PADRÕES DE PROJETO DE SOFTWARE

ACH 2003 — COMPUTAÇÃO ORIENTADA A OBJETOS

Daniel Cordeiro

8 de junho de 2016

Escola de Artes, Ciências e Humanidades | EACH | USP

PADRÕES COMPORTAMENTAIS

PADRÕES COMPORTAMENTAIS

São os padrões relacionados especificamente ao modo como os objetos se comunicam entre si.

- ~~Chain of responsibility~~
- ~~Command~~
- ~~Interpreter~~
- Iterator
- Mediator
- Memento
- Null Object
- Observer
- State
- Strategy
- Template method
- Visitor

ITERATOR

OBJETIVO

- Prover um modo de acessar sequencialmente os elementos de um objeto agregador sem expor a implementação interna desse objeto
- As bibliotecas padrão de Java e C++ a usam para desacoplar as classes de coleção de seus algoritmos
- Fazer com que o ato de percorrer uma coleção seja um objeto

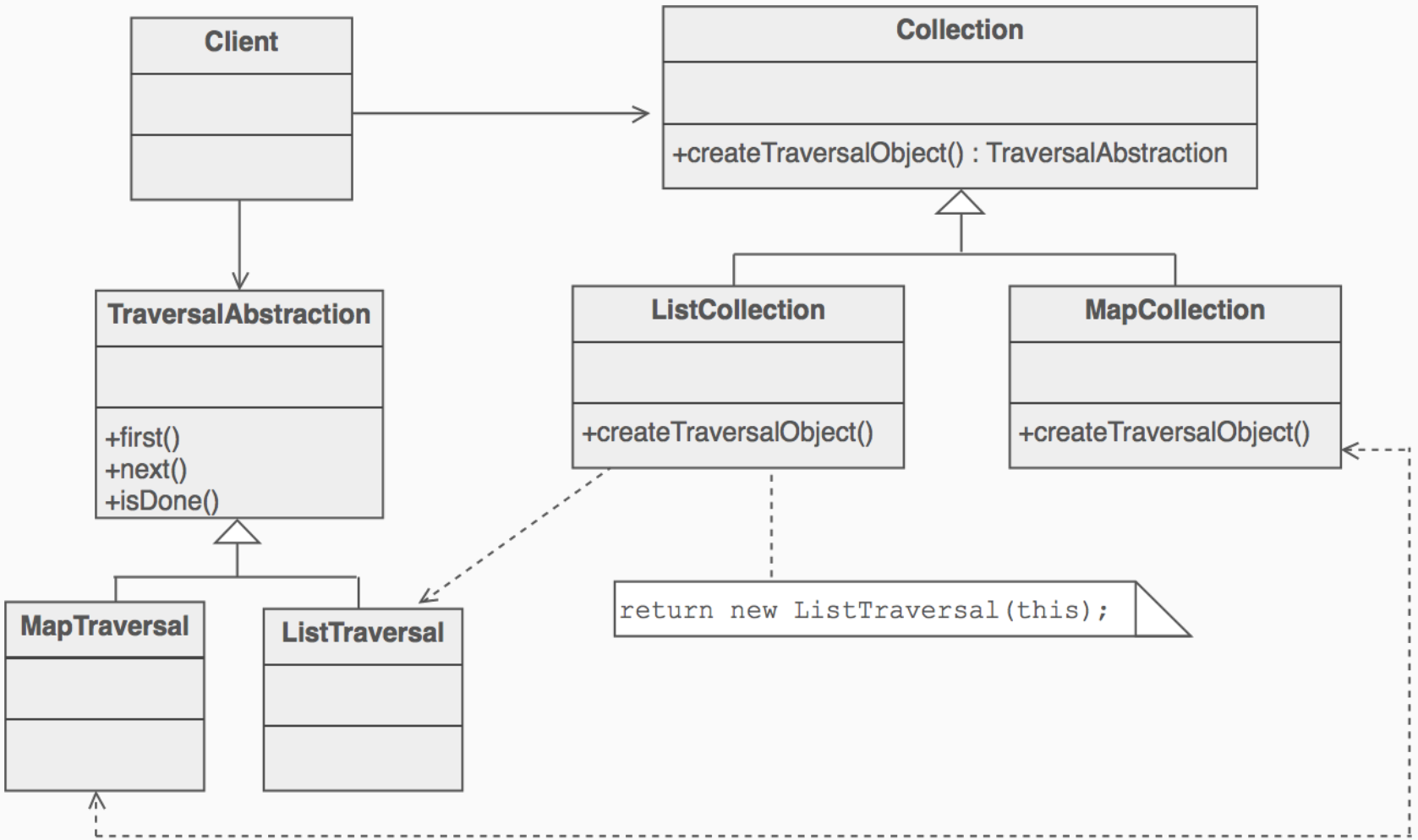
Problema

Precisamos de uma forma “abstrata” de percorrer estruturas de dados completamente diferentes para que possamos escrever algoritmos capazes de interagir com cada estrutura de forma transparente.

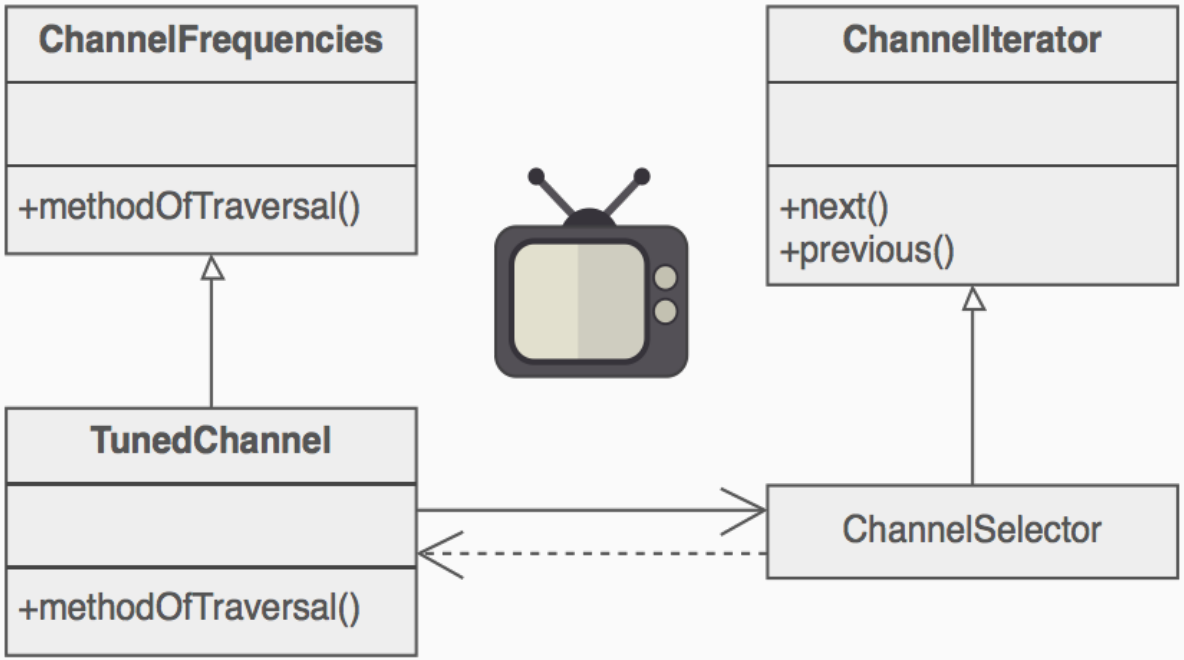
DISCUSSÃO

- Um objeto agregador tal como uma lista deve prover um meio de acesso aos seus elementos sem expor sua estrutura interna
- Você também pode querer percorrer uma lista de diferentes modos sem ter que modificar a interface de Lista para cada modo diferente
- A ideia principal é mover a responsabilidade de percorrer a coleção para uma nova classe (Iterator) que definirá um modo universal de acesso aos elementos

