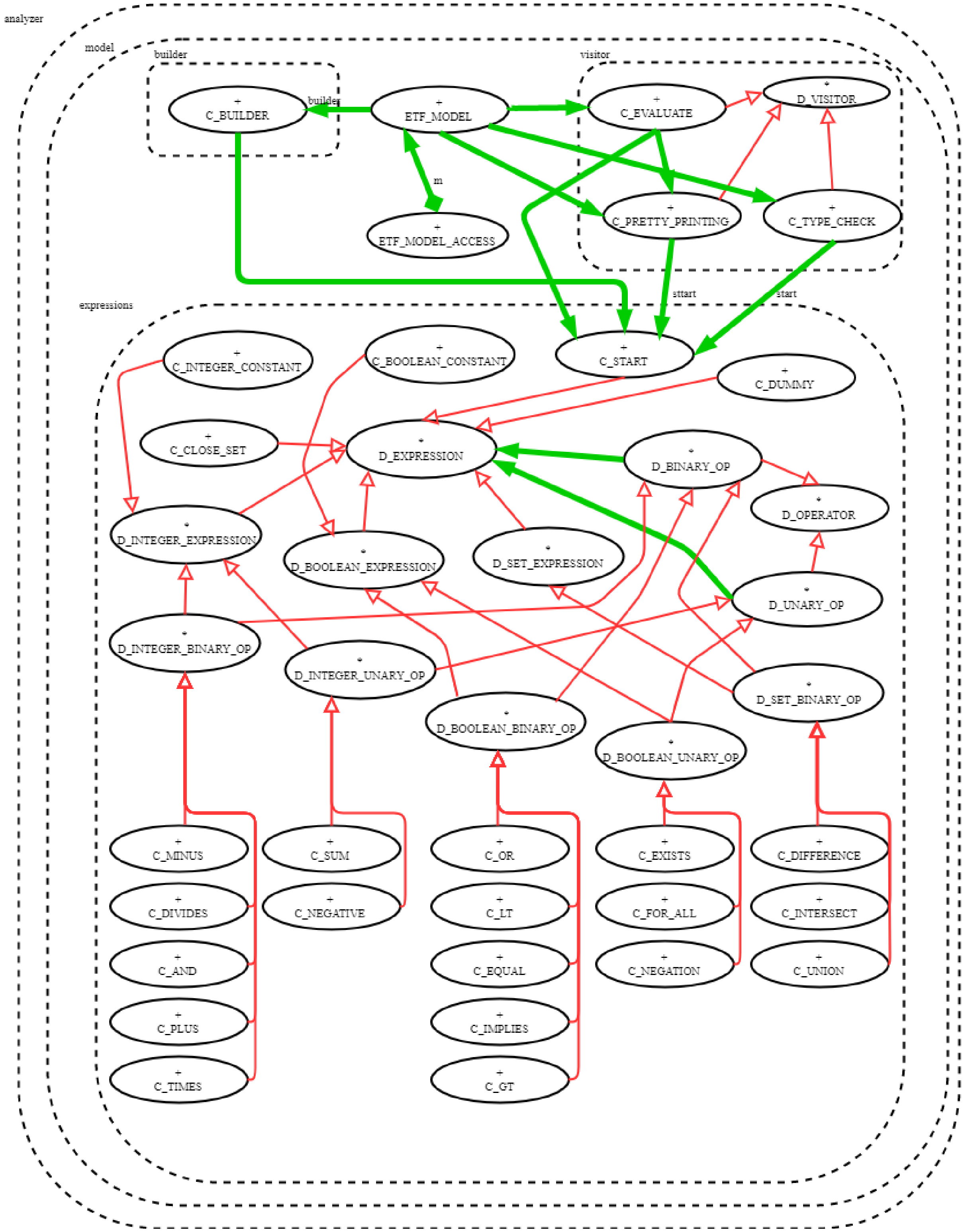
Project: Report

EECS3311,
Fall 2017.

