorece assim a criação de famílias de algoritmos
- apresenta uma alternativa ao esquema usual de herança. A classe é parametrizada pelo algoritmo
- permite eliminar expressões condicionais, na escolha do algoritmo
- aumenta o número de objectos e o peso (memória e comunicação) das aplicações.

Design Patterns

Discussão dos *patterns* comportamentais

- Encapsular (esconder) variações de comportamento.
  - existe um objecto de uma classe conhecida que é agregado. Essa classe é apenas a superclasse das concretizações que se querem utilizar.
- Objectos como argumento
  - a configuração, algoritmo, estratégia são passados como parâmetro possibilitando assim que o cliente possa mudar em tempo de compilação (e execução) de estratégia sem *danos colaterais*.

Design Patterns

Discussão sobre o catálogo de *patterns* de Gamma *et al.*

- possível sensação de *"Deja Vu"*
  - não apresenta algoritmos
  - não apresenta técnicas de programação
  - não apresenta novos construtores
  - não fornecem uma metodologia ou processo
- apenas se documentam modelos existentes (óbvio!)

No entanto ... catalogar os *patterns* existentes é de extrema importância!

- aumento de vocabulário
- descrição de técnicas de modelação

Design Patterns

A primeira pessoa a falar de *patterns* foi um arquitecto, C. Alexander, que desenhou um catálogo de *patterns*.

Convém ver agora que o caminho seguido pelos *design patterns* é ligeiramente diferente:

- as pessoas fazem edifícios desde sempre. Software desde há algumas décadas. Existem pedaços de software (ou aplicações) que sejam clássicos??
- na construção dos edifícios existe uma ordem pela qual os *patterns* são utilizados. No software isso não é verdade!
- Os *patterns* de Alexander focavam os problemas a resolver. Os *patterns* apresentados centram-se nas soluções.
- O objectivo último de Alexander era gerar, a partir de uma sequência de *patterns*, um edifício. Tal não é possível nestes *design patterns*.

Design Patterns

O que se pode esperar *já* dos *patterns*?

- Vocabulário
  - conceito
  - aplicação (regras, consequências, etc)
- Documentação
  - a maior parte dos sistemas OO existentes usa os *patterns* apresentados (ainda que não assumidamente)
  - grande parte dos problemas com os sistemas de software OO resulta na falta de compreensão destes patterns
  - grande parte das soluções software utilizadas que correspondem a *patterns* não estão documentadas

O que se pode esperar *ainda* dos *patterns*?

- Linguagens de *patterns*
- ambientes de composição de *patterns*
- ....???

Design Patterns

Patterns na indústria Os *patterns* e a sua aplicação ao "mundo real"vistos por Marshall Cline[a] numa *Communications of the ACM*:

- Vantagens:
  - os *patterns* permitem dirigir todo o processo, bem assim como a própria equipa
  - podem ser usados reactivamente
  - e também pró-activamente...vistos que não são apenas documentação
- Desvantagens:
  - não se pode tornar a parte pelo todo. Os *patterns* não são um fim em si mesmo. Servem apenas para favorecer o processo de criação de aplicações.
  - alguns *patterns* são difíceis de aprender
  - não são ainda úteis a quem começa

Conclusão: apesar de úteis precisam de tempo de maturação...

[a]Presidente da Paradigm Shift Inc.