EECS3311
Fall Semester 2019
Analyzer Project

Jonathan Lai CSE Login: yakabuff
Anas Butt CSE Login: anas2ab
Submit Account: CSE Login: anas2ab

Design Principles:

Single Choice Principle:

We carefully followed single choice principle in our implementation of the visitor pattern. Instead of executing our operations like type checker and java code generator in every class of our language structure, we had all the different operations inheriting the visitor class. This prevents us from having unrelated operations in every class. For example, in our language structure, if we wanted to add the operation “Evaluate”, we would simply make a class that inherits Visitor that visits the integer_constant, binary operation and other components in the expression tree. It would not be necessary to implement evaluate in every single descendant of the expression tree.

Single choice was also used in polymorphism. We had deferred classes that had defined some functionality. As a result, if we changed the feature in the deferred class, the changes would be seen across all descendant classes. This is seen in our composite pattern as our deferred COMPOSITE class has a “children” linked list. As a result, all subsequent children classes that inherit from COMPOSITE like unary_op and binary_op also have a “children” linked list which allows us to recursively navigate the composite tree.

Information Hiding Principle

Information hiding was used primarily with regards to inheritance and polymorphism. Encapsulation and information hiding is vital in object oriented programming. This is to ensure the client does not have access to the inner working and implementation of the supplier. For many of our attributes, we have the feature export setting set to {NONE} so that other classes cannot access the contents of the feature without contacting it through. Our internal array data structures for our 'my_class' class and our command classes are all encapsulated to ensure clients cannot simply access the attribute and read the data. This forces the client to use the queries and commands which adheres to a set of postconditions and preconditions to ensure correct usage. This principle also allowed for flexibility in our project. As our class does not know whether or not it has a command, expression or a query, it allows for proper abstraction and reusability.

Uniform Access Principle

Uniform access principle is that accessing a field or method in a class should not differ. This is beneficial as you will not damage the client's code if you ever wish to change the way of accessing a field with a getter or simply assigning it to an attribute. An example of this is in our my_class class. We have attributes like name and feature_list but could easily be transformed into a query function without impacting how the client uses the feature.

Open Close Principle

Open choice principle is when the structure of the program is open for extension but closed for modification. This means that we can add different operations on top of our existing operations(java_code_generator, type checker) but are not allowed to edit the existing language constructs like the different binary operations and unary operations. This pattern is seen in our visitor/composite pattern.

Polymorphism Principle

Polymorphism was used extensively throughout the project primarily with regards to the visitor pattern. When using the visitor pattern, each descendant of the visitor class inherits every feature can provide their own implementation depending on the desired operation. An example of this can be seen in the form of double dispatch. The client object's accept method receives a visitor object which, depending on its dynamic type, calls the correct visit method. Our language structure also used polymorphism to allow for multiple versions of each feature to be called depending on the dynamic type. Each class inherits features from the feature class but all have different implementation.

Cohesion Principle:

Every class was organized in such a way that only relevant code was put in a single class. Classes are not polluted with different concepts and objectives and are put in place where the features are related. This is observed in the visitor pattern as every operation was designated its own class and had code that was relevant to its specific operation. This was also demonstrated in the expression structure. Instead of combining all the binary and unary operations together, we split it up into its own individual class for reusability.

Patterns

Singleton Pattern

Singleton pattern was used in the ETF structure. ETF commands all communicate with the analyzer through the ETF_MODEL_ACCESS class. This restricts the model to 1 instantiation and ensures that every operation is coordinated correctly.

