A Generic Clustering Framework for Moose

Michael Meer

August 12, 2005

Abstract

Clustering helps with reengineering by gathering the software entities into meaningful and independent groups. The entities here can be any FAMIX entities, be it classes, methods, attributes etc. The affinity between two entities is calculated through the absolute difference of their properties and properties of the dependencies between the two entities; all the properties also have assigned weights. The clustering can be done by a range of clustering algorithms, including hierarchical and partitional algorithms. The result are groups of clusters, that can be examined through their quality metrics and if necessarily improved upon through another clustering run with adapted parameters.

This paper describes generic clustering framework for the Object Oriented Reengineering Environmnet Moose, developed in the Software Composition Group at the University of Bern [1].

Contents

1 Introduction

As the cost of hardware decreases and the complexity of software increases, the importance to maintain and improve existing software grows. Both for repairing errors (software maintenance) and for improving and adapting it to new requirements (software evolution), tools can facilitate the tasks for the software engineer. Such tools parse the code, build models of the code, measure the code, visualize the code etc. One of these tools is Moose, an extensible and scalable reengineering environment, developed in the Software Composition Group of the University Bern.

Engineers can use Moose to quickly get a grasp on a software system and build a mental image of it. In this student project we developed a clustering framework for Moose that enables a software engineer to automatically extract subsystems and guide their exploration of the software system.

There are many available properties that can play a role in coupling the entities of a software system. In our framework we let the user decide which properties matter most for his application, choose one of the many clustering algorithms available and also let him experiment with the parameters of the clustering process with ease.

To validate our subsystem extraction methods we also implemented a set of metrics to evaluate the results of the clustering, based on the cohesion of the clusters and their coupling with the rest of the system.

2 Solution

We split the responsibility for clustering FAMIX entities into two separate objects:

An Adaptor is making the bond between the entities we would like to cluster and the clustering algorithms. We kept the clusterers generic to allow us to cluster any kind of entities provided an adaptor exists. An adaptor takes a collection of entities and knows how to calculate the similarity between the entities. Besides the adaptor for FAMIX entities we also implemented an adaptor for two-dimensional points (the *Core.Point* class) to test the clustering procedures in an easy way.

A Clusterer takes an adaptor and uses its knowledge about the similarities between to entities to cluster these. The clusterers don't directly access the entities, their properties are encapsulated through the adaptor.

2.1 Adaptors

First we have the *AbstractAdaptor* as superclass of all Adaptors, which lists a few methods that the subclasses have to implement. These are mainly *distanceBetweenEntity: entity1 andEntity: entity2* and *middlePointOfEntities: aCollectionOfEntities*.

The class *PointsToClusteringAdaptor* simply uses the *dist: anotherPoint* method to calculate the distance between the points, and the geometrical center of gravity formula to calculate the middle of entities.

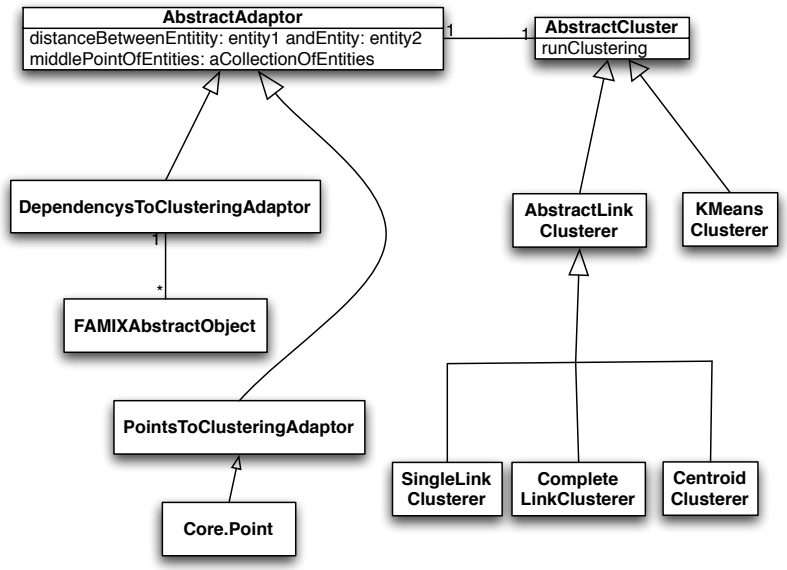

Figure 1: Class diagram of adaptors and clusterers, few of the clusterer implementations are omitted

To take FAMIX entities as input of our clustering we have the *DependencysToClusteringAdaptor*. This adaptor is much more complicated and also more configurable than the *PointsToClusteringAdaptor*, as there are many ways in which two FAMIX entities might be related to each other and the user of this framework is even able to prioritize them differently from case to case.

