# Programação Orientada a Objetos II

## Aula 07

Prof. Leandro Nogueira Couto
UFU – Monte Carmelo
05/2013


# Observer

# Observer

- Objetivo
  - Define uma dependência entre objeto e vários outros(um-para-muitos) sendo que em uma alteração neste objeto, todos os outros são notificados
- Motivação
  - Em muitas situações um objeto pode ter vários outros dependentes.
  - Neste caso sempre que este objeto (sujeito) for alterado é interessante que outros sejam notificados
  - Isto porém pode levar a um forte acoplamento entre os mesmos
- Uso
  - Quando uma abstração (Classe) possuir dois ou mais aspectos estes podem ser encapsulados em diferentes classes
  - Em situações onde a alteração de um objeto pode afetar um grupo de outros objetos não conhecidos previamente
  - Permitir que um objeto seja capaz de notificar outros sem que estejam acoplados

# Observer

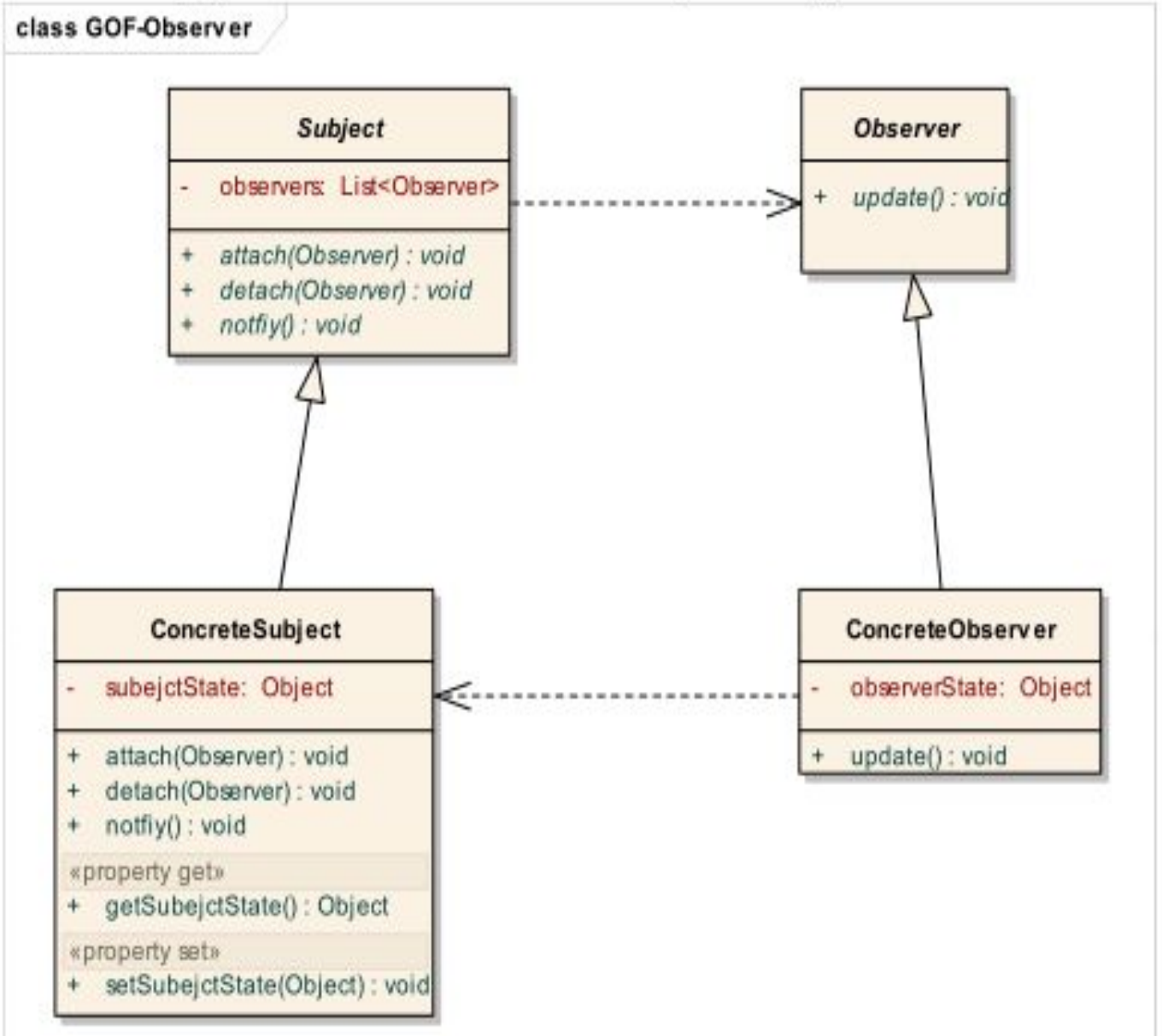
Útil pois...

- Java não suporta métodos "avulsos“
- Solução similar a callback functions

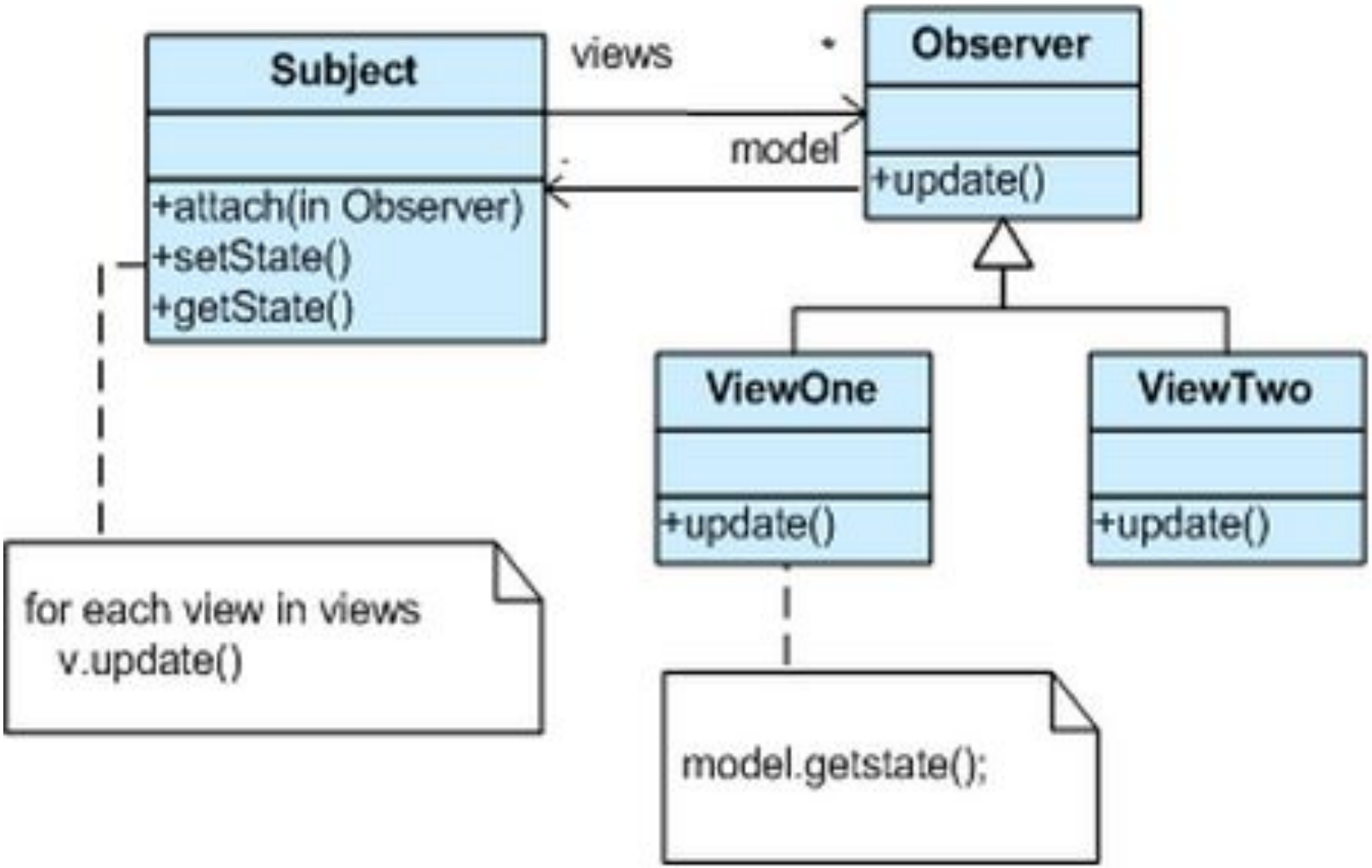
Muito usado na arquitetura de software Model-View-Controller (View observa Model)

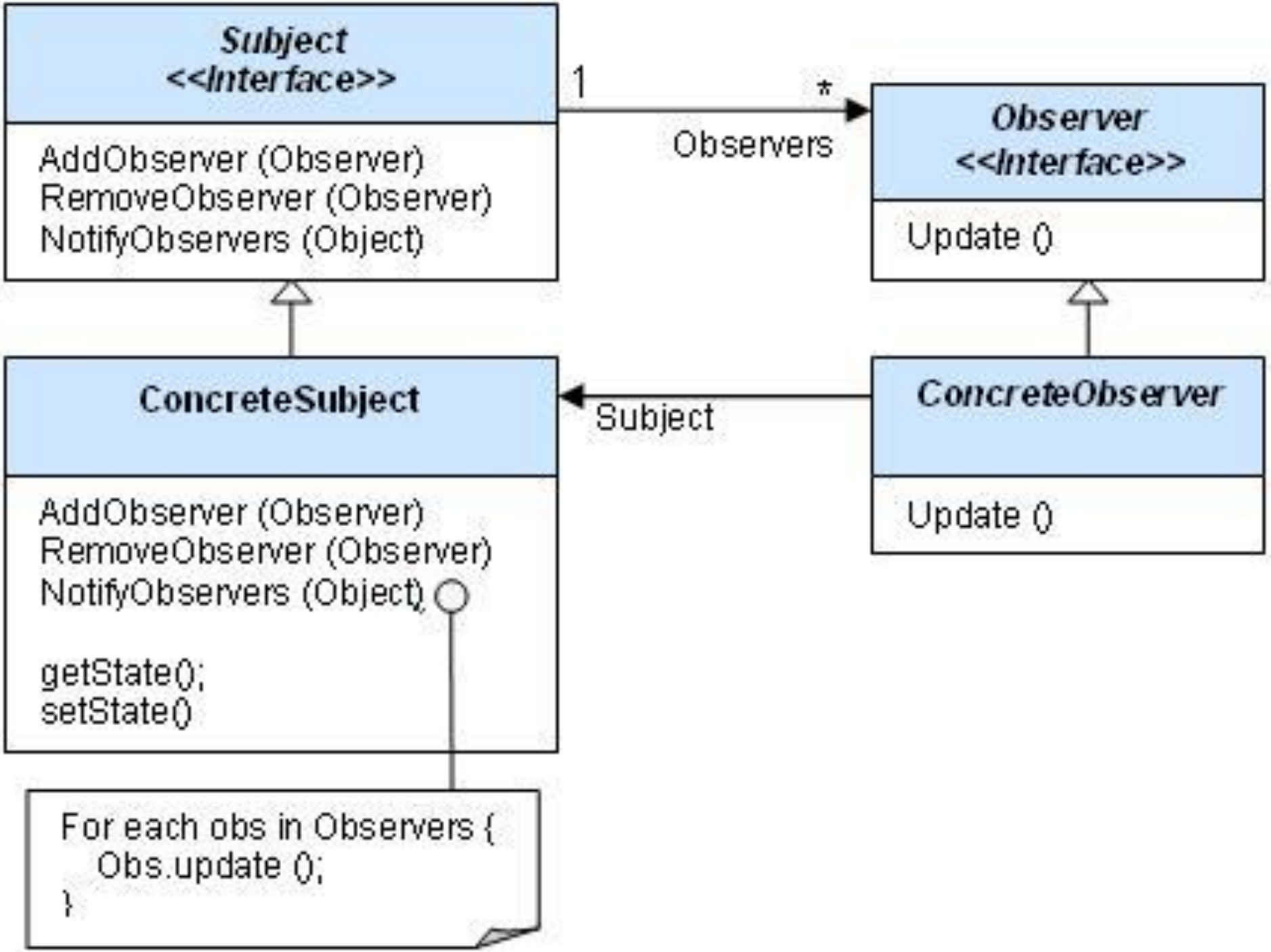
# Estrutura

- O sujeito (subject) é observado por uma lista de observadores (Observer)
- Um método notify() invoca o método update() nos observadores