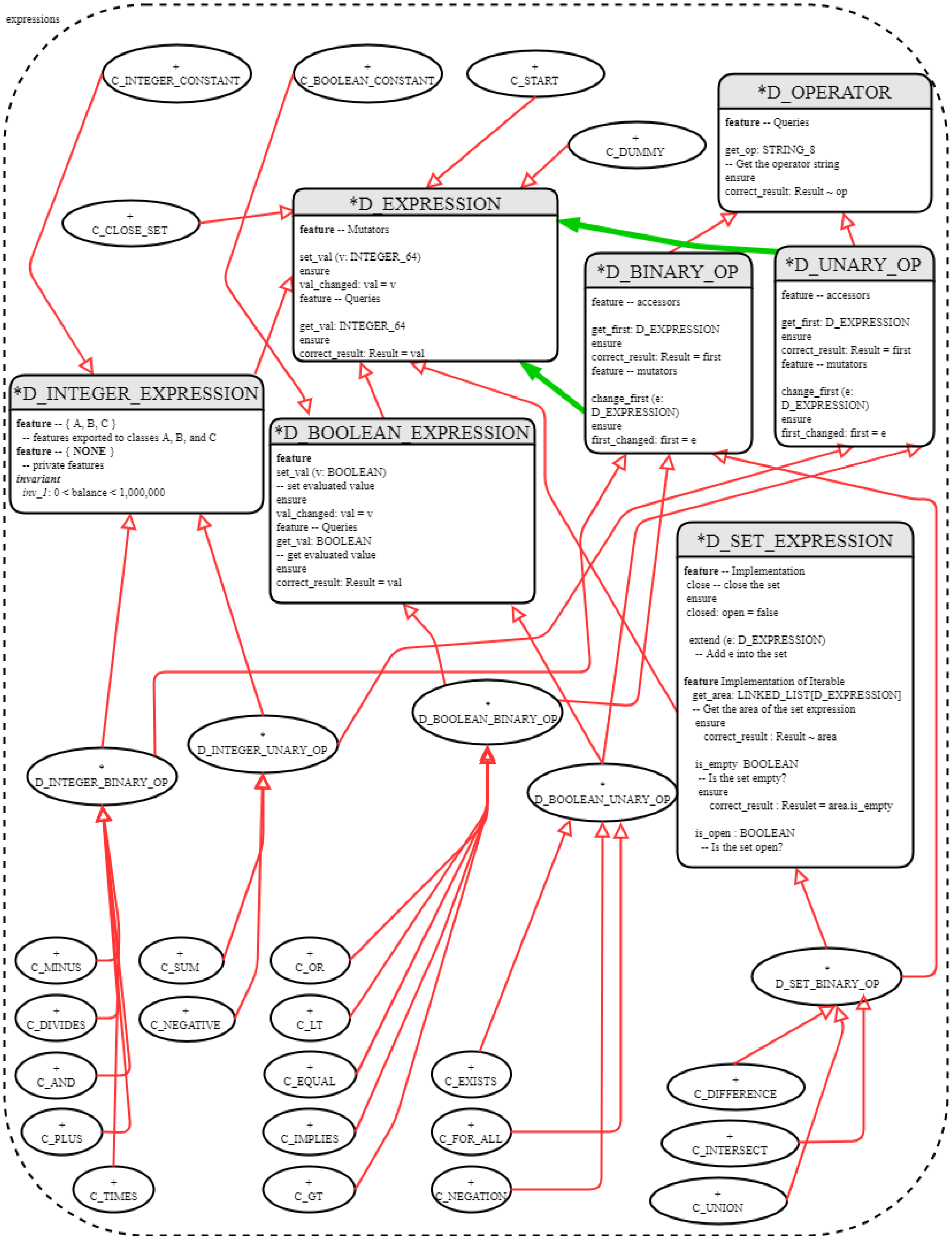
Ju-Young Kim - jyk3216,
Michael Mierzwa - mierzwam,
Dusan Putnikovic - cse03244,
Submitting Account: mierzwam

Section #1: Architectural Diagrams

BON #1: Concise Class Relationship



BON #2: Expanded Expression Classes



*D_OPERATOR

```
feature -- Queries

get_op: STRING_8
-- Get the operator string
ensure
correct_result: Result ~ op
```

*D_EXPRESSION

```
feature -- Mutators

set_val (v: INTEGER_64)
ensure
val_changed: val = v
feature -- Queries

get_val: INTEGER_64
ensure
correct_result: Result = val
```

*D_BINARY_OP

```
feature -- accessors

get_first: D_EXPRESSION
ensure
correct_result: Result = first
feature -- mutators

change_first (e:
D_EXPRESSION)
ensure
first_changed: first = e
```

*D_UNARY_OP

```
feature -- accessors

get_first: D_EXPRESSION
ensure
correct_result: Result = first
feature -- mutators

change_first (e:
D_EXPRESSION)
ensure
first_changed: first = e
```

