ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΛΟΠΟΝΝΗΣΟΥ
ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ ΚΑΙ ΤΕΧΝΟΛΟΓΙΑΣ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΝΩΝ

Πρόγραμμα Μεταπτυχιακών Σπουδών (Π.Μ.Σ.) στην Επιστήμη και την Τεχνολογία Υπολογιστών

Μεταπτυχιακή Εργασία

Εφαρμογή Σχεδιαστικών Προτύπων (Design Patterns) Αντικειμενοστραφούς Προγραμματισμού σε Μεταγλωττιστές

Ονοματεπώνυμο Φοιτητή	**Χρήστος Α. Καρανικόλας**
Αριθμός Μητρώου	2012006
Επιβλέπων Καθηγητής	**Γρηγόρης Δημητρουλάκος**
Πεδίο	Επιστήμη & Τεχνολογία Υπολογιστών
Θεματική κατεύθυνση	Τεχνολογίες Αιχμής & Ερευνητικά Θέματα Υπολογιστών

Τρίπολη, **Μάιος 2014**

Η έγκριση της μεταπτυχιακής διατριβής από το Τμήμα Πληροφορικής και Τηλεπικοινωνιών του Πανεπιστημίου Πελοποννήσου δεν υποδηλώνει απαραιτήτως και αποδοχή των απόψεων του συγγραφέα εκ μέρους του Τμήματος.

Πρόλογος

Η επιλογή του θέματος της συγκριμένης εργασίας προέκυψε μέσα από πραγματικές ανάγκες και δυσκολίες που αντιμετωπίσθηκαν κατά την ενασχόληση του συγγραφέα με την σχεδίαση και ανάπτυξη του τμήματος παραγωγής ενδιάμεσου κώδικα για μεταγλωττιστές, σε υπάρχουσες υλοποιήσεις. Οι κατευθυντήριες γραμμές, τεχνικές οδηγίες, και η σχετική βιβλιογραφία υποδείχθηκαν από τον επιβλέποντα κ. Δημητρουλάκο Γρήγορη, τον οποίο και ευχαριστώ ιδιαιτέρως για τον ουσιαστικό και καθοριστικό ρόλο του στην υλοποίηση της παρούσας μεταπτυχιακής έρευνας. Επίσης σημαντικό ρόλο διαδραμάτισαν οι οδηγίες και υποδείξεις των μελών της τριμελούς επιτροπής τους οποίους επίσης ευχαριστώ για την συνεισφορά τους.

Για την υποβοήθηση της σχεδίασης και υλοποίησης της εργασίας, διατέθηκε από το Εργαστήριο Λογικής Σχεδίασης και Αρχιτεκτονικής του Τμήματος Πληροφορικής και Τηλεπικοινωνιών του Πανεπιστημίου Πελοποννήσου, ανοικτό λογισμικό μερικής υλοποίησης μεταγλωττιστή από το οποίο και απεκόμισα σημαντική εμπειρία αλλά και προβληματισμούς σχετικά με την σχεδίαση και ανάπτυξη μεταγλωττιστών. Επίσης στα πλαίσια του μεταπτυχιακού προγράμματος διατέθηκαν από τη σχολή οι απαραίτητες άδειες εφαρμογών για τη συγγραφή κώδικα και σχεδίαση διαγραμμάτων.

Η παρούσα εργασία αφιερώνεται στους γονείς μου, στη σύζυγο μου και στα παιδία μου, οι όποιοι με στήριξαν ηθικά και ψυχολογικά, προκειμένου να αφοσιωθώ στην έρευνα και την επιτυχή ολοκλήρωσή της .

Περίληψη

Η ανάπτυξη ενός μεταγλωττιστή είναι ένα δύσκολο πρόβλημα, η επιτυχής λύση του οποίου, καθορίζεται από την πληρότητα της ανάλυσης και της αρχιτεκτονικής σχεδίασης του υπό ανάπτυξη λογισμικού. Κατά τη διαδικασία της αρχιτεκτονικής σχεδίασης του λογισμικού, επιλέγονται κατάλληλα και δοκιμασμένα σχεδιαστικά πρότυπα, δηλαδή καλές σχεδιαστικές λύσεις οι οποίες έχουν αναπτυχθεί για την επίλυση γενικών συνήθως προβλημάτων και τείνουν να επαναχρησιμοποιούνται από τους Μηχανικούς Λογισμικού.

Η παρούσα εργασία αποτελεί μια τεχνική περιγραφή για τη σχεδίαση, προσαρμογή, υλοποίηση και εφαρμογή συγκεκριμένων αντικειμενοστραφών σχεδιαστικών προτύπων σε μεταγλωττιστές, ως μία κάθετη προσέγγιση που εκκινεί από την μοντελοποιημένη αρχιτεκτονική σχεδίαση μέχρι και την ενδεικτική υλοποίηση κώδικα εφαρμογής τους. Παρουσιάζεται ένα ενδεικτικό μοντέλο της αρχιτεκτονικής σχεδίασης τμήματος ενός μεταγλωττιστή της γλώσσας C89, μέσω UML διαγραμμάτων κλάσεων, αντικειμένων και ακολουθίας. Χρησιμοποιούνται τα γνωστά αντικειμενοστραφή σχεδιαστικά πρότυπα Façade, Composite, Iterator και Visitor καθώς και προσαρμοσμένες στις ανάγκες του προβλήματος παραλλαγές τους. Η σχεδίαση επικεντρώνεται στην χρήση των συγκεκριμένων σχεδιαστικών προτύπων στον πίνακα συμβόλων, στο parse / abstract tree, στην three-address code αναπαράσταση και στις ενδεικτικές λειτουργίες scope checking, type inference & checking και graph generation. Ωστόσο το ίδιο μοντέλο σχεδίασης μπορεί να εφαρμοσθεί κατά περίπτωση για όλες τις Syntax Directed Definitions του Attribute Grammar Framework ενός μεταγλωττιστή. Παρουσιάζονται τμήματα κώδικα υλοποίησης σε C++ και παραδείγματα εφαρμογής iterator και visitor για pre order και post order διαπεράσεις σύνθετων δομών (parse / abstract trees, graphics IRs ως composites). Επίσης παρουσιάζεται μια προσαρμοσμένη σχεδίαση και υλοποίηση iterator και visitor για την μικτή pre post order διαπέραση composite δομών για την εφαρμογή L-Attributed Definitions. Ακόμη διερευνούνται τα μειονεκτήματα και πλεονεκτήματα έναντι του προτύπου Inheritance, καθώς και οι συνθήκες για την επιλογή κατάλληλων σχεδιαστικών προτύπων και προτείνονται σχεδιαστικές προσεγγίσεις για την επίλυση των προβλημάτων που αντιμετωπίσθηκαν κατά την εφαρμογή τους.

Σκοπός της εργασίας δεν είναι η επί της ουσίας εξαντλητική ανάλυση, λύση και υλοποίηση των επιμέρους προβλημάτων και λειτουργιών (business logic) ενός μεταγλωττιστή, αλλά η εφαρμογή κατάλληλων σχεδιαστικών προτύπων που συνδράμουν θετικά στην υλοποίηση αυτών ή παρόμοιων λύσεων, ενισχύοντας παράλληλα τα ποιοτικά χαρακτηριστικά του παραγόμενου λογισμικού. Επίσης παρουσιάζεται μια ανάλυση των ποιοτικών χαρακτηριστικών των σχεδιαστικών προτύπων της υλοποίησης, καθώς και η συσχέτιση τους με μετρήσιμα ποιοτικά χαρακτηριστικά γνωστών προτύπων διασφάλισης ποιότητας λογισμικού. Τέλος η εργασία μπορεί να χρησιμοποιηθεί ως ένα κείμενο εναλλακτικής αναφοράς ή εκκίνησης για όσους επιθυμούν να ασχοληθούν με την κατανόηση και ανάπτυξη μεταγλωττιστών ή παρόμοιων προβλημάτων ή γενικά για την κατανόηση και εφαρμογή των γνωστών σχεδιαστικών προτύπων Façade, Composite, Iterator και Visitor.

Abstract

The development of a compiler is a hard task, whose successful solution is determined by the completeness of the analysis and the completeness of the architectural design of the software under development. During the process of the software architectural design, appropriate and tested design patterns, that is, satisfactory design solutions, which have been developed to resolve common problems and tend to be reused by the Software Engineers, are selected.

This paper is a technical description for the design, adaptation, implementation and application of specific object oriented design patterns on compilers. It is a fast approach which starts with the architectural design model and concludes with the indicative implementation of their application code. An indicative architectural design model, for a section of a compiler of the C89 programing language is being presented, through UML class, object, and sequence diagrams. The common object oriented Façade, Composite, Iterator, and Visitor design patterns, as well as, their adapted to the specific problem variations are being used. The design process focuses on the use of the specific design patterns on the symbol table, the parse / abstract tree, the representation of three address code and the indicative functions of Scope checking, Type inference & checking and graph generation. However, the same design model can be implemented, on occasion, for all the Syntax Directed Definitions of the Attributed Grammar Framework of a compiler. Sections of the implementation code in C++ programing language and examples of the application of Iterator and Visitor patterns for pre order and post order iterations of compound structures (parse/abstract trees, graphics IRs, as composites) are being presented. Moreover, an adapted design and implementation of Iterator and Visitor patterns for the joint pre post order iteration on composite structures for the application of L-Attributed Definitions is being presented. Furthermore, the disadvantages and the advantages in contrast to the Inheritance pattern, as well as, the conditions of the selection of the appropriate design patterns are being researched by this paper. Also, design approaches for the resolution of the problems which were dealt with during the application of the design patterns are being suggested.

The substantial exhaustive analysis, solution and implementation of the sub problems and functions (business logic) of a compiler, is not the main purpose of this paper. What it is though is the application of the appropriate design patterns, which positively contribute to the implementation of these or similar solutions, while enhancing the quality characteristics of the produced software. In addition, an analysis of the quality characteristics of the design patterns of the implementation, as well as, their correlation with countable quality characteristics of well-known standards of quality assurance software is being presented. Finally, this paper can be used as a text of an alternative reference or initiation for those wishing to be occupied with the comprehension and development of compilers or similar problems or generally the understanding and application of the well-known Façade, Composite, Iterator and Visitor design patterns.

Περιεχόμενα

Λίστα Εικόνων

Λίστα Πινάκων

Συντομογραφίες

AG	Attribute Grammar
ANSI	American National Standards Institute
AOP	Aspect Oriented Programming
AST	Abstract Syntax Tree
CFG	Context Free Grammar
CS	Computer Science
DFA	Determined Finite Automaton
FA	Finite Automaton
IR	Intermediate Representation
ISO	International Organization for Standardization
NFA	Non determined Finite Automaton
OMG	Object Management Group
OO	Object Oriented
OOP	Object Oriented Programming / Program
OOPL	Object Oriented Programming Language
OOS	Object Oriented Software
QME	Quality Measure Elements
RE	Regular Expression
RG	Regular Grammar
SE	Software Engineering / Engineer
SDD	Syntax Directed Definition
SQuaRE	Systems and Software Quality Requirements and Evaluation
SSA	Static Single-Assignment Form
UML	Unified Modeling Language

1 Εφαρμογή Σχεδιαστικών Προτύπων Αντικειμενοστραφούς Προγραμματισμού σε Μεταγλωττιστές

1.1 Προβληματισμός και ερευνητική προσέγγιση

Κατά την προσπάθεια δημιουργίας, βελτιστοποίησης και επέκτασης μεταγλωττιστών, και γενικά εργαλείων ανάλυσης τυποποιημένων γλωσσών, ανέκυψε δυσκολία στην κατανόηση τόσο της σχεδίασης όσο και του κώδικα των υλοποιήσεων. Διαπιστώθηκε ελλιπής τεκμηρίωση ή και απουσία μοντέλων σχεδίασης καθώς και κώδικας με δυσνόητες ενδιάμεσες αναπαραστάσεις δομών και διάσπαρτες λειτουργίες. Η προσπάθεια δε προσθήκης νέων λειτουργιών επιφέρει συνήθως μαζικές και χρονοβόρες αλλαγές σε πολλαπλά τμήματα του κώδικα. Η σχετική με τη θεωρία μεταγλωττιστών βιβλιογραφία επικεντρώνεται κυρίως σε θέματα επιμέρους αλγορίθμων και προσεγγίσεων για την επίλυση συγκεκριμένων υπό προβλημάτων, χωρίς ωστόσο να περιγράφονται αναλυτικές και συνεκτικές σχεδιαστικές προσεγγίσεις για την υλοποίηση τους, κυρίως λόγω της γενικότητας και της αφαιρετικής προσέγγισης που οφείλουν να χαρακτηρίζουν τα θεωρητικά κείμενα. Περεταίρω και η σχετική με τη μηχανική μεταγλωττιστών βιβλιογραφία επικεντρώνεται μεν σε θέματα επιμέρους αλγορίθμων, ενδιάμεσων δομών αναπαράστασης, ιδιοτήτων, τεχνικών σημασιολογικής ανάλυσης για την επίλυση συγκεκριμένων υπό προβλημάτων, χωρίς ωστόσο και πάλι να δίνεται έμφαση στον τρόπο σχεδίασης και υλοποίησης ενός συστήματος που θα τις εφαρμόζει.

Στην πράξη τόσο κατά την δημιουργία όσο και κατά την επέκταση – συντήρηση ενός μεταγλωττιστή, διαφαίνεται η ανάγκη αντιμετώπισης του προβλήματος πρωτίστως από τη σκοπιά του Μηχανικού Λογισμικού. Η Μηχανική Λογισμικού αποτελεί ένα είδος εφαρμοσμένης-πειθαρχημένης μηχανικής που ασχολείται με όλες τις πτυχές της παραγωγής λογισμικού, όπου με τον όρο «πειθαρχημένης μηχανικής» γίνεται ο προσδιορισμός του πλαισίου εργασίας των Μηχανικών Λογισμικού, δηλαδή την εφαρμογή τεκμηριωμένων (επιστημονικά) θεωριών, μεθόδων και εργαλείων όπου κρίνεται σκόπιμο, με σκοπό την παραγωγή ποιοτικού λογισμικού (Sommerville, 2007). Βασικό στάδιο της Μηχανικής Λογισμικού είναι η Αρχιτεκτονική Σχεδίαση στα πλαίσια της ανάπτυξης του μοντέλου σχεδίασης του λογισμικού. Η Αρχιτεκτονική Σχεδίαση καθορίζει τις σχέσεις μεταξύ των βασικών δομικών στοιχείων του λογισμικού καθώς και τα Σχεδιαστικά Πρότυπα που μπορούν να χρησιμοποιηθούν για την ικανοποίηση των απαιτήσεων του συστήματος (Pressman, 2001). Η σχετική με την Αρχιτεκτονική Λογισμικού βιβλιογραφία, παρέχει ένα ευρύ πλαίσιο μεθοδολογιών και μοντέλων ανάλυσης, σχεδίασης, υλοποίησης, ελέγχου και συντήρησης λογισμικού, χωρίς ωστόσο να υπάρχει άμεσος συσχετισμός και εξειδίκευση στο πεδίο των μεταγλωττιστών.

Προκύπτει λοιπόν η ανάγκη για αναζήτηση και διερεύνηση σχετικά με την εφαρμογή δοκιμασμένων Σχεδιαστικών Προτύπων (design patterns) για την σχεδίαση και υλοποίηση ποιοτικού λογισμικού μεταγλωττιστών, ως μία κάθετη προσέγγιση που εκκινεί από την μοντελοποιημένη αρχιτεκτονική σχεδίαση μέχρι και την ενδεικτική υλοποίηση κώδικα εφαρμογής τους.

1.2 Ανασκόπηση βιβλιογραφίας

Η σχετική με το υπό διερεύνηση πρόβλημα βιβλιογραφία επικεντρώνεται όσο αναφορά το επιστημονικό πεδίο της μηχανικής λογισμικού (software engineering) και αρχιτεκτονικής σχεδίασης στους τίτλους «Software Engineering : A practitioner's approach» (Pressman, 2001) και «Software Engineering» (Sommerville, 2007), όσο αναφορά το πεδίο της σχεδίασης μεταγλωττιστών στους τίτλους «Compilers Principles, Techniques, & Tools» (Aho, Lam, Sethi, & Ullman, 2007) και «Engineering a Compiler» (Cooper & Torczon, 2012), όσο αναφορά τη σχεδίαση με βάση Object Oriented σχεδιαστικά πρότυπα (design pattern) στον τίτλο «Design Patterns : Elements of Reusable Object-Oriented Software» (Gamma, Helm, Johnson, & Vlisssides, 1994), όσο αναφορά την μοντελοποίηση της αρχιτεκτονικής σχεδίασης στον τίτλο «Appling UML and Patterns : An introduction to Object-Oriented Analysis and Design and Iterative Development» (Larman, 2004), και όσο αναφορά την υλοποίηση του scanner και parser στον τίτλο «flex & bison» (Levine, 2009). Επιπλέον σχετικά με το υπό διερεύνηση πρόβλημα λήφθηκαν υπόψη, η έρευνα που παρουσιάζεται στην δημοσίευση «Attribute Based Compiler Implemented Using Visitor Pattern» (Norman, 2004), η προσέγγιση «Design Patterns Automation with Template Library» (Sergiu, Ning, & Narayan, 2005), καθώς και η δημοσίευση «The VISITOR Pattern as a Reusable, Generic, Type-Safe Component» (Bruno, Meng, & Jeremy, 2008).

1.3 Σκοπιμότητα / Στόχοι

Με βάση την ανάγκη για αναζήτηση και διερεύνηση σχετικά με την εφαρμογή δοκιμασμένων Σχεδιαστικών Προτύπων (design patterns) για την σχεδίαση και υλοποίηση μεταγλωττιστών, η παρούσα εργασία θέτει μια σειρά από επιμέρους στόχους:

- Διερεύνηση της βιβλιογραφίας σχετικά με την κατασκευή (μηχανική) μεταγλωττιστών προκειμένου να εντοπισθούν κοινά ή επαναλαμβανόμενα υπό προβλήματα όπως μεθοδολογίες, λειτουργίες ή δομές αναπαράστασης πληροφορίας, των οποίων η υλοποίηση μπορεί να σχεδιασθεί και υλοποιηθεί καλύτερα δια μέσου δοκιμασμένων σχεδιαστικών προτύπων (design pattern)
- Διερεύνηση της βιβλιογραφίας σχετικά με κοινά προβλήματα και δυσκολίες που αντιμετωπίζονται κατά την σχεδίαση και υλοποίηση λογισμικού μεταγλωττιστών
- Διερεύνηση βιβλιογραφίας και εύρεση κατάλληλων και δοκιμασμένων σχεδιαστικών προτύπων, τα οποία με βάση τις προδιαγραφές τους, αναμένεται να βελτιώσουν την ποιότητα της υλοποίησης ενός μεταγλωττιστή, αντιμετωπίζοντας τυχόν συνηθισμένα προβλήματα. Παράλληλα προσδιορισμός κατάλληλου τύπου γλώσσας και στυλ προγραμματισμού που εξυπηρετούν τα συγκεκριμένα σχεδιαστικά πρότυπα
- Επιλογή τυποποιημένης μοντελοποίησης για την γραφική αναπαράσταση της αρχιτεκτονικής σχεδίασης της υλοποίησης με έμφαση στην εφαρμογή και λειτουργία των σχεδιαστικών προτύπων
- Σχεδίαση και μερική υλοποίηση τμήματος μεταγλωττιστή με χρήση των επιλεγμένων σχεδιαστικών προτύπων, προκειμένου να διερευνηθεί στην πράξη η αποτελεσματικότητα τους, η ευχρηστία τους, τυχών σχεδιαστικά προβλήματα ή αδυναμίες, τρόποι επίλυσης προβλημάτων, η ποιότητα του

παραγόμενου λογισμικού (κατανόηση, ευκολία συντήρησης και επέκτασης, κλπ), η ικανοποίηση των προδιαγραφών του μεταγλωττιστή

- Ανάλυση των προτύπων διασφάλισης ποιότητας λογισμικού και αναζήτηση τρόπων για την μέτρηση των ποιοτικών χαρακτηριστικών του λογισμικού. Έλεγχος της δυνατότητας εφαρμογής μετρήσεων ποιότητας λογισμικού για την αποτύπωση των χαρακτηριστικών που προκύπτουν από την εφαρμογή των σχεδιαστικών προτύπων
- Διερεύνηση εναλλακτικών ερευνητικών προσεγγίσεων και σύγκριση με την υλοποίηση, ανάδειξη ομοιοτήτων και διαφορών, τρόποι εφαρμογής τους

Σκοπός της εργασίας, μέσω της ικανοποίησης των ανωτέρω στόχων, είναι η δημιουργία ενός πρότυπου αντικειμενοστραφούς (αρχιτεκτονικού) σχεδιαστικού μοντέλου μεταγλωττιστών, με χρήση δοκιμασμένων σχεδιαστικών προτύπων, με δυνατότητα επέκτασης και χαρακτηριστικά επαναληψιμότητας για την εφαρμογή του σε παρόμοια ή διαφορετικά (αλλά επαναλαμβανόμενα) στάδια μεταγλώττισης ή/και σε αντίστοιχα προβλήματα της αρχιτεκτονικής σχεδίασης λογισμικού. Ιδιαίτερη έμφαση έχει δοθεί στα επιλεγμένα σχεδιαστικά πρότυπα ως μια κάθετη προσέγγιση που εκκινεί από την μοντελοποιημένη αρχιτεκτονική σχεδίαση μέχρι και την ενδεικτική υλοποίηση κώδικα εφαρμογής τους. Επιπρόσθετος σκοπός είναι η παρούσα εργασία να μπορεί να χρησιμοποιηθεί ως ένα κείμενο / οδηγός εναλλακτικής αναφοράς ή εκκίνησης για όσους επιθυμούν να ασχοληθούν με την κατανόηση και ανάπτυξη μεταγλωττιστών. Ακόμη διερευνούνται τα μειονεκτήματα και πλεονεκτήματα καθώς και οι συνθήκες για την εφαρμογή κατάλληλων σχεδιαστικών προτύπων. Επίσης μέσα από την διαδικασία της έρευνας αναμένεται η δημιουργία νέων η προσαρμοσμένων σχεδιαστικών προτύπων (προερχόμενα από το πεδίο των μεταγλωττιστών) με στοιχεία επαναληψιμότητας και δυνατότητα εφαρμογής σε παρόμοια προβλήματα.

Είναι σημαντικό να τονισθεί ότι η εργασία δεν επικεντρώνεται στην επί της ουσίας εξαντλητική ανάλυση, λύση και υλοποίηση των επιμέρους προβλημάτων και λειτουργιών (business logic) ενός μεταγλωττιστή, αλλά στην εφαρμογή κατάλληλων σχεδιαστικών προτύπων (design pattern) που συνδράμουν θετικά στην υλοποίηση αυτών ή παρόμοιων λύσεων, ενισχύοντας παράλληλα τα ποιοτικά χαρακτηριστικά του παραγόμενου λογισμικού.

1.4 Μεθοδολογία

Για την αποτύπωση της αρχιτεκτονικής σχεδίασης του μεταγλωττιστή της εργασίας χρησιμοποιείται το τυποποιημένο πρότυπο διαγραμμάτων UML και συγκεκριμένα τα διαγράμματα κλάσεων (class diagrams), αντικειμένων (object diagrams) και ακολουθίας (sequence diagrams). Για την υλοποίηση καθώς και τα παραδείγματα κώδικα του μεταγλωττιστή της εργασίας χρησιμοποιείται η αντικειμενοστραφή γλώσσα προγραμματισμού C++. Η εργασία παρουσιάζει την ενδεικτική σχεδίαση και υλοποίηση μέρους ενός μεταγλωττιστή για την γλωσσά προγραμματισμού C, χρησιμοποιώντας την γραμματική της γλώσσας C89 με βάση το σχετικό πρότυπο. Η υλοποίηση του λεκτικού και συντακτικού αναλυτή υλοποιήθηκε με χρήση του λογισμικού Flex & Bison με είσοδο τις κανονικές εκφράσεις (RE) των λέξεων καθώς και τη γραμματική (CFG) της γλώσσας εισόδου.

Κατά τη σχεδίαση του μεταγλωττιστή χρησιμοποιήθηκαν τα αντικειμενοστραφή σχεδιαστικά πρότυπα (design pattern) Façade, Composite, Iterator και Visitor.

Η σχεδίαση και υλοποίηση περιλαμβάνει την ανάπτυξη του front-end τμήματος ενός μεταγλωττιστή, του πίνακα συμβόλων (symbol table), του parse / abstract tree, της three-address code σύνθεσης, καθώς και τη σχεδίαση και υλοποίηση των ενδεικτικών λειτουργιών scope checking, type inference & checking και graph generation για την εξαγωγή των εσωτερικών δομών σε γραφική αναπαράσταση. Παράλληλα δίνονται παραδείγματα εφαρμογής των λειτουργιών των σχεδιαστικών προτύπων σε μια προσπάθεια να διερευνηθούν και παρουσιασθούν τα πλεονεκτήματα και οι τρόποι χρήσης τους. Η μερική αξιολόγηση της υλοποίησης πραγματοποιήθηκε σε επίπεδο μεθόδου ανά module για ένα υποσύνολο των μετρικών McCabe μέσω του λογισμικού SourceMonitor. Η εκτίμηση της ποιότητας της υλοποίησης επικεντρώνεται στο μοντέλο αρχιτεκτονικής σχεδίασης με έμφαση στα ποιοτικά χαρακτηριστικά της συντηρησιμότητας, επεκτασιμότητας, και ευκολίας κατανόησης του παραγόμενου λογισμικού.

1.5 Προαπαιτούμενα

Στην παρούσα εργασία έχει συμπεριληφθεί, σε συνοπτική μορφή, το απαραίτητο θεωρητικό και επιστημονικό υπόβαθρο, προκειμένου να είναι δυνατή η κατανόηση του περιεχομένου της ακόμη και από αναγνώστες που δεν είναι εξοικειωμένοι με την ανάπτυξη μεταγλωττιστών. Ωστόσο δεομένου ότι τα σχεδιαστικά πρότυπα αποσκοπούν στην αναλυτική σχεδίαση και μοντελοποίηση ποιοτικού λογισμικού για δύσκολα προβλήματα και προκειμένου να κατανοηθούν σε βάθος τα αποτελέσματα της εργασίας, συστήνεται ο αναγνώστης να είναι τουλάχιστον εξοικειωμένος με την θεωρία και την εφαρμογή στα επιμέρους πεδία : αντικειμενοστραφής σχεδίασης και προγραμματισμός σε γλώσσα C++, μοντελοποίηση αρχιτεκτονικής σχεδίασης με UML διαγράμματα, βασική θεωρία γράφων, μηχανική των μεταγλωττιστών, και δενδροειδών δομών αναπαράστασης.

Γενικά έχει καταβληθεί ιδιαίτερη προσπάθεια ώστε ο αναγνώστης να μπορεί, με βάση μόνο τις συνοπτικές παρουσιάσεις (σχήματα, διαγράμματα, κώδικας) της εργασίας, να είναι σε θέση να κατανοήσει την εφαρμογή των σχεδιαστικών προτύπων στα επιμέρους προβλήματα ενός μεταγλωττιστή. Επίσης να μπορεί να κατανοήσει τα οφέλη, τα προβλήματα, τις προοπτικές που προκύπτουν από την εφαρμογή τους, και παράλληλα να μπορεί να αντιστοιχήσει τα συμπεράσματα με άλλα παρόμοια προβλήματα στο ευρύτερο πεδίο της αρχιτεκτονικής σχεδίασης λογισμικού.

1.6 Διάρθρωση εργασίας

Στο δεύτερο κεφάλαιο παρουσιάζονται συνοπτικά τα πλεονεκτήματα και τα βασικά χαρακτηριστικά του αντικειμενοστραφούς προγραμματισμού και προσδιορίζεται η εφαρμογή των σχεδιαστικών προτύπων ως μέρος της αρχιτεκτονικής σχεδίασης λογισμικού, στα πλαίσια του ευρύτερου επιστημονικού πεδίου του Software Engineering. Επίσης παρουσιάζεται μια συνοπτική περιγραφή των βασικών διαγραμμάτων UML για την μοντελοποίηση της αρχιτεκτονικής σχεδίασης, καθώς και τα βασικά χαρακτηριστικά της αντικειμενοστραφούς γλώσσας προγραμματισμού C++.

Στο τρίτο κεφάλαιο παρατίθεται μια συνοπτική αναφορά στα πρότυπα διασφάλισης ποιότητας λογισμικού καθώς και του τρόπου καταγραφής της ποιότητας λογισμικού μέσα

από επιμέρους μετρικές, σε μια προσπάθεια αποτύπωσης των αποτελεσμάτων της εφαρμογής των σχεδιαστικών προτύπων στην ποιότητα λογισμικού.

Στο τέταρτο κεφάλαιο δίνεται μια πιο εκτενή παρουσίαση των σχεδιαστικών προτύπων Façade, Composite, Iterator και Visitor με βάση τις προδιαγραφές της σχετικής βιβλιογραφίας.

Στο πέμπτο κεφάλαιο παρατίθεται μια συνοπτική αναφορά στις βασικές έννοιες και θεωρία των επιμέρους πεδίων σχετικά με την σχεδίαση μεταγλωττιστών. Δίνεται μια συνοπτική παρουσίαση των βασικών τμημάτων και λειτουργιών εντός τυπικού μεταγλωττιστή, σε βαθμό και έκταση που να εξυπηρετούν την κατανόηση της εφαρμογής των σχεδιαστικών προτύπων της παρούσας εργασίας.

Στο έκτο κεφάλαιο δίνεται μια συνοπτική παρουσίαση της χρήσης και των χαρακτηριστικών του λογισμικού Flex και Bison με τα οποία και υλοποιείται ο λεκτικός και συντακτικός αναλυτής της υλοποίησης της εργασίας.

Στο έβδομο κεφάλαιο διερευνούνται στην πράξη η εφαρμογή των συγκριμένων σχεδιαστικών προτύπων αντικειμενοστραφούς προγραμματισμού Façade, Composite, Iterator και Visitor, μέσω της αρχιτεκτονικής σχεδίασης του front-end τμήματος ενός μεταγλωττιστή της γνωστής γλώσσας προγραμματισμού C89. Επίσης παρουσιάζεται η σχεδίαση του πίνακα συμβόλων (symbol table), του parse / abstract tree, της three-address code σύνθεσης και της εσωτερικής αναπαράστασής του, καθώς και η σχεδίαση και υλοποίηση των ενδεικτικών λειτουργιών scope checking, type inference & checking και graph generation. Για κάθε υποπερίπτωση παρουσιάζονται διαγράμματα σχεδίασης, ενδεικτικός κώδικας υλοποίησης καθώς και παραδείγματα εφαρμογής και χρήσης των σχεδιαστικών προτύπων. Παράλληλα αποτυπώνονται τυχόν σχεδιαστικά προβλήματα και προτείνονται σχεδιαστικές λύσεις για την αντιμετώπισή τους.

Στο όγδοο κεφάλαιο παρουσιάζεται μια σύγκριση μεταξύ του σχεδιαστικού προτύπου visitor και της σχεδιαστικής προσέγγισης inheritance, καθώς και μια συνοπτική αναφορά της πρόσφατης προγραμματιστικής προσέγγισης Aspect-Oriented Programming. Επιχειρείται μια αξιολόγηση – μέτρηση των σχεδιαστικών προτύπων που εφαρμόστηκαν, με βάση σύνολα γνωστών μετρικών καθώς και μέσω ασυμπτωτικών εκτιμήσεων σε σχέση με συγκεκριμένους δείκτες και χαρακτηριστικά. Παρουσιάζεται μία εναλλακτική προσέγγιση του σχεδιαστικού προτύπου visitor με βάση σχετική έρευνα και συγκρίνεται με τα σχεδιαστικά πρότυπα της εργασίας. Επίσης παρατίθεται ένας συγκεντρωτικός πίνακας εφαρμογής των σχεδιαστικών προτύπων της εργασίας, καθώς και ένας διαγραμματικός οδηγός επιλογής τους, και μπορούν να χρησιμοποιηθούν συνδυαστικά ως αναλυτικός οδηγός για την σχεδίαση, εφαρμογή και υλοποίηση των σχετικών σχεδιαστικών προτύπων κατά περίπτωση.

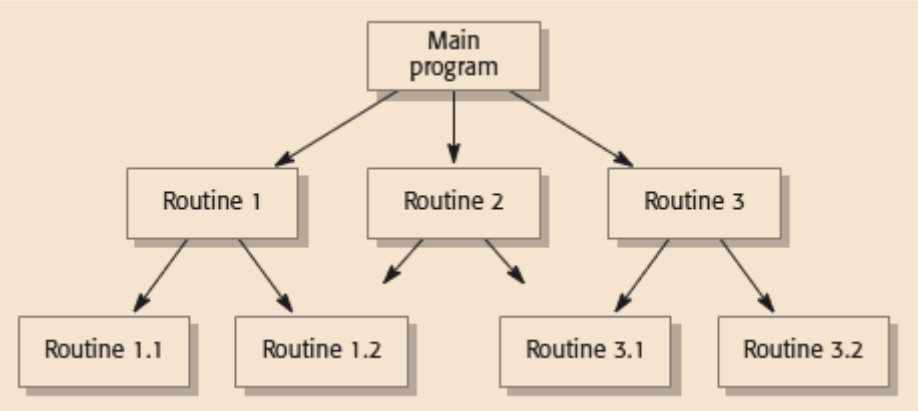
Στο ένατο κεφάλαιο παρατίθενται συγκεντρωτικά τα ευρήματα και τα συμπεράσματα που προέκυψαν από την εκπόνηση της εργασίας και της σχετικής έρευνας. Επίσης επισημαίνονται οι αντικειμενικές δυσκολίες που προέκυψαν και προτείνονται κατευθύνσεις για μελλοντική έρευνα.

Στα παραρτήματα περιλαμβάνονται οι κανονικές εκφράσεις (REs) και η γραμματική (CFG) της γλώσσας C89, σχηματικές δενδροειδής αναπαραστάσεις καθώς και πίνακες προσδιορισμού των ιδιοτήτων των αναγνωριστικών της C89 γραμματικής. Επίσης περιλαμβάνονται ενδεικτικές γραφικές αναπαραστάσεις του πίνακα συμβόλων καθώς και παραδειγμάτων pares / abstract trees. Τέλος παρουσιάζονται ενδεικτικές αναφορές μετρήσεων της υλοποίησης της εργασίας.

2 Αντικειμενοστραφής Προγραμματισμός

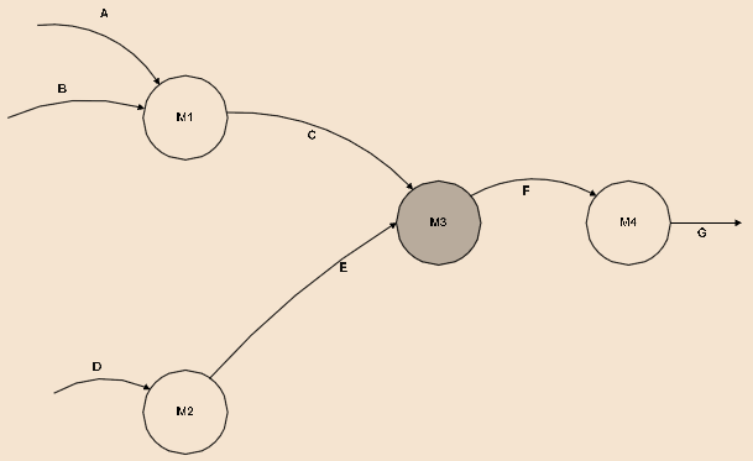
2.1 Γλώσσες Προγραμματισμού

Η συντριπτική πλειοψηφία των μηχανικών λογισμικού σχεδιάζουν και αναπτύσσουν κώδικα χρησιμοποιώντας γλώσσες προγραμματισμού υψηλού επιπέδου. Οι πρώτες γλώσσες προγραμματισμού που αναπτύχθηκαν χρησιμοποιούσαν ρητές αναφορές στην κατάσταση του περιβάλλοντος εκτέλεσης (imperative languages), οι περισσότερες από τις οποίες ακολουθούσαν το διαδικαστικό πρότυπο προγραμματισμού. Οι γλώσσες διαδικαστικού προγραμματισμού (procedural programming languages), χρησιμοποιούνται ευρέως μέχρι και σήμερα, βασίζονται στη λογική της top down ανάλυσης ενός προβλήματος, όπου το πρόβλημα αρχικά αντιμετωπίζεται ως ενιαίο (procedure) και στη συνέχεια κατακερματίζεται σε μικρότερα υπό προβλήματα (sub procedures), μέχρι την πλήρη ανάλυσή του σε απλά προβλήματα που μπορούν να λυθούν (Εικόνα 2-1). Η ροή του προγράμματος εκκινεί από την αρχική διαδικασία και εξελίσσεται μέσω πραγματοποίησης διαδοχικών κλήσεων διαδικασιών. Το βασικό πλεονέκτημα της διαδικαστικών γλωσσών είναι το ίδιο το δομικό τους στοιχείο, δηλαδή η διαδικασία, μέσω της οποίας είναι δυνατή η ανάλυση του προβλήματος σε μικρότερα μέρη, ή διαφορετικά η δόμηση της λύσης από επιμέρους προβλήματα συνθέτοντας την τελική λύση μέσω διαδικασιών (Sommerville, 2007). Επίσης βασικό χαρακτηριστικό μιας διαδικασίας είναι η δυνατότητα επαναχρησιμοποίηση του κώδικα μιας της.



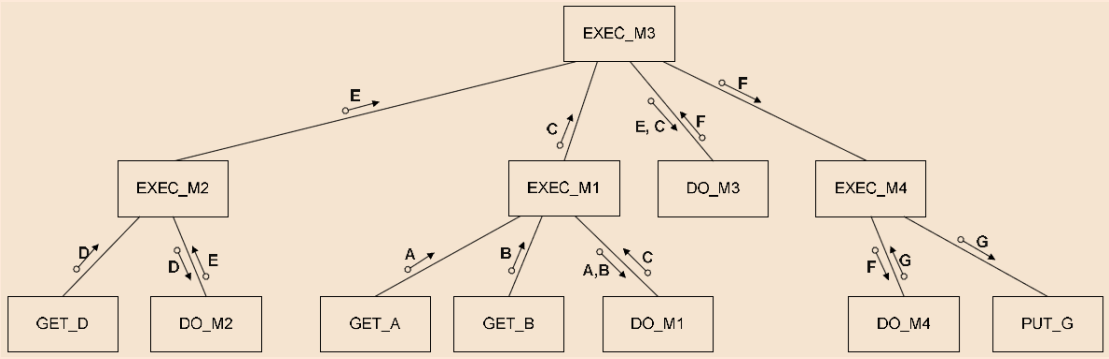
Εικόνα 2-1 : Διάγραμμα Δομής (διαδικασιών) Προγράμματος

Η αρχιτεκτονική σχεδίαση λογισμικού διαδικαστικού προγραμματισμού καθώς και η τεκμηρίωσή του, αναπαριστάται με μια σειρά από τυποποιημένα διαγράμματα, όπως το Διάγραμμα Ροής Δεδομένων (ΔΡΔ, dataflow diagram, Εικόνα 2-2) το οποίο, με συγκεκριμένη μεθοδολογία μετατρέπεται σε Διαγράμματα Δομής Προγράμματος (ΔΔ, program structure diagram, Εικόνα 2-3), όπου στο δεύτερο αποτυπώνεται ο κατακερματισμός του προγράμματος σε επιμέρους διαδικασίες.



Εικόνα 2-2 : Διάγραμμα Ροής Δεδομένων

Η ανάλυση ενός προβλήματος και υλοποίηση της λύσης του μέσω του διαδικαστικού προγραμματισμού, παράγει προγράμματα δομημένα όσο αναφορά τον κατακερματισμό σε επιμέρους διαδικασίες, ωστόσο προσαρμοσμένα και στενά συνδεδεμένα με το εκάστοτε πρόβλημα. Η κάθε διαδικασία ή ένα σύνολο διαδικασιών, επικεντρώνεται αποκλειστικά στη λύση ενός μέρους του προβλήματος χωρίς να περιγράφει ή να αναπαριστά το ίδιο το πρόβλημα. Επίσης επειδή τα δεδομένα δεν συσχετίζονται με κάποιο μοντελοποιημένο τρόπο με τις ενέργειες που πραγματοποιούνται σε αυτά, η κατανόηση ενός διαδικαστικού προγράμματος καθίσταται αρκετά δύσκολη και χρονοβόρα (ακόμα και από την τεκμηρίωση του), ειδικά όταν προγραμματιστές καλούνται να πραγματοποιούσουν αλλαγές ή βελτιώσεις σε μεταγενέστερο χρόνο. Οι ίδιες δε οι τροποποιήσεις στη δομή ενός προγράμματος (αφορά πολλές διαδικασίες) συνήθως επιφέρουν αλλαγές στο σύνολο του προγράμματος καθιστώντας τον κώδικα δύσκολα συντηρήσιμο.



Εικόνα 2-3 : Διάγραμμα Δομής προγράμματος

Για το πεδίο της αρχιτεκτονικής λογισμικού (software engineering) που σχετίζεται με τον διαδικαστικό προγραμματισμό, υπάρχουν αναλυτικά κείμενα και εκτενή βιβλιογραφία, ενδεικτικά (Pressman, 2001), (Sommerville, 2007). Ωστόσο η παρούσα εργασία επικεντρώνεται στη διερεύνηση της εφαρμογής σχεδιαστικών προτύπων σε αντικειμενοστραφείς γλώσσες προγραμματισμού.

2.2 Αντικειμενοστραφής προσέγγιση

Μια ευρέως διαδεδομένη (κατά πολλούς πλέον κυρίαρχη), εναλλακτική προσέγγιση προγραμματισμού είναι ο αντικειμενοστραφής προγραμματισμός, δια μέσου μιας

πληθώρας ισχυρών και συνεχώς εξελισσόμενων γλωσσών προγραμματισμού (C++, Java, κλπ).

Μια αντικειμενοστραφής γλώσσα προγραμματισμού (Object-Oriented Programming) ή αλλιώς γλώσσα προσανατολισμένη στο αντικείμενο, είναι μία γλώσσα προγραμματισμού που χρησιμοποιεί κλάσεις (classes) και αντικείμενα (objects) για τη δημιουργία μοντέλων βασισμένων στο περιβάλλον του πραγματικού κόσμου. Τα αντικείμενα ενσωματώνουν τόσο δεδομένα (data) όσο και υπηρεσίες – ενέργειες (methods) και ανταλλάσουν (περνούν) μεταξύ τους μηνύματα (η κλήσεις) κατά την απαίτηση μίας υπηρεσίας ή δεδομένων. Οι κλάσεις δηλώνονται ως συλλογές αντικειμένων τα οποία δημιουργούνται, μεταβάλλονται και διαγράφονται κατά τον χρόνο εκτέλεσης του προγράμματος. Χαρακτηριστικό των κλάσεων στον αντικειμενοστραφή προγραμματισμό είναι η δυνατότητα δημιουργίας κλάσεων που κληρονομούν χαρακτηριστικά (δεδομένα και λειτουργίες) από άλλες κλάσεις, ιδιότητα αναφερόμενη ως Κληρονομικότητα.

Το γεγονός ότι τα αντικείμενα υφίστανται μόνο κατά τον χρόνο εκτέλεσης, απαιτεί ένα σχολαστικό και μοντελοποιημένο τρόπο (με χρήση διαγραμμάτων) σχεδίασης και αναπαράστασης της δομής και λειτουργίας του λογισμικού. Γενικά ένα προγραμματιστής προκειμένου να επιλύσει ένα πρόβλημα με αντικειμενοστραφή προγραμματισμό πρέπει πρώτα να σχεδιάσει το λογισμικό μέσα από μια σειρά διαγραμμάτων. Ωστόσο το αποτέλεσμα προσδίδει σαφέστερη εικόνα της δομής του προγράμματος (σε σχέση με τον διαδικαστικό προγραμματισμό) καθιστώντας τον κώδικα εύκολα συντηρήσιμο σε τυχών μεταγενέστερες παρεμβάσεις. Καλά σχολιασμένες κλάσεις και αντικείμενα σε συνδυασμό με τυποποιημένα διαγράμματα, προσδίδουν μία σαφή εικόνα του προγράμματος και της λειτουργίας που αυτό επιτελεί ακόμα και για προγραμματιστές που θα ασχοληθούν για πρώτη φορά με την συντήρηση ή βελτίωσή του. Επίσης οι τυχών αλλαγές σε δεδομένα ή λειτουργίες έχουν περιορισμένη έκταση εφόσον η δομή των κλάσεων (δεδομένα και λειτουργίες) έχουν ποιο συμπαγή και αυτοτελή συμπεριφορά. Ειδικά η κληρονομικότητα συνήθως επιτρέπει στους προγραμματιστές να πραγματοποιούν σύντομα στοχευόμενες και περιορισμένης έκτασης παρεμβάσεις για την τροποποίηση και συντήρηση του λογισμικού (Sommerville, 2007).

Γενικά ο αντικειμενοστραφής προγραμματισμό χρησιμοποιεί αρκετά διαφορετική μεθοδολογία τόσο στην μοντελοποίηση και σχεδίαση όσο και κατά την παραγωγή του κώδικα σε σχέση με τον διαδικαστικό προγραμματισμό. Ο παρακάτω πίνακας (Wikipedia, 2014) διατυπώνει τις βασικές διαφορές τους :

Πίνακας 2-1 : Διαφορές Διαδικαστικού - Αντικειμενοστραφούς προγραμματισμού

Procedural	Object-oriented
procedure	method
record	object
module	class
procedure call	message

Η ανάπτυξη διαγραμμάτων αποτύπωσης της δομής ενός αντικειμενοστραφούς προγράμματος σε αρχικό στάδιο, παρέχει στον προγραμματιστή ένα πλάνο της λειτουργίας και δομής του προγράμματος πριν την ανάπτυξη του κώδικα. Η δυνατότητα συσχέτισης του κώδικα με πραγματικά παραδείγματα καθιστά τη διαδικασία συγγραφής κώδικα

απλούστερη. Ο διαδικαστικός προγραμματισμός μπορεί να δελεάζει ως λύση, εφόσον μεταβαίνουμε γρήγορα στη συγγραφή κώδικα, ωστόσο ενδέχεται να επιφέρει σημαντική καθυστέρηση σε μεταγενέστερες παρεμβάσεις.

Η μοντελοποίηση του προγράμματος σε κλάσεις αντικειμένων σε συνδυασμό με την ιδιότητα της Κληρονομικότητας καθιστούν τον αντικειμενοστραφή προγραμματισμό ιδανικό για σύνθετα προβλήματα που διαχειρίζονται **συλλογές** ή **συνθέσεις αντικειμένων** με διακριτά δεδομένα, λειτουργίες και παρεμφερή ή επαναλαμβανόμενη συμπεριφορά ανά ομάδα.

Ένα χαρακτηριστικό του αντικειμενοστραφούς προγραμματισμού είναι ότι το μοντέλο (κλάσεις αντικειμένων) της υλοποίησης που χρησιμοποιήθηκε για την επίλυση ενός προβλήματος, μπορεί να επαναχρησιμοποιηθεί ή/και τροποποιηθεί, ως πρότυπο σχεδίασης, για την επίλυση παρόμοιων προβλημάτων με την ίδια ή παρεμφερή μοντελοποίηση. Στον διαδικαστικό προγραμματισμό κάτι τέτοιο είναι σαφώς πιο δύσκολο, εφόσον η κάθε διαδικασία επικεντρώνεται στην επίλυση συγκεκριμένου υπό προβλήματος. Αρκεί τα μοντέλα αυτά να είναι σχεδιασμένα κατά τέτοιο τρόπο ώστε να διασφαλίζεται κατά το δυνατό η επαναχρησιμοποίηση τους. Στην περίπτωση αυτή μπορούμε να πούμε ότι μία λύση ή μία αφαιρετική προσέγγιση της λύσης αποτελεί **σχεδιαστικό πρότυπο** (**design pattern**) για προβλήματα του ιδίου τύπου ή με ίδια επιμέρους χαρακτηριστικά.

Στις ενότητες που ακολουθούν θα παρουσιάσουμε συνοπτικά μερικά από τα βασικά δομικά χαρακτηριστικά της αντικειμενοστραφούς προσέγγισης τόσο σε σχέση με την σχεδίαση όσο και με την υλοποίηση.

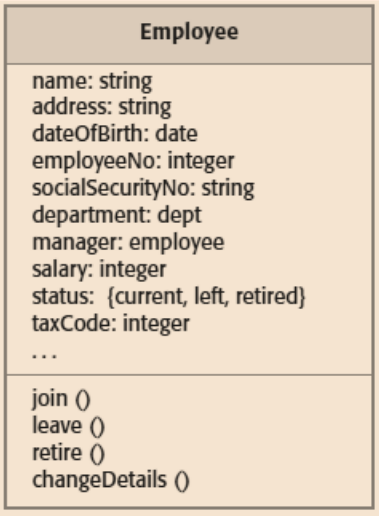
2.2.1 Κλάση αντικειμένων / Αντικείμενο

Υπάρχει μια γενική παραδοχή ότι αντικείμενο είναι η ενθυλάκωση (encapsulation) πληροφορίας, η οποία και αντανακλάται στους ακόλουθους ορισμούς:

> *"Ενα αντικείμενο (object) είναι μια οντότητα που έχει μία κατάσταση και ένα συγκεκριμένο σύνολο λειτουργιών που εφαρμόζονται σε αυτή την κατάσταση. Η κατάσταση αναπαριστάται ως ένα σύνολο ιδιοτήτων του αντικειμένου. Οι εργασίες που σχετίζονται με το αντικείμενο παρέχουν υπηρεσίες σε άλλα αντικείμενα (πελάτες) που αιτούνται αυτές τις υπηρεσίες όταν απαιτείται κάποιος υπολογισμός (Sommerville, 2007)."*

> *"Τα αντικείμενα δημιουργούνται σύμφωνα με τον ορισμό μιας κλάσης αντικειμένου (object class definition). Ο ορισμός μίας κλάσης αντικειμένου είναι ταυτόχρονα μια προδιαγραφή τύπου και ένα πρότυπο (template) για τη δημιουργία αντικειμένων. Περιλαμβάνει δηλώσεις όλων των ιδιοτήτων και λειτουργιών που πρέπει σχετίζονται με ένα αντικείμενο αυτής της κλάσης (Sommerville, 2007)."*

Μία κλάση αντικειμένων (object class) αναπαριστάται στην UML (ενότητα 2.3.1) ως ένα παραλληλόγραμμο, με δύο τμήματα όπου το πρώτο περιέχει τις ιδιότητες (attributes) και το δεύτερο τις λειτουργίες (methods) της κλάσης και των αντικειμένων που αυτή ορίζει και παράγει.

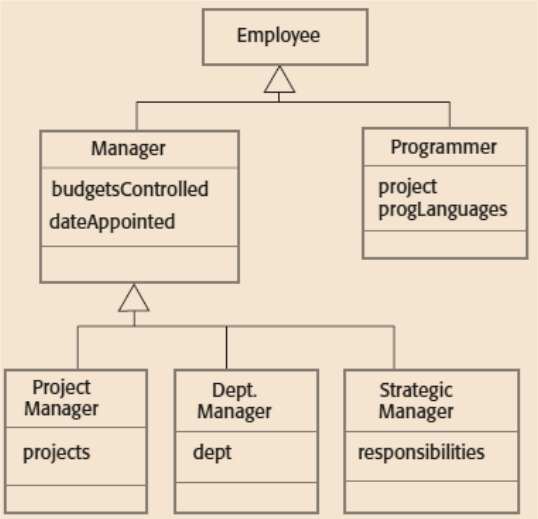


Εικόνα 2-4 : Γραφική αναπαράσταση Κλάσης Αντικειμένων

Η Εικόνα 2-4 (Sommerville, 2007) παρουσιάζει ένα παράδειγμα αναπαράστασης μιας κλάσης αντικειμένων υπαλλήλων με συγκεκριμένες ιδιότητες καθώς και λειτουργίες.

2.2.2 Κληρονομικότητα

Η Κληρονομικότητα (**inheritance**) ως το βασικότερο χαρακτηριστικό του αντικειμενοστραφούς προγραμματισμού επιτρέπει την ιεράρχηση των κλάσεων κατά τέτοιο τρόπο ώστε ένα αντικείμενο να κληρονομεί τις ιδιότητες και τις λειτουργίες από το γονικό του αντικείμενο καθώς και προσθέσει νέες ιδιότητες ή λειτουργίες που αφορούν την κλάση του. Είναι δυνατό μάλιστα να υπερκαλύπτεται (με το ίδιο όνομα) μια λειτουργία του γονικού αντικειμένου. Σε ένα μοντέλο πολύ επίπεδης ιεράρχηση κλάσεων και για μια γονική κλάση, ένα αντικείμενο μπορεί να είναι είτε στιγμιότυπο της γονικής κλάσης είτε στιγμιότυπο κλάσης που την κληρονομούν. Ο σχεδιασμός ένας μοντέλου κλάσεων και των σχέσεων κληρονομικότητας μεταξύ τους αποτελεί καθοριστικό παράγοντα για την επιτυχή σχεδίαση, ποιότητα και υλοποίηση του συστήματος.



Εικόνα 2-5 : Γραφική αναπαράσταση κληρονομικότητας κλάσεων

Η εικόνα Εικόνα 2-5 (Sommerville, 2007) παρουσιάζει ένα παράδειγμα αναπαράστασης της κληρονομικότητας κλάσεων αντικειμένων υπαλλήλων, όπου επιμέρους κλάσεις αντικειμένων υπαλλήλων, κληρονομούν ιεραρχικά ιδιότητες και λειτουργίες από

όλες τις γονικές τους κλάσεις. Η κληρονομικότητα στην UML αποτυπώνεται με ένα βέλος από την υπό κλάση προς την γονική κλάση.

2.3 Ανάλυση προβλήματος και σχεδίαση λύσης

Η μοντελοποίηση ενός προβλήματος / συστήματος υπάγεται στο επιστημονικό πεδίο υπό τον όρο **Software Engineering** (SE), και αποτελεί ένα είδος εφαρμοσμένης-πειθαρχημένης μηχανικής που ασχολείται με όλες τις πτυχές της παραγωγής λογισμικού, δηλαδή από την σύλληψη, την ανάλυση, σχεδίαση, παραγωγή, δοκιμή, τεκμηρίωση, εγκατάσταση και συντήρηση. Ειδικότερα με τον όρο «πειθαρχημένης μηχανικής» γίνεται ο προσδιορισμός του πλαισίου εργασίας των Μηχανικών Λογισμικού (**Software Engineers**) που είναι η εφαρμογή τεκμηριωμένων (επιστημονικά) θεωριών, μεθόδων και εργαλείων όπου κρίνεται σκόπιμο, με σκοπό την παραγωγή ποιοτικού λογισμικού, ωστόσο χρησιμοποιούνται επιλεκτικά και πάντα προσπαθώντας για την ανακάλυψη (η βελτιστοποίηση) λύσεων σε προβλήματα ακόμη και εκεί που δεν υπάρχουν εφαρμόσιμες θεωρίες και μέθοδοι (Sommerville, 2007). Ο προαναφερόμενος ορισμός εξάγει εμμέσως πλην σαφώς το συμπέρασμα ότι η παραγωγή λογισμικού, ως διαδικασία, δεν είναι μια μορφή τέχνης ή ένας στόχος που επιτυγχάνεται τυχαία.

Γενικά και ανεξάρτητα από το επιλεγμένο μοντέλο ανάπτυξης λογισμικού (Software Development Models)[1], η αντικειμενοστραφή μοντελοποίηση ενός προβλήματος ή του προγράμματος / συστήματος που το επιλύει διαχωρίζεται σε τρία βασικά στάδια :

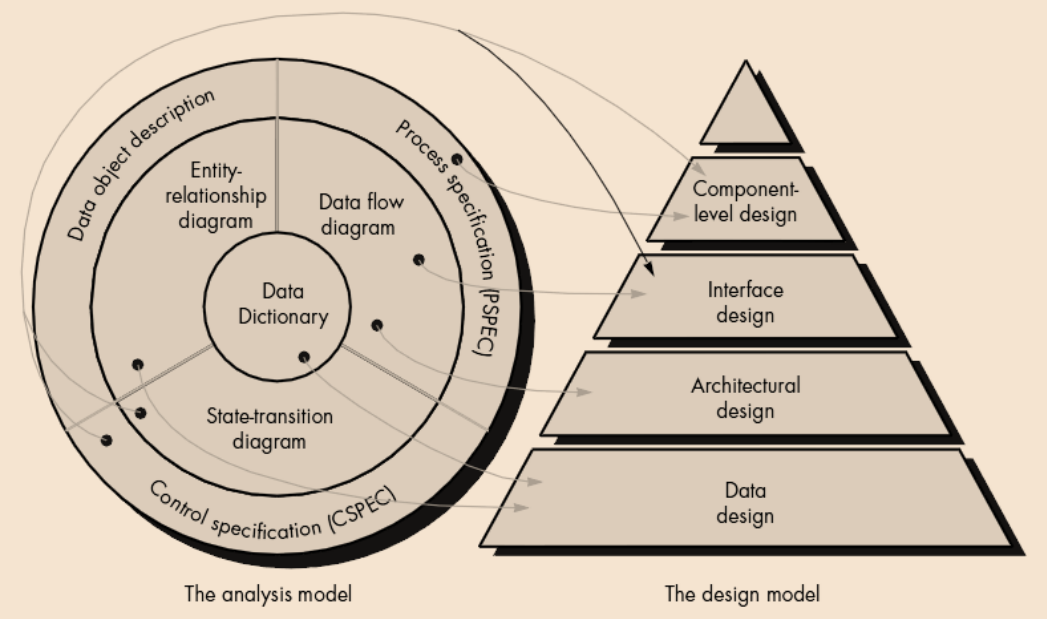
- Object-oriented analysis που παράγει το analysis model
- Object-oriented design που παράγει το design model
- Object-oriented programming που παράγει τον τελικό κώδικα

Το μοντέλο της ανάλυσης του προβλήματος είναι το πρώτο που αναπτύσσεται και συνήθως αποτελείται από ένα σύνολο κειμένων προδιαγραφών (περιγραφή δεδομένων, διεργασιών, ελέγχων) και διαγραμμάτων (οντοτήτων – συσχετίσεων, ροής δεδομένων, μετάβασης καταστάσεων, περιπτώσεων χρήστη, κλπ). Η ανάλυση επικεντρώνεται στην περιγραφή και διερεύνηση του προβλήματος παρά στην λύση του. Από το μοντέλο ανάλυσης μεταβαίνουμε στο μοντέλο σχεδίασης του συστήματος το οποίο συνήθως αποτυπώνεται με ένα σύνολο διαγραμμάτων (κλάσεων, αντικειμένων, ακολουθίας, αλληλεπίδρασης, κλπ). Η σχεδίαση επικεντρώνεται σε μία εννοιολογική λύση (λογισμικού και υλικού) του προβλήματος που ικανοποιεί τις τιθέμενες απαιτήσεις παρά την ίδια τη λύση του. Υπάρχει μια ευρεία γκάμα εργαλείων[2] για την παραγωγή διαγραμμάτων ανάλυσης και σχεδίασης με δυνατότητες αυτόματης παραγωγής / μετατροπής από το ένα στο άλλο ή ακόμα και την παραγωγή της βασικής δομής του τελικού κώδικα. Ανάλυση του τρόπου παραγωγής και της μεθοδολογίας σχεδίασης λογισμικού δεν αποτελεί στόχο της παρούσας εργασίας, ωστόσο λεπτομέρειες και εκτενή ανάλυση υπάρχει στη σχετική βιβλιογραφία, όπως (Pressman, 2001), (Sommerville, 2007).

