Dicas sobre Orientação para Objetos e filas de mensageria

Glauco Reis
Explicando tecnologia com metáforas. Mestre em Inteligência Artificial e Aprendiz na vida, aprendendo todos os dias com vocês. IBM Technical Sales...

Published Jul 21, 2022

Houve uma época em que eu programava todo dia, de manhã até a noite. Hoje, como parte das minhas atividades eu não programo mais, ou melhor, programo por passatempo ou para tentar resolver algum problema. Algum tempo atrás acabei codificando por passatempo o Microinvader, um videogame do tipo space invaders criado com Microsserviços. Eu uso ele para muitas demonstrações aqui

*demonstrando uma ferramenta da IBM com o jogo, me dá a sensação de que as pessoas não conseguem associar um jogo com um programa "**sério**", ou as vezes dá aquela impressão de que "**um jogo não representa um programa de computador que fazemos aqui na empresa!!!**".*

*Mas outro dia aconteceu uma coisa especial, e preciso contar uma pequena história para entrar no contexto do programa que irei mostrar aqui. Fui envolvido em um cenário onde o cliente estava testando o MQ Series da IBM, e ele alegava que o IBM MQ "**não performava**". Baixei uma imagem Docker do MQ na minha máquina e comecei a fazer testes. Pedi ao cliente que me enviasse o código fonte dele, mas não "**me ambientei com ele**". Simplesmente não consegui compilar os códigos. Então resolvi partir para o JMeter. Peguei um script Groovy (**que é muito similar ao Java em sintaxe**) e comecei a estressar o MQ com o JMeter. Fiquei decepcionado quando percebi que ele não performou na minha máquina também. Mudei insistentemente coisas no Script, tentando reduzir tempos e rodar o mais rápido que podia. Mas não ia mesmo. Bom, depois de inúmeras tentativas, ficamos imaginando se não poderia ser o JMeter que por algum motivo desconhecido pudesse não estar performando. Como uma medida desesperada, algo sussurrou no meu ouvido: "**ei, cara, você já foi programador. Não seja um covarde!!!**", e em um ímpeto transformei o script Groovy em um programa Java. Um programa simples, que iniciava um conjunto de Threads e estabelecia a comunicação com o MQ mandando mensagens que nem um maluco. O resultado foi inacreditável, porque passou de 1500 mensagens por segundo para números próximos a 10.000 mensagens por segundo, com picos de 20.000. Enquanto criava o programa, fui pensando formas de não criar um **pastel** que seria jogado fora, mas algo que pudesse ser reutilizado em outros cenários. Mas, será que um*

isto vou explorar aqui neste artigo. Vou contar as decisões que tomei para chegar ao código final do programa. Lógico que existem inúmeras formas de se chegar ao mesmo resultado, e vou mostrar minhas decisões neste mini projeto.

Primeiro configurações

Uma das primeiras coisas que qualquer programa precisa ter é configurações. Elas podem estar ***hardcoded*** *no programa, mas é uma péssima decisão porque qualquer alteração força a uma recompilação. Também os parâmetros podem ser passados por linha de comando, o que facilita o uso por ferramentas de automação, tipo Jenkins, mas minha escolha neste caso foi manter TODAS as configurações em um arquivo Properties. Quando o programa executa inicialmente, ele carrega as propriedades e deixa disponível em uma classe onde pode-se buscar valores por chave. Aqui foram tomadas duas decisões. TODAS as propriedades do programa estavam em um único arquivo. Foi uma decisão de arquitetura. E portanto inúmeras partes do programa precisariam acessar as propriedades. Mas como fazer isto? Iniciar uma vez e passar toda vez a instância da classe para todos os outros objetos é ruim (muitos parametros passados). Criar uma nova instância em cada lugar onde fosse necessário era igualmente ruim (carregaria muitas vezes do disco). A decisão foi então utilizar um design pattern criacional chamado singleton. Ele é muito simples :*

Não foi fornecido texto alternativo para esta imagem

*Em Java isto pode ser conseguido criando-se uma variável estática e cria-se um método (****getSingleton****) também estático. Ele pode ser chamado de qualquer lugar, sem precisar instanciar o* ***PropertiesCollection****. Quando se chama ele,*

criado previamente. Desta forma, economizamos parâmetros, e qualquer outra classe que precisar de uma informação, é só chamar o método ***getSingleton*** *e pegar o valor de que precisa.* ***O Singleton garante que uma única instancia de um objeto vai existir em um sistema.*** *Outra decisão, que não identifiquei no primeiro momento, foi a de uniformizar as coleções. Existem inúmeras estruturas de dados possíveis para uso. Pilhas, Filas, FIFO, LIFO, cada uma otimizada para uma finalidade. No tratamento interno se usa a mais eficiente. Eu decidi exportar TODAS as coleções de objetos como arrays estáticos em Java. Isto não é muito eficiente, mas tem uma razão que vai ficar mais clara mais adiante.*

Não foi fornecido texto alternativo para esta imagem

Qual a entidade mais importante do programa?

