Mestrado Integrado em Engenharia Informática

Arquiteturas de Software

Design Pattern Command

GRUPO:
Pedro Vale a72925
Rui Freitas a72399

13 de Novembro de 2016

1 Desenvolvimento

Neste trabalho começamos por analisar um conjunto de design patterns que nos podiam ser úteis para o nosso problema. Com a recolha de informação chegamos a conclusão que o design pattern ótimo seria o design pattern comportamental *Comando*. Este padrão resolve o nosso problema, que se assemelha a uma smart home, para saber quando abrir/-fechar os estores e ligar/desligar o ar condicionado.

Para implementar a nossa solução utilizando o padrão *Comando*. Começamos por criar uma interface chamada *Atuador*, esta interface vai ser implementada por todos os comandos que pretendam realizar uma certa ação sobre um determinado aparelho. No nosso caso temos o *ligarArcondiconado* e *desligarArCondiconado* vão aplicar-se ao *arcondicionado* enquanto que o *fecharEstore* e *abrirEstore* aplicam-se ao *estore*. Esta interface contem um método *execute()* para que em cada comando seja executado a respetiva ação para o respetivo aparelho. Em especifico, no caso da class *desligarArcondicionado* que implementa a interface *Atuador*, o método execute consiste em chamar o método *turnOff()* do aparelho de ar condicionado ao qual o comando está associado. Desta forma, torna-se obrigatório que cada comando esteja associado a um aparelho, para assim aplicar a ação pretendida ao respetivo aparelho.

Os aparelhos criados para o nosso sistema foram o *Estore* e o *Arcondicionado* que tem uma estrutura muito basica. São compostos por um identificador(id), util nas situações em que temos varias aparelhos do mesmo tipo, e um booleano que representa o estado do aparelho. Cada aparelho deve ainda implementar as ações necessárias ao seu funcionamento. No caso do Arcondicionado, temos os métodos *turnOn()* que ativa o ar condicionado e o método *turnOff()* que desliga o ar condicionado

