Universidade do Minho

Mestrado em Engenharia Informática

Arquiteturas Aplicacionais

Ano lectivo 2014/2015

Trabalho Prático

Design Patterns

Autores:
José Morgado (pg27759)
Luís Miguel Pinto (pg27756)
Pedro Carneiro (pg25324)

Braga, 23 de Março de 2015

Resumo

Conteúdo

1 Introdução

2 Patterns de Criação

2.1 Singleton

Este *pattern* é utilizado quando se pretende a existência de apenas uma instância de uma classe. Para uma classe ser *singleton* deve garantir-se que há apenas uma instância e um ponto de acesso à mesma.

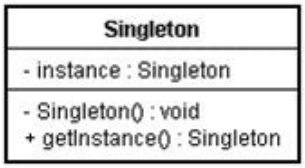


Figura 1: Diagrama de classe

No diagrama de classe apresentado anteriormente existe um atributo *singleton* que é do tipo da própria classe. Nos métodos da classe podemos verificar a presença do construtor de classe, *Singleton()*, que é privado. Ora, para que a classe seja instanciada é necessário utilizar o método estático *getInstance()*, assim pode ser acedido por qualquer outra classe.

Exemplo 1: Solução de implementação

```
public class MyClass {
  private static MyClass instance = null;

  private MyClass() {
  }

  public static MyClass getInstance() {
    if(instance==null) {
      instance = new MyClass();
    }
    return MyClass.instance;
  }
}
```

Com a implementação anterior assegurasse que ao tentar-se criar duas instâncias da mesma classe isso não é possível, pois o atributo *instance* já não está a *null*.

Vantagens É um *Pattern* simples e é utilizado quando se pretende ter um acesso único e global para organizar os recursos.

Desvantagens A utilização exagerada deste *pattern* pode provocar depedências muito fortes entre os objetos e dificultar alterações de configurações futuras.

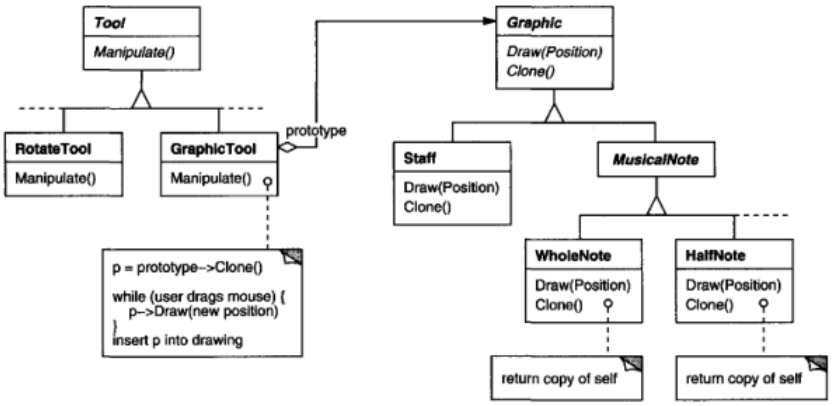
2.2 Prototype

2.2.1 Objetivo

Especificar tipos de objetos usando uma instância prototípica a partir da qual serão criadas cópias da mesma.

2.2.2 Motivação

Considere-se uma framework para criação de editores gráficos de partituras:



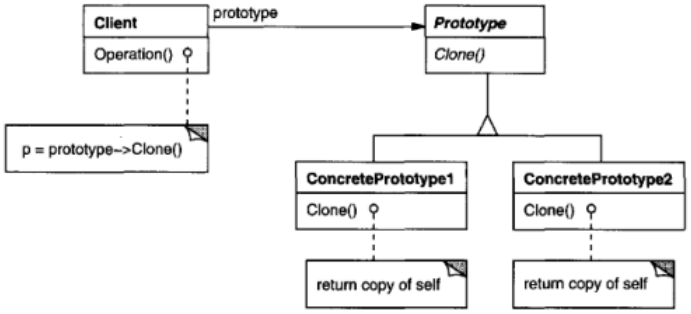
A classe GraphicTool necessita de componentes gráficos, como por exemplo notas musicais. Estas têm diversas variantes pelo que criar uma subclasse para cada uma pode ser impraticável. Nestes caso, uma alternativa mais adequada será obter novas instâncias da mesma classe a partir de um método Clone() e inicializá-las com diferentes *bitmaps* e durações.