ESTRUTURA


Client
Collection
+createTraversalObject() : TraversalAbstraction
TraversalAbstraction
+first()
+next()
+isDone()
ListCollection
+createTraversalObject()
MapCollection
+createTraversalObject()
MapTraversal
ListTraversal
return new ListTraversal(this);

EXEMPLO


ChannelFrequencies
+methodOfTraversal()
ChannelIterator
+next()
+previous()
TunedChannel
+methodOfTraversal()
ChannelSelector

IMPLEMENTAÇÃO

```
class BooksCollection implements IContainer {
        private String m_titles[] = {"Design Patterns","1","2","3","4"};

        public IIterator createIterator() {
                BookIterator result = new BookIterator();
                return result;
        }


        private class BookIterator implements IIterator {
                private int m_position;

                public boolean hasNext() {
                        if (m_position < m_titles.length)
                                return true;
                        else
                                return false;
                }
                public Object next() {
                        if (this.hasNext())
                                return m_titles[m_position++];
                        else
                                return null;
                }
        }
}
```

LISTA DE VERIFICAÇÃO

1. Adicione um método `create_iterator` à classe “agregadora” e dê acesso privilegiado à classe Iterator
2. Projete uma classe Iterator que encapsule a lógica para percorrer a coleção
3. Os clientes pedem para a coleção criar um iterador
4. Clientes usam os métodos `first`, `is_done`, `next()` e `current_item` para acessar os elementos da coleção

MEDIATOR

OBJETIVO

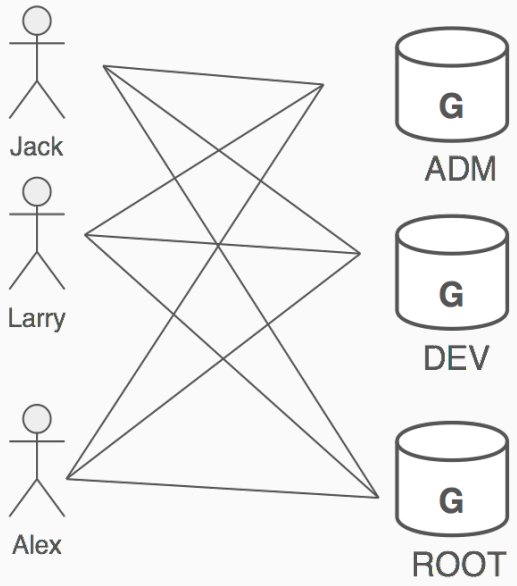
- Definir um objeto que encapsula a interação entre um grupo de objetos
- Promover um acoplamento fraco ao fazer com que os objetos não se referenciem explicitamente
- Projetar um objeto intermediário para desacoplar vários objetos
- Criar um objeto que represente uma relação muitos–para–muitos

Problema

Eu quero fazer um projeto de componentes reutilizáveis, mas as dependências entre as partes reutilizáveis acabaram fazendo do meu código um “código espaguete”.