A *DependencysToClusteringAdaptor* calculates the similarity of two FAMIX entities by computing a list of absolute and relative distances between the dependencies, weigh the different kind of distances according to the users priorities and then aggregate them using an affinity scheme.

2.1.1 Absolute Distances

Two entities of the same FAMIX entity type have the same properties. For example they both have the property WLOC (Weighted Number of Lines of Code). An absolute distance is the absolute difference between the values of given property. Using two FAMIXClasses *FileServer* and *OutputServer* from the LAN test bundle the absolute distance for their WLOC property would be $|\text{WLOC}_{FileServer} - \text{WLOC}_{OutputServer}| = |11 - 13| = 2$.

So far this only works when all the entities to be clustered have the same class. It would also be possible to collect the intersection of two entities of different entity type.

2.1.2 Relative Distances

Relative distances are distances based on direct relationships between two entities, not on the comparison of the values of properties of the two entities themselves like above. Such a relative distance might be how many times a class

$entity_1$ calls methods of the class $entity_2$ or if $entity_1$ is a direct or indirect subclass of $entity_2$.

To model relative distances we introduced a new FAMIX Entity Type named FAMIXDependency. It is a directed relationship with an attribute *from* and an attribute *to*, both containing FAMIX entities. The exact entity type of such a dependency is modelled as the combination of the entity types of both *from* and *to*, i.e. for a dependency from a FAMIXClass entity to a FAMIXMethod the entity type will be *FAMIXClassToFAMIXMethodDependency*.

Through that variable entity type it is possible to define properties adjusted to the kind of dependency. For a FAMIXClassToFAMIXClassDependency a property *#IsSameNamespace* makes sense, but not for a FAMIXMethodTo-FAMIXAttributeDependency. There are a few of these properties already defined in the class *DependencyOperatorFactory*.

2.1.3 Affinity Schemes

We have now much information that we can use for clustering: the absolute and relative distances between entities on different properties plus the importance the user assigns to the different kind of distances. With affinity schemes we seek now to aggregate all this information into one number, the *affinity*. The affinity $A(i,j)$ is in the range $[0,\infty[$.

These are the currently implemented affinity schemes, as defined in [2]:

Unweighted Binary	$A_{UB}(i,j) = \sum_{cc=DI}^{MR} C_{cc}(i,j)$
Weighted Binary	$A_{WB}(i,j) = \sum_{cc=DI}^{MR} \alpha_{cc} C_{cc}(i,j)$
Unweighted Additive	$A_{UA}(i,j) = \sum_{cc=DI}^{MR} N_{cc}(i,j)$
Weighted Additive	$A_{WA}(i,j) = \sum_{cc=DI}^{MR} \alpha_{cc} N_{cc}(i,j)$
Unweighted Multiplicative	$A_{UM}(i,j) = \prod_{cc=DI}^{MR} C_{cc}(i,j)$
Weighted Multiplicative	$A_{WM}(i,j) = \prod_{cc=DI}^{MR} N_{cc} C_{cc}(i,j)$

where:

$\alpha_(CC)$ - weight associated with the property CC;

$C_{CC}(i,j)$ - boolean predicate, 1 if the two entities are coupled in the property CC, 0 if not;

$N_{CC}(i,j)$ - value of the distance of the property.

In the weighted / unweighted affinity schemes, terms with $N_{CC}(i,j) = 0$ are ignored.

The similarity $\mathrm{Sim}(i,j) \in [0,1]$ of two entities i and j is defined as:

$$D(i,j) = \begin{cases} 1 - \frac{1}{1+A(i,j)} & \forall i \neq j \\ 1 & \forall i = j \end{cases}$$

2.2 Clustering Algorithms

There are plenty of different clustering algorithms available nowadays, a good overview is presented by Jain et. al. [3]. Our clustering tool has a modular approach to make implementing a new algorithm easy.