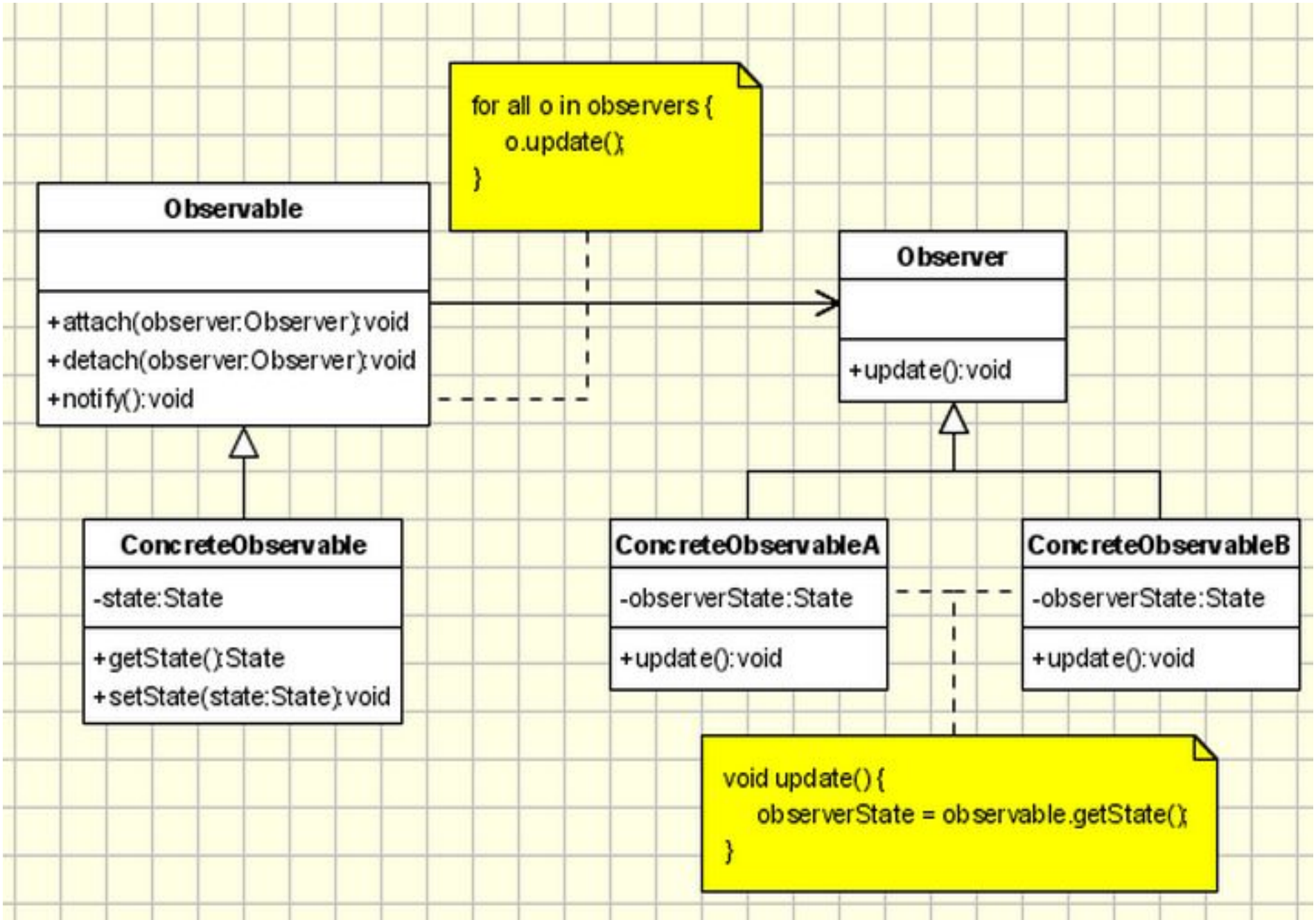


# Observer


Subject
views
*
Observer
model
+attach(in Observer)
+setState()
+getState()
+update()
ViewOne
+update()
ViewTwo
+update()
for each view in views
v.update()
model.getstate();

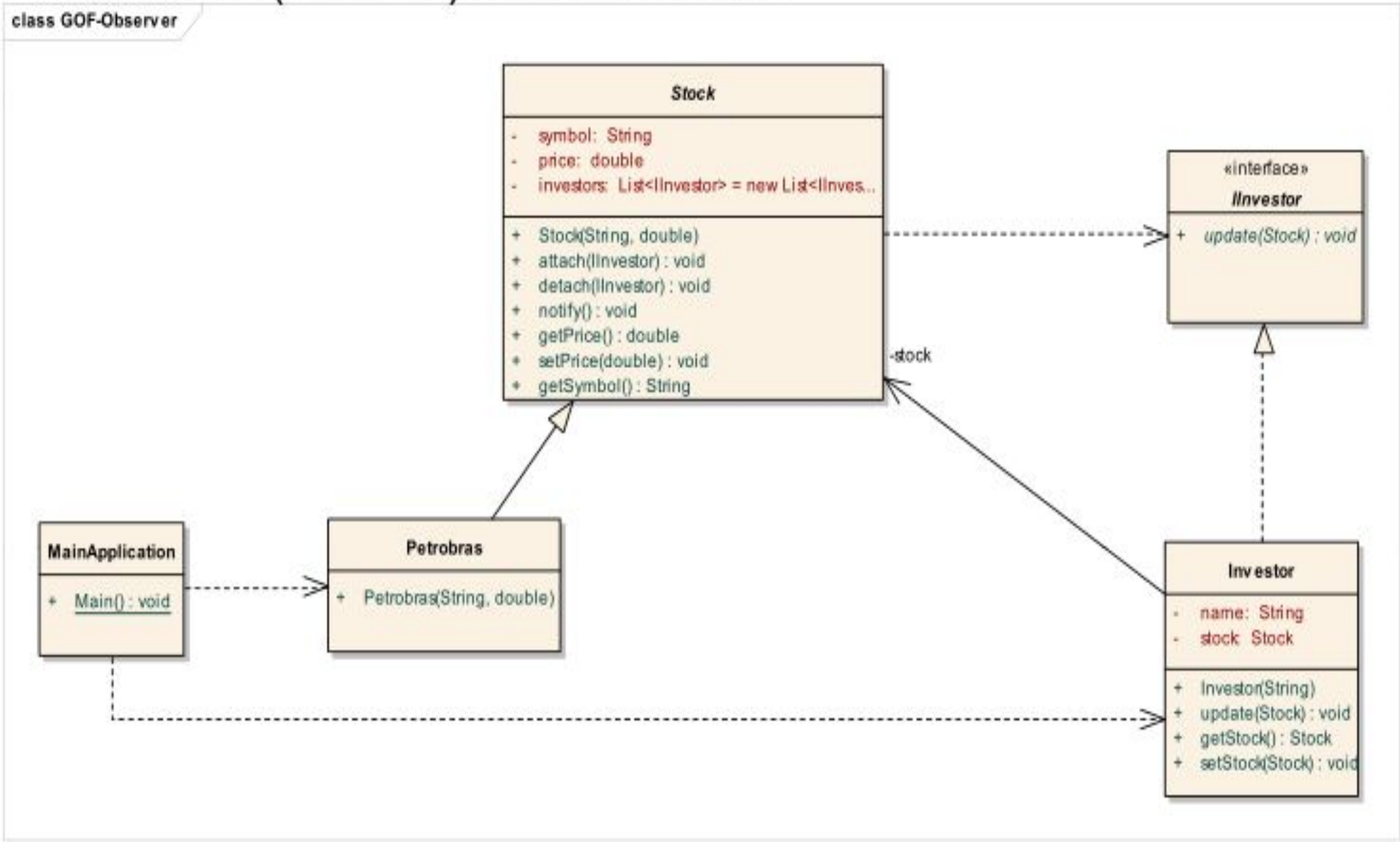

Subject
<<Interface>>
AddObserver (Observer)
RemoveObserver (Observer)
NotifyObservers (Object)
1
*
Observers
Observer
<<Interface>>
Update ()
ConcreteSubject
AddObserver (Observer)
RemoveObserver (Observer)
NotifyObservers (Object);
getState();
setState()
Subject
ConcreteObserver
Update ()
For each obs in Observers {
Obs.update ();
}

# Observer


for all o in observers {
o.update();
}
Observable
+attach(observer:Observer):void
+detach(observer:Observer):void
+notify():void
Observer
+update():void
ConcreteObservable
-state:State
+getState():State
+setState(state:State):void
ConcreteObservableA
-observerState:State
+update():void
ConcreteObservableB
-observerState:State
+update():void
void update() {
observerState = observable.getState();
}

# Exemplo

- A medida que as ações de uma empresa (Stock) forem alteradas os investirores (Inverstor) serão notificados



# Observer

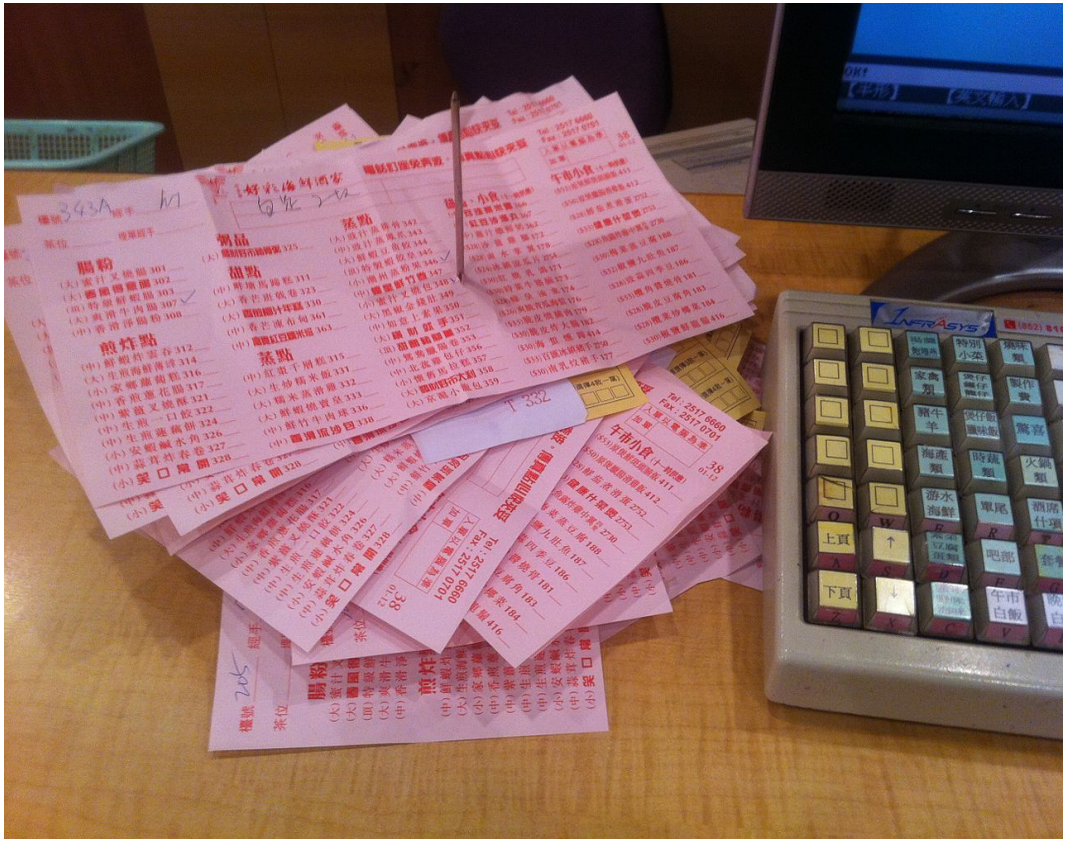
- Passos
  - Diferencie entre funcionalidade dependente e independente
  - Modele a funcionalidade independente como um Subject
  - Modele a funcionalidade dependente como uma hierarquia de Observer.
  - O Subject está acoplado apenas ao Observer base (superclasse ou interface)
  - O Cliente configura o tipo e número dos Observers
  - Observers se registram com o Subject
  - Subject transmite (broadcast) eventos a todos os Observers
  - O Subject “empurra” informação pros Observers, ou os Observers “puxam” a informação de que precisam

