Abstract
Several coupling frameworks have been introduced in the literature to identify and measure the design complexity of object-oriented software systems. The coupling metric COF presented by the unified framework considers inheritance, polymorphism, method overriding, and direct methods of invocations to identify possible interactions in the system that contribute to the software complexity. We evaluate the COF metric in the context of Java interfaces. Interactions are identified, the effectiveness of the model is discussed, and an extension of the model is offered for more accurate measurement of complexity.

1. Introduction
Because complexity relates directly to increasing cost, managers are continuously striving for better quality software systems. The coupling models presented in the literature show many possible interactions that can occur between objects in the software systems and offer metrics to measure its design complexity. The notion is that high coupling between objects increases complexity and this reflects in increasing design cost. Therefore low coupling should be aimed for when designing OO software. The COF metric offered by the unified framework [1, 2] is based on class method invocation and class attribute referencing and provides a measurement based on the interactions of all classes in the system. It considers relationships through polymorphism, method overriding, and inheritance. While inheritance introduces more interactions among classes in the system and therefore increases complexity, the COF metric does not count coupling resulted from inheritance. This is because inheritance is a good thing, it gives software reusability and that the benefits outweighs the cost. This poster presents relationships between Java interfaces and all classes in the system. It identifies new interactions resulting from the use of interfaces that impact the coupling measurement. We further justify that interfaces are a good thing and that they should get the same treatment as the inheritance relationships meaning that class connections with the interfaces should not be counted as coupling. Finally, we offers an extension of COF metric to better represent the coupling measurement of the software system.

2. Background
The coupling factor (COF) metric introduced by the unified framework (Figure 1) represents the actual number of client-server relationships between classes that are not related via inheritance to the maximum possible number of such client-server relationships. It is normalized between 0 and 1 to allow comparisons between systems of different sizes. To obtain the actual number of couplings, the metric goes through each class in the system (considering all its methods and attributes) and finds its relationships to all other classes in the system as described by the numerator of the model.

\[ COF(C) = \frac{\sum_{d \in C} |\text{uses}(c, d)|}{\sum_{c \in C} |\text{Descendants}(c)|} \]

Figure 1: Coupling Factor (COF) Metric Model

The decision to whether count the interaction between two classes as one coupling is based on the \( \text{uses}(c, d) \) predicate (Figure 2). The predicate requires that when a method in class \( c \) invokes a method in class \( d \), then the implementation of that method has to be inside class \( d \) and not declared through inheritance. If the relationship is via inheritance, the predicate \( \text{uses}(c, d) \) returns false. If the predicate returns true, the metric counts the relationship as 1 coupling meaning that the class \( c \) is coupled to class \( d \) through either method invocation or attribute referencing.

\[ \text{uses}(c, d) \iff (\exists m \in \text{M}(c) : \exists m' \in \text{M}(d) : m' \in \text{PIM}(m)) \lor (\exists m \in \text{M}(c) : \exists a \in \text{A}(d) : a \in \text{AR}(m)) \]

Figure 2: predicate \( \text{uses}(c, d) \)
3. Java Interfaces

Java interfaces allow only method definitions and constant attributes. Methods defined in the interfaces cannot have implementations in the interface. Classes can “implement” the interface by providing bodies for the methods defined in the interface. An interface is a contract between a client class and a server class. It helps to decouple the client from the server. Any intended change on the methods defined in the interface will impact both the client and server classes. Possible changes are as follows: 1) changing the name of a method, 2) changing the signature of a method, and 3) changing the return type of a method. There are two other possible changes that worth noting. If a new method is added to an interface, this will also impact the server and client classes that currently use or implement the interface. On the other hand if the implementation detail of a method inside a server class is changed, this change only effects the client class and not the interface. This specific case is more a code issue than a design issue and therefore it is not a concern in this evaluation.

To evaluate whether the current COF metric measures the true design complexity of a software system with interface implementations, the predicate uses(c, d) needs to be evaluated. The predicate requires that the client invocation of a method or attribute reference to be explicitly implemented in the server class. When the predicate returns true, it means that there is a connection between class c and class d and that the number of possible changes to the software is two, one to the server class and one to the client class. When the predicate returns false, it means that there is no connection between a client class and a server class and therefore no coupling exists between classes c and d. Consider the possible changes in Java interfaces as discussed above. When a designer adds a new method to the server class, or changes the signature or the return type of an existing method, or changes the name of the method, all impact the interface. It first needs to make the changes in the interface, then make the required changes to the server class, and finally make the appropriate changes to the client class. So a change with an interface connection requires three changes. Most development organizations have different development teams where one team works on the client side and another works on the server side. In fact this is exactly what the interfaces are intended to enable and allow teams to work independently using a defined contract. If either the client or the server class tries to make a change that does not comply with the contract, things will break.

4. Discussions and Conclusions

The current COF metric considers the connection between a client and a server class as one coupling. In systems with interface implementations, the client/server class relationships have two connections. This means a change in the server class which impacts the interface, will require three software changes, one to the server, one to the interface, and one to the client. Two issues are discussed to decide how and why the existing metric should be considered for extension. First, if the metric is extended to count for connections with the interfaces then the metric represents higher coupling among classes, mis-representing the use of interfaces as a bad thing because it increases complexity. Where in fact interfaces are designed to allow building software based on contracts, and increase extendibility and allow interchangeable components. On the other hand, if the COF metric stays unchanged, this could also be flawed in that it misrepresents the true interactions between classes and the use of interfaces in the system. Perhaps we can extend COF metric in such a way that it treats interfaces as it treats inheritance. We recall that the COF metric does not count coupling due to the inheritance because inheritance offers software reusability even though it introduces higher coupling. The benefits outweighs the costs and that class interactions due to the inheritance should not and is not counted for coupling.

An alternative solution is offered to extend the existing model to not count coupling due to the interactions between classes that have connections with the interfaces. In this case let represent I as a set of all interfaces in the object-oriented system. Then for i ∈ I the predicate uses(c, d, i) is to find if there is a connection between interface i and class c class d. If there is a connection, then do not count it as coupling. This may in fact give a lower and more accurate picture of coupling in the system. The extended predicate is shown in Figure 3.

5. References