A new clustering algorithm should be a subclass of *AbstractCluster* and implement the method *runClustering* to do the actual work and give back the results. If the algorithm to be implemented is a hierarchical clustering algorithm, there's a good chance that it can be done as a subclass *AbstractLinkClusterer* with only the method *computeDistanceForCluster: cluster1 and: cluster2* in need to overwritten. This is the case with all the hierarchical links already implemented by us.

Not all clustering algorithms can work with relative distances. Some of them like the *CentroidClusterer* and the *KMeansClusterer* need to calculate centroids of sets of points, this is not possible with relative distances between entities as there might be no transitivity:

$$\mathrm{Sim}(entity_\alpha, entity_\beta) \cap \mathrm{Sim}(entity_\beta, entity_\gamma) \nRightarrow \mathrm{Sim}(entity_\alpha, entity_\gamma)$$

These clustering algorithms do not call the adaptor's distanceFromEntity: toEntity: (the combination of absolute and relative distances), but its method euclidianDistanceFromEntity: toEntity: (which calculates only the absolute distance). All weights for properties on relative distances are ignored.

Table 1: Implemented clustering algorithms

Clusterer	**Classification**	**Used similarity metric**
SingleLinkClusterer	Hierarchical Cl.	Absolute and relative
CentroidClusterer	Hierarchical Cl.	Absolute
CompleteLinkClusterer	Hierarchical Cl.	Absolute and relative
GroupAverageClusterer	Hierarchical Cl.	Absolute and relative
WithinGroupsClusterer	Hierarchical Cl.	Absolute and relative
KMeansClusterer	Partitional Cl.	Absolute

The **Hierarchical Clustering algorithms** implemented here all work the same way:

1. Put every entity in its own cluster. Find the closest distance between all the clusters.
2. Join the two clusters with the biggest similarity.
3. If the specified number of clusters is already reached, abort and return the resulting clusters. If the number is not yet reached, again compute the closest distance between all the clusters and go back to step 2).

The only difference of all the algorithms is the definition of the distance between two clusters:

- The **SingleLinkClusterer** defines the distance of two clusters as the distance between their closest members.

- In **CompleteLinkClusterer** the distance of two clusters is defined as the distance between their two most remote members.
- The **CentroidClusterer** defines the distance as the distance between the centroids of the two clusters.
- In **GroupAverageClusterer** the distance is the average of all pairs of entities from the two different clusters.
- In **WithinGroupsClusterer** this distance is defined as the average of all pairs of entities in the two clusters were combined.

The **k-means** Clusterer as a partitional clusterer works differently:

1. **k** initial cluster centers are chosen, in this framework they are by default chosen randomly from among the entities to be clustered.
2. Assign each entity to the nearest cluster
3. As all entities are distributed to clusters, calculate the clusters centroids
4. If a defined convergence criterion is met, return the resulting clusters. Here the convergence criterion is that no entity switched its cluster in the last run. If the criterion fails to hold, we start again at step 2).

2.3 Cluster Metrics

The result of the clustering process is a *ClusterGroup* object containing several *Cluster* objects themselves containing all the fed entities. The user wants now to judge the quality of the individual clusters and the whole clustered system. For that he can rely on one hand on his knowledge of the domain and see if entities that semantically belong together also now appear in the same cluster, or if there are entities spread to different clusters that should belong together. On the other hand he can make use of clustering metrics such as cohesion and coupling.

Both the *ClusterGroup* and the *Cluster* class are subclasses of *MSEEnumeratedGroup*, which allows me to define properties on them modelled after several clustering metrics found in the literature, especially in [4] and [5]. These properties are defined in *ClusterGroupOperatorFactory* respectively in *ClusterOperatorFactory*.

We have already mentioned above the method $\mathrm{Sim}_W(x, y)$ to calculate the similarity between two entities x and y.

Now we look into the cohesiveness of a cluster; the more dependent its entities are, they more cohesive is a cluster. Thus in [4] cohesiveness is defined as the average similarity of all distinctive pairs of entities in a cluster P (where m is the number of entities in P and p_x are the entities):

$$\mathrm{Coh}_\mathrm{W}(P) = \frac{\sum_{(>)} \mathrm{Sim}_W(p_i, p_j)}{m(m-1)/2}$$

Coupling is the metric for the dependency of the cluster with the rest of the system. In [4] Kuhn defines it as the average similarity of all entities inside the

cluster P with all the other entities in the system; n is the number of entities not in the cluster P.

$$\mathrm{Coup}_W(P) = \frac{\sum_{(i,j)} \mathrm{Sim}_W(p_i, p_j)}{m \times n}$$

A good cluster should have high cohesion within and low coupling with the rest of the system. Kuhn tries to combine the two metrics for cohesion and coupling into one quality metric by subtracting them.

$$Q_W(P) = \mathrm{Coh}_W(P) - \mathrm{Coup}_W(P)$$

Lethbridge and Anquetil [5] see problems with quality metric:

- According to their experiments both the cohesion and the coupling increased with better coupling of a system, not only the cohesion as was to be expected.
- As cohesion increases more quickly than coupling, the latter easily becomes irrelevant.