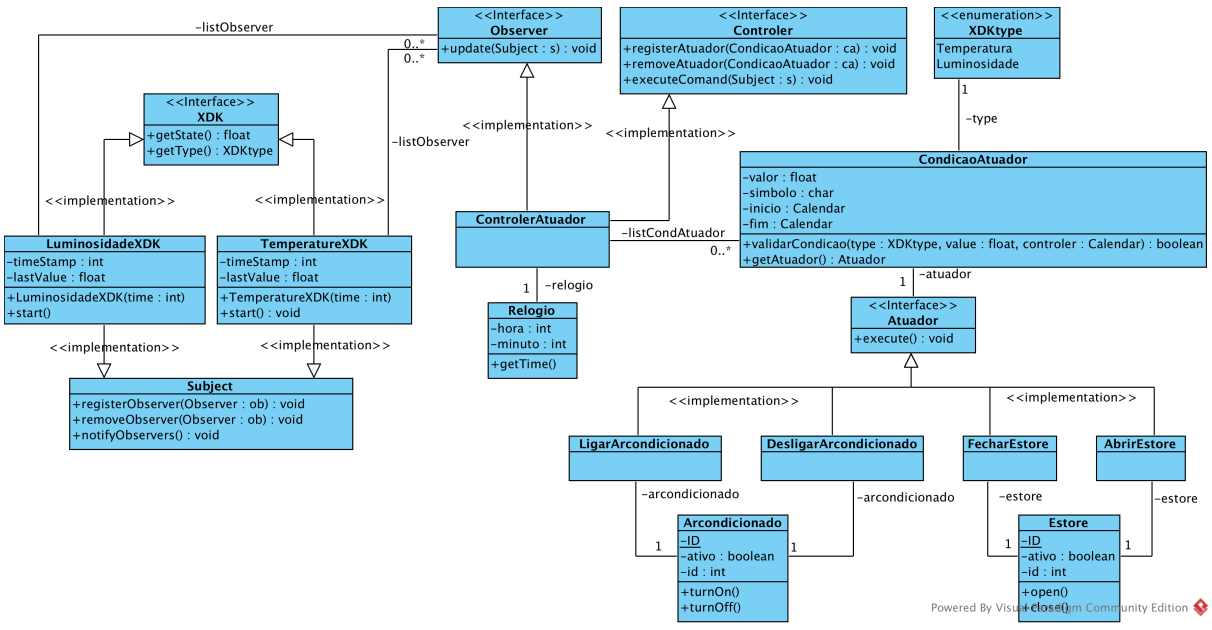

Figura 1: Diagrama de classe da aplicação desenvolvida.

Um comando pode ser executado em diferente condições do sistema, desta forma é necessário agrupar os diversos parâmetros de uma determinado condição para que quando essa condição for verificada o comando seja aplicado. Para este fim foi criada uma classe

chamada *CondicaoAtuador* que irá conter o atuador, o tipo de valor do xdk preciso para este atuador, o símbolo ('>' ou '<'), o valor a partir do qual o atuador ativa, um *Calendar* com a hora de inicio e um com a hora do fim ao qual o atuador vai atuar. Esta classe contem o método que valida se o comando vai realizar-se ou não dependendo da hora a que um atuador quer ativar e valor acima ou baixo do qual ele ativa. Cada atuador utiliza apenas os valores de um *xdk*, para diferenciar os diversos xdk foi criado um enumerador que contem os tipos do sensores existentes. No nosso caso, apenas são usados dois tipos de xdk's temperatura e a luminosidade. Mas se de futuro for necessário acrescentar mais outros tipos de xdk's, apenas necessitamos de mudar neste enumerador ficando o resto do código igual. Para completar esta solução relativa ao tipo de cada xdk foi adicionada á interface xdk que deve ser implementada por todos os objetos que queiram representar um xdk o método *getType()*

De forma a existir uma forma eficaz de gerir os valores lidos pelos xdk's e a respetiva execução dos comandos foi implementado o conceito de um controlador. Assim foi criada uma interface *Controler* que contem os métodos *registerAtuador* para registar as condições atuadores, *removeAtuador* para remover as condições do atuador e o *executeComand* para executar o atuador de forma a ele realizar a sua ação.

Com uma simples analise do problema foi identificado que o controlador tinha as características de observador relativamente aos xdk's, desta forma, foi reutilizado o trabalho anterior com a implementação do design pattern *observer* e assim aplicada á classe *ControlerAtuador* a interface Observer.

Esta classe implementa a interface *Controler* de forma a ter o comportamento desejado para um controlador. Contem uma lista com todas as condições de cada atuador registado. De forma a simular o tempo cada controler tem acesso a uma classe Relogio que implementa a interface *Runnable* para que execute de forma paralela ao programa. A classe relogio tem a funcionalidade de aumentar de 1 em 1 segundo 5 minutos ao relógio, e contem ainda um método *getTime()* para disponibilizar a hora atual do relogio para assim comparamos com o tempo da condição de cada atuador.

Cada vez que um xdk simular um valor vai notificar o *Observer* que no nosso caso é apenas o *ControlerAtuador*, este contem o método *executeComand* para procurar em todos as condições dos atuadores e ver quais se verificam relativamente ao valor dado pelo xdk e das horas que são no momento, caso a condição seja verdadeira o atuador é executado.

Os xdk's, Luminosidade e Temperatura, são reutilizados do projeto anterior contendo da mesma forma a implementaçãoda interface *Subject* do design pattern observer.

2 Conclusão

Na implementação deste exercício de simular uma smart home, foi possível consolidar que a utilização de padrões previamente estabelecidos torna o código mais fácil de interpretar por parte de qualquer programador que tente fazer. Embora existam diferentes padrões, cada um deles resolve um problema. Cada um deles deve ser estudo e aplicado á necessidade que cada um resolve.