DISCUSSÃO

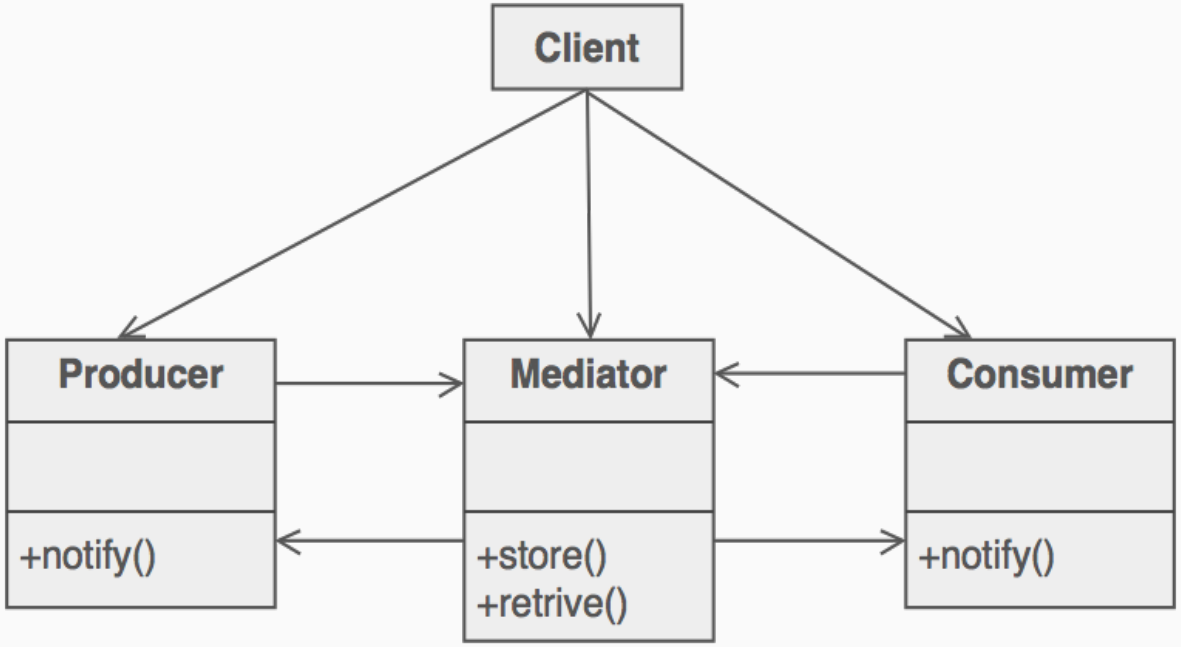
- Peguemos como exemplo o sistema de controle de acesso do Unix
- Os acessos são gerenciados em três níveis: *todos*, *grupo* (um conjunto de usuários) e *dono* (o usuário dono do recurso)
- Cada usuário pode ser membro de mais de um grupo e cada grupo pode ter zero ou mais usuários
- Se estivéssemos modelando esse software, grupos poderiam estar acoplados a usuários e vice-versa



DISCUSSÃO

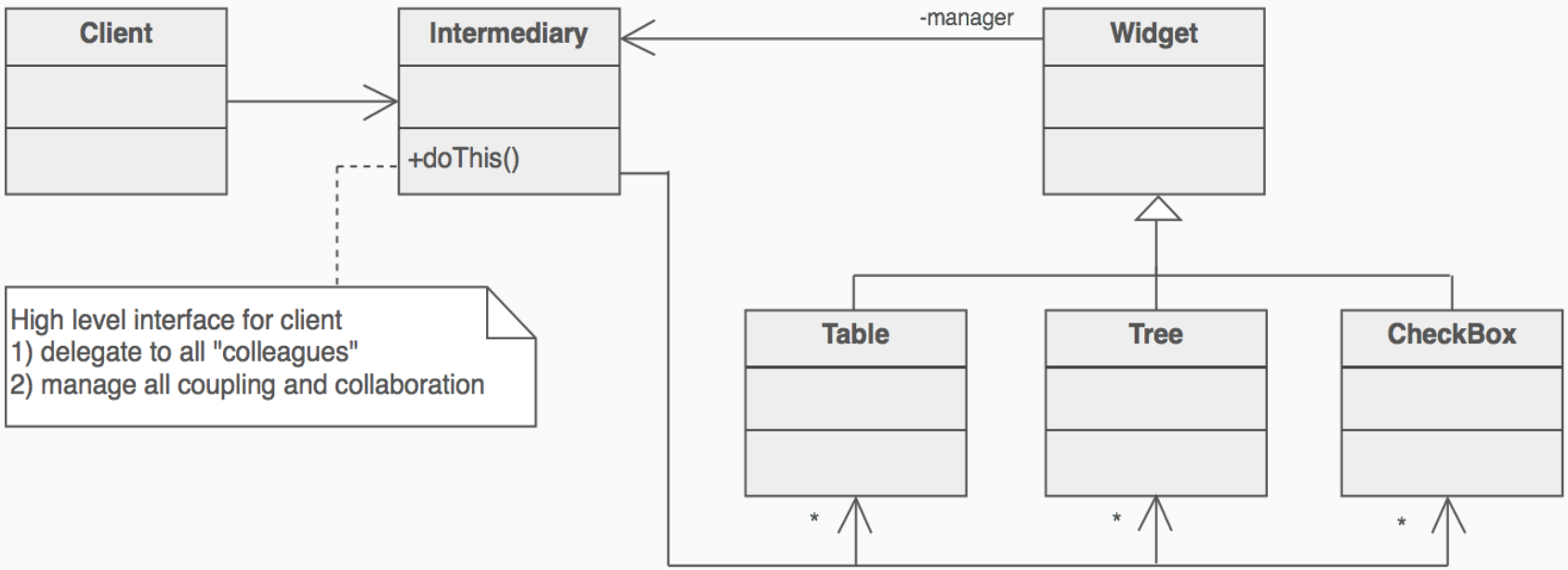
- Uma alternativa seria introduzir um nível de indireção: pegue o mapeamento de usuários em grupos e de grupos em usuários e crie uma nova abstração
- Essa abordagem tem várias vantagens:
  - usuários e grupos não são acoplados entre si
  - vários mapeamentos podem ser manipulados ao mesmo tempo
  - a abstração do mapeamento pode ser estendida no futuro com uma classe derivada
- O particionamento do sistema em muitos objetos promove reusabilidade, mas uma proliferação de interconexões de objetos (o “espaguete”) acaba com a reusabilidade conseguida