[1] Στη βιβλιογραφία τα τρία βασικά παραδείγματα Μοντέλων Ανάπτυξης Λογισμικού αναφέρονται ως: The waterfall approach, Iterative development, Component-based software engineering (CBSE) (Sommerville, 2007)

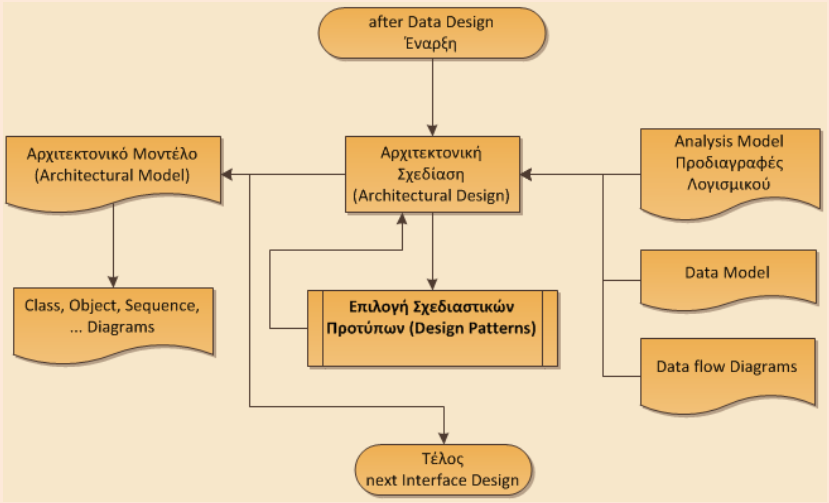
[2] Γενικά αναφέρονται στη βιβλιογραφία ως CASE (Computer-Aided Software Engineering)

Στην Εικόνα 2-6 παρουσιάζεται μια γενική δομή (αφορά και τον διαδικαστικό προγραμματισμό) καθώς και ο συσχετισμός του μοντέλου ανάλυσης με το μοντέλο σχεδίασης ενός συστήματος (Pressman, 2001).



Εικόνα 2-6 : Μετάβαση από το μοντέλο ανάλυσης στο μοντέλο σχεδίασης

Στο μοντέλο σχεδίασης, η σχεδίαση των δεδομένων (data design) αναφέρεται συνήθως στα δεδομένα του συστήματος που αποθηκεύονται σε κάποιο φυσικό μέσο (πχ βάση δεδομένων). Η σχεδίαση της διαπροσωπίας περιγράφει τον τρόπο επικοινωνίας του λογισμικού με τον εαυτό του, με άλλα συστήματα και με τον χρήστη. Η αρχιτεκτονική σχεδίαση (**architectural design**) καθορίζει τις σχέσεις μεταξύ των βασικών δομικών στοιχείων του λογισμικού καθώς και τα σχεδιαστικά πρότυπα (**design patterns**) που μπορούν να χρησιμοποιηθούν για την ικανοποίηση των απαιτήσεων του συστήματος (Pressman, 2001). Το στάδιο αυτό αποτελεί ένα από τα σημαντικότερα στάδια στη σχεδίαση του λογισμικού, ειδικά για δύσκολα προβλήματα που διαχειρίζονται σύνθετες και μεγάλες δομές αντικειμένων.



Εικόνα 2-7 : Αρχιτεκτονική σχεδίαση λογισμικού

Από τα έως τώρα αναφερόμενα, είναι σαφές ότι η εφαρμογή κατάλληλων και δοκιμασμένων σχεδιαστικών προτύπων (**design patterns**) κατά τη φάση της αρχιτεκτονικής σχεδίασης (Εικόνα 2-7) του αντικειμενοστραφούς λογισμικού, αποτελεί καθοριστικό παράγοντα τόσο για την ικανοποίηση των προδιαγραφών όσο και για την ποιότητα του παραγόμενου λογισμικού και γενικά του συστήματος.

2.3.1 Μοντελοποίηση Ανάλυσης και Σχεδίασης

Όπως έχει αναφερθεί, τα μοντέλα της ανάλυσης και της σχεδίασης του λογισμικού περιέχουν ένα σύνολο από κείμενα και διαγράμματα που τα περιγράφουν. Το σύνολο των διαγραμμάτων που χρησιμοποιούνται προκειμένου να έχουν ενιαία δομή και να μπορούν να κατανοηθούν από ανεξάρτητες ομάδες εργασίας, ακολουθούν καθορισμένα πρότυπα. Ένα από τα ποιο γνωστά και ευρέως χρησιμοποιούμενα πρότυπα είναι η ενοποιημένη γλώσσα μοντελοποίησης UML (**Unified Modeling Language**).

Η **UML** είναι μια οπτική (visual) γλώσσα για την προδιαγραφή, κατασκευή και τεκμηρίωση των αντικειμένων – συστατικών ενός συστήματος (Larman, 2004). Με τον όρο οπτική γλώσσα, η UML υποδεικνύεται ως ένα εξορισμού πρότυπο συμβολισμού με διαγράμματα για τον σχεδιασμό και παρουσίαση εικόνων (συμπεριλαμβανημένου και κειμένου) σχετικά με τη σχεδίαση και μοντελοποίηση αντικειμενοστραφούς κυρίως λογισμικού.

2.3.2 Διαγράμματα UML

Η UML[3] παρέχει μια πληθώρα διαγραμμάτων για την σχεδίαση αντικειμενοστραφούς λογισμικού που αναλύονται εκτενώς από τα σχετικά πρότυπα[4] και τη σχετική βιβλιογραφία (Larman, 2004). Ωστόσο για τις ανάγκες της παρούσας εργασίας, παρατίθεται παρακάτω μία σύντομη (όχι πλήρης) περιγραφή τριών (σημαντικότερων) από αυτά, τα οποία χρησιμοποιούμε τόσο για να παρουσιάσουμε τα σχεδιαστικά πρότυπα που εφαρμόζουμε όσο και για την μοντελοποίηση της αρχιτεκτονικής σχεδίασης της υλοποίησης της εργασίας.

2.3.2.1 Διάγραμμα κλάσεων (class diagrams)

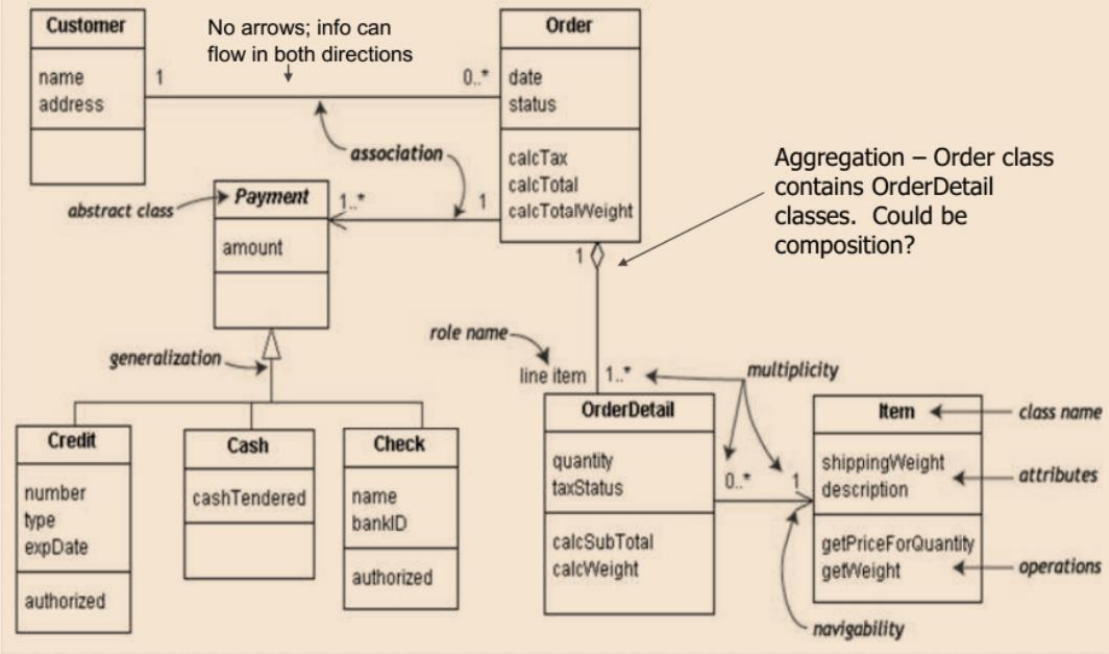
Ένα διάγραμμα κλάσης (**class diagrams**) αποτυπώνει την στατική δομή των κλάσεων του συστήματος και των σχέσεων μεταξύ τους και συνήθως αποτελείται από κλάσεις (**classes**), Διαπροσωπίες (**interfaces**), Συνεργασίες (**collaborations**), Σχέσεις (**relationships**). Χρησιμοποιείται α) κατά την ανάλυση προκειμένου να περιγράψει τις λειτουργικές απαιτήσεις και β) κατά τον σχεδιασμό για να περιγράψει το λεξιλόγιο του συστήματος, τις συνεργασίες, καθώς και το λογικό σχήμα της βάσης δεδομένων.

Οι αντικειμενοστραφείς σχέσεις (**OO relationships**) αναλύονται ως εξής :

- Γενίκευση - **Generalization** (parent-child relationship) η Κληρονομικότητα
- Συσχέτιση - **Association** (student enrolls in course), κατηγοριοποιείται περεταίρω σε:
    - **Composition** (kind of Whole-Part)
    - **Aggregation** (kind of Container-Containee)

[3] Πίνακας με τα πρότυπα όλων των διαθέσιμων εκδόσεων της UML περιέχεται στην σελίδα http://www.omg.org/spec/UML/ του Object Management Group (OMG).

[4] Η ποιο πρόσφατη έκδοση 2.4.1 (Αύγουστος 2011) περιγράφεται από τα πρότυπα ISO/IEC 19505-1 (ISO, 2012) και ISO/IEC 19505-2 (ISO, 2012)

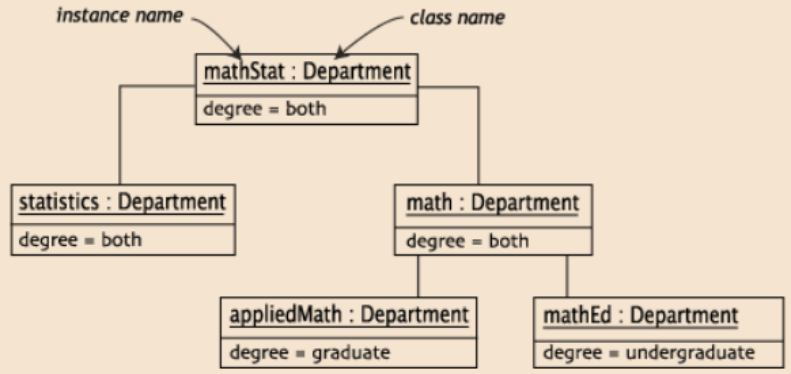


Εικόνα 2-8 : Διάγραμμα Κλάσεων (Class Diagram)

Η Εικόνα 2-8 παρουσιάζει συνοπτικά τα κυριότερα χαρακτηριστικά ενός διαγράμματος κλάσεων, σε ένα ενδεικτικό παράδειγμα σχεδίασης ενός συστήματος παραγγελιών.

2.3.2.2 *Διαγράμματα αντικειμένων (object diagrams)*

Τα διαγράμματα αντικειμένων (**object diagrams**) περιγράφουν – ορίζουν ένα σύνολο αντικειμένων (που δημιουργούνται με βάση την κλάση τους) καθώς και την σχέσεις τους μια δεδομένη χρονική στιγμή. Ένα διάγραμμα αντικειμένων συνήθως περιέχει Αντικείμενα (objects) και Συνδέσμους (Links). Τα διαγράμματα αντικειμένων χρησιμοποιούνται για να καταγράψουν στατικές δομές αντικειμένων και είναι ένα στιγμιότυπο του διαγράμματος κλάσεων ή στατική όψη ενός διαγράμματος συνεργασίας / ακολουθίας.



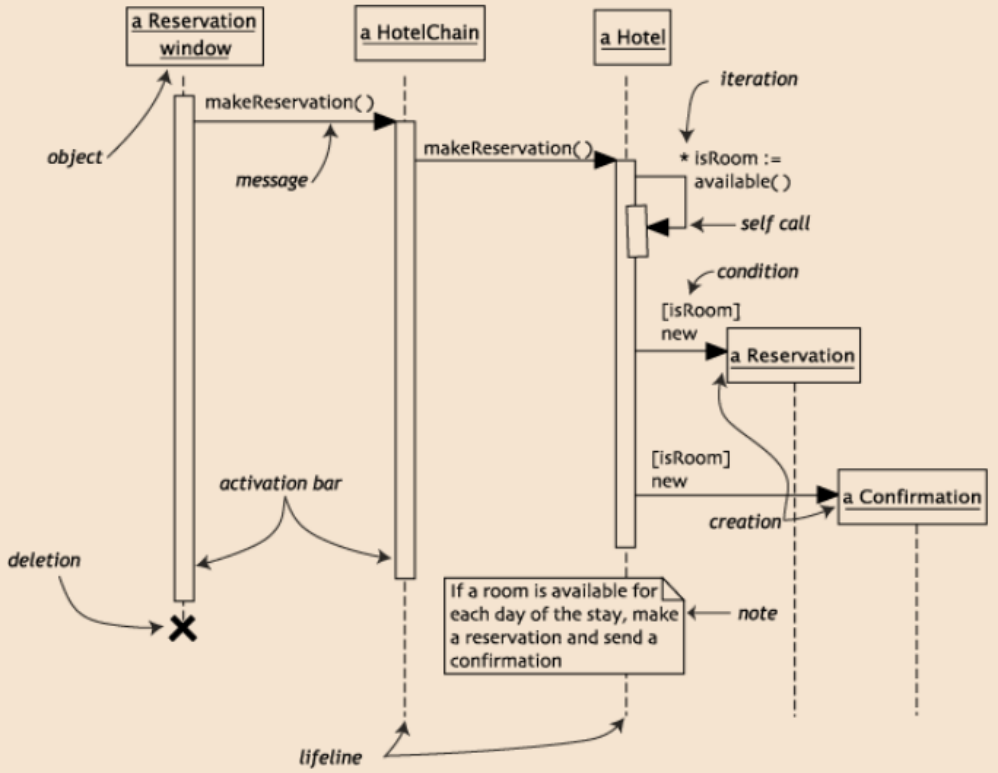
Εικόνα 2-9 : Διάγραμμα Αντικειμένων (Object Diagram)

Η Εικόνα 2-9 παρουσιάζει συνοπτικά τα κυριότερα χαρακτηριστικά ενός διαγράμματος αντικειμένων της UML, σε ένα ενδεικτικό παράδειγμα.

2.3.2.3 *Διαγράμματα ακολουθίας (sequence diagrams)*

Τα διαγράμματα ακολουθίας (**sequence diagrams**) παρουσιάζουν την αλληλεπίδραση των αντικειμένων μέσω της ανταλλαγής μηνυμάτων και δίνουν έμφαση στη χρονική αλληλουχία των μηνυμάτων. Ένα διάγραμμα ακολουθίας περιέχει Χειριστές, Αντικείμενα και Μηνύματα που ανταλλάσσουν τα αντικείμενα. Τα διαγράμματα ακολουθίας

χρησιμοποιούνται για να περιγράψουν τον κύκλο ζωής των αντικειμένων. Τα διαγράμματα συνεργασίας και ακολουθίας καταγράφουν δυναμικές δομές αντικειμένων.



Εικόνα 2-10 : Διάγραμμα ακολουθίας (Sequence Diagram)

Η Εικόνα 2-10 παρουσιάζει συνοπτικά τα κυριότερα χαρακτηριστικά ενός διαγράμματος ακολουθίας, σε ένα ενδεικτικό παράδειγμα σχεδίασης της διαδικασίας κράτησης σε ένα σύστημα κρατήσεων.

2.4 Η γλώσσα Προγραμματισμού C++

Η γλώσσα προγραμματισμού C++[5] είναι μία ευρέως διαδεδομένη και ισχυρή δομημένη (structure), αντικειμενοστραφής (object oriented) γλώσσα που υποστηρίζει όλες τις σχετικές δηλώσεις και δομές του αντικειμενοστραφούς προγραμματισμού. Αποτελεί εξέλιξη της γλώσσας C (Πρότυπο, σελ. 84) και χαρακτηρίζεται, όπως η C, ως γλώσσα μεσαίου επιπέδου (Middle level language) διότι διαθέτει όλα τα χαρακτηριστικά μίας γλώσσας υψηλού επιπέδου (High level language), ενώ παράλληλα παρέχει τον έλεγχο και την ευελιξία της συμβολικής γλώσσας Assembly[6].

Η C++ με την πάροδο των χρόνων έχει πλέον καθιερωθεί ως γλώσσα αντικειμενοστραφούς προγραμματισμού και αναμένεται από τον αναγνώστη να είναι εξοικειωμένος με τον κώδικα της, προκειμένου να είναι σε θέση να κατανοήσει τόσο την περιγραφή των σχεδιαστικών προτύπων όσο και την ίδια την σχεδίαση και υλοποίηση του μεταγλωττιστή με την οποία πραγματεύεται η παρούσα εργασία. Σε περίπτωση που ο αναγνώστης δεν είναι εξοικειωμένος με την C++, συστήνεται πρώτα να διερευνηθεί και

[5] Το ποιο πρόσφατο πρότυπο της γλώσσας C++ είναι το ISO/IEC 14882:2011 και δημοσιεύθηκε τον 09-2011

[6] Η συμβολική γλώσσα Assembly είναι μια low level language η οποία συσχετίζεται άμεσα με το instruction set που υποστηρίζει ο επεξεργαστής του συστήματος. Με τα σύγχρονα εργαλεία ανάπτυξης λογισμικού είναι δυνατός ο συνδυασμός - συγγραφή κώδικα Assembly και κώδικα C++ για το ίδιο module ή project (Irvine, 2011)

μελετηθεί η σχετική με τον προγραμματισμό σε C++ βιβλιογραφία (ενδεικτικά: (Herbert, 1998), γρήγορος οδηγός (Soulie, 2007)).

Πέραν των τυπικών και συνηθισμένων εντολών της C++, ιδιαίτερη προσοχή πρέπει να δοθεί στην χρήση και κατανόηση των (αντικειμενοστραφών) εννοιών : object, class, abstract class, default constructor, destructor, constructor overloading, class/object pointer, operator overloading, static members, friendship, inheritance, polymorphism, virtual member, abstract class, καθώς και (class) templates, type casting. Για τις ανάγκες της υλοποίησης της εργασίας παρουσιάζονται επιλεκτικά και συνοπτικά μερικές βασικές ΟΟ ιδιότητες της C++.

2.4.1 Inheritance

Μία βασική ιδιότητα της ΟΟ γλώσσας C++ είναι η κληρονομικότητα (**Inheritance**), που επιτρέπει την δημιουργία μίας κλάσης η όποια προέρχεται από άλλη κλάση (ή κλάσεις), έτσι ώστε αυτόματα να περιλαμβάνει, επιπλέον των δικών της, της ιδιότητες και μεθόδους όλων των κλάσεων που κληρονομεί ιεραρχικά.

2.4.2 Polymorphism

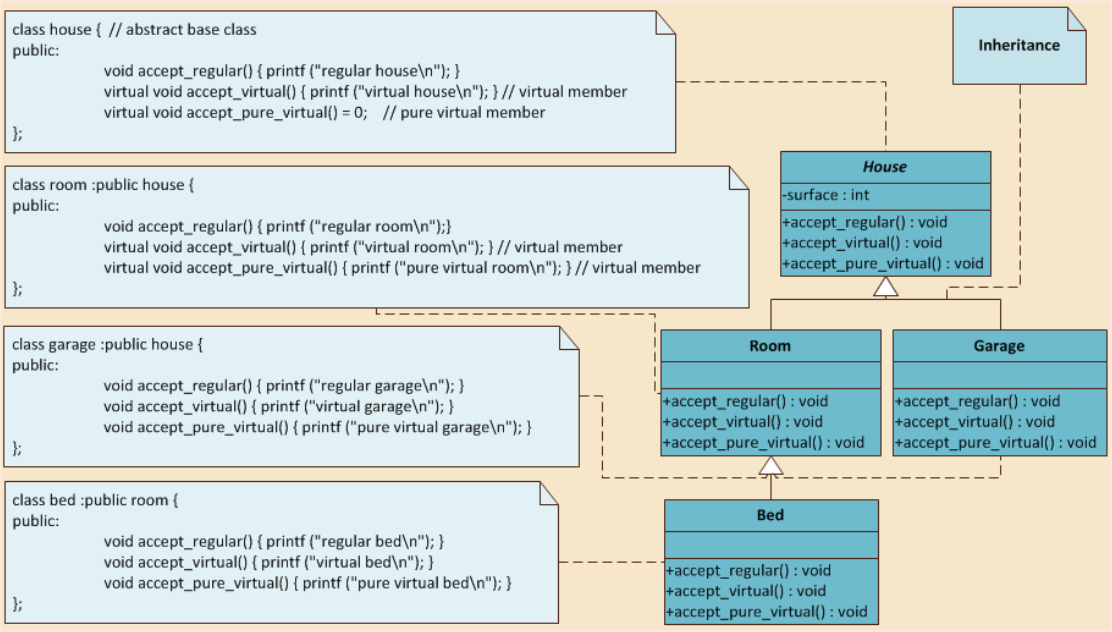
Άλλο ένα βασικό χαρακτηριστικό είναι ότι ο δείκτης (pointer) μιας προερχόμενης κλάσης, έχει συμβατό τύπο με τον δείκτη της γονικής κλάσης την οποία κληρονομεί. Πολυμορφισμός (**polymorphism**) είναι η χρησιμοποίηση αυτού του απλού αλλά ισχυρού και ευέλικτου χαρακτηριστικό της ΟΟ μεθοδολογίας.

Μια μέθοδος μιας κλάσης η οποία μπορεί να επναορισθεί (με ίδιο όνομα και παραμέτρους) σε παραγόμενες κλάσεις της, ονομάζεται εικονικό μέλος (**virtual member**), και σε συνδυασμό με τον πολυμορφισμό προσφέρει ευελιξία κατά την κλήση μεθόδων αντικειμένων παραγόμενων, δια μέσου της κληρονομικότητας, κλάσεων.

2.4.3 Abstract classes

Αν μια κλάση (για την οποία έχουν ορισθεί παραγόμενες κλάσεις) διαθέτει τουλάχιστον μία μέθοδο χωρίς υλοποίηση (**pure virtual member**), τότε η κλάση αυτή δεν μπορεί να παράγει αντικείμενα και ονομάζεται αφηρημένη κλάση (**abstract base class**).

Ένα συγκεντρωτικό παράδειγμα κληρονομικότητας κλάσεων παρουσιάζεται στην Εικόνα 2-11, ακολουθούμενη από ένα τμήμα κώδικα για την κατανόηση του χαρακτηριστικού του πολυμορφισμού δια μέσου των virtual και pure virtual members. Τα σχεδιαστικά πρότυπα που παρουσιάζονται και χρησιμοποιούνται στην υλοποίηση της παρούσας εργασίας βασίζονται στα συγκεκριμένα χαρακτηριστικά της ΟΟ μεθοδολογίας.



Εικόνα 2-11 : OO Inheritance, virtual & pure virtual member ιδιότητες

```
void main() {
        house *hr = new room();             // polymorphism
        house *hb = new bed();              // polymorphism
        house *hg = new garage();           // polymorphism

        room *rr = new room();
        room *rb = new bed();               // polymorphism

        bed *bb = new bed();

        garage *gg = new garage();

        hr->accept_regular();               // prints (regular house)
        hr->accept_virtual();               // prints (virtual room)
        hr->accept_pure_virtual();          // prints (pure virtual room)
        hb->accept_regular();               // prints (regular house)
        hb->accept_virtual();               // prints (virtual bed)
        hb->accept_pure_virtual();          // prints (pure virtual bed)
        hg->accept_regular();               // prints (regular house)
        hg->accept_virtual();               // prints (virtual garage)
        hg->accept_pure_virtual();          // prints (pure virtual garage)

        rr->accept_regular();               // prints (regular room)
        rr->accept_virtual();               // prints (virtual room)
        rr->accept_pure_virtual();          // prints (pure virtual room)
        rb->accept_regular();               // prints (regular room)
        rb->accept_virtual();               // prints (virtual bed)
        rb->accept_pure_virtual();          // prints (pure virtual bed)

        bb->accept_regular();               // prints (regular bed)
        bb->accept_virtual();               // prints (virtual bed)
        bb->accept_pure_virtual();          // prints (pure virtual bed)

        gg->accept_regular();               // prints (regular garage)
        gg->accept_virtual();               // prints (virtual garage)
        gg->accept_pure_virtual();          // prints (pure virtual garage)
}
```

3 Πρότυπα ποιότητας λογισμικού

Προκειμένου το αποτέλεσμα της σχεδίασης, ανάλυσης και υλοποίησης λογισμικού (Τεχνολογία Λογισμικού, Software Engineering) να ικανοποιεί τις τιθέμενες απαιτήσεις και παράλληλα να διαθέτει σταθερά και μετρήσιμα ποιοτικά χαρακτηριστικά, είναι αναγκαίος ο καθορισμός κατάλληλων προτύπων τα οποία θα περιγράφουν τρόπους, διαδικασίες και μεθόδους για την διασφάλιση καθώς και τον προσδιορισμό της Ποιότητας Λογισμικού. Για το λόγω αυτό έχουν αναπτυχθεί από οργανισμούς όπως ISO, IEEE, CMM, CMMI, αναλυτικά κείμενα προτύπων, υποδείξεων και οδηγιών, αναφερόμενα ως «**Πρότυπα Ποιότητας Λογισμικού**». Στη συνέχεια παρατίθενται δύο γενικοί ορισμοί των όρων ποιότητας και προτύπου.

Ως **ποιότητα** (quality) ορίζεται η συλλογή των χαρακτηριστικών σχεδιασμού, κατασκευής και συντήρησης, δια μέσω των οποίων το προϊόν κατά τη χρήση του θα εκπληρώσει τις προσδοκίες των πελατών (Feigenbaum, 1983).

Πρότυπο (standard) είναι η τεκμηριωμένη σύμβαση που περιέχει τεχνικές προδιαγραφές ή άλλα ακριβή κριτήρια που χρησιμοποιούνται ως κανόνες και κατευθυντήριες γραμμές για την εξασφάλιση της τυποποίησης των κατάλληλων υλικών, προϊόντων, διεργασιών και εξυπηρέτησης για τη διευκόλυνση της διεθνούς ανταλλαγής αγαθών και υπηρεσιών και της ανάπτυξης συνεργασίας στη σφαίρα των επιστημονικών, τεχνολογικών και οικονομικών ενεργειών. Γενικά ένα πρότυπο καθορίζει τις ιδιότητες του ποιοτικού λογισμικού (όχι τον τρόπο κατασκευής του) και είναι ανεξάρτητο της τεχνολογίας. Μέσω εμπειρίας και για όλες τις γλώσσες προγραμματισμού, μπορούν να καθαρισθούν εμπειρικοί κανόνες για την πιστοποίηση της ποιότητας λογισμικού.

Γενικά θα λέγαμε ότι η ουσία και στόχος της ποιότητας λογισμικού είναι η διάσπαση του λογισμικού σε μικρά κομμάτια, και αυτά σε μικρότερα, κ.ό.κ. μέχρι να προκύψει μία αυστηρά (κατά το δυνατό) ιεραρχική διάταξη. Στα «φύλλα» αυτής της διάσπασης αντιστοιχίζονται μετρικές ή υποκειμενικές «έρευνες», προσπαθώντας οι μέθοδοί να είναι όσο πιο αντικειμενικές γίνεται. Με την διάσπαση της αφηρημένης «ποιότητας» σε ποιο μετρήσιμα χαρακτηριστικά, είναι δυνατός ο «**σχεδιασμός της ποιότητας**» του λογισμικού που αναπτύσσεται.

- Είναι δυνατό να δοθεί έμφαση στις πρακτικές ανάπτυξης, ώστε το λογισμικό να αναπτύσσεται με σωστό τρόπο, ώστε να εξασφαλίζεται (a priori) η ποιότητα του προϊόντος.
- Είναι δυνατό να μετρηθεί η ποιότητα του λογισμικού που αναπτύχθηκε και να εξάγονται συμπεράσματα (posteriori) για τις μεθόδους ανάπτυξης που χρησιμοποιήθηκαν

3.1 Πρότυπα ποιότητα λογισμικού ISO

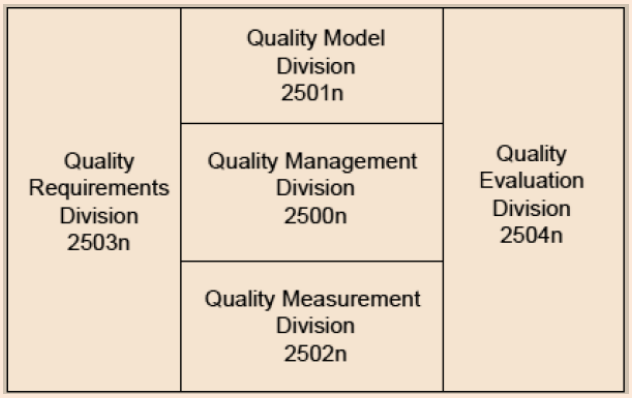
Αναφερόμενοι στα πρότυπα ποιότητας του ISO που σχετίζονται με την τεχνολογία λογισμικού παρατίθεται λίστα (Πίνακας 3-1) με τα σχετικά ισχύοντα πρότυπα. Για την εφαρμογή των προτύπων λογισμικού συνήθως εφαρμόζεται το γενικό πρότυπο ISO 9001 (πρότυπο διασφάλισης ποιότητας κατά τη σχεδίαση, ανάπτυξη, εγκατάσταση ή παροχή υπηρεσιών, πρόγραμμα ποιότητας σε όλες τις φάσεις) σε συνδυασμό με τις οδηγίες του

προτύπου ISO/IEC 90003:2004 (οδηγίες εφαρμογής του ISO:9001 σε λογισμικό) καθώς και η εφαρμογή εξειδικευμένων στην ποιότητα λογισμικού προτύπων της σειράς ISO/IEC 250xx.

Πίνακας 3-1 : Βασικά Πρότυπα Ποιότητας Λογισμικού του ISO

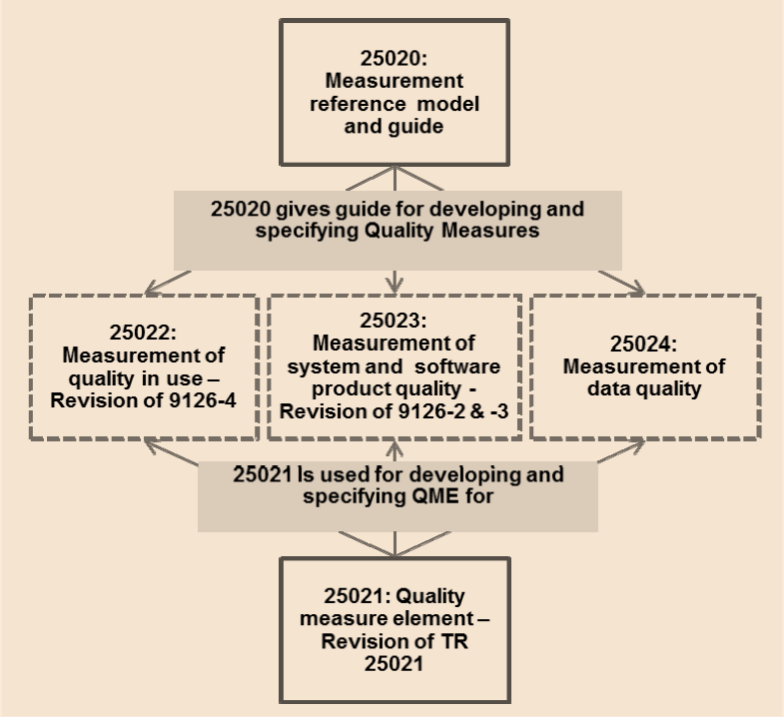
Πρότυπο	Περιγραφή
ISO 9001:2008 Replaces: ISO 9001:2000	**Quality management systems – Requirements**
ISO 9004:2009	Managing for the sustained success of an organization -- A quality management approach
ISO/IEC 90003:2004	**Software engineering -- Guidelines for the application of ISO 9001:2000 to computer software**
ISO/IEC 25000:2014	Systems and software engineering -- Systems and software Quality Requirements and Evaluation (SQuaRE) -- Guide to SQuaRE
ISO/IEC 25001:2014 Replaces: ISO/IEC 14598-2:2000	Systems and software engineering -- Systems and software Quality Requirements and Evaluation (SQuaRE) -- Planning and management
ISO/IEC 14598-6:2001	Software engineering -- Product evaluation -- Part 6: Documentation of evaluation modules This standard has been reviewed and then confirmed in 2008
ISO/IEC 25010:2011 Replaces: ISO/IEC 9126:1991 ISO/IEC 9126-1:2001	**Systems and software engineering -- Systems and software Quality Requirements and Evaluation (SQuaRE) -- System and software quality models**
ISO/IEC TR 9126-2:2003	Software engineering -- Product quality -- Part 2: External metrics
ISO/IEC TR 9126-3:2003	**Software engineering -- Product quality -- Part 3: Internal metrics**
ISO/IEC TR 9126-4:2004	Software engineering -- Product quality -- Part 4: Quality in use metrics
ISO/IEC 25012:2008	Software engineering -- Software product Quality Requirements and Evaluation (SQuaRE) -- Data quality model
ISO/IEC 25020:2007	**Software engineering -- Software product Quality Requirements and Evaluation (SQuaRE) -- Measurement reference model and guide**
ISO/IEC 25021:2012	**Systems and software engineering -- Systems and software Quality Requirements and Evaluation (SQuaRE) -- Quality measure elements**
ISO/IEC 25030:2007	Software engineering -- Software product Quality Requirements and Evaluation (SQuaRE) -- Quality requirements
ISO/IEC 25040:2011 Replaces: ISO/IEC 14598-1:1999	Systems and software engineering -- Systems and software Quality Requirements and Evaluation (SQuaRE) -- Evaluation process
ISO/IEC 25041:2012 Replaces: ISO/IEC 14598-3:2000 ISO/IEC 14598-4:1999 ISO/IEC 14598-5:1998	Systems and software engineering -- Systems and software Quality Requirements and Evaluation (SQuaRE) -- Evaluation guide for developers, acquirers and independent evaluators
ISO/IEC TR 25060:2010	Systems and software engineering -- Systems and software product Quality Requirements and Evaluation (SQuaRE) -- Common Industry Format (CIF) for usability: General framework for usability-related information

Η οικογένεια των παλαιότερων προτύπων ISO/IEC 14598 και ISO/IEC 9126 έχει σχεδόν αντικατασταθεί από την οικογένεια προτύπων ISO/IEC 25000 με την υποδιαίρεση που παρουσιάζεται στην Εικόνα 3-1.



Εικόνα 3-1: Τμηματοποίηση οικογένειας προτύπων ISO/IEC 25000

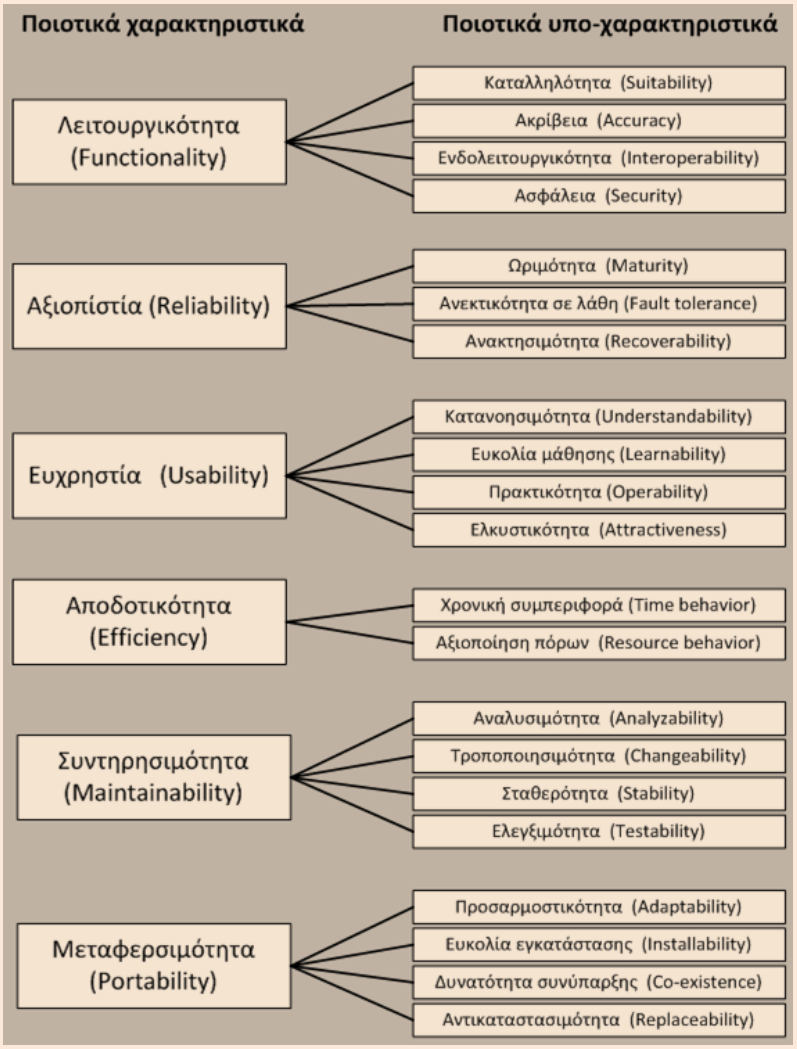
Οι σειρές των προτύπων ISO/IEC 2500n, ISO/IEC 2501n, ISO/IEC 2503n, και ISO/IEC 2504n, παρουσιάζουν το σχεδιασμό και τη χρήση των μετρήσεων ποιότητας λογισμικού που ορίζονται στη σειρά προτύπων ISO/IEC 2502n για την μέτρηση της ποιότητας των προϊόντων λογισμικού (Εικόνα 3-2). Επικεντρωνόμαστε στην σειρά προτύπων ISO/IEC 25020 και ISO/IEC 25021 που έχουν ως σκοπό τον καθορισμό ή/και σχεδιασμό ενός αρχικού συνόλου Στοιχείων Ποιοτικών Μετρήσεων (**Quality Measure Elements**, **QME**) προκειμένου να χρησιμοποιηθούν καθ' όλη τη διάρκεια του κύκλου ζωής του λογισμικού για τις ανάγκες των απαιτήσεων και αξιολόγησης της ποιότητας λογισμικού και συστημάτων (Systems and Software Quality Requirements and Evaluation, **SQuaRE**).



Εικόνα 3-2: Τμηματοποίηση οικογένειας προτύπων ISO/IEC 2502n

Ειδικότερα το πρότυπο ISO/IEC TR 9126-3:2003 (έχει περιληφθεί στην σειρά ISO/IEC 2502n), παρέχει ένα σύνολο προτεινόμενων ποιοτικών (εσωτερικών) μετρικών λογισμικού. Ωστόσο αναφέρει ρητά τη δυνατότητα του χρήστη να τροποποιήσει μια ορισμένη μετρική ή ακόμη και να δημιουργήσει μια νέα, με την προϋπόθεση να καθορίσει το συσχετισμό της με

το εφαρμοζόμενο πρότυπο μοντέλου ποιότητας (π.χ. ISO/IEC 9126-1:2001 όπως αναθεωρήθηκε από το ISO/IEC 25010:2011). Οι μετρικές του προτύπου ISO/IEC TR 9126-3:2003 διαχωρίζονται σε κατηγορίες ποιοτικών χαρακτηριστικών (Εικόνα 3-3).

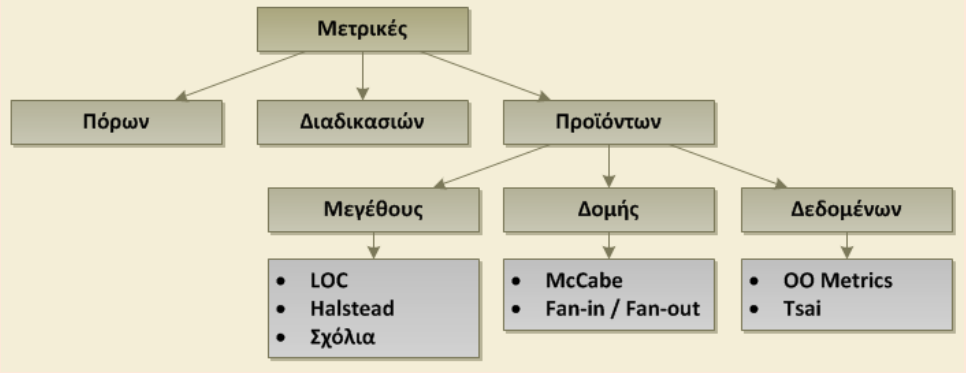


Εικόνα 3-3: Ποιοτικά χαρακτηριστικά ISO 9126 / ISO 25000

Εκτός των ISO προτύπων, υπάρχουν πολλές ομαδοποιήσεις / διασπάσεις των ποιοτικών χαρακτηριστικών π.χ. Boehm, IEEE Standards, GRQM (Goal-Rule-Question-Metrics), Dromey, FCM (Factor-Criteria-Metrics) του McCall, κ.λ.π., χωρίς να υπάρχει απαραίτητα κάποιο κοινή πρακτική μεταξύ τους.

3.2 Μετρικές λογισμικού

Ως **μέτρηση** μπορούμε να περιγράψουμε τη διαδικασία με την οποία αριθμοί ή σύμβολα αντιστοιχίζονται σε ιδιότητες (attributes) οντοτήτων του πραγματικού κόσμου με τέτοιο τρόπο, ώστε να τις περιγράφουν σύμφωνα με ξεκάθαρα καθορισμένους κανόνες. Συνεπώς **μετρική** είναι η ανάθεση αριθμού ή συμβόλου σε μία οντότητα που χαρακτηρίζει μία συγκεκριμένη ιδιότητά της (attribute). Τα χαρακτηριστικά των ιδιοτήτων αυτών θα πρέπει είναι ίδια για κάθε πρόγραμμα και γλώσσα προγραμματισμού, θα εκφράζονται με μαθηματικό τρόπο και θα υπολογίζονται αυτόματα.



Εικόνα 3-4 : Είδη και κατηγορίες Μετρικών

Οι μετρικές κατατάσσονται γενικά σε είδη και κατηγορίες (Εικόνα 3-4) ανάλογα με το είδος των οντοτήτων που καλούνται να χαρακτηρίσουν. Επίσης υπάρχει και ο γενικός διαχωρισμός των «Εσωτερικών» και «Εξωτερικών» μετρικών. Οι **εξωτερικές μετρικές** (external metrics) περιγράφουν τις ιδιότητες του λογισμικού κυρίως από την συμπεριφορά του ως συνολικού προϊόντος κατά την λειτουργία του (black-box αντιμετώπιση). Οι **εσωτερικές μετρικές** (internal metrics) περιγράφουν τις ιδιότητες του λογισμικού κυρίως από την πλευρά της ανάλυσης, σχεδίασης και ανάπτυξης (white-box αντιμετώπιση). Στις επόμενες ενότητες παρουσιάζονται επιλεγμένες εσωτερικές μετρικές διαφόρων προτύπων, προκειμένου να εντοπισθούν οι μετρικές που επηρεάζονται και αποτυπώνουν τα ποιοτικά χαρακτηριστικά του λογισμικού τα οποία σχετίζονται με την εφαρμογή σχεδιαστικών προτύπων (design patterns) κατά την ανάπτυξη και χρήση δομών δεδομένων σε ΟΟ λογισμικό.

3.2.1 Μετρικές ISO 9126 / ISO 25000

Η κατηγορία ποιοτικών χαρακτηριστικών «Maintainability» των προτύπων ποιότητας ISO 9126 / ISO 25000 (Εικόνα 3-3), αναλύεται σε υποχαρακτηριστικά ως εξής:

Maintainability (Συντηρησιμότητα): είναι εύκολο να συντηρηθεί

- Analyzability (Αναλυσιμότητα): εύκολο να εντοπιστούν ανεπάρκειες ή λάθη
- Changeability (Ευκολία Υλοποίησης Αλλαγών)
- Stability (Σταθερότητα): ελαχιστοποιούνται οι παρενέργειες των τροποποιήσεων
- Testability (Ελεγξιμότητα): είναι εύκολο να ελεγχθεί η αξιοπιστία μετά από αλλαγή

Τμήμα των μετρικών που προτείνονται από το πρότυπο ISO/IEC TR 9126-3:2003 σχετικά με την κατηγορία «Maintainability» παρουσιάζεται παρακάτω (Πίνακας 3-2). Η υποκατηγορία «Changeability» περιέχει την μετρική με όνομα «Change recordability» η οποία απλά καταγράφει το σχολιασμό στα τμήματα του κώδικα στα οποία και επέρχονται αλλαγές και όχι το βαθμό δυσκολίας των ενδεχόμενων αλλαγών από τον προγραμματιστή ή το βαθμό πολυπλοκότητας του κώδικα. Ωστόσο η μετρική «Modification impact localization» καταγράφει τον αριθμό των επηρεαζόμενων μεταβλητών και αποτυπώνει μερικώς το βαθμό πολυπλοκότητας του κώδικα, όπως και η μετρική «Change impact».

Πίνακας 3-2: Μετρικές Συντηρησιμότητας Λογισμικού (ISO 9126 / ISO 25000)

Όνομα	Σκοπός	Μέθοδος	Μέτρηση
Analysability Activity recording	Πόσο πλήρης είναι η καταγραφή της κατάστασης του συστήματος	Καταγραφή του αριθμού των καταγεγραμμένων τμημάτων και σύγκριση με τον αριθμό των τμημάτων που απαιτούν καταγραφή	X=A/B A=Αριθμός υλοποιημένων καταγραφών αναθεώρησης B=Αριθμός καταγραφών βάση προδιαγραφών
Analysability Readiness of diagnostic function	Πόσο πλήρης είναι οι παρεχόμενες διαγνωστικές λειτουργίες	Καταγραφή του αριθμού των διαγνωστικών λειτουργιών και σύγκριση με τον αριθμό των διαγνωστικών λειτουργιών που απαιτούνται βάση προδιαγραφών	X=A/B Αριθμός υλοποιημένων διαγνωστικών λειτουργιών αναθεώρησης B=Αριθμός διαγνωστικών λειτουργιών βάση προδιαγραφών
Changeability Change recordability	Οι αλλαγές στις προδιαγραφές και στα επιμέρους τμήματα του προγράμματος, καταγράφονται επαρκώς στον κώδικα με σχόλια	Record ratio of module change information	X=A/B A=Αριθμός functions/modules με σχόλια τροποποίησης κατά την αναθεώρηση B=Συνολικός αριθμός functions/modules που τροποποιήθηκαν από τον αρχικό κώδικα
Stability Change impact	Ποια είναι η συχνότητα εμφάνισης δυσμενών επιπτώσεων μετά την τροποποίηση	Καταγραφή αριθμού δυσμενών επιπτώσεων μετά από αλλαγή και σύγκριση με τον αριθμό των τροποποιήσεων που υλοποιήθηκαν	X=1-A/B A=Αριθμός εντοπισμένων δυσμενών επιπτώσεων μετά από αλλαγές B=Αριθμός πραγματοποιούμενων αλλαγών
Stability Modification impact localization	Πόσο σημαντικός είναι ο αντίκτυπος των αλλαγών στο λογισμικό	Καταγραφή αριθμού επηρεαζόμενων μεταβλητών από την αλλαγή και σύγκριση με το συνολικό αριθμό μεταβλητών του λογισμικού	X=A/B A=Αριθμός επηρεαζόμενων μεταβλητών από την αλλαγή κατά την αναθεώρηση B=Συνολικός αριθμός μεταβλητών

Γενικά οι προτεινόμενες μετρικές λογισμικού των προτύπων ποιότητας ISO 9126 / ISO 25000 χαρακτηρίζονται από μια απλότητα και δεν φαίνεται να εντοπίζουν τα ποιοτικά χαρακτηριστικά και πλεονεκτήματα του OO programming (inheritance, polymorphism, κλπ).

3.2.2 Μετρικές Halstead

Ο Halstead παρουσίασε το 1977 μία θεωρία που την ονόμασε «Επιστήμη Λογισμικού» και οι μετρικές που πρότεινε ονομάζονται «μετρικές της επιστήμης λογισμικού» (Halstead, 1977). Σύμφωνα με την Επιστήμη Λογισμικού του Halstead, κάθε πρόγραμμα θεωρείται ως ένα σύνολο από λεκτικά σύμβολα (tokens), που είναι οι τελεστές (operators, n_1) και τα έντελα (operands, n_2). Όλες οι μετρικές του Halstead βασίζονται πάνω σε αυτούς τους δύο ορισμούς και παρουσιάζονται στον Πίνακας 3-3.

Πίνακας 3-3: Μετρικές (Επιστήμης Λογισμικού) Halstead

Όνομα	Υπολογισμός
αριθμός των διακριτών τελεστών που εμφανίζονται στο πρόγραμμα	n_1
αριθμός των διακριτών εντέλων που εμφανίζονται στο πρόγραμμα	n_2
αριθμός των συνολικών εμφανίσεων τελεστών στο πρόγραμμα	N_1
αριθμός των συνολικών εμφανίσεων εντέλων στο πρόγραμμα	N_2
Λεξιλόγιο προγράμματος	$n = n_1 + n_2$

Όνομα	Υπολογισμός
Μήκος προγράμματος	$N = N_1 + N_2$ $N_{est} = n_1 \cdot \log_2 n_1 + n_2 \cdot \log_2 n_2$
Όγκος προγράμματος	$V = N \cdot \log_2 n$
Επίπεδο του προγράμματος	$L = V^* / V$ $L_{est} = (2 \cdot n_2) / (n_1 \cdot N_2)$
Επίπεδο γλώσσας στο πρόγραμμα	$\lambda = L^2 \cdot V = ((2 \cdot n_2) / (n_1 \cdot N_2))^2 \cdot N \cdot \log_2 n$
Δυσκολία προγράμματος	$D = 1 / L = (n_1 \cdot N_2) / (2 \cdot n_2)$
Προσπάθεια υλοποίησης προγράμματος	$E = D \cdot V = V/L = ((n_1 \cdot N_2) / (2 \cdot n_2)) \cdot N \cdot \log_2 n$
Χρόνος υλοποίησης προγράμματος	$T = E / S = (n_1 \cdot N_2 \cdot N \cdot \log_2 n) / (2 \cdot n_2 \cdot S)$, S=18
Εκτίμηση αριθμού λαθών στο πρόγραμμα	$B = E^{2/3} / E_0 = \frac{\left(\frac{n_1 \cdot N_2}{2 \cdot n_2} N \cdot log_2 n\right)^{\frac{2}{3}}}{E_0}$, $E_0 \in [3000, 3200]$

Σήμερα υπάρχουν αρκετά εργαλεία που δέχονται ως είσοδο κώδικα σε κάποια γλώσσα προγραμματισμού, επιστρέφουν αυτόματα αποτελέσματα για αυτές τις μετρικές. Εδώ ωστόσο θα πρέπει να αναφερθεί ότι οι μετρικές του Halstead έχουν νόημα και υπολογίζονται για κάθε τμήμα και για κάθε ρουτίνα του κώδικα χωριστά. Είναι μετρικές μεγέθους (ή ποσοτικές μετρικές) και μετρούν αριθμήσιμα στοιχεία του λογισμικού σχετιζόμενα με το μέγεθος του πηγαίου κώδικα και δεν εντοπίζουν τα ποιοτικά χαρακτηριστικά και πλεονεκτήματα του OO programming.

3.2.3 Μετρικές McCabe

Οι μετρικές McCabe (McCabe Software, 2014) έχουν καθιερωθεί από την ομώνυμη εταιρεία «McCabe Software», δια μέσου των διαδεδομένων εργαλείων αυτόματης μέτρησης ποιοτικών χαρακτηριστικών λογισμικού που αναπτύσσει και διαθέτει. Παρακάτω (Πίνακας 3-4) παρουσιάζονται οι μετρικές McCabe με μία σύντομη περιγραφή. Ωστόσο πρέπει να σημειωθεί ότι πολλές μετρικές όπως η «Cyclomatic Complexity» είναι κοινές μετρικές και περιλαμβάνονται σχεδόν σε όλα τα λογισμικά που πραγματοποιούν ποιοτικές μετρήσεις λογισμικού. Κάθε εργαλείο βέβαια μπορεί να παρέχει εναλλακτικούς τρόπους εφαρμογής, συνδυαστικών υπολογισμών ή γραφικών αναπαραστάσεων.

Πίνακας 3-4: Μετρικές McCabe

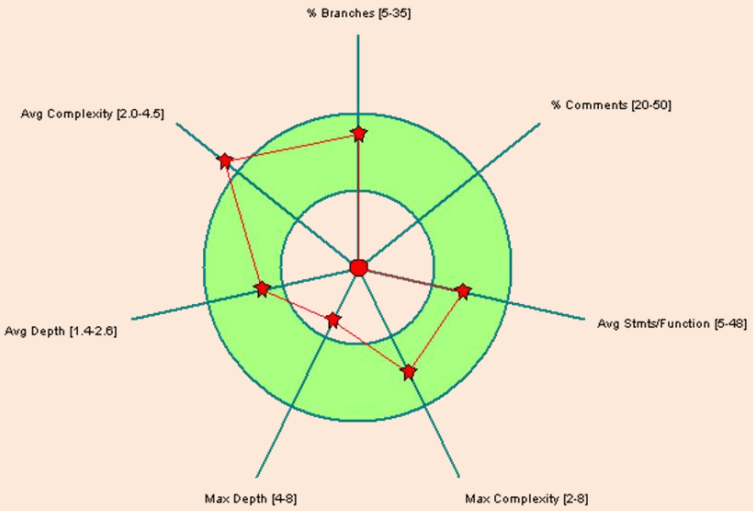
Ομάδα	Όνομα μετρικής	Περιγραφή
Software Metrics	Cyclomatic Complexity Metric (v(G))	Η κυκλωματική πολυπλοκότητα είναι μια μέτρηση της πολυπλοκότητας των δομών αποφάσεων ενός τμήματος κώδικα. Είναι ο αριθμός των γραμμικά ανεξάρτητων μονοπατιών και ως εκ τούτου ο ελάχιστος αριθμός μονοπατιών που πρέπει να δοκιμασθούν.
	Actual Complexity Metric (ac)	Ο αριθμός των ανεξάρτητων μονοπατιών που διαπερνώνται κατά τη δοκιμή
	Module Design Complexity Metric (iv(G))	είναι η πολυπλοκότητα του design-reduced module και αντανακλά την πολυπλοκότητα των προτύπων κλήσεων μιας μονάδας λογισμικού προς τις άμεσες υποδεέστερες μονάδες της
	Essential Complexity Metric (ev(G))	είναι η μέτρηση του βαθμού στον οποίο ένα module περιέχει μη δομημένες κατασκευές. Μετρά το βαθμό της δομής και της ποιότητας του κώδικας. Χρησιμοποιείται για την πρόβλεψη της προσπάθειας συντήρησης και ως βοήθειας στη διαδικασίας ενοποίησης του λογισμικού

Ομάδα	Όνομα μετρικής	Περιγραφή
	Pathological Complexity Metric (pv(G))	μέτρηση του βαθμού στον οποίο ένα module περιέχει υπερβολικά μη δομημένες κατασκευές
	Design Complexity Metric (S0)	μετρά το σύνολο των αλληλεπιδράσεων μεταξύ των modules σε ένα σύστημα
	Integration Complexity Metric (S1)	μετρά το σύνολο των ολοκληρωμένων δοκιμών που απαιτούνται για την προστασία από σφάλματα
	Object Integration Complexity Metric (OS1)	υπολογίζει τον αριθμό των δοκιμών που είναι απαραίτητες για την πλήρη ενοποίηση ενός αντικειμένου ή κλάσης σε ένα ΟΟ σύστημα
	Global Data Complexity Metric (gdv(G))	υπολογίζει την κυκλωματική πολυπλοκότητα της δομής μιας μονάδας λογισμικού καθώς αναφέρεται / συσχετίζεται με global δεδομένα ή με δεδομένα παραμέτρων
OO Software Metrics ENCAPSULATION	Percent Public Data (PCTPUB)	το ποσοστό των ορισμένων ως PUBLIC και PROTECTED δεδομένων εντός μίας κλάσης
	Access to Public Data (PUBDATA)	προσδιορίζει τον αριθμό των προσβάσεων σε PUBLIC και PROTECTED δεδομένα
POLYMORPHISM	Percent of Unoverloaded Calls (PCTCALL)	ο αριθμός των non-overloaded κλήσεων σε ένα σύστημα
	Number of Roots (ROOTCNT)	ο αριθμός του συνόλου των ιεραρχημένων κλάσεων ριζών σε ένα πρόγραμμα
	Fan-in (FANIN)	ο αριθμός των κλάσεων από τις οποίες μια κλάση προέρχεται
QUALITY	Maximum v(G) (MAXV)	η μέγιστη τιμή κυκλωματικής πολυπλοκότητας για κάθε μέθοδο μίας κλάσης
	Maximum ev(G) (MAXEV)	η μέγιστη essential complexity τιμή για κάθε μέθοδο μίας κλάσης
	Hierarchy Quality(QUAL)	ο αριθμός των κλάσεων σε ένα σύστημα που εξαρτώνται από τους απογόνους τους
OTHER	Depth (DEPTH)	Προσδιορίζει το επίπεδο μιας κλάσης εντός της ιεραρχίας των κλάσεων
	Lack of Cohesion of Methods (LOCM)	μέτρηση της αλληλεπίδρασης μεταξύ μεθόδων και δεδομένων μίας κλάσης
	Number of Children (NOC)	ο αριθμός των κλάσεων που προέρχονται απευθείας από μια συγκεκριμένη κλάση
	Response For a Class (RFC)	ο αριθμός των μεθόδων που υλοποιούνται σε μία κλάση προσαυξημένος με τον αριθμό των προσιτών μεθόδων σε ένα αντικείμενο του ιδίου τύπου κλάσης λόγω της ιεραρχίας (κληρονομικότητας)
	Weighted Methods Per Class (WMC)	ο αριθμός των μεθόδων που υλοποιούνται σε μία κλάση

Οι μετρικές McCabe περιλαμβάνουν και μετρικές που εντοπίζουν τα ποιοτικά χαρακτηριστικά και πλεονεκτήματα του ΟΟ programming (inheritance, polymorphism, κλπ). Μάλιστα διατίθεται και ένα τεχνικό κείμενο (Watson & McCabe, 1996) που περιγράφει μια δομημένη μεθοδολογία δοκιμών λογισμικού, παρουσιάζει σχετικές μετρικές πολυπλοκότητας λογισμικού καθώς και παρουσιάζει θεμελιώδεις τεχνικές δομικού ελέγχου για ΟΟ συστήματα.

3.2.4 Εφαρμογή και Αποτελέσματα μετρικών

Οι μετρικές ποιότητας λογισμικού προκειμένου να έχουν πρακτική και συστηματική εφαρμογή, σχεδιάζονται έτσι ώστε να είναι δυνατή η αυτοματοποιημένη εφαρμογή τους από έτοιμα εργαλεία (προγράμματα) ομαδικά σε πολλά αρχεία πηγαίου κώδικα. Παραδείγματα ελεύθερου λογισμικού αξιολόγησης ποιότητας κώδικα είναι τα Source Monitor[7], RMS.