*D_INTEGER_EXPRESSION

```
feature -- { A, B, C }
  -- features exported to classes A, B, and C
feature -- { NONE }
  -- private features
invariant
  inv_1: 0 < balance < 1,000,000
```

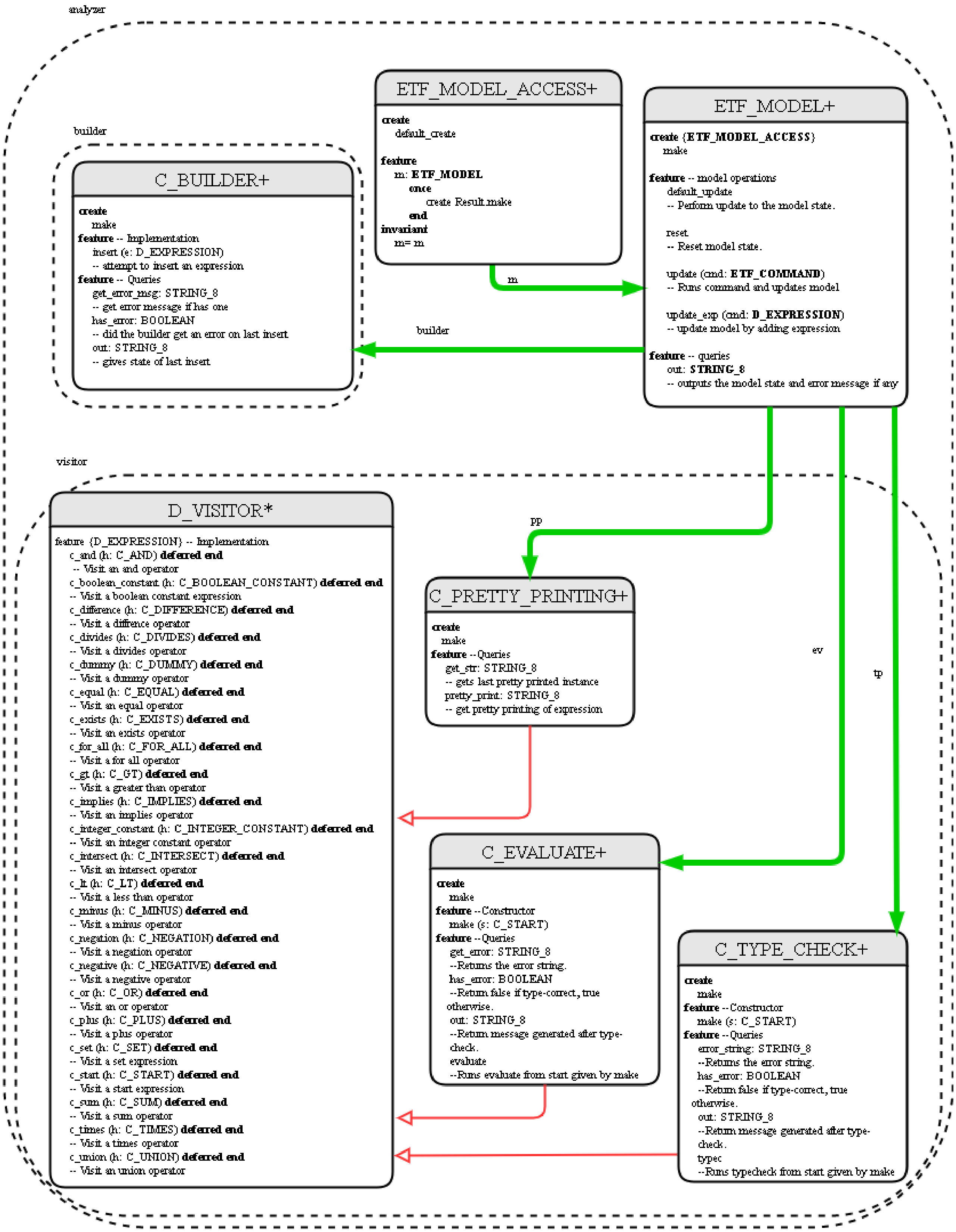
*D_BOOLEAN_EXPRESSION

```
feature
set_val (v: BOOLEAN)
-- set evaluated value
ensure
val_changed: val = v
feature -- Queries
get_val: BOOLEAN
-- get evaluated value
ensure
correct_result: Result = val
```