*Eu já explorei em outro artigo que considero o polimorfismo o conceito mais importante da orientação para Objetos. Devemos encontrar os objetos que possamos lidar de forma agnóstica. Se você ler o GOF (****Design Patterns****) verá que a maioria dos design patterns foram criados para lidar com objetos polimórficos. No meu videogame, transformei tudo em objetos que aparecem na tela.* ***Inimigos****,* ***Bombas****, todos eles são tratados de uma mesma forma. Se você deseja que um objeto se mova basta chamar um único método, sem saber o que é o objeto.* ***Mas se mover para onde? Isto é uma decisão do objeto. Ele deve saber para onde se mover.*** *Mas um programa com a simplicidade deste seria meio impossível encontrar objetos para hierarquizar, não é mesmo? Afinal a única coisa que ele executava é mandar mensagens que nem um maluco para o MQ Series. Vamos ver? Uma fila de Mensageria envia mensagens para*

*programas? Entendi que o conceito mais importante do programa era lidar com coleções de de "**entidades**" que estariam estressando alguma solução. Então decidi abstrair cada "**unidade de estresse**". Ficou algo assim:*

Não foi fornecido texto alternativo para esta imagem

*Hoje é uma interface genérica (**confesso não faz muitas coisas**) e uma classe concreta para Enviar mensagens e outra para Receber mensagens. E por tabela, foi criado um "**Factory**", outro pattern importante. Ele faz dois papeis, o primeiro é decidir qual objeto será criado. **Quantos Senders o programa irá criar? Quantos Recievers? Crio eles alternadamente ou todos de um tipo primeiro e todos do outro depois?** O Factory toma a decisão de qual criar, em qual ordem. E o mais importante, ele retorna sempre uma mesma interface, no desenho OneThreadMQ (entendo que o nome deveria ser mais agnóstico). Ou seja, Main não sabe o que é OndeSender nem o que é OneReceiver. Em outro artigo usei a metáfora de um baile de máscaras, onde todos usam uma mesma fantasia que esconde o corpo todo. O tipo de objeto ou pessoa, neste caso, fica contido dentro da fantasia. Esta técnica foi utilizada em La Casa de Papel. Leia isto aqui.*

Não foi fornecido texto alternativo para esta imagem

Neste caso o algoritmo é simples, só recebo um número e escolho qual objeto criar. Gente, mas para que esta parafernalha toda ? **Imagine que amanhã você deseja criar um outro tipo de Objeto (talvez nem Sender nem Receiver, mas um objeto que valida se a conexão está ativa - Validate). Basta criar o Objeto, extendendo da interface (OneThreadMQ) e mudar o Factory para criar o objeto**

necessariamente o programa mais simples de ser codificado, mas o principal objetivo é facilitar a evolução e minimizar impactos na mudança.

Quando criei os Objetos OneSender e OneReceiver percebi que tinham um monte de variáveis que eram comum aos dois. Não faz sentido manter as mesmas variáveis nos dois locais (em ***Sender*** *e em* ***Receiver****). Uma decisão poderia ser levar tudo que é comum para a classe pai (****OneThreadMQ****). Mas também podemos usar algo chamado composição, que é Adicionar classes dentro de um objeto, que "****compõe ou confere****" mais funcionalidades a ele. Como que uma árvore de natal, onde você pode adicionar piscas, enfeites e torná-lo mais completo adicionando partes. Um benefício do Composite neste caso, é que se amanhã decidirmos criar um Objeto que lide com TCP/IP, o conteúdo do que agregamos pode ser diferente para objetos específicos. Colocamos em classes pai somente aquilo que é comum a todos os objetos.*

Não foi fornecido texto alternativo para esta imagem

Não foi fornecido texto alternativo para esta imagem

O mecanismo de composição é uma forma de atribuir novas funcionalidades ao programa sem usar herança. Na verdade, algumas linguagens usam a composição para implementar herança. *Um ultimo requinte que acabei fazendo no programa foi de flexibilizar a impressão dos resultados.* ***Eu queria poder mudar o que fosse impresso na tela, mas sem ter que recompilar tudo.*** *Gostaria que fosse algo externo ao programa. Pensei em várias alternativas, uma delas usar o mesmo mecanismo de compilação de páginas JSP. Acabei recaindo para uma rotina simples incorporada nos Java mais recentes, que é permitir executar*