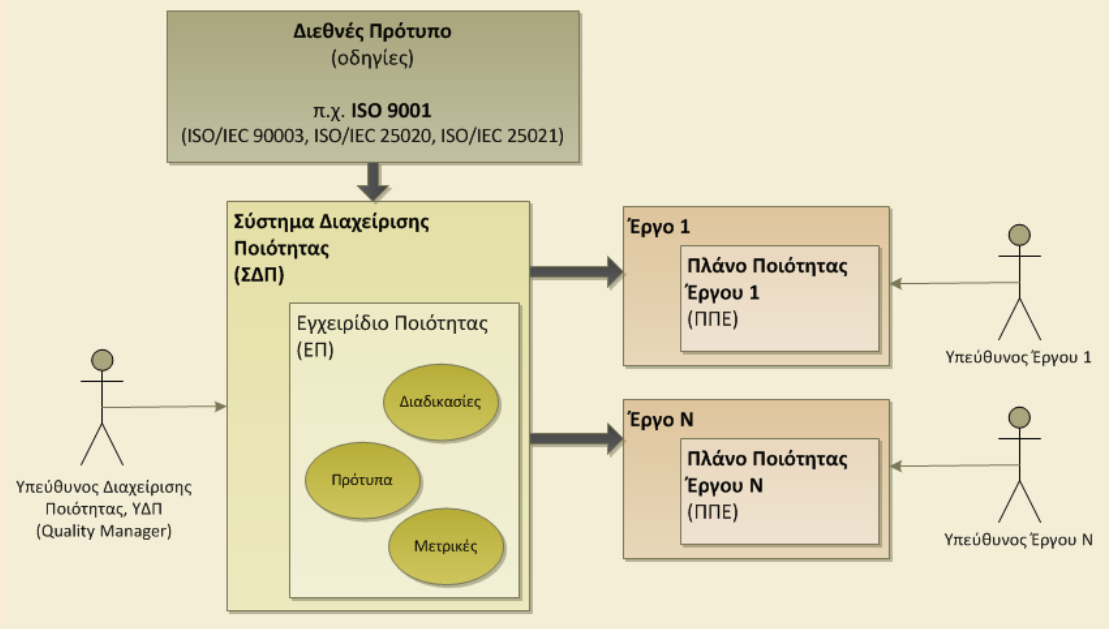
Εικόνα 3-5 : Παράδειγμα γράφου Kiviat

Τα αποτελέσματα των ελέγχων και μετρικών, εξάγονται συνήθως σε δομημένα κείμενα με στατιστικές ομαδοποιημένες μετρήσεις ανά module, συνάρτηση, κλάση, μέθοδο, ή ομάδες αρχείων projects, κλπ. Για την άμεση εξαγωγή συμπερασμάτων και την οπτικοποίηση των αποτελεσμάτων εξάγονται και γραφικές παραστάσεις όπως ο γράφος **Kiviat** (Εικόνα 3-5) που απεικονίζει πολλές μετρικές καθώς και την περιοχή των αποδεκτών ορίων που τίθενται από το εφαρμοζόμενο πρότυπο ή/και προκύπτουν από την εμπειρία.

3.3 Οργάνωση Προγράμματος Ποιότητας Λογισμικού

Όλα τα παραπάνω πρότυπα, διαδικασίες, οδηγίες, μετρικές προκειμένου να εφαρμοσθούν στον έλεγχο και διασφάλιση ποιότητας λογισμικού σε μεγάλη κλίμακα, οργανώνονται (Εικόνα 3-6) μέσω ενός κεφαλικού προτύπου ποιότητας (π.χ. ISO 9001, ISO/IEC 90003), και συνδυάζονται με άλλα επιμέρους πρότυπα και οδηγίες.

[7] Σύνδεσμος εφαρμογής : http://sourcemonitor.software.informer.com/3.2/



Εικόνα 3-6: Οργάνωση Προγράμματος Ποιότητας Λογισμικού

Το αποτέλεσμα είναι η δημιουργία του Συστήματος Διαχείρισης Ποιότητα (ΣΔΠ), το οποίο περιλαμβάνει ένα Εγχειρίδιο Ποιότητα (ΕΠ) βασισμένο σε προσαρμοσμένες αναλυτικές και σαφής διαδικασίες, πρότυπα και μετρικές. Με βάση τα τελευταία για κάθε έργο (ανάπτυξη συστήματος λογισμικού) συντάσσεται το Πλάνο Ποιότητα Έργου (ΠΠΕ).

Οι αποδεκτές τιμές ή τα αποδεκτά όρια για κάθε μετρική καθορίζονται από την πολιτική ποιότητας που ορίζεται κατά την σχεδίαση του εγχειριδίου ποιότητας (ΕΠ) και εξειδικεύεται με το πλάνο ποιότητας (ΠΠΕ) του εκάστοτε έργου λογισμικού λαμβάνοντας υπόψη τις προδιαγραφές του. Η πολιτική ποιότητας είναι μια δέσμευση της διοίκησης (του κατασκευαστή λογισμικού) δια μέσου μετρήσιμων δεικτών (μετρικών) σχετικά με την αναμενόμενη ή/και λαμβανόμενη ποιότητα του προϊόντος (λογισμικού).

3.4 Επιπτώσεις ΟΟ Σχεδιαστικών Προτύπων στην Ποιότητα Λογισμικού

Προκειμένου να διαπιστωθούν και καταγραφούν οι επιπτώσεις της εφαρμογής των ΟΟ σχεδιαστικών προτύπων (design patterns), με όσο το δυνατό ποιο μετρήσιμο τρόπο, και στα πλαίσια της παρούσας εργασίας, επιχειρείται η συσχέτιση των αναμενόμενων (θετικών) αποτελεσμάτων από την εφαρμογή τους, με τις μετρικές των σχετικών προτύπων ποιότητας λογισμικού. Δηλαδή ο προσδιορισμός αυτών των μετρικών που αποτυπώνουν τα ποιοτικά χαρακτηριστικά του λογισμικού που σχετίζονται με την εφαρμογή των ΟΟ design patterns της εργασίας καθώς και κατά το πόσο αυτό είναι δυνατό.

Γενικά η εφαρμογή σχεδιαστικών προτύπων αναμένεται να επηρεάσει εμμέσως τα ποιοτικά χαρακτηριστικά του λογισμικού και άρα και τις αντίστοιχες μετρικές στο βαθμό που αυτές τις εντοπίζουν. Οι περισσότερες εσωτερικές μετρικές που παρουσιάσθηκαν είναι γενικού σκοπού και επικεντρώνονται κυρίως στην πολυπλοκότητα της υλοποίησης της συγκεκριμένης λύσης, δηλαδή στον κώδικα σε επίπεδο μονάδας (module) κώδικα ή/και ανά διαδικασία. Τα ποιοτικά οφέλη που αναμένουμε από την εφαρμογή των ΟΟ design patterns έχουν να κάνουν κυρίως με την σωστή ΟΟ δομή και την ευκολία (ή δυσκολία) στις

προσθήκες και αλλαγές νέων λειτουργιών σε ΟΟ δομές, καθώς και την αποφυγή επανασχεδιασμών τους. Βέβαια σωστά σχεδιασμένες ΟΟ δομές αναμένεται να συμβάλουν προς το καλύτερο (ή έστω να δημιουργούν προϋποθέσεις), σχεδόν σε όλους τους ποιοτικούς δείκτες (μετρικές) του λογισμικού, ακόμα και σε επίπεδο module κώδικα. Οι μετρικές είναι σε μεγάλο βαθμό υποκειμενικές και δεν είναι εύκολο να αποτυπώσουν με σαφήνεια τα ζητούμενα ποιοτικά χαρακτηριστικά.

4 Σχεδιαστικά Πρότυπα (Design patterns)

4.1 Γενικά

Ο σχεδιασμός αντικειμενοστραφούς (**object oriented**) λογισμικού είναι από μόνος του αρκετά δύσκολος και ο σχεδιασμός επαναχρησιμοποιήσιμου (**reusable**) αντικειμενοστραφούς λογισμικού είναι ακόμα ποιο δύσκολος. Εμπεριέχει την εξεύρεση των κατάλληλων αντικειμένων, το διαχωρισμό του σε κλάσεις με τον κατάλληλο βαθμό διάκρισης καθώς και τον ορισμό της διπροσωπίας των κλάσεων με τις κατάλληλες συσχετίσεις και ιεραρχία μεταξύ τους. Ο σχεδιασμός πρέπει αφενός να εξυπηρετεί τη συγκεκριμένη λύση και ταυτόχρονα να είναι τόσο γενικός ώστε να μπορεί να εξυπηρετήσει μελλοντικά προβλήματα και απαιτήσεις. Επιπλέων θα πρέπει να ελαχιστοποιεί, κατά το δυνατό, τον επανασχεδιασμό του λογισμικού.

Γενικά είναι πολύ δύσκολο να επιτύχουμε από το μηδέν ένα καλό σχεδιασμό για ένα δύσκολο πρόβλημα και πιθανότατα, όπως έχει δείξει η εμπειρία, θα χρειαστούν αρκετές παρεμβάσεις και τροποποιήσεις μέχρι να καταλήξουμε. Στην προσπάθεια αυτή οι σχεδιαστές - προγραμματιστές έχουν αναπτύξει ή αναζητούν καλές σχεδιαστικές λύσεις, τις οποίες επαναχρησιμοποιούν και βελτιώνουν. Τέτοιες σχεδιαστικές λύσεις αποτελούν στην ουσία τον πυρήνα της εμπειρίας ενός σχεδιαστή - προγραμματιστή και καθώς εξελίσσονται γίνονται ποιο γενικές και επαναχρησιμοποιήσιμες για συγκεκριμένα προβλήματα ή ομάδες προβλημάτων. Αυτές οι σχεδιαστικές λύσεις κάνουν τον αντικειμενοστραφή προγραμματισμό ποιο ευέλικτο, ποιοτικό και επαναχρησιμοποιήσιμο, συμβάλλοντας στην επιτυχή σχεδίαση συστημάτων με βάση την υπάρχουσα εμπειρία. Ο εξοικειωμένος με τέτοια πρότυπα σχεδιαστής, μπορεί να τα εφαρμόσει ή και προσαρμόσει άμεσα σε νέα ή παρόμοια προβλήματα χωρίς να χρειαστεί να τα αναπτύξει εκ νέου. Τελικά οι δοκιμασμένες και επιτυχημένες σχεδιαστικές λύσεις που ανταποκρίνονται στην σχεδίαση πολλαπλών προβλημάτων, στην ουσία συλλαμβάνουν τον πυρήνα της λύσης του προβλήματος, και τείνουν να επαναχρησιμοποιούνται ως σχεδιαστικά πρότυπα (**design patterns**).

Υπάρχει σημαντική βιβλιογραφία στον τομέα των σχεδιαστικών προτύπων καθώς και πληθώρα προτύπων για ποικίλα προβλήματα. Ωστόσο για τις ανάγκες της εργασίας θα επικεντρωθούμε σε τέσσερα βασικά πρότυπα που αναλύονται εκτενώς από τους (Gamma, Helm, Johnson, & Vlisssides, 1994) στη σχετικά βιβλιογραφία και συνοπτικά παρουσιάζονται στις επόμενες ενότητες.

4.2 Περιγραφή σχεδιαστικού προτύπου

Τα σχεδιαστικά πρότυπα προορίζονται για να επιλύουν συγκεκριμένα προβλήματα ή ομάδες παρόμοιων προβλημάτων που τείνουν να επαναλαμβάνονται κατά την διαδικασία σχεδίασης λογισμικού. Συνεπώς η περιγραφή ενός προτύπου ξεκινά πρώτα από το είδος των προβλημάτων που αυτό εφαρμόζεται και επιλύει και κατόπιν από την ίδια τη σχεδιαστική λύση (κλάσεις, συσχετίσεις, συνεργασίες, ιδιότητες, μεθόδους, κλπ). Τα σχεδιαστικά πρότυπα που περιγράφονται και χρησιμοποιούνται στη συνέχεια είναι μοντελοποιημένες (αντικειμενοστραφείς) περιγραφές από συνεργαζόμενα αντικείμενα και κλάσεις, κατάλληλα παραμετροποιημένα έτσι ώστε να επιλύουν γενικά σχεδιαστικά προβλήματα (Gamma, Helm, Johnson, & Vlisssides, 1994).

Ως αντικειμενοστραφείς σχεδιαστικές περιγραφές – λύσεις, η υλοποίησή τους μπορεί να πραγματοποιηθεί από αντικειμενοστραφείς γλώσσες προγραμματισμού όπως η C++, Java, κλπ. Η εφαρμογή τους ενδέχεται να απαιτεί κατάλληλη προσαρμογή, ωστόσο αυτή θα πρέπει να είναι μικρής έκτασης, διαφορετικά θα πρέπει να εξετασθεί αν το πρότυπο που επιλέχθηκε είναι το καταλληλότερο για το συγκεκριμένο πρόβλημα.

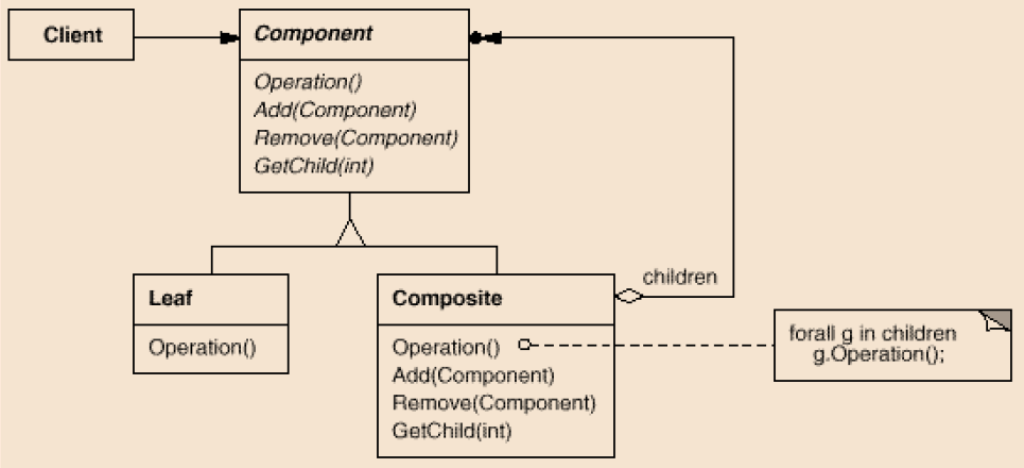
4.3 Σχεδιαστικά πρότυπα

4.3.1 Composite

Σκοπός του σχεδιαστικού προτύπου **Composite** είναι η σύνθεση αντικειμένων σε δομές δέντρων για την αναπαράσταση της part-whole ιεραρχίας. Μια τέτοια σύνθεση επιτρέπει τον χειρισμό ανεξάρτητων αντικειμένων και συνθέσεων με ενιαίο τρόπο.

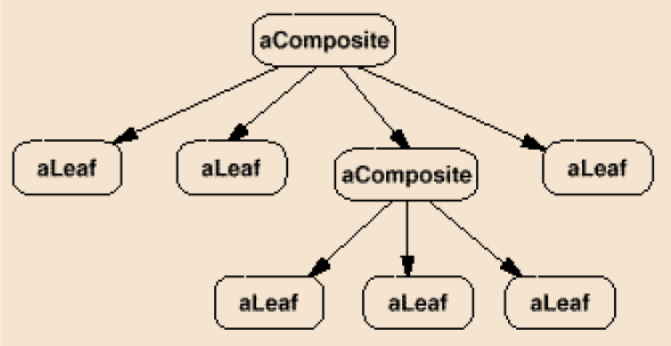
Αρκετά προβλήματα απαιτούν την ομαδοποίηση στοιχειωδών συστατικών (κλάσεις αντικειμένων) σε μεγαλύτερα συστατικά που με τη σειρά τους να ομαδοποιούνται σε ακόμη μεγαλύτερα συστατικά. Έστω ότι τα στοιχειώδη συστατικά περιγράφονται από μία κλάση (Leaf) και τα μεγαλύτερα συστατικά που τα ομαδοποιούν (περιέχουν) περιγράφονται από μία διαφορετική κλάση (Composite). Ο κώδικας που χρησιμοποίει αυτές τις κλάσεις πρέπει να συμπεριφέρεται στις στοιχειώδεις κλάσεις (Leaf) διαφορετικά από ότι στις συνθέσεις (Composite), ακόμα και εάν τις περισσότερες φορές ο χρήστης τις αντιλαμβάνεται και συμπεριφέρεται σε αυτές με ενιαίο τρόπο. Αναφερόμενοι στο τυπικό διάγραμμα κλάσεων στην Εικόνα 4-1, ο χρήστης μπορεί να καλεί μία λειτουργία-μέθοδο (Operation) σε ένα αντικείμενο ανεξάρτητα αν το αντικείμενο αυτό είναι στοιχειώδες ή σύνθεση. Το σχεδιαστικό πρότυπο Composite περιγράφει την χρήση αναδρομικών συνθέσεων κλάσεων αντικειμένων, έτσι ώστε ο χρήστης να μην χρειάζεται να κάνει τον διαχωρισμό μεταξύ του τύπου των αντικειμένων.

Η λύση είναι η δημιουργία μίας αφηρημένης (χωρίς υλοποίηση μεθόδων) κλάσης (Component), η οποία με χρήση της Κληρονομικότητας θα αναπαριστά και τους δύο τύπους (στοιχειώδη και συνθέσεις) αντικειμένων. Η υλοποίηση μίας κοινής λειτουργίας (Operation) για ένα αντικείμενο σύνθεσης θα καλεί την αντίστοιχη λειτουργία όλων των αντικειμένων που αυτή περιέχει.



Εικόνα 4-1 : Τυπικό διάγραμμα κλάσεων του σχεδιαστικού προτύπου Composite

Ο μηχανισμός της αντικειμενοστραφούς γλώσσας αναλαμβάνει να εκτελέσει την κατάλληλη υλοποίηση της λειτουργίας (Operation) ανάλογα με τον τύπο (κλάση) του αντικειμένου. Η Εικόνα 4-2 αναπαριστά μια τυπική σύνθεση στοιχειωδών αντικειμένων που ομαδοποιούνται σε αντικείμενα συνθέσεων. Η κλήση της μίας κοινής λειτουργίας (Operation) στο αντικείμενο της ρίζας του δέντρου, επιφέρει την αναδρομική κλήση (μέσω των Composites) της αντίστοιχης λειτουργίας για όλα τα στοιχειώδη αντικείμενα (Leafs).



Εικόνα 4-2 : Τυπικό διάγραμμα αντικειμένων του σχεδιαστικού προτύπου Composite

Γενικά χρησιμοποιούμε το σχεδιαστικό πρότυπο Composite όταν θέλουμε :

- ιεραρχική αναπαράσταση part-whole αντικειμένων
- ο καλών να αγνοεί τη διαφορά μεταξύ συνθέσεων αντικειμένων και στοιχειωδών αντικειμένων, ή αλλιώς να συμπεριφέρεται σε όλα τα αντικείμενα της δομής με ενιαίο τρόπο

4.3.2 Iterator

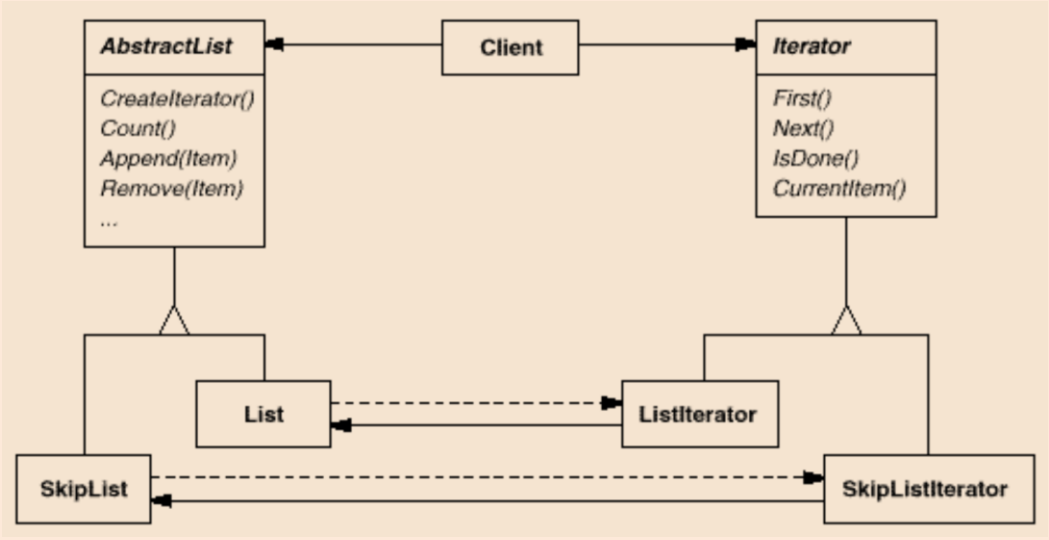
Σκοπός του σχεδιαστικού προτύπου **Iterator** είναι να παρέχει ένα τρόπο πρόσβασης στα στοιχεία μίας σύνθεσης αντικειμένων χωρίς να αποκαλύπτει στον χρήση τον τρόπο που αυτά αναπαρίστανται (οργανώνονται).

Σε πολλά προβλήματα παρουσιάζεται η ανάγκη προσπέλασης (διαπέρασης) των στοιχείων μίας σύνθεσης, όπως μίας λίστα αντικειμένων, χωρίς να μας ενδιαφέρει η εσωτερική της δομή. Επίσης ενδέχεται να θέλουμε διαφορετικούς τρόπους διαπέρασης των στοιχείων της, χωρίς να θέλουμε να επιβαρύνουμε την διεπαφή της σύνθεσης με επιπλέον λειτουργίες. Ακόμη μπορεί να είναι αναγκαία η ταυτόχρονη εκτέλεση περισσότερων από μία διαπεράσεις στην ίδια σύνθεση αντικειμένων.

Η λύση είναι η απόσπαση της λειτουργίας(ων) διαπέρασης και πρόσβασης εκτός του αντικειμένου της σύνθεσης και η τοποθέτησή τους σε ένα ξεχωριστό αντικείμενο διαπέρασης (iterator). Η Iterator κλάση ορίζει μια διεπαφή (σύνολο λειτουργιών) για την πρόσβαση των στοιχείων της σύνθεσης. Ένα Iterator αντικείμενο έχει την ευθύνη να γνωρίζει το τρέχον αντικείμενο και αρά και αυτά που έχουν ήδη διαπερασθεί. Η διεπαφή του Iterator πρέπει να περιέχει κατ ελάχιστο λειτουργίες για την εκκίνηση της διαπέρασης (First), την προώθηση στο επόμενο στοιχείο (Next), έλεγχο ολοκλήρωσης της διαπέρασης (IsDone) καθώς και λειτουργία πρόσβασης (CurrentItem) που να επιστρέφει το τρέχον στοιχείο της σύνθεσης. Κατά την δημιουργία ενός αντικειμένου Iterator θα πρέπει να το εφοδιάσουμε (μέσω παραμέτρου) με μια αναφορά στην σύνθεση της οποίας τα αντικείμενα πρόκειται να διαπεράσει. Ο διαχωρισμός του μηχανισμού διαπέρασης από τη

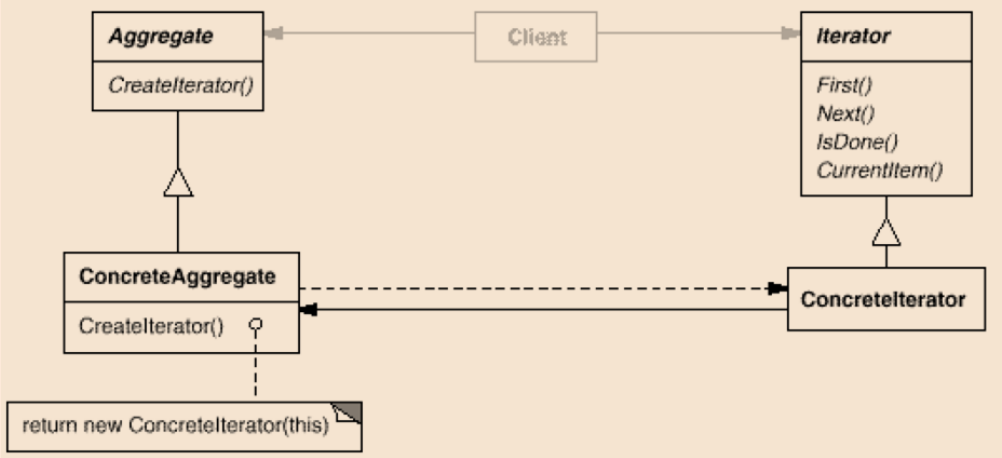
σύνθεση, μας επιτρέπει να δημιουργούμε Iterators (κλάσεις) για διαφορετικού τύπου διαπεράσεις, χωρίς να τις απαριθμούμε στη διεπαφή της σύνθεσης.

Αν μάλιστα η σύνθεση είναι μια δομή ανάλογη με αυτή του προτύπου Composite (4.3.1), η οποία μπορεί να διαθέτει διαφορετικού τύπου (κλάσεις) σύνθετων αντικειμένων (Composites), τότε για κάθε τύπο (κλάση) σύνθετου αντικειμένου θα πρέπει να ορίσουμε ένα αντίστοιχο τύπο (κλάση) Iterator, χωρίς ο καλών να γνωρίζει τη διαφορά μεταξύ τους. Αυτό επιτυγχάνεται με την γενικοποίηση των διαφορετικών Iterator με μια abstract κλάση προκειμένου να επιτύχουμε πολυμορφική διαπέραση (**polymorphic iteration**). Ένα παράδειγμα παρουσιάζεται στην Εικόνα 4-3, όπου για δύο διαφορετικούς τύπους αντικειμένων συνθέσεων (List και SkipList), ορίζουμε αντίστοιχους τύπους αντικειμένων Iterator (ListIterator και SkipListIterator) κάτω από μία abstract κλάση (Iterator) με κοινή διεπαφή. Τέλος προκειμένων ο καλών (Client) να μη χρειάζεται να γνωρίζει τον τύπο της σύνθεσης που θέλει να διαπεράσει (και κατ επέκταση τον Iterator που αντιστοιχεί σε αυτή), καθιστούμε υπεύθυνο για την δημιουργία του κατάλληλου Iterator αντικειμένου το ίδιο το αντικείμενο της σύνθεσης. Αυτό απαιτεί την προσθήκη μίας λειτουργίας (π.χ. CreateIterator), με την κλήση της οποίας ο καλών θα αιτείται (ενιαία με τον ίδιο τρόπο) τη δημιουργία ενός Iterator για οποιονδήποτε τύπο σύνθεσης.



Εικόνα 4-3 : Παράδειγμα μοντέλου κλάσεων Polymorphic Iteration

Ένα τυπικό διάγραμμα κλάσεων για το σχεδιαστικό πρότυπο Iterator παρουσιάζεται στην Εικόνα 4-4



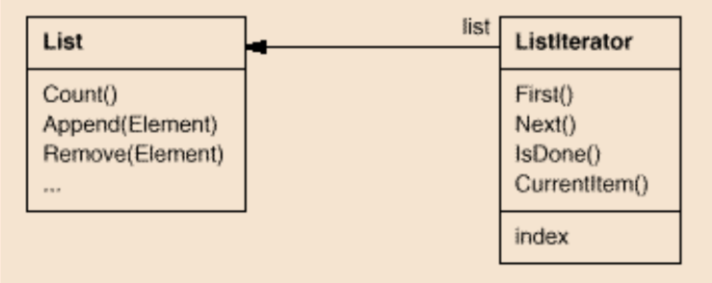
Εικόνα 4-4 : Τυπικό διάγραμμα κλάσεων του σχεδιαστικού προτύπου Iterator

Γενικά χρησιμοποιούμε το σχεδιαστικό πρότυπο Iterator όταν θέλουμε :

- να προσπελάσουμε τα αντικείμενα μιας σύνθεσης αντικειμένων χωρίς να γνωρίζουμε τις μεταξύ τους σχέσεις
- να διεξάγουμε πολλαπλές διαπεράσεις στην ίδια σύνθεση αντικειμένων ταυτόχρονα
- να παρέχουμε μια ενιαία διεπαφή για την διαπέραση διαφορετικών τύπων συνθέσεων αντικειμένων (polymorphic iteration)

4.3.2.1 *Iteration σε απλές δομές*

Για την διαπέραση των στοιχείων μιας απλής σύνθεσης (π.χ. μίας λίστας, List), αρκεί η δημιουργία μίας αντίστοιχης κλάσης αντικειμένων (ListIterator). Για μία διαπέραση δημιουργούμε ένα αντικείμενο ListIterator (περνώντας ως παράμετρο το αντικείμενο List) και στη συνέχεια καλούμε τις λειτουργίες του ListIterator για να διαπεράσουμε τα στοιχεία της List (Εικόνα 4-5).



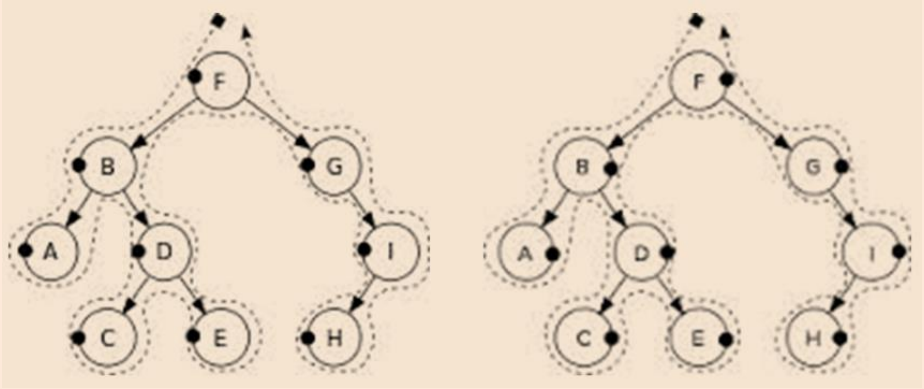
Εικόνα 4-5 : Διαπέραση (Iteration) Λίστας αντικειμένων

4.3.2.2 *Iteration σε σύνθετες (δενδροειδείς) δομές*

Η διαπέραση σύνθετων δομών (Εικόνα 4-3) όπως αυτές του προτύπου Composite (4.3.1), απαιτούν τη δημιουργία ξεχωριστών τύπων Iterator για διαφορετικούς τύπους συνθέσεων (4.3.2). Για την δημιουργία ενός Iterator καλούμε την λειτουργία CreateIterator της συγκεκριμένης (αντικείμενο) σύνθεσης, η οποία δημιουργεί και επιστρέφει το αντικείμενο του κατάλληλου Iterator (μέσω του μηχανισμού ιεράρχησης των κλάσεων - κληρονομικότητα).

Ωστόσο σε δενδροειδείς δομές αντικειμένων όπως αυτές στην Εικόνα 4-2, συχνά απαιτούνται διαπεράσεις ειδικού τύπου όπως οι **preorder** και **postorder traversals** (Εικόνα

4-6). Στην περίπτωση αυτή μπορούμε να δημιουργήσουμε επιπλέον μια κλάση αντικειμένων ειδικού για το σκοπό αυτό iterator, όπως PreOrderIterator, ως εξειδίκευση της αφηρημένης κλάσης Iterator. Ο PreOrderIterator θα υλοποιεί την ίδια διεπαφή με τον γονέα του (λειτουργίες First, Next, IsDone, CurrentItem) καλώντας κατά περίπτωση τον κατάλληλο Iterator του τρέχοντος αντικειμένου (σύνθεσης – κόμβου δέντρου). Η λειτουργία Next είναι αυτή που περιέχει τον αλγόριθμο που υλοποιεί την διαπέραση, χρησιμοποιώντας μια stack όπου και αποθηκεύει τα ενδιάμεσα αντικείμενα (συνθέσεις – κόμβοι δέντρου), προκειμένου να είναι σε θέσει να επανέλθει σε ένα αντικείμενο (κόμβο δέντρου) μετά την διαπέραση όλων των παιδιών του. Ειδικά τα στοιχειώδη αντικείμενα (φύλα δέντρου, δεν έχουν άλλα παιδιά) θα πρέπει να καλούν έναν ειδικού τύπου ουδέτερου iterator, όπως NullIterator, ο οποίος επίσης θα υλοποιεί την ίδια διεπαφή με τον γονέα του με τη διαφορά ότι στην πράξη θα τερματίζει αμέσως (IsDone = always true). Τέλος για την εκκίνηση της διαπέρασης αρκεί η απευθείας δημιουργία ενός αντικειμένου PreOrderIterator, η κλήση της λειτουργίας First, όπου στη συνέχεια με επαναληπτική κλήση των λειτουργιών Next και CurrentItem αποκτάμε πρόσβαση σε όλα τα αντικείμενα (κόμβους – συνθέσεις και φύλλα – στοιχειώδη αντικείμενα) της δομής με preorder σειρά.



Εικόνα 4-6 : Διάγραμμα PreOrder - PostOrder διαπέρασης δέντρου

Αντίστοιχα για την Post order διαπέραση, δημιουργήσουμε επιπλέον μια κλάση αντικειμένων, όπως PostOrderIterator, ως εξειδίκευση της αφηρημένης κλάσης Iterator. Πάλι η λειτουργία Next είναι αυτή που περιέχει τον αλγόριθμο που υλοποιεί την διαπέραση, χρησιμοποιώντας επίσης μια stack όπου και αποθηκεύει τα ενδιάμεσα αντικείμενα που επισκέπτεται.

Εκτενή ανάλυση και εφαρμογή των προαναφερόμενων Iterators καθώς και κάποιων παραλλαγών τους, παρουσιάζεται κατά την σχεδίαση και υλοποίηση του μεταγλωττιστή στο κεφάλαιο 7.

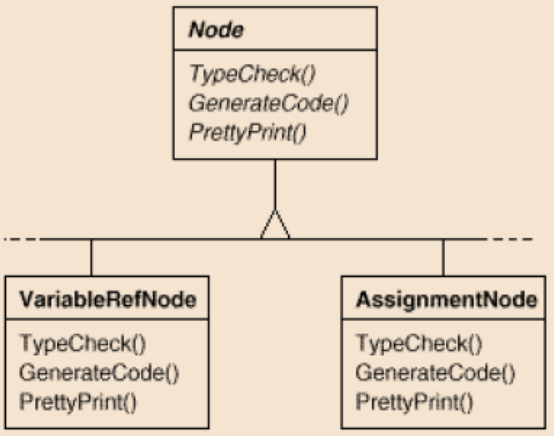
4.3.3 Visitor

Σκοπός του σχεδιαστικού προτύπου **Visitor** είναι να αναπαριστά μία λειτουργία που επιδρά στα στοιχεία μίας δομής αντικειμένων. Οι visitors μας επιτρέπουν να τροποποιούμε ή να ορίζουμε μια νέα λειτουργία χωρίς να τροποποιούμε την κλάση των αντικειμένων στα οποία αυτή επιδρά.

Σε πολλά προβλήματα δομών - συνθέσεων αντικειμένων (Εικόνα 4-1, Εικόνα 4-2) παρουσιάζεται η ανάγκη να ορίσουμε μία λειτουργία (method) στα αντικείμενα (κόμβους) της δομής, η οποία μάλιστα έχει διαφορετική υλοποίηση για κάθε είδος (κλάση)

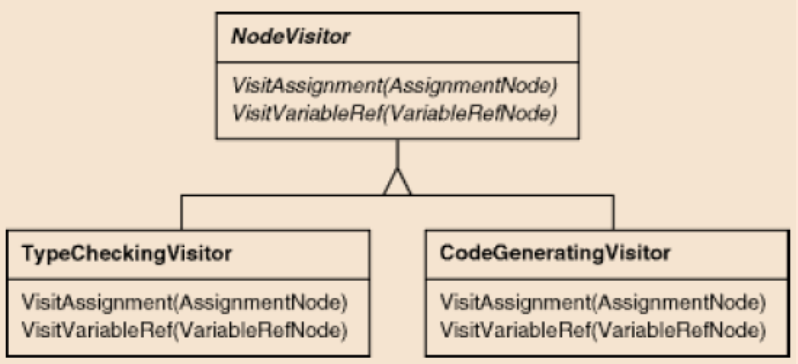
αντικειμένων της δομής. Επίσης είναι σύνηθες να ορίζουμε πολλές διαφορετικές λειτουργίες για κάθε κλάση αντικειμένων της δομής. Επειδή η κατανόηση του προτύπου είναι σημαντική, το παρακάτω παράδειγμα των (Gamma, Helm, Johnson, & Vlisssides, 1994) που αναφέρεται σε απλοποιημένες λειτουργίες ενός μεταγλωττιστή, παρουσιάζεται ποιο αναλυτικά.

Έστω Node η αφηρημένη κλάση αντικειμένων των στοιχείων μίας γλώσσας προγραμματισμού για την οποία θέλουμε να αναπτύξουμε ένα μεταγλωττιστή (Εικόνα 4-7). Έστω επίσης δύο εξειδικευμένες κλάσεις αντικειμένων VariableRefNode και AssigmentNode για την αναπαράσταση των αντίστοιχων στοιχείων, αναφορών μεταβλητών και εκφράσεων απόδοσης τιμής της γλώσσας. Θέλοντας να ορίσουμε δύο λειτουργίες για τον έλεγχο τύπων και την παραγωγή κώδικα, ορίζουμε τις μεθόδους TypeCheck() και GenerateCode(), τις οποίες δια μέσου της κληρονομικότητας και του πολυμορφισμού τις αναπτύσσουμε στις υποκλάσεις και δηλώνουμε το interfaces τους στην γονική αφηρημένη κλάση Node. Αν τώρα υποθέσουμε (κάτι που συμβαίνει συνήθως) ότι η υλοποίηση της λειτουργίας TypeCheck() είναι διαφορετική για την κάθε κλάση αντικειμένων, τότε το πρόβλημα αρχίζει και είναι εμφανές. Η κατανομή όλων των λειτουργιών διάσπαρτα στις κλάσεις των δομικών αντικειμένων προκαλεί σύγχυση και υποβαθμίζει την ποιότητα του λογισμικού. Επιπλέον η προσθήκη μιας νέα λειτουργίας συνήθως απαιτεί την επαναμεταγλώττιση όλων των κλάσεων. Θα ήταν καλύτερα αν κάθε νέα λειτουργία μπορούσε να προστεθεί κάπου ανεξάρτητα, ξεχωριστά από τις κλάσεις των στοιχείων στις οποίες και επιδρούν.



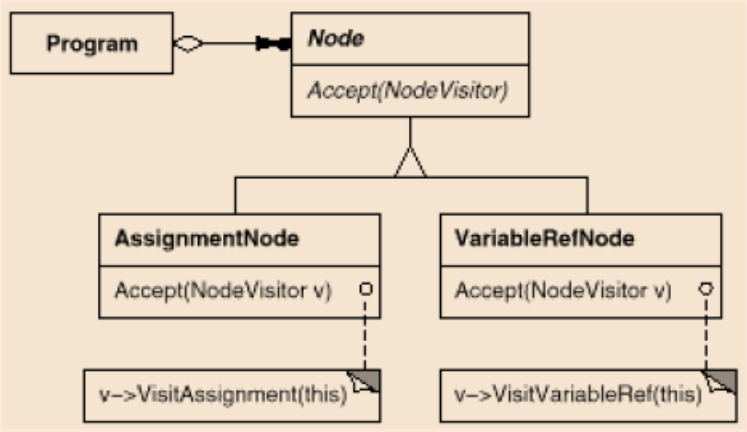
Εικόνα 4-7 : Παράδειγμα δομής κλάσεων στοιχείων μεταγλωττιστή

Η λύση είναι να δημιουργήσουμε μια ξεχωριστή αφηρημένη κλάση, καλούμενη NodeVisitor, στην οποία θα τοποθετούνται οι σχετικές μεταξύ τους λειτουργίες, για κάθε κλάση στοιχείων της δομής, σε ξεχωριστά αντικείμενα των κλάσεων TypeChekingVisitor και CodeGeneratingVisitor που κληρονομούν την NodeVisitor (Εικόνα 4-8).



Εικόνα 4-8 : Παράδειγμα δομής κλάσεων Visitor μεταγλωττιστή

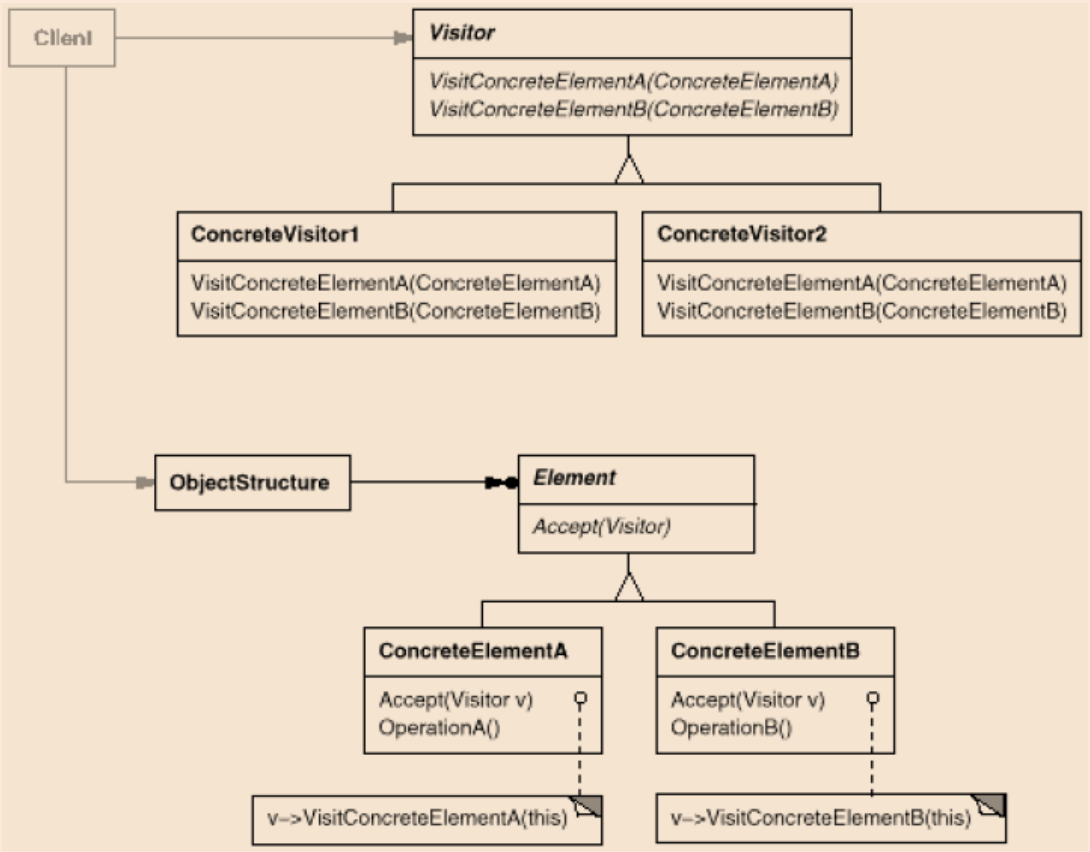
Το επιθυμητό (ανάλογα με την λειτουργία που επιλέγουμε) αντικείμενο της visitor (TypeChekingVisitor ή CodeGeneratingVisitor), το περνάμε στα αντικείμενα των στοιχείων της γλώσσας κατά την διαπέρασή τους. Όταν ένα στοιχείο (αντικείμενο VariableRefNode ή AssigmentNode) της γλώσσας δέχεται «accepts» το συγκεκριμένο αντικείμενο visitor, στέλνει ένα αίτημα (message - κλήση) στον visitor στο οποίο κωδικοποιεί την κλάση του στοιχείου (object) της γλώσσας (Εικόνα 4-9). Επιπλέον η κλήση αυτή περιλαμβάνει και το ίδιο το στοιχείο (object) της γλώσσας ως παράμετρο προκειμένου στη συνέχεια ο visitor να εκτελέσει την κατάλληλη λειτουργία για/σε αυτό το στοιχείο.



Εικόνα 4-9 : Παράδειγμα κλάσεων με αποδοχή visitor

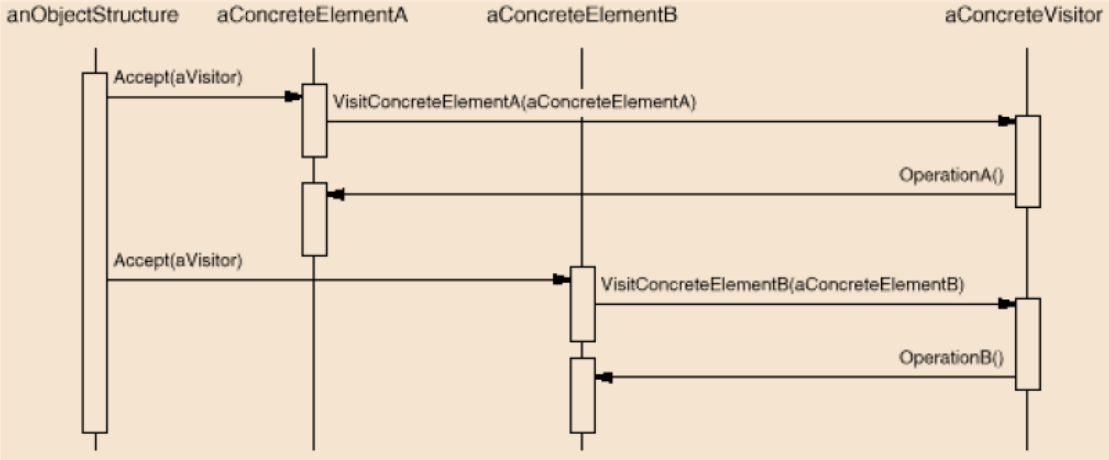
Για την δημιουργία μίας νέας λειτουργίας, αρκεί να δημιουργήσουμε μία νέα υποκλάση στον NodeVisitor ιεραρχικά, στην οποία και τα τοποθετήσουμε όλες τις σχετικές μεταξύ τους λειτουργίες, για κάθε κλάση στοιχείων της δομής Node.

Ένα τυπικό διάγραμμα κλάσεων για το σχεδιαστικό πρότυπο Visitor παρουσιάζεται στην Εικόνα 4-10.



Εικόνα 4-10 : Τυπικό διάγραμμα κλάσεων του σχεδιαστικού προτύπου Visitor

Το διάγραμμα ακολουθίας (sequence diagram) στην Εικόνα 4-11, παρουσιάζει την συνεργασία και τα μηνύματα μεταξύ ενός βασικού αντικειμένου ObjectStructure, ενός αντικειμένου visitor και δύο διαφορετικού τύπου στοιχείων (elements).



Εικόνα 4-11 : Τυπικό διάγραμμα ακολουθίας σχεδιαστικού προτύπου Visitor

Γενικά χρησιμοποιούμε το σχεδιαστικό πρότυπο Visitor όταν :

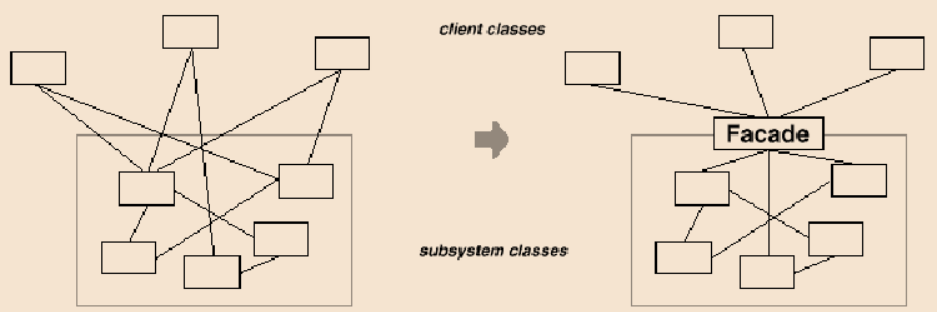
- θέλουμε να εφαρμόσουμε λειτουργίες στα αντικείμενα μιας δομής με βάση τις κοινές διεπαφές των γονικών τους κλάσεων
- πολλές διακριτές και ανεξάρτητες λειτουργίες χρειάζεται να εφαρμοσθούν σε αντικείμενα μίας δομής και επιθυμείτε να αποφύγετε τη συμπερίληψή τους σε αυτές τις κλάσεις

- οι κλάσεις που ορίζουν τα αντικείμενα της δομής σπάνια αλλάζουν, αλλά συχνά ορίζονται νέες λειτουργίες για τα στοιχειά αυτής της δομής

4.3.4 Façade

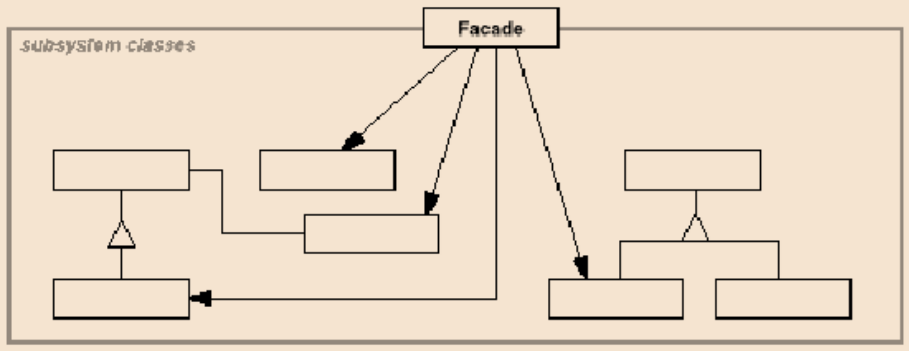
Σκοπός του σχεδιαστικού προτύπου **Facade,** είναι η παροχή μιας ενιαίας διαπροσωπίας (**interface**) σε ένα σύνολο διαπροσωπιών ενός υποσυστήματος.

Η δόμηση ενός συστήματος από πολλά υποσυστήματα αποτελεί σχεδόν κανόνα για μεγάλα και πολύπλοκα κυρίως συστήματα. Επιλέγεται προκειμένου να μειωθεί η πολυπλοκότητα της λύσης, να παραχθεί ποιοτικότερο λογισμικό ή και για λόγους επαναχρησιμοποίηση τμημάτων του λογισμικού. Ωστόσο κοινό σχεδιαστικό στόχο αποτελεί η ελαχιστοποίηση των αλληλεξαρτήσεων μεταξύ των υποσυστημάτων αυτών. Ένα τρόπος για την επίτευξη του στόχου αυτού είναι η δημιουργία ενός αντικειμένου (κλάσης) το οποίο και θα παρέχει μια μοναδική ενοποιημένη διαπροσωπία, η οποία και θα εξυπηρετεί όλες τις ανάγκες επικοινωνίας με τις επιμέρους διαπροσωπίες των υποσυστημάτων (Εικόνα 4-12).



Εικόνα 4-12 : Διάγραμμα επικοινωνίας συστήματος μέσω ενιαίας διεπαφής

Ένα τυπικό διάγραμμα κλάσεων για το σχεδιαστικό πρότυπο Facade παρουσιάζεται στην Εικόνα 4-13.



Εικόνα 4-13 : Τυπικό διάγραμμα κλάσεων του σχεδιαστικού προτύπου Facade

5 Μεταγλωττιστές

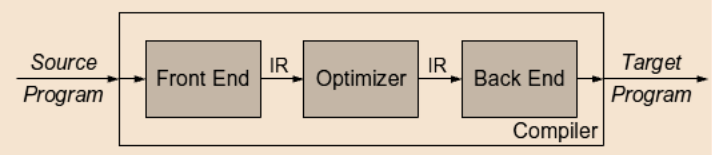
5.1 Γενικά

Οι μεταγλωττιστές είναι προγράμματα που δεχόμενα ως είσοδο προγράμματα σε γλώσσα υψηλού επιπέδου και τα μετατρέπουν σε προγράμματα χαμηλού επιπέδου (συνήθως γλώσσα μηχανής[8] προς εκτέλεση). Η πλειονότητα της σχετικής βιβλιογραφίας ασχολείται κυρίως με την παρουσίαση θεμελιωδών τεχνικών αυτόματης μετάφρασης που χρησιμοποιούνται κατά την υλοποίηση ενός μεταγλωττιστή (περιγραφές, δομές, αλγόριθμους, διεργασίες, κ.λ.π.).

Στη συνέχεια γίνεται μια συνοπτική παρουσίαση των βασικών τμημάτων και λειτουργιών εντός τυπικού μεταγλωττιστή, σε βαθμό και έκταση που να εξυπηρετούν την κατανόηση της εφαρμογής των σχεδιαστικών προτύπων της παρούσας εργασίας. Μια πλήρη ανάλυση της αρχιτεκτονικής σχεδίασης μεταγλωττιστών παρέχεται στην σχετική βιβλιογραφία (Cooper & Torczon, 2012), (Aho, Lam, Sethi, & Ullman, 2007).

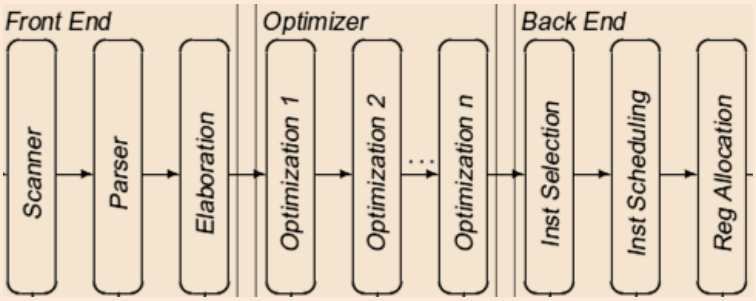
5.2 Δομή μεταγλωττιστή

Ένας τυπικός μεταγλωττιστής αποτελείται από δύο ή τρία βασικά μέρη : Front End, Optimizer (προαιρετικά), Back End. Η διασύνδεση του front end με το back end πραγματοποιείτε διαμέσου μιας (ή και περισσότερων) τυπικής δομής για την αναπαράσταση του προγράμματος σε μια ενδιάμεση μορφή (IR, Intermediate Representation). Ο optimizes (αν υπάρχει) αναλύει και τροποποιεί την IR για την βελτιστοποίηση της μεταγλώττισης (Εικόνα 5-1, (Cooper & Torczon, 2012)).



Εικόνα 5-1 : Βασική δομή μεταγλωττιστή

Κάθε μέρος ασχολείται με διαφορετικές διεργασίες και συνήθως χρειάζονται περισσότερα από ένα περάσματα (υπό διεργασίες) για την ολοκλήρωση κάθε μέρους (Εικόνα 5-2).



Εικόνα 5-2 : Τυπικές διεργασίες μεταγλώττισης

Στην πράξη ένας μεταγλωττιστής συνήθως χρησιμοποιεί ένα σύνολο από διαφορετικές IR (trees, lists, structures, graphs, κλπ) κατά την διάρκεια των ενδιάμεσων περασμάτων

[8] Γλώσσα μηχανής είναι τυπικά το Instruction Set που υποστηρίζεται από έναν επεξεργαστή (CPU). Συνολικά η σχεδίαση ενός instruction set καλείται Instruction Set Architecture (ISA)

προκειμένου να υλοποιήσεις τις αναλύσεις και τους μετασχηματισμούς που απαιτούνται για την παραγωγή του τελικού αποτελέσματος.

5.2.1 Front End

Το **Frond End** τμήμα ενός μεταγλωττιστή, ασχολείται με την γλώσσα εισόδου, δηλαδή την κατανόηση της και την αποτύπωσή της ανάλυσης αυτής σε IR μορφή. Μια γλώσσα εισόδου πρέπει ακολουθεί συγκεκριμένη δομή (φορμαλισμό) προκειμένου να είναι συνεπής και σαφής (χωρίς καμία ασάφεια). Όλες οι διαδεδομένες γλώσσες προγραμματισμού είναι αυστηρά τυποποιημένες με βάση ένα διεθνώς αναγνωρισμένο πρότυπο (π.χ. ISO[9] standard).

Μια τυποποιημένη γλώσσα αναλύεται σε στοιχειώδεις λέξεις (αλφαριθμητικά) όπως : identifiers, constants, strings, keywords. Την αναγνώριση των token την αναλαμβάνει το τμήμα του **Scanner** του front end του μεταγλωττιστή με βάση την περιγραφή τους με κανονικές εκφράσεις (**Regular Expressions**). Στη συνέχεια τα στοιχειώδη στοιχεία ακολουθούν τη δομή – σύνταξη της γραμματικής (**grammar**) της γλώσσας η οποία και περιγράφεται με ένα πεπερασμένο σύνολο αυστηρά οριζόμενων κανόνων. Την αναγνώριση της γραμματικής την αναλαμβάνει το τμήμα του **Parser** του front end του μεταγλωττιστή. Τέλος, εφόσον η είσοδος περάσει από τους συντακτικούς ελέγχους (**syntax checking**) του parser ο compiler κατασκευάζει μια ενδιάμεση αναπαριστά (IR) της γλώσσας προκειμένου να πραγματοποιήσει τα επόμενα στάδια επεξεργασίας. Υπάρχουν αρκετοί τρόποι ενδιάμεση αναπαράστασης (IR) μιας γλώσσας και η επιλογή του καταλληλότερου εξαρτάται από την υλοποίηση του εκάστοτε μεταγλωττιστή. Στο μέρος του front end ενός μεταγλωττιστή συνήθως υλοποιούνται και οι σημασιολογικοί έλεγχοι (**semantic checking**) της γλώσσας εισόδου όπως declaration analysis, type checking, scope checking, κ.λ.π.

5.2.2 Optimizer

Ο **Optimizer** είναι προαιρετικό τμήμα ενός μεταγλωττιστή και σκοπός του είναι η ανάλυση της IR μορφής της γλώσσας για την ανακάλυψη στοιχείων σχετικά με το περιεχόμενο και την χρήση της περιεχόμενης γνώσης, προκειμένου να αναμορφωθεί ο κώδικας ώστε να υπολογίζει το ίδιο αποτέλεσμα με αποδοτικότερο τρόπο. Ως αποδοτικότερος συνήθως επιδιώκουμε ποιο γρήγορο ή μικρότερο κώδικα ή συνδυασμό αυτών. Τυπικά ο optimizer πραγματοποιεί ένα είδος ανάλυσης της IR και κατόπιν ένα μετασχηματισμό της. Ως ανάλυση νοούνται έλεγχοι όπως, Data-flow analysis (σχετικά με την ροή των δεομένων κατά τον χρόνο εκτέλεσης), Dependency analysis, κ.λ.π.

5.2.3 Back End

Το **Back-end** τμήμα ενός μεταγλωττιστή έχει ως σκοπό την επιλογή των κατάλληλων εντολών της γλώσσας εξόδου (συνήθως target machine) προκειμένου να υλοποιηθεί η κάθε επιμέρους λειτουργία της IR. Η τελική επιλογή και διαμόρφωση των εντολών πραγματοποιείται συνήθως μετά από μια σειρά διεργασιών – ανάλυσης όπως: Instruction selection, Instruction scheduling, register allocation.

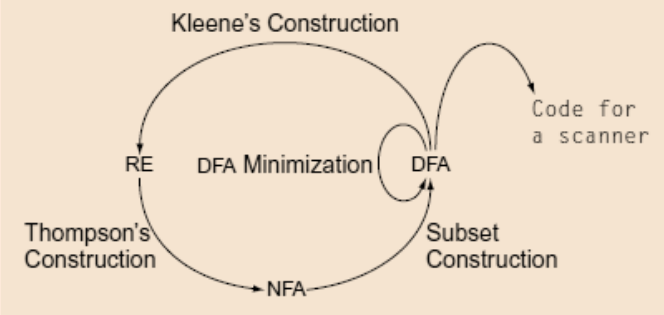
[9] International Organization for Standardization : δραστηριοποιείται στην ανάπτυξη και δημοσίευση διεθνών προτύπων

5.3 Scanner

Ο **scanner** (ή λεκτικός αναλυτής, **lexical analyzer**), ως το αρχικό στάδιο επεξεργασίας ενός μεταγλωττιστή, διαβάζει μια ακολουθία χαρακτήρων και παράγει μια ακολουθία λέξεων κατηγοριοποιημένες (**syntactic category**) με βάση ένα σύνολο κανόνων (**microsyntax**). Τέτοιες λέξεις μιας γλώσσας μπορεί να είναι identifiers, constants, strings, keywords, καλούμενα και ως **tokens**. Η αναγνώριση των λέξεων πραγματοποιείται από τον scanner με μία σειρά πεπερασμένες καταστάσεις η αλλιώς πεπερασμένα αυτόματα (**finite automaton - FA**) που αποτελούν έναν αυστηρό (σαφή) φορμαλισμό για την αναγνώριση λέξεων έχοντας ένα πεπερασμένο σύνολο καταστάσεων, ένα αλφάβητο, μια συνάρτηση μετάβασης, μία κατάσταση έναρξης και μία ή περισσότερες αποδεκτές καταστάσεις. Κάθε FA μπορεί να περιγραφεί με μία σημειογραφία καλούμενη **Regular Expression** (RE).

Στη θεωρία της επιστήμης των υπολογιστών μία RE είναι μια σειρά από χαρακτήρες που περιγράφει ένα πρότυπο (pattern) αναζήτησης και χρησιμοποιείται για λειτουργίες όπως το ταίριασμα προτύπων (pattern matching) σε αλφαριθμητικά. Ο φορμαλισμός (αλγεβρικοί ορισμοί) της regular language έγινε από τον Αμερικανό μαθηματικό Stephen Kleen και η χρήση της έγινε ευρέως αποδεκτή (Aho & Ullman, 1995).

Ο φορμαλισμός της περιγραφής λέξεων με κανονικές εκφράσεις (RE) είναι ισοδύναμος με τα πεπερασμένα αυτόματα (FA). Στη βιβλιογραφία περιγράφονται μια σειρά αλγόριθμοι για την μετατροπή των RE σε NFA[10] στη συνέχεια σε DFA[11] από όπου και μπορεί να εξαχθεί ο κώδικας του αντίστοιχου scanner για την αναγνώριση των λέξεων των αρχικών RE (Εικόνα 5-3).