*D_SET_EXPRESSION

```
feature -- Implementation
 close -- close the set
 ensure
 closed: open = false

  extend (e: D_EXPRESSION)
   -- Add e into the set

feature Implementation of Iterable
  get_area: LINKED_LIST[D_EXPRESSION]
  -- Get the area of the set expression
   ensure
      correct_result : Result ~ area

   is_empty  BOOLEAN
   -- Is the set empty?
   ensure
      correct_result : Resulet = area.is_empty

   is_open : BOOLEAN
    -- Is the set open?
```

BON #3: Expanded Language Classes



```
ETF_MODEL_ACCESS+
create
   default_create
feature
   m: ETF_MODEL
      once
         create Result.make
      end
invariant
   m= m
```

```
ETF_MODEL+
create {ETF_MODEL_ACCESS}
   make
feature -- model operations
   default_update
   -- Perform update to the model state.

   reset
   -- Reset model state.

   update (cmd: ETF_COMMAND)
   -- Runs command and updates model

   update_exp (cmd: D_EXPRESSION)
   -- update model by adding expression

feature -- queries
   out: STRING_8
   -- outputs the model state and error message if any
```

```
C_BUILDER+
create
   make
feature -- Implementation
   insert (e: D_EXPRESSION)
   -- attempt to insert an expression
feature -- Queries
   get_error_msg: STRING_8
   -- get error message if has one
   has_error: BOOLEAN
   -- did the builder get an error on last insert
   out: STRING_8
   -- gives state of last insert
```

```
D_VISITOR*
feature {D_EXPRESSION} -- Implementation
   c_and (h: C_AND) deferred end
    -- Visit an and operator
   c_boolean_constant (h: C_BOOLEAN_CONSTANT) deferred end
   -- Visit a boolean constant expression
   c_difference (h: C_DIFFERENCE) deferred end
   -- Visit a diffrence operator
   c_divides (h: C_DIVIDES) deferred end
   -- Visit a divides operator
   c_dummy (h: C_DUMMY) deferred end
   -- Visit a dummy operator
   c_equal (h: C_EQUAL) deferred end
   -- Visit an equal operator
   c_exists (h: C_EXISTS) deferred end
   -- Visit an exists operator
   c_for_all (h: C_FOR_ALL) deferred end
   -- Visit a for all operator
   c_gt (h: C_GT) deferred end
   -- Visit a greater than operator
   c_implies (h: C_IMPLIES) deferred end
   -- Visit an implies operator
   c_integer_constant (h: C_INTEGER_CONSTANT) deferred end
   -- Visit an integer constant operator
   c_intersect (h: C_INTERSECT) deferred end
   -- Visit an intersect operator
   c_lt (h: C_LT) deferred end
   -- Visit a less than operator
   c_minus (h: C_MINUS) deferred end
   -- Visit a minus operator
   c_negation (h: C_NEGATION) deferred end
   -- Visit a negation operator
   c_negative (h: C_NEGATIVE) deferred end
   -- Visit a negative operator
   c_or (h: C_OR) deferred end
   -- Visit an or operator
   c_plus (h: C_PLUS) deferred end
   -- Visit a plus operator
   c_set (h: C_SET) deferred end
   -- Visit a set expression
   c_start (h: C_START) deferred end
   -- Visit a start expression
   c_sum (h: C_SUM) deferred end
   -- Visit a sum operator
   c_times (h: C_TIMES) deferred end
   -- Visit a times operator
   c_union (h: C_UNION) deferred end
   -- Visit an union operator
```

```
C_PRETTY_PRINTING+
create
   make
feature --Queries
   get_str: STRING_8
   -- gets last pretty printed instance
   pretty_print: STRING_8
   -- get pretty printing of expression
```

```
C_EVALUATE+
create
   make
feature --Constructor
   make (s: C_START)
feature --Queries
   get_error: STRING_8
   --Returns the error string.
   has_error: BOOLEAN
   --Return false if type-correct, true
  otherwise.
   out: STRING_8
   --Return message generated after type-
   check.
   evaluate
   --Runs evaluate from start given by make
```

```
C_TYPE_CHECK+
create
   make
feature --Constructor
   make (s: C_START)
feature --Queries
   error_string: STRING_8
   --Returns the error string.
   has_error: BOOLEAN
   --Return false if type-correct, true
  otherwise.
   out: STRING_8
   --Return message generated after type-
   check.
   typec
   --Runs typecheck from start given by make
```

Section #2: Information Hiding

The expressions and operators all have their attributes hidden from the user and can only be accessed and mutated by queries and mutators. Some features are completely hidden from the client because they are completely irrelevant to the client. All operators have an operator string called op that is the string that represents the operation. Not all operators are unique for example sum has the same operator string as plus but differ because one is an unary operator and other a binary operator. However the client only needs the operator and that can be accessed through get_op feature from operator. Boolean and integer expressions have an attribute called *val* that is hidden from the client. This attribute is used to make evaluating easier as we can store the evaluated value in there. Constants have their values set upon creation therefore any constant is already evaluated by C_EVALUATE.

