EECS 3311 Project Report

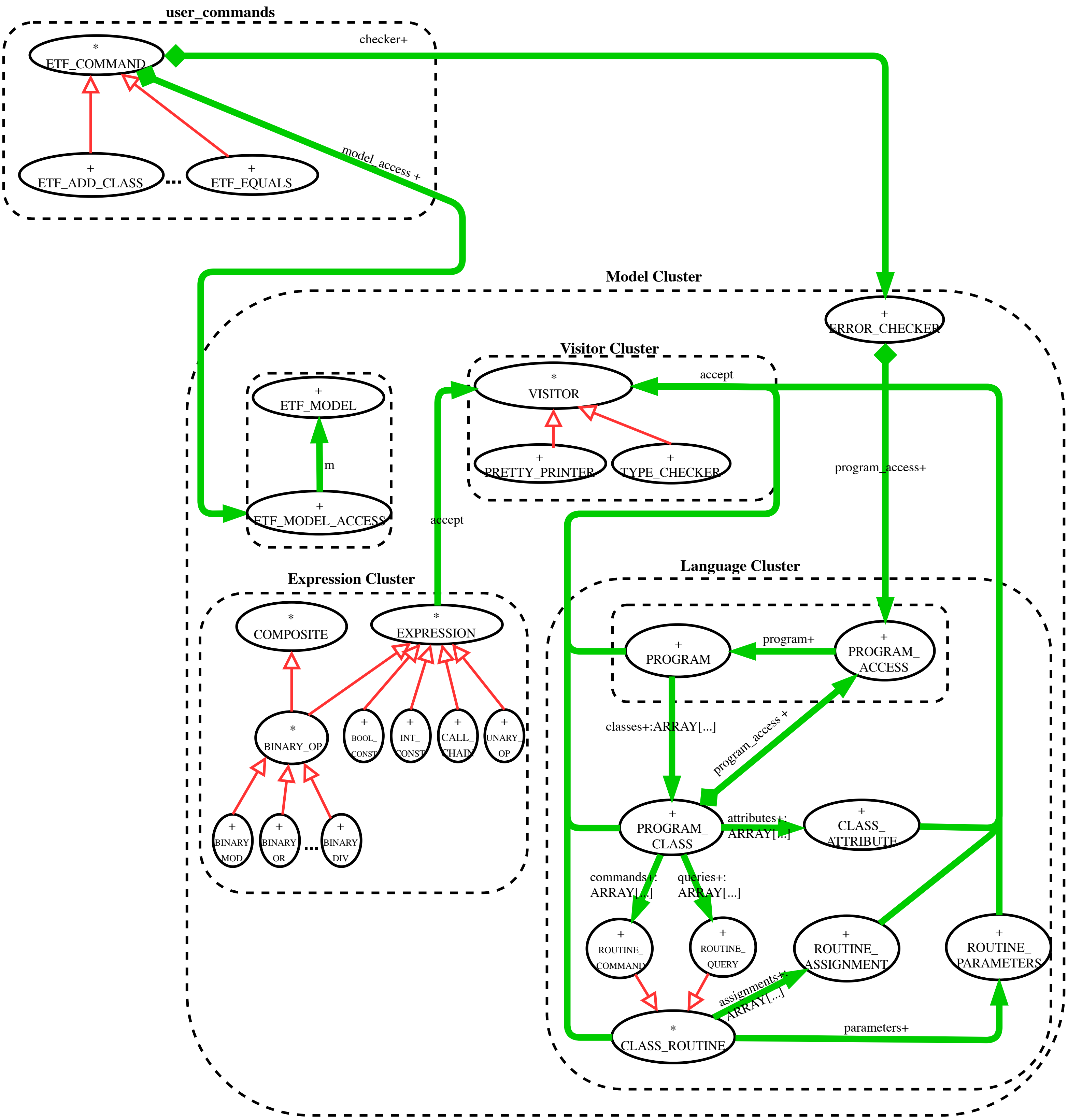
Fall 2019

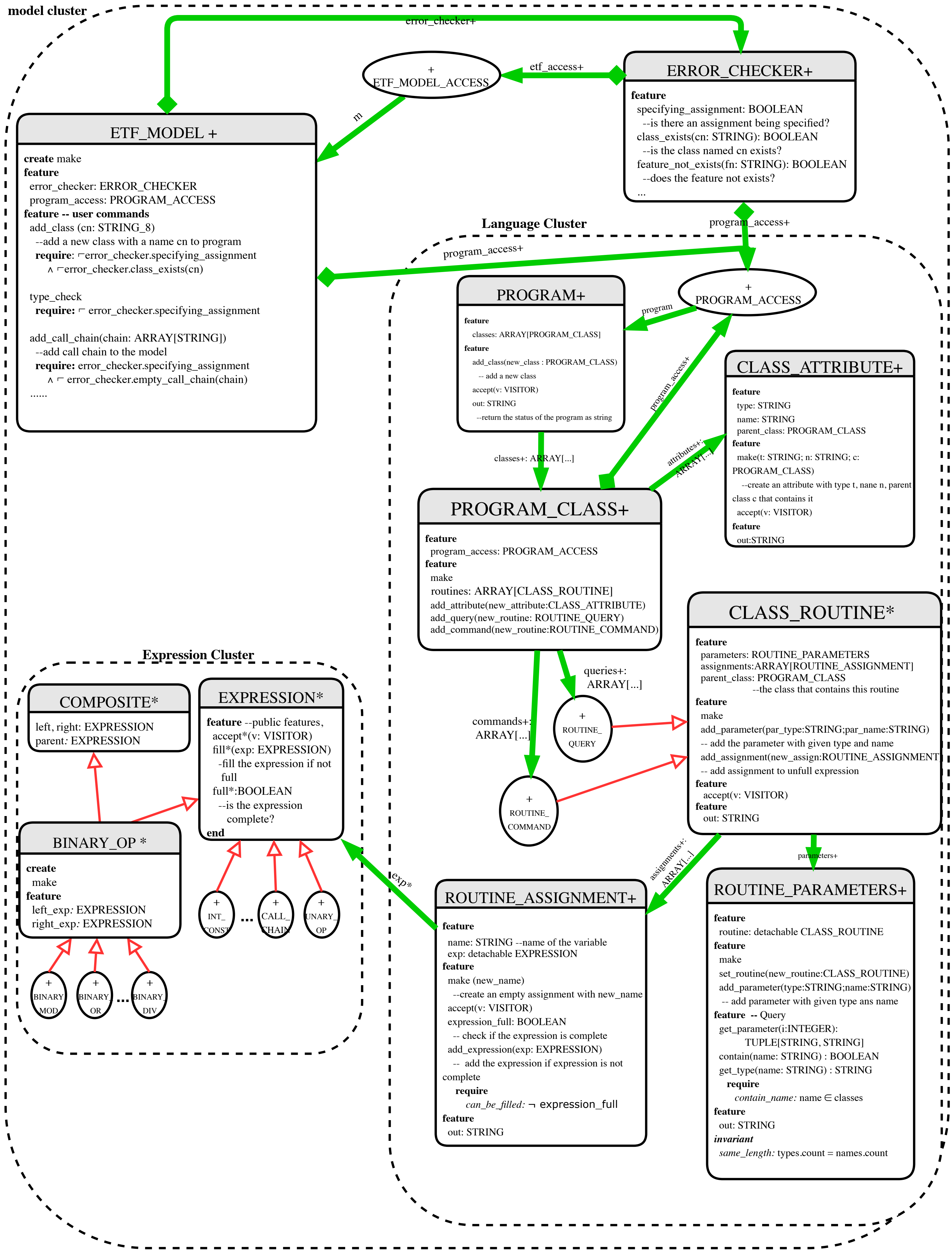
Section A

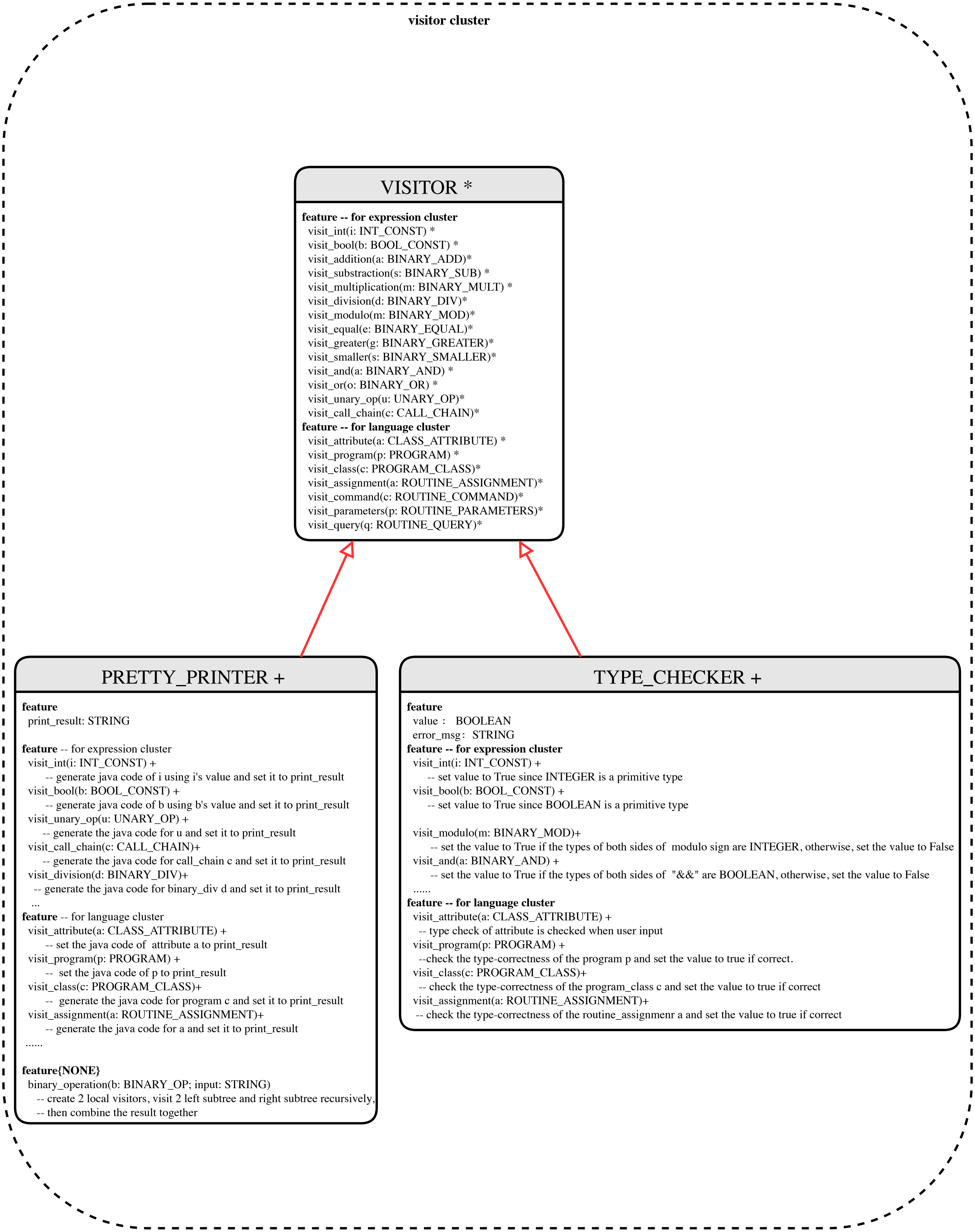
Cse login name: namesdu

helenyu

Cse submitting login: namesdu


user_commands
checker+
* ETF_COMMAND
+ ETF_ADD_CLASS
...
+ ETF_EQUALS
model_access +
Model Cluster
+ ERROR_CHECKER
Visitor Cluster
accept
* VISITOR
+ PRETTY_PRINTER
+ TYPE_CHECKER
program_access+
+ ETF_MODEL
m
+ ETF_MODEL_ACCESS
accept
Expression Cluster
* COMPOSITE
* EXPRESSION
* BINARY_OP
+ BOOL_ CONST
+ INT_ CONST
+ CALL_ CHAIN
+ UNARY_ OP
+ BINARY MOD
+ BINARY OR
...
+ BINARY DIV
Language Cluster
+ PROGRAM
program+
+ PROGRAM_ ACCESS
classes+:ARRAY[...]
program_access +
+ PROGRAM_ CLASS
attributes+: ARRAY[...]
+ CLASS_ ATTRIBUTE
commands+: ARRAY[...]
queries+: ARRAY[...]
+ ROUTINE_ COMMAND
+ ROUTINE_ QUERY
+ ROUTINE_ ASSIGNMENT
+ ROUTINE_ PARAMETERS
assignments+: ARRAY[...]
* CLASS_ROUTINE
parameters+

model cluster
error_checker+
+
ETF_MODEL_ACCESS
etf_access+
ERROR_CHECKER+
feature
specifying_assignment: BOOLEAN
