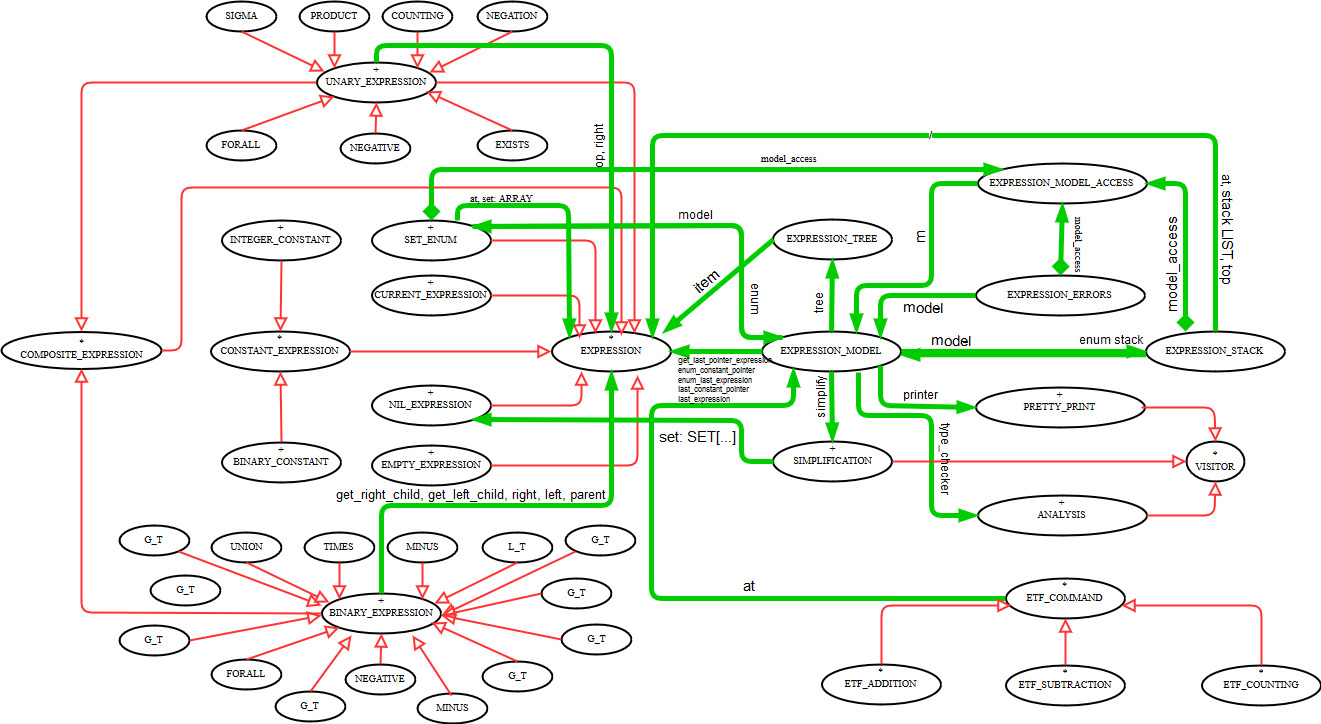
Fall 2018

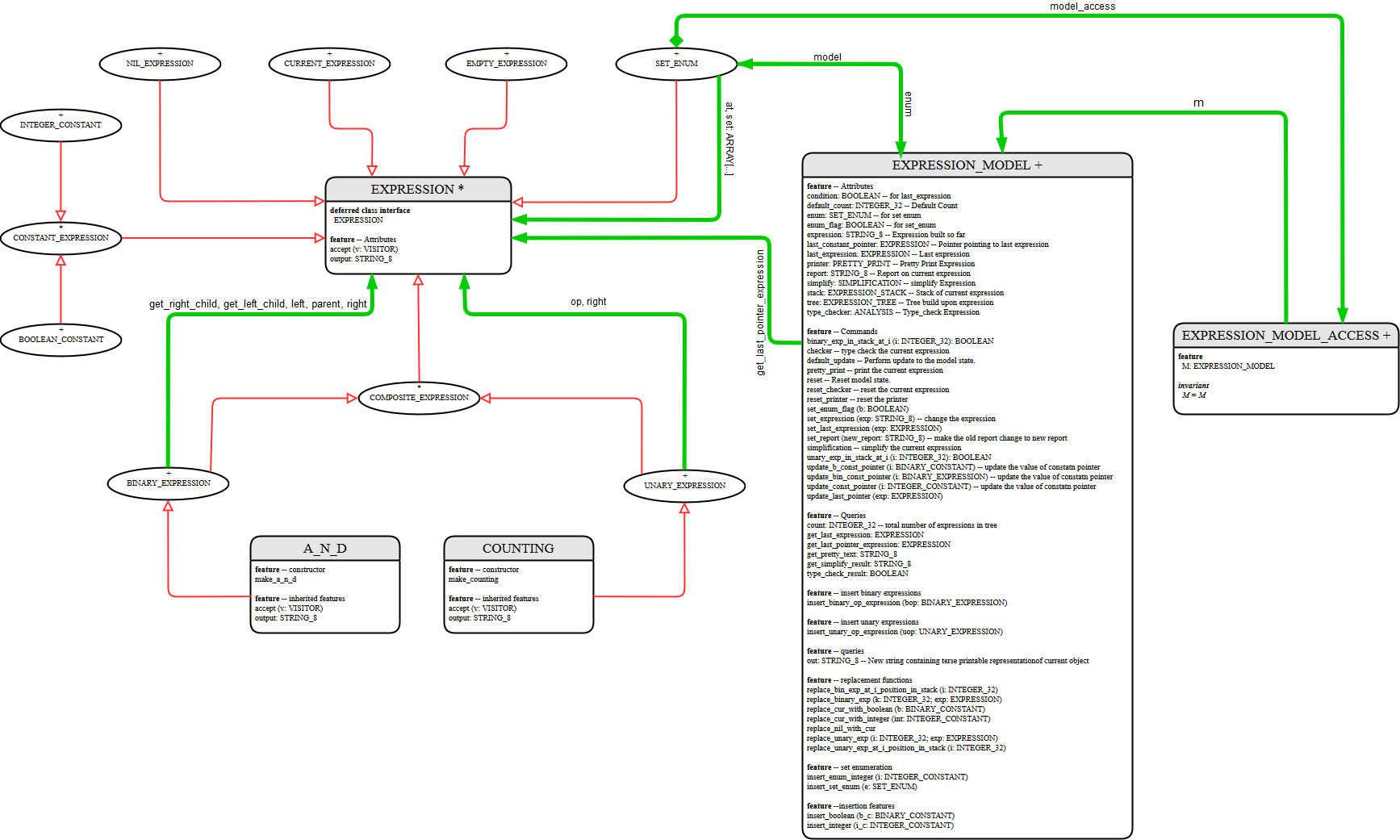
EECS 3311

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Mustafa Alsoufi | Fahad Qayyum | Ahmed Sayed Ahmed |
| ID | 213970173 | 215287253 | 206899736 |
| CSE login | mzuhair | Fahadq | ahmedyrk |