Εικόνα 5-3 : Κύκλος αναπαραστάσεων RE - NFA – DFA

Η ανάλυση του φορμαλισμού των RE και FAs καθώς και των αλγόριθμων μετατροπής μεταξύ τους καθώς και της κατασκευής ενός scanner δεν εξυπηρετούν τους σκοπούς της παρούσας εργασίας. Άλλωστε υπάρχουν δοκιμασμένα έτοιμα λογισμικά τα οποία δεχόμενα ως είσοδο ένα σύνολο REs παράγουν αυτόματα τον κώδικα του scanner ο οποίος και μπορεί να ενσωματωθεί στον υπό ανάπτυξη μεταγλωττιστή. Η υλοποίηση της εργασίας χρησιμοποιεί για την δημιουργία του scanner το ελεύθερο λογισμικό Flex, το οποίο και περιγράφεται στο κεφάλαιο 6.

[10] Τα NFAs (nondeterministic finite automaton) είναι FAs που έχουν από μία κατάσταση πολλαπλές μεταβάσεις για τον ίδιο χαρακτήρα ή μετάβαση στο κενό string

[11] Τα DFAs (deterministic finite automaton) είναι FAs που από μία κατάσταση έχουν μόνο μία μετάβαση για ένα χαρακτήρα, χωρίς μετάβαση στο κενό string

5.4 Parser

Ο **Parser**, (ή συντακτικός αναλυτής, **syntax analyzer**) ως το επόμενο στάδιο επεξεργασίας ενός μεταγλωττιστή, διαβάζει μια ακολουθία λέξεων (προερχόμενη από τον scanner μαζί με την semantic category της κάθε λέξης) και εξάγει μια συντακτική δομή του προγράμματος, προσαρμόζοντας τες (τις λέξεις) σε ένα γραμματικό μοντέλο της γλώσσας εισόδου. Αν η είσοδος γίνει αποδεκτή από τον parser ως έγκυρο πρόγραμμα, ο μεταγλωττιστής παράγει ένα συμπαγές μοντέλο του προγράμματος για τις επόμενες φάσεις της μεταγλώττισης.

Το γραμματικό μοντέλο της γλώσσας εισόδου καθορίζεται από μία γραμματική ελεύθερη συμφραζομένων (**context-free grammars**, **CFGs**), μια σημειογραφία για τον σαφή καθορισμό – περιγραφή της σύνταξης. Η διαδικασία του parsing, δοθέντος μίας ακολουθίας λέξεων s και μιας γραμματικής G, εντοπίζει μια παραγωγή-αναγωγή (**derivation**) στην G που παράγει την s. Στη βιβλιογραφία υπάρχουν πολλοί αλγόριθμοι για την εύρεση μιας derivation όπως: **top-down parsing** με parsers αναδρομικής καθόδου και **LL(1) parsers**, και **bottom-up parsing** με **LR(1) parsers**.

Η ανάλυση του φορμαλισμού των CFGs καθώς και των αλγόριθμων αναγωγής (derivation) και της κατασκευής ενός parser δεν εξυπηρετούν τους σκοπούς της παρούσας εργασίας. Άλλωστε, όπως για τους scanners, υπάρχουν δοκιμασμένα έτοιμα λογισμικά τα οποία δεχόμενα ως είσοδο μία CFG, παράγουν αυτόματα τον κώδικα του parser ο οποίος και μπορεί να ενσωματωθεί στον υπό ανάπτυξη μεταγλωττιστή. Η υλοποίηση της εργασίας χρησιμοποιεί ως parser το ελεύθερο λογισμικό Bison, το οποίο και περιγράφεται στο κεφάλαιο 6. Ωστόσο επειδή ο parser συνήθως περιέχει κώδικα που σχετίζεται και με άλλες λειτουργίες του μεταγλωττιστή, ακολούθως παραθέτουμε μια συνοπτική αναφορά των εννοιών που πραγματεύεται η παρούσα εργασία.

5.4.1 Context-Free Grammars

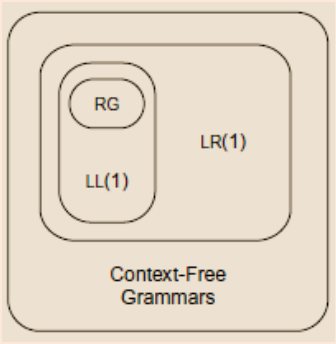
Μια **CFG** ορίζει για μία γλώσσα L, το σύνολο των συμβόλων τα οποία αποτελούν έγκυρες εκφράσεις της L. Έκφραση (**sentence**) είναι η ακολουθία (string) των συμβόλων που μπορούν να αναχθούν (derived) από τους κανόνες της γραμματικής. Μια CFG είναι ένα σύνολο κανόνων (**productions**), αναφερόμενο ως G, για την περιγραφή του σχηματισμού των εκφράσεων (sentence). Το σύνολο των γλωσσών που ορίζονται από τις CFG ονομάζεται σύνολο από **context-free languages**. Οι κανόνες περιέχουν Nonterminal και Terminal symbols. Τα **Nonterminal symbols** είναι σημασιολογικές μεταβλητές που χρησιμοποιούνται στους κανόνες της γραμματικής. Τα **Terminal symbols** είναι οι στοιχειώδεις λέξεις της έκφρασης, προέρχονται από τον scanner, και αναπαρίστανται στη γραμματική από την συντακτική τους κατηγορία.

Τυπικά μία CFG, G ορίζεται από το τετράπτυχο (T, NT, S, P) όπου:

- T : είναι το σύνολο των **terminal symbols**
- NT : είναι το σύνολο των **nonterminal symbols**
- S : είναι ένα nonterminal symbol σχεδιασμένο ως το **start symbol** της γραμματικής
- P : είναι το σύνολο των **productions** της μορφής $\boldsymbol{NT} \rightarrow (\boldsymbol{T} \cup \boldsymbol{NT})^{+}$

5.4.2 LR(1) Grammars

Το πεδίο των CFGs περιέχει αρκετά υποσύνολα γραμματικών, τα ποιο βασικά από τα οποία παρουσιάζονται (Εικόνα 5-4) ιεραρχημένα σε σχέση με την δυσκολία αναγνώρισης (parsing) αυτών των γραμματικών (εξωτερικά παρουσιάζονται οι λιγότερο αποδοτικές γραμματικές).



Εικόνα 5-4 : Ιεράρχηση CFGs

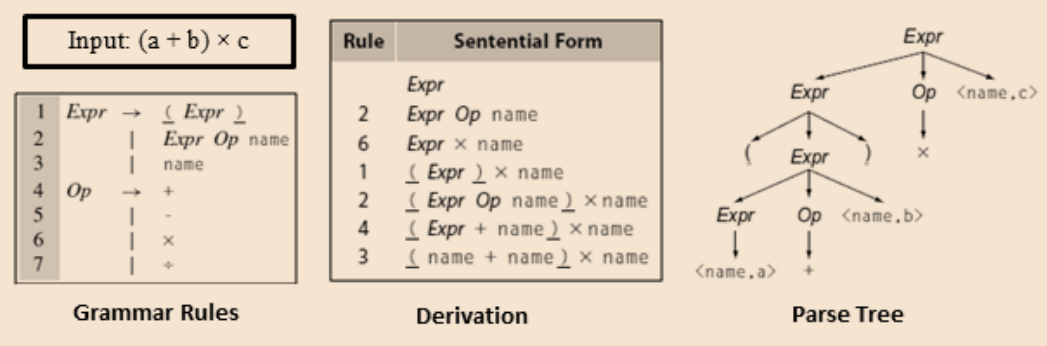
Η κλάση των **LR(1)** γραμματικών, περιλαμβάνει ένα μεγάλο υποσύνολο σαφών (**unambiguous**) CFGs, οι οποίες μπορούν να αναχθούν (parsed) από κάτω προς τα πάνω (bottom-up, **R**evers), με γραμμική σάρωση από αριστερά (**L**eft to right) προς τα δεξιά, ελέγχοντας κατά μέγιστο μία (**1**) προπορευόμενη λέξη (look ahead token) σε σχέση με το τρέχον σύμβολο της εισόδου. Οι ιδιότητες των LR(1) γραμματικών καθιστούν δυνατή την κατασκευή γρήγορων και αποδοτικών αντίστοιχων **LR(1) parsers**.

Οι regular grammars (RGs) είναι CFGs με κανόνες περιορισμένους στις μορφές $A \to \alpha$ και $A \to \alpha B$ όπου $A, B \in NT$ και $\alpha \in T$. Οι RGs είναι ισοδύναμες με τις regular expressions (RE) και κωδικοποιούν ακριβώς αυτέ τις γλώσσες που μπορούν να αναγνωρισθούν από ένα DFA (ενότητα 5.3).

Σχεδόν όλες οι γλώσσες προγραμματισμού μπορούν να εκφρασθούν από τις LR(1) γραμματικές και οι περισσότεροι μεταγλωττιστές χρησιμοποιούν ένα γρήγορο αλγόριθμο βασιζόμενο στις περιορισμένες κλάσεις LL(1) και LR(1) των CFGs. Η μεγάλη διαθεσιμότητα έτοιμων εργαλείων parser από LR(1) γραμματικές τις έχει καθιερώσει στην υλοποίηση των μεταγλωττιστών.

5.4.3 Bottom-Up Parsing

Ο bottom-up parser λειτουργεί με βάση τις χαμηλού επιπέδου λεπτομέρειες, δηλαδή τις λέξεις (προερχόμενες από τον scanner), συσσωρεύοντας το περιεχόμενο τους μέχρις ότου η αναγωγή (derivation) να γίνει εμφανής. **Derivation** είναι μια σειρά βημάτων που εκκινούν με το αρχικό σύμβολο (**start symbol**) της γραμματικής, και τελειώνουν με μία έκφραση της γλώσσας (Cooper & Torczon, 2012, p. 116).



Εικόνα 5-5 : Παράδειγμα bottom-up derivation

Παράλληλα ο bottom-up parser χτίζει ένα parse-tree εκκινώντας από τα φύλλα του δέντρου με κατεύθυνση προς τη ρίζα. Ο parser κατασκευάζει ένα φύλλο (leaf) για κάθε λέξη (token) που επιστρέφεται από τον scanner. Για την δόμηση μιας derivation ο parser προσθέτει επίπεδα από κόμβους (nodes) nonterminal symbols, πάνω από τα φύλλα με δομή που κατευθύνεται από την γραμματική σε συνδυασμό με το μερικώς ολοκληρωμένο κάτω μέρος του parse-tree. Η ρίζα του δέντρου είναι πάντα το start symbol της γραμματικής (Εικόνα 5-5).

Συνεπώς το **parse-tree** είναι μια ακριβής αναπαράσταση της δομής (derivation) του προγράμματος, όπου τα φύλλα του είναι τα token του προγράμματος και κάθε κόμβος είναι μια εφαρμογή (reduction) ενός κανόνα της γραμματικής της γλώσσας.

5.4.4 LR(1) Bottom-Up parsing

Συνοπτικά ένας LR(1) parser με είσοδο μια ακολουθία λέξεων (tokens) ενός προγράμματος, εργάζεται χρησιμοποιώντας μια ειδική αναπαράσταση (συνήθως ένα πίνακα μετάβασης) της γλώσσας και μια stack. Σε κάθε βήμα εξετάζει την αναπαράσταση της γλώσσας, το περιεχόμενο της stack, το τρέχον σύμβολο καθώς και το προπορευόμενο σύμβολο (look ahead token), προκειμένου να αποφανθεί αν θα εισάγει (**shift**) το επόμενο token στη stack ή θα εφαρμόσει έναν κανόνα (**reduce**) αντικαθιστώντας στην stack τα σύμβολα του δεξιού τμήματος του κανόνα με το αριστερό nonterminal σύμβολο. Η διαδικασία επαναλαμβάνεται μέχρι να εξαντληθούν οι λέξεις και στην stack να παραμείνει το start-symbol της γραμματικής. Σε διαφορετική περίπτωση ο parser έχει αποτύχει να εξάγει μια reduction από το πρόγραμμα και άρα η είσοδος δεν είναι δεκτή με βάση την συγκεκριμένη γραμματική (Εικόνα 5-5).

Βασικό χαρακτηριστικό των LR(1) γραμματικών που τις καθιστούν γρήγορες κατά την διαδικασία του parsing, ως unambiguous CFGs, είναι το γεγονός ότι με βάση την παραπάνω περιγραφή σε κάθε βήμα της διαδικασία η απόφαση για shift μίας λέξης ή reduce ενός κανόνα είναι μονοσήμαντη και μάλιστα σε περίπτωση reduction μόνο ένας κανόνας ταιριάζει κάθε φόρα. Η προτεραιότητα των κανόνων της γραμματικής μπορεί να καθορίζεται είτε από την ίδια την δομή των κανόνων, είτε από την συγκεκριμένη υλοποίηση του parser (π.χ. σειρά εμφάνισης κανόνων, ρητή δήλωση προτεραιοτήτων). Αυτή η ιδιότητα των LR(1) γραμματικών μας εξασφαλίζει ότι δεδομένου μις ακολουθίας λέξεων (tokens), η παραγόμενη αναγωγή (derivation) ή αλλιώς η δομή του παραγόμενου parse-tree θα είναι πάντα μοναδική/ο.

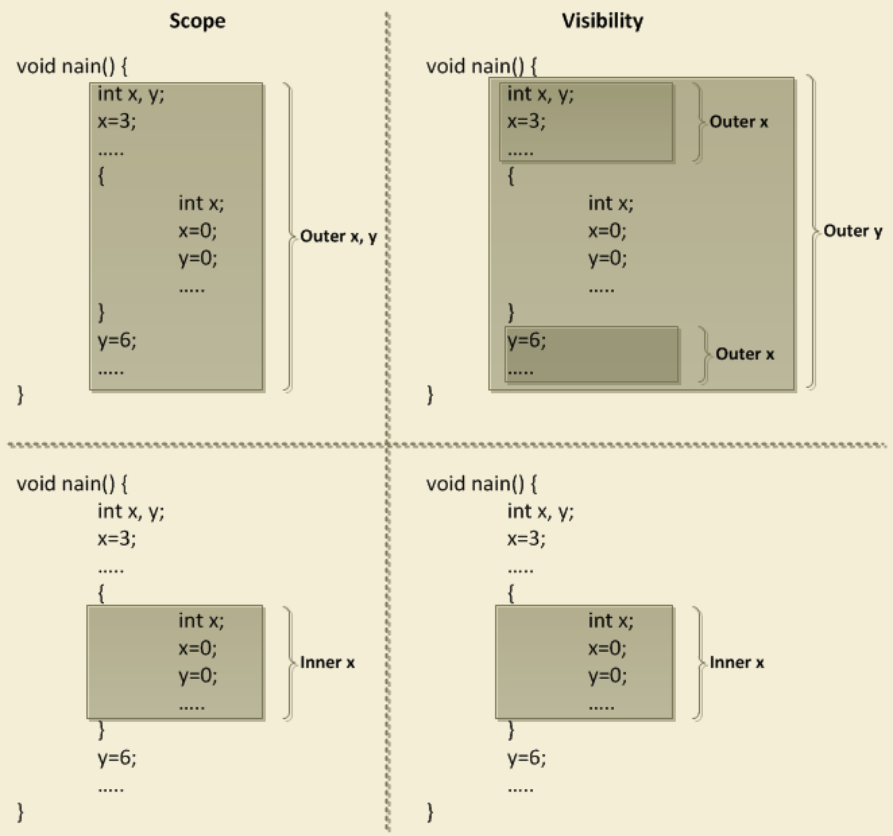
Συνεπώς το **parse-tree** είναι μια ακριβής αναπαράσταση της δομής (derivation) του προγράμματος, όπου κάθε φύλλο του είναι ένα token του προγράμματος (παραγόμενο κατά το shifting του token στην stack του parser), και κάθε κόμβος είναι μια εφαρμογή (reduction) ενός κανόνα της γραμματικής της γλώσσας (παραγόμενος κατά το reduce ενός κανόνα στην stack του parser).

5.5 Context-Sensitive Analysis

Προκειμένου ο μεταγλωττιστής να είναι σε θέση να μετατρέψει το πηγαίο πρόγραμμα σε απευθείας εκτελέσιμο κώδικα, απαιτείται βαθύτερη γνώση του νοήματος του προγράμματος πέραν της συντακτικής ανάλυσης (syntax analysis) που προκύπτει από τον scanner και parser. Για παράδειγμα πρέπει να γνωρίζει το είδος των τιμών (σταθερών και μεταβλητών), την ροή τους από όνομα σε όνομα, την περιοχή ισχύ τους, την δομή των υπολογισμών, κ.λ.π. Αυτού του είδους ανάλυση με βάση το περιεχόμενο του προγράμματος ονομάζεται σημασιολογική ανάλυση (**Context-Sensitive Analysis**) και συνήθως πραγματοποιείται από το τελευταίο στάδιο του front-end του μεταγλωττιστή.

5.5.1 Scope system

Scope System είναι το γενικό πλαίσιο που προδιαγράφουν οι γλώσσες προγραμματισμού υψηλού επιπέδου σχετικά την ισχύ και την ορατότητα των συμβόλων (identifiers) του προγράμματος στα επιμέρους τμήματά του (blocks, function, modules, κλπ). Το **scope** ενός identifier είναι το τμήμα του προγράμματος εντός του οποίου ο identifier μπορεί να χρησιμοποιηθεί ως αναφορά στην οντότητα του προγράμματος που του αντιστοιχεί. Γενικά ένα block του προγράμματος μπορεί να είναι ένα ολόκληρο αρχείο (program module), ή το σώμα μιας συνάρτησης το τμήμα κώδικα μιας δομής επανάληψης ή μιας δομής απόφασης, ή ακόμα και τμήματα ορισμένα από τον προγραμματιστή με ειδικό συμβολισμό. Τα τμήματα αυτά ακολουθούν συνήθως μια συνέπεια σχετικά με την αρχή και το τέλος τους. Όταν ένα τέτοιο τμήμα ξεκινά μέσα σε ένα άλλο τμήμα, τότε οι δηλώσεις του εξωτερικού τμήματος είναι ορατές και στο εσωτερικό τμήμα και οι δηλώσεις του εσωτερικού επικαλύπτουν τις δηλώσεις με τα ίδια ονόματα του εξωτερικού. Η λογική αυτή επεκτείνεται ιεραρχικά σε πολλαπλά επίπεδα επικάλυψης των scope (σχετικό παράδειγμα Εικόνα 5-6).



Εικόνα 5-6 : Παράδειγμα scope – visibility

Δεν επιτρέπεται η δήλωση ενός identifier με το ίδιο όνομα εντός του ιδίου scope με εξαίρεση ειδικές περιπτώσεις όπου από τον τρόπο δήλωσης ή/και αναφοράς μπορεί να προσδιορισθεί η διαφορετικότητά[12] του. Ο έλεγχος της συνέπειας των scopes ως προς την αρχή και το τέλος τους πραγματοποιείται από το συντακτικό (γραμματική) της γλώσσας από τον parser. Ωστόσο ο έλεγχος της συνέπειας των ονομάτων των identifiers καθώς και η συσχέτιση των αναφορών (χρήση identifier) με τις κατάλληλες δηλώσεις τους στο κατάλληλο scope υπάγεται στην διαδικασία της context-sensitive analysis. Ο μεταγλωττιστής πρέπει να κάνει τον απαραίτητο έλεγχο και συσχετίσεις με βάση την ανάλυση του parse-tree και (συνήθως) τη δημιουργία ενός πίνακα συμβόλων για την αποθήκευση και αναζήτηση της σχετικής πληροφορίας.

5.5.2 Type System

Οι περισσότερες γλώσσες προγραμματισμού συσχετίζουν μια συλλογή από ιδιότητες (**type**) με κάθε τιμή δεδομένων του προγράμματος. Οι τύποι μπορούν να καθορίζονται με πολλούς τρόπους σε μια γλώσσα με συνηθέστερο τις δηλώσεις τύπων (type declaration) των γλωσσών προγραμματισμού. Ωστόσο αν σε κάποια γλώσσα δεν απαιτείται η ρητή δήλωση των τύπων, τότε οι τύποι μπορούν να καθορίζονται από τον μεταγλωττιστή με βάση τις προδιαγραφές του προτύπου της γλώσσας. Το σύνολο των τύπων μιας γλώσσας προγραμματισμού μαζί με τους κανόνες που χρησιμοποιούν τύπους για τον καθορισμό της συμπεριφοράς του προγράμματος, καλούνται συνολικά σύστημα τύπων (**type system**) (Cooper & Torczon, 2012, p. 165).

[12] Στη γλώσσα C ορίζονται τέσσερα διαφορετικά name spaces για τα ονόματα των identifiers, όπου σε ένα scope μπορεί να συνυπάρχουν identifiers με το ίδιο όνομα αρκεί να ανήκουν σε διαφορετικά name spaces.

Κατά την φάση της Context-Sensitive Analysis ο μεταγλωττιστής είναι απαραίτητο να πραγματοποιήσει έλεγχο τύπων (**type-checking**) του προγράμματος, προκειμένου να παράγει εκτελέσιμο κώδικα αποφεύγοντας ασυμβατότητες τύπων κατά το χρόνο εκτέλεσης του. Ένα τυπικό type system αποτελείται από τους base types, κανόνες για την κατασκευή νέων types, μια μέθοδο καθορισμού ισότητας ή συμβατότητας μεταξύ δύο τύπων, και κανόνες για τον προσδιορισμό του τύπου για κάθε έκφραση (expression) της γλώσσας εισόδου.

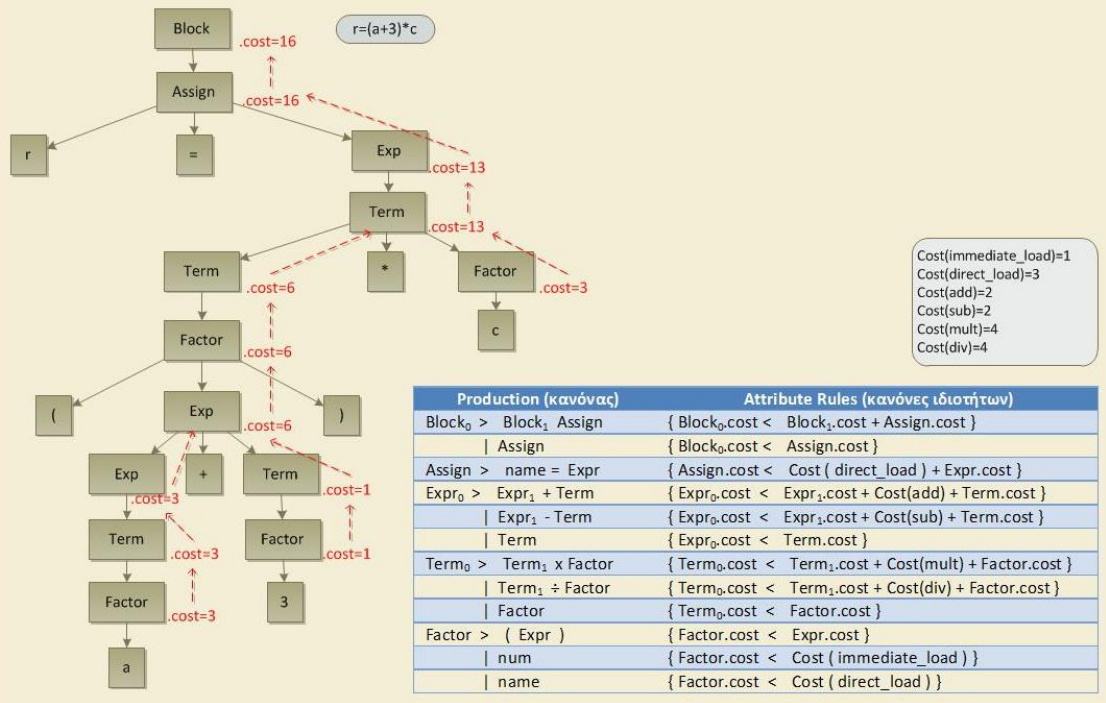
5.5.3 Attribute-Grammar Framework

Attribute Grammar Framework είναι ο φορμαλισμός για την υλοποίηση context-sensitive analysis δια μέσου μιας γραμματικής ιδιοτήτων (**attribute grammar**) η οποία και συνίσταται από μια CFG, προσαυξημένη από ένα σύνολο ενεργειών για τον προσδιορισμό των υπολογισμών επί συγκεκριμένων ιδιοτήτων. Οι ιδιότητες (**attributes**) των κανόνων της γραμματικής σε έναν μεταγλωττιστή συνήθως οργανώνονται σε δομές οι οποίες συσχετίζονται με το κάθε φύλλο ή κόμβο του parse-tree και συνεργάζονται με μία εξωτερική δομή (λίστα, πίνακα, κλπ). Για παράδειγμα το type system της γλώσσας επεξεργάζεται αυτές τις πληροφορίες (με κλήση των κατάλληλων ενεργειών) προκειμένου να αποδώσει συγκεκριμένες ιδιότητες του τύπου σε κάθε κόμβο δήλωσης ή έκφρασης του parse-tree. Μια CFG μαζί με ένα σύνολο κανόνων και ιδιοτήτων καλείται στη βιβλιογραφία και **Syntax-Directed Definition** ή **SDD** (Aho, Lam, Sethi, & Ullman, 2007). Επίσης μια SDD εκτός από υπολογισμούς μεταξύ ιδιοτήτων μπορεί να εκτελεί και άλλες ενέργειες, όπως εμφάνιση μηνυμάτων, καλούμενες και ως **side effects**. Μια SDD χωρίς side effect συχνά καλείται και **Attribute Grammar** ή **AG**.

Η κλήση των ενεργειών (του κάθε κανόνα) πραγματοποιείται με συγκεκριμένη σειρά για κάθε κόμβο (κανόνα γραμματικής) του parse-tree, ανάλογα με τον τύπο των ενεργειών αυτών. Γενικά μια post-order (Εικόνα 4-6) διαπέραση του parse-tree ακολουθεί στην ουσία την σειρά με την οποία δημιουργήθηκαν οι κόμβοι του δέντρου (shifts - reduces) κατά την διαδικασία του LR(1) bottom-up parsing.

5.5.3.1 Synthesized attributes

Αν οι ενέργειες των κανόνων (κόμβων) περιέχουν attributes των οποίων οι τιμές καθορίζονται μόνο από τις attributes του τρέχοντος κόμβου, των κόμβων παιδιών του και σταθερές, τότε οι ιδιότητες αυτές ονομάζονται συντιθέμενες (**Synthesized attributes**) και η εκτέλεσή τους μπορεί να πραγματοποιηθεί μέσω μιας κοινής post-order (bottom-up) διαπέρασης του parse-tree. Στο παράδειγμα (Εικόνα 5-7) δίνεται μέρος κανόνων γραμματικής για την αποτίμηση μιας απλής μαθηματικής έκφρασης όπου οι (Synthesized) κανόνες ιδιοτήτων υπολογίζουν σταδιακά (bottom-up) το κόστος υπολογισμού της έκφρασης.

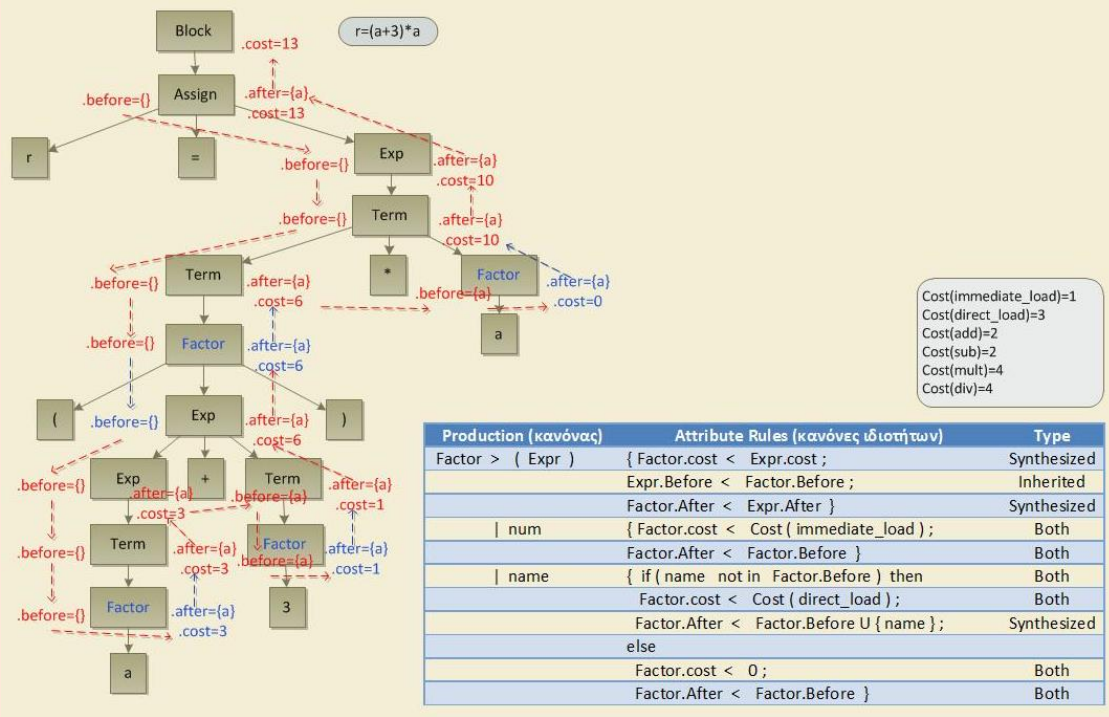


Production (κανόνας)	Attribute Rules (κανόνες ιδιοτήτων)
$Block_0$ > $Block_1$ Assign	{ $Block_0$.cost < $Block_1$.cost + Assign.cost }
| Assign	{ $Block_0$.cost < Assign.cost }
Assign > name = Expr	{ Assign.cost < Cost (direct_load) + Expr.cost }
$Expr_0$ > $Expr_1$ + Term	{ $Expr_0$.cost < $Expr_1$.cost + Cost(add) + Term.cost }
| $Expr_1$ - Term	{ $Expr_0$.cost < $Expr_1$.cost + Cost(sub) + Term.cost }
| Term	{ $Expr_0$.cost < Term.cost }
$Term_0$ > $Term_1$ x Factor	{ $Term_0$.cost < $Term_1$.cost + Cost(mult) + Factor.cost }
| $Term_1$ ÷ Factor	{ $Term_0$.cost < $Term_1$.cost + Cost(div) + Factor.cost }
| Factor	{ $Term_0$.cost < Factor.cost }
Factor > (Expr)	{ Factor.cost < Expr.cost }
| num	{ Factor.cost < Cost (immediate_load) }
| name	{ Factor.cost < Cost (direct_load) }

Εικόνα 5-7 : Παράδειγμα εφαρμογής (Synthesized) Attribute Grammar

5.5.3.2 Inherited attributes

Αν οι ενέργειες των κανόνων (κόμβων) περιέχουν attributes των οποίων οι τιμές καθορίζονται εξολοκλήρου από τις attributes του τρέχοντος κόμβου, των κόμβων συγγενών τους (αδέλφια) ή του γονικού κόμβου και σταθερές, τότε οι ιδιότητες αυτές ονομάζονται κληρονομούμενες (**Inherited attributes**) και η εκτέλεσή τους μπορεί να πραγματοποιηθεί μέσω μιας pre-order (top-down) διαπέρασης του parse-tree. Ωστόσο στους μεταγλωττιστές δεν έχει νόημα η παρουσία μόνο Inherited attributes στους κανόνες δεδομένου ότι συνήθως οι ιδιότητες μετακινούνται – διαδίδονται προς τους γονικούς κόμβους (bottom-up) και σε κάποιες περιπτώσεις προς τους συγγενικούς κόμβους ή κόμβους παιδιά (top-down) ανάλογα με την υλοποίηση.



Production (κανόνας)	Attribute Rules (κανόνες ιδιοτήτων)	Type
Factor > (Expr)	{ Factor.cost < Expr.cost ;	Synthesized
	Expr.Before < Factor.Before ;	Inherited
	Factor.After < Expr.After }	Synthesized
| num	{ Factor.cost < Cost (immediate_load) ;	Both
	Factor.After < Factor.Before }	Both
| name	{ if (name not in Factor.Before) then	Both
	Factor.cost < Cost (direct_load) ;	Both
	Factor.After < Factor.Before U { name } ;	Synthesized
	else	
	Factor.cost < 0 ;	Both
	Factor.After < Factor.Before }	Both

Εικόνα 5-8 : Παράδειγμα εφαρμογής (Inherited/Synthesized) Attribute Grammar

Τέλος αν οι ενέργειες των κανόνων (κόμβων) περιέχουν συνδυασμό Synthesized και Inherited attributes τότε η εκτέλεσή τους μπορεί να πραγματοποιηθεί μέσω μιας ειδικής διαπέρασης του parse-tree η οποία είναι συνδυασμός post-order και pre-order διαπέρασης και με την οποία θα ασχοληθούμε στην υλοποίησή μας. Στο παράδειγμα (Εικόνα 5-8) δίνεται μέρος κανόνων γραμματικής για την αποτίμηση μιας απλής μαθηματικής έκφρασης όπου οι κανόνες ιδιοτήτων υπολογίζουν σταδιακά το κόστος υπολογισμού της έκφρασης περνώντας ωστόσο τιμές στις ιδιότητες μεταξύ των κόμβων και προς τα πάνω (Synthesized attributes) και προς τα κάτω (Inherited attributes). Σε αυτή την παραλλαγή του παραδείγματος (Εικόνα 5-7), η attribute grammar εντοπίζει αν ένα όνομα μεταβλητής έχει ήδη φορτωθεί σε κάποιον καταχωρητή (απλοποιημένη προσέγγιση χωρίς έλεγχο των διαθέσιμων) μέσω των ιδιοτήτων συνόλων (After, Before), όπου οι τιμές του συνόλου After περνούν από τον Expr στον Factor (bottom-up) και του Before από τον Factor στον Expr (top-down). Το γεγονός (πληροφορία) ότι η μεταβλητή με όνομα a έχει χρησιμοποιηθεί (φορτωθεί) καταγράφεται και διαχέεται σε όλο το δέντρο μέσω των συνόλων after και before.

5.5.3.3 S-Attributed Definitions

Ένα σημαντικό στοιχείο μιας SDD, που δεν αναφέρθηκε προηγουμένως, είναι η σειρά αποτίμησης των ιδιοτήτων. Αν χρησιμοποιούμε τις synthesized και inherited ιδιότητες αυθαίρετα, υπάρχουν περιπτώσεις που μια SSD δεν μπορεί να εγγυηθεί μια συγκεκριμένη σειρά αποτίμησης των ιδιοτήτων ενός parse tree. Για παράδειγμα η περίπτωση synthesized και inherited ιδιοτήτων που ενημερώνουν τις ίδιες ιδιότητες μεταξύ ενός κόμβου και του παιδιού του (κυκλική αναφορά). Για το λόγο αυτό πρακτικά οι μεταγλωττιστές υλοποιούνται χρησιμοποιώντας συγκεκριμένες κλάσεις από SDD που εγγυούνται μια συγκεκριμένη σειρά αποτίμησης (Aho, Lam, Sethi, & Ullman, 2007). Οι SDDs που αποτελούνται μόνο από ιδιότητες synthesize, ορίζουν την κλάση των **S-Attributed Definitions**, όπου η σειρά αποτίμησης των κόμβων καθορίζεται από μια post order

διαπέραση, ανεξάρτητα από τη σειρά διαπέρασης των πεδίων (του κάθε κόμβου), εφόσον όλες οι synthesized ιδιότητες του κόμβου αποτιμούνται κατά την τελευταία (post) επίσκεψη σε αυτόν. Το παράδειγμα στην Εικόνα 5-7 αναφέρεται σε μία S-Attributed SDD εφόσον περιέχει μόνοι synthesized ιδιότητες.

5.5.3.4 L-Attributed Definitions

Μια άλλη πιο γενική κλάση SDDs είναι η **L-Attributed Definitions**, όπου αποτελούνται από ιδιότητες και των δύο τύπων synthesized και inherited, με τον περιορισμό ότι η αποτίμηση τους θα πραγματοποιείται από τα αριστερά προς τα δεξιά του δέντρου. Δηλαδή η διαπέραση των κόμβων παιδιών ενός κόμβου καθώς και η αποτίμηση των ιδιοτήτων τους, θα πραγματοποιείται από τον αριστερό προς τον δεξιό κόμβο παιδί. Πιο συγκεκριμένα κάθε ιδιότητα μπορεί να είναι :

- synthesized,
- inherited, με τον περιορισμό ότι για μία παραγωγή $A->X_1\ X_2\ \dots\ X_n$, με μία inherited ιδιότητα $X_i.a$ απομιμούμενη από ένα κανόνα της παραγωγής, ο κανόνας μπορεί να περιέχει
  - inherited ιδιότητες συσχετισμένες με το A,
  - synthesized ή inherited ιδιότητες συσχετισμένες με τα σύμβολα $X_1\ X_2\ \dots\ X_{i-1}$ στα αριστερά του X_i,
  - synthesized ή inherited ιδιότητες συσχετισμένες με το ίδιο το X_i με τέτοιο τρόπο ώστε να μην σχηματίζονται κυκλικές αναφορές από τις ιδιότητες του X_i

Το παράδειγμα στην Εικόνα 5-8, αναφέρεται σε μία L-Attributed SDD.

Γενικά, χρησιμοποιώντας το attribute-grammar framework, οι ιδιότητες των κόμβων (κανόνων) σε ένα grammar-tree, μπορούν να μεταφέρονται από κόμβο σε κόμβο και προς τις δύο κατευθύνσεις (top-down, bottom-up), δίνοντας τη δυνατότητα στο μεταγλωττιστή (ανάλογα με την υλοποίηση) να πραγματοποιεί context-sensitive analysis για διαφορετικά επιμέρους προβλήματα-στάδια της μεταγλώττισης (type checking, data flow analysis, register allocation, instruction scheduling, instruction selection, κ.λ.π.).

5.5.4 Ad Hoc Syntax-Directed Translation

Έχουμε αναφέρει ότι η post-order διαπέραση του parse-tree αντιστοιχεί στο bottom-up parsing του προγράμματος. Δηλαδή στην post-order διαπέραση, οι κόμβοι επισκέπτονται με την ίδια σειρά που δημιουργήθηκαν από τον parser, εφόσον κάθε κόμβος αντιστοιχεί σε ένα κανόνα (reduction) της γραμματικής και κάθε φύλο σε μία λέξη (token) της.

Συνεπώς αν σε κάθε κανόνα της γραμματικής του parser προσαρτήσουμε ένα σύνολο ενεργειών, αυτές θα εκτελεσθούν από τον parser ως να ήταν μια post-order διαπέραση του υπό κατασκευή parse-tree. Αυτή η τεχνική της προσάρτησης ενεργειών κατά το parsing είναι συνηθισμένη στους μεταγλωττιστές λόγω της ευκολίας και της ταχύτητας που παρέχει από το αρχικό στάδιο επεξεργασίας της εισόδου και ονομάζεται **Syntax-Directed Translation**. Ωστόσο οι ενέργειες των κανόνων της γραμματικής πρέπει να περιέχουν μόνο Synthesized attributes. Στην περίπτωση αυτή η context-sensitive analysis μπορεί να ενσωματωθεί και εκτελεσθεί παράλληλα με το parsing του προγράμματος. Η ανάγκη

παρουσίας Inherited attributes μπορεί να αντικατασταθεί (σε κάποιες περιπτώσεις) με τη χρήση μίας εξωτερικής δομής, όπως ο πίνακας συμβόλων, ή με αναδιατύπωση (διάσπαση) των κανόνων της γραμματικής.

Η επιλογή των ενεργειών και ιδιοτήτων και του τρόπου – σειράς που αυτές θα επιτελεσθούν σε κάθε κανόνα της γραμματικής ή κόμβο του parse-tree είναι καθαρά θέμα επιλογής υλοποίησης του μεταγλωττιστή. Κάποιοι μεταγλωττιστές υλοποιούν την context-sensitive analysis στον parser (Syntax-Directed Translation), άλλοι με μετέπειτα διαπέραση του parse-tree, άλλοι πάλι δεν κατασκευάζουν καθόλου parse-tree ή/και το μετατρέπουν σε abstract-tree με ανάλογες ιδιότητες. Στην υλοποίηση τη εργασίας θα ασχοληθούμε και με τους δύο τρόπους.

5.6 IR (Intermediate Representations)

Ένας τυπικός μεταγλωττιστής οργανώνει τη λειτουργία του σε ένα σύνολο από περάσματα. Σε κάθε πέρασμα αντλεί από τον κώδικα προς μεταγλώττιση γνώση την οποία και μεταφέρει από το ένα πέρασμα στο άλλο δια μέσου αναπαραστάσεων. Η αναπαράσταση που χρησιμοποιεί ο μεταγλωττιστής για τις πληροφορίες και στοιχεία που αντλεί από το πρόγραμμα καλείται Ενδιάμεση Αναπαράσταση (**Intermediate Representation** ή **IR**). Ένας μεταγλωττιστής μπορεί να χρησιμοποιεί μία ή περισσότερες IR για τον μετασχηματισμό του προγράμματος εισόδου σε εκτελέσιμο κώδικα. Ο μεταγλωττιστής δεν αναφέρεται στο αρχικό κείμενο του προγράμματος (εκτός της πρώτης φάσης) αλλά επεξεργάζεται και εξελίσσει την/τις IR του κώδικα μέχρι να επιτύχει το σκοπό του.

Γενικά οι IR κατατάσσονται σε τρις δομικές κατηγορίες : γραφικές (Graphical IRs), γραμμικές (Linear IRs) και υβριδικές (Hybrid IRs) και συνοπτικά παρουσιάζονται ακολούθως.

5.6.1 Graphical IRs

Οι **Graphical IRs** κωδικοποιούν τη γνώση του μεταγλωττιστή σε ένα γράφο (graph) και οι σχετικοί αλγόριθμοι εκφράζονται με τους αντίστοιχους όρους αντικειμένων όπως : nodes, edges, list ή trees. Το parse-tree και το abstract-tree (AST) είναι κλασικά παραδείγματα syntax-directed Graphical IRs, που χρησιμοποιούνται στους μεταγλωττιστές.

Η διαπέραση και επεξεργασία δομών γράφων–δέντρων καθώς και ο τρόπος αναπαράστασης και οργάνωσης των αντικειμένων τέτοιων δομών από συστατικά λογισμικού, καθορίζεται από πλευράς ποιότητας και απόδοσης από την φάση της σχεδίασης–μοντελοποίησης και την επιλογή των σχεδιαστικών προτύπων (design patterns) που εφαρμόζονται από τον σχεδιαστή του λογισμικού. Από την πλευρά της αρχιτεκτονικής λογισμικού, ο σχεδιασμός, χειρισμός και επεξεργασία αναπαραστάσεων γραφικών δομών σε ένα μεταγλωττιστή είναι καθοριστικής σημασίας δεδομένου ότι οι Graphical IRs, ανάλογα με την υλοποίηση, χρησιμοποιούνται σχεδόν σε κάθε φάση της μεταγλώττισης. Για τους προαναφερόμενους λόγους θα ασχοληθούμε εκτενώς με τις Graphical IRs στην υλοποίηση της εργασίας.

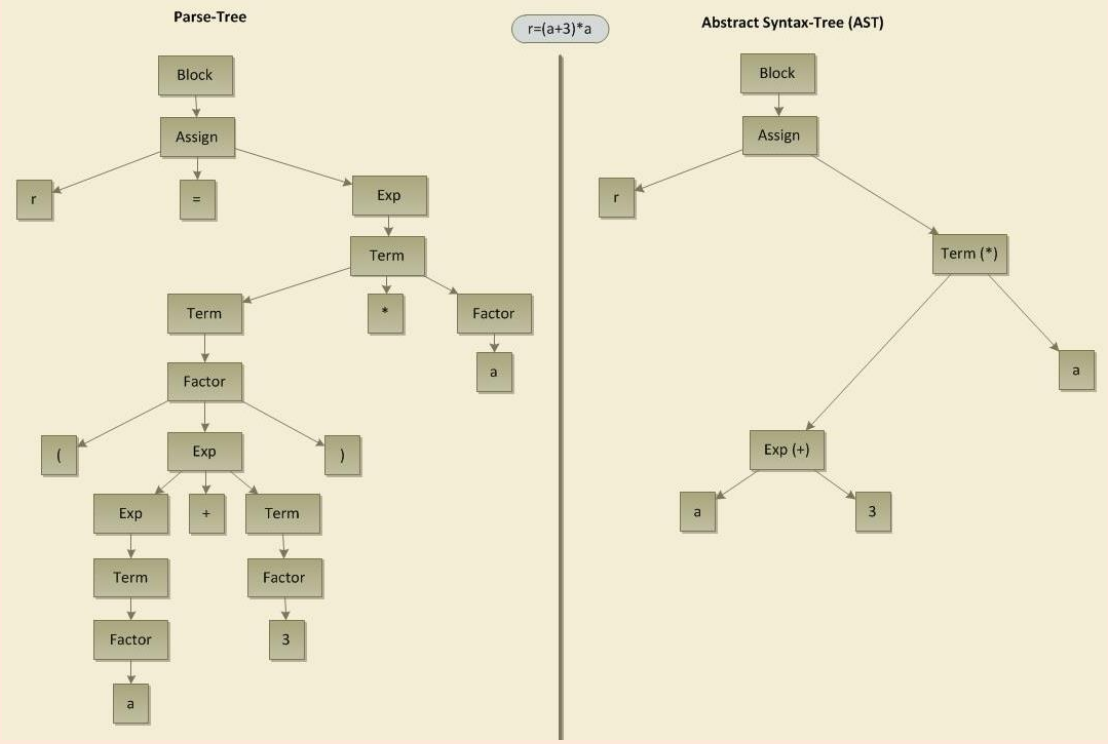
5.6.1.1 Parse-Tree

Το **parse-tree** είναι μια γραφική αναπαράσταση της derivation (ή parsing) ενός προγράμματος με βάση ένα σύνολο κανόνων μιας (CFG) γραμματικής. Συνήθως χαρακτηρίζονται ως syntax-related trees όπου κάθε κόμβος (node) αντιστοιχεί στην

εφαρμογή ενός κανόνα (reduce) της γραμματικής και κάθε φύλο σε μία λέξη (word, token) της γραμματικής. Στην Εικόνα 5-5 παρουσιάζεται ένα ενδεικτικό παράδειγμα pare-tree.

5.6.1.2 Abstract-Tree

Το abstract-tree ή **abstract syntax-tree** (**AST**) διατηρεί την βασική δομή και πληροφορία του parse-tree παραλείποντας ωστόσο του περιττούς κόμβους. Είναι μία κοντινή (near-source-level) αναπαράσταση της δομής του προγράμματος η οποία και μπορεί να παραχθεί απευθείας από τον parser του μεταγλωττιστή. Στην Εικόνα 5-9, παρουσιάζεται ένα ενδεικτικό παράδειγμα αντιστοιχίας μεταξύ των δύο αναπαραστάσεων.



Εικόνα 5-9 : Παράδειγμα αντιστοιχίας Parse-Tree, AST

5.6.1.3 Control-Flow, Data-Dependence, Call Graphs

Ένας **Control-Flow Graph** μοντελοποιεί τη ροή του ελέγχου μεταξύ των βασικών δομών (block) του προγράμματος που καθορίζονται από δηλώσεις επανάληψης, συνθήκες ελέγχου, εντολές αλμάτων (jumps), κ.λ.π. Ένας control-flow graph, έχει ένα κόμβο για κάθε βασικό block εντολών του προγράμματος εισόδου και μία ακμή (edge) για κάθε πιθανή μεταφορά του ελέγχου μεταξύ αυτών των blocks.

Ένας **Data-Dependence Graph** μοντελοποιεί τη ροή των τιμών από την δήλωσή τους (definition) μέχρι και τη χρήση τους σε οποιοδήποτε σημείο του προγράμματος. Οι κόμβοι του γραφήματος αναπαριστούν τις λειτουργίες ορισμών και χρήσης μίας τιμής και οι ακμές συνδέουν τον κόμβο ορισμού (της κάθε τιμής) με την κόμβο χρήσης αυτής της τιμής.

Ένας **Call Graph** μοντελοποιεί τις σχέσεις των κλήσεων μεταξύ των διαδικασιών (procedures) σε ένα πρόγραμμα. Κάθε κόμβος του γραφήματος αντιστοιχεί σε μια διαδικασία του προγράμματος και κάθε ακμή σε μια κλήση από μία διαδικασία σε άλλη. Η επεξεργασία μιας τέτοιας δομής συνεισφέρει στην **Interprocedural analysis** και εξετάζει την

αλληλεπίδραση μεταξύ πολλαπλών διαδικασιών, σε αντίθεση με την **Intraprocedural analysis** που περιορίζει την προσοχή της σε μία μόνο διαδικασία.

Οι τρις τελευταίοι τύποι Graphical IRs χρησιμοποιούνται κατά περίπτωση κυρίως στο τμήμα του Optimization του μεταγλωττιστή, ανάλογα με την υλοποίηση. Σαν δομές αναπαράστασης ακολουθούν παρόμοιες σχεδιαστικές αρχές με τα parse-trees, με ανάλογη εφαρμογή των σχεδιαστικών προτύπων.

5.6.2 Linear IRs

Οι **Linear IRs** είναι ένα είδος αναπαράστασης σε μορφή ψευδοκώδικα (pseudo-code) για μία αφηρημένη μηχανή. Οι σχετικοί αλγόριθμοι διαπερνούν απλές, γραμμικές ακολουθίες λειτουργιών. Οι στοιχειώδεις αυτές λειτουργίες είναι πολύ κοντά στις εντολές του εκτελέσιμου κώδικα αλλά η σύνταξη τους ακολουθεί μια γενική αλλά τυποποιημένη μορφή, ανεξάρτητη από τη μορφή του εκτελέσιμου κώδικα συγκεκριμένης μηχανής.

5.6.2.1 Three-Address Code

Ο **Three-Address Code** είναι pseudo-code που διαθέτει ένα σύνολο λειτουργιών (εντολών) με γενική μορφή **i ← j op k**, με έναν operator (op), δύο operands (j, k) και ένα αποτέλεσμα i. Κάποιες λειτουργίες χρησιμοποιούν λιγότερα ορίσματα, ωστόσο λειτουργίες με περισσότερα ορίσματα εκφράζονται ως ένα σύνολο λειτουργιών Three-Address Code, όπως συμβαίνει αντίστοιχα και σε επίπεδο εκτελέσιμου κώδικα.

5.6.2.2 ILOC

Έναν καλά ορισμένος Three-Address Code, αντίστοιχος με κώδικα assembly για μια αφηρημένη μηχανή, βρίσκεται στο παράρτημα A (Cooper & Torczon, 2012) της βιβλιογραφίας.

5.7 Symbol Table

Ο πίνακας συμβόλων (**symbol table**) είναι μια (τύπου IR) δομή που χρησιμοποιείται ευρέως στους μεταγλωττιστές. Αποθηκεύει τα ονόματα των συμβόλων (identifiers όπως ονόματα μεταβλητών, τύπων, διαδικασιών, συναρτήσεων, επικεφαλίδων, κ.λ.π.) του προγράμματος μαζί με διάφορες ιδιότητες (type attributes, scope attributes, κλπ) και ο μεταγλωττιστής τον χρησιμοποιεί σχεδόν σε όλες τις φάσεις της μεταγλώττισης. Ανάλογα με την υλοποίηση, συμβάλει στο scope checking, στο type checking, μπορεί να συνεργάζεται με τις λειτουργίες του attribute grammar framework (αποθηκεύοντας μέρος των ιδιοτήτων του), να συμμετέχει στο optimization των IRs, καθώς και να συμμετέχει στην ανάλυση των εξαρτήσεων (dependences) των τιμών μαζί ή και αντικαθιστώντας τον data-dependence graph. Στην πράξη αποτελεί ένα σύνδεσμο των συμβόλων του προγράμματος μεταξύ των δηλώσεων και των αναφορών τους (χρήση). Αποτελεί έναν αποδοτικό τρόπο για την αναζήτηση των δηλώσεων και τύπων (ιδιοτήτων) των συμβόλων χωρίς να απαιτείται η χρονοβόρα αναζήτηση (η διάχυση ιδιοτήτων) με διαπέραση του parse-tree.

5.7.1 Hashing with Linear Probing

Η υλοποίηση του symbol-table μπορεί να γίνει με διάφορους τρόπους και τύπους αναπαραστάσεων ανάλογα με την υλοποίηση και τις ιδιαιτερότητες της γλώσσας εισόδου. Ωστόσο κατά την σχεδίαση και ανάπτυξη ενός μεταγλωττιστή πέραν της επιδίωξης της ικανοποίησης των λειτουργικών προδιαγραφών, σημαντικό ρόλο έχει η ποιότητα του λογισμικού σε σχέση με τους πόρους (μνήμη) που καταναλώνει αλλά και την

αποδοτικότητά του (χρόνος εκτέλεσης). Ειδικά για τον πίνακα συμβόλων και εξαιτίας της ανάγκης πρόσβασης σε αυτόν από όλες σχεδόν τις φάσεις μεταγλώττισης, είναι σημαντικό να επιλεχθεί τέτοια δομή ώστε να εξασφαλίζει γρήγορη εισαγωγή και αναζήτηση των συμβόλων. Με δεδομένο ότι κατά τη φάση της λεκτικής και γραμματικής ανάλυσης τα σύμβολα εισάγονται στον πίνακα συμβόλων και διατηρούνται εκεί μέχρι που ο πίνακας να μην χρειάζεται πλέον, δεν υπάρχει ανάγκη για επιλεκτικές διαγραφές ή μετονομασία συμβόλων. Η παρατήρηση αυτή οδηγεί στην απλή προσέγγιση ενός πίνακα με σταθερό μέγεθος θέσεων, όπου η επιλογή της θέσης εισαγωγής (ή αναζήτησης) ενός στοιχείου προκύπτει από την επεξεργασία του αλφαριθμητικού του ονόματος του συμβόλου μέσω ενός hash αλγόριθμου (**hash function**[13]). Στην περίπτωση εισαγωγής στοιχείων, αν η θέση είναι κατειλημμένη από διαφορετικό σύμβολο (σύγκρουση), τότε αναζητείται γραμμικά το πρώτο κενό στοιχείο του πίνακα. Αν δεν βρεθεί κενή θέση τότε ο πίνακας είναι πλήρης. Αντίστοιχα στην αναζήτηση, αν δεν βρεθεί το σύμβολο στη αναμενόμενη θέση, τότε το σύμβολο αναζητείται γραμμικά μέχρι το πρώτο κενό στοιχείο ή την διαπέραση όλων των στοιχείων του πίνακα (αποτυχία αναζήτησης). Η τεχνική αυτή, γνωστή ως **hashing with linear probing**, αποδεικνύεται ότι στις περισσότερες περιπτώσεις (με αρκετά μεγάλο μέγεθος πίνακα και hash function με καλή κατανομή) παρέχει προσθήκη και αναζήτηση στοιχείων σε μέσο χρόνο **O(1)**[14] (Cooper & Torczon, 2012). Η απόδοση αυτή είναι εξαιρετική, ωστόσο έχει τους περιορισμούς ότι το μέγεθος του πίνακα πρέπει να είναι αρκετά μεγάλο κατά την δημιουργία του και δεν μπορεί να αλλάξει κατά το χρόνο εκτέλεσης (κάτι τέτοιο θα αχρήστευε τη λογική της εισαγωγής και αναζήτησης στοιχείων). Στην περίπτωση που ο πίνακας τείνει να γεμίσει (χειρότερη περίπτωση) ο χρόνος προσπέλασης πλησιάζει την γραμμική αναζήτηση O(n). Αν ο πίνακας γεμίσει κατά την εκτέλεση τότε πρέπει να αναδομηθεί από την αρχή σε έναν με μεγαλύτερο μέγεθος. Επίσης για τους ίδιους λόγους δεν είναι δυνατή η διαγραφή στοιχείων καθώς και η τροποποίηση του ονόματος ενός στοιχείου.

Γενικά ένας πίνακας συμβόλων πρέπει να έχει μια καλή συνάρτηση hash (h), η οποία να κάνει δίκαια κατανομή και έναν αποδοτικό μηχανισμό για την διαχείριση των συγκρούσεων (collisions) όταν η hash function επιστρέφει την ίδια θέση για διαφορετικό σύμβολο h(a)=h(b) για a≠b. Το παράρτημα B.4 (Cooper & Torczon, 2012) περιέχει μια εκτενή ανάλυση για την υλοποίηση πίνακα συμβόλων, hash functions και τη διαχείριση συγκρούσεων που προκύπτουν (π.χ. **Open hashing** (bucket hashing), **Open addressing** (rehashing)).

Στην υλοποίηση της εργασίας επιλέγουμε μία σύνθετη (γραφική) δομή για τον πίνακα συμβόλων, προκειμένου να αποτυπώσει την πολυεπίπεδη δομή των scopes και των name spaces της γλώσσας εισόδου, εφαρμόζοντας παράλληλα κατάλληλα σχεδιαστικά πρότυπα.

5.8 Optimization & Back-End

Όπως έχει ήδη αναφερθεί τόσο ο optimizer όσο και το back-end του μεταγλωττιστή ασχολείται με την βελτίωση της IR και τελικά την παραγωγή του εκτελέσιμου κώδικα μέσα

[13] Μια συνάρτηση hash δέχεται ως όρισμα το αλφαριθμητικό (string) του ονόματος ενός συμβόλου και επιστρέφει μια ακέραια τιμή εντός συγκεκριμένων ορίων ή έναν δείκτη σε πίνακα. Σκοπός της είναι η κατά το δυνατό δικαιότερη κατανομή των επιστρεφόμενων τιμών σε σχέση με το αλφαριθμητικό εισόδου.

[14] Η γραμμική αναζήτηση έχει χρόνο απόκρισης O(n) και η δυαδική $O(\log_2 n)$, όπου n ο αριθμός των συνολικών στοιχείων. O(1) σημαίνει σταθερός χρόνος αναζήτησης ανεξάρτητα από το σύνολο των στοιχείων.

από μια σειρά περασμάτων, πραγματοποιώντας λειτουργίες ανάλυσης όπως data flow analysis, register allocation, instruction selection, instruction scheduling, κλπ. Οι λειτουργίες αυτές επιδρούν πάνω σε διάφορους κατά περίπτωση τύπους IR. Ο σημαντικότερος από άποψη σχεδίασης και διαχείρισης τύπος αναπαράστασης είναι οι graphic IRs, η σχεδίαση και διαχείριση των οποίων (από άποψη δομής) είναι παρεμφερής, με δυνατότητα εφαρμογής ανάλογων σχεδιαστικών προτύπων. Το attribute-grammar framework μπορεί να εφαρμοσθεί (επί των graphic IRs) με κατάλληλες ιδιότητες για την υλοποίηση σχεδόν όλων των προαναφερόμενων αναλύσεων.

6 Flex & Bison

6.1 Γενικά

Λόγω του φορμαλισμού και της τυποποίησης που παρέχει η θεωρία της επιστήμης των υπολογιστών σχετικά με την ακριβή αναπαράσταση λεκτικών εκφράσεων μέσω των Regular Expressions και συντακτικών εκφράσεων (κανόνων) μέσω των Context Free Grammars, είναι δυνατή η ανάπτυξη έτοιμων προγραμμάτων λεκτικών (scanners) και συντακτικών (parsers) αναλυτών.

Για τις ανάγκες της υλοποίησης της εργασίας χρησιμοποιήθηκε το ευρέως διαδεδομένο έτοιμο λογισμικό Flex και Bison το οποίο έχει αναπτυχθεί από το University of California, Berkeley και διατίθεται ελεύθερα στο σύνδεσμο http://gnuwin32.sourceforge.net/, συνοδευόμενο από αντίστοιχη τεκμηρίωση (Paxson, 1995), (Donnelly & Stallman, 2008). Παρακάτω γίνεται μια συνοπτική περιγραφή του λογισμικού επικεντρωμένη στα σημεία που αφορούν την συγκεκριμένη υλοποίησης της εργασίας.

