Mestrado Integrado em Engenharia Informática

Arquiteturas de Software

Design Pattern Observer

GRUPO:
Pedro Vale a72925
Rui Freitas a72399

19 de Outubro de 2016

1 Desenvolvimento

Neste relatório iremos mencionar as varias soluções que optamos fazer ao longo do trabalho pratico, desde do exercício 1 ate á implementação desejada concebida no exercício 4. Falaremos das varias alterações ao longo dos exercícios e as restrições que obtemos para solucionar o problema de forma correta.

1.1 Exercício 1

Para criar o componente que simulava o xdk que era pedido no exercício 1 criamos uma classe com as 7 variáveis estáticas, sendo que cada uma destas representava um dos 7 sensores contidos no xdk. Para o oitavo sensor, o giroscópio não criamos nenhuma variável, pois na nossa perspetiva os valores são constantemente alterados não tendo qualquer dependência com os valores anteriores. Esta classe foi implementada de forma totalmente estática.

Foram também adicionados 8 métodos públicos que permitem obter a simulação de cada sensor, mais especificamente o valor de cada variável estática. Quando é efetuado o primeiro pedido por um valor, através dos 8 métodos referidos anteriormente, são gerados valores random dentro de um padrão o mais realista possível, para cada variável. Posteriormente, sempre que é pedido um novo valor para um determinado sensor é gerada uma variação e adicionada ao valor inicialmente gerado, podendo esta ser negativa ou mesmo nula. Esta classe irá manter-se inalterada para os restantes exercícios, não necessitando de alterações, podendo desta forma ser reutilizado.

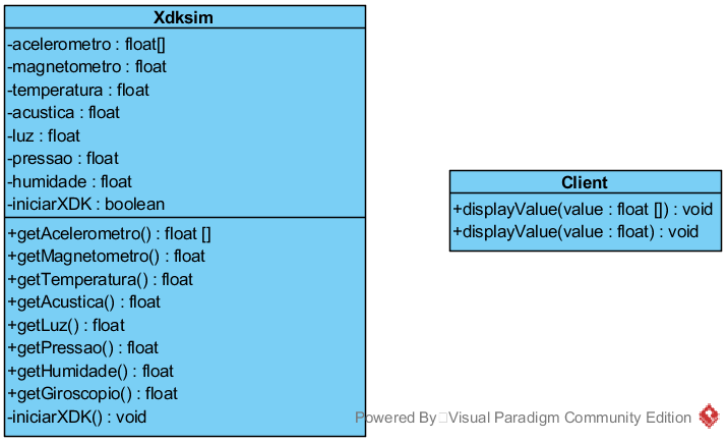

Figura 1: Diagrama de classe do componente xdk implementado(Xdksim).

Na segunda parte deste exercício necessitávamos de um cliente, como tal criamos um cliente que apenas contem uma função implementada que é o update, sendo que este update vai imprimir para o cliente o valor do sensor escolhido pelo mesmo. Para haver passagem do valor desse sensor foi então criado uma função main com vários clientes e o componente xdk fazendo assim a passagem do valor do xdk para o cliente.

1.2 Exercício 2

No exercício 2 foi nos pedido que criássemos um programa que simulasse o xdk de uma forma mais realística, enviando informação ao cliente de x em x tempo. Para tal decidimos começar com uma implementação com sockets, em que tínhamos uma classe ServidorXDK que tinha 8 threads como variáveis, fazendo cada uma delas o papel de um sensor do xdk. Os cliente quando queriam receber a informação de um dado sensor enviavam ao ServidorXDK um registo a mencionar o sensor desejado. Assim, cada cliente erra escalonado para a respetiva Thread que emulava o respetivo sensor. Desta forma, cada cliente registado recebia com uma determinada cadencia, definida inicialmente, um valor obtido na Thread do sensor recorrendo ao componente definido no exercício 1. Estes valores eram impressos para o output do cliente.

