Trabalho de Síntese: Object Pool

Ricardo Jardim[2040416], Leonardo Abreu[2067513], João Santos[2095415], and José Freitas[2041716]

Universidade da Madeira, Madeira, PT

Abstract. Nesta síntese iremos realizar uma descrição do padrão de desenho *Object Pool*, também conhecido por *Pooling*, onde seguimos a estrutura utilizada no livro do GoF.

Keywords: padrões · object pool · pooling · reutilizar · libertar · adquirir.

1 Nome

O padrão *object pooling* descreve como evitar aquisições e libertações de recursos de custos elevados através da reciclagem de recursos que não necessita.

2 Contexto

O desempenho é um ponto fulcral durante o processo de desenvolvimento software e a criação de objetos (instanciação de classes) é um processo dispendioso. Enquanto o padrão *Prototype* ajuda a melhorar o desempenho através da clonagem dos objetos, o padrão *Object pool* oferece um mecanismo de reutilização de objetos que são dispendiosos de criar.

Quando é necessário trabalhar com um grande número de objetos que são particularmente caros de instanciar e que cada objeto é necessário apenas por um curto período de tempo, o desempenho de toda uma aplicação pode ser afetado adversamente. Um padrão *Object pool* pode ser considerado desejável em casos como estes.[7]

Uma das situação em que este padrão é usado na sua melhor capacidade é nas várias classes padrão do *NET Framework*, como por exemplo o *.NET Framework Data Provider for SQL Server*. Como as conexões da base de dados do SQL*Server* podem ser lentas de criar, uma *pool* de conexões é mantida, quando uma conexão fecha, na verdade ela não abandona o link para o SQL *Server*, [1] invés disso, a conexão é mantida numa *pool* da qual ela pode ser recuperada ao solicitar uma nova conexão. Isto aumenta substancialmente a velocidade e desempenho das operações. [8]

3 Problema

Os *Object pools* (também conhecidos como *resource pools*) são usados para gerir caches de objetos. Um cliente com acesso a uma *pool* de objetos pode evitar a

criação de um novo objeto simplesmente pedindo a essa mesma *pool* um que já tenha sido instanciado. Geralmente o *Object pool* possui uma *pool* em crescimento, ou seja, está irá criar novos objetos se estiver vazia, ou então restringe o número máximo de objetos criados.

É desejável manter todos os objetos reutilizáveis que não estão actualmente em uso na mesma *pool* para que eles possam ser geridos por uma política coerente. Para conseguir isto, a classe *Object pool* é desenhada para ser uma classe Singleton, de modo a garantir a existência de apenas uma instância por classe.

4 Solução

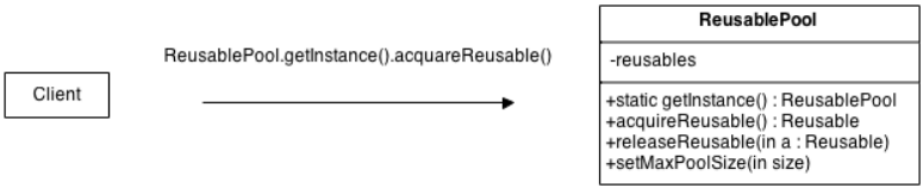

Fig. 1. Object Pool UML

O diagrama UML mostra-nos a estrutura do padrão *Object pool*, onde é possível verificar a sua influência em outros padrões, como por exemplo *singleton* através do método *getInstance()* para criar um objeto da classe *ReusablePool*. O método *acquire()*, é responsável por criar o objeto e armazená-lo na *pool* após a sua utilização. Finalmente o método *release()*, é responsável por libertar o objeto, e inseri-lo nos objetos disponíveis para serem utilizados. [1]

O cliente invoca o método *acquire()* quando precisa de utilizar um objecto reutilizável presente na *pool*. A *ReusablePool* mantém uma colecção de objectos reutilizáveis que não estão em uso com o objetivo de retorná-los quando necessário.[8]

Se houver algum objeto reutilizável na *pool* quando o método *acquire()* é chamado, este remove um objeto da *pool* e devolve-o. Se a *pool* estiver vazia, se possível cria um novo objeto, caso contrário espera até que um objeto seja devolvido à colecção.[8]

Em muitas aplicações do padrão *Object pool*, há razões para limitar o número total de objetos reutilizáveis que podem existir. A classe *ReusablePool* terá um método para especificar o número máximo de objetos a serem criados. Esse método é indicado no UML como *setMaxPoolSize()*. [8]

5 Consequências

Efeitos negativos e positivos do efeito do padrão.

