How to add your own design pattern definition in DPD

In Figure 1 you can see the classes which are related to the definition of design patterns.

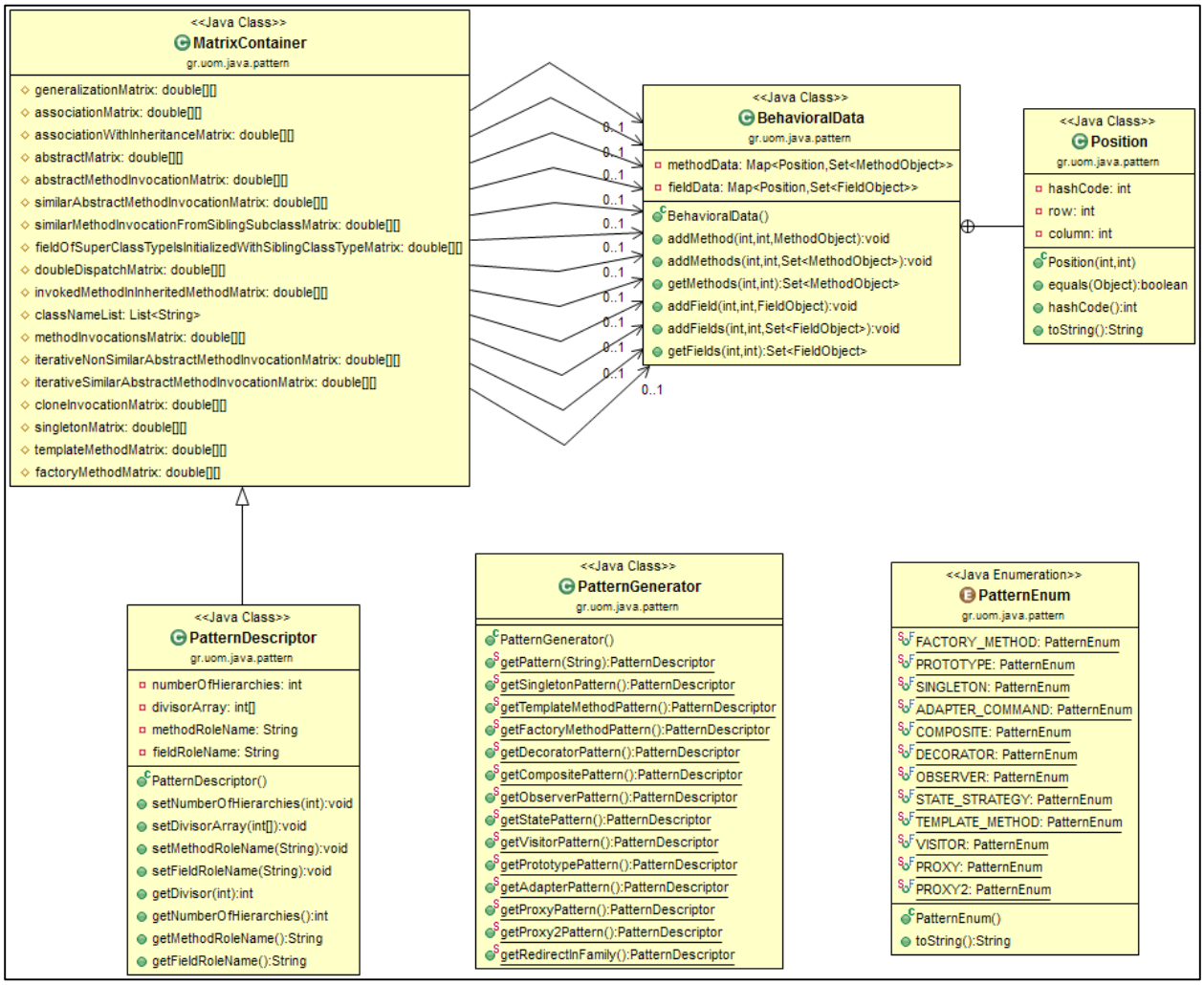

Figure 1: Pattern Description Architecture

To define a new design pattern, you basically need to create an instance of the **PatternDescriptor** class.

The definition of a **PatternDescriptor** involves 5 steps as shown in Figure 2.

```
public static PatternDescriptor getDecoratorPattern() {
    PatternDescriptor patternDescriptor = new PatternDescriptor();

    List<String> rowNameList = new ArrayList<String>();
1   rowNameList.add("Component");
    rowNameList.add("Decorator");
    patternDescriptor.setClassNameList(rowNameList);

    double[][] associationWithInheritanceMatrix = new double[2][2];
    associationWithInheritanceMatrix[1][0] = 1;
    patternDescriptor.setAssociationWithInheritanceMatrix(associationWithInheritanceMatrix);

    double[][] abMatrix = new double[2][2];
2   abMatrix[0][0] = 1;
    abMatrix[1][1] = 1;
    patternDescriptor.setAbstractMatrix(abMatrix);

    double[][] simMatrix = new double[2][2];
    simMatrix[1][0] = 1;
    patternDescriptor.setSimilarAbstractMethodInvocationMatrix(simMatrix);

3   patternDescriptor.setNumberOfHierarchies(1);

    int[] divisorArray = new int[2];
4   divisorArray[0] = 3;
    divisorArray[1] = 3;
    patternDescriptor.setDivisorArray(divisorArray);
    patternDescriptor.setMethodRoleName("Operation()");
5   patternDescriptor.setFieldRoleName("component");

    return patternDescriptor;
}
```

Figure 2: Steps to define a design pattern (see class PatternGenerator for more examples)

Step 1. Define the class roles participating in the design pattern

Create a List<String> with the names of the class roles participating in the design pattern. For example, for the Decorator design pattern, we have two main class roles, namely **Component** and **Decorator**. The order that you add the role names in the list is very important. In the example shown in Figure 2, the **Component** role corresponds to index 0, and the **Decorator** role corresponds to index 1. These indices should be used consistently in the next steps.