--is there an assignment being specified?
class_exists(cn: STRING): BOOLEAN
--is the class named cn exists?
feature_not_exists(fn: STRING): BOOLEAN
--does the feature not exists?
...
m
ETF_MODEL +
create make
feature
error_checker: ERROR_CHECKER
program_access: PROGRAM_ACCESS
feature -- user commands
add_class (cn: STRING_8)
--add a new class with a name cn to program
require: ⌐error_checker.specifying_assignment
∧ ⌐error_checker.class_exists(cn)
type_check
require: ⌐ error_checker.specifying_assignment
add_call_chain(chain: ARRAY[STRING])
--add call chain to the model
require: error_checker.specifying_assignment
∧ ⌐ error_checker.empty_call_chain(chain)
......
program_access+
Language Cluster
program_access+
+
PROGRAM_ACCESS
program
PROGRAM+
feature
classes: ARRAY[PROGRAM_CLASS]
feature
add_class(new_class : PROGRAM_CLASS)
-- add a new class
accept(v: VISITOR)
out: STRING
--return the status of the program as string
program_access+
CLASS_ATTRIBUTE+
feature
type: STRING
name: STRING
parent_class: PROGRAM_CLASS
feature
make(t: STRING; n: STRING; c: PROGRAM_CLASS)
--create an attribute with type t, nane n, parent class c that contains it
accept(v: VISITOR)
feature
out:STRING
classes+: ARRAY[...]
attributes+: ARRAY[...]
PROGRAM_CLASS+
feature
program_access: PROGRAM_ACCESS
feature
make
routines: ARRAY[CLASS_ROUTINE]
add_attribute(new_attribute:CLASS_ATTRIBUTE)
add_query(new_routine: ROUTINE_QUERY)
add_command(new_routine:ROUTINE_COMMAND)
CLASS_ROUTINE*
feature
parameters: ROUTINE_PARAMETERS
assignments:ARRAY[ROUTINE_ASSIGNMENT]
parent_class: PROGRAM_CLASS
--the class that contains this routine
feature
make
add_parameter(par_type:STRING;par_name:STRING)
-- add the parameter with given type and name
add_assignment(new_assign:ROUTINE_ASSIGNMENT)
-- add assignment to unfull expression
feature
accept(v: VISITOR)
feature
out: STRING
queries+:
ARRAY[...]
commands+:
ARRAY[...]
+
ROUTINE_
QUERY
+
ROUTINE_
COMMAND
Expression Cluster
COMPOSITE*
left, right: EXPRESSION
parent: EXPRESSION
EXPRESSION*
feature --public features,
accept*(v: VISITOR)
fill*(exp: EXPRESSION)
-fill the expression if not full