5.1 Vantagens

Performance Em casos onde um dado recurso já foi adquiridos pela *pool*, este fornece vantagens a nível de performance, pois reduz o os custos e tempo necessário para obter e reutilizar recursos, ou seja, tem como objetivo armazenar na *pool* o maior número de recursos possíveis, retirando os custos de inicialização dos custos de trabalho efectivo. [3] [4]

Previsibilidade Grande parte dos recursos armazenados possuem custos temporais determinísticos, através de uma boa implementação do padrão é possível obter uma previsão precisa dos custos associados à procura e execução dos recursos presentes na *pool*. [3]

Simplicidade Não é necessário implementar métodos de gestão de memória, sendo possível o cliente adquirir e libertar recursos diretamente da *pool*. [3]

Estabilidade e escalabilidade Novos recursos são criados, caso a necessidade seja superior aos recursos já presentes no sistema, como tal é possível que a *pool* atrase a libertação de recursos consoante o estado atual do sistema, isto permite optimizar os processos de libertação e reaquisição de recursos e consequentemente a estabilidade do próprio programa. [3]

Partilha Recursos não utilizados podem ser partilhados por vários clientes, o que beneficia os custos associados a memória. [3]

5.2 Desvantagens

Sobrecarga A gestão de recursos na *pool* possui custos associados, como tal para ambientes de pouca instanciação os custos para libertação e criação de recursos poderão ser menores aos utilizados no padrão. Também devido a novas evoluções tecnológicas, alternativas como "*garbage collecters*" poderão se tornar mais eficientes do que manter guardado um grande número de objetos "vivos", mas não utilizados. [3] [9]

Complexidade Clientes têm que manualmente libertar recursos para a *pool*. [3]

Sincronização Em ambientes concorrentes, pedidos para a *pool* têm que ser sincronizados de modo a não corromper o estado associado da mesma [3], isto pois a *pool* não realiza a sincronização de acesso, logo os recursos têm que fornecer os seus próprios de sincronização. Existem três métodos que necessitam ser sincronizados:

- getInstance
- acquire, de modo a não retornar o mesmo recursos a diferentes clientes. [7]
- release, sincronização interna ao sistema necessária. [7]

Recursos expirados Como já referido, os recursos têm que ser manualmente libertados da *pool*. Existem diversos exemplos onde o cliente se esquece de o fazer, o que provoca desperdício da utilização de memória e consequentemente problemas de performance. [3] [7]

Recursos limitados A *pool* é responsável pela partilha e reutilização de recursos, logo poderá ocorrer que o número de recursos armazenados exceda o limite permitido pelo sistema, caso isto aconteça, o programa retorna uma exceção e fica em espera até que um recurso seja libertado. [7]

Dimensões dos recursos Grande parte das implementações armazenam os recursos da *pool* numa lista de objetos, como tal a memória dedicada para cada elemento será fixa. Isto faz com que seja necessário implementar a lista com o tamanho do objeto maior que possuir, obviamente isto poderá trazer grandes complicações futuras, onde um objeto relativamente maior que os outros afeta todo o sistema negativamente. [6] Esta caraterística poderá ser contornada criando múltiplas *sub-pools* onde ficam armazenados apenas objetos do mesmo tipo. [3]

6 Outros Padrões

6.1 Mediator

Como referido anteriormente recursos não utilizados podem ser partilhados por vários clientes, o que poderá causar problemas de sincronização no sistema. Uma solução poderá ser a implementação do padrão *Mediator* de modo a centralizar a comunicação e realizar a filtragem concorrente. [3]