6.2 Flex scanner

Ο **Flex** είναι ένα λογισμικό, το οποίο δεχόμενο ως είσοδο ένα αρχείο εκφράσεων (regular expressions, RE) παράγει τον κώδικα (C/C++) του scanner ή του λεκτικού αναλυτή (ενότητα 5.3), για την συγκεκριμένη γλώσσα που περιγράφεται από τις RE. Η εκτέλεση του παραχθέντος scanner δέχεται ως είσοδο ένα (κείμενο) πηγαίο πρόγραμμα και επιστρέφει την ακολουθία (με τη σειρά εμφάνισης στο κείμενο) των λέξεων (tokens) του προγράμματος που αναγνωρίζονται με βάση τις αρχικές RE. Οι RE συντάσσονται από τον χρήστη με βάση το πρότυπο προδιαγραφής της γλώσσας για την οποία και υλοποιούμε την λεκτική ανάλυση. Τα token επιστρέφονται κατάλληλα κωδικοποιημένα (ακέραια τιμή) για χρήση από το συνεργαζόμενο λογισμικό Bison. Παράλληλα δίνεται η δυνατότητα κατά την αναγνώριση ενός token να εκτελείται κώδικας C++, ο οποίος εισάγεται από τον χρήστη δίπλα από κάθε RE και μεταφέρεται αυτούσιος στον κώδικα του παραγόμενου scanner. Το αρχείο που περιέχει τις RE, τον αντίστοιχο κώδικα ενεργειών καθώς και ένα σύνολο οδηγιών και συμβόλων για την παραμετροποίηση και συμπεριφορά του scanner έχει συγκεκριμένη δομή και αποθηκεύεται με όνομα *.l

Για την αρχική παραμετροποίηση, τις βασικές εντολές και γενικά την σε βάθος κατανόηση του Flex κρίνεται απαραίτητη η μελέτη της τεκμηρίωσης που το συνοδεύει (Paxson, 1995), καθώς και της σχετικής με τη χρήση του βιβλιογραφίας (Levine, 2009).

6.2.1 Regular expression (Patterns)

Όπως έχει αναφερθεί στην ενότητα 5.3, μία RE είναι μια σειρά από χαρακτήρες που περιγράφει ένα πρότυπο (pattern) αναζήτησης για λειτουργίες όπως το ταίριασμα προτύπων (pattern matching) σε αλφαριθμητικά. Αν και τυπικά μια RE περιγράφεται από αλγεβρικούς τύπους και κανόνες (Aho & Ullman, 1995), για τον Flex ορίζεται μια παρεμφερή με τη βιβλιογραφία σημειογραφία (Πίνακας 6-1, (Paxson, 1995)).

Πίνακας 6-1 : Flex Regular Expression Patterns

Pattern	Περιγραφή
‘x’	ταιριάζει τον χαρακτήρα ‘x’
‘.’	οποιονδήποτε χαρακτήρα (byte) εκτός της νέας γραμμής
‘[xyz]’	μια κλάση χαρακτήρων, π.χ. ‘x’ ή ‘y’ ή ‘z’
‘[abj-oZ]’	μια κλάση χαρακτήρων με μια περιοχή, π.χ. ‘a’ ή ‘b’ ή οποιονδήποτε χαρακτήρα από ‘j’ μέχρι ‘o’, ή ‘Z’
‘[^A-Z]’	μια αρνητική κλάση χαρακτήρων, π.χ. οποιοσδήποτε χαρακτήρας εκτός των χαρακτήρων από ‘A’ έως ‘Z’
‘[^A-Z\n]’	οποιοσδήποτε χαρακτήρας εκτός των χαρακτήρων από ‘A’ έως ‘Z’ ή της νέας γραμμής
‘r*’	καμία ή περισσότερες r’s, όπου r οποιαδήποτε RE
‘r+’	μία ή περισσότερες r’s
‘r?’	καμία ή μία r’s
‘r{2,5}’	2 έως 5 επαναλήψεις της r
‘r{2,}’	2 ή περισσότερες επαναλήψεις της r
‘r{4}’	ακριβώς 4 επαναλήψεις της r
‘{name}’	η ορισμένη RE με όνομα name (βοηθητική ενδιάμεση δήλωση)
‘”[yyz]\”foo”’	το αλφαριθμητικό: ‘[yyz]\”foo’
‘\x’	αν x={‘a’,’b’,’f’,’n’,’r’,’t’,’v’}, τότε η ANSI-C αντιστοιχία του \x, διαφορετικά ο χαρακτήρας x (χρησιμοποιείται για τους operators, όπως ‘*’,’+’,’?’,’\’, κλπ)
‘\0’	ο χαρακτήρας NUL (ASCII κώδικας 0)
‘\123’	ο χαρακτήρας με οκταδική τιμή 123
‘\x2a’	ο χαρακτήρας με δεκαεξαδική τιμή 2a
‘(r)’	ταιριάζει την RE r (χρησιμοποιείται για την παράκαμψη της προτεραιότητας των τελεστών)
‘rs’	η RE r ακολουθούμενη από RE s (concatenation)
‘r|s’	ή τη r ή την s
‘r/s’	την r μόνο εφόσον ακολουθείται από την s (μόνο η r επιστρέφεται, η s παραμένει στην ουρά)
‘^r’	την r αλλά μόνο στην αρχή μιας γραμμής
‘r$’	την r αλλά μόνο στο τέλος μιας γραμμής
‘<s>r’	την r αλλά μόνο για την κατάσταση εκκίνησης s
‘<*>r’	την r αλλά σε οποιαδήποτε κατάσταση εκκίνησης
‘<<EOF>>’	το τέλος του αρχείου

Τα ονόματα name είναι συμβολικά ονόματα ενδιάμεσων εκφράσεων προκειμένου να απλοποιούνται οι σύνθετες εκφράσεις. Ένα name αρχίζει πάντα από χαρακτήρα ή ‘_’ ακολουθούμενο από ένα ή περισσότερα ψηφία, γράμματα, ‘_’ και ‘-’. Οι καταστάσεις εκκίνησης είναι ένας μηχανισμός για την υπό συνθήκη ενεργοποίηση εκφράσεων. Οι καταστάσεις ενεργοποιούνται και αλλάζουν από τις εκφράσεις και μια έκφραση μπορεί να ενεργοποιείται μόνο για συγκεκριμένες καταστάσεις. Για περισσότερες πληροφορίες ανατρέξτε στην σχετική τεκμηρίωση.

6.2.2 Tokens and Values

Ο χρήστης μπορεί για κάθε RE να αντιστοιχεί μια ενέργεια, διατυπωμένη με κώδικα C++. Το περιβάλλον παρέχει μια σειρά από μεταβλητές για χρήση εντός αυτών των ενεργειών (Πίνακας 6-2). Οι τιμές αυτές μπορούν να χρησιμοποιηθούν και από

συνεργαζόμενα λογισμικά (όπως το λογισμικό Bison) καθώς και από κώδικα άλλων modules.

Πίνακας 6-2 : Ονόματα τιμών περιβάλλοντος flex (βασικά)

Values	Περιγραφή
yytext	περιέχει το αλφαριθμητικό του τρέχοντος token (λέξης)
yyleng	περιέχει το μήκος του τρέχοντος token
yyin	δείκτης στο file stream εισόδου
yyout	δείκτης στο file stream εξόδου

6.3 Bison parser

Ο **Bison** είναι ένα λογισμικό, το οποίο δεχόμενο ως είσοδο ένα αρχείο κανόνων γραμματικής (context-free grammar, CFG) παράγει τον κώδικα (C/C++) του parser ή του συντακτικού αναλυτή (ενότητα 5.4), για την συγκεκριμένη γλώσσα που περιγράφεται από τους CFG rules. Η εκτέλεση του παραχθέντος parser δέχεται ως είσοδο τις λέξεις (tokens) του πηγαίου προγράμματος που παράγονται από τον λεκτικό αναλυτή (scanner) και κατασκευάζει (αν καταστεί δυνατό) μια derivation με βάση την αρχική CFG. Ποιο συγκεκριμένα με βάση μια τύπου LR(1) Grammar (ενότητα 5.4.2), επεξεργάζεται τα token από το λογισμικό Flex και δομεί μια derivation από κάτω προς τα πάνω (bottom-up parsing, ενότητα 5.4.4) για το πρόγραμμα εισόδου.

Αναλυτικότερα ο Bison αν και μπορεί να χειριστεί όλες σχεδόν τις CFGs, είναι βελτιστοποιημένος στον χειρισμό των καλούμενων **LALR(1) grammars**. Στην ουσία πρόκειται για τις LR(1) grammars με κάποιους επιπλέον περιορισμούς, ωστόσο στην πράξη είναι σπάνιο μια LR(1) grammar να μην είναι και LALR(1) grammar. Συνοπτικά μια τέτοια γραμματική είναι σε θέση να αναλύσει οποιαδήποτε ακολουθία εισόδου χαρακτήρων (string) μόνο με την ανάλυση ενός προπορευόμενου (lookahead) token.

- Οι parsers για τις LALR(1) grammars είναι **deterministic**, που σημαίνει ότι ο επόμενος προς εφαρμογή κανόνας σε οποιοδήποτε σημείο της εισόδου είναι μοναδικά επιλέξιμος με βάση την τρέχουσα είσοδο και μία σταθερή, πεπερασμένη δέσμη (καλούμενη lookahead) της εναπομένουσας εισόδου. Αυτή η ιδιότητα συνεπάγεται ότι είναι και **unambiguous**, που σημαίνει ότι υπάρχει μόνο ένας τρόπος (derivation) για την εφαρμογή των κανόνων της γραμματικής για μια είσοδο (ακολουθία χαρακτήρων).
- Ωστόσο μια CFG μπορεί να είναι **ambiguous**, που σημαίνει ότι υπάρχουν πολλαπλοί τρόποι (derivations) για την εφαρμογή των κανόνων της γραμματικής για την ίδια είσοδο (ακολουθία χαρακτήρων). Ακόμα και μία **unambiguous** γραμματική μπορεί να είναι **nondeterministic**, που σημαίνει ότι δεν υπάρχει σταθερό loohkahead που να καθορίζει με σαφήνεια τον επόμενο προς εφαρμογή κανόνα.

Ο Bison (με κατάλληλες ρυθμίσεις) μπορεί να χειριστεί οποιαδήποτε CFG για τις οποίες, ο αριθμός των πιθανών τρόπων (derivations) εφαρμογής των κανόνων της γραμματικής για οποιαδήποτε είσοδο, είναι πεπερασμένος. Αυτό το επιτυγχάνει με μία τεχνική που καλείται GLR parsing (Generalized LR). Στη υλοποίηση της εργασίας θα

ασχοληθούμε με το LR(1) top-down parsing που είναι σαφώς ποιο αποδοτικός αλγόριθμος συντακτικής ανάλυσης.

Οι κανόνες της γραμματικής (σε μορφή LALR(1) Grammar) συντάσσονται από τον χρήστη με βάση το πρότυπο προδιαγραφής της γλώσσας για την οποία και υλοποιούμε την συντακτική ανάλυση. Τα token εισάγονται (shifting) κατάλληλα κωδικοποιημένα (ως ακέραια τιμή) από το συνεργαζόμενο λογισμικό Flex. Παράλληλα δίνεται η δυνατότητα κατά την αναγνώριση/εφαρμογή (reduction) ενός κανόνα να εκτελείται κώδικας C/C++, ο οποίος εισάγεται από τον χρήστη δίπλα από κάθε κανόνα και μεταφέρεται αυτούσιος στον κώδικα του παραγόμενου parser. Το αρχείο που περιέχει του κανόνες της CFG, τον αντίστοιχο κώδικα ενεργειών καθώς και ένα σύνολο οδηγιών και συμβόλων για την παραμετροποίηση και συμπεριφορά του parser έχει συγκεκριμένη δομή και αποθηκεύεται με όνομα *.y

Για την αρχική παραμετροποίηση, τις βασικές εντολές και γενικά την σε βάθος κατανόηση του Bison κρίνεται απαραίτητη η μελέτη της τεκμηρίωσης που το συνοδεύει (Donnelly & Stallman, 2008), καθώς και της σχετικής με τη χρήση του βιβλιογραφίας (Levine, 2009).

6.3.1 Terminal & Nonterminal Symbols

Τα terminal symbols της γραμματικής είναι τα tokens που επιστρέφονται από τον scanner (κωδικοποιημένη ακεραία τιμή) και μπορούν να αναπαρίστανται στους κανόνες της γραμματικής με τρις διαφορετικούς τρόπους : α) ως αναγνωριστικό (named token) με την ειδική δήλωση %token, β) ως χαρακτήρες (character token type) με απευθείας χρήση σε ένα κανόνα όπως ‘+’, γ) ως αλφαριθμητικό (literal string token) με απευθείας χρήση σε ένα κανόνα όπως “<=”.

Τα nonterminal symbols είναι ομαδοποιήσεις άλλων nonterminal ή/και terminal symbols συνθέτοντας κατά αυτόν τον τρόπο τους κανόνες της γραμματικής. Τα συμβολικά ονόματα μπορούν να περιέχουν χαρακτήρες και ψηφία (εκτός του πρώτου), παύλες και κάτω παύλες.

6.3.2 Rule syntax

Οι κανόνες της γραμματικής ακολουθούν την παρακάτω γενική μορφή (Εικόνα 6-1). Αν το σύμβολο του αποτελέσματος εμφανίζεται και στην δεξιά πλευρά του κανόνα τότε ο κανόνας καλείται αναδρομικός (recursive rule).

```
result:    rule1-components...
         | rule2-components...
         ...
         ;
```

Εικόνα 6-1 : Γενική μορφή κανόνα γραμματικής (Bison)

Το σύνολο των κανόνων αποτελούν τη γραμματική της γλώσσας, και προκειμένου αυτή να είναι μια LR(1) grammar θα πρέπει να είναι deterministic και συνεπώς unambiguous. Σε κάθε κανόνα θα πρέπει να ορίζεται με σαφήνεια ο συσχετισμός (**associativity**) καθώς και προτεραιότητα (**precedence**) των συστατικών του (rule components). Αυτό επιτυγχάνεται είτε με κατάλληλη σύνταξη των κανόνων ώστε να προκύπτει από την ίδια την ιεράρχησή τους, είτε με κατάλληλες δηλώσεις (%left, %right, %nonoassoc) ορισμού συσχετισμού και

ιεράρχησης για κάθε symbol της γραμματικής. Γενικά το ίδιο το λογισμικό (bison) μας προειδοποιεί για τις αστοχίες της γραμματικής με κατάλληλα μηνύματα (shit/reduce[15] ή reduce/reduce conflicts) μέχρι αυτή να είναι μια αποδεκτή LR(1) γραμματική.

Πίνακας 6-3 : Οδηγίες Bison για τον ορισμό της Γραμματικής

Οδηγίες	Περιγραφή
%union	ορίζει τη συλλογή των τύπων δεδομένων των σημασιολογικών ιδιοτήτων (semantic value) που μπορεί να έχει ένα σύμβολο της γραμματικής
%token	ορίζει ένα τερματικό σύμβολο (terminal symbol) της γραμματικής χωρίς προκαθορισμένο συσχετισμό (associativity) και προτεραιότητα (precedence)
%right	ορίζει ένα τερματικό σύμβολο (terminal symbol) της γραμματικής με προκαθορισμένο αριστερό συσχετισμό (right-associative)
%left	ορίζει ένα τερματικό σύμβολο (terminal symbol) της γραμματικής με προκαθορισμένο δεξιό αριστερό συσχετισμό (left-associative)
%nonassoc	ορίζει ένα τερματικό σύμβολο (terminal symbol) της γραμματικής ως μη συσχετίσημο (non associative)
%type	ορίζει τον τύπο σημασιολογικών ιδιοτήτων (semantic value) για ένα μη τερματικό σύμβολο (nonterminal symbol)
%start	ορίζει το μη τερματικό σύμβολο (nonterminal symbol) που αντιστοιχεί στο σύμβολο εκκίνησης (start symbol) της γραμματικής
%expect	ορίζει τον αριθμό των αναμενόμενων – αποδεκτών διενέξεων shift-reduce της γραμματικής

Γενικά ο Bison διαθέτει ένα σύνολο από οδηγίες με την χρήση των οποίων, ορίζονται όλες οι παράμετροι σχετικά με τις συσχετισμούς και τις προτεραιότητες των συμβόλων στους κανόνες, τον ορισμό των τύπων δεδομένων των σημασιολογικών ιδιοτήτων (semantic value) του κάθε συμβόλου. κλπ. Ο Πίνακας 6-3 περιέχει μια σύντομη περιγραφή των βασικότερων οδηγιών του bison για τον ορισμό της γραμματικής.

6.3.3 Semantic Values

Ο Bison υποστηρίζει ένα μηχανισμό για απόδοση ιδιοτήτων σε κάθε συστατικό των κανόνων του προκειμένου να είναι δυνατή η σημασιολογική ανάλυση και διαχείριση των συμβόλων του κάθε κανόνα. Σε κάθε σύμβολο αντιστοιχείται μια ιδιότητα (YYSTYPE, εξορισμού ακέραιος αριθμός) ή οποία μπορεί ωστόσο να ορισθεί από τον χρήστη ως μια δομή πολλαπλών ιδιοτήτων. Σε κάθε σύμβολο μπορεί να ορισθεί ο τύπος των ιδιοτήτων του με χρήση της οδηγίας %type. Αυτές οι ιδιότητες αποτελούν για τη γραμματική ένα πρωτόγεννες Attribute-Grammar Framework (ενότητα 5.5.3) για την υλοποίηση context sensitive analysis κατά τη συντακτική ανάλυση του προγράμματος από τον bison ή αλλιώς ad-hoc syntax directed translation (ενότητα 5.5.4).

[15] Σε κάποιες περιπτώσεις ενδέχεται κάποια shift/reduce conflicts να είναι αποδεκτά, σε αυτήν την περίπτωση τα σχετικά προειδοποιητικά μηνύματα μπορούν να απενεργοποιηθούν με την οδηγία %expect.

6.3.4 Semantic Actions

Η πρόσβαση στις ιδιότητες των συστατικών του κάθε κανόνα πραγματοποιείται μέσω ενός ειδικού συμβολισμού αναφοράς (π.χ. $$ για το αποτέλεσμα, $1 για το πρώτο σύμβολο του κανόνα κ.ο.κ.). Ο κώδικας C/C++ που επιδρά στις ιδιότητες εισάγεται στο τέλος κάθε κανόνα εντός των συμβόλων '{…}' και εκτελείται κατά την επιλογή (reduction) του κανόνα από τον parser.

6.3.4.1 Mid-Rule Actions

Κώδικας C/C++ που επιδρά στις ιδιότητες μπορεί να εισαχθεί, εντός των συμβόλων '{…}', και ανάμεσα στα συστατικά στοιχεία του κάθε κανόνα. Σε αυτή την περίπτωση το κάθε τμήμα ενδιάμεσου κώδικα θα εκτελείται κατά την είσοδο του τελευταίου token του προηγούμενου τμήματος και με την προϋπόθεση ότι το επόμενο token καθορίζει μονοσήμαντα ότι τελικά ο συγκεκριμένος κανόνας πρόκειται να γίνει τελικά reduction. Η προϋπόθεση αυτή είναι δύσκολο να διασφαλισθεί και απαιτεί πολύ προσεκτική σύνταξη του κανόνα χωρίς να είναι πάντα εφικτό και διαχειρήσιμο. Πολλές φορές η λύση είναι η διάσπαση του κανόνα σε επιμέρους, έτσι ώστε τα ενδιάμεσα τμήματα κώδικα του αρχικού κανόνα να μετατραπούν σε τερματικά τμήματα κώδικα των παραγόμενων κανόνων.

6.3.5 Locations

Μια εξορισμού ιδιότητα που διατηρεί και ενημερώνει ο bison είναι η τοποθεσία (**location**) του κάθε συμβόλου (τερματικού και μη) σε σχέση με το αρχείο εισόδου του προγράμματος. Η εξ ορισμού δομή που αναπαριστά την θέση ενός συμβόλου παρουσιάζεται παρακάτω (YYLTYPE, Εικόνα 6-2), ωστόσο μπορεί να ορισθεί και από τον χρήση. Η πρόσβαση στις ιδιότητες της τοποθεσίας του κάθε κανόνα πραγματοποιείται μέσω ενός ειδικού συμβολισμού αναφοράς (π.χ. @$ για το αποτέλεσμα, @1 για το πρώτο σύμβολο του κανόνα κ.ο.κ.).

```
typedef struct YYLTYPE
{
  int first_line;
  int first_column;
  int last_line;
  int last_column;
} YYLTYPE;
```

Εικόνα 6-2 : Εξ ορισμού ιδιότητα Location του Bison

Οι τιμές της ιδιότητας location αρχικοποιούνται για κάθε token από τον scanner και περνούν στον parser, όπου με βάση μια σειρά εξ ορισμού ενεργειών, ενημερώνονται αυτόματα για κάθε σύμβολο της γραμματικής με βάση τον κανόνα που εκτελείται.

6.3.6 Building Parse-Tree

Χρησιμοποιώντας τις semantic values εντός των semantic actions ως ένα πρωτογενές attribute grammar framework σε συνδυασμό με μια δομή αντικειμένων γραφικής αναπαράστασης (graphic IR) που ορίζει ο χρήστης για κάθε σύμβολο (ή ομάδα συμβόλων) της γραμματικής, ο bison ως bottom-up parser (ενότητα, 5.4.4), είναι σε θέση να δομήσει το parse tree (ενότητα, 5.6.1.1) ή αλλιώς την derivation του προγράμματος εισόδου. Όπως έχει αναφερθεί οι semantic actions θα πρέπει αυστηρά να περιέχουν μόνο synthesized attributes λόγω της bottom-up κατεύθυνσης κατασκευής του δέντρου (ενότητα, 5.5.3).

6.3.7 Calling Pure Parsers

Εξ ορισμού ο Bison παράγει έναν parser με στατικές μεταβλητές και δομές επιτρέποντας την εκτέλεση μόνο ενός threat συντακτικής ανάλυσης να εκτελείται σε ένα πρόγραμμα σε οποιαδήποτε χρονική στιγμή. Ωστόσο είναι δυνατό με κατάλληλη παραμετροποίηση, να παράγει έναν επαναχρησιμοποιούμενο (**pure** ή **reentrant**) parser, με δυναμικές μεταβλητές και δομές (τύπου αντικειμένων) ο ποίος και μπορεί να εκτελείται σε πολλαπλά threats ταυτόχρονα για διαφορετικά ή και το ίδιο αρχείο/α εισόδου.

6.3.8 Parser Interface

Ο parser που δημιουργείται από τον bison για τον χρήση είναι μια C συνάρτηση με όνομα yyparse. Η κλήση της προκαλεί την εκτέλεση της διαδικασίας του bottom-up parsing με βάση την οριζόμενη CFG, αλληλεπιδρώντας με τον scanner για την άντληση των tokens μέσω των global μεταβλητών yylval, yylloc. Ο τερματισμός της συνάρτησης επιστρέφει α) τιμή 0 αν η συντακτική ανάλυση ήταν επιτυχής (βρέθηκε derivation) και ολοκληρώθηκε η είσοδος, β) τιμή 1 αν η συντακτική ανάλυση απέτυχε λόγω μη έγκυρης εισόδου (δεν βρέθηκε derivation) και γ) τιμή 2 αν η συντακτική ανάλυση διακόπηκε λόγω εξάντλησης της μνήμης του συστήματος. Κατά την κλήση της συνάρτησης, εξ ορισμού, δεν υπάρχουν παράμετροι, ωστόσο υπάρχει η δυνατότητα να ορίσει ο χρήστης παραμέτρους μέσω της δήλωσης %parse-param.

Τέλος το αποτέλεσμα της συντακτικής ανάλυσης (συνήθως ένα parse-tree, ή/και symbol table) παράγεται από τον κώδικα των semantics actions των κανόνων κατά την εκτέλεσή τους.

Πίνακας 6-4 : Οδηγίες Bison για τον καθορισμό συμπεριφοράς

Οδηγίες	Περιγραφή
%code {code}	εισάγει αυτούσιο τον περιλαμβανόμενο κώδικα στο αρχείο του πηγαίου κώδικα του parser αμέσως μετά το περιεχόμενο του αρχείου επικεφαλίδων του parser
%code requires {code}	εισάγει αυτούσιο τον περιλαμβανόμενο κώδικα στο αρχείο του πηγαίου κώδικα καθώς και στο αρχείο των επικεφαλίδων του parser, πριν την παραγωγή των YYSTYPE και YYLTYPE ορισμών (προσφέρεται για δηλώσεις που χρησιμοποιούνται από την %union οδηγία καθώς και για τον επαναορισμό των YYSTYPE και YYLTYPE)
%code provides {code}	εισάγει αυτούσιο τον περιλαμβανόμενο κώδικα στο αρχείο του πηγαίου κώδικα καθώς και στο αρχείο των επικεφαλίδων του parser, μετά την δημιουργία των YYSTYPE, YYLTYPE και token ορισμών
%code top {code}	εισάγει αυτούσιο τον περιλαμβανόμενο κώδικα στο αρχείο του πηγαίου κώδικα του parser, κοντά στην κορυφή του αρχείου
%debug	ορίζει το YYDEBUG macro στην τιμή 1, προκειμένου ο σχετικός με την αποτελμάτωση κώδικα να συμμετάσχει στη μεταγλώττιση
%define variable “value”	ορίζει μια μεταβλητή για τον καθορισμό της συμπεριφοράς του bison

Οδηγίες	Περιγραφή
%defines	δημιουργεί ένα αρχείο επικεφαλίδων (*. h) που περιέχει τους ορισμούς των token names της γραμματικής καθώς και ένα σύνολο απαραίτητων δηλώσεων όπως YYSTYPE, YYLTYPE, yylval, yylloc, κ.λ.π.
%file-prefix “prefix”	καθορίζει το όνομα για όλα τα αρχεία εξόδου του bison (θεωρώντας ως όνομα του αρχείου εισόδου prefix.y)
%language “language”	καθορίζει την γλώσσα προγραμματισμού για τον παραγόμενο parser μεταξύ των επιλογών C, C++, Java
%locations	παράγει τον κώδικα για την επεξεργασία των locations (ενεργοποιείται με την χρήση της αναφοράς @$ εντός των semantic actions)
%name-prefix “prefix”	μετονομάζει τα εξωτερικά σύμβολα που χρησιμοποιούνται από τον parser προκειμένου το όνομα τους να ξεκινά με prefix αντί για yy (πχ. yyparse, yylex, yyerror, yynerrs, yylval, yylloc, yychar, yydebug)
%no-lines	αποτρέπει την παραγωγή των #line εντολών του προεπεξεργαστή στο παραγόμενο αρχείο του parser
%output “file”	καθορίζει το όνομα αρχείου του parser
%pure-parser	ορίζει την παραγωγή pure parser
%require “version”	καθορίζει την ελάχιστη έκδοση του λογισμικού bison που απαιτείται για την παραγωγή του parser
%skeleton “file”	καθορίζει ένα ειδικού τύπου αρχείου σκελετού βάση του οποίου ο bison δομεί τον παραγόμενο parser
%token-table	Παράγει έναν πίνακα με τα ονόματα των tokens στο αρχείο του parser. Ο πίνακας ονομάζεται yytname όπου τα token αναφέρονται ως yytname[i]
%verbose	παράγει ένα επιπλέον αρχείο που περιέχει αναλυτική περιγραφή των καταστάσεων του parser καθώς και τις ενέργειες για κάθε τύπου lookahead token από τις καταστάσεις αυτές

Γενικά ο Bison διαθέτει ένα σύνολο από οδηγίες με την χρήση των οποίων, ορίζονται από τον χρήστη όλες οι παράμετροι σχετικά με την συμπεριφορά του. Ο Πίνακας 6-4, περιέχει μια σύντομη περιγραφή των βασικότερων οδηγιών του bison για τον ορισμό της συμπεριφοράς του.

6.3.8.1 Header file

Η χρήση της οδηγίας %defines, δημιουργεί ένα αρχείο επικεφαλίδων (*. h) που περιέχει τους ορισμούς των token names της γραμματικής καθώς και ένα σύνολο απαραίτητων δηλώσεων όπως YYSTYPE, YYLTYPE, yylval, yylloc, κ.λ.π. Το αρχείο αυτό είναι απαραίτητο προκειμένου ο parser να συνεργάζεται με τον scanner εφόσον περιέχει τους ορισμούς της κωδικοποίησης των token. Συγκεκριμένα το αρχείο επικεφαλίδων πρέπει να συμπεριλαμβάνεται στο αρχείο (*.l) του scanner. Επίσης χρησιμεύει για την χρησιμοποίηση των αναφορών του parser και από διαφορετικά αρχεία (modules) κώδικα. Τέλος ο κώδικας που δηλώνεται από τον χρήστη με χρήση των οδηγιών %code requires {code} και %code provide {code}, περιέχεται (μεταφέρεται) και στο παραγόμενο αρχείο επικεφαλίδων.

6.3.9 Conflicts

Στην περίπτωση που οι κανόνες της γραμματικής που εισάγουμε δεν ικανοποιούν την περιγραφή μια LR(1) grammar, τότε είναι πιθανό να λάβουμε μηνύματα διενέξεων τύπου shift/reduce conflicts ή/και reduce/reduce conflicts. Στην πρώτη περίπτωση ενδέχεται να μην είναι σοβαρές διενέξεις, ωστόσο στη δεύτερη πρόκειται για σοβαρές διενέξεις που θα πρέπει να επιλυθούν είτε με αναδιατύπωση των κανόνων είτε με τον καθορισμό συσχετισμών (associativity) και προτεραιοτήτων (precedence) των συστατικών των κανόνων της γραμματικής. Για τον εντοπισμό τέτοιων σφαλμάτων – διενέξεων ο bison παρέχει μια σειρά από οδηγίες (πχ %verbose) προκειμένου να εξάγει αναλυτικές πληροφορίες των πεπερασμένων καταστάσεων και μεταβάσεων του parser ανά κανόνα και lookahed token.

6.3.9.1 Shift / Reduce Conflicts

Μια γραμματική παρουσιάζει **shift/reduce conflict** όταν κατά τη διάρκεια της συντακτικής ανάλυσης σε ένα σημείο η τρέχουσα κατάσταση της stack ταιριάζει με έναν κανόνα (προκειμένου αυτός να εφαρμοσθεί, reduction) και ταυτόχρονα ο έλεγχος του lookahead token δημιουργεί προϋποθέσεις να ταιριάξει με άλλον κανόνα σε περίπτωση που το token εισέλθει (shift) στην stack του parser. Μία τέτοια κατάσταση ενδέχεται να οδηγήσει τον parser στην παραγωγή μιας λανθασμένης ή ανεπιθύμητης derivation.

6.3.9.2 Reduce / Reduce Conflicts

Μια γραμματική παρουσιάζει **reduce/reduce conflict** όταν υπάρχουν δύο οι περισσότεροι κανόνες που μπορούν να εφαρμοσθούν στην ίδια ακολουθία εισόδου. Μία τέτοια ασάφεια (ambiguity), συνήθως υποδεικνύει ένα σοβαρό λάθος στη γραμματική, εφόσον υπάρχουν περισσότεροι του ενός τρόποι για την αναγωγή (reduction) μίας λέξης (word, token) σε μία ακολουθία εισόδου. Μία τέτοια κατάσταση ενδέχεται να οδηγήσει τον parser στην παραγωγή μιας λανθασμένης ή ανεπιθύμητης derivation.

6.3.10 C++ Parsers

Με την χρήση της οδηγίας %skeleton “lalr1.cc” εντός του αρχείου *.y, παράγεται ένας αντικειμενοστραφής LALR(1) parser σε γλώσσα C++. Στην περίπτωση αυτή τα δεδομένα και οι λειτουργίες του parser περιγράφονται από μια κλάση αντικειμένων, όπου ένας parser είναι ένα στιγμιότυπο (αντικείμενο) της κλάσης. Συνεπώς κάθε αντικείμενο δρα ανεξάρτητα από το άλλο σε ίδια ή διαφορετική ακολουθία εισόδου και στην ουσία πρόκειται για έναν pure parser (ενότητα 6.3.7).

Παράλληλα με τα προαναφερόμενα αρχεία, δημιουργούνται και κάποια επιπλέον όπως τα postiton.hh και location.hh για τον ορισμό των αντίστοιχων κλάσεων, καθώς και το stack.hh για τον ορισμό μιας κλάσης που χρησιμοποιείται από τον parser. Στα αρχεία εξόδου output.tab.c και output.tab.h, ορίζεται και δηλώνεται ή κλάση του parser με εξ ορισμού όνομα “parser”, το οποίο και μπορεί να αλλάξει με την χρήση της οδηγίας %fefine parser_class_name “name”. Το interface της κλάσης παρουσιάζεται συνοπτικά (Πίνακας 6-5) και μπορεί να επεκταθεί με την χρήση της οδηγίας %parse-param.

Πίνακας 6-5 : Interface of C++ Parser class

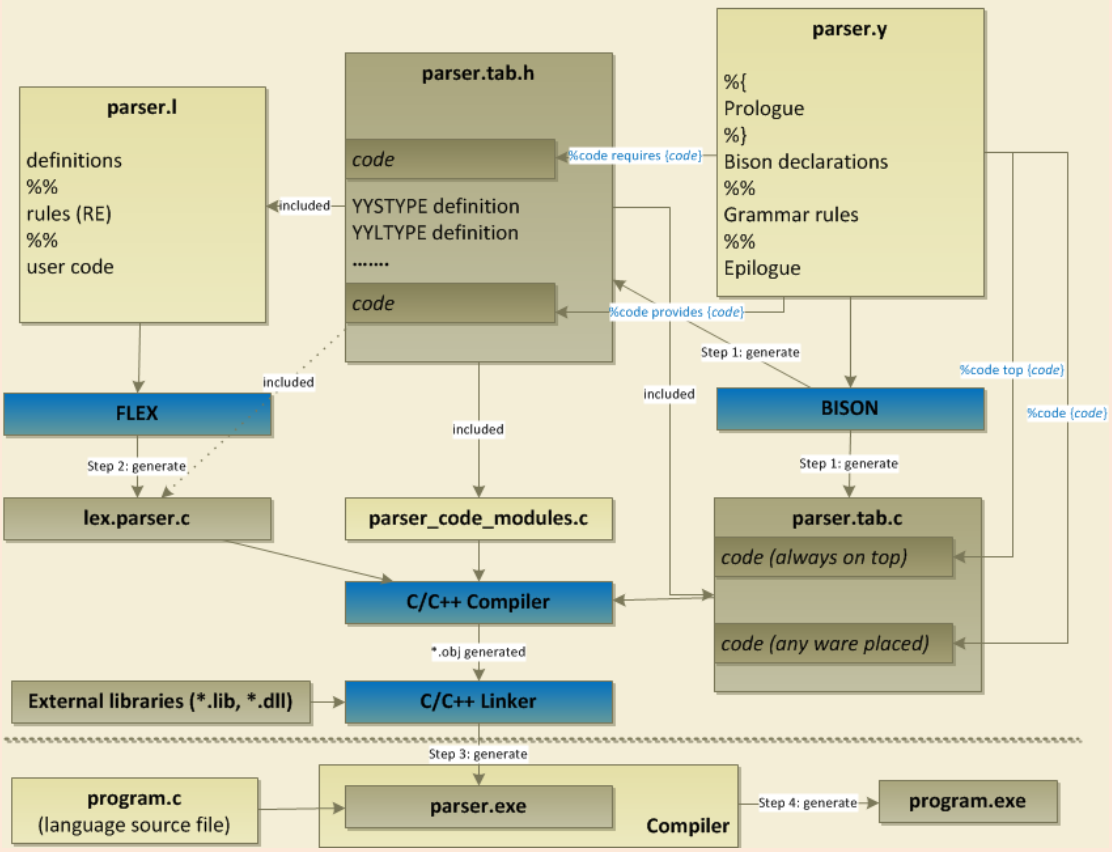
Όνομα	Περιγραφή
semantic_value_type	Type of parser
location_value_type	Type of parser

Όνομα	Περιγραφή
parser (type1 arg1, ...)	Method on parser : Constructor για την δημιουργία νέου (αντικειμένου) parser
int parse ()	Method on parser : Εκκίνηση συντακτικής ανάλυσης (semantic analysis) ή έναρξη του parser
std::ostream& debug_stream () void set_debug_stream (std::ostream& o)	Method on parser : διαχειρίζεται το stream για την λειτουργία tracing parsing
debug_level_type debug_level () void set_debug_level (debug level l)	Method on parser : διαχειρίζεται το επίπεδο (no, full) της λειτουργίας tracing parsing
void error (const location type& l, const std::string& m)	Method on parser : ο ορισμός της μεθόδου διαχείρισης σφάλματος παρέχεται από τον χρήστη και αναφέρει ένα σφάλμα του parser στην τοποθεσία l και περιγράφεται από το m
int yylex (semantic value type& yylval, location type& yylloc, type1 arg1, ...)	Method on parser : επιστρέφει το επόμενο token (ακέραια κωδικοποίηση) καθώς και την τοποθεσία και τη semantic value, μέσω των (by reference) παραμέτρων

Στο εγχειρίδιο χρήσης του Bison (Donnelly & Stallman, 2008, pp. 111-117), παρουσιάζεται ένα πλήρες πρότυπο δημιουργίας ενός C++ parser με χρήση του λογισμικού flex και bison.

6.4 Σύνοψη

Στο παρακάτω σχήμα (Εικόνα 6-3) αποτυπώνονται συνοπτικά οι σχέσεις και η αλληλεπίδραση μεταξύ των συστατικών στοιχείων (αρχείων κώδικα, αρχείων επικεφαλίδων, εργαλείων, μεταγλωττιστών, κ.λ.π.) που εμπλέκονται στην υλοποίηση του λεκτικού και συντακτικού αναλυτή ενός μεταγλωττιστή με τη χρήση του έτοιμου λογισμού Flex & Bison.



Εικόνα 6-3 : Σχήμα συνεργασίας - αλληλεπίδρασης στοιχείων Flex & Bison

Αναφερόμενοι στην Εικόνα 6-3, παρατηρούμε ότι η δημιουργία του τελικού εκτελέσιμου αρχείου (parser.exe) απαιτεί την εκτέλεση των διαδοχικών βημάτων α) εκτέλεσης του Bison, β) εκτέλεσης του Flex και γ) μεταγλώττιση (compiling) και διασύνδεση (linking) των εμπλεκόμενων αρχείων. Κατά την ανάπτυξη ενός parser η εκτέλεση των ανωτέρω βημάτων από το περιβάλλον εκτέλεσης εντολών (command prompt) αποτελεί μια χρονοβόρα και μη παραγωγική διαδικασίας. Για το λόγο αυτό το λογισμικό (Flex, Bison) παρέχει έτοιμα αρχεία παραμετροποίησης (custom build rules) με τα οποία διαδεδομένα περιβάλλοντα ανάπτυξης λογισμικού (π.χ. MS Visual Studio) αυτοματοποιούν την εκτέλεση των ανωτέρω βημάτων.

7 Υλοποίηση Μεταγλωττιστή με χρήση Σχεδιαστικών Προτύπων

7.1 Εισαγωγή

Προκειμένου να διερευνηθούν στην πράξη τα αποτελέσματα της εφαρμογής συγκριμένων σχεδιαστικών προτύπων (design patterns) αντικειμενοστραφούς προγραμματισμού, η παρούσα εργασία επικεντρώνεται στην ανάλυση, σχεδίαση και ανάπτυξη μέρους ενός μεταγλωττιστή (compiler). Συγκεκριμένα αναλύεται, σχεδιάζεται και αναπτύσσεται το front-end τμήμα ενός μεταγλωττιστή της γνωστής γλώσσας προγραμματισμού ANSI C. Επίσης παρουσιάζεται η σχεδίαση του πίνακα συμβόλων (symbol table), του parse / abstract tree και της εσωτερικής αναπαράστασής του, της three-address code αναπαράστασης, καθώς και η σχεδίαση και εφαρμογή των ενδεικτικών λειτουργιών scope checking, type inference & checking και graph generation.

Η υλοποίηση ενός πλήρους μεταγλωττιστή αφενός είναι ένα είναι πολύ μεγάλο project για να αναπτυχθεί και ενταχθεί σε μία εργασία, και αφετέρου δεν εξυπηρετεί τον κύριο στόχο της εργασίας. Επιπλέων οι τεχνικές και τρόποι υλοποίησης ενός μεταγλωττιστή για την ίδια γλώσσα εισόδου μπορούν να διαφέρουν σημαντικά από υλοποίηση σε υλοποίηση, υιοθετώντας κατά περίπτωση διαφορετικές προσεγγίσεις σχετικά με τις ενδιάμεσες εσωτερικές αναπαραστάσεις, τα στάδια επεξεργασίας, κ.λ.π. Εντούτοις μελετώντας τη σχετική βιβλιογραφία, είναι δυνατό να εντοπιστούν κάποιες βασικές αρχές με κοινότυπες ή παρεμφερείς δομές αναπαράστασης και στάδια επεξεργασίας που χρησιμοποιούνται αρκετά συχνά (όπως οι Graphical IRs, ενότητα 5.6.1). Η σχεδίαση και υλοποίηση του front-end του μεταγλωττιστή μαζί με ένα ενδεικτικό υποσύνολο των κυριότερων δομών και λειτουργιών του επαρκεί, όπως θα διαπιστώσουμε στη συνέχεια, προκειμένου να εξάγουμε τα συμπεράσματά μας σχετικά με την αποτελεσματικότητα και τα οφέλη της εφαρμογής συγκριμένων αντικειμενοστραφών σχεδιαστικών προτύπων (design patterns) σε ένα δύσκολο πρόβλημα όπως είναι ένας μεταγλωττιστής.

Για την υλοποίηση του (μέρους) μεταγλωττιστή χρησιμοποιήθηκαν κατά περίπτωση:

- Για την ίδια την υλοποίηση του μεταγλωττιστή χρησιμοποιήθηκε η ευρέως διαδεδομένη αντικειμενοστραφής γλώσσα προγραμματισμού C++ και το περιβάλλον ανάπτυξης MS Visual Studio, έκδοση Professional 2008.
- Για την υλοποίηση του λεκτικού αναλυτή χρησιμοποιήθηκε το ελεύθερο λογισμικό Flex, έκδοση 2.5.4.a (ενότητα 6.2).
- Για την υλοποίηση του συντακτικού αναλυτή (parser) χρησιμοποιήθηκε ελεύθερο το λογισμικό Bison, έκδοση 2.4.1 (ενότητα 6.3) και συγκεκριμένα ένας pure OO parser σε C++, με βάση το σχετικό πρότυπο της τεκμηρίωσης (ενότητα, 6.3.10).
- Για την γραφική αναπαράσταση και εμφάνιση των Graphical IRs, χρησιμοποιήθηκε το ελεύθερο λογισμικό Graphviz[16], έκδοση 2.30.1, όπου η αναπαράσταση ενός γράφου πραγματοποιείται μέσω της DOT Language[17].

[16] Graph Visualization Software (σύνδεσμος εφαρμογής: http://www.graphviz.org/)

[17] Γλώσσα αναπαράστασης γράφων (abstract grammar: http://www.graphviz.org/content/dot-language)

- Για την δημιουργία των UML διαγραμμάτων χρησιμοποιήθηκε το λογισμικό MS Visio, έκδοση 2010.
- Για την εξαγωγή αναφορών των μετρικών ποιότητας λογισμικού χρησιμοποιήθηκε το ελεύθερο λογισμικό SourceMonitor[18], έκδοση 3.4.6.297.

Τα σχήματα, τα UML διαγράμματα και ο κώδικας που παρουσιάζονται αποτελούν μέρος μια πλήρους ανάλυσης – σχεδιασμού και υλοποίησης, ωστόσο για πρακτικούς λόγους επιλέγονται τα σημαντικότερα και αντιπροσωπευτικότερα, προκειμένου να δοθεί μια πλήρη και κατανοητή εικόνα του υπό ανάπτυξη λογισμικού και του τρόπου λειτουργίας του.

7.2 Γλώσσα προγραμματισμού ANSI C – C89

7.2.1 Πρότυπο

Κάθε μεταγλωττιστής υλοποιεί κατά κανόνα την μεταγλώττιση (μετατροπή) μίας γλώσσας προγραμματισμού υψηλού επιπέδου (είσοδος) σε μία γλώσσα χαμηλού (έξοδος). Ως γλώσσα εισόδου για την υλοποίηση του μεταγλωττιστή της εργασίας, επιλέχθηκε μία από τις γνωστότερες και διαχρονικότερες γλώσσες προγραμματισμού, η C.

Προκειμένου η υλοποίηση του μεταγλωττιστή να καταστεί όσο το δυνατό πιο πλήρης, απλή, κατανοητή και κυρίως προσηλωμένη στο στόχο της, που είναι η διερεύνηση της χρήσης σχεδιαστικών προτύπων αντικειμενοστραφούς προγραμματισμού, επιλέχθηκε το πρότυπο **ANSI C Standard ANSI X3.159-1989 "Programming Language C"**. Το πρότυπο αυτό, ονομαζόμενο και ως "**C89**", εκδόθηκε το έτος 1990 ως έργο της επιτροπής X3J11 του American National Standards Institute και αποτελεί ένα σύνολο τυποποιημένων προδιαγραφών της γλώσσας καθώς και των βασικών βιβλιοθηκών της. Λίγο αργότερα το ίδιο πρότυπο επικυρώθηκε, με μικρές τροποποιήσεις, από τον **International Organization for Standardization** ως **ISO/IEC 9899:1990**, αναφερόμενο και ως "**C90**". Αν και τα το πρότυπο έχει αποσυρθεί[19], αποτελεί την βάση των σύγχρονων εκδόσεων της γλώσσας (C, C++), είναι μια πλήρης και ισχυρή γλώσσα προγραμματισμού και προσφέρεται για την υλοποίηση του μεταγλωττιστή της εργασίας.

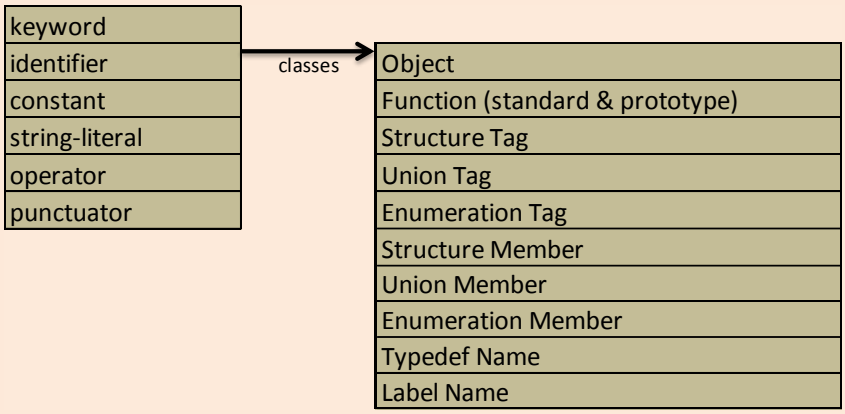
Στις επόμενες ενότητες παρουσιάζονται συγκεντρωτικά τα βασικά σημεία του προτύπου C89 που αφορούν την υλοποίηση της εργασίας. Σημειώνεται ότι η υλοποίηση επικεντρώνεται στον κυρίως μεταγλωττιστή και όχι στον προεπεξεργαστή του και τα preprocessing-tokens.

7.2.2 Tokens

Το C89 ορίζει τους παρακάτω (Εικόνα 7-1) τύπους λεκτικών στοιχείων (tokens) καθώς και τις αντίστοιχες κλάσεις αναγνωριστικών (identifiers).

[18] Λογισμικό μέτρησης ποιότητας κώδικα SourceMonitor (σύνδεσμος http://www.campwoodsw.com/)

[19] Έχει αποσυρθεί και από τους δύο οργανισμού INCITS (TechstreetStore, 2014) and ISO/IEC (ISO Organization, 2014), το ισχύον πρότυπο είναι το ISO/IEC 9899:2011 (ISO Organization, 2014), αναφερόμενο και ως "C11"



Εικόνα 7-1 : Tokens & Identifier classes

7.2.3 Name spaces of identifiers

Στο πρότυπο C89 ορίζονται τέσσερεις διαφορετικοί χώροι ονομάτων (Labels name space, Tags name space, Members name space, Ordinary name space) όπου και κατατάσσονται τα αναγνωριστικά με βάση την κλάση τους και παρουσιάζεται σχηματικά στο Παράρτημα II : C89 Identifiers (Εικόνα 11-1, σελ. 169).

7.2.4 Scopes of identifiers

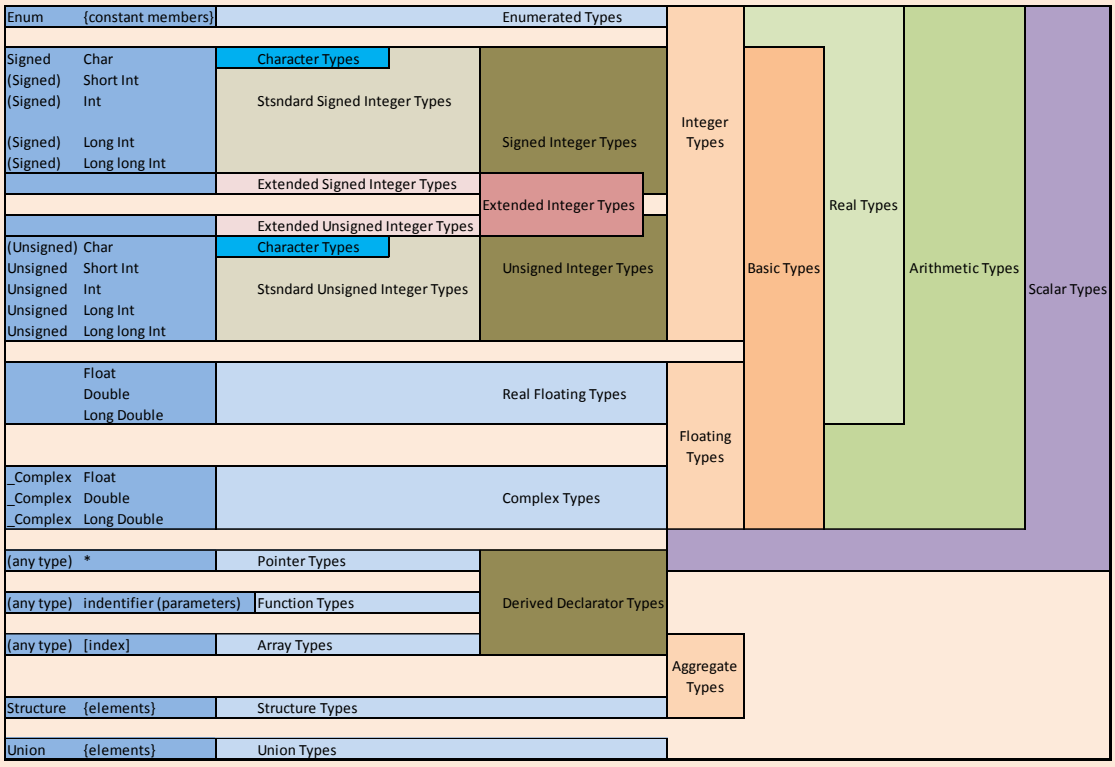
Η έκταση (scope) της δήλωσης κάθε αναγνωριστικού (identifier), διαχωρίζεται με βάση την κλάση του σε τέσσερεις τύπους (Function scope, File scope, Block, scope, Function prototype scope) και παρουσιάζεται σχηματικά στο Παράρτημα II : C89 Identifiers (Εικόνα 11-2, σελ. 170). Τα scopes μπορεί να επικαλύπτει το ένα το άλλο (ενότητα 5.5.1) και σε κάθε scope μπορεί να δηλωθεί μόνο ένα αναγνωριστικό με το ίδιο όνομα. Ωστόσο αν δύο αναγνωριστικά ανήκουν σε διαφορετικά Name Spaces τότε επιτρέπεται να έχουν το ίδιο όνομα εντός του ιδίου scope.

7.2.5 Linkage of Identifiers

Ο Τύπος διασύνδεσης (internal, external) του κάθε αναγνωριστικού ορίζεται με βάση την κλάση του και παρουσιάζεται σχηματικά στο Παράρτημα II : C89 Identifiers (Εικόνα 11-3, σελ. 171).

7.2.6 C Types

Οι βασικοί τύποι των αναγνωριστικών που ορίζονται στο C89/90 παρουσιάζονται στο παρακάτω σχήμα (Εικόνα 7-2) όπου και διαφαίνονται οι επιμέρους ομαδοποιήσεις τους.



Εικόνα 7-2 : C Types classification

7.3 Υλοποίηση λεκτικής ανάλυσης

Τα λεκτικά στοιχεία (tokens) της γλώσσας εισόδου κωδικοποιούνται δια μέσου του bison (Παράρτημα I : C89 – Flex & Bison Grammar, Εικόνα 10-1, σελ. 164) και αναγνωρίζονται με τις κανονικές εκφράσεις του flex (Flex Tokens Regular Expressions (REs), σελ. 163) στο σχετικό αρχείο (*.l). Οι βασικές κατηγορίες των λεκτικών στοιχείων εκτός από τις λέξεις κλειδιά και τους τελεστές της γλώσσας είναι :

- Identifier (ή Type name)
- Constant (hexadecimal integer)
- Constant (octal integer)
- Constant (decimal integer)
- Floating point (integral floating point με exponent)
- Floating (προεραιτικό integral part, mandatory decimal part, optional exponent)
- Floating (mandatory integral part, προεραιτικό decimal part, optional exponent)
- Floating (hexadecimal)
- Constant character
- String literal

7.4 Υλοποίηση συντακτικής ανάλυσης

Οι γραμματική της γλώσσας εισόδου δίνεται σε μορφή LR(1) Grammar στον bison (Παράρτημα I : C89 – Flex & Bison Grammar, σελ. 164), στο αρχείο (*.y). Η παραμετροποίηση του λογισμικού (flex, bison) βασίζεται στο σχετικό πρότυπο του εγχειριδίου του, ως ένας pure C++ OO parser (ενότητα, 6.3.10).

7.4.1 Έλεγχος Type name

Σχεδόν όλα τα λεκτικά στοιχεία της C89 είναι μονοσήμαντα, εκτός του Identifier, ο οποίος προηγουμένως έχει δηλωθεί ως τύπος με την εντολή typedef. Αν για παράδειγμα έχουμε τη δήλωση | typedef int id_01; |, όπου id_01 είναι Identifier token, τότε η ακόλουθη

δήλωση | id_01 var1; | θεωρεί το id_01 ως TYPE_NAME token του κανόνα type_specifier της γραμματικής. Το γεγονός ότι ο ίδιος identifier μπορεί να αναγνωρίζεται ως διαφορετικό token κατά περίπτωση, καθιστά αναγκαία την πραγματοποίηση μίας μορφής context sensitive analysis στο αρχικό στάδιο της λεκτικής και συντακτικής αναγνώρισης της γλώσσας. Προκειμένου ο scanner να είναι σε θέση να αποφανθεί πότε ένας identifier είναι τύπου IDENTIFIER ή TYPE_NAME, θα πρέπει να γίνεται μια καταγραφή όλων των identifier που παρουσιάζονται εντός των δηλώσεων (declarator) με τον TYPEDEF προσδιοριστή. Επίσης ο έλεγχος αυτός θα πρέπει να γίνεται με βάση την περιοχή (scope) ισχύος του κάθε identifier.

7.4.2 Έλεγχος Scope

Μια λύση καταγραφής των typedef identifier ανά scope, μπορεί να συνδυαστεί με την context sensitive analysis και τη καταγραφή όλων των identifier του προγράμματος παράλληλα με το scope checking (Εικόνα 11-2) με βάση τα name spaces της γλώσσας (Εικόνα 11-1). Γενικά στους μεταγλωττιστές συνηθίζεται η δημιουργία ενός symbol table (ενότητα 5.7) για την καταγραφή όλων των συμβόλων του προγράμματος, τον έλεγχο των scopes – name spaces καθώς και τη συλλογή οποιασδήποτε πληροφορίας (attributes), σχετικά με αυτά τα symbols, που μπορεί να χρησιμοποιηθεί σε επόμενο στάδιο της μεταγλώττισης.

7.5 Symbol Table

Λαμβάνοντας υπόψη τις κλάσεις των identifier (Εικόνα 7-1), τα scopes (Εικόνα 11-2), τα name spaces (Εικόνα 11-1), το πρότυπο της C89 που αποτελούν μέρος του μοντέλου ανάλυσης, καθώς και τα σχεδιαστικά πρότυπα (design patterns) Façade (ενότητα 4.3.4) και Composite (ενότητα 4.3.1), μεταβαίνουμε στο μοντέλο σχεδίασης, σχεδιάζοντας τον symbol table του μεταγλωττιστή με χρήση UML Class Diagrams (ενότητα 2.3.2).

7.5.1 Σχεδίαση symbol table με Composite και Facade

Προκειμένου το interface του symbol table να είναι ενιαίο, σχεδιάσθηκε μια κλάση (CSymbolTable_Facade) η ο ποία και συγκεντρώνει όλε τις λειτουργίες του. Ο symbol table αποτελείται (περιέχει) τα τέσσερα name space, όπου κάθε name space περιέχει ένα (file) scope. Κάθε αντικείμενο scope, της κλάσης Scope, μπορεί να περιέχει άλλα scope διαχωριζόμενα σε τέσσερεις επιμέρους τύπους (subclasses File_Scope, Block_Scope, Function_Scope, Proto_Function_Scope). Κάθε scope περιέχει έναν hash symbol table (κλάση Symbol_Table) ο οποίος μπορεί να περιέχει πολλά σύμβολα (κλάση Symbol) του scope στον οποίο και ανήκει. Το διάγραμμα κλάσεων του symbol table της υλοποίησης παρουσιάζεται στην Εικόνα 7-3. Η σχεδίαση περιέχει όλες τις απαραίτητες λειτουργίες (methods) για τους ελέγχους που απαιτούνται κατά την λεκτική και συντακτική ανάλυση. Περιέχει μεθόδους για τη δημιουργία (εντός του τρέχοντος) νέου scope σε συγκεκριμένο name space, για το κλείσιμο (του τρέχοντος) scope σε συγκεκριμένο name space, για την επίλυση των ambiguous περιοχών proto function scopes και block function scopes, για την επίλυση των φωλιασμένων block των δηλώσεων structure και union, για την προσθήκη (στο τρέχον scope) ενός συμβόλου, για την ανάκτηση (από το τρέχον scope) ενός συμβόλου, για την ανάκτηση (σε όλα τα γονικά scope) ενός συμβόλου, κ.λ.π.

Ο συγκεκριμένος symbol table εισάγει τα σύμβολα σε έναν πίνακα (για κάθε scope) σταθερού μεγέθους μέσω μιας hash function (Get_Hash_Key) κατά τα πρότυπα της τεχνικής

hashing with linear probing (ενότητα, 5.7.1), η οποία και εξασφαλίζει (στη γενική περίπτωση) χρόνο εισαγωγής και αναζήτησης συμβόλου ίσο με O(1). Ακόμη και στην περίπτωση που η αναζήτηση ενός συμβόλου πραγματοποιείται ιεραρχικά προς τα γονικά scopes (το μέγιστο επίπεδο επικάλυψης των scopes είναι ένας σταθερός αριθμός K που δίνεται από την προδιαγραφή της γλώσσας), ο χρόνος αναζήτησης συμβόλου είναι ίσο με K×O(1) = O(1), δηλαδή παραμένει σταθερός.

