Documentação

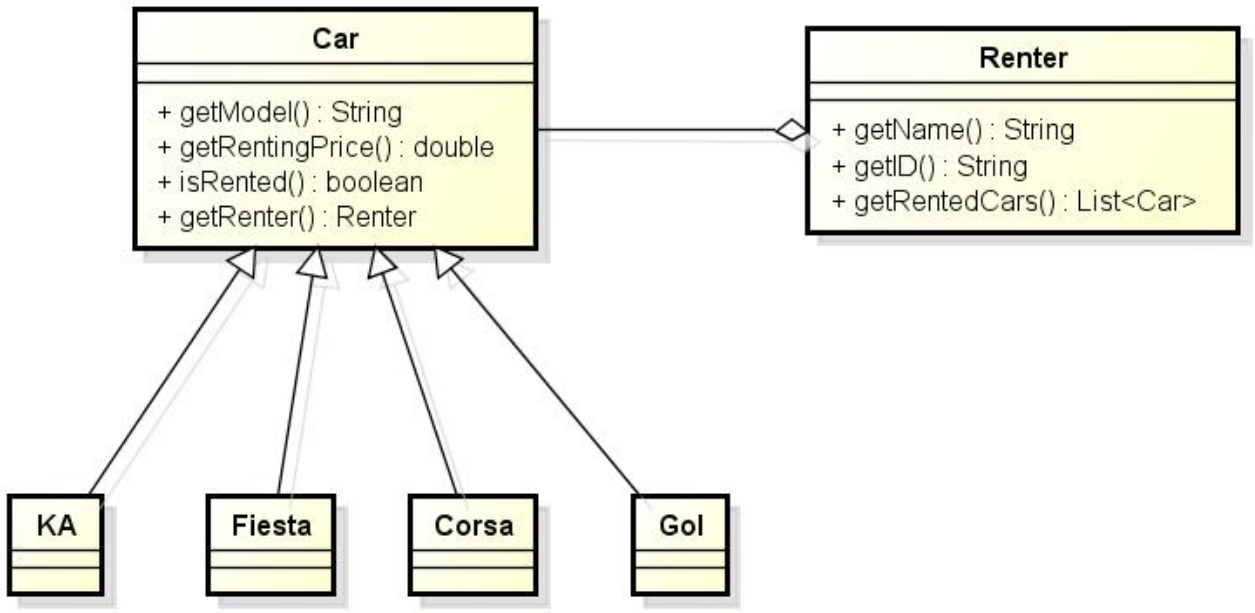
1. Noções básicas de AOM

Um Modelo Adaptativo de Objetos (Adaptive Object Model – AOM) é um estilo arquitetural comum para sistemas em que classes, atributos, relacionamentos e comportamentos são representados como metadados consumidos em tempo de execução. Isso faz com que sistemas criados com essa arquitetura sejam muito flexíveis e possam ser modificados em tempo de execução, não apenas por programadores, mas também por usuários finais, permitindo que mudanças sejam realizadas e disponibilizadas ao mercado em tempo hábil.

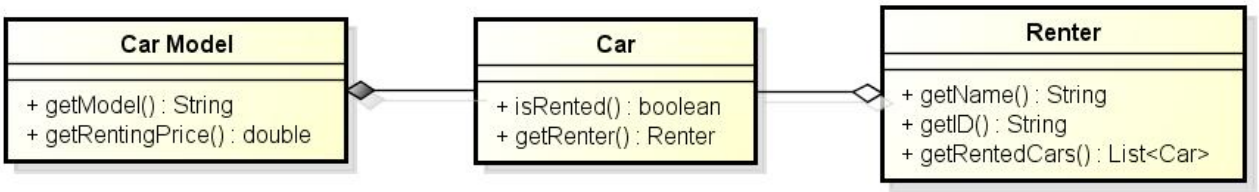
1.1. Type Object

O padrão Type Object é utilizado em situações em que o número de subclasses que uma classe pode precisar não pode ser determinada durante o desenvolvimento do sistema.

Por exemplo, em um sistema de aluguel de carros, é necessário que diferentes carros de diferentes modelos possam ser inseridos no sistema. Uma forma de implementar esse sistema é criar uma classe Car com diversas subclasses - uma para cada modelo - conforme a figura abaixo.



Com essa solução, se quisermos inserir um novo modelo de carro ao sistema, teríamos que criar uma nova subclasse de Car, recompilar o código e instalar a atualização do sistema. Além do trabalho necessário para realizar essa mudança, com a evolução do sistema é possível que a classe Car passe a ter um grande número de subclasses com pequenas diferenças entre elas. O padrão Type Object resolve essa situação representando as subclasses que não são conhecidas em tempo de desenvolvimento como instâncias de uma classe genérica que representa o tipo de um objeto.