full*:BOOLEAN
--is the expression complete?
end
BINARY_OP *
create
make
feature
left_exp: EXPRESSION
right_exp: EXPRESSION
+
INT_
CONST
...
+
CALL_
CHAIN
+
UNARY_
OP
+
BINARY_
MOD
+
BINARY_
OR
...
+
BINARY_
DIV
exp*
assignments+:
ARRAY[...]
parameters+
ROUTINE_ASSIGNMENT+
feature
name: STRING --name of the variable
exp: detachable EXPRESSION
feature
make (new_name)
--create an empty assignment with new_name
accept(v: VISITOR)
expression_full: BOOLEAN
-- check if the expression is complete
add_expression(exp: EXPRESSION)
-- add the expression if expression is not complete
require
can_be_filled: ¬ expression_full
feature
out: STRING
ROUTINE_PARAMETERS+
feature
routine: detachable CLASS_ROUTINE
feature
make
set_routine(new_routine:CLASS_ROUTINE)
add_parameter(type:STRING;name:STRING)
-- add parameter with given type ans name
feature -- Query
get_parameter(i:INTEGER):
TUPLE[STRING, STRING]
contain(name: STRING) : BOOLEAN
get_type(name: STRING) : STRING
require
contain_name: name ∈ classes
feature
out: STRING
invariant
same_length: types.count = names.count