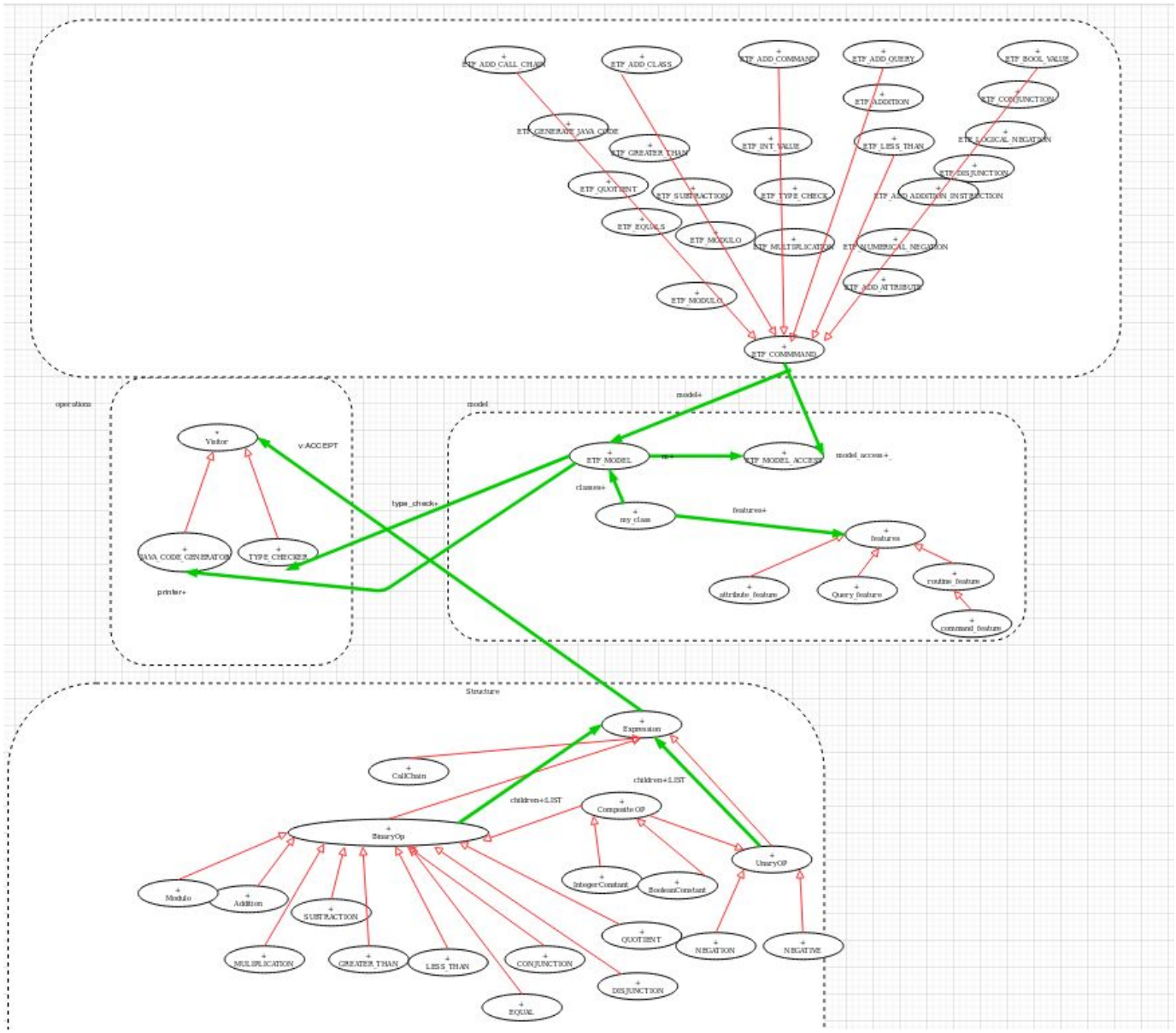
Visitor Pattern

We used visitor pattern to follow single choice and cohesion principle while using composite pattern. It allows to easily insert an operation at the cost of not changing the structure. The operations PRETTY_PRINT and TYPE_CHECK inherit the VISITOR class which call the visit features.