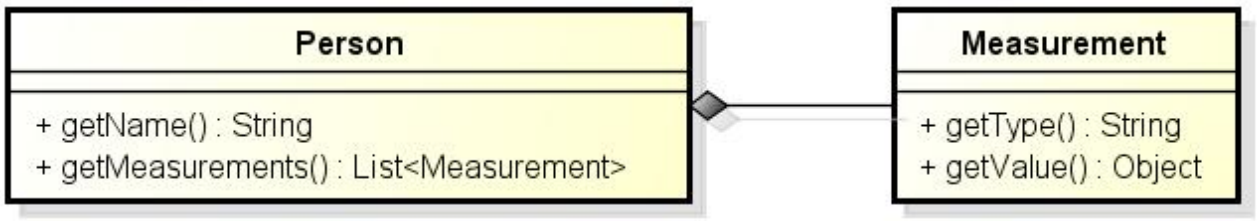


Neste exemplo, as subclasses de Car foram representadas como instâncias da classe CarModel. A relação classe-instância que existia no modelo antigo passaram a ser representadas como uma relação instância-instância, onde as instâncias de Car possuem uma referência a instâncias de CarModel. Por meio dessas referências é possível determinar o modelo de um carro. Esta solução desacopla instâncias de objetos de suas classes, permitindo a adição dinâmica de novas "classes" (representadas por instâncias) ao sistema.

1.2. Property

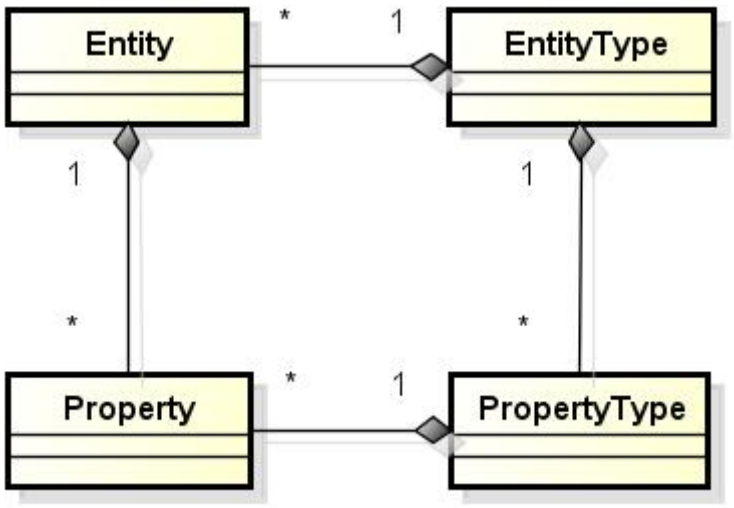
Em situações em que instâncias de uma mesma classe podem possuir diferentes tipos de propriedades, criar um atributo para representar cada uma dessas propriedades pode não ser a melhor solução. Por exemplo, em um sistema médico pode-se criar uma classe Person para armazenar informações de pacientes, como altura, peso e tipo sanguíneo. Uma solução para o sistema seria adicionar um atributo para cada tipo de informação necessária para o paciente na classe Person. No entanto, um hospital pode conter diferentes departamentos que necessitam de diferentes tipos de informação. Para fazer com que o sistema possa ser utilizado pelos diferentes departamentos do hospital, poderia ser necessário criar um grande número de atributos na classe Person, sendo que apenas uma pequena quantidade desses atributos seria efetivamente utilizada por uma instância dessa classe (apenas os atributos necessários pelo departamento que está tratando o paciente seriam efetivamente utilizados).

O padrão Property resolve esse problema representando as propriedades de uma entidade por meio de uma classe e fazendo com que essa entidade possua uma coleção de instâncias dessa classe. Aplicando o padrão Property no exemplo, uma classe Measurement pode ser criada para representar os dados de um paciente. Com a mudança, os atributos de Person podem ser substituídos por uma coleção de Measurements apenas com as medidas necessárias para o paciente específico.



1.3. Type Square

Em um AOM, os padrões Type Object e Property são usualmente utilizados em conjunto, resultando no padrão Type Square. Nesse padrão, o Type Object é utilizado duas vezes – uma vez para representar as entidades e tipos de entidades; e uma vez para representar propriedades e tipos de propriedades.



