02

Escrevendo Menos Código com Templates

Você pode fazer o download do projeto deste capítulo aqui (https://s3.amazonaws.com/caelum-online-public/eclipse/eclipse-6.zip). Ele será necessário para a realização dos exercícios.

Embora a abordagem que fizemos com nosso DAO no capítulo anterior seja válida e muito comum em códigos da indústria, ela apresenta alguns problemas. Se não quisermos que algum método do nosso `DAO` seja disponibilizado, por exemplo, se não pudermos permitir que os gastos sejam atualizados após sua inserção, ou não permitir que se remova um funcionário porque podem haver gastos associados a ele, teremos que sobrescrever o método sem fazer nada nele - ou ainda lançar uma `Exception` avisando que a operação não é suportada. Ambas alternativas são desaconselhadas, pois induzem a erros, disponibilizando métodos que não devem ser chamados.

Boas práticas de orientação a objetos (http://www.caelum.com.br/curso/online/orientacao-objetos-java/) nos dizem para evitar o uso desnecessário de herança, e utilizar composição em seu lugar (http://blog.caelum.com.br/como-nao-aprender-orientacao-a-objetos-heranca/). Não precisamos herdar de `DAO` para utilizarmos seus métodos, podemos criar um atributo do tipo `DAO` e delegar a ele as chamadas dos métodos que precisamos.

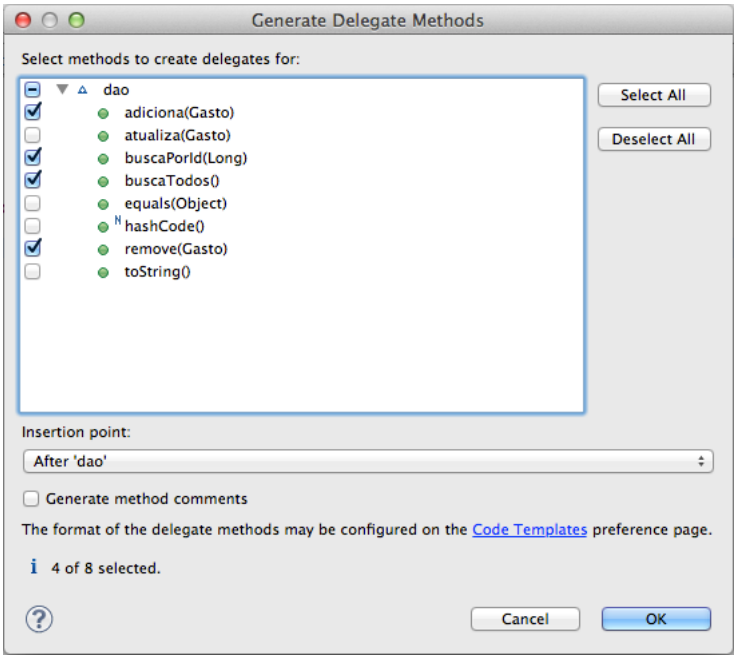
Vamos começar removendo a herança e o construtor, e em seguida criar um atributo do tipo `DAO` em nosso `GastoDAO`. O código deve estar assim:

```
public class GastoDAO {
    private DAO<Gasto> dao = new DAO<Gasto>(Gasto.class);

}
```

Agora é só criarmos os métodos passando para o atributo `dao` a responsabilidade de executar o acesso ao banco. Mas não precisamos fazer isso manualmente, pois existe uma atalho no Eclipse para gerar exatamente o código que queremos.

Aperte ctrl + 3, e digite delegate, para acessar o menu Generate Delegate Methods. Uma janela será exibida para selecionarmos quais métodos queremos delegar. Vamos selecionar todos exceto o `atualiza`, os métodos de `Object` (`equals`, `toString`, etc.) e clicar em Ok em seguida.



O resultado deve ser o seguinte:

```
public class GastoDAO {
    private DAO<Gasto> dao = new DAO<Gasto>(Gasto.class);

    public void adiciona(Gasto entity) {
        dao.adiciona(entity);
    }
```

```
    public void remove(Gasto entity) {
        dao.remove(entity);
    }

    public Gasto buscaPorId(Long id) {
        return dao.buscaPorId(id);
    }

    public List<Gasto> buscaTodos() {
        return dao.buscaTodos();
    }

}
```

É muito simples gerar uma classe para testar nosso `GastoDAO`. Vamos criar a classe `TesteDAO` utilizando ctrl + N. Insira um método main, usando main. Agora, vamos criar uma variável do tipo `GastoDAO`, digitando `new GasD`. Utilizando ctrl + 1, atribuiremos este objeto a uma variável chamada `dao`. Vamos invocar o método `buscaTodos` e atribuir seu resultado a uma variável `gastos` com os mesmos atalhos de antes.