visitor cluster

VISITOR *

feature -- for expression cluster
visit_int(i: INT_CONST) *
visit_bool(b: BOOL_CONST) *
visit_addition(a: BINARY_ADD)*
visit_substraction(s: BINARY_SUB) *
visit_multiplication(m: BINARY_MULT) *
visit_division(d: BINARY_DIV)*
visit_modulo(m: BINARY_MOD)*
visit_equal(e: BINARY_EQUAL)*
visit_greater(g: BINARY_GREATER)*
visit_smaller(s: BINARY_SMALLER)*
visit_and(a: BINARY_AND) *
visit_or(o: BINARY_OR) *
visit_unary_op(u: UNARY_OP)*
visit_call_chain(c: CALL_CHAIN)*
feature -- for language cluster
visit_attribute(a: CLASS_ATTRIBUTE) *
visit_program(p: PROGRAM) *
visit_class(c: PROGRAM_CLASS)*
visit_assignment(a: ROUTINE_ASSIGNMENT)*
visit_command(c: ROUTINE_COMMAND)*
visit_parameters(p: ROUTINE_PARAMETERS)*
visit_query(q: ROUTINE_QUERY)*

PRETTY_PRINTER +

feature
print_result: STRING

feature -- for expression cluster
visit_int(i: INT_CONST) +
-- generate java code of i using i's value and set it to print_result
visit_bool(b: BOOL_CONST) +
-- generate java code of b using b's value and set it to print_result
visit_unary_op(u: UNARY_OP) +
-- generate the java code for u and set it to print_result
visit_call_chain(c: CALL_CHAIN)+
-- generate the java code for call_chain c and set it to print_result
visit_division(d: BINARY_DIV)+
-- generate the java code for binary_div d and set it to print_result
...
feature -- for language cluster
visit_attribute(a: CLASS_ATTRIBUTE) +
-- set the java code of attribute a to print_result
visit_program(p: PROGRAM) +
-- set the java code of p to print_result
visit_class(c: PROGRAM_CLASS)+
-- generate the java code for program c and set it to print_result
visit_assignment(a: ROUTINE_ASSIGNMENT)+
-- generate the java code for a and set it to print_result
......

feature{NONE}
binary_operation(b: BINARY_OP; input: STRING)
-- create 2 local visitors, visit 2 left subtree and right subtree recursively,
-- then combine the result together