Εικόνα 7-3 : Διάγραμμα κλάσεων Symbol Table

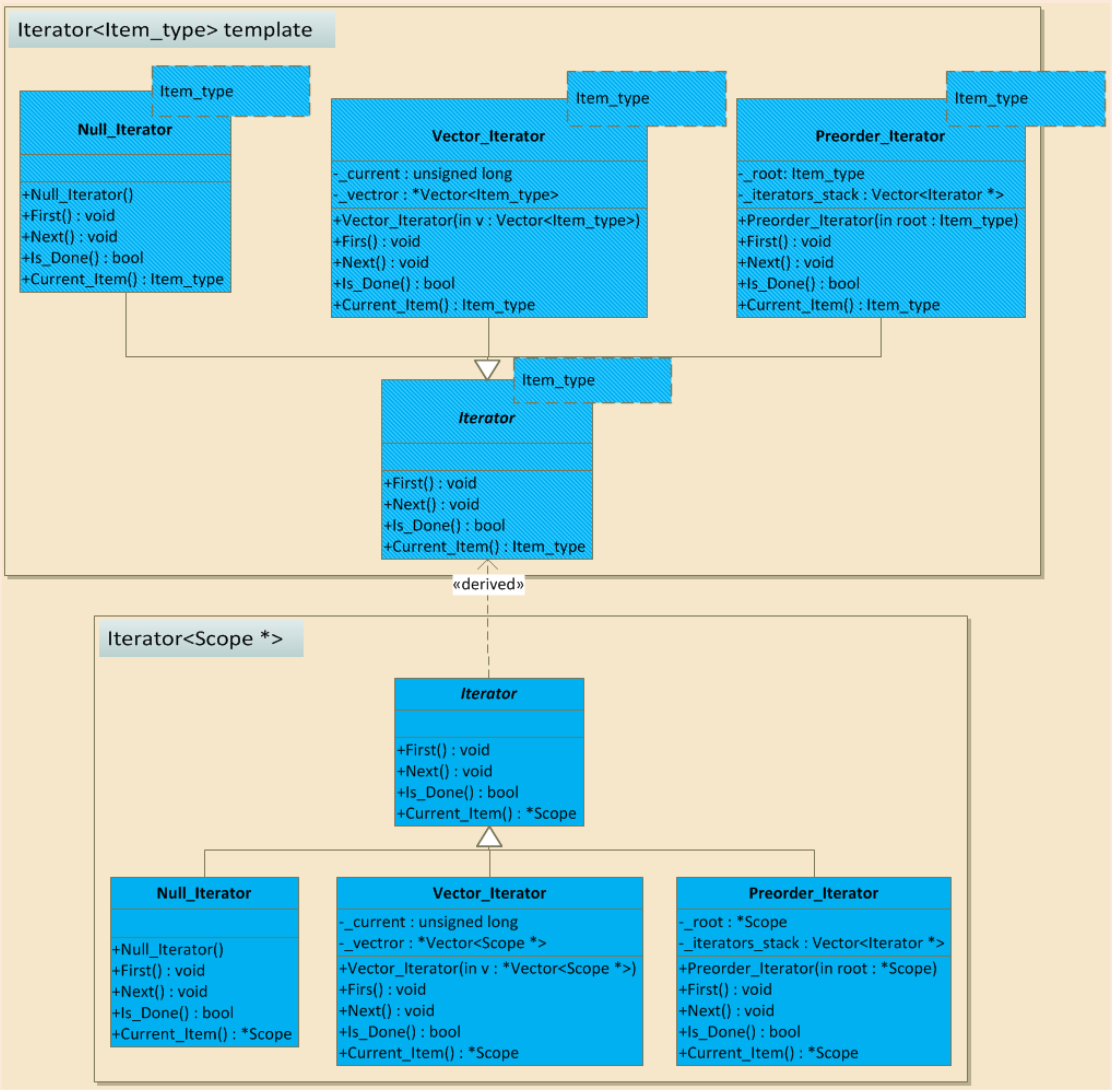
Είναι εμφανές ότι η κλάση CSymbolTable_Facade είναι η εφαρμογή του Façade σχεδιαστικού προτύπου. Επίσης η κλάση Scope (συμπεριλαμβανομένων των subclasses) είναι μια δομή του σχεδιαστικού προτύπου Composite, δεδομένου ότι ένα scope μπορεί να είναι υποχρεωτικά ενός συγκεκριμένου τύπου (subclass) και μπορεί να περιέχει κανένα ή περισσότερα sub scopes. Δηλαδή τα scopes οργανώνονται ως μια δενδροειδή μορφή όπου

ένα scope μπορεί να είναι node ή/και leaf. Κάθε scope περιέχει ένα πίνακα Vector<Scope *>, με δείκτες προς τα scopes παιδιά του. Όπως έχει αναφερθεί, σε τέτοιου είδους δενδροειδές δομές, ο σχεδιασμό ενός μηχανισμού διαπέρασης των στοιχείων τους χωρίς ο χρήστης να γνωρίζει την εσωτερική δομή αναπαράστασης τους, αποτελεί ένα καλό πεδίο εφαρμογής του σχεδιαστικού προτύπου Iterator που παρουσιάζεται παρακάτω.

7.5.2 Διαπέραση δομής Scopes με Iterators

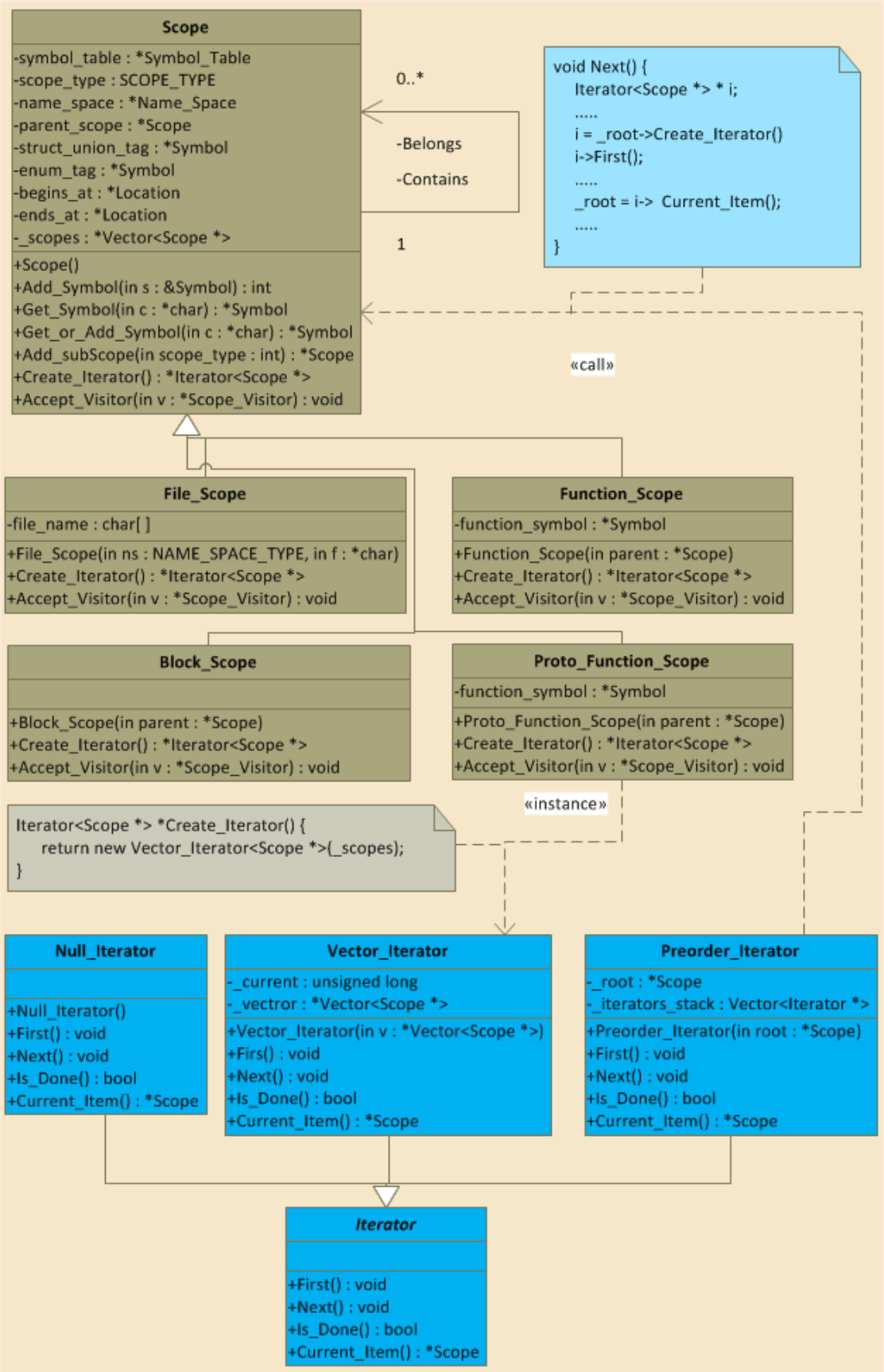
7.5.2.1 Σχεδίαση Iterator template

Στην composite δομή των scopes η βασική δομή διαπέρασης είναι αυτή του πίνακα τύπου Vector<Scope *>, με δείκτες προς τα scopes παιδιά του. Η κλάση vector ορίζεται στην βοηθητική βιβλιοθήκη της C++ και αποτελεί μια δομή (δυναμικού μεγέθους) πίνακα στοιχείων με λειτουργίες (methods) για την εισαγωγή, διαγραφή, τροποποίηση, αναζήτηση, των στοιχείων με ποικίλους τρόπους. Συνεπώς αντί να σχεδιαστεί μια κλάση Scope_Iterator μόνο για scope αντικείμενα, επιλέγεται η σχεδίαση μιας πιο γενικής κλάσης Vector_Iterator, η ευθύνη της οποίας είναι η διαπέραση των στοιχείων ενός vector που δέχεται ως παράμετρο (μέσω δείκτη) κατά την δημιουργία των αντικειμένων της. Επιπλέον προκειμένου η κλάση Vector_Iterator να είναι γενική και να μπορεί να εφαρμοσθεί (ως ανοικτό πρότυπο) και για διαφορετικούς τύπους στοιχείων (εκτός των Scopes*), επιλέγεται η σχεδίαση της κλάσης μέσω ενός template με παράμετρο Item_type. Η σχεδίαση των Iterator δομών τόσο του template όσο και του στιγμιότυπου του για Item_type = Scope* παρουσιάζεται στην Εικόνα 7-4 και βασίζεται στα αναφερόμενα στην ενότητα 4.3.2.



Εικόνα 7-4 : Διάγραμμα κλάσεων Iterator templates - Iterator για Scopes

Όταν ένα scope δεν έχει κανένα sub scope τότε μπορεί να επιστρέφει ένα Null_Iterator αν και ο Vector_Iterator θα έχει την ίδια συμπεριφορά ως κενός. Η κλάση Preorder_Iterator (ενότητα 4.3.2.2) είναι ένας οδηγός μιας preorder διαπέρασης της δομής, χρησιμοποιώντας τους Iterators (π.χ. Vector_Iterator) των στοιχείων που διαπερνά. Όλες οι υπό κλάσεις έχουν κοινή διεπαφή (Firs, Next, IS_Done, Current_Item) και ιεραρχούνται κάτω από την abstract κλάση Iterator. Η αλληλεπίδραση των κλάσεων Scope και Iterator παρουσιάζεται στο διάγραμμα κλάσεων στην Εικόνα 7-5.



Εικόνα 7-5 : Διάγραμμα κλάσεων Scope - Iterator

Στην κλάση Scope έχει δηλωθεί η μέθοδος Create_Iterator(), και υλοποιείται από κάθε υπό κλάση της Scope (πχ Prot_Function_Scope), έτσι ώστε κάθε επιμέρους υλοποίηση να

επιστρέφει τον κατάλληλο τύπο Iterator (π.χ. Vector_Iterator) για τη συγκεκριμένη υποκλάση της Scope. Με αυτόν τον τρόπο κάθε τύπος της Scope επιστρέφει τον κατάλληλο Iterator, με κλήση της ίδιας μεθόδου (Create_Iterator), εκμεταλλευόμενος την ιδιότητας του πολυμορφισμού της κληρονομικότητας των κλάσεων. Ειδικά η λειτουργία Next της κλάσης Preorder_Iterator είναι αυτή που περιέχει τον αλγόριθμο που υλοποιεί την preorder διαπέραση, χρησιμοποιώντας μια stack όπου και αποθηκεύει τα ενδιάμεσα αντικείμενα (Scopes), προκειμένου να είναι σε θέση να επανέλθει σε ένα αντικείμενο μετά την διαπέραση όλων των παιδιών του.

7.5.2.2 Υλοποίηση null, vector και preorder Iterators

Επειδή η υλοποίηση του Iterator που παρουσιάσθηκε στην σχεδίαση έχει γενικά ενιαία δομή και ευρεία χρήση, παρακάτω δίνεται ο ενδεικτικός κώδικας της υλοποίησης όλων των κλάσεων και υποκλάσεων Iterator, Null_Iterator, Vector_Iterator, Preorder_Iterator, καθώς και της μεθόδου Create_Iterator() της κλάσης Proto_Function_Scope.

```
Iterator<Scope*> *Proto_Function_Scope::Create_Iterator() {
	return new Vector_Iterator<Scope*> (_scopes);
};
```

```
template <class ItemType>
class Iterator {
public:
	Iterator(){}
	virtual void First() = 0;
	virtual void Next() = 0;
	virtual bool Is_Done() = 0;
	virtual ItemType Current_Item() = 0;
};
```

```
template <class ItemType>
class Null_Iterator : public Iterator<ItemType> {
public:
	Null_Iterator(){}
	void First(){}
	void Next(){}
	bool Is_Done() {return true;}
	ItemType Current_Item() {return 0;}
};
```

```
template <class ItemType>
class Vector_Iterator  : public Iterator<ItemType> {
private:
	unsigned long _current;
	vector<ItemType> *_vector;
public:
	Vector_Iterator(vector<ItemType> *aVector) {
		_vector=aVector;
		_current=0;
	};
	void First() {_current=0;} ;
	void Next() {_current++;};
	bool Is_Done() {
		if (   (_current>=_vector->size() ) ||
			(_vector->at(_current)==0) )
			return true;
		else return false;
	};
	ItemType Current_Item(){
		if (!Is_Done()) return _vector->at(_current);
		else return 0;
	};
```

```
};
```

```
template <class ItemType>
class Preorder_Iterator: public Iterator<ItemType> {
private:
	ItemType _root;
	vector<Iterator*> *_iterators_stack;
	Iterator* _i;
public:
	Preorder_Iterator(ItemType root) {
		_root=root;
		_iterators_stack =
			new vector<Iterator*> (PREORDER_ITERATOR_STACK_SIZE);
	};
	void First() {
		_i = _root->Create_Iterator();
		if (_i) {
			_i->First();
			_iterators_stack->clear();
			_iterators_stack->push_back(_i);
		}
	};
	void Next(){
		_i =
		_iterators_stack->back()->Current_Item()->Create_Iterator();
		_i->First();
		_iterators_stack->push_back(_i);
		while ((_iterators_stack->size()>0) &&
				(_iterators_stack->back()->Is_Done())) {
			delete _iterators_stack->back();
			_iterators_stack->pop_back();
			if (!_iterators_stack->empty())
				_iterators_stack->back()->Next();
		}
	};
	bool Is_Done() {
		return (_iterators_stack->empty()) ||
			(_iterators_stack->back()->Is_Done()) ;
	};
	ItemType Current_Item(){
		if (_iterators_stack->size()>0)
			return _iterators_stack->back()->Current_Item();
		else return 0;
	};
};
```

7.5.2.3 *Εφαρμογή vector Iterator*

Με βάση την παραπάνω ενδεικτική υλοποίηση, για την απλή διαπέραση των sub scopes αντικειμένων ενός Scope αντικειμένου, έστω _root, αρκεί μια ενιαία δομή επανάληψης :

```
void Simple_Scope_Iteration_Application (Scope *_root) {
	Iterator<Scope *> *vi = _root->Create_Iterator();
	for (vi->First();!vi->Is_Done();vi->Next()) {
		Scope *s = vi->Current_Item();
		. . . . .
		s->Get_Symbol("symbol_01"); // παράδειγμα πρόσβασης
		. . . . .
	}
}
```

Η πρόσβαση στα επιμέρους αντικείμενα (s) του _root πραγματοποιείται εύκολα και χωρίς ο χρήσης να γνωρίζει την εξειδίκευση του sub Scope (πχ Proto_Function_Scope). Η κλήση οποιασδήποτε πολυμορφικής μεθόδου του αντικειμένου s (Scope), οδηγείται μέσω

της κληρονομικότητας των κλάσεων στην αντίστοιχη υλοποίηση μεθόδου της εξειδικευμένης Scope.

7.5.2.4 Εφαρμογή Preorder Iterator

Αντίστοιχα, για την preorder διαπέραση μίας δενδροειδής δομής από Scope αντικείμενα με ρίζα το αντικείμενο _root, αρκεί επίσης μια ενιαία δομή επανάληψης:

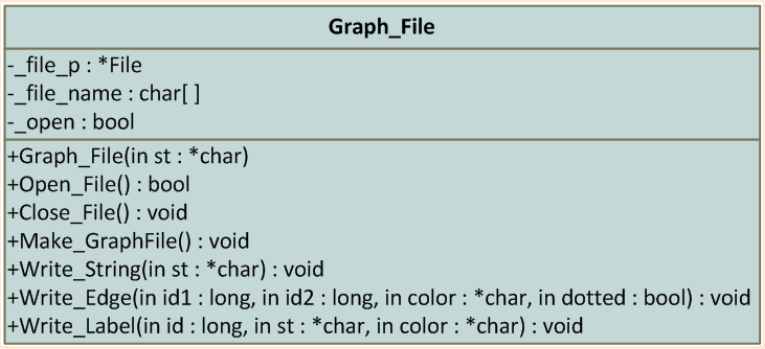
```
void Preorder_Scope_Iteration_Application (Scope *_root) {
	Iterator<Scope *> *poi = new Preorder_Iterator<Scope *>(_root);

	_root->Get_Symbol("symbol_01"); // pre πρόσβαση στη ρίζα
	for (poi->First();!poi->Is_Done();poi->Next()) {
		Scope *s = poi->Current_Item();
		. . . . .
		s->Get_Symbol("symbol_01"); // παράδειγμα πρόσβασης
		. . . . .
	}
}
```

Για την εκκίνηση της preorder διαπέρασης αρκεί η απευθείας δημιουργία ενός αντικειμένου Preorder_Iterator με παράμετρο τη ρίζα (_root) της δενδροειδής δομής. Η πρόσβαση στα επιμέρους αντικείμενα (s, Scope) του δένδρου πραγματοποιείται εύκολα και χωρίς ο χρήσης να γνωρίζει την εξειδίκευση του sub Scope (πχ Proto_Function_Scope) του κάθε κόμβου. Η κλήση οποιασδήποτε πολυμορφικής μεθόδου του αντικειμένου s (Scope), οδηγείται μέσω της κληρονομικότητας των κλάσεων στην αντίστοιχη υλοποίηση μεθόδου της εξειδικευμένης Scope. Από το παράδειγμα διαφαίνεται η εντυπωσιακή απλότητα του παραγόμενου κώδικα, για μία κατά τα άλλα περίπλοκη preorder διαπέραση μιας Composite δενδροειδούς δομής πολυμορφικών αντικειμένων.

7.5.3 Κατασκευή Symbol Table Graph με Visitors

Στην δενδροειδή μορφή των Scopes του symbol table, ενδέχεται για κάθε scope και ανάλογα με τον τύπο του, να χιαστεί η εκτέλεση μίας σειράς λειτουργιών. Για παράδειγμα στην παρούσα φάση μια βοηθητική λειτουργία είναι η δημιουργία ενός γράφου για την εξαγωγή της δομής των scopes προς τον χρήστη. Για τον σχεδιασμό των γράφων εξάγεται (σε αρχείο *.dot) μια σειρά εντολών σε Dot Language προκειμένου στη συνέχεια να σχεδιαστεί ο γράφος (αρχείο εξόδου *.jpg) από το λογισμικό GraphViz. Προκειμένου να τυποποιηθεί η δημιουργία σχετικού αρχείου εξόδου καθώς και η σύνταξη των βασικών εντολών της dot language για κόμβους και ακμές του γράφου, σχεδιάζεται η κλάση διεπαφής Graph_File (Εικόνα 7-6).



Εικόνα 7-6 : Κλάση διεπαφής GraphViz

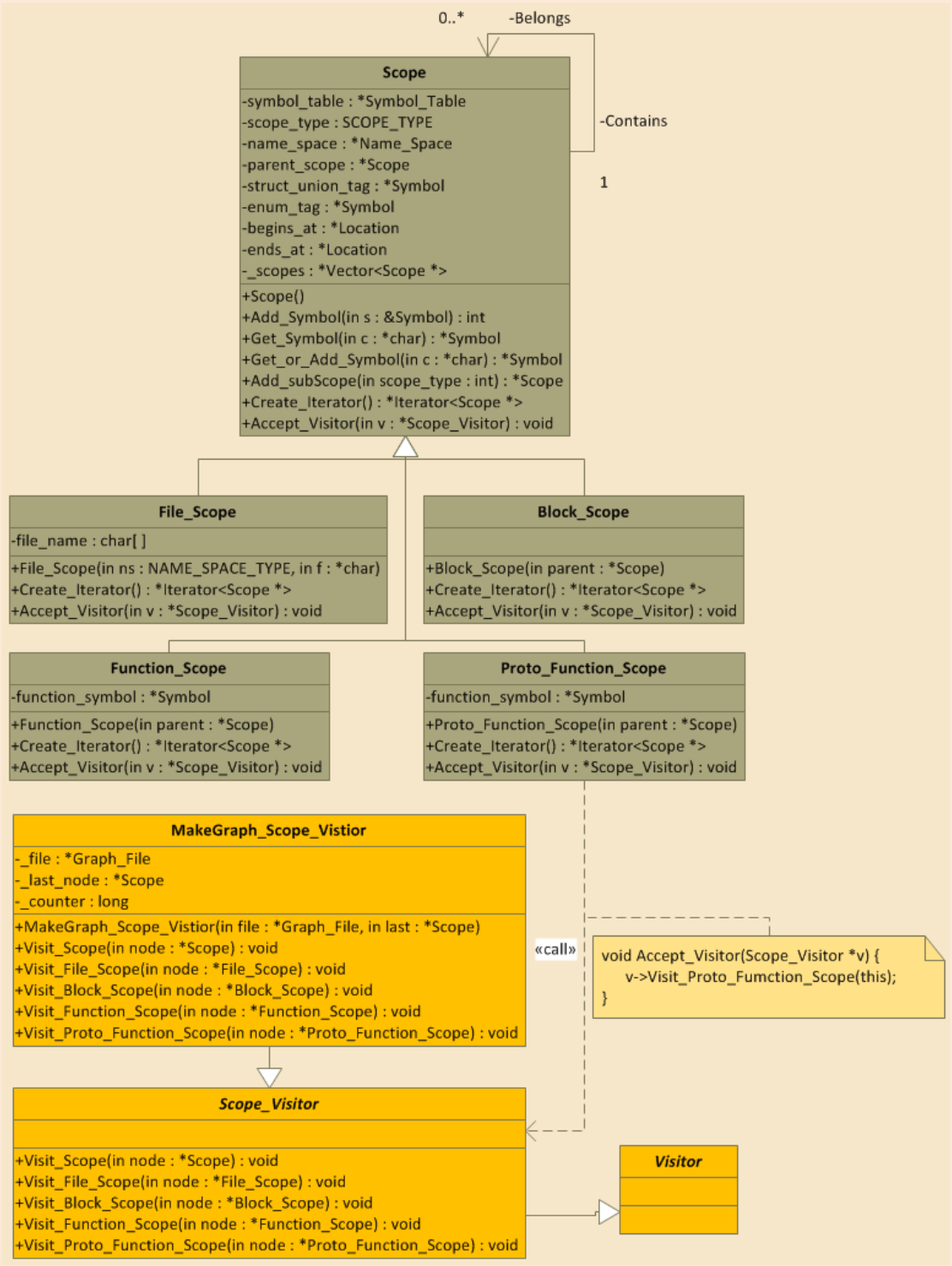
Οι εντολές (της Dot Language) σχηματισμού του γράφου παράγονται βηματικά κατά την preorder διαπέραση της δενδροειδούς δομής, όπου κάθε κόμβος (Scope) ανάλογα με τον τύπο του, εκτελεί μία λειτουργία παραγωγής ενός νέου κόμβου και μιας νέας ακμής.

7.5.3.1 Σχεδίαση Visitor

Δεδομένου ότι η δομή των Scopes είναι σταθερή και με ανοικτό το ενδεχόμενο για εφαρμογή / εκτέλεση επιπλέων λειτουργιών επί των κόμβων της δομής, ενδείκνυται η εφαρμογή του σχεδιαστικού προτύπου (design pattern) visitor για την υλοποίηση της λειτουργίας παραγωγής εντολών dot language.

Εικόνα 7-7 : Διάγραμμα κλάσεων Visitor για Scopes

Ο σχεδιασμός των Visitors αποτυπώνεται στο διάγραμμα κλάσεων στην Εικόνα 7-7, όπου με βάση το σχετικό σχεδιαστικό πρότυπο, για κάθε λειτουργία που θέλουμε να εφαρμόσουμε, δημιουργούμε μια κλάση (πχ MakeGraph_Scope_Visitor) όπου και περιλαμβάνουμε τόσες μεθόδους όσες και οι διαφορετικοί τύποι κλάσεων των κόμβων της δομής όπου εφαρμόζεται η λειτουργία. Κάθε μέθοδος εκτελεί την λειτουργία για το συγκεκριμένο τύπο κόμβου δομής, αναφορά του οποίου έχει λάβει ως παράμετρο. Στο συγκεκριμένο διάγραμμα παρουσιάζεται και μια επιπλέον κλάση (Process1_Scope_Visitor) για κάποια πιθανή επιπλέον λειτουργία που επιδρά στα scopes. Επίσης η abstract κλάση Scope_Visitor, ιεραρχείται κάτω από μια γενική abstract κλάση Visitor προκειμένου, όπως θα δούμε αργότερα, να μπορούμε να επεκτείνουμε την ιεραρχία με visitors που επιδρούν σε διαφορετικές δομές, χωρίς ωστόσο αυτό να είναι απαραίτητο.



Εικόνα 7-8 : Διάγραμμα κλάσεων Scope – Visitors

Στην Εικόνα 7-8, δίνεται το διάγραμμα κλάσεων με έμφαση στην αλληλεπίδραση μεταξύ της κλάσης Proto_Function_Scope και Scope_Visitor. Αν και η κλήση (εντός της Accept_Visitor) απευθύνεται στην γενική αφηρημένη κλάση Scope_Visitor, ο μηχανισμός της κληρονομικότητας θα εκτελέσει την κατάλληλη μέθοδο Visit_Proto_Function_Scope με βάση των εξειδικευμένο τύπο visitor (στη συγκεκριμένη περίπτωση τον MakeGraph_Scope_Visitor).

7.5.3.2 Εφαρμογή Visitor σε μεμονωμένο Scope

Αν επιθυμούμε την εκτέλεση μιας λειτουργίας (έστω της MakeGraph_Scope_Visitor) σε ένα μεμονωμένο κόμβο (έστω τύπου node: Proto_Function_Scope), τότε αρκεί η δημιουργία ενός αντικειμένου mgsv: MakeGraph_Scope_Visitor και η κλήση της μεθόδου **node->Accept_Visitor(operation)** του Proto_Function_Scope με παράμετρο τον συγκεκριμένο visitor. Η τελευταία μέθοδος, με τη σειρά της θα καλέσει την κατάλληλη μέθοδο του αντικειμένου MakeGraph_Scope_Visitor, δηλαδή θα κάνει κλήση της **mgsv->Visit_Proto_Function(this)** με παράμετρο την αναφορά στο ίδιο το αντικείμενο της Proto_Fynction_Scope, ώστε η λειτουργία να έχει πρόσβαση στο αντικείμενο του κόμβου που θα επισκεφθεί.

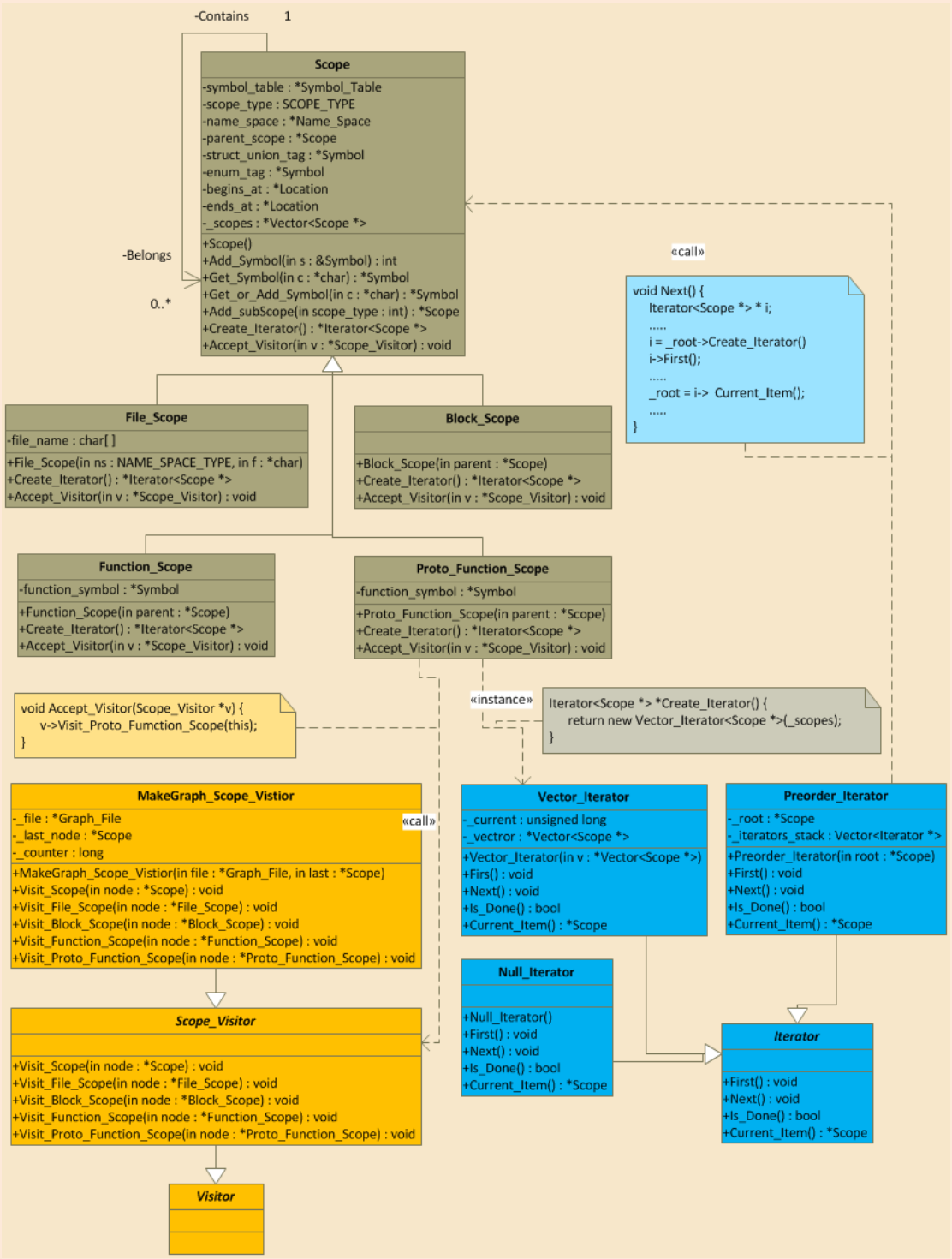
```
void Simple_Scope_Visitor_Application (Scope *node, Graph_File *gf) {
      MakeGraph_Scope_Visitor *mgsv = new MakeGraph_Scope_Visitor(gf, node);

      node->Accept_Visitor(mgsv);  // παράδειγμα εφαρμογής visitor
}
```

Αυτή η διπλή κλήση, χωρίς ο καλών να γνωρίζει τον τύπο του αντικειμένου και τον τύπο της λειτουργίας που θα εφαρμόσει και χωρίς η Accept_Visitor να γνωρίζει τον τύπο της λειτουργίας που αποδέχεται, αποτελεί και την δύναμη του συγκεκριμένου προτύπου που μας δίνει την δυνατότητα να συγκεντρώσουμε όλες τις μεθόδους (για κάθε τύπο κόμβου) μίας λειτουργίας σε μία και μοναδική κλάση με κοινό interface για κάθε λειτουργία. Την διπλή αυτή αντιστοίχηση των τύπων κόμβου (Scope) και λειτουργίας (Scope_Visitor) αναλαμβάνει να επιλύσει ο μηχανισμός κλήσεων των ιεραρχημένων κλάσεων μέσω της ιδιότητας της κληρονομικότητας και του πολυμορφισμού των μεθόδων.

7.5.3.3 Εφαρμογή Graph Visitor σε δομή Scopes (preorder διαπέραση)

Στο επόμενο βήμα για την παραγωγή του γράφου της δομής των scopes, απαιτείται η εφαρμογή της λειτουργίας MakeGraph_Scope_Visitor σε όλους τους κόμβους (scopes) της δομής με μια top-down διαπέραση, δηλαδή μια preorder διαπέραση. Στο σημείο αυτό απαιτείται η συνεργασία όλων των σχεδιαστικών προτύπων Composite, Iterator, Visitor, το συγκεντρωτικό διάγραμμα κλάσεων των οποίων παρουσιάζεται στην Εικόνα 7-9.



Εικόνα 7-9 : Διάγραμμα κλάσεων Scope - Iterator - Visitor

Για την εφαρμογή της λειτουργίας MakeGraph_Scope_Visitor σε κάθε κόμβο (scope) της δομής, τοποθετούμε τον κώδικα εκτέλεσης της λειτουργίας σε μεμονωμένο κόμβο, μέσα σε μια δομή επανάληψης preorder διαπέρασης με χρήση του Preorder_Iterator.

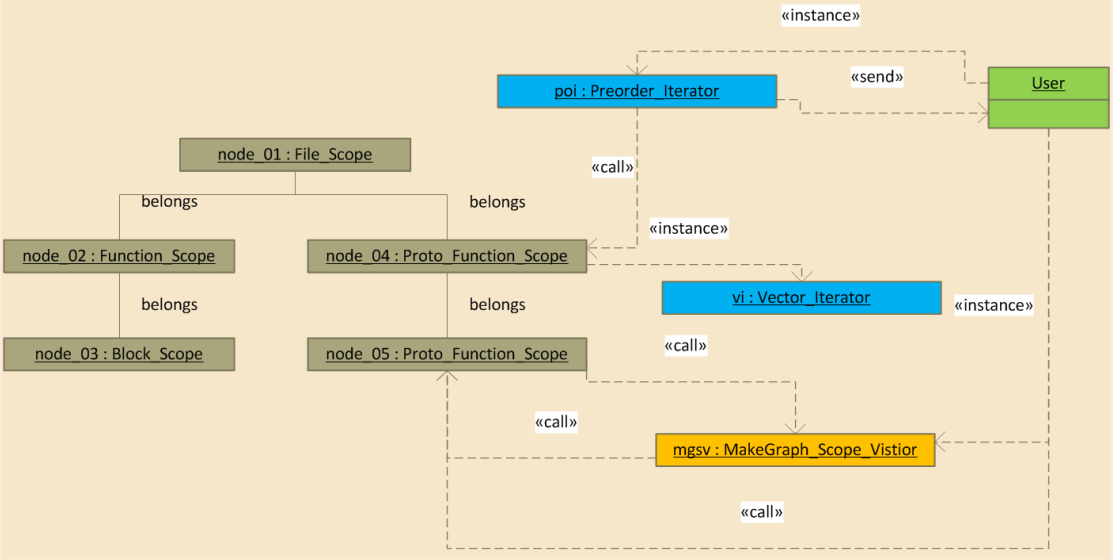
```
void Preorder_Scope_Visitor_Application (Scope *_root, Graph_File *gf) {
       MakeGraph_Scope_Visitor *mgsv =new MakeGraph_Scope_Visitor(gf, _root);
       Iterator<Scope *> *poi = new Preorder_Iterator<Scope *>(_root);

       _root->Accept_Visitor(mgsv);  // pre εφαρμογή visitor στη ρίζα
       for (poi->First();!poi->Is_Done();poi->Next()) {
             Scope *s = poi->Current_Item();
```

```
            s->Accept_Visitor(mgsv);   // παράδειγμα εφαρμογής visitor
        }
    }
```

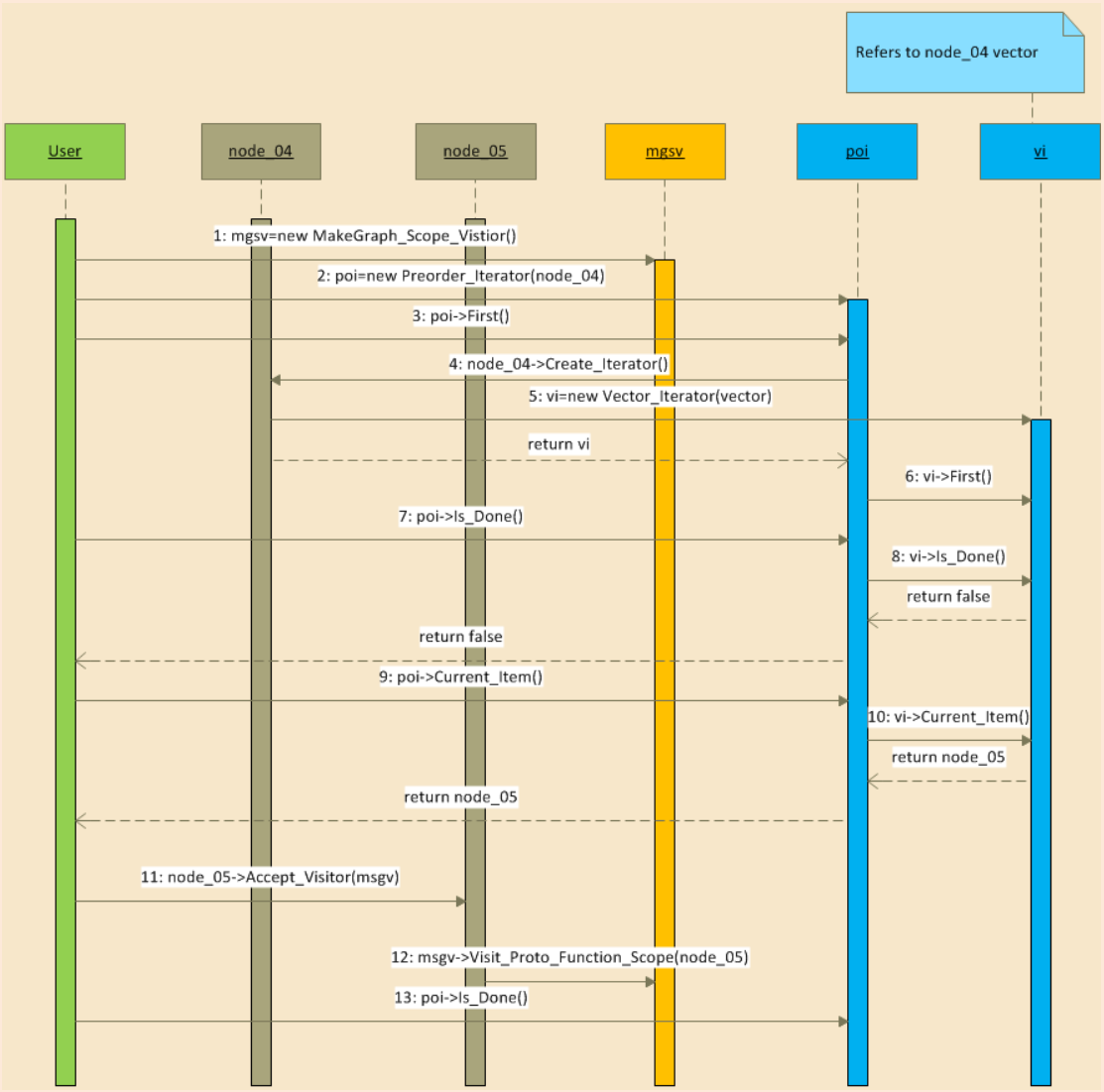
Στην πράξη πρόκειται για τον ίδιο κώδικα της preorder διαπέρασης, με τη διαφορά ότι για κάθε κόμβο (Scope) της δομής πραγματοποιείται κλήση της μεθόδου Accept_Visitor(mgv), όπου mgv το αντικείμενο visitor της συγκεκριμένης λειτουργίας.

Για την καλύτερη αναπαράσταση της λειτουργίας των σχεδιαστικών προτύπων iterator και visitor σε preorder διαπέραση, δίνεται ως παράδειγμα (Εικόνα 7-10), διάγραμμα αντικειμένων συγκεκριμένης δομής με αντικείμενα node_01, node_02, node_03, node_04, node_05, καθώς και τα αντικείμενα των εμπλεκόμενων iterators και visitor.



Εικόνα 7-10 : Διάγραμμα Αντικειμένων preorder iterator με visitor

Στο παράδειγμα θεωρούμε ότι κατά την δημιουργία του Preorder_Iterator(node_04) περνάμε ως παράμετρο το αντικείμενο node_04 το οποίο και θεωρείται ως ρίζα ή σημείο εκκίνησης της preorder διαπέρασης. Τέλος για την καλύτερη κατανόηση των μηνυμάτων (κλήσεων μεθόδων) μεταξύ των εμπλεκομένων αντικειμένων, παρατίθεται σχετικό διάγραμμα ακολουθίας (Εικόνα 7-11).



Εικόνα 7-11 : Διάγραμμα Ακολουθίας preorder iterator με visitor

Παρατηρούμε ότι η δομή των σχεδιαστικών προτύπων composite, iterator και visitor αποκρύπτουν από τον χρήστη όλες τις κλήσεις και διεργασίες σχετικά με την διαπέραση που υλοποιείται. Ο χρήστης απλά βλέπει το απλό interface (First, Next, Is_Done, Current_Item) ενός iterator και καλεί την Accept_Visitor(v) των αντικειμένων. Η όλη διαπέραση υλοποιείται από τον χρήση με μια απλή δομή επανάληψης, με κυκλωματική πολυπλοκότητα αλγόριθμου μόλις δύο (2). Παράδειγμα της παραγόμενης γραφικής αναπαράστασης ενός symbol table δίνεται στο Παράρτημα III : Παραδείγματα (σελίδα 172).

7.6 Λεκτική & Συντακτική ανάλυση με symbol table

Επανερχόμενοι στα αναφερόμενα στις ενότητες 7.3 & 7.4, στη συνέχεια παρουσιάζεται ενδεικτικά ο τρόπος που πραγματοποιείται η σημασιολογική ανάλυση του scope checking καθώς και η αναγνώριση των TYPE_NAME tokens, κατά τη λεκτική και συντακτική ανάλυση, εκμεταλλευόμενοι τον symbol table και τις ενέργειες των REs και CFG Rules της γλώσσας.

7.6.1 Λεκτική ανάλυση με symbol table

Προκειμένου να είναι δυνατός ο διαχωρισμός ενός Identifier από ένα Type name token από τον λεκτικό αναλυτή (scanner, Flex) απαιτείται η προσθήκη κατάλληλου κώδικα (ενέργειας) στην σχετική RE αναγνώρισης. Αρχικά θα αναζητείται στον symbol table ο υπό

αναγνώριση identifier και αν αυτός υπάρχει και είναι δηλωμένος ως typedef, τότε θα επιστρέφεται Type Name token, διαφορετικά Identifier token.

```
({LETTER_}|{U_CHAR})({U_CHAR}|{LETTER_}|{DEC_DIGIT})* {
	/* identifier or type name*/
	Symbol *_temp_symbol;
	_temp_symbol = driver.symbol_table.Is_Symbol_Type_Name (yytext);
	if (_temp_symbol) {
		. . . . .
		return token::TYPE_NAME;
	} else {
		. . . . .
		return token::IDENTIFIER;
	}
}
```

7.6.2 Συντακτική ανάλυση με symbol table

Για το scope checking και την αναγνώριση και καταχώριση στο symbol table όλων των Identifiers, από τον συντακτικό αναλυτή (parser, Bison), απαιτείται μια σειρά από ενέργειες σε κατάλληλα επιλεγμένους κανόνες της LR(1) CFG. Συγκεκριμένα πρέπει (τουλάχιστον) να αναγνωρίζεται η ενεργοποίηση των declaration specifies όπως typedef, extern, struct, union, enum, κλπ, ώστε να ληφθεί υπόψη κατά την εισαγωγή των identifiers στον symbol table, θα πρέπει να εκτελούνται ενέργειες για την δημιουργία και απομάκρυνση από τα scopes σε συγκεκριμένα name spaces, καθώς και να επιλύονται μια σειρά από διενέξεις σχετικά με τις struct και union δηλώσεις ή μεταξύ function και proto function scopes. Ενδεικτικά παρακάτω παρατίθεται η ενέργεια (κώδικας) ελέγχου των reference identifiers του κανόνα primary_expression, η οποία ελέγχει αν υπάρχει ο σχετικός identifier στο τρέχον scope και σε κάθε γονικό scope (επιστρέφει το πρώτο που βρίσκει) εντός του Ordinary name space. Αν βρεθεί, τότε το συσχετίζει με το τρέχον token, διαφορετικά εμφανίζει σχετικό σφάλμα.

```
primary_expression  : IDENTIFIER {
	// ******* PRIMARY EXPRESSION IDENTIFIER (REFERENCE) *****
	Symbol *_temp_symbol;
	_temp_symbol = driver.symbol_table.Get_Symbol_Name_Space
		(((CAST_IDENTIFIER *)$1)->Get_Token_String(), NST_ORDINARY);
	if ( _temp_symbol ){   // OK symbol (identifier) name is declared
		((CAST_IDENTIFIER *)$1)->Set_Symbol(_temp_symbol);
		. . . . .
		. . . . .
	} else {
		driver.error(yyloc,
		"Error: Reference to a non defined identifier name");
	}
}
```

Η αναλυτική παράθεση όλων των ενεργειών (κώδικα) κατά την συντακτική ανάλυση δεν αποτελεί στόχο της εργασίας, ωστόσο στο Παράρτημα Ι : C89 – Flex & Bison Grammar (σελ. 167), δίνεται μια σειρά σχηματικών αναπαραστάσεων της C89 LR(1) Grammar όπου και αντιστοιχίζονται οι σχετικές ενέργειες ανά κανόνα.

7.7 Parse / Abstract Trees

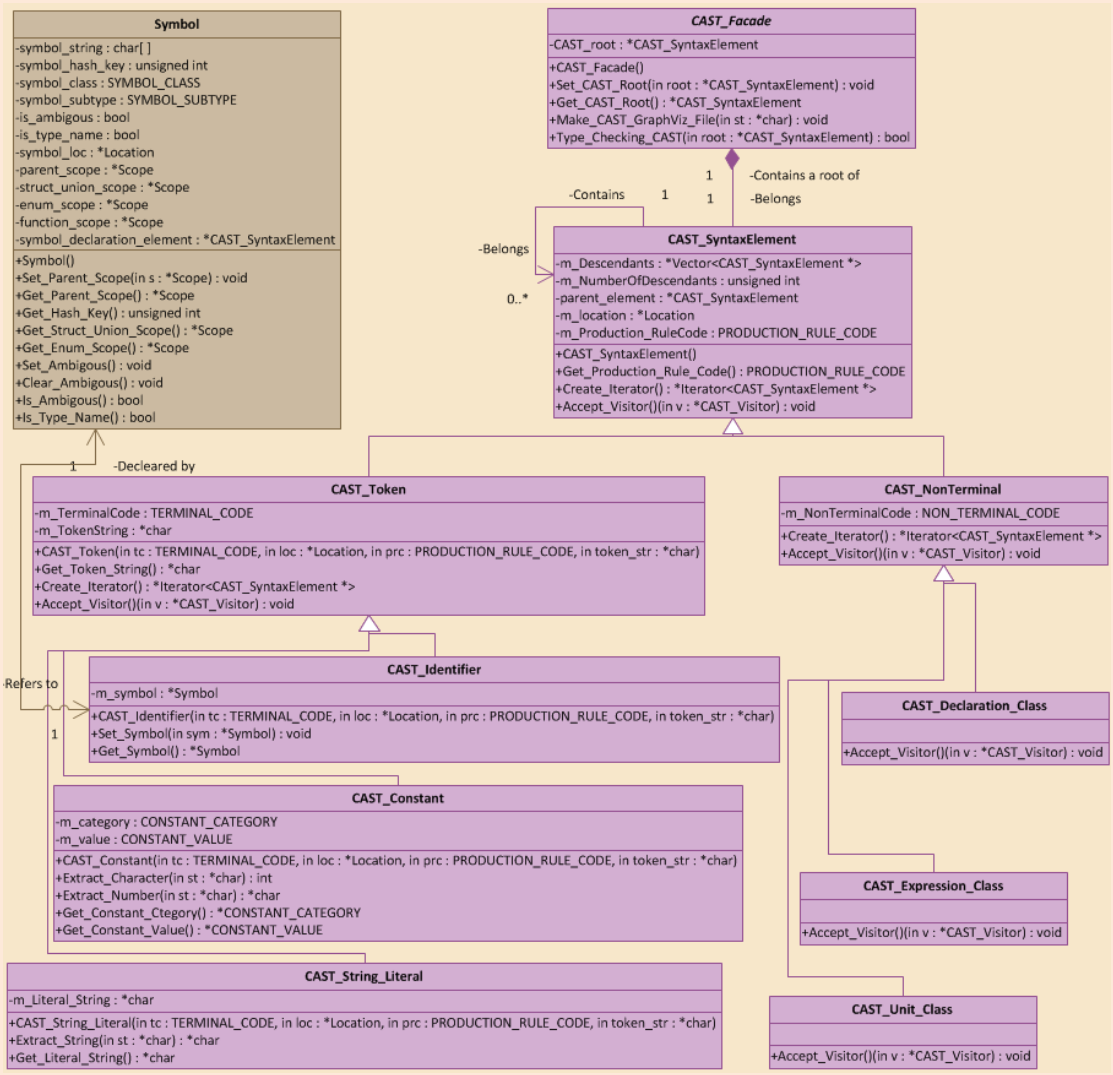
Λαμβάνοντας υπόψη τα τερματικά και μη τερματικά σύμβολα της γραμματικής της γλώσσας C89 (Παράρτημα Ι : C89 – Flex & Bison Grammar, σελ. 164) και το πρότυπο της C89 που αποτελούν μέρος του μοντέλου ανάλυσης, καθώς και τα σχεδιαστικά πρότυπα (design

patterns) Façade (ενότητα 4.3.4) και Composite (ενότητα 4.3.1), μεταβαίνουμε στο μοντέλο σχεδίασης, σχεδιάζοντας τη δομή του Parse-tree του μεταγλωττιστή με χρήση UML Class Diagrams (ενότητα 2.3.2).

7.7.1 Σχεδίαση Parse-Tree με Composite και Façade

Προκειμένου το interface του parse-tree να είναι ενιαίο, σχεδιάσθηκε μια κλάση (CAST_Facade) η ο ποία και συγκεντρώνει τις λειτουργίες του (ενδεικτικά περιλαμβάνει τις λειτουργίες σχεδιασμού γράφου δέντρου και ελέγχου τύπων των εκφράσεων). Το parse-tree αποτελείται ουσιαστικά από στοιχεία της κλάσης CAST_SyntaxElement όπου κάθε αντικείμενό της μπορεί να περιέχει άλλα CAST_SyntaxElement αντικείμενα διαχωριζόμενα σε δύο επιμέρους τύπους (subclasses CAST_Token για τα τερματικά, CAST_NonTerminal για τα μη τερματικά στοιχεία της γραμματικής). Περεταίρω κάθε τερματικό CAST_Token εξειδικεύεται σε τρις επιμέρους τύπους (subclasses CAST_Identifier, CAST_Constant, CAST_String_Literal) και κάθε CAST_Non_Terminal σύμβολο σε άλλους τρις (subclasses CAST_Declaration_Class, CAST_Expressin_Class, CAST_Unit_Class) για ομάδες μη τερματικών συμβόλων που σχετίζονται με τις δηλώσεις τύπων, εκφράσεων, και δομικών-συστατικών στοιχείων της γλώσσας/γραμματικής αντίστοιχα. Γενικά κάθε CAST_SyntaxElements αντιστοιχεί σε έναν κανόνα της LR(1) γραμματικής της γλώσσας και εξειδικεύεται ανάλογα με την λειτουργία και τον σκοπό του κανόνα.

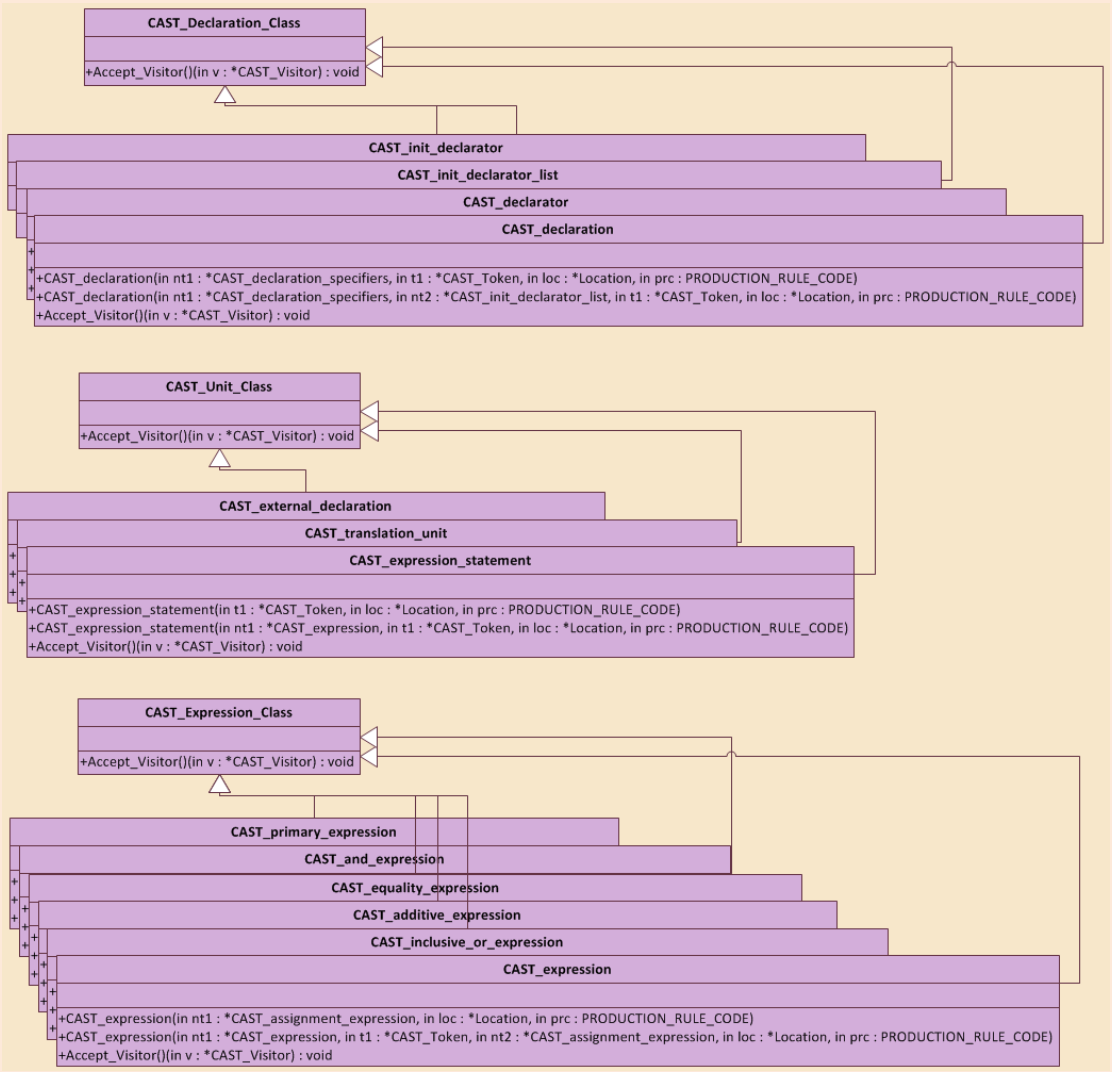
Το διάγραμμα κλάσεων του parse-tree που περιγράφηκε παρουσιάζεται στην Εικόνα 7-12, όπου η ομαδοποίηση των αντικειμένων είναι ενδεικτική και φυσικά μπορεί να διαφέρει από υλοποίηση σε υλοποίηση. Με την υπάρχουσα δομή μία λειτουργία που εφαρμόζεται μέσω του σχεδιαστικού προτύπου visitor μπορεί να εφαρμοσθεί σε μία ομάδα κανόνων (π.χ. CAST_Token, CAST_NonTerminal, CAST_Declaration_Class) αλλά όχι στην ομάδα των CAST_Identifier (αυτό μπορεί να επιτευχθεί με την προσθήκη μιας Accept_Vistior μεθόδου στην εν λόγω κλάσης καθώς και αντίστοιχης μεθόδου Visit_CAST_Identifier στις κλάσεις των λειτουργιών των visitors). Η σχεδίαση περιέχει όλες τις απαραίτητες λειτουργίες (methods) για την δημιουργία iterators και την αποδοχή visitors καθώς και ειδικές λειτουργίες για την επεξεργασία των constants (πχ Extract_Number). Αυτές οι λειτουργίες επιλέγονται να ορισθούν εντός της κλάσης (και όχι μέσω visitor) προκειμένου, ως βασικές λειτουργίες, να καλούνται άμεσα (π.χ. από τον constructor της κάθε κλάσης). Ωστόσο, όπως θα εξετάσουμε παρακάτω, είναι δυνατό η λειτουργία να σχεδιαστεί μέσω visitor και να εκτελεσθεί μεμονωμένα ή και συνδυασμό με άλλες λειτουργίες.



Εικόνα 7-12 : Διάγραμμα κλάσεων Parse-tree

Επιπλέον στο διάγραμμα κλάσεων του parse-tree (Εικόνα 7-12) παρουσιάζεται και η συσχέτιση των identifier token (CAST_Identifier) του parse-tree με τα σύμβολα (Symbol) του πίνακα συμβόλων του μεταγλωττιστή. Κάθε σύμβολο στον πίνακα συμβόλων έχει δηλωθεί (declared) από ένα identifier token της γλώσσας / γραμματικής και κάθε identifier token (δήλωση ή αναφορά) του προγράμματος αντιστοιχεί σε ένα σύμβολο στον πίνακα συμβόλων.

Κατά την εφαρμογή – εκτέλεση διαφόρων λειτουργιών (π.χ. type checking) στα στοιχεία (κανόνες) του parse-tree, παρουσιάζεται συνήθως η ανάγκη διαφορετικής αντιμετώπισης για κάθε κανόνα. Συνεπώς η εξειδίκευση των κλάσεων των μη τερματικών συμβόλων μπορεί να προχωρήσει μέχρι και την δημιουργία μίας subclass για κάθε κανόνα όπως παρουσιάζεται στην Εικόνα 7-13 (για λόγους οικονομίας χώρου παρουσιάζονται κλάσεις μέρους των κανόνων της γραμματικής). Εδώ η μέθοδος Accept_Visitor() υλοποιείται για κάθε (sub class) κανόνα προκειμένου να είναι δυνατή η διαφορετική αντιμετώπιση τους από τις λειτουργίες των visitor.



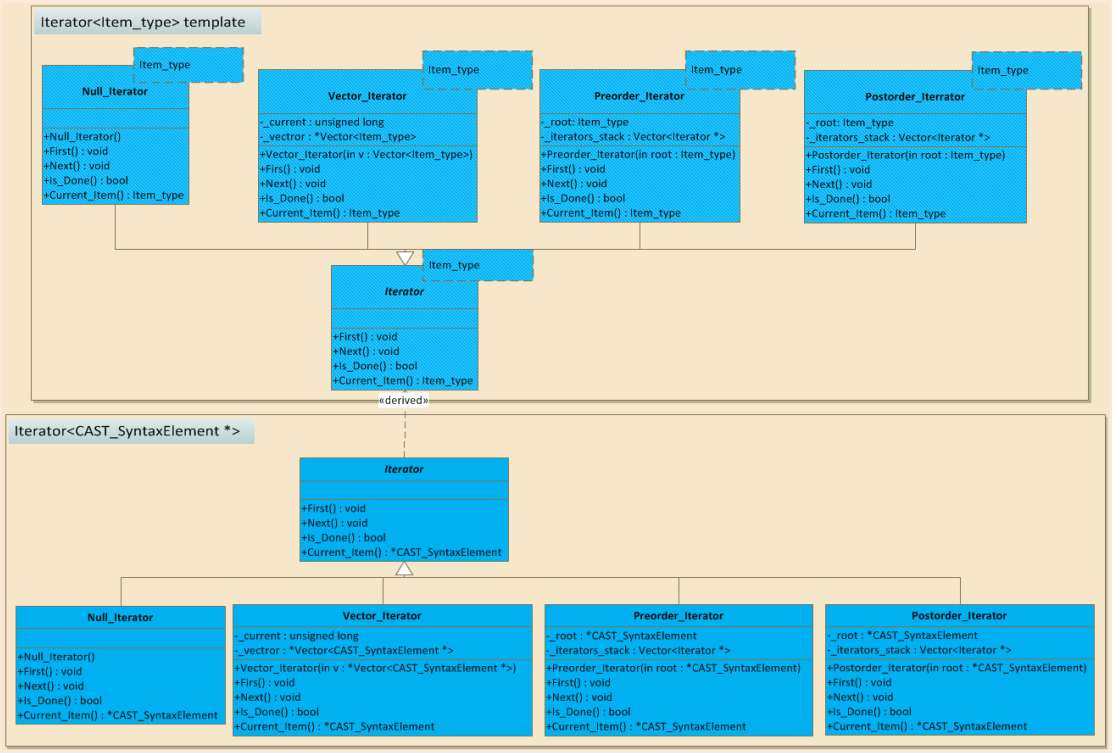
Εικόνα 7-13 : Διάγραμμα κλάσεων Parse-tree (σχεδίαση / κανόνα)

Και πάλι τα CAST_SyntaxElements οργανώνονται ως μια δενδροειδή μορφή όπου αυτή τη φορά τα CAST_NonTerminal αντικείμενα είναι μόνο nodes και τα CAST_Token μπορεί να είναι μόνο leafs. Κάθε CAST_SyntaxElement περιέχει ένα πίνακα Vector<CAST_SyntaxElement *>, με δείκτες προς τα CAST_SyntaxElement παιδιά του.