# Observer

- Como fazer para suportar múltiplos Subjects, ou objetos observados?
-
- Subject pode passar referência a si mesmo para Observers no método notify():

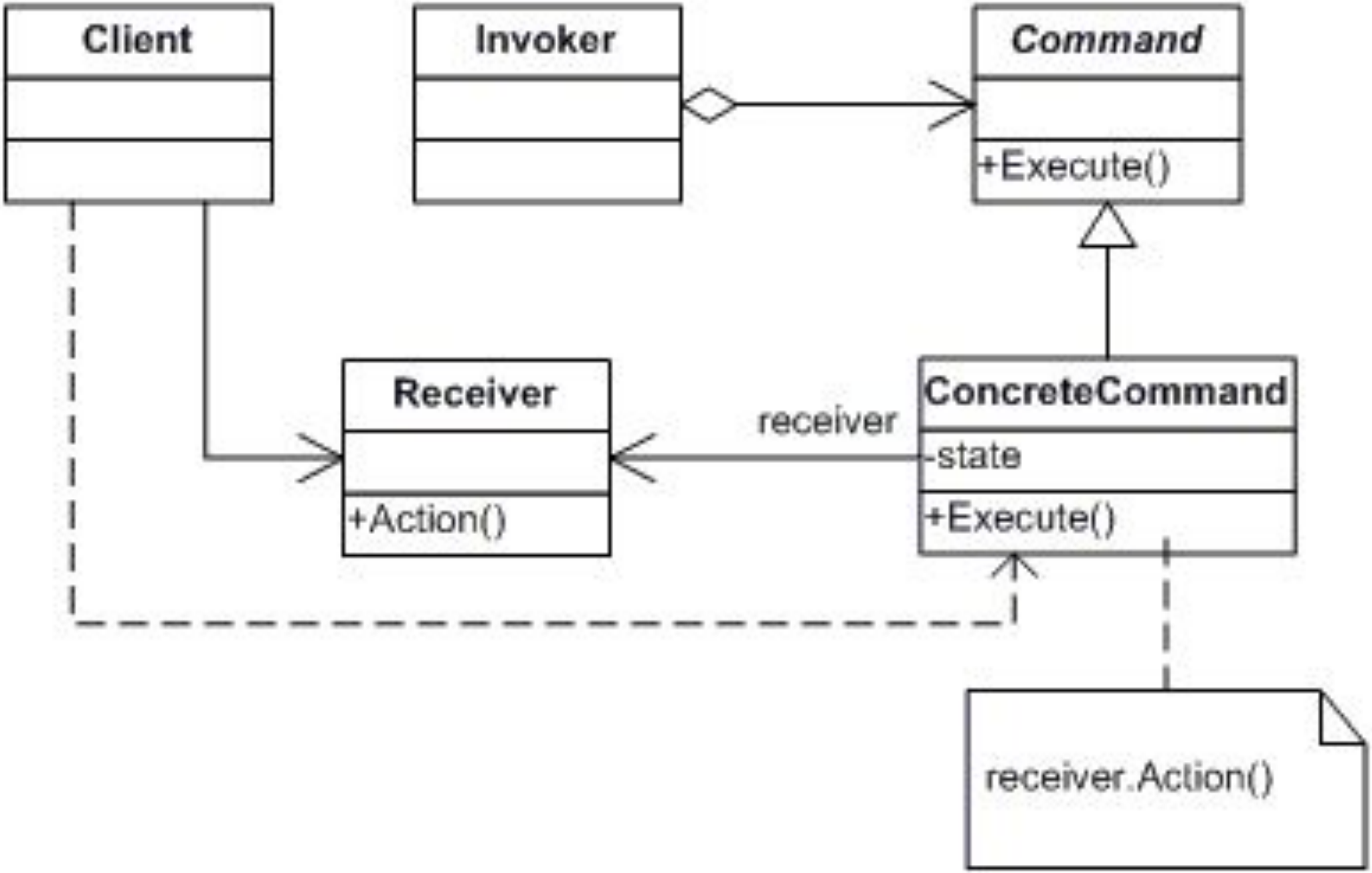
  **obs.update(this);**

# Command

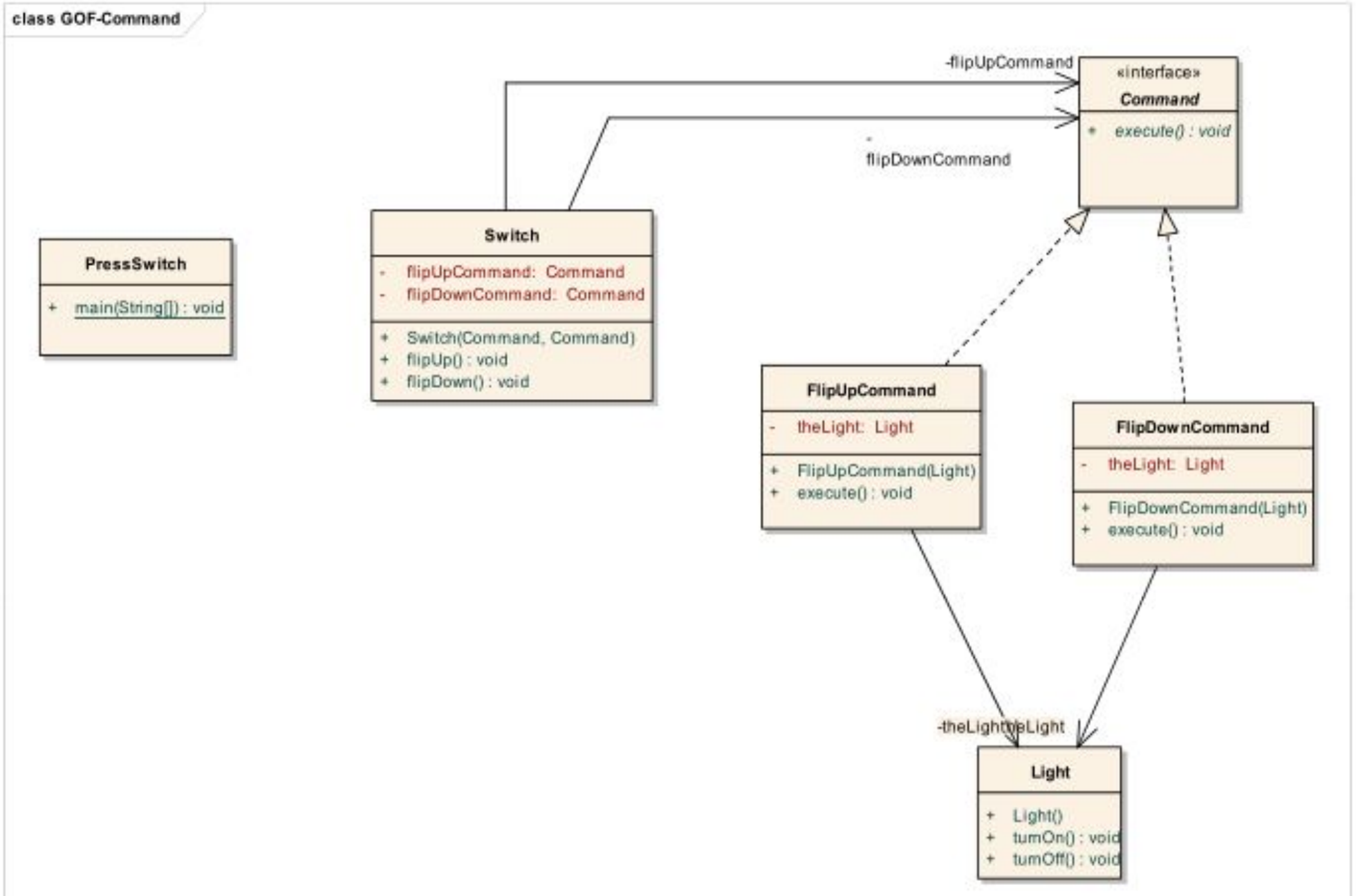


# Command

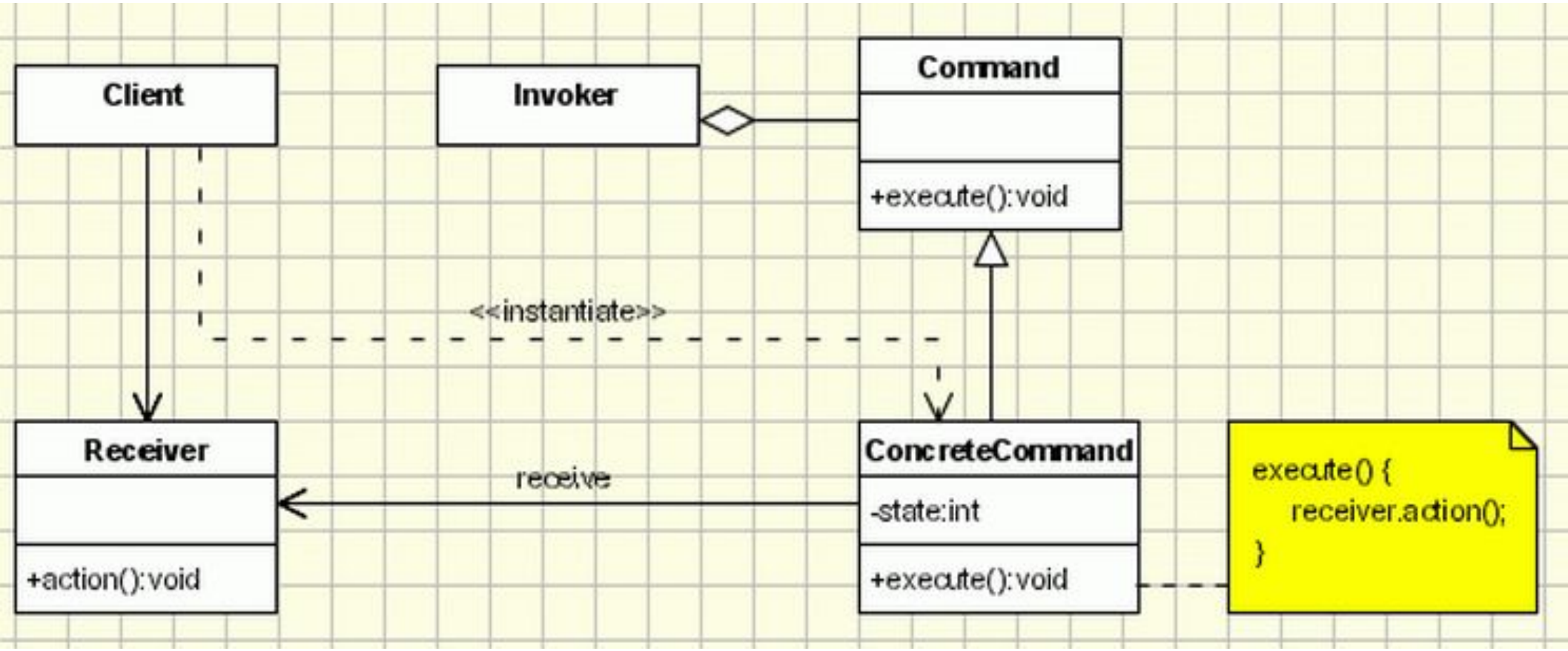
- Conceito de **macro**
- Encapsula uma sequência de comandos, ou request/order/pedido em um objeto
- Permite parametrização de requests
- Permite formação e uma fila de execução
- "Promove" a invocação de um método de um objeto a um objeto


Client
Invoker
Command
+Execute()
Receiver
+Action()
receiver
ConcreteCommand
-state
+Execute()
receiver.Action()