Numa aula pratica foi mencionado que não podíamos usar uma implementação com sockets, tivemos então de utilizar uma nova forma de resolver o problema. Optamos por uma solução em que a forma de comunicação entre o simulador e o cliente seria idêntica a solução realizada no 1º exercício. Assim em vez de termos sockets adicionados na lista de cada sensor, teríamos então o objeto do cliente, mudando assim o que fazíamos na versão anterior, permitindo na mesma que o sensor envie a informação para o cliente.

Foi ainda adicionado outro tipo de cliente ao nosso sistema que é a base de dados, este novo tipo de cliente contem uma lista com todos os valores encaminhados pelo sensor pretendido, simulando assim o armazenamento da informação.

1.3 Exercício 3

O exercício 3 é o mesmo problema que o anterior mas usando o padrão observador referido nas aulas teorias. Este padrão tem como perspetiva que existe um dado observador e algo observável. No nosso caso, tínhamos os sensores do xdk que eram observados pelos clientes(observadores). Para implementar este padrão, começamos pela criação de duas interfaces Subject e Observer que identificação os objetos observáveis e os observadores respetivamente.

A interface Subject é constituída por métodos que façam estritamente o necessário para relacionar um observador com o observável, mais concretamente:.

- registerObserver $->$ registar um observador
- removeObserver $->$ remover um observador
- notifyObservers $->$ notificar todos os observadores registados que existe uma alteração no estado do objeto observável.

Desta forma, é possível que todos os objetos que desejem se observados contenham a mesma forma de um observador os observar.

Na perspetiva do observável não é importante que exista qualquer diferença entre os observadores. A interface Observer será utilizada pelos clientes de forma a implementar um único método(update) que será utilizado pelo observável para os notificar. Com este método o observável envia-se ao observador para que este tenha depois consiga visualizar o seu estado. Desta forma, conseguimos resolver o problema que podia existir, em enviar diferentes tipos de valor, ou no adicionar mais valores a serem enviados pelos sensores.

Sendo todos os observadores identificado pela interface Observer tornasse mais fácil ao observável guardar uma coleção de todos os observadores.

Com todas estas facilidades conseguidas com o padrão observador, resolvemos criar uma classe para cada sensor. Assim passamos a conter no nosso projeto as classes AccelerometerXDK e TemperatureXDK que implementam a interface Subject. Cada uma destas classe contem também um método getState e getName é utilizado pelos clientes para obterem as informações necessárias do Subject. Cada um deste sensores, apos ser criado, cria uma Thread para implementar a função de com uma determinada cadencia enviar para os clientes registados os novos valores obtidos. Em relação aos nossos clientes, para alem de fazermos a implementação da interface Observer foi necessário no MostradorObserver realizar uma diferenciação entre os Subjects pois era o único que podia receber valores dos dois sensores.