Neste padrão, as classes EntityType e PropertyType representam o modelo e a associação entre elas permite a identificação de quais tipos de propriedades são aplicáveis a que tipos de entidades. As classes Entity e Property estão relacionadas à representação das instâncias no sistema. Cada instância de Entity referenciam uma instância de EntityType, que representa o tipo dessa entidade. Para cada PropertyType de um EntityType, uma Property é criada para armazenar o valor da propriedade em uma Entity com o tipo dessa EntityType. Com este padrão, novos tipos de entidades com diferentes tipos de propriedades podem ser criados dinamicamente. Da mesma forma, tipos de entidades existentes podem ser modificados em tempo de execução, uma vez a modelagem é feita no nível das instâncias do sistema.

2. Utilização do Esfinge AOM Role Mapper

O framework Esfinge AOM Role Mapper implementa as classes genéricas do AOM, provendo uma implementação do Type Square.

As seções seguintes apresentam as funcionalidades do AOM Role Mapper que precisarão ser utilizadas no experimento. Entre os exemplos das seções, os nomes das variáveis foram mantidas para que seja mais fácil identificar o tipo da variável passada como parâmetro.

2.1. Criação de tipos de entidades

Para criar um novo tipo de entidade genérica:

```
IEntityType tipoEntidade = new GenericEntityType("Tipo");
```

Para adicionar um tipo de propriedade nesse tipo de entidade:

```
GenericPropertyType propertyType = new GenericPropertyType("propriedade", String.class);
tipoEntidade.addPropertyType(propertyType);
```

2.2. Criação e manipulação de entidades

Criar uma entidade de um tipo de entidade existente:

```
IEntity entidade = tipoEntidade.createNewEntity();
```

Configurar um valor para uma propriedade de uma entidade:

```
entidade.setProperty("propriedade", valor);
```

Recuperar o valor de uma propriedade da entidade:

```
String valor = (String) entidade.getProperty("propriedade").getValue();
```

2.3. Configuração de Metadados

No framework, é possível adicionar propriedades em tipos de propriedades, sendo que essas representam metadados. Por exemplo, adicionar o valor a uma propriedade na entidade, seria o equivalente a configurar o valor de um atributo em um objeto. Configurar uma propriedade no tipo de propriedade, seria o equivalente a adicionar uma anotação nesse atributo.

Configurar um metadado simples (sem atributos) em uma propriedade:

```
propertyType.setProperty("metadadoSimples",true);
```

Configurar um metadado complexo (com atributos) em uma propriedade:

```
Map<String, Object> atributosMetadado = new HashMap<String, Object>();
atributosMetadado.put("prop1", 10);
atributosMetadado.put("prop2", "valor");
propertyType.setProperty("metadadoComplexo", atributosMetadado);
```

2.4. Geração de Java Beans a partir de entidades

Como forma de permitir que frameworks destinados a Java Beans (classes Java comuns cujas propriedades são acessadas por métodos get e set) possam ser utilizados para as entidade AOM, o framework Esfinge AOM Role Mapper provê uma funcionalidade que gera um adaptador no formato Java Bean para uma entidade AOM. A classe desse adaptador é gerada em tempo de execução baseada nas propriedades da entidade.

Criar um objeto de uma classe no formato Java Bean que encapsula a entidade AOM da classe IEntity:

```
AdapterFactory af = AdapterFactory.getInstance("AnnotationMapping.json");
Object javaBean = af.generate(entidade);
```

O nome do arquivo passado como parâmetro possui o mapeamento entre anotações de código e os metadados do AOM. No exemplo abaixo, a anotação @CPF foi mapeada para o metadado “cpf” e a anotação @Size com os atributos “max” e “min” foi mapeada para o metadado “size” com os mesmos atributos. Nesse caso, quando a entidade for transformada em um Java Bean, serão adicionadas anotações de acordo com os metadados da propriedade.

```
{ "size":[
    {"target":"attribute"},
       {"annotationPath":"javax.validation.constraints.Size"},
       {"parameter_1": "max"},
       {"parameter_2": "min"},
  ],
  "cpf":[
    {"target":"attribute"},
       {"annotationPath":"org.hibernate.validator.constraints.br.CPF"}
  ]
}
```

2.5. Exemplo de uso do Esfinge AOM Role Mapper

Para entender como as coisas funcionam em conjunto, abaixo está um código que cria um tipo de entidade, adiciona metadados, cria a entidade e cria um adapter Java Bean. Nesse exemplo, iremos considerar o mapeamento de anotações utilizado como exemplo na seção anterior:

```
IEntityType exemploTipo = new GenericEntityType("Exemplo");

//criando property types
GenericPropertyType textoPropertyType = new GenericPropertyType("texto", String.class);
GenericPropertyType documentoPropertyType = new GenericPropertyType("documento", String.class);

//Adicionando metadados
Map<String,Object> textoMetadados = new HashMap<>();
textoMetadados.put("min", 5);
textoMetadados.put("max", 50);
textoPropertyType.setProperty("size", textoMetadados);
documentoPropertyType.setProperty("cpf", true);

//criando entidade
IEntity exemplo = exemploTipo.createNewEntity();
exemplo.setProperty("nome", "texto");
exemplo.setProperty("documento", "012.717.436-20");

//Transformando para Java Bean
AdapterFactory af = AdapterFactory.getInstance("AnnotationMapping.json");
Object exemploJavaBean = af.generate(exemplo);
```