2.2.3 Aplicação

Usar quando:

- as classes a instanciar são especificadas em tempo de execução;
- se pretende evitar ter uma hierarquia de classes *factory* extensa;
- instâncias de uma classe apenas podem ter algumas combinações de estado e, por isso, é mais conveniente clonar um protótipo e alterar parte do seu estado do que criar uma nova instância e inicializar todo o seu estado.

2.2.4 Estrutura



2.2.5 Participantes

- Prototype (Graphic), declara uma interface para criar clones
- ConcretePrototype (Staff, WholeNote, HalfNote), implementa o método de clonagem
- Client (GraphicTool), cria novos objetos a partir do protótipo

2.2.6 Colaborações

Um cliente solicita a um protótipo cópias do mesmo.

2.2.7 Consequências

- Permite incorporar facilmente novas classes concretas de um produto num sistema de uma forma mais flexível relativamente aos restantes padrões de criação porque o cliente pode adicionar e remover protótipos em tempo de execução.
- Permite reduzir significativamente o número de classes que um sistema necessita.

2.2.8 Implementação

- Quando o número de protótipos é dinâmico devemos considerar a utilização de um gestor de protótipos a partir do qual o cliente possa obter e guardar os protótipos.
- Ao implementar a operação Clone devemos ter em atenção as referências a outros objetos.
- Se o objeto a clonar não disponibilizar métodos para alterar o seu estado, pode ser necessário adicionar um método Initialize que aceite os parâmetros a inicializar como argumentos.

3 Patterns Estruturais

3.1 Facade

O *pattern* Facade esconde toda a complexidade de uma ou mais classes do sistema e fornece uma interface, que o cliente pode aceder para utilizar o sistema.

Ao se utilizar uma Facade, que implementa uma interface, pretende-se reduzir o nível de complexidade no subsistema, aumentando a facilidade de utilização.

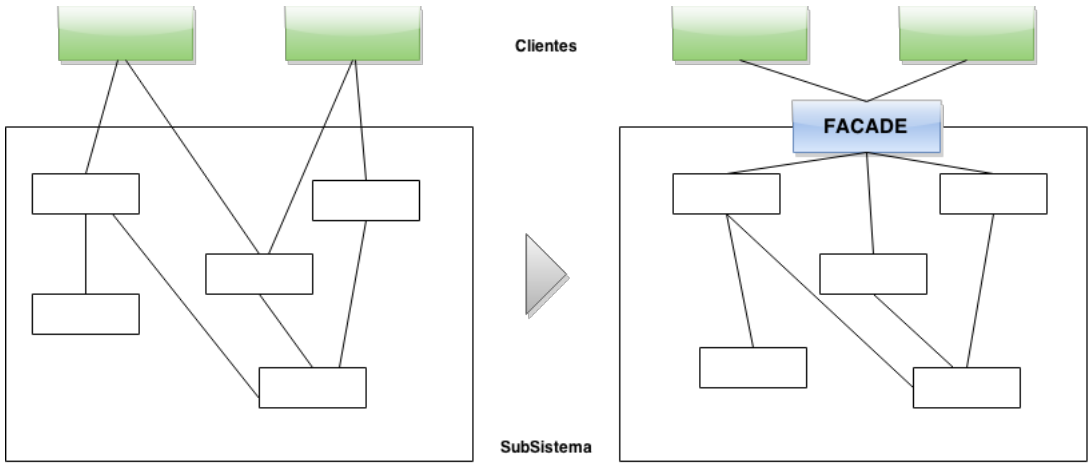


Figura 2: Antes e depois da utilização do Facade

Como vemos na figura anterior, este *pattern* é definido através de uma única classe que fornece os métodos necessários ao cliente e envia os pedidos deste às classes do subsistema.

Aplicação
Utilizar Facade quando:

- Se deseja fornecer uma interface simples para um sistema complexo.
- Existe muitas dependências entre clientes e as classes de implementação de uma abstração.
- Se pretende ter uma estrutura em camadas no sistema.

Colaboração
Os clientes comunicam com o subsistema através do envio de pedidos ao Facade, que redireciona os mesmos para as classes apropriadas do subsistema. Ainda que, os clientes que utilizam este *pattern* não tem que aceder aos objetos do seu subsistema diretamente.