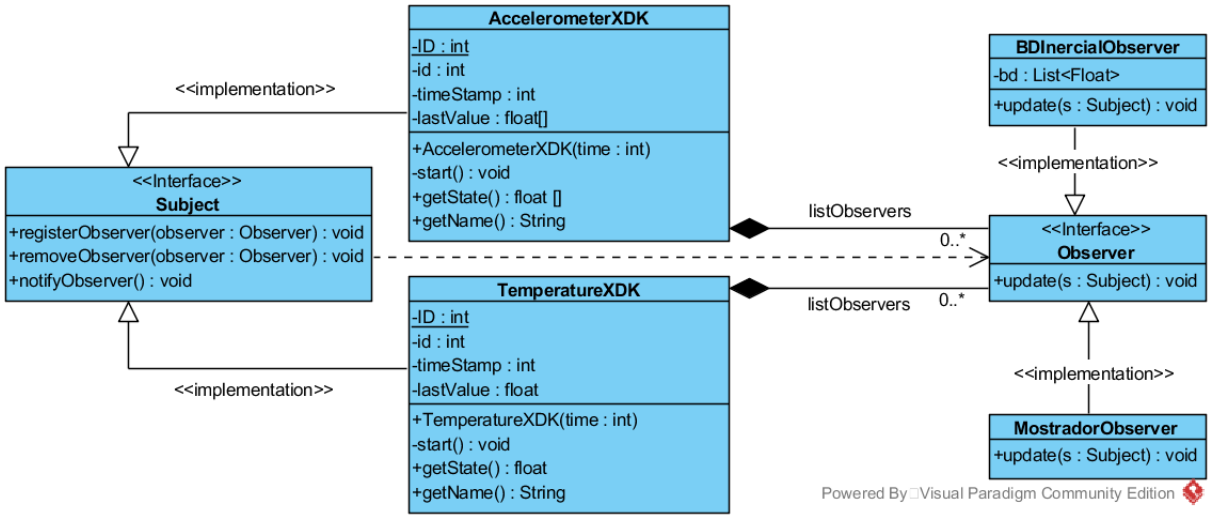

Figura 2: Diagrama de classes da implementação do exercício 3.

1.4 Exercício 4

O desenvolvimento do exercício 4 foi relativamente simples, devido à implementação realizada no exercício 3. A maior alteração foi a adição de um novo cliente ao sistema chamado NoiseMonitor, este cliente terá um map com uma string que é o nome do sensor e uma lista com todos valores que esse sensor enviou até ao momento, podendo assim o cliente receber dos 3 sensores noise, sabendo sempre qual enviou um determinado valor. Sempre que a função update é invocada vai imprimir para o output o nome do sensor, a hora do último valor recebido e os valores todos registados até ao momento, pela ordem de chegada(aparecendo em primeiro lugar os mais recentes).

Criamos a classe NoiseXDK visto que apenas precisamos da temperatura e acelerometro nos exercícios anteriores. Esta classe implementa a interface Subject, tendo o mesmo funcionamento que as anteriores mudando apenas o nome e o seu get, visto que recolhe apenas os dados da acústica para enviar para os seus clientes.

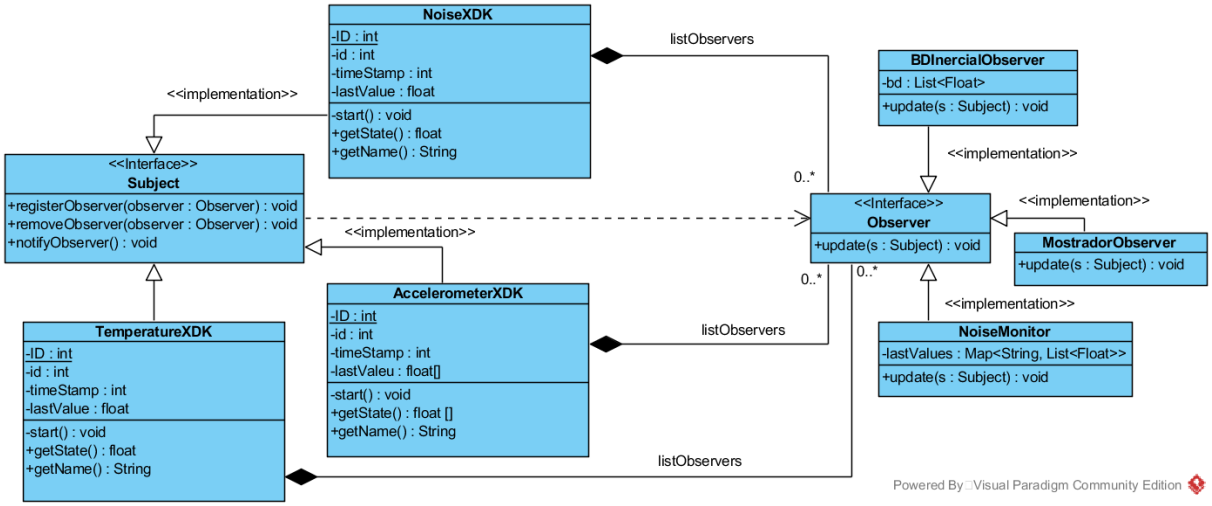

Figura 3: Diagrama de classes da implementação do exercício 4.

2 Conclusão

Na implementação do pequeno exercício de simular o xdk, foi possível observar que utilizar padrões previamente estabelecidos torna o código mais fácil de interpretar por parte de qualquer programador que tente fazer. No nosso caso, a utilização do padrão observador concretiza as operação que queríamos concretizar no nosso programar, observar e ser observado.