O objeto gerado é de uma classe gerada em tempo de execução, então não é possível mostrar qual o código dela. Porém abaixo segue um código que mostra como seria sua interface externa:

```
public class ExemploAOMBeanAdapter {

        @Size(min=5,max=50)
        public String getTexto(){...}
        @CPF
        public String getDocumento(){...}
        public void setTexto(String s){...}
        public void setDocumento(String s){...}
}
```

Um framework que acessar essa classe através de reflexão irá enxergar esses métodos com essas anotações.

3. Utilização do Hibernate Validator

O framework Hibernate Validator tem o objetivo de validar restrições de dados nos objetos de uma classe. Ele pode verificar, por exemplo, se um valor numérico do objeto está dentro de valores limite ou se uma String possui um determinado formato. O framework define essas restrições a partir de anotações na classe.

As seções seguintes apresenta o uso das funcionalidades do framework Hibernate Validator que serão necessárias para o experimento.

3.1. Invocação da funcionalidade de validação de Java Bean

Segue o código do Hibernate Validator para validar os dados de um Java Bean de acordo com as regras definidas nas anotações:

```
ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
Validator validator = factory.getValidator();
Set<ConstraintViolation<Object>> constraintViolations = validator.validate(javaBean);
```

O retorno na variável `constraintViolations` possui os erros da validação. No caso, como o experimento pede apenas para retornar se a instância está válida ou não (verdadeiro ou falso), basta verificar se existe algum elemento no Set com as violações. Caso não haja violação, o Set estará vazio.

3.2. Anotações do Hibernate Validator

Seguem algumas anotações do Hibernate Validator que se aplicam as validações solicitadas no experimento. Vale ressaltar que elas se aplicam ao atributo que estiverem anotando:

- @Future : a data precisa ser no futuro
- @Max(value=) : o valor numérico deve ser menor ou igual a "value" (o tipo do atributo é Long – o número precisa ter um L depois → 100L).
- @Min(value=) : o valor numérico deve ser maior ou igual a "value" (o tipo do atributo é Long – o número precisa ter um L depois → 100L).
- @NotNull : o valor não pode ser nulo
- @Past : a data precisa estar no passado
- @Pattern(regexp=) : a String deve obedecer o padrão com expressão regular
- @Size(min=, max=) : a String deve ter uma quantidade de caracteres entre "min" e "max"
- @Email : verifica se a String contém um e-mail válido
- @NotBlank : verifica se a String não é nula, não é vazia e não é formada apenas por espaços
- @NotEmpty : verifica se a String não é nula e não é vazia
- @Range(min=, max=) : verifica se o valor numérico é maior ou igual a "min" e menor ou igual a "max" (os tipos dos atributos são Long - o número precisa ter um L depois → 100L).
- @CPF : verifica se o valor contém um CPF válido
- @CNPJ : verifica se o valor contém um CNPJ válido

Obs: O arquivo AnnotationMapping.json possui todas essas anotações mapeadas para metadados que podem ser utilizados em um tipo de entidade AOM.

3.3. Exemplo de Uso

Esta seção mostra um exemplo de uso do framework Hibernate Validator. Considere a seguinte classe que define uma propriedade chamada "texto", que precisa ter uma String de tamanho entre 20 e 100 caracteres, e uma propriedade "numero", que deve possuir um inteiro com valor mínimo de 10. Repare que as anotações definem essas restrições:

```
public class JavaBean {

	@Size(min=20,max=100)
	private String texto;

	@Min(10)
	private int numero;

	//métodos get e set omitidos
}
```

Agora, veja um exemplo de criação de uma instância dessa classe, e posteriormente como as

restrições dos dados seriam validadas com o Hibernate Validator. No exemplo, após a validação, as violações encontradas são impressas no console:

```
JavaBean jb = new JavaBean();
jb.setTexto("texto da propriedade");
jb.setNumero(30);

ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
Validator validator = factory.getValidator();
Set<ConstraintViolation<Object>> violations = validator.validate(jb);
for(ConstraintViolation<Object> violation : violations){
    System.out.println(violation.getMessage());
}
```