Vantagens

- A utilização deste *pattern* permite diminuir as ligações entre os clientes e o sistema.

- Para adicionar novas funcionalidades ao sistema seria necessário alterar apenas o Facade, ao invés de alterar os vários objetos do sistema, sem afetar o cliente.

Implementação

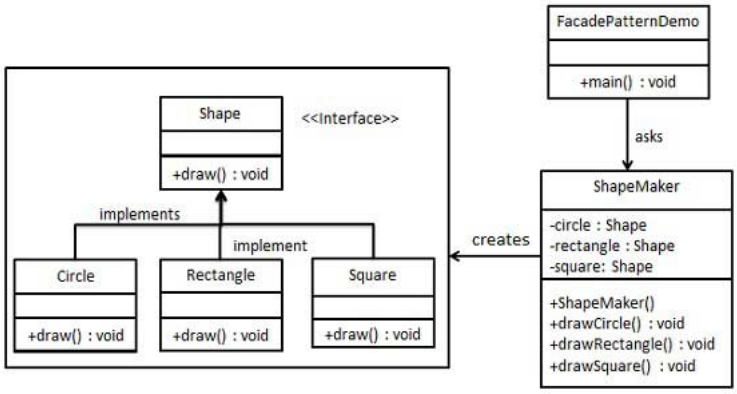


Figura 3: Antes e depois da utilização do Facade

Nós vamos criar uma interface Forma e classes concretas implementando a interface Shape. A ShapeMaker classe de fachada é definido como um próximo passo.

3.2 Flyweight

3.2.1 Objetivo

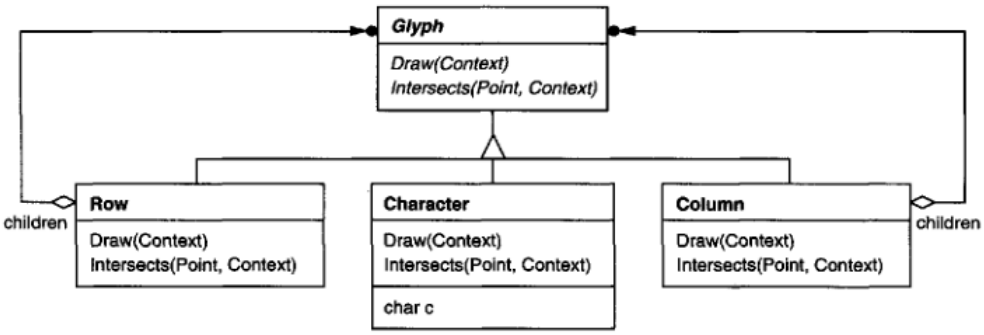
Suportar um grande número de objetos de tamanho reduzido através da partilha de estado entre eles.

3.2.2 Motivação

Considere-se por exemplo um editor de texto que necessita de manter a informação aos caracteres existentes num documento. Uma solução seria ter um objeto por caracter, mas isso seria inviável em termos de memória. O padrão Flyweight fornece um mecanismo para contornar esta limitação.

Um flyweight é um objeto partilhado que pode ser utilizado em múltiplos contextos simultaneamente. O conceito chave é a distinção entre estado intrínseco (guardado no flyweight, partilhado) e extrínseco (passado como parâmetro pelos clientes).

Usando este padrão passaríamos a ter a seguinte estrutura:



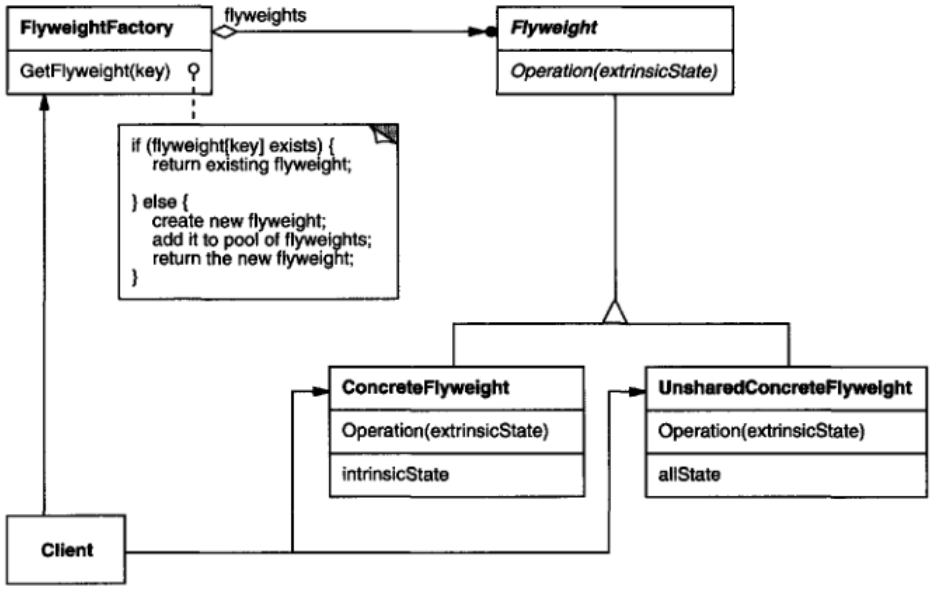
Neste caso temos um flyweight para cada letra do alfabeto, responsável por guardar o código do caracter, e quando um cliente necessitar de desenhar uma dada letra recorre ao respetivo flyweight indicando a localização e a fonte pretendidas.

3.2.3 Aplicação

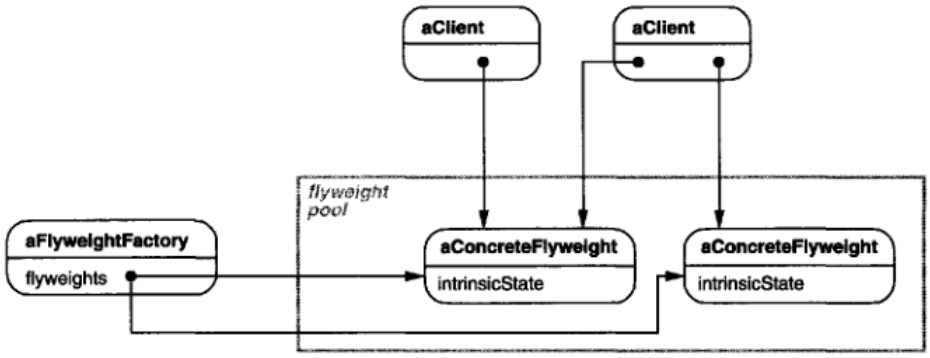
Usar quando:

- a aplicação usa um grande número de objetos;
- é necessário reduzir a quantidade de objetos em memória;
- uma grande parte do estado do objeto pode ser partilhada;
- é possível substituir vários grupos de objetos por um número bem menor de objetos se o estado extrínseco for removido.

3.2.4 Estrutura



O seguinte diagrama demonstra como os flyweights são partilhados:



3.2.5 Participantes

- Flyweight (Glyph), declara uma interface a partir da qual os flyweights recebem e manipulam o estado extrínseco
- ConcreteFlyweight (Character), implementa a interface Flyweight e adiciona o estado intrínseco
- UnsharedConcreteFlyweight (Row, Column), não partilha estado, tipicamente agrega coleções de objetos ConcreteFlyweight
- FlyweightFactory, cria e gere objetos flyweight
- Client, mantém as referências para os flyweights e define o seu estado extrínseco

3.2.6 Colaborações

- O estado intrínseco é guardado no objeto ConcreteFlyweight e o estado extrínseco é guardado ou calculado por objetos Client.
- Os clientes não devem criar instâncias ConcreteFlyweight, mas obtê-las a partir de uma FlyweightFactory.

3.2.7 Consequências

- As operações associadas à gestão do estado extrínseco podem ter impacto no desempenho.
- Quantos mais flyweights forem partilhados e quanto maior for o estado partilhado, maior será a poupança de memória.
- Caso se pretenda combinar este padrão com o padrão Composite, as referências para os pais têm de ser passadas como parte do estado extrínseco.

3.2.8 Implementação

- O estado extrínseco deve ser reduzido, caso contrário não será possível tirar partido das vantagens pretendidas.