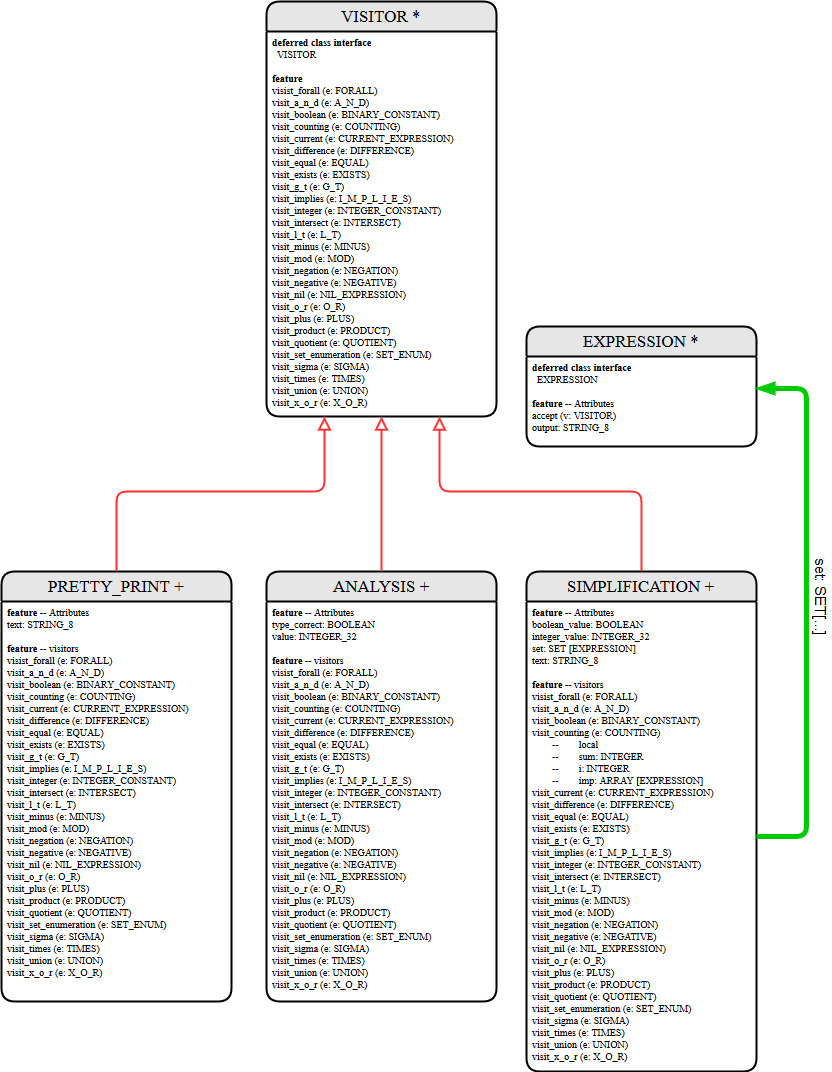
**Bon 1 - Relationships Between All Relevant Classes**



**Bon 2 - Architecture Design Modeling Expression Language Structure - Critical Classes In Expanded View (With Contracts)**



**Bon 3 - Architecture Design Modeling Three Language Operations - Critical Classes In Expanded View (With Contracts)**



**How Our Design For The Expression Obeys The Following Design Principles:**

* Information Hiding:

Information hiding is a design decision principle which ensures a lot less modification when there is a change of design decision especially when the computer program is long. By applying information hiding principle to the program, supplier secures the implementation details from its clients and they cannot access it directly, so a lot less risk is involved and thus the confidentiality, efficiency, implementation cannot be altered by the any clients of its class in any way. We can also restrict certain clients to have access to the secrets of the program and deny access to others. So therefore, information hiding is an indispensable part of any good design.

Our program satisfies this principles by restricting access to the most frequently accessed attributes of the class. In our main class i.e Expression\_Model we have hidden our attributes from its clients by exporting them to {NONE}, so the attributes like TREE and STACK and other flags, which are accessed within the class often, to update the expressions and to keep track of the expression built so far, the benefit we achieved with this is that we restricted our clients to have the direct access to its building secrets, so that they may not alter something unknowingly and getting an absolutely wrong answer. Similar to the expression\_model class, this feature is honored in other classes too

In our design, to give clients access to its basic structure while still achieving the perfect functionality, commands and queries have been incorporated in the design. the benefit of having this in our program is that if later the suppler (i.e us) decide to choose a different design e.g. implement TREE using array instead of linked\_list, the supplier would not have to change any of their implementations since they are calling the same commands and queries which hold the actual change, so functionality without losing efficiency is properly maintained in the program. It also maximizes the software reusability, since the client doesn’t have to care about if the design is change later on, since this is irrelevant for them, at their end they simply provide input and get results.