7.7.2 Διαπέραση δομής Parse-Tree με Iterators

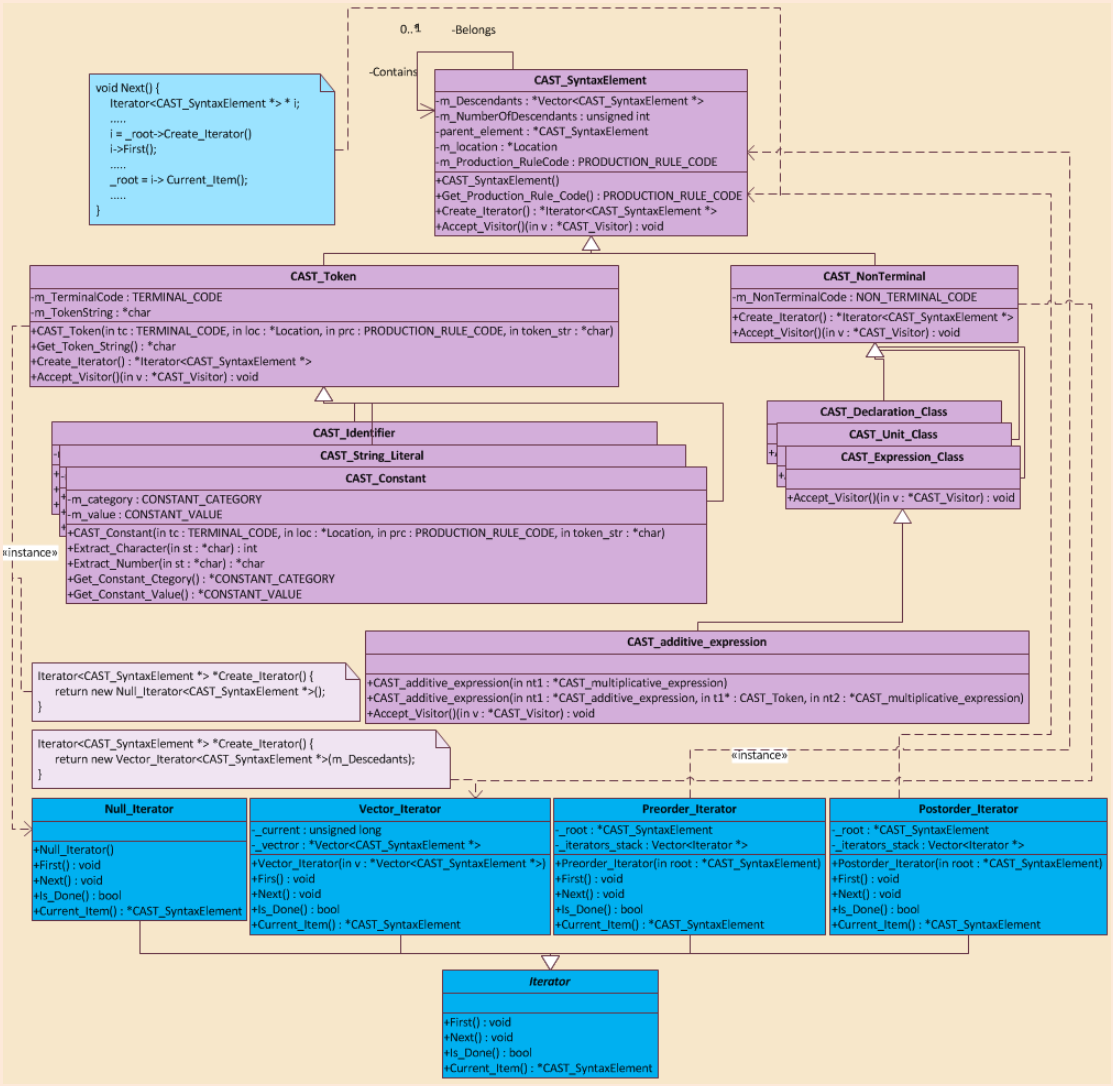
7.7.2.1 Σχεδίαση Iterators / Postorder Iterator

Για την διαπέραση της δομής του parse-tree από CAST_SyntaxElements χρησιμοποιούμε το ίδιο template των Null_Iterator, Vector_Iterator και Preorder_Iterator της ενότητας 7.5.2.1. με κατάλληλη παράμετρο Item_type. Η σχεδίαση των Iterator δομών τόσο του template όσο και του στιγμιότυπου του για Item_type = CAST_SyntaxElement* παρουσιάζεται στην Εικόνα 7-14.



Εικόνα 7-14 : Διάγραμμα κλάσεων Iterator templates για CAST_SyntaxElements

Η σχεδίαση του (ανοικτού) προτύπου της ενότητας 7.5.2.1 έχει επεκταθεί με την προσθήκη μίας κλάσης για post order (bottom up, ενότητα 4.3.2.2) διαπεράσεις με όνομα Postorder_Iterator. Στο σημείο αυτό είναι εμφανής η δυνατότητα επαναχρησιμοποίησης του προτύπου (template) των iterators για δενδροειδείς (composite) δομές δεδομένων με διαφορετικά μάλιστα στοιχεία. Δηλαδή από το ίδιο τμήμα κώδικα (template) παράγονται εύκολα iterators για διαφορετικού τύπου (Item_type) στοιχεία. Η αλληλεπίδραση των κλάσεων CAST_SyntaxElement και Iterators παρουσιάζεται (ενδεικτικά) στο διάγραμμα κλάσεων στην Εικόνα 7-15.



Εικόνα 7-15 : Διάγραμμα κλάσεων Parse tree - Iterators

Στην κλάση CAST_SyntaxElement έχει δηλωθεί η μέθοδος Create_Iterator(), και υλοποιείται μόνο από τις υπό κλάσεις CAST_Token και CAST_NonTerminal, όπου η πρώτη υλοποίηση επιστρέφει έναν Null_Iterator (εφόσον τα tokens είναι τερματικά σύμβολα ή φύλα του δέντρου) και η δεύτερη έναν Vector_Iterator (ως μη τερματικά σύμβολα ή κόμβοι του δέντρου). Με αυτόν τον τρόπο κάθε υπό τύπος της κλάσης CAST_SyntaxElement επιστρέφει τον κατάλληλο Iterator, με κλήση της ίδιας μεθόδου (Create_Iterator), εκμεταλλευόμενος τις ιδιότητες του πολυμορφισμού και της κληρονομικότητας των κλάσεων. Αντίστοιχα με τον Preorder_Iterator, η λειτουργία Next της κλάσης Postorder_Iterator είναι αυτή που περιέχει τον αλγόριθμο που υλοποιεί την postorder διαπέραση, χρησιμοποιώντας επίσης μια stack όπου και αποθηκεύει τα ενδιάμεσα αντικείμενα (syntax elements), προκειμένου να είναι σε θέση να επανέλθει σε ένα αντικείμενο μετά την διαπέραση όλων των παιδιών του.

7.7.2.2 Υλοποίηση Postorder Itrator

Επειδή η υλοποίηση του Iterator που παρουσιάσθηκε στην σχεδίαση έχει γενικά ενιαία δομή και ευρεία χρήση, παρακάτω δίνεται ο ενδεικτικός κώδικας της υλοποίησης της νέας κλάσης Postorder_Iterator (σε συνέχεια των υλοποιήσεων της ενότητας 7.5.2.2).

```
template <class ItemType>
class Postorder_Iterator: public Iterator<ItemType> {
private:
	ItemType _root;
	vector<Iterator*> *_iterators_stack;
	Iterator* _i;
public:
	Postorder_Iterator(ItemType root){
		_root=root;
		_iterators_stack =
			new vector<Iterator*> (POSTORDER_ITERATOR_STACK_SIZE);
	};
	void First(){
		_i = _root->Create_Iterator();
		if (_i) {
			_iterators_stack->clear();
			_i->First();
			while (!_i->Is_Done()) {
				_iterators_stack->push_back(_i);
				_i=_i->Current_Item()->Create_Iterator();
				_i->First();
			}
		}
	};
	void Next(){
		_i=_iterators_stack->back();
		_i->Next();
		if (!_i->Is_Done()) {
			_i =
		_iterators_stack->back()->Current_Item()->Create_Iterator();
			_i->First();
		}
		else {
			delete _iterators_stack->back();
			_iterators_stack->pop_back();
			return;
		}
		while (!_i->Is_Done()) {
			_iterators_stack->push_back(_i);
			_i=_i->Current_Item()->Create_Iterator();
			_i->First();
		}
	};
	bool Is_Done(){ return (_iterators_stack->empty()); };
	ItemType Current_Item() {
		if (_iterators_stack->size()>0)
			return _iterators_stack->back()->Current_Item();
	};
};
```

7.7.2.3 *Εφαρμογή vector Iterator*

Για την απλή διαπέραση των sub αντικειμένων ενός CAST_SyntaxElement αντικειμένου, έστω _root, αρκεί μια ενιαία δομή επανάληψης :

```
void Simple_SyntaxElement_Iteration_Application (CAST_SyntaxElement *_root) {
	Iterator<CAST_SyntaxElement *> *vi = _root->Create_Iterator();
	for (vi->First();!vi->Is_Done();vi->Next()) {
		CAST_SyntaxElement *s = vi->Current_Item();
		. . . . .
		int prc = s->Get_Production_Rule_Code(); //παράδειγμα πρόσβασης
		. . . . .
	}
}
```

Η πρόσβαση στα επιμέρους αντικείμενα (s) του _root πραγματοποιείται εύκολα και χωρίς ο χρήσης να γνωρίζει την εξειδίκευση του sub elements (πχ CAST_Identifier). Η κλήση

οποιασδήποτε πολυμορφικής μεθόδου του αντικειμένου s (CAST_SyntaxElement), οδηγείται μέσω της κληρονομικότητας των κλάσεων στην αντίστοιχη υλοποίηση μεθόδου της πιο εξειδικευμένης κλάσης.

7.7.2.4 Εφαρμογή Preorder Iterator / Postorder Iterator

Αντίστοιχα, για την preorder / postorder διαπέραση μίας δενδροειδής δομής από CAST_SyntaxElement αντικείμενα με ρίζα το αντικείμενο _root, αρκεί επίσης μια ενιαία δομή επανάληψης:

```
void Preorder_SyntaxElement_Iteration_Application (CAST_SyntaxElement *_root)
{
	Iterator<CAST_SyntaxElement *> *poi =
			new Preorder_Iterator< CAST_SyntaxElement *>(_root);

	_root->Get_Production_Rule_Code; // pre πρόσβαση στην ρίζα
	for (poi->First();!poi->Is_Done();poi->Next()) {
		Scope *s = poi->Current_Item();
		. . . . .
		int prc = s->Get_Production_Rule_Code(); //παράδειγμα πρόσβασης
		. . . . .
	}
}

void Postorder_SyntaxElement_Iteration_Application (CAST_SyntaxElement
*_root) {
	Iterator<CAST_SyntaxElement *> *poi =
			new Postorder_Iterator< CAST_SyntaxElement *>(_root);

	for (poi->First();!poi->Is_Done();poi->Next()) {
		CAST_SyntaxElement *s = poi->Current_Item();
		. . . . .
		int prc = s->Get_Production_Rule_Code(); //παράδειγμα πρόσβασης
		. . . . .
	}
	_root->Get_Production_Rule_Code; // post πρόσβαση στην ρίζα
}
```

Πάλι για την εκκίνηση της preorder/postorder διαπέρασης αρκεί η απευθείας δημιουργία ενός αντικειμένου Preorder_Iterator/Postorder_Iterator με παράμετρο τη ρίζα (_root) της δενδροειδής δομής. Η πρόσβαση στα επιμέρους αντικείμενα (s, CAST_SyntaxElement) του δένδρου πραγματοποιείται εύκολα και χωρίς ο χρήσης να γνωρίζει την εξειδίκευση του sub CAST_SyntaxElement (πχ CAST_additive_expression) του κάθε κόμβου. Η κλήση οποιασδήποτε πολυμορφικής μεθόδου του αντικειμένου s (syntax element), οδηγείται μέσω της κληρονομικότητας των κλάσεων στην αντίστοιχη υλοποίηση μεθόδου της πιο εξειδικευμένης κλάσης. Από το παράδειγμα διαφαίνεται πάλι η εντυπωσιακή απλότητα του παραγόμενου κώδικα, για μία κατά τα άλλα περίπλοκη διαπέραση μιας Composite δενδροειδούς δομής αντικειμένων.

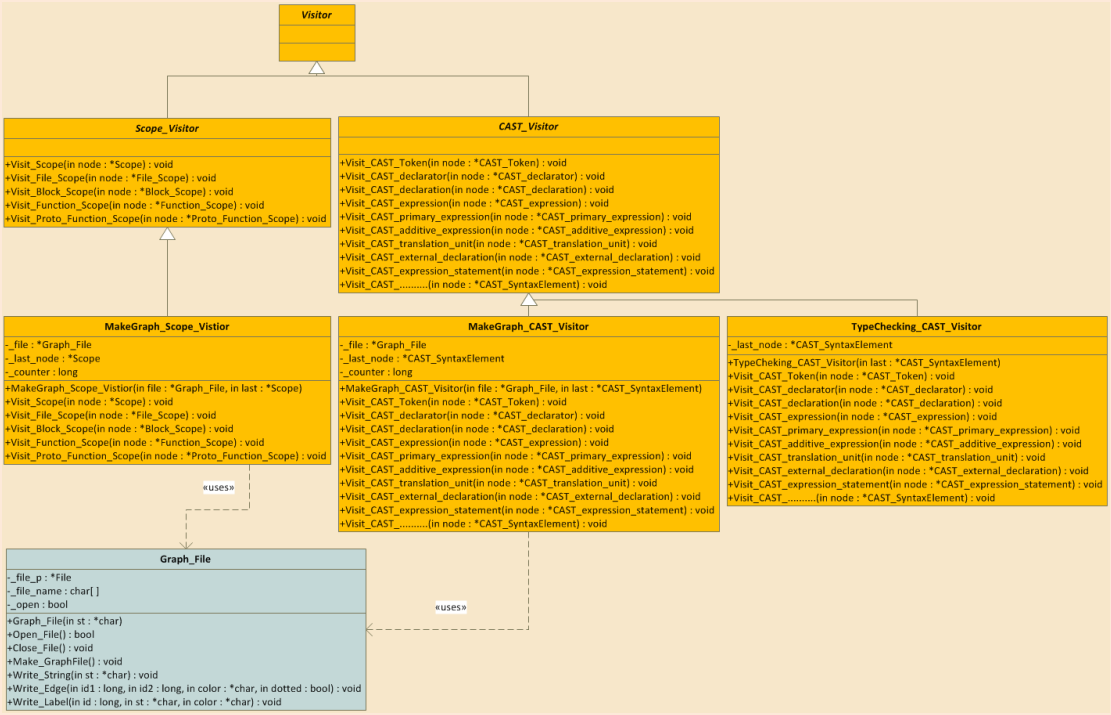
7.7.3 Κατασκευή Parse_Tree Graph με Visitors

Αντίστοιχα με τον symbol table, παρουσιάζεται η λειτουργία δημιουργίας ενός γράφου για την εξαγωγή της δομής των CAST_SyntaxElements του Parse-tree προς τον χρήστη, βασιζόμενοι στην υπάρχουσα κλάση διεπαφής Graph_File (Εικόνα 7-6).

7.7.3.1 Σχεδίαση Visitor

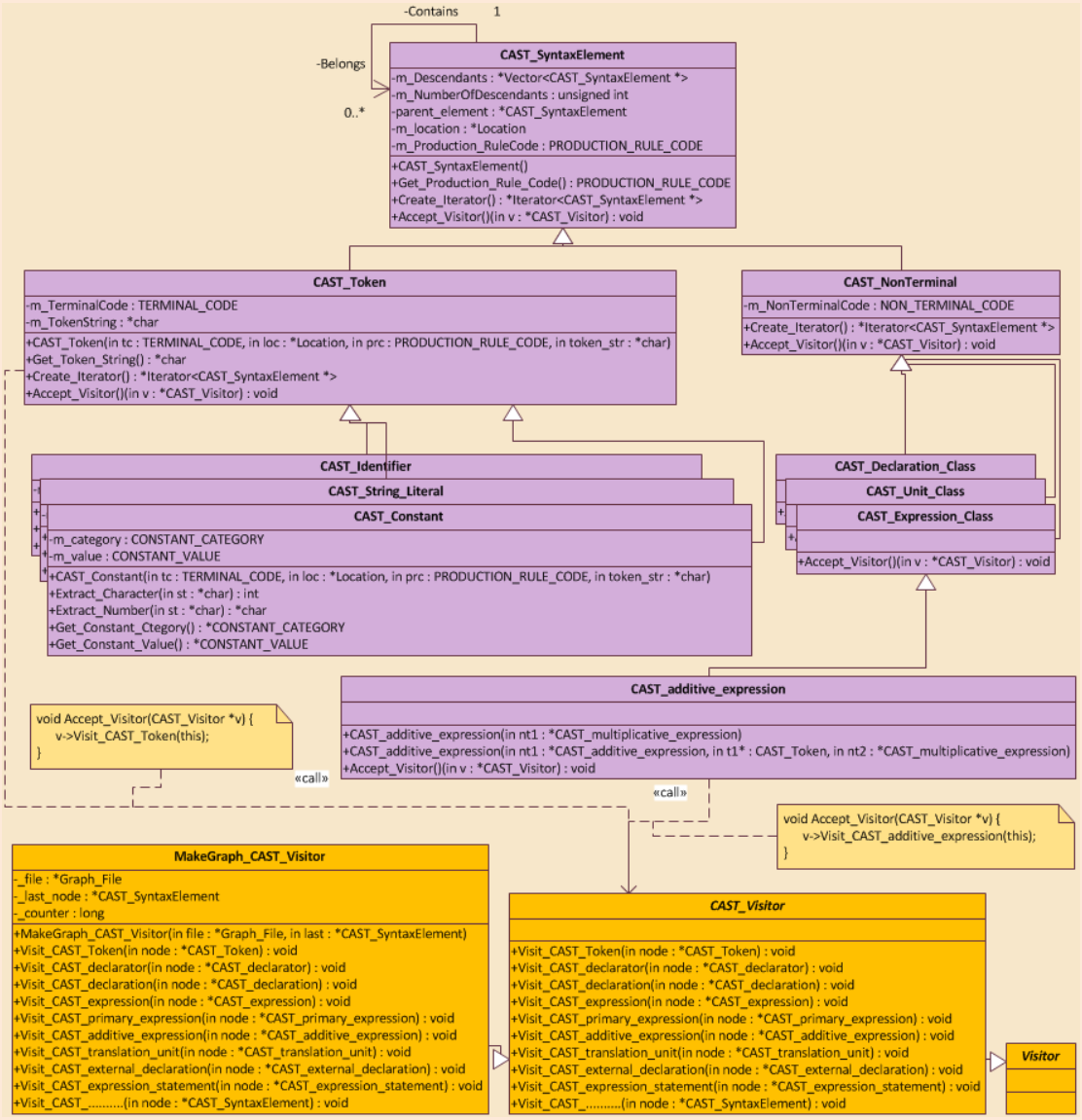
Δεδομένου ότι η δομή του parse-tree (δηλαδή της γραμματικής της γλώσσας) είναι σταθερή και με ανοικτό το ενδεχόμενο για εφαρμογή / εκτέλεση επιπλέων λειτουργιών επί

των κόμβων της δομής, ενδείκνυται η εφαρμογή του σχεδιαστικού προτύπου (design pattern) visitor για την υλοποίηση της λειτουργίας παραγωγής εντολών dot language. Προτιμάται δε η επέκταση της σχεδίασης του αντίστοιχου visitor της ενότητας 7.5.3.1.



Εικόνα 7-16 : Διάγραμμα κλάσεων Visitor για Scopes & CAST Syntax elements

Ο νέος σχεδιασμός των Visitors αποτυπώνεται στο διάγραμμα κλάσεων στην Εικόνα 7-16, όπου με βάση το σχετικό σχεδιαστικό πρότυπο, για κάθε λειτουργία που θέλουμε να εφαρμόσουμε, δημιουργούμε μια κλάση (πχ MakeGraph_CAST_Visitor) όπου και περιλαμβάνουμε τόσες μεθόδους όσες και οι διαφορετικοί τύποι κλάσεων των κόμβων της δομής όπου εφαρμόζεται η λειτουργία. Κάθε μέθοδος εκτελεί την λειτουργία για το συγκεκριμένο τύπο κόμβου δομής, αναφορά του οποίου έχει λάβει ως παράμετρο. Στο συγκεκριμένο διάγραμμα παρουσιάζεται και μια επιπλέον κλάση (Type_Checking_CAST_Visitor) ως μια επιπλέον λειτουργία που επιδρά στα syntax elements.



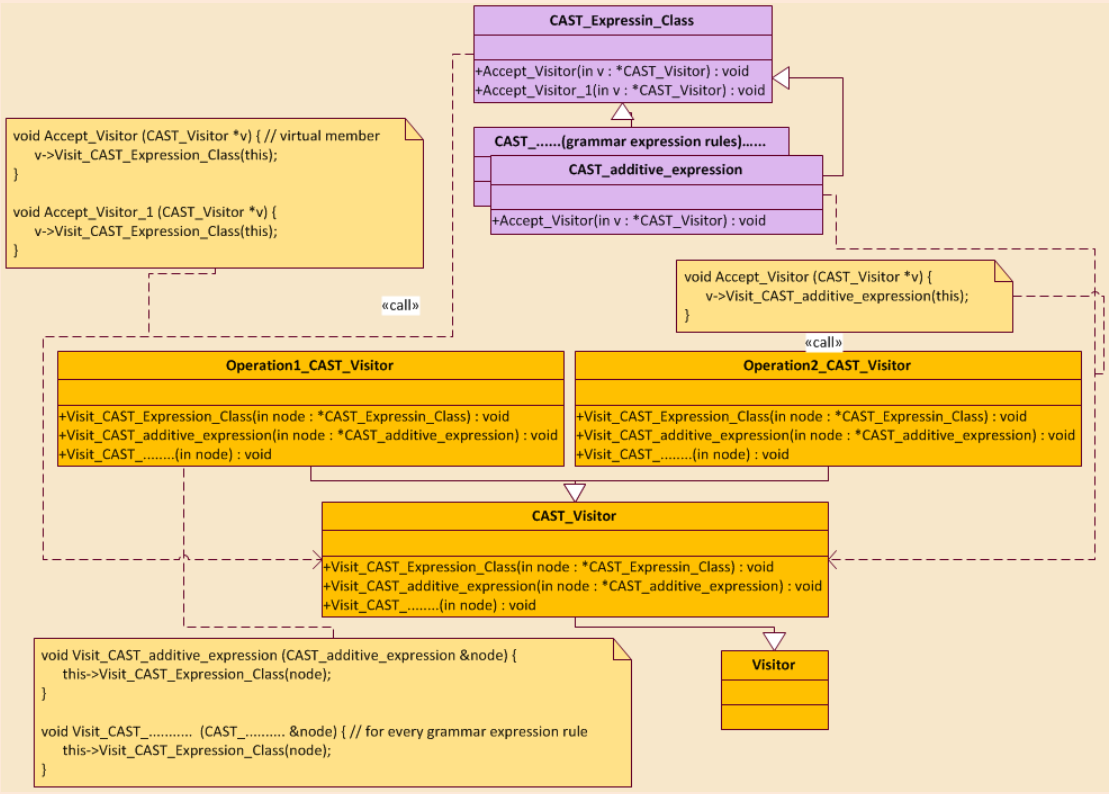
Εικόνα 7-17 : Διάγραμμα κλάσεων Parse tree - Visitors

Στην Εικόνα 7-17, δίνεται το διάγραμμα κλάσεων με έμφαση στην αλληλεπίδραση μεταξύ των κλάσεων CAST_Token, CAST_additive_expression και CAST_Visitor. Αν και οι κλήσεις (εντός της Accept_Visitor) απευθύνονται στην γενική αφηρημένη κλάση CAST_Visitor, ο μηχανισμός της κληρονομικότητας θα εκτελέσει την κατάλληλη μέθοδο Visit_CAST_Token ή Visit_CAST_additive_expression με βάση των εξειδικευμένο τύπο visitor (στη συγκεκριμένη περίπτωση τον MakeGraph_CAST_Visitor).

Είναι σημαντικό να επισημανθεί ότι στην συγκεκριμένη σχεδίαση έχει επιλεχθεί η υλοποίηση της Accept_Visitor() στην κλάση CAST_Token και όχι στις sub classes όπως CAST_Constant. Αυτό σημαίνει ότι τα αντικείμενα των sub classes (CAST Constant, Identifier, String literal) θα αντιμετωπίζονται από οποιαδήποτε λειτουργία που εφαρμόζεται μέσω ενός CAST_Visitor, ενιαία ως CAST_Token τύπου αντικείμενα. Αντίθετα έχει επιλεχθεί η υλοποίηση της Accept_Visitor() σε όλες τις sub classes της CAST_NonTerminal κλάσης, όπως CAST_additive_expression. Αυτό σημαίνει ότι τα αντικείμενα των sub classes (CAST additive expression, κλπ) θα αντιμετωπίζονται από οποιαδήποτε λειτουργία που εφαρμόζεται μέσω ενός CAST_Visitor, ως ανεξαρτήτου τύπου αντικείμενα. Ακόμα και αν υπάρχει υλοποίηση της Accept_Visitor() στην κλάση πχ CAST_Expreesion_Class, αυτή δεν μπορεί να εκτελεσθεί

από ένα αντικείμενο πχ. CAST_additive_expression, εφόσον ο μηχανισμός της κληρονομικότητας και του πολυμορφισμού επιλέγει αυτόματα την εκτέλεση της (virtual) μεθόδου του πιο εξειδικευμένου αντικειμένου της μεταξύ τους ιεραρχίας. Όπως παρουσιάζεται και στην Εικόνα 2-11 (σελ. 33), προκειμένου ο visitor να εκμεταλλευτεί τον πολυμορφισμό, σε όλες τις κλάσεις ενδιάμεσα της ιεραρχίας η μέθοδος Accept_Visitor δηλώνεται ως virtual member (με ή χωρίς υλοποίηση). Συνεπώς, με την παρούσα σχεδίαση, δεν είναι δυνατή η εφαρμογή μιας ενιαίας λειτουργίας στα αντικείμενα των subclasses της πχ CAST_Expreesion_Class και θα πρέπει να κατασκευάζονται ξεχωριστές μέθοδοι για κάθε παραγόμενη κλάση (πχ Visit_CAST_additive_expession) εντός όλων των CAST_Visitor. Από τα παραπάνω είναι εμφανές ότι ειδικά σε σύνθετου τύπου δομές, όπως ένα parse tree, είναι καθοριστικής σημασίας η σχεδίαση της ιεραρχίας μεταξύ των κλάσεων καθώς και τα επίπεδα εντός των οποίων επιδρούν τα σχεδιαστικά πρότυπα.

Προκύπτει το ερώτημα πως είναι δυνατή η αποδοτική εφαρμογή μιας ενιαίας λειτουργίας σε όλες τις παραγόμενες κλάσεις μίας γονικής κλάσης και η εφαρμογή μιας άλλης λειτουργίας με διαφορετική αντιμετώπιση ανά παραγόμενη κλάση, με χρήση κοινού σχεδιαστικού προτύπου visitor. Στην Εικόνα 7-18, προτείνεται μία συνοπτική σχεδιαστική λύση με δύο διαφορετικές προσεγγίσεις. Υποθέτοντας ότι η λειτουργία του Operation2_CAST_Visitor επιδρά μεμονωμένα και κατά περίπτωση σε αντικείμενα των παραγόμενων κλάσεων πχ. CAST_additive_expression, η υλοποίηση παραμένει ως έχει με την κλήση της Accept_Visitor για κάθε αντικείμενο που επισκεπτόμαστε. Υποθέτοντας ότι η λειτουργία του Operation1_CAST_Visitor επιδρά ενιαία σε όλα τα αντικείμενα των παραγόμενων κλάσεων πχ. CAST_additive_expression, α) η πιο απλή λύση είναι η συμπερίληψη κατάλληλης μεθόδου για τον τύπο της γονικής κλάσης (Visit_CAST_Expression_Class) στον visitor και έμμεση κλήση της, από όλες τις μεθόδους επίσκεψης των παραγόμενων κλάσεων (πχ. Visit_CAST_additive_expression) του συγκεκριμένου visitor, β) εναλλακτική λύση είναι η προσθήκη μίας επιπλέον μεθόδου Accept_Vistor_1 στις κλάσεις της δομής, με υλοποίηση μέχρι και την κλάση που αντιστοιχεί στο επιθυμητό επίπεδο ομαδοποίησης (CAST_Expression_Class), όπου και θα καλείται ο κατάλληλος τύπου visitor (Visit_CAST_Expression_Class).



Εικόνα 7-18 : Διάγραμμα κλάσεων εφ/γής visitor σε ομάδες/υποομάδες στοιχείων

Η πρώτη λύση απαιτεί την έμμεση κλήση της Visit_CAST_Expression_Class από όλες τις μεθόδους του visitor, ωστόσο παραμένει πιστή στο πρότυπο επιτρέποντας στο χρήστη να το χρησιμοποιεί με τον ίδιο τρόπο. Η δεύτερη λύση απαιτεί την προσθήκη μίας επιπλέον μεθόδου Accept_Visitor_1 στις κατάλληλες κλάσεις της δομής, επιτρέποντας την δήλωση μόνο των απαραίτητων μεθόδων εντός του visitor, ωστόσο ξεφεύγει ελαφρώς από τις αρχές του προτύπου εφόσον ο χρήστης (με ευθύνη του), κατά την διαπέραση των αντικειμένων της δομής, θα πρέπει να κάνει έλεγχο του τύπου του τρέχοντος αντικειμένου προκειμένου να αποφανθεί αν θα κάνει κλήση της Accept_Visitor_1 αντί της Accept_Visitor.

Τέλος ένα ακόμα ερώτημα που προκύπτει είναι τι γίνεται στην περίπτωση που κάποια λειτουργία εκτελείται σε συγκεκριμένους μόνο τύπους κόμβων. Σε αυτή την περίπτωση προφανώς δεν είναι αναγκαία ή υλοποίηση αντίστοιχων Visit_..... μεθόδων εντός του σχετικού visitor παρά μόνο η δήλωση τους (με απλή αντιγραφή) χωρίς κώδικα.

7.7.3.2 Εφαρμογή Visitor σε μεμονωμένο Syntax element

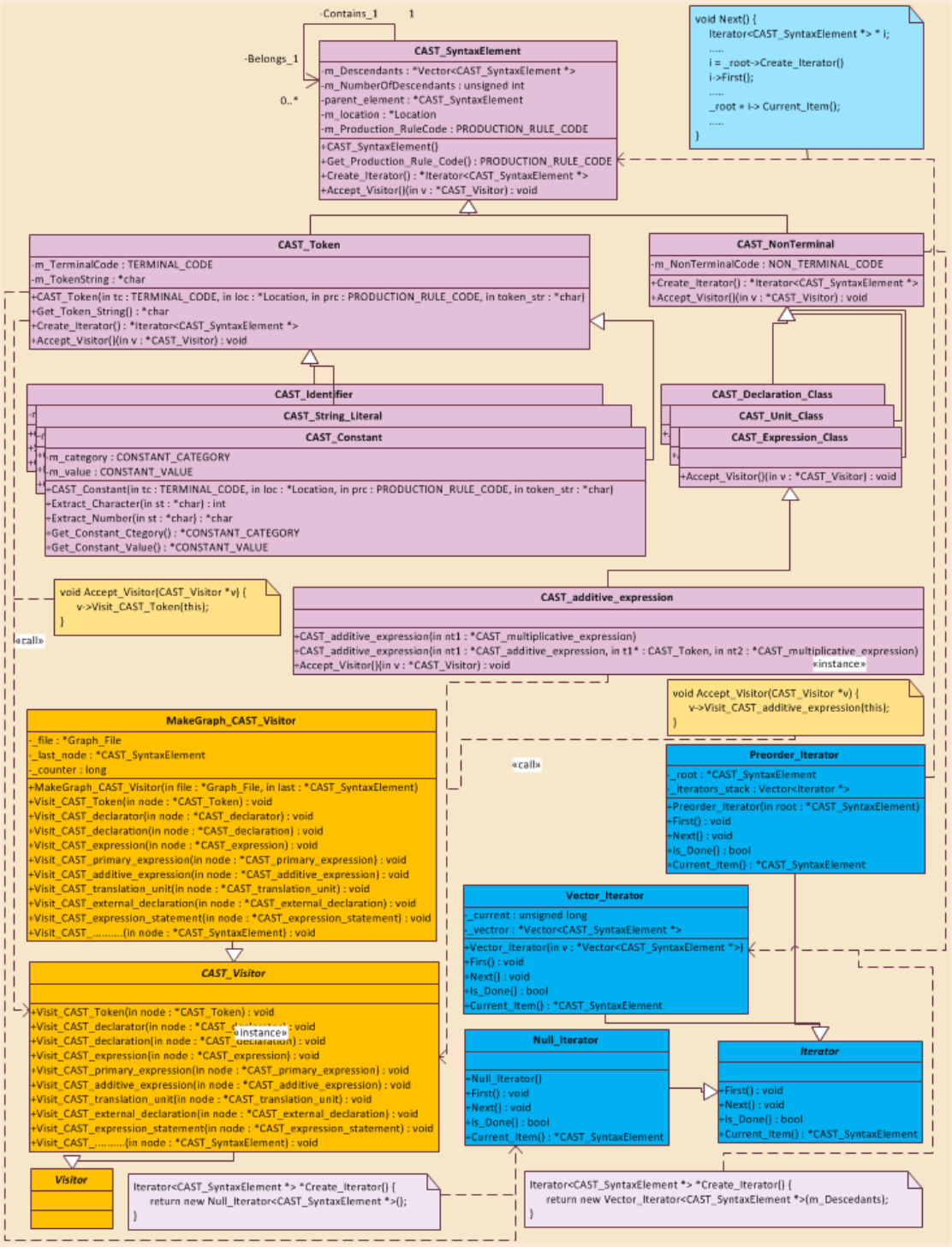
Αν επιθυμούμε την εκτέλεση μιας λειτουργίας (έστω της MakeGraph_CAST_Visitor) σε ένα μεμονωμένο κόμβο (έστω τύπου node: CAST_additive_expression), τότε αρκεί η δημιουργία ενός αντικειμένου mgsv: MakeGraph_CAST_Visitor και η κλήση της μεθόδου **node->Accept_Visitor(operation)** του CAST_additive_expression αντικειμένου με παράμετρο τον συγκεκριμένο visitor.

```
void Simple_CAST_Visitor_Application (CAST_SyntaxElement *node, Graph_File
*gf) {
       MakeGraph_CAST_Visitor *mgastv = new MakeGraph_CAST_Visitor(gf, node);

       node->Accept_Visitor(mgastv);   // παράδειγμα εφαρμογής visitor
}
```

7.7.3.3 Εφαρμογή Graph Visitor σε δομή Parse-Tree (preorder διαπέραση)

Για την παραγωγή του γράφου της δομής του parse tree, απαιτείται η εφαρμογή της λειτουργίας MakeGraph_CAST_Visitor σε όλους τους κόμβους (syntax element) της δομής με μια top-down διαπέραση, δηλαδή μια preorder διαπέραση. Στο σημείο αυτό απαιτείται η συνεργασία όλων των σχεδιαστικών προτύπων Composite, Iterator, Visitor, το συγκεντρωτικό διάγραμμα κλάσεων των οποίων παρουσιάζεται στην Εικόνα 7-19.



Εικόνα 7-19 : Διάγραμμα κλάσεων Parse tree - Iterator - Visitor

Για την εφαρμογή της λειτουργίας MakeGraph_CAST_Visitor σε κάθε κόμβο (syntax element) της δομής, τοποθετούμε τον κώδικα εκτέλεσης της λειτουργίας σε μεμονωμένο

κόμβο, μέσα σε μια δομή επανάληψης preorder διαπέρασης με χρήση του Preorder_Iterator.

```
void Preorder_CAST_Visitor_Application (CAST_SyntaxElement *_root, Graph_File
*gf) {
	MakeGraph_CAST_Visitor *mgastv =new MakeGraph_CAST_Visitor(gf,_root);
	Iterator< CAST_SyntaxElement *> *poi =
			new Preorder_Iterator< CAST_SyntaxElement *>(_root);

	_root->Accept_Visitor(mgastv); // pre εφαρμογή visitor στην ρίζα
	for (poi->First();!poi->Is_Done();poi->Next()) {
		CAST_SyntaxElement *se = poi->Current_Item();

		se->Accept_Visitor(mgastv);  // παράδειγμα εφαρμογής visitor
	}
}
```

Πίσω από την παραπάνω απλή preorder εφαρμογή λειτουργιών με μία μόνο επανάληψη, κρύβεται η αλληλεπίδραση των σχεδιαστικών προτύπων composite, iterator και visitor, όπως αυτή παρουσιάζεται αντίστοιχα στο διάγραμμα αντικειμένων (Εικόνα 7-10) και διάγραμμα ακολουθίας (Εικόνα 7-11) της ενότητας 7.5.3.3 (Εφαρμογή Graph Visitor σε δομή Scopes (preorder διαπέραση)).

7.7.3.4 Εφαρμογή Visitor σε δομή Parse-Tree (postorder διαπέραση)

Όπως έχει αναφερθεί στην ενότητα 5.5.4, ο parser δημιουργεί το parse-tree ως μία bottom-up διαδικασία ή αλλιώς ως μια post order διαπέραση του δέντρου. Μάλιστα σε κάθε κανόνα της γραμματικής (στον parser) μπορούν να αντιστοιχισθούν ενέργειες (κώδικας) που θα εκτελούνται ταυτόχρονα με την bottom-up (post order) δημιουργία του parse tree, τεχνική γνωστή ως Syntax-Directed Translation.

Οι περισσότερες λειτουργίες που εφαρμόζονται στα στοιχεία (κανόνες – τερματικά σύμβολα) ενός parse-tree, σε μεταγενέστερα στάδια της μεταγλώττισης, ακολουθούν μία syntax directed διαπέραση του δέντρου, δηλαδή μια post order διαπέραση. Με την χρήση των σχεδιαστικών προτύπων composite, iterator, και visitor η εφαρμογή μίας οποιασδήποτε λειτουργίας (ορισμένη ως visitor) σε ένα parse tree (composite δομή) μπορεί να υλοποιηθεί μέσω των iterator με τον παρακάτω ενδεικτικό κώδικα.

```
void Postorder_CAST_Visitor_Application (CAST_SyntaxElement *_root) {
	TypeChecking_CAST_Visitor *tcastv =
					new TypeChecking_CAST_Visitor(_root);
	Iterator< CAST_SyntaxElement *> *poi =
			new Postorder_Iterator< CAST_SyntaxElement *>(_root);

	for (poi->First();!poi->Is_Done();poi->Next()) {
		CAST_SyntaxElement *se = poi->Current_Item();

		se->Accept_Visitor(tcastv);  // παράδειγμα εφαρμογής visitor
	}
	_root->Accept_Visitor(tcastv); // post εφαρμογή visitor στην ρίζα
}
```

Ωστόσο οι ενέργειες των visitor επί των στοιχείων (κανόνων της γραμματικής) του parse-tree, πρέπει να περιέχουν μόνο Synthesized attributes, προκειμένου να λειτουργήσει σωστά η bottom-up διαδοχική σύνθεση των ιδιοτήτων.

7.7.4 Bottom-Up κατασκευή Parse / Abstract Tree

Η κατασκευή του parse tree από στοιχεία CAST_SyntaxElement γίνεται από τον parser (και τον scanner) μέσω της bottom-up τεχνικής Syntax-Directed Translation (ενότητα 5.5.4). Συγκεκριμένα σε κάθε κανονική έκφραση του scanner και σε κάθε κανόνα του parser, εισάγουμε κώδικα για την δημιουργία των κατάλληλων CAST_SyntaxElement αντικειμένων καλώντας τους σχετικούς constructor των κλάσεων.

7.7.4.1 Bottom-Up κατασκευή Parse-Tree

Το parse tree (ενότητα 5.6.1.1) περιέχει όλα τα δομικά στοιχειά της derivation ενός προγράμματος εισόδου και συνεπώς δημιουργεί ένα CAST_SyntaxElement στοιχείο για κάθε τερματικό σύμβολο και κανόνα που αναγνωρίζει. Παρακάτω δίνεται ενδεικτικός κώδικας δημιουργίας κόμβων του parse tree με χρήση της composite δομής CAST_SyntaxElement. Για την δημιουργία ενός στοιχείου CAST_Constant από τον scanner, εισάγουμε στην αντίστοιχη κανονική έκφραση (αρχείο *.l) τον παρακάτω ενδεικτικό κώδικα.

```
{DEC_DIGIT}+{INT_SUFFIX}? { /* decimal integer constant */
    (yylval->ast) = new CAST_Constant(T_DECIMAL_INTEGER_CONSTANT,
                                      yylloc, TOKEN_DECIMAL_INTEGER_CONSTANT,
                                      yytext);
    return token::CONSTANT;
}
```

Για την δημιουργία στοιχείων CAST_primary_expression από τον parser με βάση τον κανόνα primary_expression, εισάγουμε στον αντίστοιχο κανόνα (αρχείο *.y) τον παρακάτω ενδεικτικό κώδικα.

```
primary_expression  : IDENTIFIER {……}
| CONSTANT {
   $$ = new CAST_primary_expression((CAST_TOKEN *)$1
                    ,(MemScopt_Parser::location *)&yylloc, UNITIALIZED);
}
| STRING_LITERAL {
   $$= new CAST_primary_expression((CAST_TOKEN *)$1
                    ,(MemScopt_Parser::location *)&yylloc,
                    PT_Primary_expression__STRING_LITERAL_);
}
| '(' expression ')' {
   $$= new CAST_primary_expression((CAST_TOKEN *)$1 ,(CAST_expression *)$2
                    ,(CAST_TOKEN *)$3 ,(MemScopt_Parser::location *)&yylloc
                    ,PT_Primary_expression__LPARENTHESIS_Expression_RPARENTHE
                    SIS_ );
}
```

Παράδειγμα της παραγόμενης γραφικής αναπαράστασης ενός parse-tree δίνεται στο Παράρτημα III : Παραδείγματα (σελίδα 173).

7.7.4.2 Bottom-Up κατασκευή Abstract-Tree

Ένα abstract tree (ενότητα 5.6.1.2) περιέχει μόνο τα απαραίτητα δομικά στοιχειά της derivation ενός προγράμματος εισόδου. Η δομή του μπορεί να διαφέρει από υλοποίηση σε υλοποίηση, ωστόσο μια απλή, αλλά λειτουργική προσέγγιση είναι η απαλοιφή των περιττών κόμβων, όπως οι κόμβοι (κανόνες) που έχουν μόνο ένα παιδί. Για την δημιουργία στοιχείων CAST_primary_expression (περιέχουν μόνο ένα token) από τον parser με βάση τον κανόνα primary_expression, εισάγουμε στον αντίστοιχο κανόνα (αρχείο *.y) τον παρακάτω ενδεικτικό κώδικα θέτοντας την λογική σταθερά parse_tree = false.

```
primary_expression  : IDENTIFIER {……}
| CONSTANT {
    if (parse_tree) $$ = new CAST_primary_expression((CAST_TOKEN *)$1
                    ,(MemScopt_Parser::location *)&yylloc, UNITIALIZED);
    else $$= $1;
}
| STRING_LITERAL {
    if (parse_tree) $$= new CAST_primary_expression((CAST_TOKEN *)$1
                    ,(MemScopt_Parser::location *)&yylloc,
                    PT_Primary_expression__STRING_LITERAL_);
    else $$= $1;
}
| '(' expression ')' {
    if (parse_tree) $$= new CAST_primary_expression((CAST_TOKEN *)$1
                    ,(CAST_expression *)$2 ,(CAST_TOKEN *)$3
                    ,(MemScopt_Parser::location *)&yylloc
                    ,PT_Primary_expression__LPARENTHESIS_Expression_RPARENTHE
                    SIS_ );
    else $$= $2;
}
```

Στον παραπάνω κώδικα παραλείπεται η δημιουργία κόμβου για την εκτέλεση του κανόνα primary_expression ακόμα και στην τελευταία επιλογή ‘(‘ expression ’)’ εφόσον ο κανόνας δεν προσφέρει ουσιαστική λειτουργία στο πρόγραμμα. Παράδειγμα της παραγόμενης γραφικής αναπαράστασης ενός abstract-tree δίνεται στο Παράρτημα III : Παραδείγματα (σελίδα 173).

7.8 Context-Sensitive Analysis σε Parse / Abstract Tree

Γενικά η ανάλυση και επεξεργασία του περιεχομένου (πληροφορίας) σε μία Graphic IR δενδροειδούς μορφής, είναι συνήθως η ίδια είτε εφαρμόζεται σε parse trees είτε σε abstract trees, εφόσον τα δεύτερα είναι συνήθως συντμήσεις των πρώτων, περιέχοντας ωστόσο όλη την πληροφορία του αρχικού προγράμματος.

7.8.1 Attribute Grammar Framework

Στην ανάλυση της ενότητας 5.5.3, αναφέραμε ότι η context-sensitive analysis επιτυγχάνεται μέσω της attribute grammar ή μίας SDD (syntax directed definition), η οποία και συνίσταται από μια CFG προσαυξημένη από ένα σύνολο ενεργειών για τον προσδιορισμό των υπολογισμών επί ενός συνόλου ιδιοτήτων. Οι ιδιότητες (attributes) των τύπων σε ένα μεταγλωττιστή συνήθως οργανώνονται σε δομές οι οποίες συσχετίζονται με το κάθε φύλλο ή κόμβο του parse-tree και συνεργάζονται με μία εξωτερική δομή (λίστα, πίνακα συμβόλων, κλπ). Οι ενέργειες (ή λειτουργίες) που επιδρούν στις ιδιότητες των κόμβων, εφαρμόζονται με συγκεκριμένη σειρά, ανάλογα με το είδος της ενέργειας και με τον τύπο των ιδιοτήτων που αυτή χειρίζεται. Επίσης αναφέραμε ότι αν οι ενέργειες των κανόνων (κόμβων) περιέχουν-χειρίζονται συνδυασμό Synthesized και Inherited attributes μίας L-Attributed Definition, τότε η εκτέλεσή τους μπορεί να πραγματοποιηθεί μέσω μιας ειδικής διαπέρασης του parse-tree η οποία είναι συνδυασμός post-order και pre-order διαπέρασης.

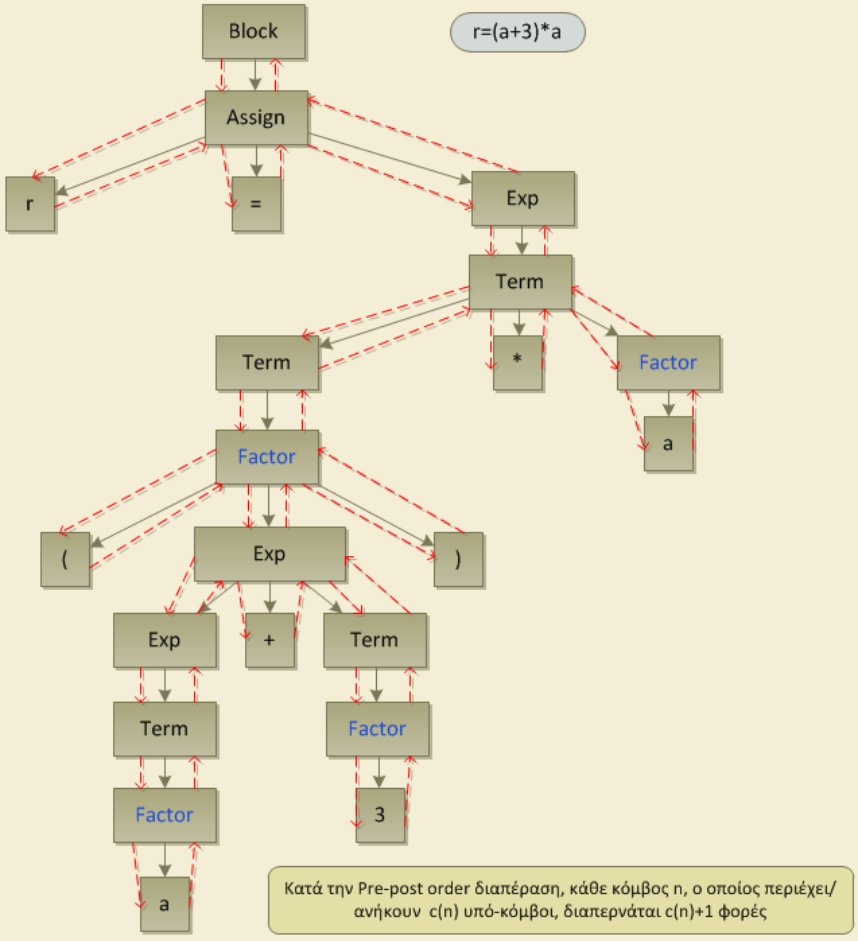
Συγκεκριμένα αναφερόμενοι στην Εικόνα 5-8 (σελίδα 65), παρατηρούμε ότι η διαπέραση επισκέπτεται ένα κόμβο του parse tree, κατόπιν επισκέπτεται το πρώτο παιδί του, όπου επιστρέφοντας από αυτό επισκέπτεται πάλι τον γονικό κόμβο, κατόπιν επισκέπτεται το επόμενο παιδί του κ.ο.κ. μέχρι την επιστροφή από το τελευταίο παιδί του όπου και πάλι επισκέπτεται τον γονικό κόμβο για τελευταία φορά. Σε κάθε επίσκεψη στον κόμβο πιθανό να εκτελεί και διαφορετική ενέργεια επί των ιδιοτήτων. Η ίδια διαδικασία

επαναλαμβάνεται αναδρομικά για κάθε κόμβο παιδί κ.ο.κ. Οι ιδιότητες before και after του παραδείγματος (Εικόνα 5-8 , σελίδα 65) αποτελεί κλασικό και συνηθισμένο παράδειγμα εφαρμογής λειτουργίας επί Synthesized και Inherited ιδιοτήτων μίας L-Attributed Definition.

Στους μεταγλωττιστές είναι συνηθισμένο σχεδόν όλοι οι τύποι context-sensitive analysis να πραγματοποιούνται μέσω μίας SDD, δηλαδή ενός συνδυασμού ιδιοτήτων και ενεργειών-λειτουργιών επ' αυτών, προσδιοριζόμενες ανά κανόνα της γραμματικής. Οι λειτουργίες αυτές, μπορούν να δηλωθούν-οργανωθούν ως visitors και να εφαρμοσθούν σε μια composite δομή (π.χ. parse/abstract tree, graphic IRs, κλπ) μέσω iterators, με βάση τα σχεδιαστικά πρότυπα που έχουν παρουσιασθεί με δύο σημαντικές παρατηρήσεις για την περίπτωση ταυτόχρονου χειρισμού Synthesized και Inherited attributes. Α) οι έως τώρα αναφερόμενοι preorder και postorder iterators δεν καλύπτουν την περίπτωση όπου ο γονικός κόμβος αποτελεί μέρος της διαπέρασης πριν και μετά την επίσκεψη σε κάθε παιδί του, Β) η δομή των visitors, όπως έχει παρουσιασθεί, δεν είναι δυνατό να διαχωρίσει κατά την επίσκεψη σε έναν κόμβο αν και πόσα παιδία του έχουν ήδη επισκεφθεί προηγουμένως, προκειμένου να καθορίσει την ενέργεια που θα εκτελέσει επί των αντίστοιχων ιδιοτήτων του κόμβου. Για την κάλυψη της ειδικής αυτής περίπτωσης παρακάτω παρουσιάζεται ένας ειδικού τύπου iterator με όνομα Prepostorder_Iterator καθώς και μία παραλλαγή της εφαρμογής του σχεδιαστικού προτύπου visitor.

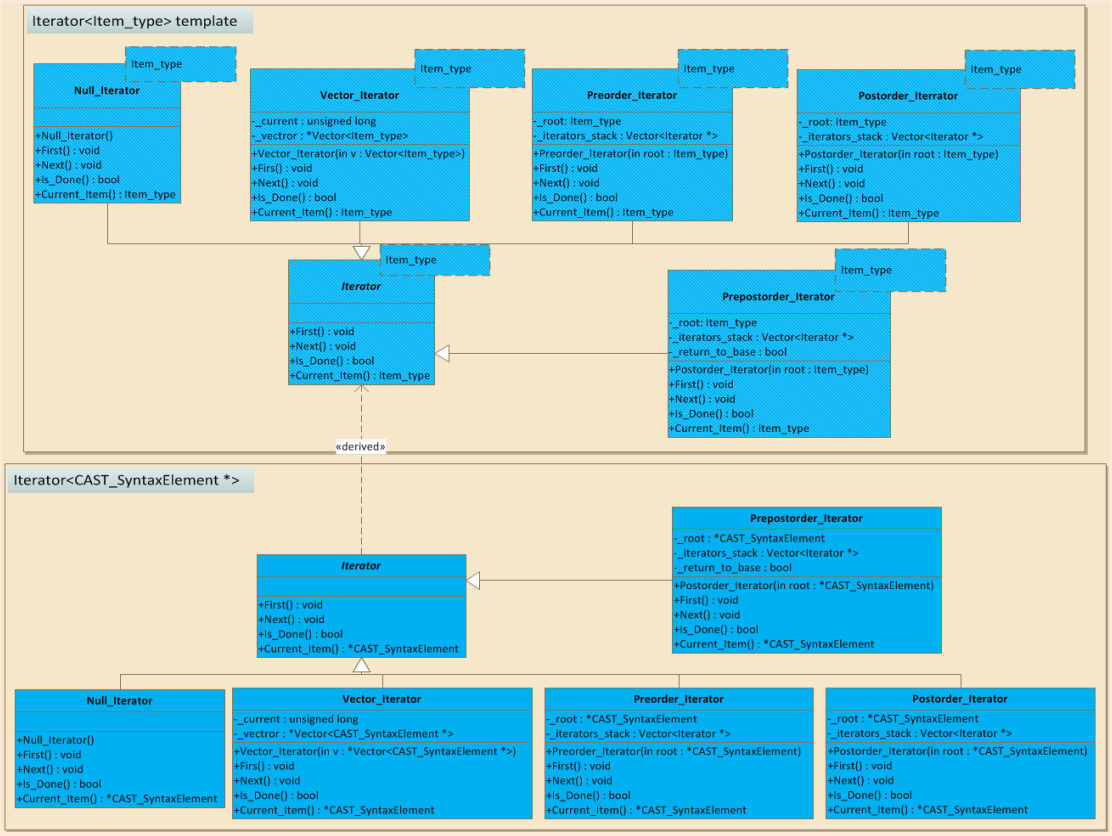
7.8.1.1 Σχεδίαση PrePostorder Iterator

Κατά την pre-post order διαπέραση (Εικόνα 7-20) το βασικό χαρακτηριστικό είναι ότι κάθε κόμβος διαπερνάται τόσες φορές όσες ο αριθμός των παιδιών του +1. Για παράδειγμα ένας κόμβος με 2 υπο-κόμβους διαπερνάται 2+1=3 φορές και ένας κόμβος φύλλο διαπερνάται 0+1=1 φορά.



Εικόνα 7-20 : Ενδεικτικό διάγραμμα pre-post order διαπέρασης κόμβων

Για την ολοκλήρωση της σχεδίασης του μοντέλου των iterators χρησιμοποιούμε το ίδιο template των Null_Iterator, Vector_Iterator και Preorder_Iterator της ενότητας 7.5.2.1. καθώς και τον Postorder_Iterator της ενότητας 7.7.2.1 με κατάλληλη παράμετρο Item_type. Η σχεδίαση του (ανοικτού) προτύπου έχει επεκταθεί με την προσθήκη μίας κλάσης για την μικτή pre-post order (ενότητα 5.5.3) διαπεράση με όνομα Prepostorder_Iterator. Η πλήρης σχεδίαση των Iterator δομών τόσο του template όσο και του στιγμιότυπου του για Item_type = CAST_SyntaxElement* παρουσιάζεται στην Εικόνα 7-21.



Εικόνα 7-21 : Πλήρες διάγραμμα κλάσεων Iterator template (CAST_SyntaxElement)

Και πάλι είναι εμφανής η απλότητα χρησιμοποίησης των όλων των iterators από τον χρήστη με κοινό interface για διαφορετικού τύπου (Item_type) στοιχεία.

7.8.1.2 Υλοποίηση PrePostorder Iterator

Επειδή η υλοποίηση του Iterator που παρουσιάσθηκε στην σχεδίαση έχει γενικά ενιαία δομή και ευρεία χρήση, παρακάτω δίνεται ο ενδεικτικός κώδικας της υλοποίησης της νέας κλάσης Prepostorder_Iterator (σε συνέχεια των υλοποιήσεων των ενοτήτων 7.5.2.2 και 7.7.2.2).

```
template <class ItemType>
class Prepostorder_Iterator: public Iterator<ItemType> {
private:
       ItemType _root;
       vector<Iterator*> *_iterators_stack;
       Iterator* _i;
       bool _return_to_base;
public:
Prepostorder_Iterator(ItemType root) {
       _root=root;
       _iterators_stack =
            new vector<Iterator*> (PREPOSTORDER_ITERATOR_STACK_SIZE);
       _return_to_base = false;
};
void First() {
       _i = _root->Create_Iterator();
       if (_i) {
             _i->First();
             _iterators_stack->clear();
             _iterators_stack->push_back(_i);
       }
};
void Next(){
       if (_return_to_base==true) {
```

```
                _return_to_base=false ;
                if (_iterators_stack->back()->Is_Done()) {
                        delete _iterators_stack->back();
                        _iterators_stack->pop_back();
                         if (!_iterators_stack->empty())
                                _iterators_stack->back()->Next();
                        _return_to_base=true;
                }
        }
        else {
            _i = _iterators_stack->back()->Current_Item()->Create_Iterator();
                _i->First();
                _iterators_stack->push_back(_i);
                if (   (_iterators_stack->size()>0) &&
                        (_iterators_stack->back()->Is_Done()) ) {
                        delete _iterators_stack->back();
                        _iterators_stack->pop_back();
                        if (!_iterators_stack->empty())
                                _iterators_stack->back()->Next();
                        _return_to_base = true;
                }
        }
};
bool Is_Done() { return (_iterators_stack->empty()); };
ItemType Current_Item(){
        int i_size = _iterators_stack->size();
        if ( i_size > 0 )
                if (!_return_to_base)
                        return _iterators_stack->back()->Current_Item();
                else if ( i_size > 1 )
                    return (_iterators_stack->at(i_size -2))->Current_Item();
                else return _root;
        else return 0;
        };
};
```

Η ανωτέρω υλοποίηση επισκέπτεται και την ρίζα (παράμετρος constructor) της δομής εκτός από την αρχική επίσκεψη (έναρξη διαπέρασης από το πρώτο παιδί της ρίζας), δηλαδή τόσες φορές όσες ο αριθμός των υπο-κόμβων της.