Glauco Reis

Não foi fornecido texto alternativo para esta imagem

*Isto permitiu que fossem criadas três rotinas externas, uma para inicialização, outra para finalização e mais uma cada vez que uma estatística deveria ser gerada. O princípio é parecido com o Arduino, que esconde a complexidade com rotinas simples de inicialização e loop para execução. Todas as rotinas recebem quatro parâmetros (**tenho dúvidas se foi a melhor escolha, sabe?**), com todos os dados das conexões, os erros acumulados, todos os resultados com os tempos dos envios e as propriedades que vieram do arquivo de configuração. Todos estes objetos tem um método **getArray()**, que retornam todas as instâncias coletadas e um método **clear()** que limpa a coleção. Desta forma, com arquivos externos é possível gerar uma impressão na tela que pode variar desde uma geração de CSV até gerar um HTML ou mesmo escrever texto na tela. Dentro do arquivo de configuração tem um parâmetro "**scripts**" que indica quais "formatadores" serão utilizados para apresentar as informações na saída. Mas como ficam fora do java, e são interpretados, podem ser modificados facilmente para gerar outros resultados.*

Não foi fornecido texto alternativo para esta imagem

Como ultima sofisticação, já que o programa está interpretando códigos externos, coloquei comandos que poderiam ser criados e executados quando o programa estiver em execução. Todo comando tem dois métodos, um chamado help que mostra o que ele faz, e outro execute que faz a ação. Se você digitar help quando o programa está em execução ele navega no diretório e dinamicamente carrega todos os helps de todos os comandos. Olha só :

Agora é possível gerar novos comandos, comente criando um arquivo javacript e colocando dentro do diretório **commands**. Basta que tenha o método execute, que receba os parâmetros **connections**, **errors**, **results** e **properties** e um método help para explicar o que ele faz.

Não foi fornecido texto alternativo para esta imagem

Lições aprendidas (ou relembradas)

Depois que o programa está criado, ele gerou uma estrutura igual a esta aqui :

Não foi fornecido texto alternativo para esta imagem

Eu não quero chover no molhado não, mas se imagine como um desenvolvedor que não participou do projeto e está sentado a primeira vez na frente dos códigos. O que você acha que explica melhor o programa, em um primeiro momento? A lista de classes da esquerda ou o diagrama de classes da direita ? *Pois é, artefatos foram criados para simplificar o entendimento, e não complica-lo. Aqueles que acham que um papel de pão ou um postit definem a explicação do programa estão enganados.*

O segundo aspecto é que, olhando para um programa que utilizou princípios fundamentais de OOP, algumas coisas são feitas como que uma infraestrutura do programa, e de forma colaborativa, afetam o todo. Por exemplo, as classes ***OneSender*** *e* ***OneReceiver*** *são usadas de forma colaborativa por todos os outros códigos. Olhando para outro projeto de sucesso, o IDE Eclipse, existe toda uma parte comum que foi criada para que novos blocos fossem criados acima dele, e dificilmente seria possível por pessoas isoladas sem um claro*

atividades isoladas.

O projeto está aqui neste link : https://github.com/gsreis/gmequerv2

Você pode encontrar outros artigos meus aqui neste link

E tem alguns artigos específicos sobre orientação para objetos, que é algo de que gosto demais de falar e pensar sobre !!!!

- **Quando decisões são ruins em programação**
- **Quando decisões são ruins em programação**
- **Polimorfismo e os power rangers**
- **Herança, Múltipla Herança e a metáfora dos papeis rasgados**
- **Linguagens de programação (ligeiramente) comparadas**
- **Tetris , Orientação para objetos e um incrível baile de máscaras !**
- **Circuit Breaker e MicroInvader**
- **Uma paleta sobre OO e Polimorfismo**
- **A Catedral e o Bazar em Java x Node.JS**
- **Mais um pouco sobre linguagens de programação**
- **Porque usar 12Factor ?**

- **12Factor, o universo e tudo mais**
- **Design Patterns - de 1994 para o futuro**
- **Design Patterns Criacionais by Marvin and BB8**
- **Design Patterns estruturais by Marvin and BB8**

Obrigado e até a próxima !!!!!

#glaucoreis #technology #innovation #futurism #ai #oop #objectorientation #RUP #designpatterns

Glauco Reis

Aplicações Cloud Native, Nostradamus...
Jul 15, 2022

kafka e o conflito de gerações
Jul 2, 2022

Sobre martelos e pregos na IA
Jun 29, 2022

Explore topics

- Starting a job
- Salary and Compensation
- Remote Work
- Work from Home
- Internships
- Retirement
- Freelancer