The C_BUILDER class has both attributes and features hidden. The error messages are completely hidden from the user and have to be accessed by get_error_msg. This feature however only gives the error message of the error that occurred when an error occurred in the last insert. Another reason why all the error messages are hidden is because our model does not need to do any error handling. Therefore the model has no use to try and check which error it occurred and handle it differently. The attribute *inserted* is also hidden to the client. This attribute is used to stop traversing the expressions upon inserting, and helps determine what error to put in. Each individual feature is hidden to all clients except for the specific classes that use them, therefore other classes are unable to use them.

The C_PRETTY_PRINTING class has all its attributes hidden and the only accessible is *outstr* and only though get_message which only returns a copy of the string instead of the string directly. The attribute *qmark* in C_PRETTY_PRINTING is used to choose between printing a "?" or a "nil" for the not properly specified areas. This attribute is not needed to be known about or seen by the user and therefore is completely inaccessible and immutable. The start is passed through upon constructing a pretty print but is inaccessible and unchangeable upon being set into the pretty print.

The C_TYPE_CHECK class has its C_START node, its error booleans, and its error messages, all as hidden attributes. Since each of these attributes are key to the execution of C_TYPE_CHECK, they are hidden. The C_START is the starting point passed over from ETF_MODEL. The client is the one passing over this attribute, but it is crucial that this isn't changed before C_TYPE_CHECK evaluates, otherwise the result could be incorrect. The error messages do not need to be known by the client at all, all that needs to be known by the client is the current error message generated. Via information hiding, C_TYPE_CHECK can have as many error messages as it likes, but all the client sees, is a copy of the error string produced from running type_check. Finally, the hidden booleans *not_full* and *not_tc* are crucial for type_check returning the correct error, and as such, they are not directly accessible. The has_error query ensures that the client does not have direct access and only knows when there is an error. The error_string feature creates a string given the current error to ensure the boolean is not modified.

In our analyzer, the class C_EVALUATE follows the principle of information hiding. In order to function properly on input supplied to the class C_EVALUATE by C_TYPE_CHECK and to pass on the result to C_PRETTY_PRINTING without disclosing any extraneous information to clients and risking mutability, we make use of exporting to none and exporting to specific classes. Our constructor feature make and class attributes that handle the casting errors, divide by zero error, and entry into the expression, are all private features with export to none specified. The get_error, has_error and out which passes pretty printing into the ETF_MODEL are all hidden from the client with an export specific to ETF_MODEL only. These are crucial to the proper functioning of C_EVALUATE class and remain hidden from the client as do the private features and attributes of C_PRETTY_PRINTING and C_TYPE_CHECK.In fact all of the information of C_EVALUATE is hidden from the client as it is not required to be exported anywhere except to the ETF_MODEL.

Section #3: Single Choice Principle

The classes in analyzer properly conform to the Single Choice Principle because of one core reason, the fact that our expression language is fixed. This means that we have created all the expressions we need, and ever will need. Given this, we can justify the fact that our C_PRETTY_PRINT, C_TYPE_CHECK, and C_EVALUATE, all have unique implementations of the D_VISITOR's features. Single Choice Principle would be violated if there was a new expression in D_VISITOR that would then need to be implemented in all of the descendant classes C_PRETTY_PRINT, C_TYPE_CHECK, and C_EVALUATE. Since our language is fixed however, we do not need to worry about the modification of D_VISITOR. The D_VISITOR itself also does not violate the principle. The reason being is that each of the expressions is separated into classes themselves, meaning that a new implementation of D_VISITOR can use all of the code defined in each expressions' class.