Para exibir os nossos gastos, vamos escrever um `for` para percorrer a coleção e imprimir os valores. Na linha de baixo, escreva `foreach` e aperte enter. O Eclipse vai gerar o corpo do `for` automaticamente usando a sintaxe do Java 5. Falta só imprimir o conteúdo da variável `gasto`. Podemos fazer isso escrevendo `syso`, para gerar a impressão e adicionarmos a variável dentro dos parênteses.

Todas essas estruturas de código, como o método `main`, o corpo do `for`, ou ainda o `System.out.println()`, que são geradas automaticamente pelo Eclipse são templates.

Templates são pequenas estruturas de código configuráveis que podemos usar para criar trechos pré-definidos de código. Para acessá-los é só digitar o nome do template e pressionar ctrl+espaço. O Eclipse já vem com vários desses templates configurados, por exemplo:

main - Para gerar o método main em uma classe.

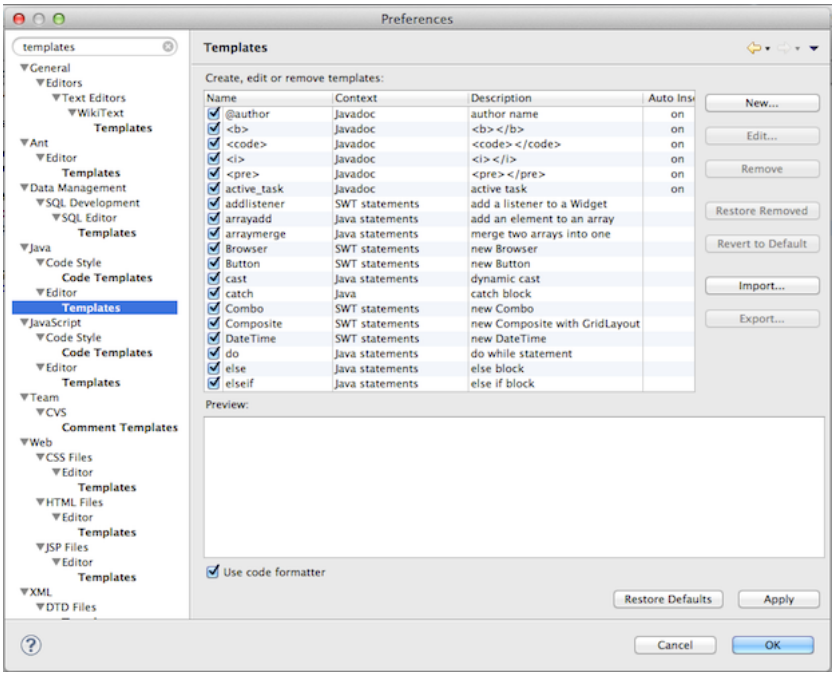
for - Para gerar estruturas de laço usando `for` Existem três variações, cada uma com um tipo de `for` diferente.

foreach - Para gerar estruturas de laço usando a sintaxe do novo `for` do Java 5.

syso - Para imprimir no console.

while - Mesma coisa que o `for`, mas para laços `while`.

Existem muitos outros templates, para consultar os existentes, acesse o menu Window e selecione Preferences.... Na janela de preferências digite templates para filtrar um pouco e selecione a opção Java -> Editor -> Templates, para ver a lista existente.