Inside the expression\_model, the queries and the commands and attributes which needs to be used by only a certain class e.g. ETF\_COMMAND are exported only to that class i.e only those specific class have access to it and if an unauthorized class would want to gain access to those feature, then they wouldn’t be allowed to do so whatsoever. So the software integrity is maintained

Therefore, by using information hiding we achieved benefit both for us and the client, as being a supplier, we would not have to worry if client can change delicate parts of the data since only relevant classes are given access. and for the clients they don’t have to worry if the design decision is changed later by the supplier since they were gaining access to the implementation only through the commands and queries.

* Single Choice Principle:

Single choice principle states that whenever a change is required, there should be a minimum number of places where you should make a change, typically one place. This necessitate that the shared code among different classes be placed in a single place so if later any amendments needs to be made, it can be done in one place.

For our design pattern, we have picked the visitor design pattern among different patterns which obeys the single choice principle.

Our program honors the single choice principle as it is very convenient to incorporate new operations to our design as there is only one place where we need to make the change and the other parts would be intact, up and running. For example later i want to introduce pattern\_generator as an operation apart from analysis pretty\_print and simplification, to make this change I wouldn’t have to make a change in any of the class and all I would have to do is to make it a descendant class of the VISITOR and implement the functionality, the number of places to make a change would be one, more or less. Therefore in our design it is easy to achieve this functionality of adding operations to the current design with minimal number of changes required elsewhere, hence the single choice principle is obeyed.

Polymorphism and dynamic binding implemented in the code further enhance the single choice principle and the code functionality and efficiency.

From structure point of view, Our program is in hierarchical order in a way that if later we want to add some other operation to our current model than all required is to make it a descendant of a proper type class and implement the functionality there. So, the single choice principle is well taken care of when making a design choice.

* Open-Close Principle:

Open close principle states that our program should be close to anything that’s not subject to change later on and if there is something that we might have to change later on that its must remain open i.e it should be open for extension but close for modification.

As a supplier it is crucial to decide which part of the program should be open and which part to keep close. As part of a design decision, supplier can either choose to have their operations open for extension and structure close for modification or can either have their operations close for modification and structure open for extension. Here given the problem nature for the program we choose to have our operations open for extension and structure closed for modification.

Open-closed principle help us determine if visitor pattern is applicable or not. In our design decision, visitor pattern conform most to this principle as it works best where the operations are open and structure is close. Therefore, in our program the classes like composite\_expression and constant\_expression and its descendants and ancestors form our closed structure and adding operations i.e a descendant to a visitor would form our open for extension part.

Our program satisfies the open-close principle where operations are open for extension and if we want to perform another operation on the binary expression than it would require me to just create a new operation class and make it a descendant of the visitor class and thus also obeying the single choice principle but if we try to add something to a structure of program e.g. add dot product then this would require me to make changes in VISITOR class and all its descendants and elsewhere too and this would also violate the single choice principle that we are trying to maintain as a benefit for both supplier and client.

Another benefit of adhering the open close principle in the program is to streamline code maintenance and minimize the risk of breaking the existing code base (single choice principle). Here in our program if we need to write a new feature eg print\_stack\_contents, we would use the given code as a base to implement this feature rather than modifying the old features already in the program and changing program functionality.

* Uniform Access Principle:

Uniform access principle allows the client to use service regardless of the need to know how they are implemented and this principle give supplier the complete freedom to how to implement it either storage or computation. Regardless of computation or storage the supplier can always switch from one to another since the access to the services is uniform.

It is important to have this principle in a good design since we want our client to access the feature normally even after we switch from computation to storage or vice versa, the program behavior should not change.

Unlike other languages, in Eiffel, the uniform access principle states that all the features of a class should be available through uniform notion, whether implemented through storage or computation eg if we have a attribute name “expression” in our program and another feature name “expression” as a query, the way to access that feature is uniform regardless it’s a query or attribute, the syntax is same. So, Eiffel by default is uniform in feature access.

In our design choice we have made that features computation that are frequently being accessed such as last\_exp\_curr, last\_exp\_binary, last\_exp\_unary etc and the other features as storage which are not so frequently accessed eg grow\_tree, replace\_and\_grow etc for efficiency purpose and on need-bases.

Since in the program the way to call them is uniform, therefore this uniform principle is satisfied in our program.