Observations on using the Design Pattern Detection Software.

1. Installing and running the software

TODO

2. ArgoUML

This documentation is based on version 0.34 of the ArgoUML-notation module in a default configuration.

This is not a manual how to use ArgoUML. We want to formulate some points calling attention when designing Class diagrams in ArgoUML and using the XMI-exports as input for the Design Pattern Detector.

a. Invisible elements

In the left-upper pane, you see all the elements belonging to the model. Different perspectives can be chosen. In the right upper pane, the model is shown in a diagram. When deleting an element directly from the diagram by selecting it and pressing the delete button, the element is not deleted in the model. To delete an element definitely, you can use the trash can in the properties tab, or right click on the element in the model perspective or the diagram and choose ‘Delete from Model’.

b. About compositions and aggregations

In ArgoUML there are special icons to create aggregations and compositions. In the XMI, these are associations with a special attribute.
Keeping this in mind, you should better use normal directed associations and set explicitly the multiplicities of both ends (cardinalities). Otherwise even the composite or aggregation relationships will have the default cardinalities of ‘1’ at each end.
You can of course use the aggregation and composition, but do not forget to set the multiplicities.

c. About realizations, generalizations and dependencies

The multiplicities of realizations, generalizations and dependencies cannot be set.
Although these are three kinds of relationships, they structurally differ.
A generalization can be set by hovering the cursor on a selected class and using the arrow(s) which appears, or by the icon in the toolbar. It is a relation between classes, eventually abstract classes. It is a child - parent relation.
A realization is in fact an abstraction with a stereotype <<realize>> added. This stereotype can be deleted without consequences on the relation (abstraction). An abstraction can be used between classes and between classes and interfaces.
A dependency is a property of a class or interface as is an operation or attribute. Hence in the model views you mostly find it as a child of the class or interface, and not as an independent relation.

d. About names

It is a good practice to give names, not only to classes, interfaces, attributes and operations, but also to the relations (I always use '*source-target*') and to the model itself. If you use different class diagrams, give them also meaningful names.

3. Designing a model

a. Interfaces or abstract classes

Many Design Patterns use one or more abstract classes which has to be concretized in concrete classes. In most cases, we use java-interfaces to model this. So when you have the choice between an interface or an abstract class, choose an interface for the model, , although in real-world applications implementations as a java abstract class can function as well.
An interface cannot implement another interface. So, in that situation you must choose an interface as parent and an abstract class as child, as in e.g. the case in the Observer Pattern.

b. Overriding a method

A class which implements an interface must override the method definitions of the interface. You must not set this explicitly in the model.
If a class must override a method of a parent class in the pattern's dfinition, this must be set in the model, defining the method both in the parent and in the child class

c. Associations and attributes

Associations are meant to set a semantic relationship between elements. In a java implementation this usually means a directed relationship, where the child has a private or public attribute with the type of the parent.
Because the notion of 'association' is very broad, we explicitly define attributes in the patterns definitions. So, rather than setting a vague association between two classes, you better model the attributes.
On the other hand, multiplicities of attributes or not taken in account, so if you want to set them (e.g. for an aggregation relationship), you have to use a directed association also.
TODO: ask Sylvia

d. Undirected associations

In a object oriented system, associations are mostly directed. So we discourage the use of undirected associations.
In the Design Pattern Detector, be aware of the fact that undirected associations are interpreted as bidirected associations, which may disturb the recognition of a pattern.

4. Interpreting the feedback

a. Reading the results

If a match has been found, you can see the solution as a match between the classes and interfaces of the Design Pattern and the classes and interfaces of the model under consideration.
If mismatches are found, or you do not agree with the match(es) the software found, you can read the feedback to find more information..
The feedback has two parts. There is a part considering the classes and interfaces of the system and a part considering the relations.

The classes/interfaces are ordered as they are investigated by the program, so first look at the classes which take your interest. I do study them following the scheme of the Design Pattern.
You can read per class/interface with which pattern class/interface there was a match, and if there was a mismatch, the reason why.

The second part gives an overview of the relations. You can read per relation with which pattern relation there was a match, and if there was a mismatch, the reason why.

b. Not analyzed or missing classes/interfaces and relationships

If there is a mismatch, the software stops searching or, if possible, searches other ways to find a match. That is why sometimes certain relationships or nodes are never compared.

To troubleshoot, you first can try to fix a wrong relationship or a class which you expected to match but which did not.. That means to remove the unexpected type(s) and add the missing one(s). Then you can rerun the analysis.

If a class or interface is not compared with one of the design pattern classes or interfaces, although you expect it, you must examine the relationships, incoming and outgoing. Because the analyzing algorithm crawls from one class to another using the relations, sometimes a class is never reached, because a relationship is wrong.

Mind that a pattern only can be found if there are at least the same amount of classes/interfaces and relations in the model under consideration and the pattern. So, if none of the classes are analyzed, look if a match is possible at all.

If you use your own design pattern template, study it carefully, because a mistake is made quickly (as I experienced myself).

c. Multiple patterns found

Some Design Patterns can support an unknown number of concrete classes, e.g the Abstract Factory Pattern supports multiple Concrete Factories (A and B) and multiple Concrete Products (1 and 2). In the Pattern we always start with a number of two.
If the model under consideration also has two concrete factories (fA and fB) and two concrete products (p1 and p2), the Design Pattern Detector will find 4 matches:
A = fA; B = fB; 1 = p1; 2 = p2
A = fA; B = fB; 1 = p2; 2 = p1
A = fB; B = fA; 1 = p1; 2 = p2
A = fB; B = fA; 1 = p2; 2 = p1

If the model under consideration has 3 concrete factories, the Detector will find 12 matches, because the three concrete factories can be split in tuples of 2 in three ways, each combined as above so 3*4 = 12
To calculate the possibilities in the Abstract Factory, you multiply the number of permutations with 2 elements of the concrete factories with the number of permutations with two elements of the abstract products:

P(#CF, 2) * P(#AP, 2) = (#CF!/(#CF - 2)!) * (#AP!/(#AP - 2)!)

So with four concrete factories and three possible abstract products there are
P(4,2) * P(3,2) = 4!/2! * 3!/1! = 12 * 6 = 72
matches.

d. Study the pattern definitions

The rules on which a pattern is evaluated are described in the patterns xml files. There you find an overview of the class and interfaces involved, as well as the different relationships. You also find the types (rules) to evaluate a possible match.
These patterns are based on the descriptions of the Design Patterns in the Gang of Four book, but also on the requirements of these patterns in a java implementation. The last ones are e.g. responsible for the rules on the presence of certain attributes or methods (and their types and parameters).
You can critically study these requirements and face them with your implementation.