6.2 Factory Method

O *Factory Method* poderá ser usado para encapsular a criação lógica de objetos, desta forma este poderá ser complementado com o *Object Pool* de modo a realizar a gestão do objeto após a sua criação. [8]

6.3 Flyweight

Ambos padrões são muito semelhantes, mas diferem na forma como são utilizados. Enquanto que os objetos na *pool* são utilizados por um únicos cliente e depois o cliente retorna o objeto de volta para a *pool*, objetos *flyweight* podem ser utilizados simultâneamente por vários clientes, ou seja objetos *flyweight* são imutáveis e objetos *pool* são mutáveis. [1]

6.4 Singleton

A maior parte dos *Object Pools* são implementados como *singletons*. [7]

6.5 Data Locality

O *Object Pool* poderá ser complementado com o padrão *data locality* de modo a aceder a memória recentemente utilizada e otimizar a velocidade do sistema. [5] [6]

6.6 Leasing

Este padrão ajuda a resolver o problema onde os clientes têm que manualmente libertar os recursos, pois realiza a gestão de utilização recursos, limitando o acesso de um recurso por um período de tempo pré-definido. [3] [2]

7 Utilizações

Resumidamente podemos utilizar o *object pool* sempre que existem vários clientes que necessitam do mesmo recurso *stateless* com custos de criação elevados [7], quando é necessário criar e destruir objetos ou quando cada objeto encapsula um recurso que possui custos elevados de obter e pode ser reutilizado. [6]

7.1 Gestor de memória

O padrão permite a implementação de gestores de memória eficientes, onde um gestor possui um certo número de *pools* de dimensões diferentes. Quando é necessário alocar um bloco, o gestor encontra um espaço de tamanho apropriado e armazena o bloco. [6]

7.2 Conexões

Talvez a maior utilização deste padrão seja a gestão de conexões, pois abrir várias conexões pode afetar a performance, devido aos seus custos de criação e excesso de conexões abertas simultâneamente. Desta forma o *object pool* permite reutilizar e partilhar conexões, otimizando o sistema. [7]

7.3 Servidores Web

Este padrão é também utilizado para gerir de forma eficiente tarefas num servidor web. A criação de tarefas para muitos utilizadores concorrentes é ineficiente, logo as tarefas são reutilizadas após realizar uma data tarefa. [3]

7.4 Desenvolvimento de jogos

Finalmente outro uso comum deste padrão é no desenvolvimento de jogos, onde poderá ser utilizado para gestão de entidades, efeitos visuais, efeitos sonoros... [6]

References

1. Sławomir Kowalski. 2018. Design patterns: Object pool. (2018). https://medium.com/@sawomirkowalski/design-patterns-object-pool-e8269fd45e10.
2. Prashant Jain Michael Kircher. 2000. Leasing. (2000). https://hillside.net/plop/plop2k/proceedings/Jain-Kircher/Jain-Kircher.pdf.
3. Prashant Jain Michael Kircher. 2003. Pooling. (2003). http://www.kircher-schwanninger.de/michael/publications/Pooling.pdf.
4. microsoft. 2018. Improving Performance with Object Pooling. (2018). https://docs.microsoft.com/en-us/windows/win32/cossdk/improving-performance-with-object-pooling?redirectedfrom=MSDN.
5. Robert Nystrom. 2014a. Game Programming Patterns, Data Locality. (2014). http://gameprogrammingpatterns.com/data-locality.html.
6. Robert Nystrom. 2014b. Game Programming Patterns, Object Pool. (2014). http://gameprogrammingpatterns.com/object-pool.html.
7. OODesign. 2005. Object Pool. (2005). https://www.oodesign.com/object-pool-pattern.html.
8. Alexander Shvets. 2019. Dive Into Design Patterns. (2019). https://sourcemaking.com/design$_p atterns/object_p ool$.
9. Wikipedia. 2012. Object pool pattern. (2012). https://en.wikipedia.org/wiki/Object$_p ool_p attern$.