They set out to find a better quality metric by normalizing the above mentioned influences. First they calculate the baseline below that the cohesion and coupling of a system should not fall:

$$B_W = \mathrm{Coh}_W(S)$$

$\mathrm{Coh}_W(S)$ is here the cohesion of the whole system, so the average similarity of all the possible entity pairs. Completely random clusters will tend to have $\mathrm{Coh}_W(P) = \mathrm{Coup}_W(P) = B_W$.

On the other hand we need a ceiling that the cohesion and the coupling for no cluster can exceed. The ceiling for the cohesion is defined as the highest similarity of any pair of entities in the whole system:

$$C_{\mathrm{coh}_W} = Max_{i>j}(\mathrm{Sim}_W(q_i, q_j))$$

The ceiling for the coupling is defined as the maximum coupling that any possible cluster with just one entity in this system has:

$$C_{\mathrm{coup}_W} = Max_{1,n}(\mathrm{Coup}_W(P1))$$

No other cluster could have a higher coupling than that, just adding one entity to this cluster would lead to a lower coupling. Now we have all the parts to construct the normalized cohesion and coupling:

$$\mathrm{NCoh}_W(P) = \frac{\mathrm{Coh}_W(P) - B_W}{C_{\mathrm{coh}_W} - B_W}$$

$$\mathrm{NCoup}_W(P) = \frac{\mathrm{Coup}_W(P) - B_W}{C_{\mathrm{coup}_W} - B_W}$$

The new normalized quality metric is put together similar to the old one:

$$NQ_W(P) = \mathrm{NCoh}_W(P) - \mathrm{NCoup}_W(P)$$

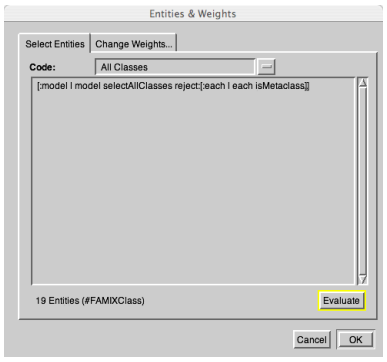

Figure 2: Selection of the entities

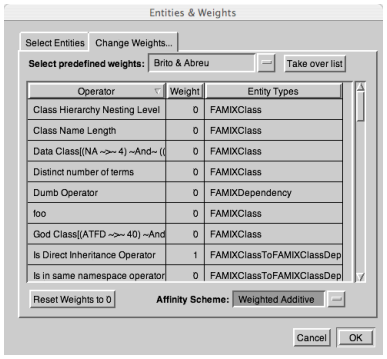

Figure 3: Configuring the weights, selecting affinity scheme

3 Measuring Clusters

We implemented a graphical user interface to allow the user to experiment with all variables and come up with the best configuration for his tasks.

1. The clustering process is started using the right click menu on any MSE-Model in Moose.

2. As seen in the appearing window (figure 1) the user has to define which FAMIX entities he wants to cluster. Either he chooses one of the pre-defined selections in the list (implemented in the *OperatorWeightEditors* method *initializePredefinedWeightLists*) or he writes a block to select them himself. He has to press *Evaluate* to take over this selection.

3. In the *Change Weights* tab (see figure 2), the user can assign a weight to the properties that define the absolute and relative distance between any two of the entities. Any property that has an assigned weight of 0 won't be considered for the clustering. The user can use a pre-made list of weights (implemented in *initializeWeightList* in the class *OperatorWeightEditor*) or adapt them to his application.