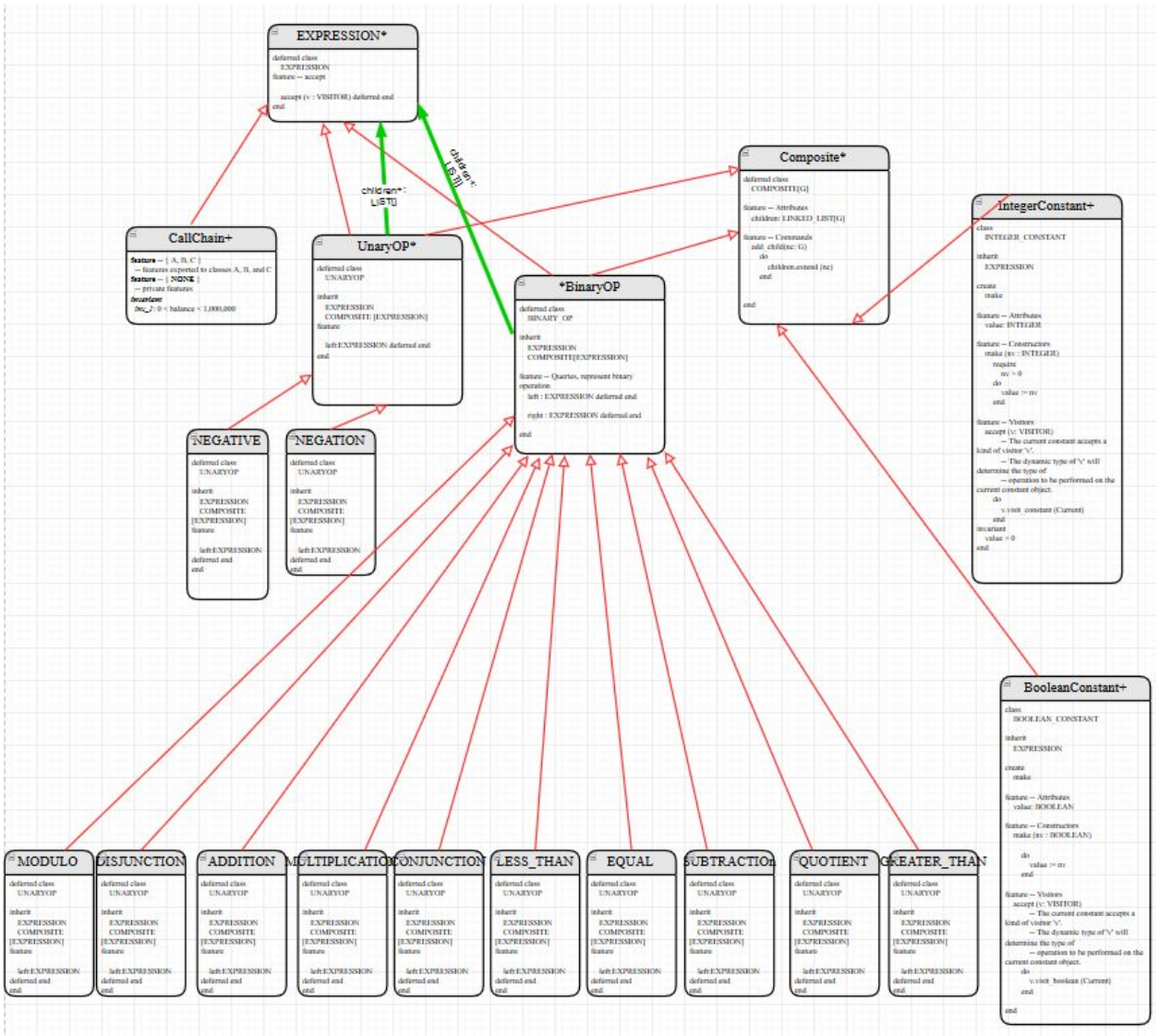
Composite Pattern

Composite pattern was used in our expression structure. Each expression became composed of INTEGER_CONSTANT, BOOLEAN_CONSTANT, binary_op, unary_op and call_chain. The program structure was also composed of classes(my_class), commands, queries and attributes. The entire expression structure was represented as a tree that could be recursively navigated until the base case was met. Once met, the value of the constant was obtained which could be used to check types and print out values. Polymorphism is once again observed here as different operations are performed depending on the dynamic type. If the node is a composite, it would keep attempting to navigate along the left or right tree whereas it would return the value once reaching the constant.

Bon Diagram 1



Bon diagram 2



Bon diagram 3