TYPE_CHECKER +

feature
value : BOOLEAN
error_msg : STRING
feature -- for expression cluster
visit_int(i: INT_CONST) +
-- set value to True since INTEGER is a primitive type
visit_bool(b: BOOL_CONST) +
-- set value to True since BOOLEAN is a primitive type

visit_modulo(m: BINARY_MOD)+
-- set the value to True if the types of both sides of modulo sign are INTEGER, otherwise, set the value to False
visit_and(a: BINARY_AND) +
-- set the value to True if the types of both sides of "&&" are BOOLEAN, otherwise, set the value to False
......
feature -- for language cluster
visit_attribute(a: CLASS_ATTRIBUTE) +
-- type check of attribute is checked when user input
visit_program(p: PROGRAM) +
--check the type-correctness of the program p and set the value to true if correct.
visit_class(c: PROGRAM_CLASS)+
-- check the type-correctness of the program_class c and set the value to true if correct
visit_assignment(a: ROUTINE_ASSIGNMENT)+
-- check the type-correctness of the routine_assignmenr a and set the value to true if correct

Some design ideas about this project.

The project is designed by visitor pattern and it follows the following principles, the design of ETF_MODEL_ACCESS and PROGRAM_ACCESS uses singlinton pattern. ERROR_CHECKER ensures the coheision of the program.

It obeyed the following principles

Open-closed principle

The structure of the language should be closed, the operation part should be opened. Hence we used visitor pattern for developing this programming language. In the future, if we want to add some new operations, the operation part is the only place to change. If we leave the structure opened, and change the structure, the place we need to modify are both structure part and operation part, this would violate the single-choice-principle. So we need to fully design the structure and then keep it closed. For example, PRETTY_PRINTER and TYPE_CHECKER both inherits from VISITOR class. If in the future we need to add an addition operation to it, we can just add another descendant class of VISITOR called ADDITION in the operation part, this is th only place we need to change.

Single-choice principle

During the development of the expression part, to avoid code duplication, we used the composite pattern. BINARY_OP inherits both COMPOSITE class and EXPRESSION class. The ROUTINE_COMMAND and ROUTINE_QUERY also inherit the CLASS_ROUTINE for the same reason, so does the VISITOR. Their descendant class can inherit the deferred features and implement the features into different versions for different use. If there's a future change, simply change the currosponding implementation in currospongding class.

The helper methods are also designed for this purpose. Like the helper method generating_string feature in ERROR_CHECKER, it generates the string with given array, since we need to get the "list" of wrong types in different method, this helper method really helps to avoid the code duplications.

Information-hiding

For example, the routines feature in PROGRAM_CLASS is a representation of the information-hiding . The feature is closed to clients, they don't need to know how it works, they just use it. If there are some future changes need to apply to this feature, the only place we need to modify is the content of the feature.

Also, there is no need for clients to know the whole design of the program, so we keep some features private since they are for development purpose and hence they opened for changes, like the helper method in ERROR_CHECKER class. To avoid some evil client, we also make some important features private, like the make feature in PROGRAM class, we declared it to {NONE}

Uniform Access Principle

In the ROUTINE_PARAMETER class, we declared 2 private array, which are types and names. One of the clients of ROUTINE_PARAMETER, CLASS_ROUTINE, uses add_parameter feature to add new parameter. As a client, CLASS_ROUTINE just need to call the feature and it doesn't know how the parameter is added. If the supplier secretly change the collections of names and types into LINKED_LIST[STRING] and change the original implementation of :

```
add_parameter (type:STRING; name:STRING)
    do
        types.force (type, types.count+1)
        names.force (name, names.count+1)
    end
```

to the following:

```
add_parameter (type:STRING; name:STRING)
    do
        types.extend(type)
        names.extend (name)
    end
```

the client can still call the feature with no concerns. This is, somehow, also belongs to information hiding.