ESTRUTURA

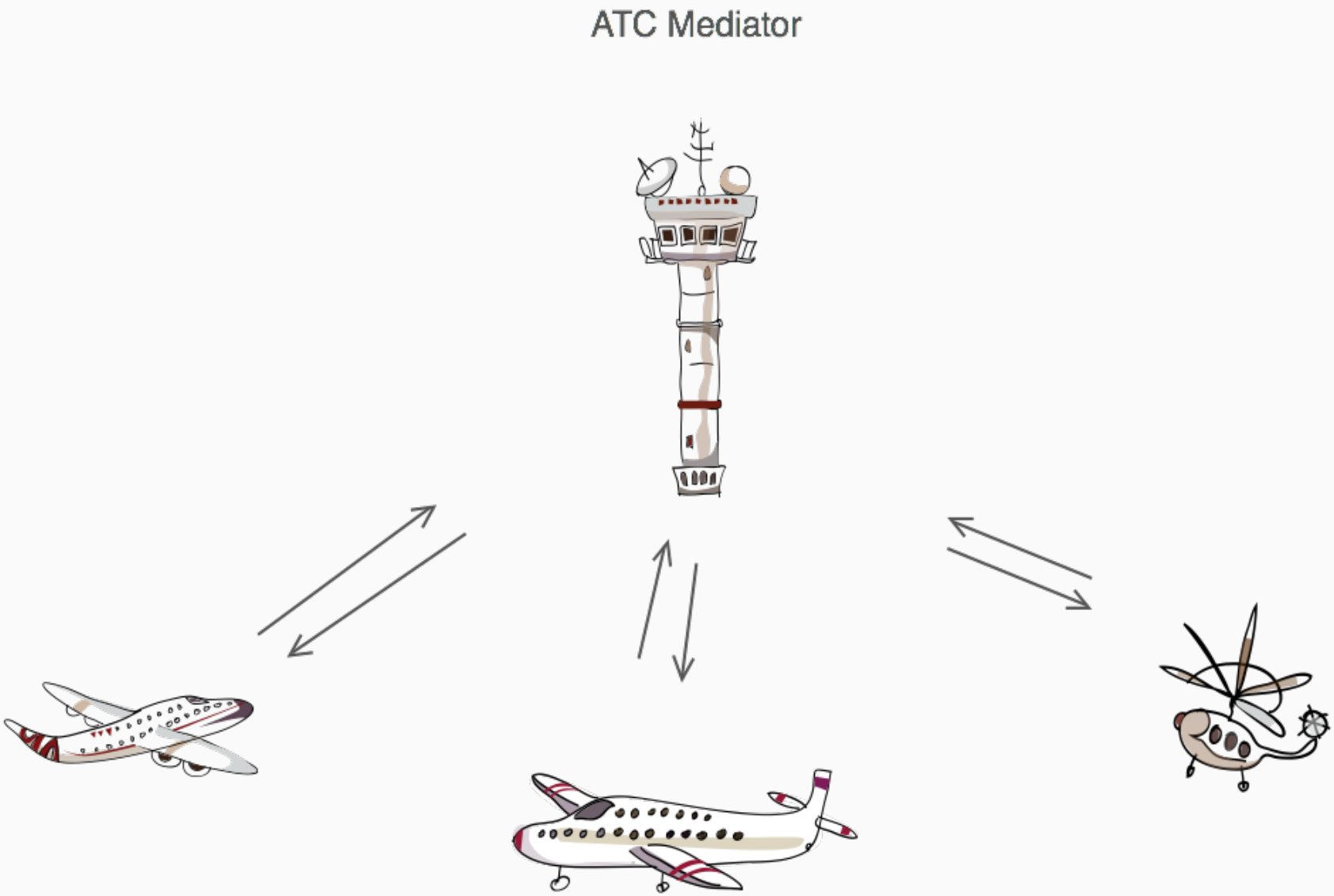


- Os colegas (ou pares) não estão acoplados entre si
- Cada um fala com o Mediator, que conhece e conduz a "orquestração" dos outros

ESTRUTURA


Client
Intermediary
+doThis()
-manager
Widget
Table
Tree
CheckBox
High level interface for client
1) delegate to all "colleagues"
2) manage all coupling and collaboration
*
*
*

EXEMPLO



LISTA DE VERIFICAÇÃO

- Identifique a coleção de objetos interagindo que se beneficiaria do desacoplamento mútuo
- Encapsule a abstração dessas interações em uma nova classe
- Crie uma instância dessa nova classe e mude os objetos para interagir somente com o Mediator
- Balanceie o princípio de acoplamento com o princípio de distribuição de responsabilidades
- Cuidado para não criar um objeto “deus”

MEMENTO

OBJETIVO

- Sem violar o encapsulamento, capturar e externalizar o estado interno de um objeto para que o estado possa ser restaurado posteriormente
- Possibilitar a criação de pontos de verificação (*check points*)
- Permitir operações para desfazer (*undo*) ou reversão (*rollback*)

Problema

Conseguir restaurar o estado original de um objeto

DISCUSSÃO

- O cliente solicita um Memento[1] de um objeto quando for necessário fazer um ponto de verificação do objeto
- O objeto inicializa seu Memento com uma caracterização de seu estado
- O cliente pode ser responsável por guardar o Memento, mas é responsabilidade do objeto original armazenar e recuperar a informação do Memento
- O Memento é “opaco” para o cliente e pros outros objetos
- Quando o cliente precisar reverter o estado do objeto, ele passa o Memento guardado anteriormente para o objeto, que consegue recuperar o estado armazenado