- Se o número de flyweights não for fixo e pequeno, pode ser necessário efetuar a gestão dos objetos partilhados por forma a reduzir o impacto ao nível do desempenho e uso de memória.

4 Patterns de Comportamento

4.1 Memento

4.1.1 Objetivo

Sem violar o encapsulamento, capturar e externalizar o estado interno de um objeto de modo a que o mesmo possa ser reposto posteriormente.

4.1.2 Motivação

Considere-se por exemplo um editor gráfico que permite definir linhas entre rectângulos e que quando se desloca um dos rectângulos a linha altera-se apropriadamente:

Uma forma recorrente de realizar este tipo de operação é através de um sistema especializado, cuja funcionalidade é encapsulada num objeto (ConstraintSolver).

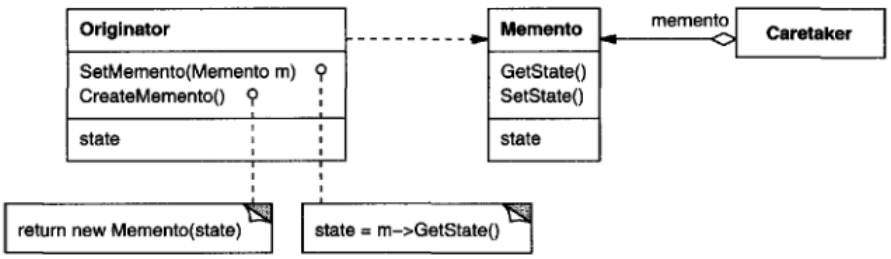
Por forma a suportar a operação undo, é necessário guardar registos dos vários estados do objeto ContraintSolver. Isto pode ser feito à custa de um memento (SolverState), um objeto que armazena um snapshot do estado interno de outro objeto (originador).

4.1.3 Aplicação

Usar quando:

- é necessário manter registos do estado de um objeto por forma a ser possível repô-lo mais tarde;
- não é possível expor esse estado através de getters.

4.1.4 Estrutura

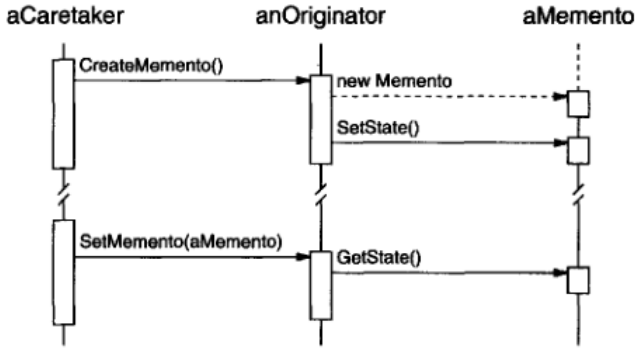


4.1.5 Participantes

- Memento (SolverState), guarda o estado do objeto Originator e não disponibiliza esse estado a outros objetos para além do originador
- Originator (ConstraintSolver), cria o memento e usa-o para repor o seu estado interno
- Caretaker, é o responsável pelo mecanismo de undo

4.1.6 Colaborações

- O caretaker requisita um memento ao originador, mantém-no durante algum tempo, e devolve-o ao originador:



- O memento é passivo, apenas o originador que o criou atribui ou recupera o seu estado.

4.1.7 Consequências

- Garante a preservação do encapsulamento do Originator.
- Simplifica a classe Originator, pois esta não necessita de gerir as versões do seu estado interno.
- Pode implicar um elevado overhead.
- Aumento do uso de memória para manter os vários mementos.

4.1.8 Implementação

- Quando se cria um memento, este não deve guardar todo o estado mas apenas as alterações relativamente ao último snapshot.

5 Conclusão

6 Referências

Referências

[1] Eric Freeman, Elisabeth Robson, Bert Bates, Kathy Sierra. Head First Design Patterns. O'Reilly Media, 2004.

[2] http://wiki.portugal-a-programar.pt/dev_geral:java:padrao_singleton

[3] http://www.devmedia.com.br/padrao-de-projeto-singleton-em-java/26392

[4] http://www.tutorialspoint.com/design_pattern/facade_pattern.htm

[5] Design Patterns: Elements of Reusable Object-Oriented Software, Erich Gamma, Richard Helm, Ralph Johnson, John M. Vlissides, November 1994