Step 2. Define the class role relationship/attribute matrices

Create a two-dimensional matrix for each kind of relationship between the roles defined in step 1. DPD supports the following directed relationships between two class roles $Role_A$ and $Role_B$:

1. **Generalization**: $Role_A$ directly extends or implements $Role_B$
2. **Association**: $Role_A$ is associated with $Role_B$ (i.e., $Role_A$ has a field having $Role_B$ or $Role_B$[] as a type)
3. **AssociationWithInheritance**: $Role_A$ is associated with $Role_B$ and extends/implements directly or indirectly $Role_B$
4. **AbstractMethodInvocationFromAbstractClass**: $Role_A$ is an abstract class and contains at least one method calling an abstract method declared in $Role_B$
5. **AbstractMethodInvocationFromConcreteClass**: $Role_A$ is a concrete class and contains at least one method calling an abstract method declared in $Role_B$

6. **CloneMethodInvocation**: $Role_A$ contains at least one method calling method clone() declared in $Role_B$
7. **DoubleDispatch**: $Role_A$ contains an abstract method X that has a parameter of type $Role_B$ and $Role_B$ contains a concrete method that has a parameter P of type $Role_A$ and calls method X though parameter P by passing "this" reference as an argument (the type of "this" is $Role_B$).
8. **SimilarAbstractMethodInvocation**: $Role_A$ contains a method X that calls an abstract method declared in $Role_B$ having the same signature with X
9. **SimilarMethodInvocationFromSiblingSubclass**: $Role_A$ contains a method X that calls a method declared in $Role_B$ having the same signature with X, and $Role_B$ is a sibling of $Role_A$ in an inheritance hierarchy
10. **IterativeSimilarAbstractMethodInvocation**: $Role_A$ contains a method X that calls inside a loop an abstract method declared in $Role_B$ having the same signature with X
11. **IterativeNonSimilarAbstractMethodInvocation**: $Role_A$ contains a method X that calls inside a loop an abstract method declared in $Role_B$ having a different signature compared to X
12. **InvokedMethodInInheritedMethod** (or Abstract method adaptation): $Role_A$ contains a method X that overrides an abstract method defined in a superclass of $Role_A$ and calls method Y declared in $Role_B$. $Role_A$ and $Role_B$ do not belong in the same inheritance hierarchy
13. **FieldOfSuperClassTypeIsInitializedWithSiblingClassType**: $Role_A$ contains a field Y having a superclass of $Role_A$ as a type, and Y is initialized with an instance of $Role_B$, and $Role_B$ is a sibling of $Role_A$ in an inheritance hierarchy

DPD also supports the following unary attributes for a given role $Role_A$, which are defined in a diagonal matrix:

1. **Abstraction**: $Role_A$ is an abstract class or interface
2. **StaticSelfReference**: $Role_A$ has a static field having $Role_A$ as a type
3. **TemplateMethod**: $Role_A$ contains a concrete method that calls an abstract method declared in $Role_A$
4. **FactoryMethod**: $Role_A$ contains an abstract method that is overridden by a subclass of $Role_A$. The overriding method in the subclass of $Role_A$ returns an instance of a class which does not belong to the inheritance hierarchy of $Role_A$

Step 3. Define the number of inheritance hierarchies involved in the design pattern

To specify the number of inheritance hierarchies involved in the defined design pattern, you can call method setNumberOfHierarchies(). This method takes 3 possible values:

- 0 means that the design pattern does not involve any inheritance hierarchy (e.g., Singleton)
- 1 means that all design pattern roles belong to a single inheritance hierarchy (e.g., Composite, Decorator, Proxy)
- 2 means that the design pattern roles belong to two different inheritance hierarchies (e.g., Observer, Adapter, Visitor, State)

This number affects the way DPD groups the classes of the system into subsystems in order to detect the defined design pattern in each subsystem.

Step 4. Define the number of relationships/attributes that each class role participates in

To specify the number of relationships/attributes that each class role participates in, you can create an array of integers, and pass it as an argument to method setDivisorArray(). For each class role corresponding to a given index, you should provide the number of relationships/attributes that it participates in based on the specifications of step 2. This number represents the maximum score that can

be achieved for a particular class role after performing the similarity computation for each one of the relationships/attributes it participates in.

Step 5. Define method and field role names (Optional step)

A design pattern definition can optionally have one method role and one field role. You can call setMethodRoleName() and setFieldRoleName() to specify the name of the method and field roles, respectively. Multiple method and field instances corresponding to these roles are automatically saved in BehavioralData objects when extracting the relationships and attributes defined in step 2 for the classes of the examined system (see class SystemGenerator for more details).