Section #4: Open-Closed Principle

The expressions in the project have all attributes hidden and can only be accessed and changed through the given features. One example of this is C_SET which inherits

D_SET_EXPRESSION. D_SET_EXPRESSION has area (LINKED_LIST[D_EXPRESSION]) and open (BOOLEAN) attributes. The area attribute can be extended through the feature extend, checked if it is empty by the feature is_empty, and obtain a copy of the area through get_area. The feature get_area is a shallow copy of the area because we want the client to be able to manipulate contents of the elements in the array. Without the client being given access to move, delete, add, or replace any elements in the array. The attribute open can have its value accessed through is_open and can only be set to false by calling close but is impossible to open. The two attributes interact by extend because the precondition of extend is is_open which makes the area unchangeable, thus satisfying the open closed principle.

The open-closed principle is enforced in our builder and visitor patterns. The class C_BUILDER has all of its attributes directly inaccessable and some only accessible under certain conditions. The start attribute contains the start expression for the heap which is passed upon constructing an C_BUILDER instance. This start attribute value that gets stored during construction is not accessible from the builder. Any building has to be done through the insert feature. Error messages are also hidden and getting an error message has a prerequisite of error being true. Any time the client wants the state of the insert is though has_error and out.

The C_PRETTY_PRINTING uses the open-closed principle for its attributes also. C_PRETTY_PRINTING has three attributes: outstr, start, and qmark. The attribute start is the same as the builder, defined once upon construction and never able to be accessed by the client from the pretty printer. The qmark is completely hidden from the client because it is used in the pretty print to determine whether to print a "?" or "nil" and therefore the client does not need to know about it. The outstr is the output string that is created upon calling pretty_print feature. To save time for doing type_check C_PRETTY_PRINTING also has a feature get_str which gives the last instance of pretty printing. Therefore the outstr is never directly accessed in any way.

C_TYPE_CHECK also follows the open-closed principle due to the fact that all of its variables are hidden from the client behind accessor queries. The error_string and has_error ensure that all variables relating to error are closed off from the client, but the relevant information about them is available through the queries. The out query is also used to hide the possible output messages, and returns a copy of the actual messages stored in C_TYPE_CHECK. This ensures that no client can tamper with it and that C_TYPE_CHECK is closed.

The C_EVALUATE class applies the open_closed principle, because its constructor and all attributes are hidden from the client. Furthermore all the calculation features are exported only to their respective classes.The class is closed to modification and open to extension, although extension is not very much applicable in the sense that we have established virtually all the expression calculations required.

Section #5: Uniform Access Principle

Our entire project follows and maintains the uniform access principle by hiding and setting mutators and accessors to its attributes. In our operators, the only way for the client to get the operator string is through calling get_op. This follows the uniform access principle because there is only one way to access the attribute op. This is also implemented in our binary and unary operators by having the children accessible and mutable only by using get_first and change_first for the first child and get_second and change_second for the second child. These attributes are unable to be accessed by any other way and are accessed the same way in all the features in the class

The C_BUILDER class maintains the uniform access principle in its features also. When checking for if there was an error found while building it always goes through the feature has_error instead of checking if error.is_empty. Therefore any time any feature in builder wants to get if there was an error it goes through the has_error. This applies to all of the attributes since all attributes are accessed and changed through one way for each attribute.

The uniform access principle is also maintained in the C_PRETTY_PRINTING class. The attribute qmark is completely hidden from the user while start is never accessible. Although the attribute outstr seems to disobey the uniform access principle it does not for the following reasons: get_str gets a copy of outstr and cannot be used to mutate outstr. The C_PRETTY_PRINTING class requires changing of outstr as it is building the string while traversing the heap. Since get_str gives a copy of the string it is not possible to use that feature to build on the string. The client however does not have any way to directly access outstr and therefore the uniform access principle is maintained.

The C_TYPE_CHECK class follows the uniform access principle via queries to access the needed information from C_TYPE_CHECK. Any information needed about the error that C_TYPE_CHECK produces can be found in the two queries, error_string and has_error. Each of these do not return the value itself, but returns a copy of it, or in the case of has_error, a logical conclusion from the error. The out feature in C_TYPE_CHECK follows the same strategy as error_string, and returns a copy of the error string to ensure that the client cannot modify the error string directly. Any modification of the values in type_check is caused via typec, which runs C_TYPE_CHECK on the current C_START node given.

Class C_EVALUATE also maintains the uniform access principle as it has no actual mutators in the class. The class queries are get_error, has_error and out which simply report the existence of an error, what that error is, which is actually in its entirety the one and only divide_by_zero error. The out feature connects to C_PRETTY_PRITNING and actually calls on its pretty_print function. Throughout the project, C_TYPE_CHECK, C_EVALATE and C_PRETTY_PRINTING are solidly abiding classes of the Universal Access Principle.

END