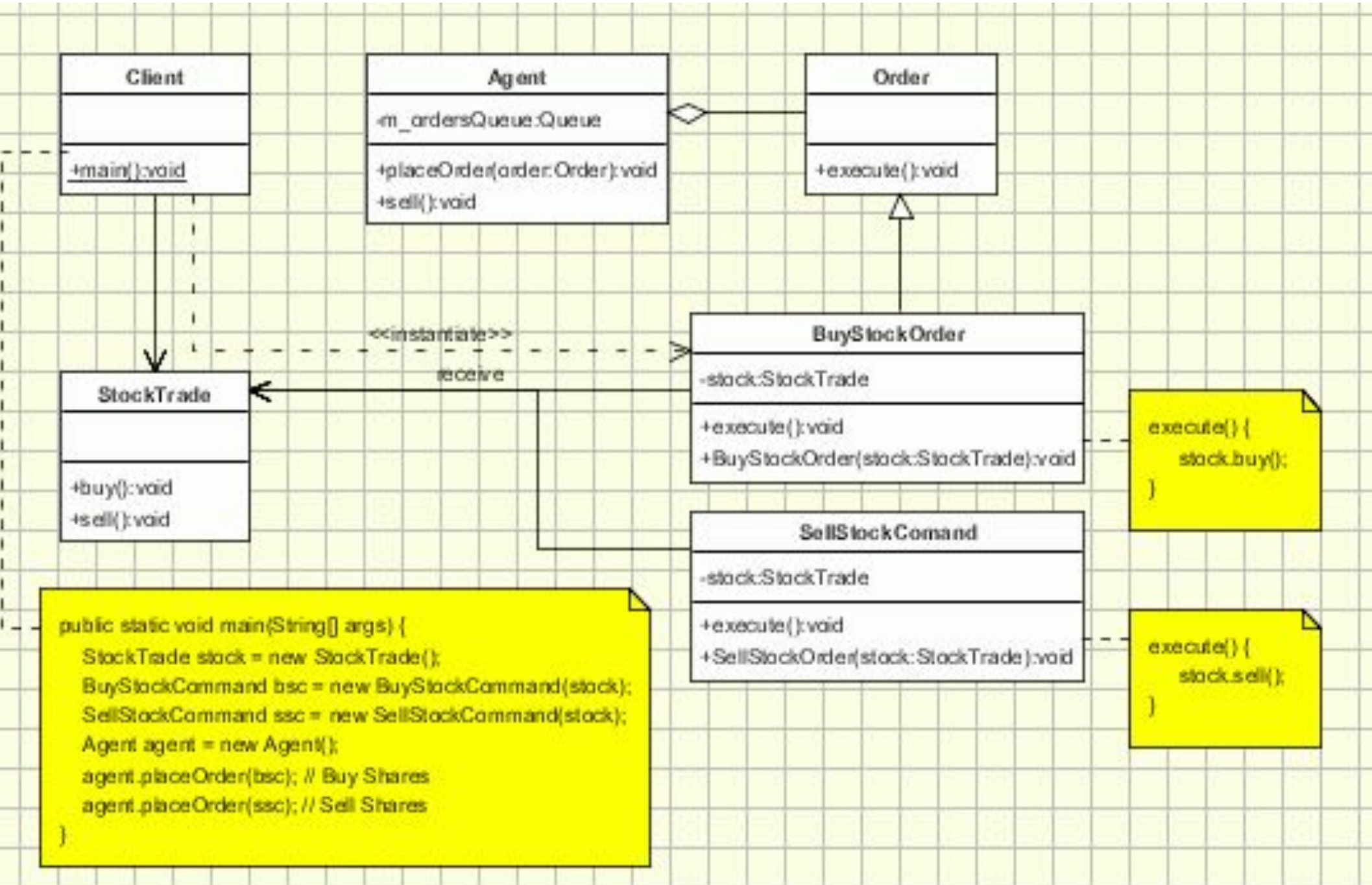
# Command


class GOF-Command
-flipUpCommand
«interface»
Command
+ execute() : void
-
flipDownCommand
PressSwitch
+ main(String[]) : void
Switch
- flipUpCommand: Command
- flipDownCommand: Command
+ Switch(Command, Command)
+ flipUp() : void
+ flipDown() : void
FlipUpCommand
- theLight: Light
+ FlipUpCommand(Light)
+ execute() : void
FlipDownCommand
- theLight: Light
+ FlipDownCommand(Light)
+ execute() : void
-theLight
-theLight
Light
+ Light()
+ turnOn() : void
+ turnOff() : void

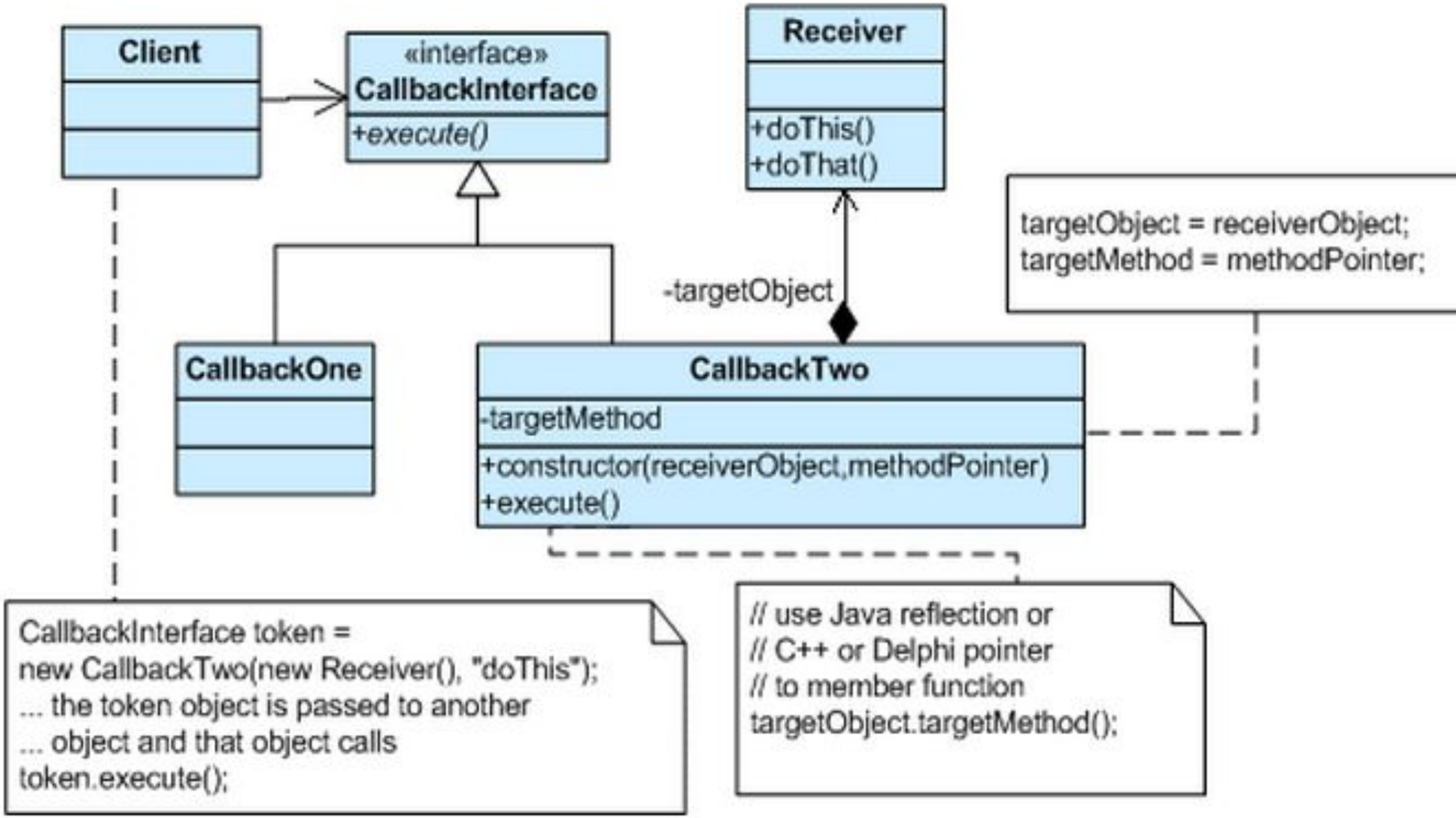
# Command


Client
Invoker
Command
+execute():void
<<instantiate>>
Receiver
+action():void
receive
ConcreteCommand
-state:int
+execute():void
execute() {
receiver.action();
}

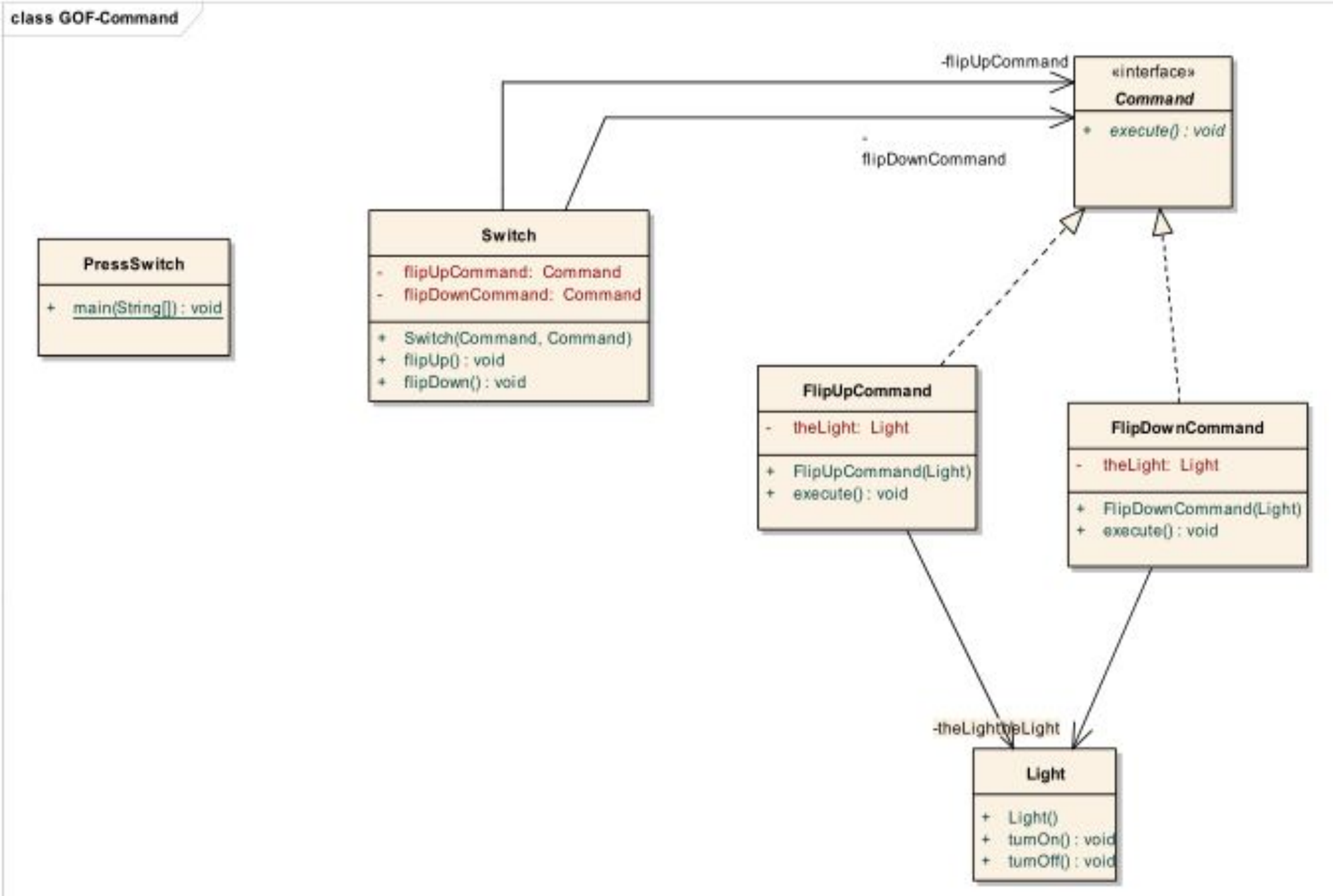
# Command


Client
+main():void
Agent
-m_ordersQueue:Queue
+placeOrder(order:Order):void
+sell():void
Order
+execute():void
<<instantiate>>
receive
StockTrade
+buy():void
+sell():void
BuyStockOrder
-stock:StockTrade
+execute():void
+BuyStockOrder(stock:StockTrade):void
SellStockComand
-stock:StockTrade
+execute():void
+SellStockOrder(stock:StockTrade):void
execute() {
stock.buy();
}
execute() {
stock.sell();
}
public static void main(String[] args) {
StockTrade stock = new StockTrade();
BuyStockCommand bsc = new BuyStockCommand(stock);
SellStockCommand ssc = new SellStockCommand(stock);
Agent agent = new Agent();
agent.placeOrder(bsc); // Buy Shares
agent.placeOrder(ssc); // Sell Shares
}