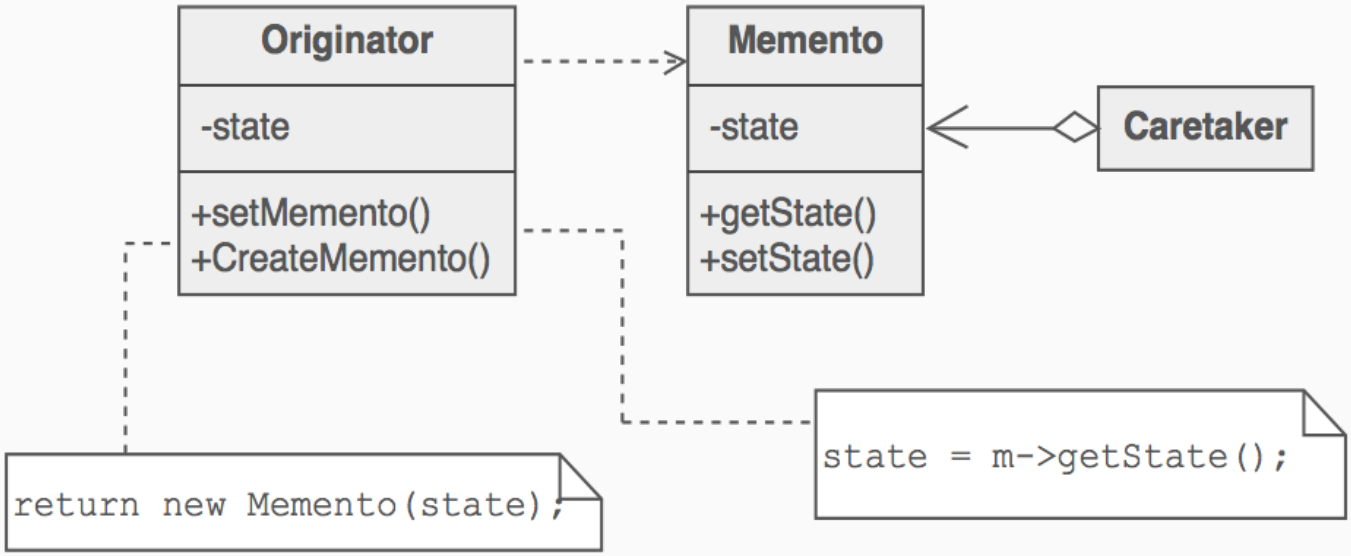
[1] Memento: livro de lembranças, agenda onde se anota o que se quer recordar.

DISCUSSÃO

A implementação de operações de “desfazer” e de “refazer” pode ser implementada naturalmente com uma pilha de objetos Command e uma pilha de objetos Memento.

Como?

ESTRUTURA



Originator o objeto que sabe como gravar e restaurar a si mesmo

Caretaker o objeto que sabe como e por quê o *Originator* precisa saber se gravar

Memento uma “caixa trancada” que só pode ser aberta pelo *Originator* e que é cuidada pelo *Caretaker*

IMPLEMENTAÇÃO I

```java
import java.util.*;

class Memento {
   private String state;

   public Memento(String stateToSave) { state = stateToSave; }
   public String getSavedState() { return state; }
}



class Caretaker {
   private ArrayList<Memento> savedStates = new ArrayList<Memento>();

   public void addMemento(Memento m) { savedStates.add(m); }
   public Memento getMemento(int index) { return savedStates.get(index); }
}
```

IMPLEMENTAÇÃO II

```
class Originator {
   private String state;
   /* o objeto pode ter vários outros dados que podem
      consumir muita memória, mas que não são
      necessários para reestabelecer o objeto */

   public void set(String state) {
       System.out.println("Originator: Setting state to "+state);
       this.state = state;
   }

   public Memento saveToMemento() {
       System.out.println("Originator: Saving to Memento.");
       return new Memento(state);
   }
   public void restoreFromMemento(Memento m) {
       state = m.getSavedState();
       System.out.println("Originator: State after restoring from Memento: "+sta
   }
}

class MementoExample {
```

Implementação III

```
    public static void main(String[] args) {
        Caretaker caretaker = new Caretaker();

        Originator originator = new Originator();
        originator.set("State1");
        originator.set("State2");
        caretaker.addMemento( originator.saveToMemento() );
        originator.set("State3");
        caretaker.addMemento( originator.saveToMemento() );
        originator.set("State4");

        originator.restoreFromMemento( caretaker.getMemento(1) );
    }
}
```

LISTA DE VERIFICAÇÃO

1. Identifique os papéis *cliente* e *objeto original*
2. Crie uma classe Memento que seja visível pelo objeto original
3. O cliente sabe quando é preciso criar um ponto de verificação do objeto original
4. O objeto original cria e copia seu estado para um Memento
5. O cliente guarda (mas não pode espiar dentro de) um Memento
6. O cliente sabe quando deve restaurar o objeto original
7. O objeto original se restaura usando o estado gravado anteriormente no Memento