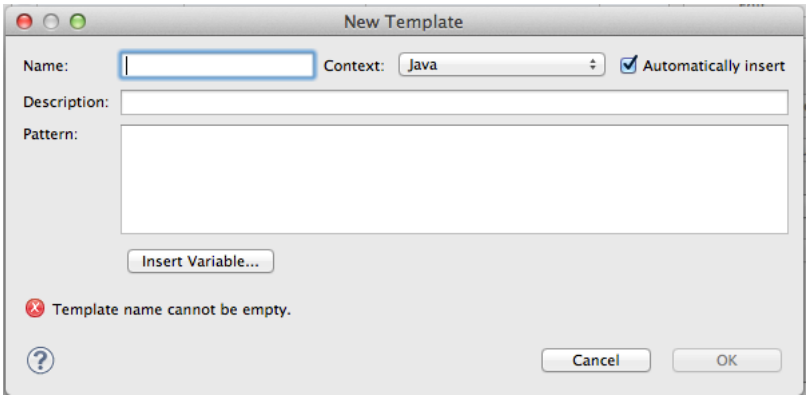
Além de usar os templates já existentes, podemos criar nosso próprio template para reduzir nosso trabalho com trechos de código que utilizamos com frequência.

Quando usamos o Log4J, precisamos adicionar um atributo estático em cada classe que queremos logar. Esse atributo é quase idêntico em cada declaração, mas o nome da classe a ser auditada tem que ser a classe em que você está agora. O código de criação do logger em si é bem simples:

```
private static final Logger logger = Logger.getLogger(MinhaClasse.class);
```

Vamos criar um template para gerar um logger do Log4J (http://blog.caelum.com.br/logar-e-preciso-debugar-nao-e-preciso/) para nossas classes.

Para iniciar a criação do template, acesse o menu Window e selecione Preferences.... Na janela de preferências digite templates para filtrar um pouco e selecione a opção `Java -> Editor -> Templates`. Na tela de template clique em New...



Preencha os dados do nosso template:

- Name - nome do nosso template é o código que vai chamar o template quando digitado e seguido de ctrl + espaço;
- Context - aqui você define onde seu template pode ser usado: Java é dentro de qualquer ponto em uma classe, Java Statements permite o uso dentro de métodos e Java Type Members pode ser dentro da classe e fora de métodos. Existem outros contextos, mas não são importantes para nós no momento;
- Description - Uma pequena descrição do template;
- Pattern - o código a ser gerado pelo template. Podemos usar várias variáveis definidas pelo Eclipse na criação do nosso código. Para ver quais estão disponíveis, podemos usar ctrl + espaço ou pressionar o botão Insert variable....

No campo Name colocaremos `logger`, em Context escolheremos `Java Type Members` já que nosso template serve para criar um atributo na classe. Vamos começar com o seguinte código dentro do campo Pattern:

```
private static final Logger logger = Logger.getLogger(${cursor});
```

A variável `${cursor}` é para que nosso cursor apareça nesse ponto após a geração do template. Vamos utilizar o template dentro de nossa classe GastoDAO, digitando `logger` logo após o atributo `dao`.

```
public class GastoDAO {
    private DAO<Gasto> dao = new DAO<Gasto>(Gasto.class);
    private static final Logger logger = Logger.getLogger();
```

Agora é só importarmos a classe `org.apache.log4j.Logger` e passar nossa classe dentro do método `getLogger`. Mas a idéia do template é reduzir ao máximo nosso trabalho, então vamos alterar o código do nosso template para já encontrar o tipo de nossa classe automaticamente.

```
private static final Logger logger = Logger.getLogger(${enclosing_type}.class);
```

Agora o Eclipse já consegue descobrir o nome de nossa classe e inserir no código gerado. O único trabalho que ainda realizamos é importar a classe `Logger`. Mas será que não dá pra automatizar isso também?

Olhando a lista de variáveis, achamos uma chamada `newType` . Quando utilizamos essa variável, podemos indicar uma classe que será incluída no código e importada se necessário. Nosso novo template ficará da seguinte maneira:

```
private static final ${type:newType(org.apache.log4j.Logger)} logger = Logger.getLogger(${enclosing_type}.class);
```

Repare que passamos o nome completo da classe, assim o Eclipse sabe qual a classe que deverá ser importada.

Em nosso código, digitamos `logger` e o resultado será o seguinte:

```
//outros imports...
import org.apache.log4j.Logger;

public class GastoDAO {
    private DAO<Gasto> dao = new DAO<Gasto>(Gasto.class);

    private static final Logger logger = Logger.getLogger(GastoDAO.class);
// resto da classe...
```

Utilizando as variáveis de template do Eclipse, podemos criar desde trechos simples até as estruturas realmente complexas, facilitando em muito a tarefa de escrever códigos que usamos com frequência.