# Command


Client
«interface»
CallbackInterface
+execute()
Receiver
+doThis()
+doThat()
targetObject = receiverObject;
targetMethod = methodPointer;
-targetObject
CallbackOne
CallbackTwo
-targetMethod
+constructor(receiverObject,methodPointer)
+execute()
CallbackInterface token =
new CallbackTwo(new Receiver(), "doThis");
... the token object is passed to another
... object and that object calls
token.execute();
// use Java reflection or
// C++ or Delphi pointer
// to member function
targetObject.targetMethod();

# Command


class GOF-Command
-flipUpCommand
-flipDownCommand
«interface»
Command
+ execute() : void
PressSwitch
+ main(String[]) : void
Switch
- flipUpCommand: Command
- flipDownCommand: Command
+ Switch(Command, Command)
+ flipUp() : void
+ flipDown() : void
FlipUpCommand
- theLight: Light
+ FlipUpCommand(Light)
+ execute() : void
FlipDownCommand
- theLight: Light
+ FlipDownCommand(Light)
+ execute() : void
-theLight
-theLight
Light
+ Light()
+ turnOn() : void
+ turnOff() : void

# Command

- Participantes
  - Command – declara uma interface para executar uma operação
  - ConcreteCommand – dá extends em Command, e implementa o método Execute invocando operações correspondentes no Receiver. Define a associação entre Receiver e a ação.
  - Client – Cria o ConcreteCommand e associa com um receptor
  - Invoker – Faz o pedido
  - Receiver – Sabe fazer as operações

# Command

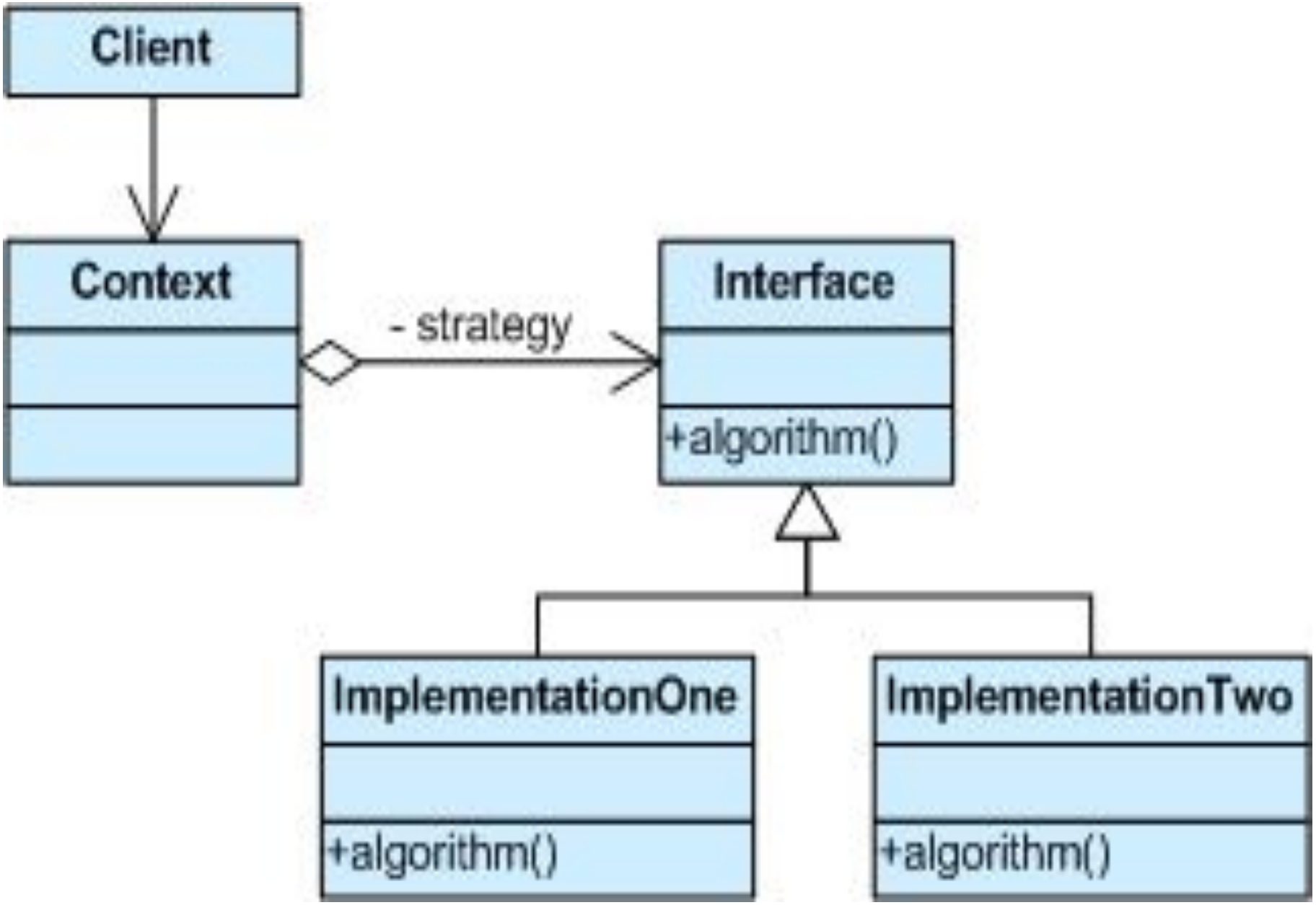
- Passos
  - Definir uma interface Command interface com uma assinatura de método como a de execute().
  - Criar uma ou mais classes derivadas que encapsulem: um objerto receptor, um método pra invocar, argumentos pra passar.
  - Instanciar um objeto Command para cada pedido.
  - Passar o objeto Command do criador (Agent) para quem sabe executar (Receiver).
  - O objeto Criador de Commands sabe quando executar.