4. In the same window the user also has to choose an affinity scheme (as listed in table [2.1.3]) to aggregate all the weighted properties to one number,

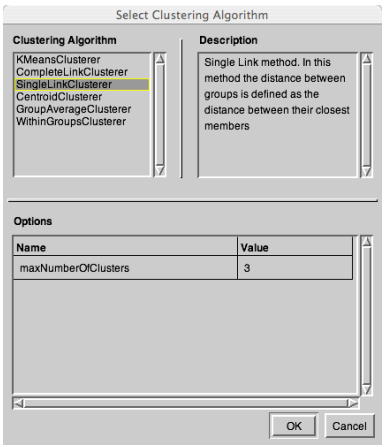

Figure 4: Selection and configuration of clusterer

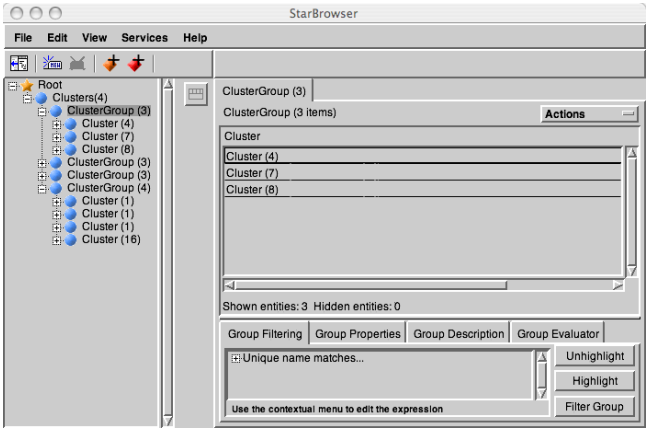

Figure 5: View of results in StarBrowser

the similarity of two FAMIX entities.

5. In the next window (see figure 4) the user has to choose which clusterer to use and can adapt its default configuration.

After some processing time a StarBrowser window with the resulting ClusterGroup will open up, (see figure 3. The user is able to manipulate everything as he is used to it from Moose, especially view properties for the Clusters and ClusterGroups and highlight values above a certain threshold etc.

The user can restart the clustering using the right-click menu on the ClusterGroup. The windows for editing the entities, weights and clusterers open up again with the same configuration as was used for the first clustering. The result of the new clustering will appear in the same StarBrowser window, but possibly the user has to refresh Starbrowser for the new ClusterGroup to appear. This is done through collapsing the Root classification and opening it up again.

Also through the right-click menu it is possible to inspect the adaptor used in this clustering, which is handy when a user wants to find out about the used

configuration in a specific clustering.

4 Conclusion

As B.S. Everitt said, "a clustering is neither true or false" ([6]). This is also the case when extracting subsystems of a large software system. For example it might make sense that a software engineer bundles utility classes for a software system in a common package. A clustering approach wouldn't find this package as the utility classes are not coupled enough.

So a software (re-)engineer as the user of our Moose clustering framework needs to have the ability to experiment and explore the software system to find the most useful clustering for his task. We deliver this through a simple graphical user interface to configure the clustering process, the possibility to restart a clustering with adapted parameters and a view of all the results side by side in Starbrowser, enabling the user to compare his different clustering attempts also with quality metrics.

Additionally the framework is easily extendable in different ways:

- New properties of FAMIX entities in Moose automatically appear also in our framework. Per default they have the weight zero and are therefore ignored until assigned a different weight by the user, so they don't mess up the results of restarted clusterings.
- There are still many meaningful properties for FAMIXDependencies to be defined (in the class *DependenciesOperatorFactory*), so far there are only a few example properties.
- The literature such as Trifu [3] describes many more clustering algorithms that could be implemented in our framework without much hassle. With the Single Link Clusterer, the Complete Link Clusterer and the **k-means** Clusterer we already implemented the three probably most used clustering algorithms.

References

[1] Stéphane Ducasse, Tudor Gîrba, Michele Lanza and Serge Demeyer: *Moose: a Collaborative and Extensible Reengineering Environment*

[2] F. Brito e Abreu, G. Pereira, P. Sousa: *A Coupling-Guided Cluster Analysis Approach to Reengineer the Modularity of Object-Oriented Systems*

[3] A.K. Jain, M.N. Murty, P.J. Flynn: *Data Clustering: A Review*

[4] T. Kunz, J. Black: *Using Automated Process Clustering for Design Recovery and Distributed Debugging*

[5] T. Lethbridge, N. Anquetil: *Experiments with Coupling and Cohesion Metrics in a Large System*

[6] M. Trifu: *Architecture-Aware, Adaptive Clustering of Object-Oriented Systems*