**INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA**

Engenharia Informática e de Computadores

**Modelação e Padrões de Desenho**

**Trabalho final**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Membros do Grupo** | | | | | |
| **31768** | **Ricardo Romão** | **31923** | **Nuno Sousa** | **26657** | **Ricardo Neto** |

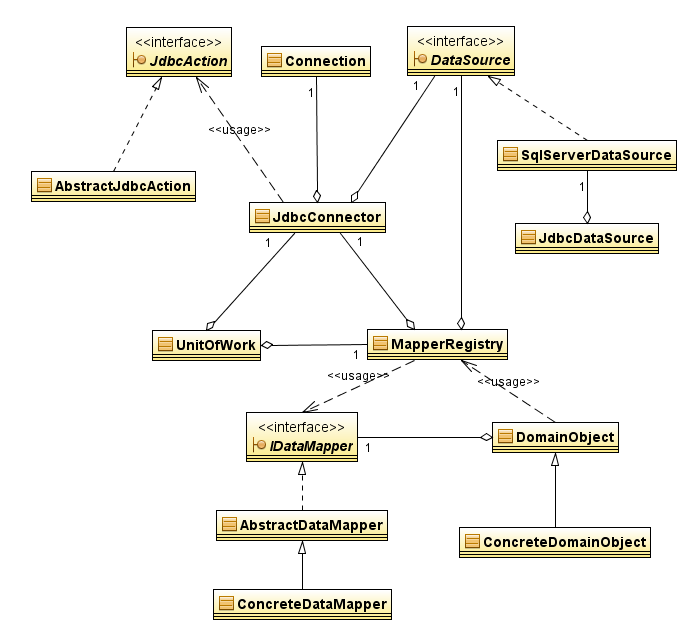
**Unit of Work**

Para a realização deste trabalho foi aproveitado muito do código já realizado ao longo do semestre, quer na resolução das séries propostas, quer na abordagem ao padrão de desenho *Unit Of Work* feita nas aulas.

Desta forma, foi opção do grupo a manutenção da abordagem estudada no que ao acesso à base de dados diz respeito, nomeadamente mantendo as classes *JdbcConnector* e *AbstractJdbcAction* e a *interface JdbcAction*, bem como as classes *AbstractDataMapper* e *DomainObject*, conforme o esquema 1.

A class *MapperRegistry* contém a correspondência entre os *DomainObjects* concretos e o respectivos *DataMappers*.

Foi também mantida a utilização quer dos *identity Map’s* quer do carregamento *lazy* dos dados.



Esquema

O código desenvolvido nas aulas manifestava algumas limitações ao nível das chaves, nomeadamente quanto ao seu tipo, assumia chave inteira para todas as entidades, e à sua posição no *ResultSet*, a primeira. Por outro lado também não estava contemplada a possibilidade de chaves compostas. Para resolver estes problemas foram alteradas as classes *DomainObject* e *AbstractDataMapper* por forma a suportarem chaves genéricas nas formas *DomainObject<IDType, DType extends DomainObject<IDType, DType>>* e *AbstractDataMapper<IDType, DType extends DomainObject<IDType,DType>>*.

Desta forma tornou-se possível a utilização de chaves de qualquer tipo, sendo necessário no caso de tipos *user type*, que os mesmos redefinam os métodos *equals* e *hashCode* uma vez que os estes serão utilizados na comparação de identidade pelos *identity map’s*.

O método *doGetId(ResultSet rs)* em *AbstractDataMapper* foi tornado abstracto delegando nos *DataMappers* concretos a responsabilidade do correcto acesso ao *ResultSet* para obtenção da chave.

Foram implementados os métodos *insert*, *update* e *delete* em *AbstractDataMapper* por forma a cumprir com os requisitos do enunciado.

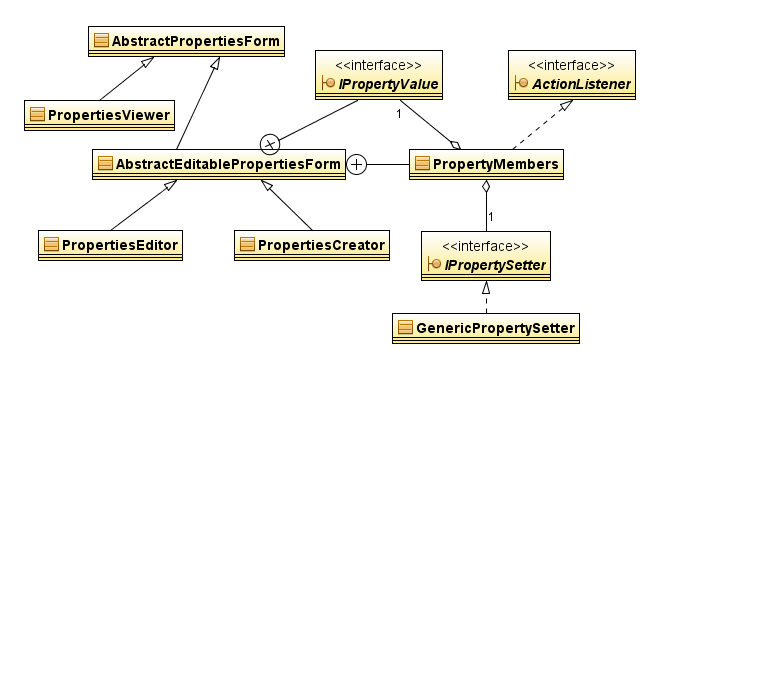
**Visualização**

O modelo desenvolvido na resolução das séries para a visualização de objectos assentava na utilização de um *PropertiesViewer*, com capacidade para mostrar as propriedades anotadas com a anotação *@VisibleProperty* de um dado objecto, e de um *PropertiesEditor* que acrescentava a possibilidade de edição dessas mesmas propriedades.