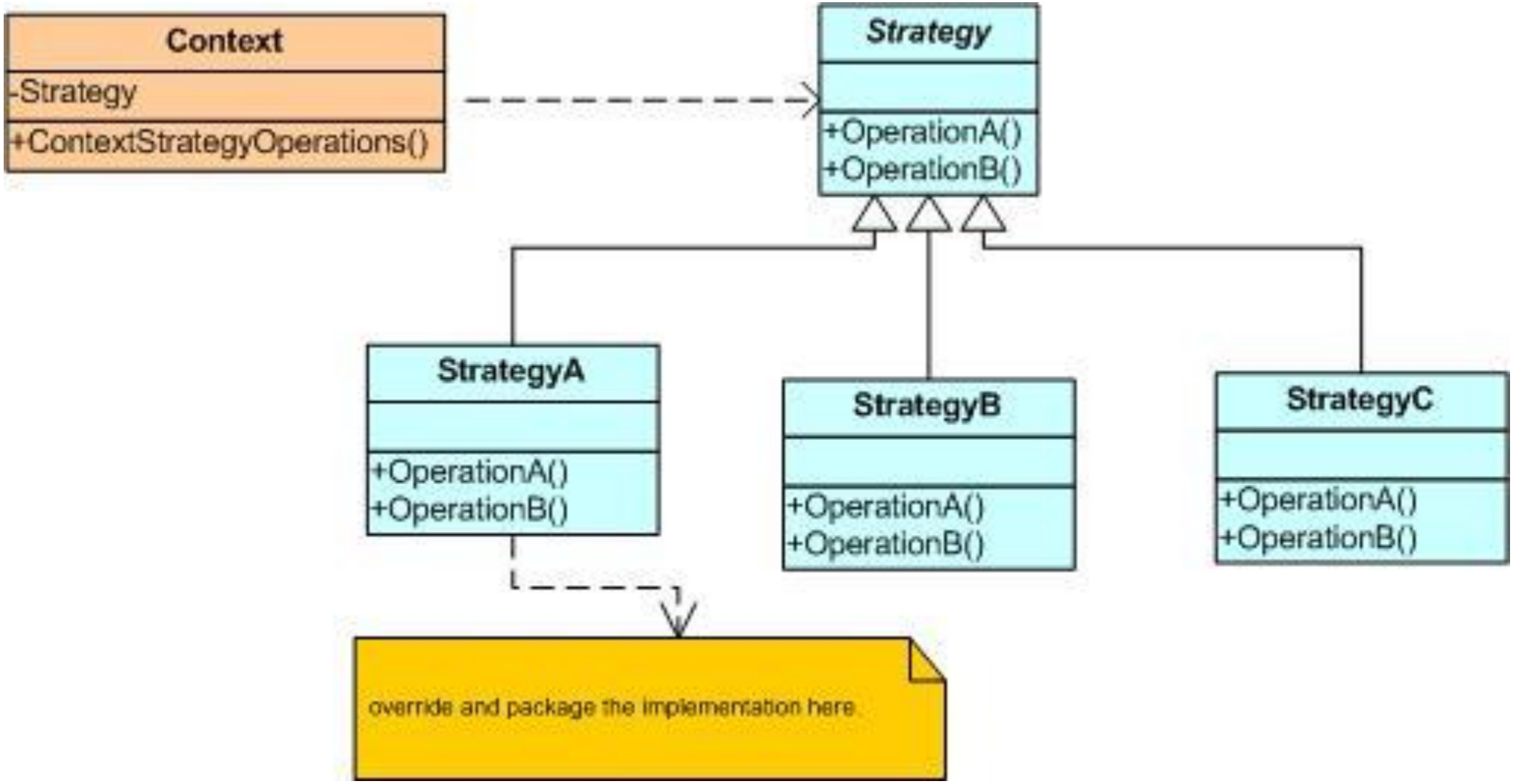
# Strategy

# Strategy

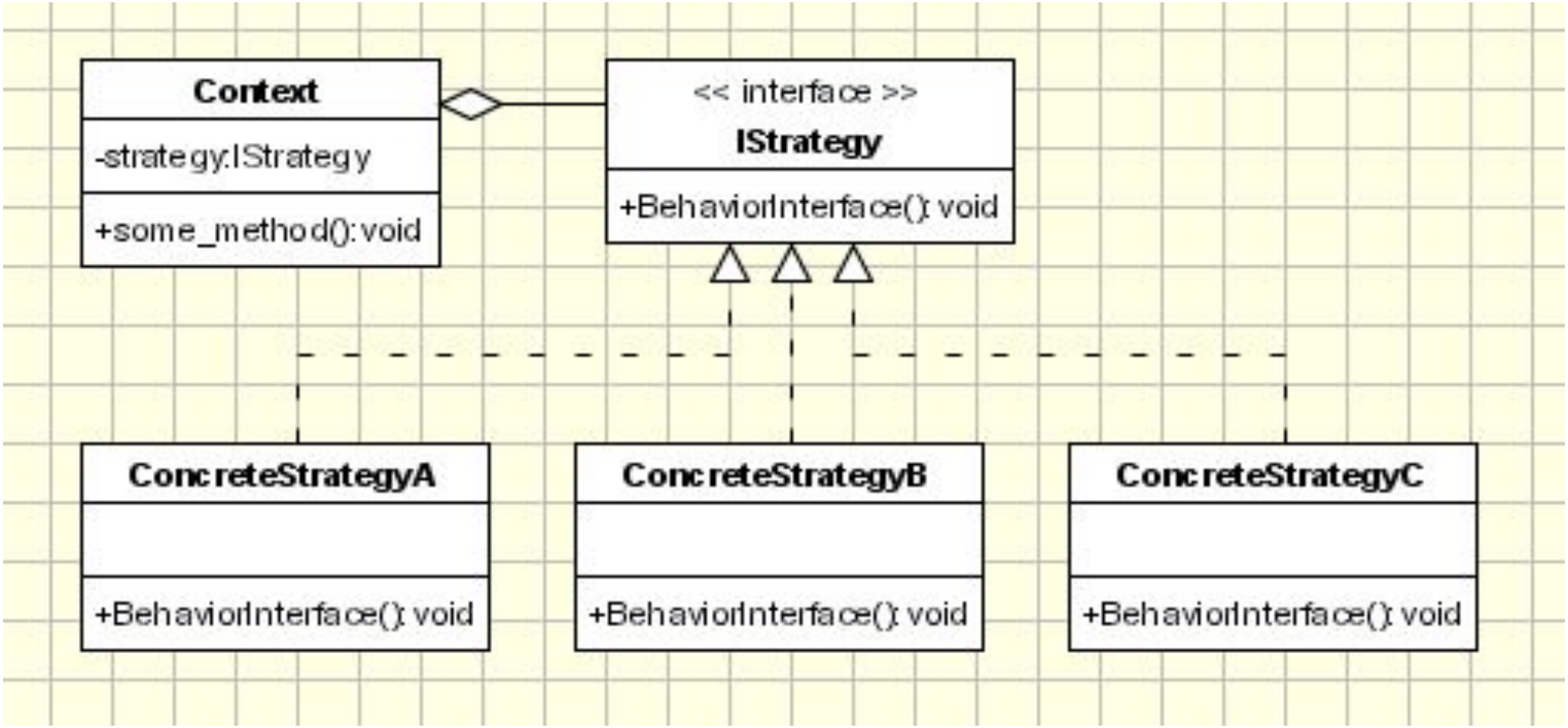
- Encapsular uma família de algoritmos
- Mudar o conteúdo (interior) de uma Classe
- Abstração fica na interface, implementações ficam nas subclasses
- Semelhante a Bridge, mas soluciona problema diferente
  - Bridge queria evitar excesso de classes
  - Strategy quer adicionar flexibilidade a uma classe
- Separa “o que fazer” de “como fazer”

# Strategy

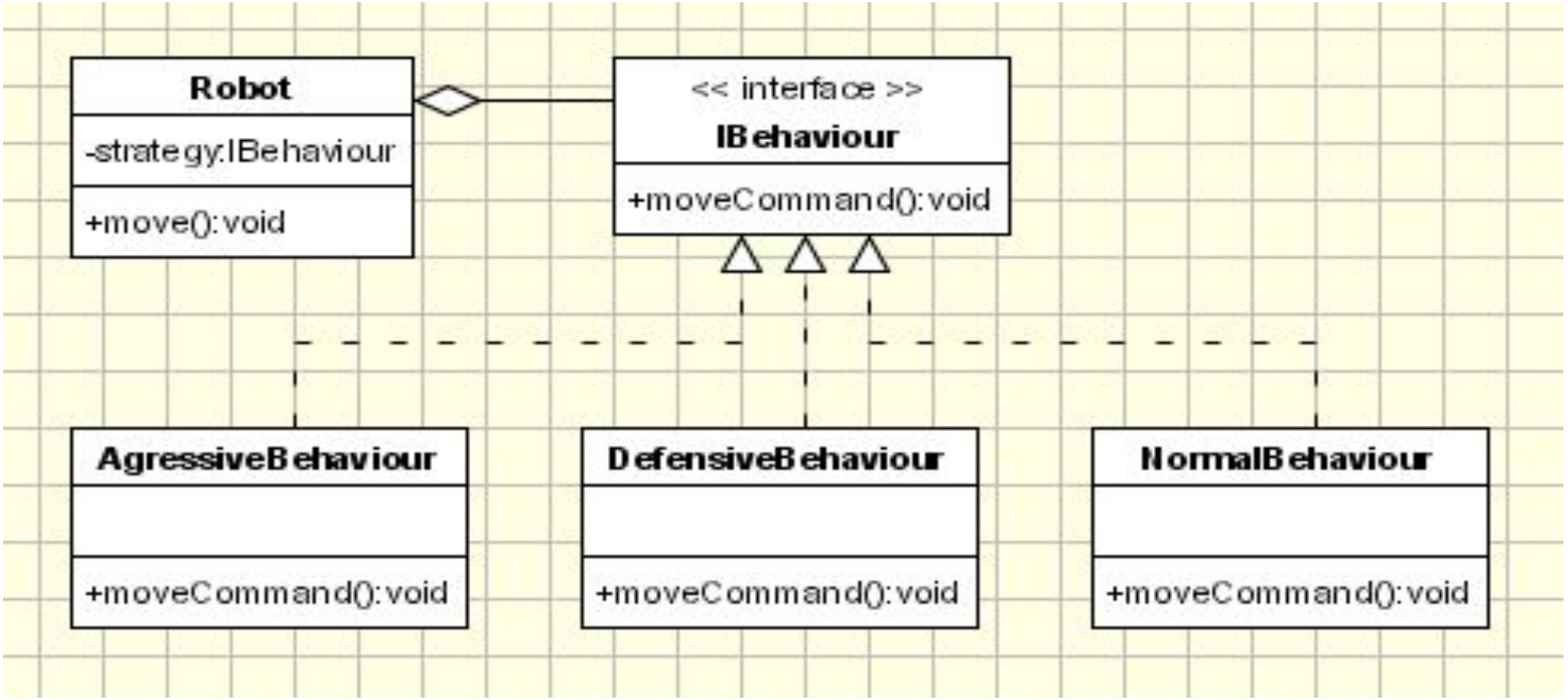

Client
Context
- strategy
Interface
+algorithm()
ImplementationOne
+algorithm()
ImplementationTwo
+algorithm()


Context
-Strategy
+ContextStrategyOperations()
Strategy
+OperationA()
+OperationB()
StrategyA
+OperationA()
+OperationB()
StrategyB
+OperationA()
+OperationB()
StrategyC
+OperationA()
+OperationB()
override and package the implementation here

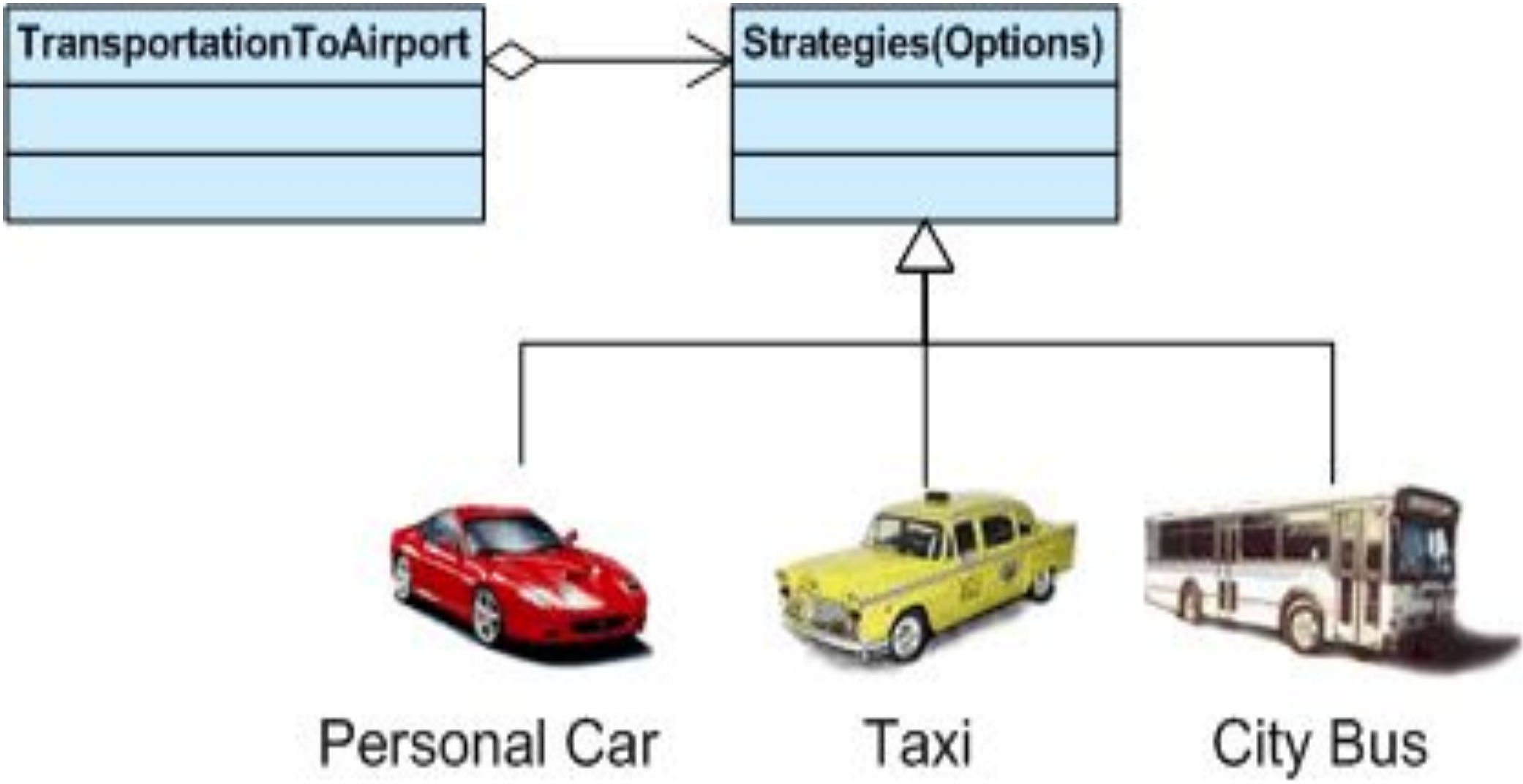
# Strategy


Context
-strategy:IStrategy
+some_method():void
<< interface >>
IStrategy
+BehaviorInterface(): void
ConcreteStrategyA
+BehaviorInterface(): void
ConcreteStrategyB
+BehaviorInterface(): void
ConcreteStrategyC
+BehaviorInterface(): void

# Strategy


Robot
-strategy:IBehaviour
+move():void
<< interface >>
IBehaviour
+moveCommand():void
AgressiveBehaviour
+moveCommand():void
DefensiveBehaviour
+moveCommand():void
NormalBehaviour
+moveCommand():void

# Strategy


TransportationToAirport
Strategies(Options)
Personal Car
Taxi
City Bus

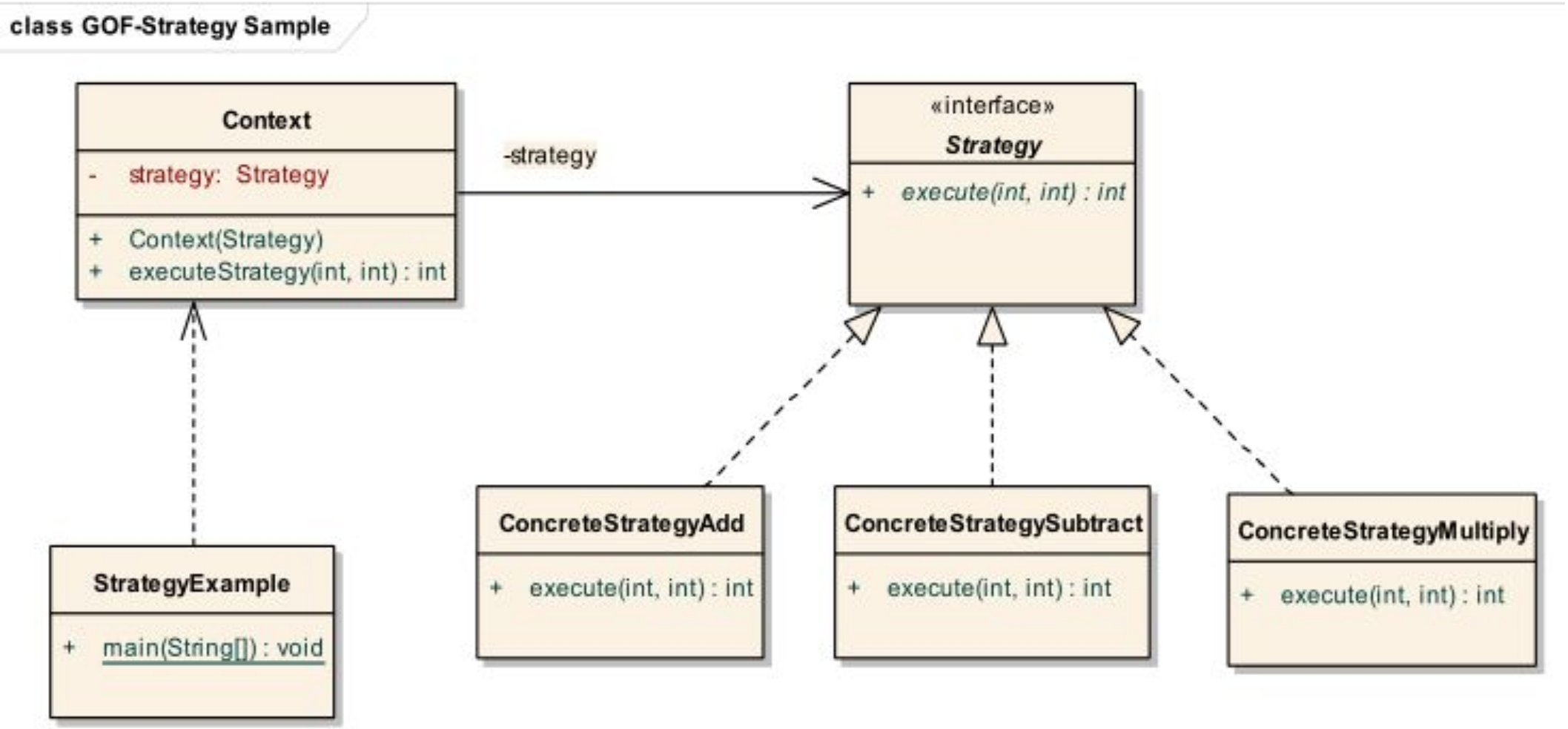
# Strategy

- Participantes
- Strategy: define uma interface comum para todos os algoritmos suportados. Context usa essa interface para chamar o algoritmo específico definido em uma ConcreteStrategy.
- ConcreteStrategy: cada estratégia concreta implementa um algoritmo específico.
- Context: 1) contém referência para um objeto Strategy. 2) pode definir uma interface que deixa Strategy acessar seus dados

# Strategy

- Passos
- Identificar algoritmo (comportamento) que o Cliente quer ter flexibilidade em acessar.
- Especificar a assinatura do algoritmo como uma interface.
- Enterrar as implementações alternativas em classes derivadas.
- Clientes do algoritmo se acoplam à interface.

# Strategy


class GOF-Strategy Sample
Context
- strategy: Strategy
+ Context(Strategy)
+ executeStrategy(int, int) : int
-strategy
«interface»
Strategy
+ execute(int, int) : int
StrategyExample
+ main(String[]) : void
ConcreteStrategyAdd
+ execute(int, int) : int
ConcreteStrategySubtract
+ execute(int, int) : int
ConcreteStrategyMultiply
+ execute(int, int) : int

# Strategy

- Note: Cliente fica exposto a variações de implementação: ruim em alguns casos

# Exercício

- O objetivo dessa atividade é implementar os padrões comportamentais: **Observer, Command** e **Strategy**.
- Considere um problema no domínio do mercado financeiro. Neste cenário existem as ações (**Acao**) e cada ação contém uma lista de Investidores (**Investidor**) que estão associados às me smas. Cada investidor possui um corretor (**AcaoBroker**)
- AcaoBroker é responsável por comprar e vender ações.
- O AcaoBroker recebe os comandos para vender e comprar ações.
- Considere que um mesmo AcaoBroker pode realizar essas operações usando diferentes estratégias.

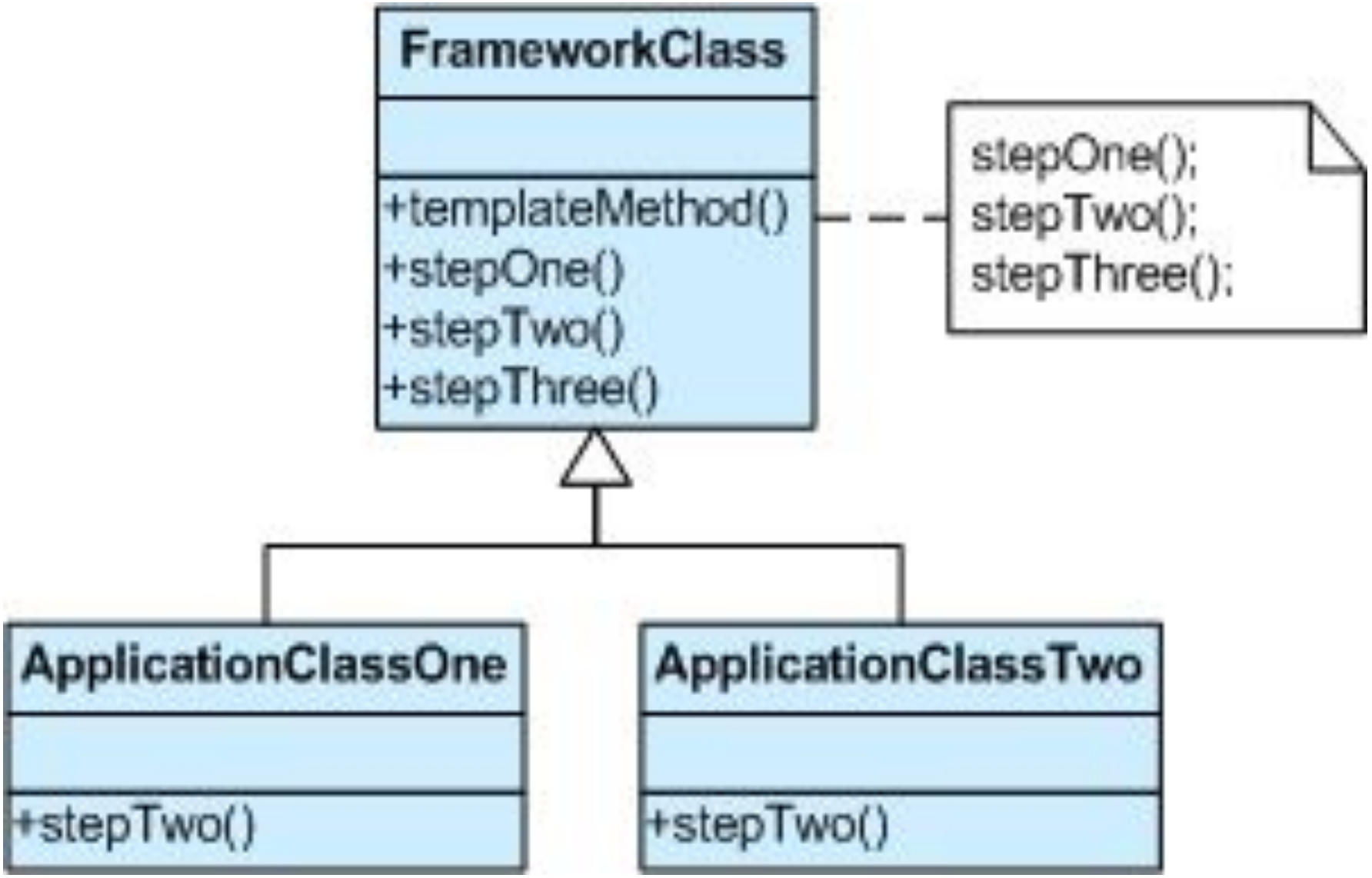
# Exercício

- Faça o projeto usando o padrão **Observer** para os Investidores serem notificados das variações de preço de aus ações.
- Cada investidor poderá definir limites mínimos e máximos no preço de suas ações. Quando o preço máximo for atingido, as ações deverão ser vendidas, ou seja, o AcaoBroker deverá receber o comando Comprar. Da mesma forma com Vender quando atingir o limite máximo.
- Neste mesmo projeto, aplique o padrão **Command** que deverá ser usado pelo AcaoBroker para executar seus comandos.
- Finalmente, utilize o padrão **Strategy** considerando que os comandos de compra e venda podem ser executados de formas diferentes, como pela internet ou por celular.
- Crie um código main simples que exemplifique o uso dos padrões.

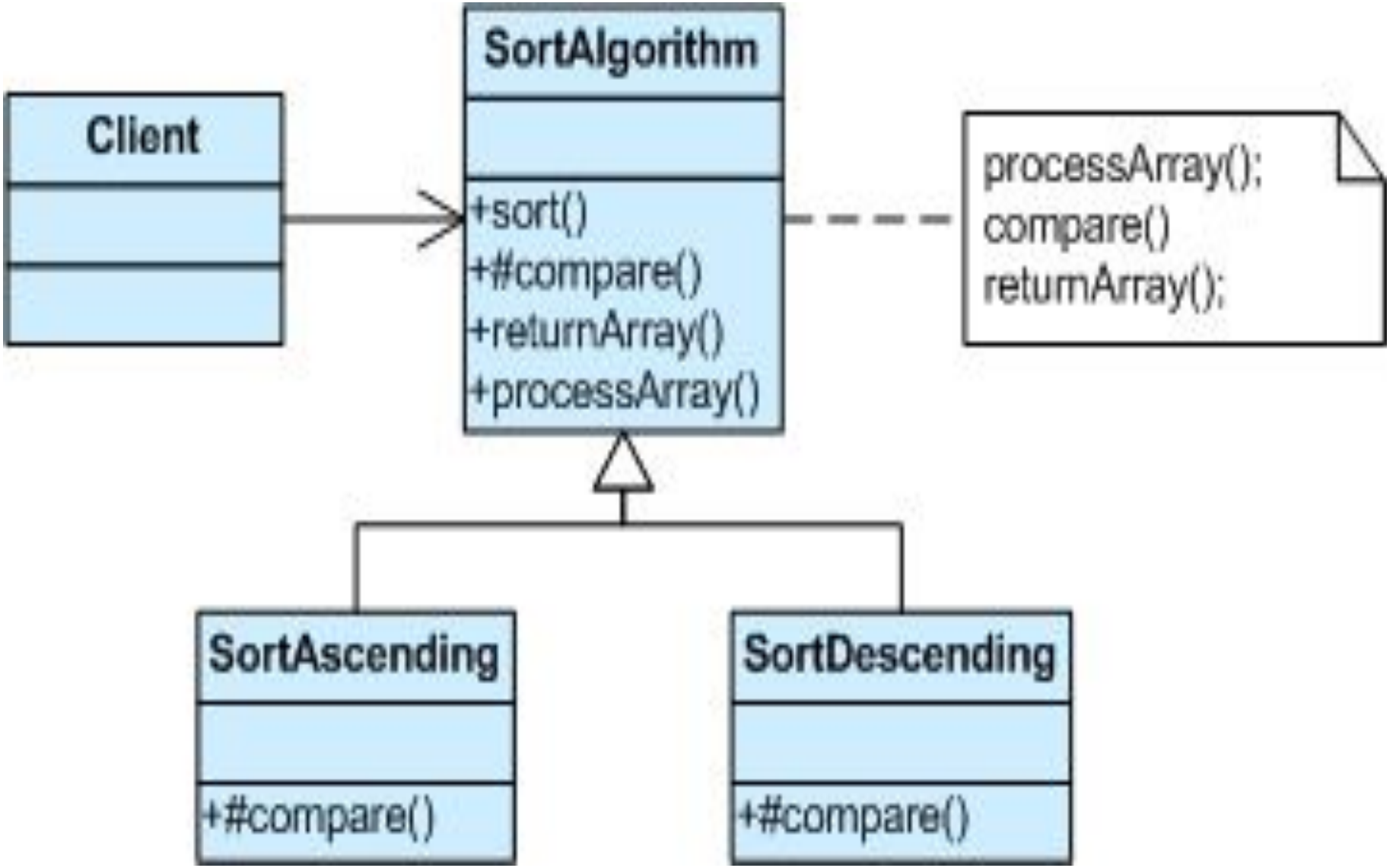
# Template Method

- Define o esqueleto de um algoritmo em uma operação, delegando alguns passo para sub-classes Cliente
- Sub-classe pode redefinir alguns passos sem alterar a estrutura geral do algoritmo
- Superclasse define "placeholders", sub-classes implementam
- Versão mais granular do Strategy: Template Method usa herança para variar parte do algoritmo, Strategy usa delegação para variar o algoritmo todo