Para a correcta visualização e criação/edição dos objectos de domínio foi necessária a introdução de algumas alterações:

Para além da alteração a *@VisibleProperty* solicitada no Grupo 2 da 3ª Série foi necessário a introdução de mais um parâmetro, *isAutoGenerated*, por forma a distinguir as propriedades cujo valor deve ser criado pela base de dados, não podendo dessa forma ser apresentado para preenchimento ou edição.

Foi também criada a anotação *@PropertyValidator* a ser usada nos métodos responsáveis pela validação do novo valor a atribuir a uma determinada propriedade de acordo com o seu domínio de valores definido pela base de dados. Estes métodos devem estar presentes, sempre que se justifique, nos *DomainObject’s* concretos e devem ter no seu parâmetro *name* o nome da propriedade que validam.



Esquema

De acordo com o esquema 2, foi desenvolvido um modelo constituido pela classe abstracta *AbstractPropertiesForm*, que delega na suas classes derivadas a responsabilidade do correcto preenchimento do painel de propriedades através da definição do método abstracto *doPopulatePropsForm*.

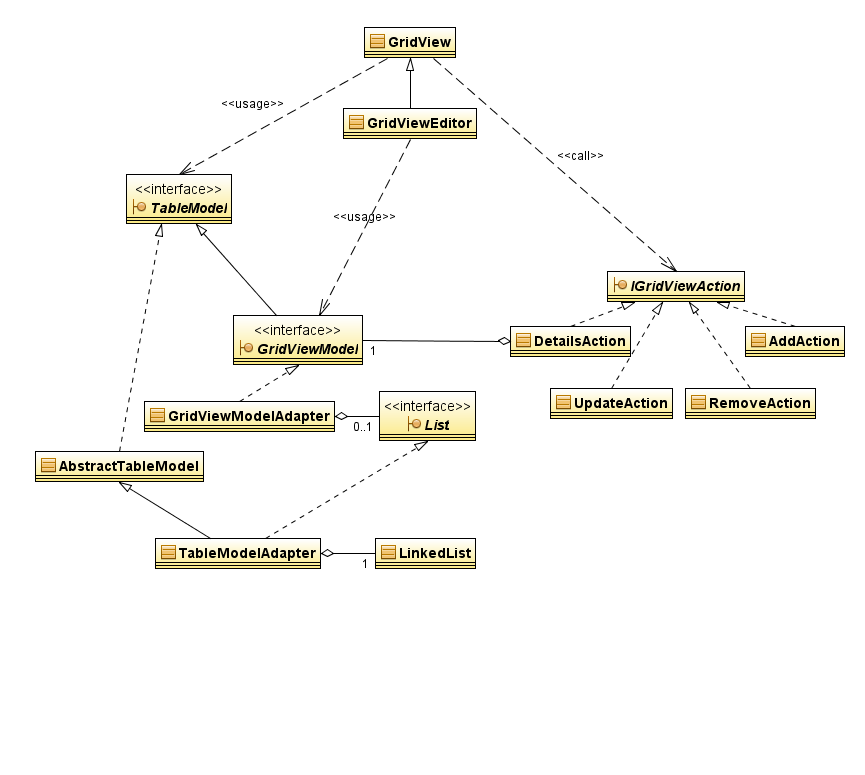
No caso de apenas ser necessária a visualização das propriedades, foi mantida a classe *PropertiesViewer*, mencionada anteriormente, já respeitando os requisitos da 3ª série, e foram implementadas as classes *AbstractEditablePropertiesForm*, *PropertiesCreator* e *PropertiesEditor* por forma a permitir a criação e edição de objectos, respectivamente.(Talvez explicar o PropertyMember?)

**MVC**

Quer para a implementação da *GridView*, quer para a *GridViewEditor*, conforme solicitado, e mantendo o desenho de *ListView*, o padrão MVC foi respeitado conforme o esquema 3.

Foram implementadas as classes *TableModelAdapter* e *GridViewModelAdapter*, conforme o solicitado, fazendo de adaptadores de *Collection<V>* para *TableModel* e *GridViewModel<V>*, respectivamente.(talvez explicar o porquê da adição de dois métodos à assinatura de GridViewModel)

Foram também implementadas as acções *DetailsAction*, *UpdateAction*, *RemoveAction* e *AddAction* por forma a permitir a total gestão dos objectos de domínio.



Esquema

Foi acrescentado ao trabalho a classe *MainView* for forma a evidenciar a utilização de *GridViewEditor*, bem como de todo o trabalho desenvolvido quer ao nível da visualização de objectos quer ao nível da sua correcta persistência ao nível da base de dados fazendo uso do modelo *Unit of Work*.