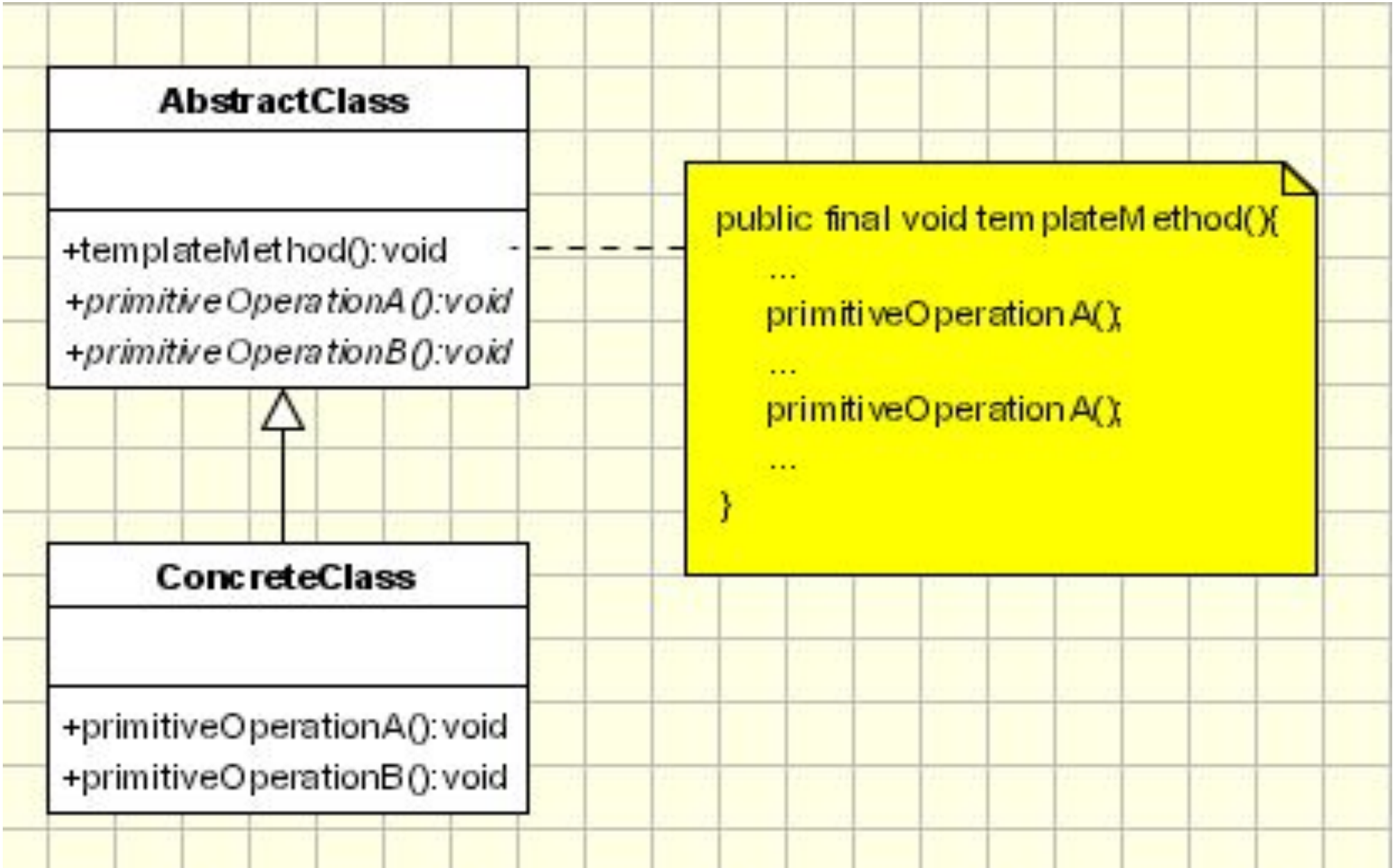
# Template Method



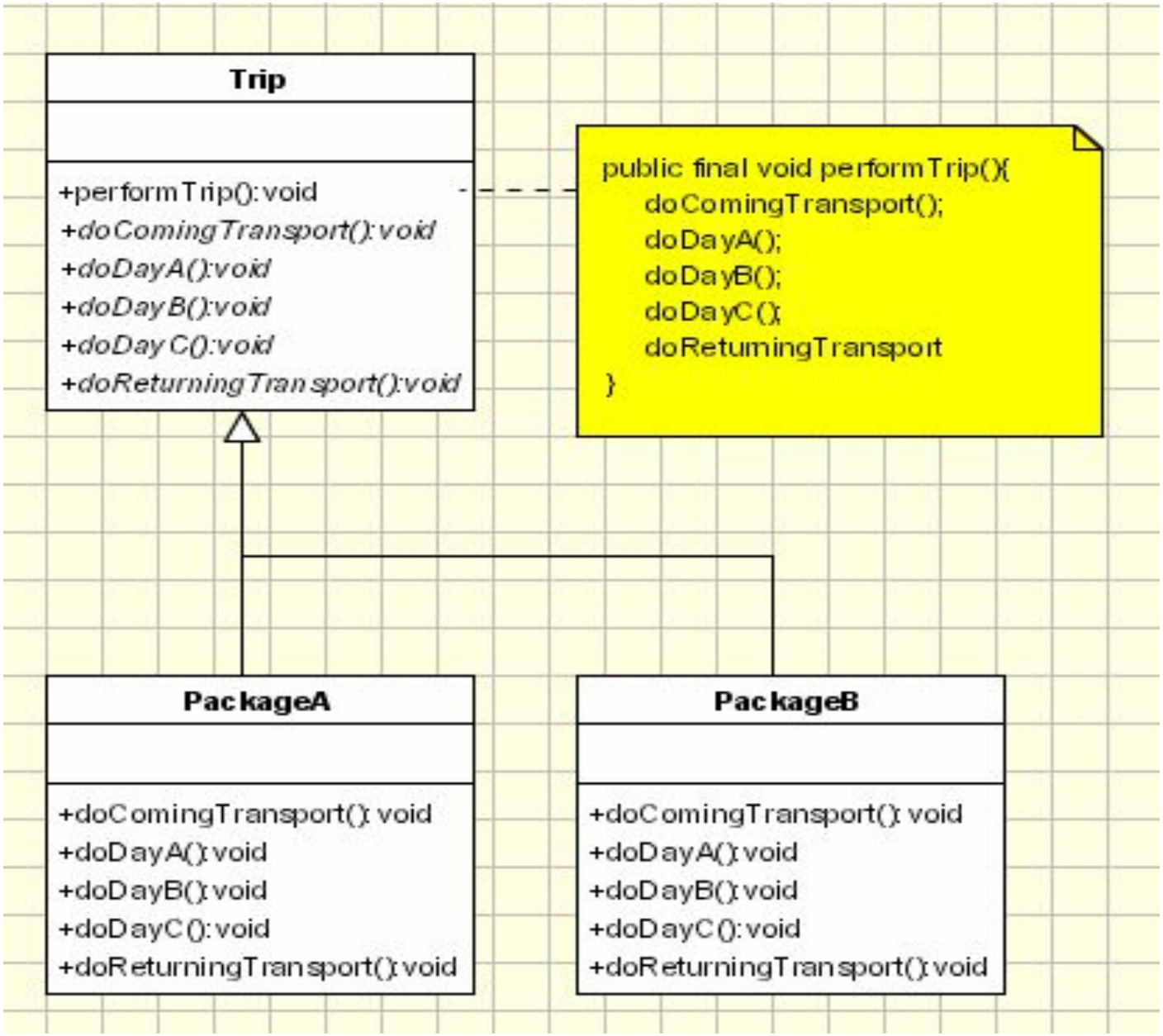
# Template Method



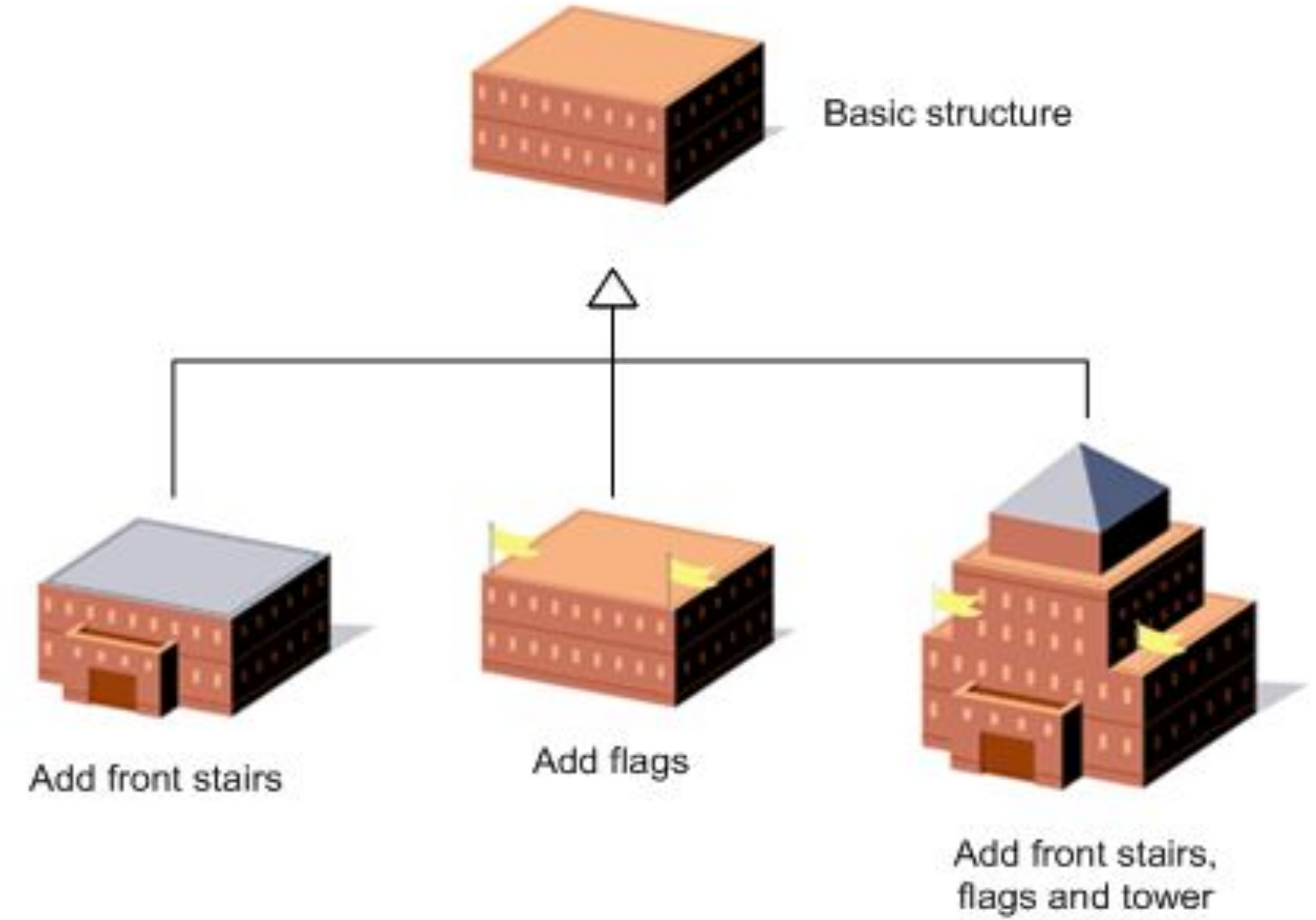
# Template Method



# Template Method



# Template Method



# Template Method

- Versão mais granular do Strategy: Template Method usa herança para variar parte do algoritmo, Strategy usa delegação para variar o algoritmo todo
- Factory Method é uma especialização do Template Method