7.8.1.3 Εφαρμογή PrePostorder Iterator

Για την pre-post order διαπέραση μίας δενδροειδής δομής από CAST_SyntaxElement αντικείμενα με ρίζα το αντικείμενο _root, αρκεί μια ενιαία δομή επανάληψης:

```
void Prepostorder_SyntaxElement_Iteration_Application
                                                (CAST_SyntaxElement *_root) {
        Iterator<CAST_SyntaxElement *> *ppoi =
                        new Prepostorder_Iterator< CAST_SyntaxElement *>(_root);

        _root->Get_Production_Rule_Code; // pre πρόσβαση στην ρίζα
        for (ppoi->First();!ppoi->Is_Done();ppoi->Next()) {
                CAST_SyntaxElement *se = ppoi->Current_Item();
                . . . . .
                int prc = se->Get_Production_Rule_Code();//παράδειγμα πρόσβασης
                . . . . .
        }
}
```

7.8.1.4 Σχεδίαση Visitor για pre-post order διαπέραση

Ο Prepostorder_Iterator μπορεί να χρησιμοποιηθεί για την επίσκεψη των κόμβων μιας δομής μέσω ενός visitor, ωστόσο, κατά την εφαρμογή του attribute grammar framework με Inherited attributes, ο visitor ενδέχεται να επιτελεί διαφορετική λειτουργία σε ένα είδος κόμβου σε κάθε επίσκεψη, ανάλογα με το πόσες φορές έχει επισκεφθεί τον εκάστοτε

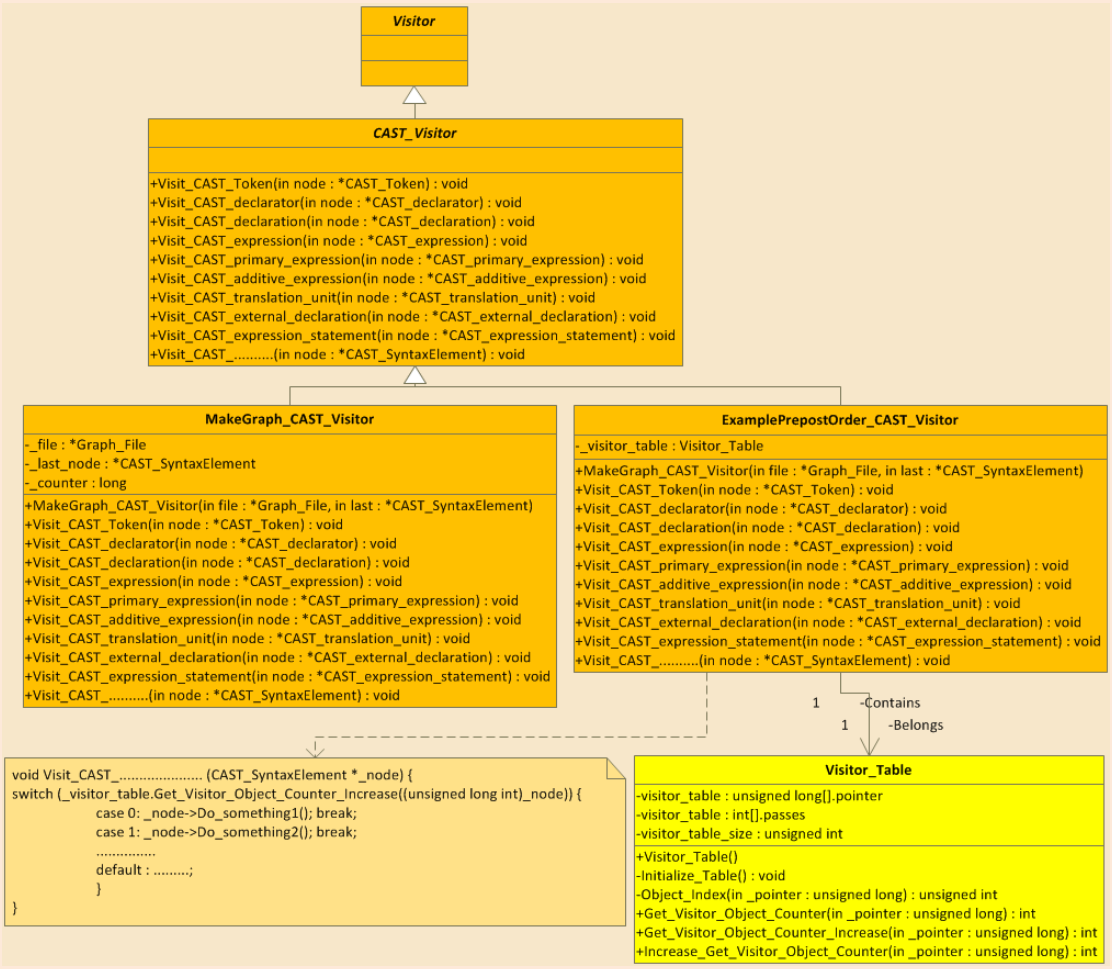
κόμβο ή αλλιώς ανάλογα με τον αριθμό των υπο-κόμβων του που έχει επισκεφθεί. Συγκεκριμένα κάθε κόμβος στην pre-post order διαπέραση, διαπερνάται μία φορά αρχικά (pre) και μία φορά (mid/post) μετά την ολοκλήρωση της διαπέρασης του κάθε υπο-κόμβου του. Προκύπτει λοιπόν η ανάγκη αποτύπωσης / αποθήκευσης του αριθμού των επισκέψεων σε ένα κόμβο για μία pre-post order διαπέραση.

Μία προσέγγιση είναι η αποθήκευση του αριθμού των επισκέψεων στον κάθε κόμβο (μέσω μετρητή μιας κεντρικής κλάσης) της δομής. Όμως κάτι τέτοιο απαιτεί την αρχικοποίηση ή/και μηδενισμό μετρητών σε κάθε αντικείμενο (κόμβο) της δομής μετά την ολοκλήρωση μιας διαπέρασης. Επίσης ταυτόχρονες διαπεράσεις από διαφορετικούς visitors θα απαιτούσαν διαφορετικούς μετρητές σε κάθε κόμβο. Το κυριότερο μια τέτοια προσέγγιση ξεφεύγει από τις αρχές των σχεδιαστικών προτύπων, εφόσον «μολύνει» τις κλάσεις της composite δομής με επιπλέων πληροφορίες και λειτουργίες σχετικά με την διαπέρση και επίσκεψη των στοιχείων της μέσω των visitors.

Μια διαφορετική προσέγγιση είναι η αποθήκευση του αριθμού των επισκέψεων στο κάθε αντικείμενο του iterator που δημιουργείται κατά την διαπέραση του εκάστοτε κόμβου. Όμως κάτι τέτοιο απαιτεί την επανασχεδίαση και προσαρμογή όλων των υπαρχόντων προτύπων iterator (π.χ. Vector_Iterator), ώστε να τηρούν και ενημερώνουν την εν λόγω πληροφορία. Επίσης η λειτουργία των Iterators, είναι ανεξάρτητη από την λειτουργία των visitors, μη επικοινωνώντας μεταξύ τους. Ο χρήστης (πρόγραμμα διαπέρασης) χρησιμοποιεί έναν iterator για να λάβει έναν δείκτη στο επόμενο αντικείμενο της δομής, στο οποίο και στη συνέχεια εφαρμόζει εάν συγκεκριμένο visitor (κλήση της Accept_Visitor). Η επανασχεδίαση iterators και visitors ώστε να επικοινωνούν μεταξύ τους περιπλέκει τόσο την υλοποίηση τους όσο και τον τρόπο χρήσης τους και ξεφεύγει από τις αρχές των αντίστοιχων σχεδιαστικών προτύπων.

Προκειμένου να διατηρήσουμε την ανεξαρτησία μεταξύ των σχεδιαστικών προτύπων composite, iterator και visitor, προτείνεται η αποθήκευση του αριθμού των επισκέψεων στο αντικείμενο του εκάστοτε Visitor. Προκύπτει το θέμα του τρόπου αποθήκευσης της σχετικής πληροφορίας, δηλαδή του αριθμού των επισκέψεων σε κάθε κόμβο της δομής. Η χρήση ενός vector ή μιας λίστας ή ενός απλού πίνακα στοιχείων για το σκοπό αυτό, εισάγει το πρόβλημα της εισαγωγής και αναζήτησης, το όποιο στην καλύτερη περίπτωση (δυαδική αναζήτηση) έχει μέσο χρόνο εκτέλεσης της τάξης $O(\log_2 n)$. Ωστόσο διαπιστώνεται ότι σε έναν τέτοιο πίνακα δεν υπάρχουν διαγραφές ούτε μετακινήσεις των στοιχείων του, εφόσον το αντικείμενο συνήθως καταστρέφεται μετά την χρήση του. Άρα ένας τέτοιος πίνακας μπορεί να υλοποιηθεί ως symbol table, εισάγοντας την ταυτότητα κάθε κόμβου που επισκέπτεται ο visitor, μέσω μιας hash function κατά τα πρότυπα της τεχνικής hashing with linear probing (ενότητα, 5.7.1), η οποία και εξασφαλίζει (στη γενική περίπτωση) χρόνο εισαγωγής και αναζήτησης στοιχείου ίσο με O(1).

Η σχεδίαση ενός visitor για επίσκεψη σε κόμβους μέσω μίας pre-post order διαπέρασης, για μία composite δομή από CAST_SyntaxElement στοιχεία, παρουσιάζεται στην Εικόνα 7-22.



Εικόνα 7-22 : Διάγραμμα κλάσεων Visitor για pre-post order διαπέραση

Αναφερόμενοι στην κλάση του ExamplePrepostOrder_CAST_Visitor, παρατηρούμε την ιδιότητα (πεδίο) τύπου Visitor_Table, η οποία και υλοποιεί ένα πίνακα κατά τα πρότυπα ενός symbol table. Ως ταυτότητα (κλειδί αναζήτησης) του εκάστοτε κόμβου επιλέγεται η ίδια η τιμή της αναφοράς (δείκτης) στον κόμβο, που εισέρχεται ως παράμετρος κατά την κλήση των μεθόδων του visitor. Έτσι δεν απαιτείται τροποποίηση του κοινού interface και των υπολοίπων visitor της συγκεκριμένης δομής. Ως hash function χρησιμοποιούμε μια απλή διαίρεση της τιμής του pointer (αναφορά κόμβου) με το μέγεθος του πίνακα, αναμένοντας να αποδώσει μία αρκετά δίκαιη κατανομή, δεδομένου ότι οι pointers διαφορετικών αντικειμένων (ως σχετικές διευθύνσεις) έχουν εξορισμού διακριτές τιμές με μία σχετική γραμμική συνέχεια. Η διαχείριση συγκρούσεων επιλύεται κατά τα πρότυπα της τεχνικής hashing with linear probing. Ο πίνακας αποθηκεύει για κάθε κόμβο την αναφορά του (δείκτη) ως κλειδί αναφοράς καθώς και τον αριθμό των επισκέψεων (passes) σε αυτόν.

Η Visitor_Table περιέχει μεθόδους για την ανάκτηση του αριθμού των περαμάτων (passes) ενός κόμβου καθώς και μεθόδους για την ταυτόχρονη ανάκτηση και την (πριν ή μετά) ενημέρωση τους. Για παράδειγμα η μέθοδος Get_Visitor_Object_Counter_Increase (_pointer) ελέγχει αν υπάρχει στον πίνακα στοιχείο με κλειδί _pointer του οποίου και αυξάνει την τιμή των passes κατά ένα και επιστρέφει την προηγούμενη τιμή (μετά ενημέρωση). Αν δεν βρεθεί τέτοιο στοιχείο εισάγει ένα με κατάλληλες αρχικές τιμές.

Τέλος κάθε μέθοδος του visitor μπορεί με μία απλή κλήση π.χ. της Get_Visitor_Object_Counter_Increase (_node), σε συνδυασμό με μία δομή ελέγχου π.χ. switch, να καθορίσει την εκτέλεση διαφορετικών ενεργειών ανάλογα με το τρέχον πέρασμα – επίσκεψη σε ένα συγκεκριμένο κόμβο (αντικείμενο) της δομής.

7.8.1.5 Υλοποίηση Visitor για pre-post order διαπέραση

Η ενδεικτική υλοποίηση της βοηθητικής κλάσης Visitor_Table, σύμφωνα με τη σχεδίαση που παρουσιάσθηκε, δίνεται παρακάτω:

```
class Visitor_Table {
public :
Visitor_Table() {
	visitor_table_size = VISITOR_TABLE_SIZE;
	Initialize_Table();
};
~Visitor_Table() {};
int Get_Visitor_Object_Counter(unsigned long long int _pointer) {
	int _iterator = Object_Index(_pointer);
	int scount = visitor_table_size; // how many entrys have we looked at
	while(--scount >= 0) {
		if ( visitor_table[_iterator].pointer == _pointer )
			{ return visitor_table[_iterator].passes; } // find it
		// empty table entry, not found, abort shearching
		if ( visitor_table[_iterator].pointer == 0 ) { return 0; }
		// try the next entry (resets index table at the end)
		if (++_iterator >= visitor_table_size ) _iterator = 0;
	}
	return -2; // table is full, not found
};
int Get_Visitor_Object_Counter_Increase(unsigned long long int _pointer) {
	int _iterator = Object_Index(_pointer);
	int scount = visitor_table_size; // how many entrys have we looked at
	while(--scount >= 0) {
		if ( visitor_table[_iterator].pointer == _pointer )
			{ return visitor_table[_iterator].passes++; } // find it
		if ( visitor_table[_iterator].pointer == 0 ) {
			// empty table entry, new entry
			visitor_table[_iterator].pointer = _pointer;
			visitor_table[_iterator].passes = 1;
			return 0;
		}
		// try the next entry (resets index table at the end)
		if (++_iterator >= visitor_table_size ) _iterator = 0;
	}
	return -2; // table is full
};
int Increase_Get_Visitor_Object_Counter (unsigned long long int _pointer) {
	int _iterator = Object_Index(_pointer);
	int scount = visitor_table_size; // how many entrys have we looked at
	while(--scount >= 0) {
		if ( visitor_table[_iterator].pointer == _pointer )
			{ return ++(visitor_table[_iterator].passes);} //find it
		if ( visitor_table[_iterator].pointer == 0 ) {
			// empty table entry, new entry
			visitor_table[_iterator].pointer = _pointer;
			visitor_table[_iterator].passes = 1;
			return 1;
		}
		// try the next entry (resets index table at the end
		if (++_iterator >= visitor_table_size ) _iterator = 0;
	}
	return -2; // table is full
};
long int Get_Visitor_Table_Size() { return visitor_table_size; };
```

```
private :
void Initialize_Table() {
       for ( unsigned int i=0 ; i<visitor_table_size ;
              visitor_table[i].pointer=0, visitor_table[i++].passes=0 );
};
unsigned int Object_Index (unsigned long long int _pointer) {
       return _pointer % visitor_table_size;
};

struct visitor_table_type {
       unsigned long long int pointer;
       int passes;
} visitor_table[VISITOR_TABLE_SIZE];
unsigned int visitor_table_size;
};
```

Με βάση τα παραπάνω η ενδεικτική υλοποίηση της μεθόδου Visit_CAST_additive_expression για τον Example_PrepostOrder_CAST_Visitor έχει ως ακολούθως :

```
void Example_PrepostOrder_CAST_Visitor::
       Visit_CAST_additive_expression (CAST_additive_expression *_node) {

       switch ( _visitor_table.Get_Visitor_Object_Counter_Increase
                                          ((unsigned long long int)_node) ) {
              case 0: _node->m_Descendants[0]->before = _node->before; break;
              case 1: _node->m_Descendants[1]->before =
                                   _node->m_Descendants[0]->after; break;
              case 2: _node->m_Descendants[2]->before =
                                   _node->m_Descendants[1]->after; break;
              case 3: node->after = _node->m_Descendants[2]->after ; break;
              default:;
       }
}
```

Στο ανωτέρω παράδειγμα θεωρούμε ότι το στοιχείο της δομής με τύπο (κλάση) CAST_additive_expression, έχει 3 υποκόμβους και συνεπώς η pre-post order διαπέραση θα επισκεφθεί το κάθε αντικείμενο του κόμβου 3+1=4 φορές. Σε κάθε επίσκεψη οι (κοινές) ιδιότητες after και before μεταφέρονται από κόμβο σε κόμβο της δομής (top-down και bottom-up) κατά pre-post order διάταξη. Συγκεκριμένα οι περιπτώσεις 0,1 και 2 ενημερώνουν μια Inherited attribute και η περίπτωση 3 ενημερώνει μία Synthesized attribute ενός υποτιθέμενου – γενικού attribute grammar framework.

Παρατηρούμε ότι παρά την περιπλοκότητα μιας pre-post order διαπέρασης σε μία δενδροειδή (composite) δομή αντικειμένων, η υλοποίηση του visitor και των μεθόδων του, έχουν αρκετά απλή και ξεκάθαρη δομή μη επηρεάζοντας τα χαρακτηριστικά και τη λειτουργία των σχεδιαστικών προτύπων. Ειδικά η τελευταία υλοποίηση της μεθόδου επίσκεψης κόμβου, χρησιμοποιεί μία δομή απόφασης (switch) με κυκλωματική πολυπλοκότητα 5. Γενικά η κυκλωματική πολυπλοκότητα μιας μεθόδου (ενός visitor) επίσκεψης κόμβου κατά pre-post order (με βάση την προτεινόμενη σχεδίαση), είναι ανάλογη (σχεδόν ίση) με το μέγιστο πλήθος των υπο-κόμβων του κόμβου ή με το πλήθος των διαφορετικών ενεργειών που επιτελεί. Ωστόσο οι κόμβοι στους οποίους εφαρμόζεται ένα attribute grammar framework σε ένα abstract-tree έχουν, συνήθως μικρό αριθμό υπο-κόμβων περιορίζοντας την πολυπλοκότητα των μεθόδων (visitor) που επιδρούν σε αυτούς.

Τέλος μια προσεκτική σχεδίαση των ιδιοτήτων του attribute grammar framework μιας δομής (π.χ. parse/abstract trees), μπορεί να επιτρέψει την υλοποίηση μεθόδων (visitor) με

ακόμη μικρότερη πολυπλοκότητα, όπως παρουσιάζεται στην παρακάτω βελτιωμένη ενδεικτική υλοποίηση της μεθόδου Visit_CAST_additive_expression.

```
void Example_PrepostOrder_CAST_Visitor::
       Visit_CAST_additive_expression (CAST_additive_expression *_node) {
       int passes = _visitor_table.Get_Visitor_Object_Counter_Increase
                                        ((unsigned long long int)_node)
       switch (passes) {
              case 0: _node->m_Descendants[0]->before = _node->before; break;
              case 1,2: _node->m_Descendants[passes]->before =
                           _node->m_Descendants[passes-1]->after; break;
              case 3: node->after = _node->m_Descendants[2]->after ; break;
              default:;
       }
}
```

7.8.1.6 *Εφαρμογή Visitor για pre-post order διαπέραση*

Για την εφαρμογή της λειτουργίας Example_PrepostOrder_CAST_Visitor σε κάθε κόμβο (syntax element) της δομής, χρησιμοποιούμε μια δομή επανάληψης pre-post order διαπέρασης με χρήση του Prepostorder_Iterator. Η πρώτη (pre, αρχική) επίσκεψη της ρίζας πραγματοποιείται ξεχωριστά πριν την έναρξη της επανάληψης.

```
void Prepostorder_CAST_Visitor_Application (CAST_SyntaxElement *_root) {
       Example_PrepostOrder_CAST_Visitor *eppoastv =
                     new Example_PrepostOrder_CAST_Visitor (_root);
       Iterator< CAST_SyntaxElement *> *ppoi =
                     new Prepostorder_Iterator< CAST_SyntaxElement *>(_root);

       _root->Accept_Visitor(eppoastv); // pre εφαρμογή visitor στην ρίζα
       for (ppoi->First();!ppoi->Is_Done();ppoi->Next()) {
              CAST_SyntaxElement *se = ppoi->Current_Item();

              se->Accept_Visitor(eppoastv);  // παράδειγμα εφαρμογής visitor
       }
    }
```

Παρατηρούμε ότι παρά την περιπλοκότητα μιας pre-post order διαπέρασης σε μία δενδροειδή (composite) δομή αντικειμένων, η εφαρμογή του visitor στη δομή έχει πολύ απλή και ξεκάθαρη υλοποίηση μη επηρεάζοντας τη δομή και λειτουργία των σχεδιαστικών προτύπων. Ειδικά η ανωτέρω υλοποίηση της εφαρμογής χρησιμοποιεί μόλις μία δομή επανάληψης με κυκλωματική πολυπλοκότητα 2. Ωστόσο η συγκεκριμένη τεχνική προκειμένου να αφήσει τα Interface των composite, iterator και visitor ανεπηρέαστα, μεταβάλλοντας ουσιαστικά μόνο τις υλοποιήσεις του σχετικού visitor, πληρώνει ένα τίμημα σε χώρο (μνήμη) για τον πίνακα αναζήτησης (Visitor_Table) του εκάστοτε visitor αντικειμένου. Αν δεν είμαστε διατεθειμένοι ή δεν έχουμε τους πόρους για να αποδεχθούμε το εν λόγω τίμημα, μπορούμε να προσφύγουμε σε διαφορετικές λύσεις (π.χ. απευθείας κλήσεις μέσω visitor), μεταβάλλοντας ωστόσο τόσο τα interface, όσο και τις υλοποιήσεις των composite και visitor προτύπων που έχουν παρουσιασθεί ως τώρα.

7.8.2 Παράδειγμα προσδιορισμού & ελέγχου τύπου εκφράσεων (Expression type inferring & checking)

Προκειμένου να αναδειχθεί ο τρόπος εφαρμογής και τα οφέλη των σχεδιαστικών προτύπων composite, iterator και visitor, παρουσιάζεται η εφαρμογή της λειτουργίας προσδιορισμού και ελέγχου τύπων εκφράσεων (**inferring expression types**), η οποία

συνήθως διενεργείται ως ξεχωριστό πέρασμα (διαπέραση) επί του parse ή abstract tree του προγράμματος.

Κάθε μεταγλωττιστής που προσπαθεί να παράγει αποδοτικό κώδικα για μία τυποποιημένη γλώσσα (όπως η C89 στην περίπτωσή μας) πρέπει να αντιμετωπίζει το πρόβλημα του προσδιορισμού και του ελέγχου των τύπων για κάθε έκφραση (expression) του προγράμματος προς μεταγλώττιση. Αυτό το πρόβλημα (όπως και πολλά άλλα) στηρίζεται εγγενώς στις σχετικές με το περιεχόμενο πληροφορίες (context-sensitive information). Ο τύπος που συσχετίζεται σε έναν σύμβολο (identifier) ή μία σταθερά (constant) εξαρτάται από την ταυτότητα (όνομα ή τιμή) του στοιχείου παρά από την συντακτική του κατηγορία (syntactic category). Έτσι ο τύπος (type) για κάθε identifier ή constant είναι μια (σύνθετη συνήθως) ιδιότητα στο πλαίσιο του attribute grammar framework τη γλώσσας, η οποία πρέπει να προσδιορισθεί και για κάθε τύπο έκφρασης της γλώσσας (πχ assignment_expression, additive_expression, κ.λ.π.).

7.8.2.1 Σχεδίαση Visitor για type inferring & checking

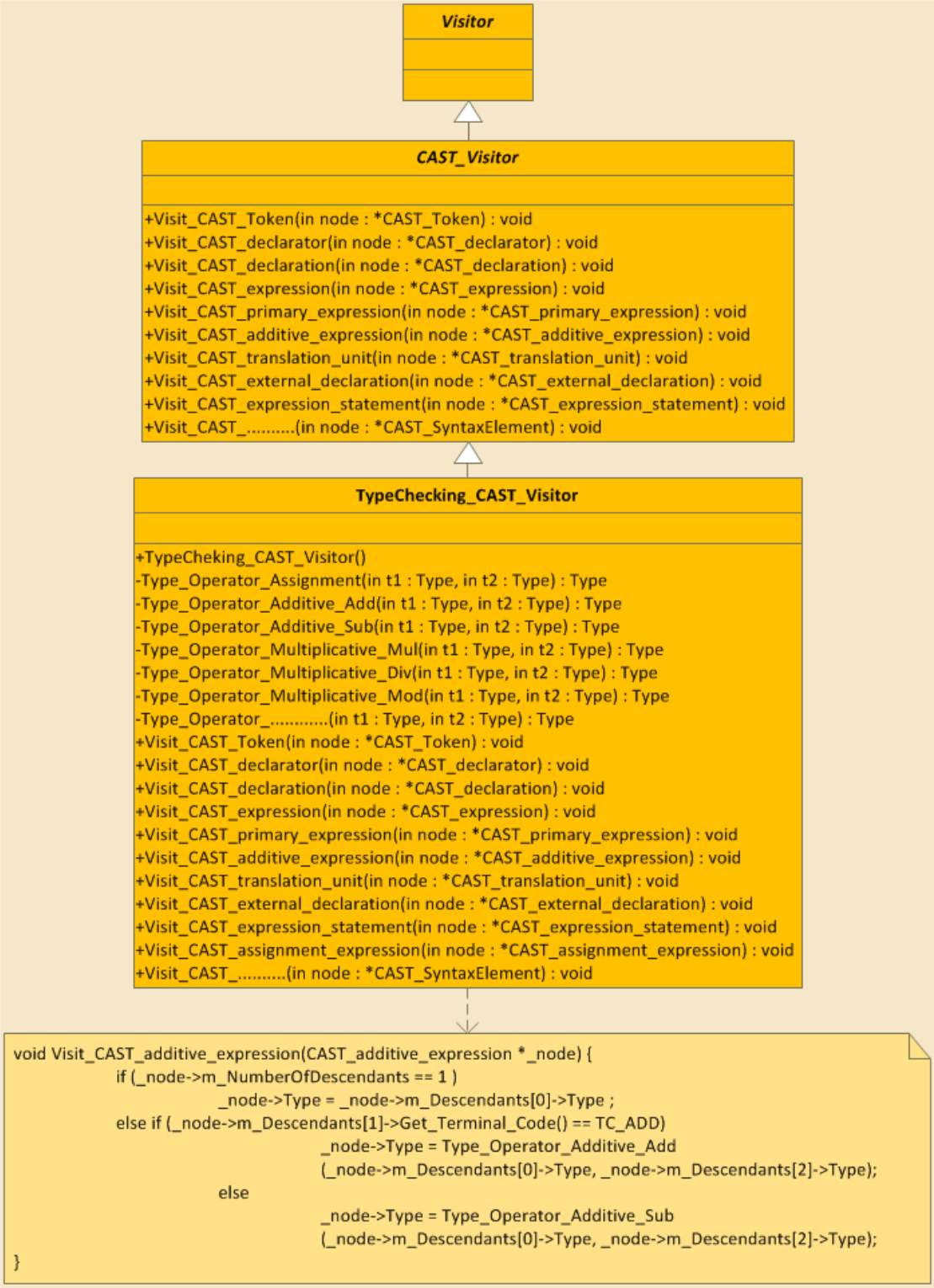
Θεωρούμε ότι σε κάθε identifier και constant που αναφέρονται στις εκφράσεις του προγράμματος έχει αποδοθεί η σχετική ιδιότητα τύπου (type attribute), μέσω της εφαρμογή κάποιας άλλης λειτουργίας σε προγενέστερο πέρασμα, η οποία αναλύει τόσο τα constant token όσο και τις δηλώσεις (declarations) των identifier tokens προκειμένου να εξάγει τους τύπους τους. Επισημάνεται ότι ο τύπος (type attribute) ενός identifier μέσω μιας δήλωσης (declaration) στις γλώσσες προγραμματισμού (όπως η C89) αποτελείται συνήθως από μία σύνθετη δομή (πχ. structure, union, sub-tree) άλλων επιμέρους ιδιοτήτων. Επειδή ο τύπος κάθε identifier εξαρτάται από την περιοχή (scope) του προγράμματος όπου και αναφέρεται, κάθε αναφορά του identifier συνήθως συσχετίζεται με μία αντίστοιχη αναφορά στο symbol table του μεταγλωττιστή. Οι εκφράσεις της γλώσσας (C89) αναπαρίστανται σε ένα parse (ή abstract) tree με συντακτικά στοιχεία / κανόνες τύπου CAST_SyntaxElement (Εικόνα 7-12 και Εικόνα 7-13), όπου κάθε element έχει μία σύνθετη ιδιότητα με όνομα «Type», που αναπαριστά τον τύπο του εκάστοτε στοιχείου (κανόνα) της γραμματικής.

Για κάθε αριθμητικό τελεστή της γραμματικής αναπτύσσεται μια σχετική συνάρτηση χαρτογράφησης του τύπου των δύο τελεστών στον παραγόμενο τύπο, με βάση το πρότυπο προδιαγραφών της γλώσσας. Για παράδειγμα για τον τελεστή της πρόσθεσης ‘+’ υλοποιείται η συνάρτηση Type_Operator_Additive_Add(type1, type2) όπου ενδεικτικά για βασικούς τύπους,

- Type_Operator_Additive_Add (int, int) = int
- Type_Operator_Additive_Add (int, real) = real
- Type_Operator_Additive_Add (int, long int) = long int
- Type_Operator_Additive_Add (int, char) = int
- Type_Operator_Additive_Add (real, *char) = Error

Οι ανωτέρω συναρτήσεις (χαρτογράφησης αριθμητικών τελεστών) μπορούν να ενσωματωθούν σε ένα visitor, έστω τον TypeChecking_CAST_Visitor (Εικόνα 7-16), όπου μαζί με τις μεθόδους επίσκεψης ανά τύπο κόμβου παρουσιάζονται στην ενδεικτική σχεδίαση στην Εικόνα 7-23. Φυσικά και μπορεί να επιλεχθεί η διάσπαση – κατανομή των

μεθόδων (Type_Operator_....) σε περισσότερες κλάσεις, ωστόσο στην σχεδίαση της υλοποίησης επιλέγουμε την συγκέντρωσή τους σε μία ενιαία.



Εικόνα 7-23 : Διάγραμμα κλάσεων Type Checking Visitor

Στο ανωτέρω διάγραμμα δίνεται και η ενδεικτική υλοποίηση της μεθόδου Visit_CAS_additive_expression για την επίσκεψη κόμβων τύπου CAST_additive_expression που αντιστοιχούν σε εφαρμογές του κανόνα :

```
    additive_expression :
            multiplicative_expression
            | additive_expression '+' multiplicative_expression
            | additive_expression '-' multiplicative_expression
```

Η μέθοδος ελέγχει για την εκδοχή του κανόνα (πλήθος υπο-κόμβων) καθώς και τον τύπο του τελεστή (ιδιότητα terminal code) του δεύτερου υπό-κόμβου, προκειμένου να εκτελέσει την κατάλληλη συνάρτηση χαρτογράφησης τύπων, με παραμέτρους τους τύπους του πρώτου και τρίτου υπο-κόμβου. Γενικά οι μέθοδοι (ανά κανόνα της γραμματική) του TypeCheching_CAST_Visitor, μαζί με την ιδιότητα Type, ορίζουν μία SDD (syntax directed definition) για την υλοποίηση της λειτουργίας inferring expression types στα πλαίσια της context sensitive analysis που πραγματοποιεί ο μεταγλωττιστής.

7.8.2.2 Εφαρμογή Type inferring & checking Visitor σε δομή Parse-Tree

Παρατηρούμε ότι όλες οι λειτουργίες (πχ Visit_CAS_additive_expression) του TypeChecking_CAST_Visitor επιδρούν και ενημερώνουν την ιδιότητα Type των κόμβων, η οποία από τον τρόπο ενημέρωσης bottom-up (sub-node to node) προκύπτει ότι είναι μία synthesized attribute. Συνεπώς πρόκειται για μία S-Attributed Definition και η επίσκεψη των κόμβων μπορεί να γίνει με μία bottom-up (post order) διαπέραση με χρήση του Postorter_Iterator (ενότητα 7.7.3.4).

```
void Postorder_CAST_Visitor_Application (CAST_SyntaxElement *_root) {
      TypeChecking_CAST_Visitor *tcastv =
                                    new TypeChecking_CAST_Visitor();
      Iterator< CAST_SyntaxElement *> *poi =
                  new Postorder_Iterator< CAST_SyntaxElement *>(_root);

      for (poi->First();!poi->Is_Done();poi->Next()) {
            CAST_SyntaxElement *se = poi->Current_Item();

            se->Accept_Visitor(tcastv);  // παράδειγμα εφαρμογής visitor
      }
      _root->Accept_Visitor(tcastv); // post εφαρμογή visitor στην ρίζα
}
```

Με την σχεδίαση όλων των λειτουργιών του προσδιορισμού τύπου εκφράσεων (inferring expression types) σε μία μόνο κλάση visitor και την εφαρμογή τους με χρήση ενός post-order iterator, επιτυγχάνεται ο διαχωρισμός της λειτουργίας (και των σχετικών μεθόδων) από την ίδια την (composite) δομή και τις διεργασίες διαπέρασης. Είναι σημαντικό ότι η λειτουργία σχεδιάστηκε και υλοποιήθηκε χωρίς να επιφέρει καμία απολύτως παρέμβαση στη σχεδίαση και την υλοποίηση ούτε των στοιχειών της composite δομής, ούτε του iterator, αλλά ούτε και του interface της abstract κλάσης CAST_Visitor.

7.8.3 Γενικά για Context-Sensitive Analysis – Optimization

Η λειτουργία inferring expression types (ενότητα 7.8.2), αποτελεί μια κλασική λειτουργία του front-end των μεταγλωττιστών. Εκτελείτε συνήθως σε ξεχωριστό πέρασμα επί της graphic IR ενός parse ή abstract tree, δια μέσου μίας attribute grammar και υπάγεται στο πεδίο της context sensitive analysis. Άλλες λειτουργίες των μεταγλωττιστών είναι η Control Flow Analysis, Data Dependence Analysis, Call Analysis που εκτελούνται επί αντίστοιχων graphic IRs, όπως Control Flow Graphs, Data Dependence Graphs, Call Graphs δια μέσου εναλλακτικών attribute grammars.

Επίσης λειτουργίες του back-end των μεταγλωττιστών όπως, Register Allocation, Instruction Selection, Tree-Address Code Generation, Low Level Code Generation, εφαρμόζονται σε αντίστοιχες linear ή graphic IRs, όπως parse ή abstract trees δια μέσου αντίστοιχων attribute grammars. Μεθοδολογίες οργάνωσης των ιδιοτήτων με βάση τους κανόνες της γραμματικής (attribute grammars) καθώς και επιμέρους αλγόριθμοι υλοποίησης των εν λόγω λειτουργιών υπάρχουν στην σχετική βιβλιογραφία.

Γενικά οι γραφικές αναπαραστάσεις των μεταγλωττιστών μπορούν να σχεδιαστούν ως composite δομές με βάση τα σχεδιαστικά πρότυπα που παρουσιάσθηκαν, όπου η διαπέραση τους μπορεί να πραγματοποιείται με τους ίδιους iterators και οι σχετικές λειτουργίες να εφαρμόζονται με αντίστοιχους visitor. Ωστόσο η απόφαση για την επιλογή σχεδίασης των λειτουργιών με visitors απαιτεί την εκπλήρωση συγκεκριμένων προϋποθέσεων, όπως η σταθερότητα της δομής, καθώς και το πλήθος των διαφορετικών λειτουργιών που εφαρμόζονται στη δομή, οι οποίες αναλύονται σε επόμενη ενότητα.

7.9 Παράλληλη / σύνθετη εφαρμογή iterator και visitor

Η σχεδίαση και οργάνωση των iterators και visitors σε κλάσεις, επιτρέπει την δημιουργία ανεξάρτητων αντικειμένων διαπέρασης και επίσκεψης composite δομών, που μπορούν να συνδυασθούν και εφαρμοσθούν παράλληλα στην ίδια ή και διαφορετικές δομές. Στο υποθετικό παράδειγμα που ακολουθεί, η διαδικασία διενεργεί μία post order διαπέραση σε μια δομή parse/abstract tree με στοιχεία CAST_SyntxElement και σημείο εκκίνησης τον κόμβο _root, εφαρμόζοντας παράλληλα σε κάθε κόμβο δύο λειτουργίες μέσω των TypeChecking_CAST_Visitor και Process1_CAST_Visitor. Επιπλέον αν κατά την πρώτη διαπέραση και εφαρμογή, βρεθεί κόμβος με συγκεκριμένη ιδιότητα (π.χ. production_rule_code=XXXXX), τότε με αφετηρία αυτόν τον κόμβο, διενεργείται μια pre order διαπέραση στο υπο-δέντρο εφαρμόζοντας σε κάθε κόμβο του τη λειτουργία του Process2_CAST_Visitor.

```
void Multi_CAST_Visitor_Application (CAST_SyntaxElement *_root) {
       TypeChecking_CAST_Visitor *tcastv = new TypeChecking_CAST_Visitor();
       Process1_CAST_Visitor *p1astv = new Process1_CAST_Visitor();
       Process2_CAST_Visitor *p2astv = new Process2_CAST_Visitor();
       Iterator< CAST_SyntaxElement *> *poi =
                    new Postorder_Iterator< CAST_SyntaxElement *>(_root);

       for (poi->First();!poi->Is_Done();poi->Next()) {
              CAST_SyntaxElement *se = poi->Current_Item();

              se->Accept_Visitor(tcastv);  // παράδειγμα εφαρμογής visitor
              se->Accept_Visitor(p1astv);  // παράδειγμα εφαρμογής visitor

              if (se->Get_Production_Rule_Code() == XXXXX) {
                     Iterator< CAST_SyntaxElement *> *proi =
                            new Preorder_Iterator< CAST_SyntaxElement *>(se);
                     se->Accept_Visitor(p2astv); // pre εφαρμογή στη ρίζα
                     for (proi->First();!proi->Is_Done();proi->Next()) {
                            // παράδειγμα εφαρμογής visitor
                            proi->Current_Item()->Accept_Visitor(p2astv);
                     }
              }
       }
       _root->Accept_Visitor(tcastv); // post εφαρμογή visitor στην ρίζα
       _root->Accept_Visitor(p1astv); // post εφαρμογή visitor στην ρίζα
}
```

Το παράδειγμα αναδεικνύει τη δύναμη και την απλότητα εφαρμογής των σχεδιαστικών προτύπων composite, iterator και visitor για την υλοποίηση πολλαπλών λειτουργιών διαμέσου σύνθετων, ταυτόχρονων και διαφορετικών τρόπων διαπέρασης.

7.10 Εφαρμογή Composite, Iterator, Visitor σε Linear IRs

Αναφορικά με τις γραμμικές αναπαραστάσεις δομών (Liner IRs, ενότητα 5.6.2) που μπορεί να χρησιμοποιεί ένας μεταγλωττιστής, είναι δυνατή η εφαρμογή των ίδιων σχεδιαστικών προτύπων. Μία γραμμική δομή στοιχείων είναι στην ουσία μία δενδροειδής δομή (σύνθεση) όπου κάθε κόμβος έχει το πολύ ένα παιδί. Συνεπώς με κατάλληλη δήλωση των γραμμικών στοιχείων ως Composite, μπορεί να χρησιμοποιηθεί το ίδιο Iterator template για την διαπέρασή της. Αντίστοιχα μπορεί να δημιουργηθεί αφηρημένη κλάση visitor για την δήλωση εφαρμογή πολλαπλών λειτουργιών επί των στοιχείων της.

7.10.1 Three-Address Code Linear IR

Μία Linear IR που χρησιμοποιείται σχεδόν από όλες τις υλοποιήσεις μεταγλωττιστών, είναι ο Three-Address Code, ένας pseudo-code που διαθέτει ένα σύνολο λειτουργιών (εντολών) με γενική μορφή **i ← j op k**, με έναν operator (op), δύο operands (j, k) και ένα αποτέλεσμα i. Κάποιες λειτουργίες χρησιμοποιούν λιγότερα ορίσματα, ωστόσο λειτουργίες με περισσότερα ορίσματα εκφράζονται ως ένα σύνολο λειτουργιών Three-Address Code, όπως συμβαίνει αντίστοιχα και σε επίπεδο εκτελέσιμου κώδικα μηχανής.

7.10.2 Σχεδίαση ILOC με Composite, Iterator, Visitor

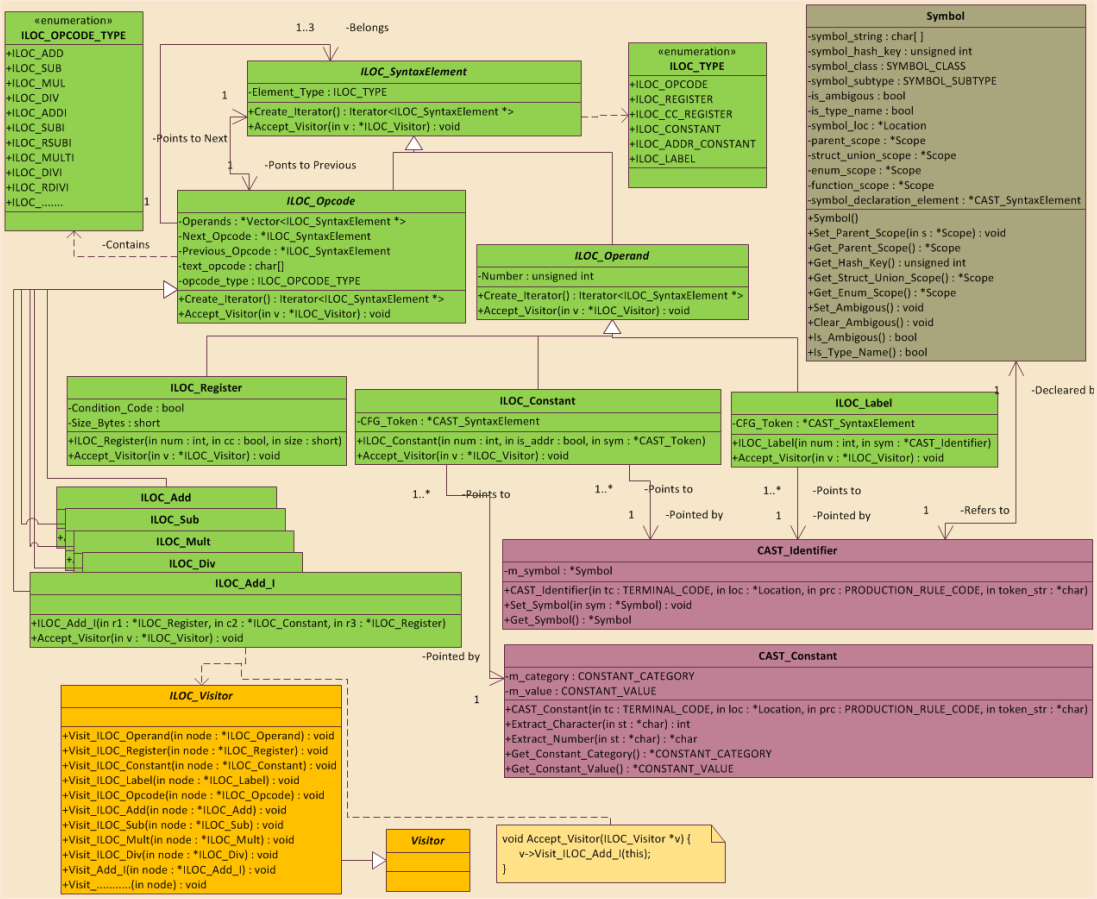
Με βάση τον καλά ορισμένο **ILOC** Three-Address Code (αντίστοιχος με κώδικα assembly για μια αφηρημένη μηχανή) του παραρτήματος Α (Cooper & Torczon, 2012), στη συνέχεια παρουσιάζεται μία ενδεικτική σχεδίαση της Composite δομής του.

Opcode	Sources	Targets	Meaning
add	r_1, r_2	r_3	$r_1 + r_2 \Rightarrow r_3$
sub	r_1, r_2	r_3	$r_1 - r_2 \Rightarrow r_3$
mult	r_1, r_2	r_3	$r_1 \times r_2 \Rightarrow r_3$
div	r_1, r_2	r_3	$r_1 \div r_2 \Rightarrow r_3$
addI	r_1, c_2	r_3	$r_1 + c_2 \Rightarrow r_3$
subI	r_1, c_2	r_3	$r_1 - c_2 \Rightarrow r_3$
rsubI	r_1, c_2	r_3	$c_2 - r_1 \Rightarrow r_3$
multI	r_1, c_2	r_3	$r_1 \times c_2 \Rightarrow r_3$
divI	r_1, c_2	r_3	$r_1 \div c_2 \Rightarrow r_3$
rdivI	r_1, c_2	r_3	$c_2 \div r_1 \Rightarrow r_3$

Εικόνα 7-24 : Three-Address Code Αριθμητικές λειτουργίες ILOC

Στην Εικόνα 7-24, παρουσιάζεται η σύνταξη των ILOC αριθμητικών λειτουργιών. Γενικά μία ILOC εντολή (**opcode**) μπορεί να περιέχει έναν έως τρις operant (sources, targets), όπου κάθε operand μπορεί να είναι τύπου **register** (ή **condition code register**), **Constant** ή **Label**.

Εξαίρεση αποτελεί η εντολή (opcode) της **ϕ-function** (για την **SSA**[20] μορφή) που μπορεί να περιέχει περισσότερους source operands.



Εικόνα 7-25 : Διάγραμμα κλάσεων Composite σύνθεσης ILOC

Η σχεδίαση (Εικόνα 7-25) της ILOC IR δομής (σύνθεσης) με χρήση των προτύπων Composite, Iterator και Visitor αποτυπώνει μια σύνθεση με ξεχωριστά αντικείμενα (κλάσεις) για κάθε εντολή (opcode) καθώς και για κάθε είδος operand της εντολής. Διαφαίνεται μια ομοιότητα με την parse/abstract tree Graphic IR σε μια πιο μικρή κλίμακα. Στην πράξη πίσω από την ILOC υπάρχει μία πιο απλή LR(1) CFG. Τα χαρακτηριστικά της ILOC που μας οδήγησαν στη συγκειμένη σύνθεση είναι α) το γεγονός ότι η ILOC είναι μια τυποποιημένη γλώσσα (χαμηλού επιπέδου) και άρα η δομής της είναι σταθερή, β) μία ILOC αναπαράσταση περιέχει σχεδόν όλη την πληροφορία του προγράμματος και πάνω στα στοιχεία της επιτελούνται (ανάλογα με την υλοποίηση) σημαντικός αριθμός λειτουργιών τόσο του optimization (πχ. μέσω SSA μορφής) όσο και του back-end (πχ. register allocation, instruction scheduling, code generation).

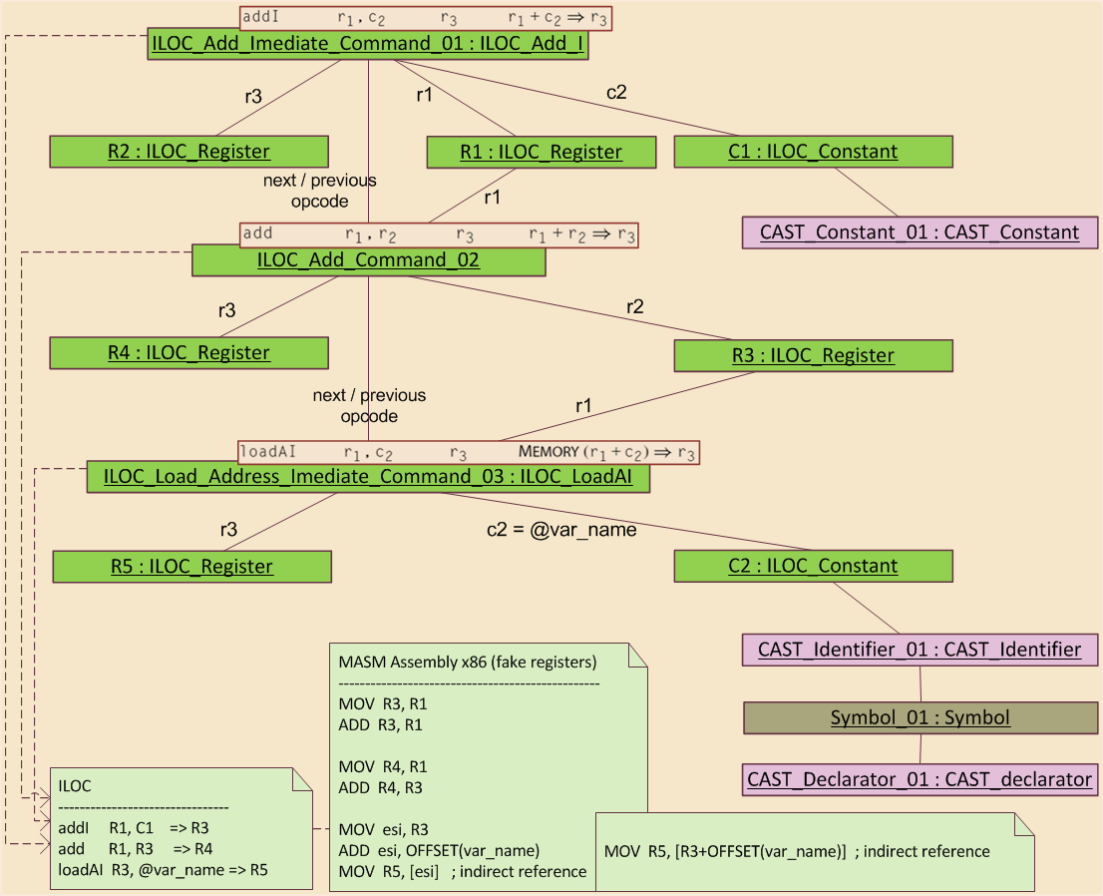
Επίσης δίνεται και η abstract class του ILOC_Visitor με μεθόδους για κάθε τύπο αντικειμένου, συμπεριλαμβανομένων και των ομαδικών τύπων ILOC_Opcode και ILOC_Operand. Ακόμη κάθε αντικείμενο ILOC_Constant και ILOC_Label συσχετίζεται με τα αντίστοιχα αντικείμενα CAST_Identifier και CAST_Constant προκειμένου να υπάρχει η διασύνδεση της ILOC με το Parse/Abstract Tree του προγράμματος. Περεταίρω κάθε

[20] **Static Single-Assignment form** : Μορφή αναπαράστασης του προγράμματος που εμπεριέχει την ροή δεδομένων και πληροφορίας (data and control flow) ταυτόχρονα στην ίδια IR. Από μια SSA IR, ο μεταγλωττιστής μπορεί να υλοποιήσει οποιαδήποτε κλασική μέθοδο βελτιστοποίησης (optimization)

CAST_Identifier αντικείμενο έχει συσχετισθεί (από τις ενέργειες του parser) με ένα αντικείμενο Symbol του πίνακα συμβόλων, οπού με τη σειρά του συσχετίζεται με το αντικείμενο (symbol_declaration_element) της declaration του parse/abstract tree.

7.10.3 Συσχέτιση αντικειμένων Symbol Table, Abstract Tree, ILOC Three-Address Code

Οι προαναφερόμενες εξαρτήσεις, συσχετίζουν και τις τρις δομές (συνθέσεις) αναπαράστασης που έχουμε παρουσιάσει στην εργασία και αποτελούν συνήθως βασικές εσωτερικές αναπαραστάσεις ενός τυπικού μεταγλωττιστή.



Εικόνα 7-26 : Διάγραμμα Αντικειμένων ILOC σύνθεσης

Στο παράδειγμα (Εικόνα 7-26), παρουσιάζεται το διάγραμμα αντικείμενων για τρις ILOC εντολές (addI, add, loadAI). Στην πρώτη εντολή η σταθερά C1 αναφέρεται σε μία constant του προγράμματος και συσχετίζεται με το αντίστοιχο CAST_Constant αντικείμενο. Στην τρίτη εντολή η σταθερά C2 αναφέρεται στη μεταβλητή var_name του προγράμματος και συσχετίζεται με το αντίστοιχο CAST_Identifier αντικείμενο του parse tree, το οποίο συσχετίζεται με το Symbol αντικείμενο του symbol table, που τελικά συσχετίζεται με το CAST_declarator αντικείμενο του parse tree. Κατά την επεξεργασία της three-address code αναπαράστασης, όλη πληροφορία των ενδιάμεσων αναπαραστάσεων είναι διαθέσιμη (εφόσον απαιτείται από τη υλοποίηση). Στο διάγραμμα παρουσιάζεται και ο ενδεικτικός Assembly κώδικας (με εικονικούς register) προκειμένου να επισημανθεί η αμεσότητα της ILOC σε σχέση με το instruction set του επεξεργαστή.

Η χρήση των iterator και visitor στην ILOC σύνθεση ακολουθεί τη λογική και τα ενδεικτικά παραδείγματα των προηγούμενων συνθέσεων, αναδεικνύοντας για άλλη μια φορά την χρησιμότητά των συγκεκριμένων σχεδιαστικών προτύπων στους μεταγλωττιστές.

7.11 Αξιολόγηση – μέτρηση ποιότητας υλοποίησης

Η μέτρηση συγκεκριμένων μετρικών και σύγκρισή τους για υλοποιήσεις μεταγλωττιστών με και χωρίς την χρήση των σχεδιαστικών προτύπων ξεφεύγει από τους σκοπούς της εργασίας. Στο Παράρτημα IV : Μετρήσεις ποιότητας λογισμικού, παρουσιάζονται αποσπάσματα γράφων Kiviat και επιμέρους πινάκων μετρήσεων ανά μέθοδο για τέσσερα ενδεικτικά module της υλοποίησης της εργασίας, με βάση τα σχεδιαστικά πρότυπα façade, composite, iterator, και visitor. Παρουσιάζονται αποτελέσματα μετρήσεων για ένα σύνολο βασικών μετρικών ποιότητας λογισμικού οι οποίες αν και δεν αποτυπώνουν άμεσα τα ποιοτικά χαρακτηριστικά από τη χρήση των σχεδιαστικών προτύπων, ωστόσο αποτυπώνουν έμμεσα την ποιότητα του λογισμικού ως αποτέλεσμα καλής σχεδίασης με βάση δοκιμασμένα και δομημένα σχεδιαστικά πρότυπα που εκμεταλλεύονται τα πλεονεκτήματα του OOP.

8 Επιλογή Σχεδιαστικών Προτύπων

Η ανάπτυξη ενός μεταγλωττιστή, αν και εξωτερικά φαντάζει ως κάτι εξαιρετικά δύσκολο και περίπλοκο, στην πραγματικότητα και σε γενικές γραμμές μπορούμε να πούμε ότι αποτελείται από τα ακόλουθα αφαιρετικά βήματα:

- Μετατροπή του προγράμματος (είσοδος) σε ένα parse / abstract tree
- Υλοποίηση semantic/context sensitive analysis (πχ type checking) επί του δέντρου
- Επεξεργασία ή/και αναμόρφωση του δέντρου για την υλοποίηση βελτιστοποιήσεων
- Μετατροπή του δέντρου στο πρόγραμμα στόχος (target code)

Όπως έχει ήδη αναφερθεί, κάθε υλοποίηση μπορεί να σχηματίζει διαφορετικού τύπου αναπαραστάσεις με εναλλακτικές δομές. Ωστόσο κυριαρχούν οι Graphic IRs όπως Parse tree, Abstract, Control Flow Graphs, Data Dependence Graphs, Call Graphs κ.λ.π. με βασικότερη την αρχική αναπαράσταση parse / abstract tree, η οποία συνήθως είναι μια αναπαράσταση του προγράμματος βασισμένη στα συντακτικά στοιχεία της γραμματικής της γλώσσας. Η δομή αυτή είναι η πιο σύνθετη και ανάλογη με την πολυπλοκότητα της γλώσσας. Οι λειτουργίες που επιτελούνται στις δομές αυτές έχουν να κάνουν με την semantic/context sensitive analysis του προγράμματος και συνήθως εξειδικεύονται για τον κάθε κανόνα ξεχωριστά. Η μοντελοποίηση της context sensitive πληροφορίας του προγράμματος πραγματοποιείται διαμέσου του attribute grammar framework με το οποίο αλληλεπιδρούν οι επιμέρους λειτουργίες. Η σχεδίαση τέτοιων δενδροειδών δομών εξυπηρετείται καλύτερα από ΟΟ γλώσσες και είναι καθοριστικής σημασίας για την ποιότητα του παραγόμενου λογισμικού. Βασική επιδίωξη των εφαρμοζόμενων σχεδιαστικών προτύπων είναι η βελτίωση των ποιοτικών χαρακτηριστικών της συντηρησιμότητας και ευχρηστίας του λογισμικού. Επιδιώκεται η κατά το δυνατό απλούστερη διαπέραση και εφαρμογή λειτουργιών επί των στοιχείων της δομής με ενιαίο τρόπο. Επίσης σημαντικό στόχο αποτελεί η κατανόηση της και η ευκολία συντήρησης και κυρίως επέκτασης των λειτουργιών επί της δομής κατά το τελικό στάδιο της χρήσης και συντήρησης.

Όπως παρουσιάσθηκε και στην ΟΟ σχεδίαση της υλοποίησης (κεφάλαιο, 7), η δομή του parse/abstract tree υλοποιείται με βάση το σχεδιαστικό πρότυπο composite, η διαπέραση των στοιχείων της δομής του δέντρου με βάση το σχεδιαστικό πρότυπο iterator και η εφαρμογή των λειτουργιών (semantic/context sensitive analysis) με βάση το σχεδιαστικό πρότυπο visitor. Από τις επιμέρους υλοποιήσεις προκύπτει το συμπέρασμα ότι η εφαρμογή των σχεδιαστικών προτύπων composite και iterator έχουν αδιαμφισβήτητο όφελος δεδομένου ότι κάνουν χρήση των ΟΟ ιδιοτήτων της ιεραρχίας, της κληρονομικότητας και του πολυμορφισμού, προσφέροντας τυποποιημένα interface για την διαπέραση των αντικειμένων της δομής. Μάλιστα οι iterators δια μέσου του σχετικού template που παρουσιάσθηκε, έχουν ευρεία χρήση σε διαφορετικές δομές ανεξάρτητα από των τύπο και την ιεράρχηση των επιμέρους στοιχείων τους. Ωστόσο η εφαρμογή του σχεδιαστικού προτύπου visitor σε composite δομές ενδέχεται να επιδράσει αρνητικά στην ποιότητα του λογισμικού.

Γενικά θα λέγαμε ότι στους μεταγλωττιστές παρατηρείται συχνά η ανάγκη εφαρμογής των σχεδιαστικών προτύπων composite, iterator και visitor σε αρκετές φάσεις μεταγλώττισης που έχουν να κάνουν κυρίως με την διαχείριση (εφαρμογή λειτουργιών) σε δομές / συλλογές δεδομένων γραφικής αναπαράστασης (graphics IRs).

8.1 Πρότυπα και σχεδίαση μεταγλωττιστών

Η δυσκολία στους μεταγλωττιστές δεν εντοπίζεται τόσο στα εμπλεκόμενα βήματα, τα οποία με μια καλή σχεδίαση είναι αρκετά διακριτά μεταξύ τους, αλλά στον πολύ μεγάλο αριθμό των λεπτομερειών ανά λειτουργία και κανόνα της γραμματικής που πραγματοποιούνται στις εσωτερικές δομές αναπαράστασης (πχ parse/ abstract trees). Στην προσπάθεια απλούστευσης εφαρμογής των επιμέρους λειτουργιών σε τέτοιου είδους δομές, δύο σχεδιαστικά πρότυπα έχουν επικρατήσει, αυτό της Inheritance και του Visitor.

8.1.1 Σχεδιαστική προσέγγιση μέσω Inheritance

Στο σχεδιαστικό πρότυπο **Inheritance**,

- δηλώνεται ένας τύπος κόμβου για κάθε σύνθετο αντικείμενο της γλώσσας εξειδικεύοντας τα αντικείμενα μέχρι τους στοιχειώδεις τύπους αντικειμένων, έτσι ώστε όλοι οι τύποι να κληρονομούν την κύρια κλάση (τύπο) κόμβου,
- δηλώνονται virtual methods στον κύρια κλάση (τύπο) κόμβου για το σύνολο των λειτουργιών που εφαρμόζονται στους επιμέρους τύπους αντικειμένων (π.χ. semantic analysis, optimization, code generation),
- Επικαλύπτεται η υλοποίηση των virtual methods από επιμέρους υλοποιήσεις μεθόδων των υπό κλάσεων, προκειμένου να υλοποιηθούν οι εκάστοτε εξειδικευμένες λειτουργίες (ανά τύπο κόμβου)

Στο συγκεκριμένο σχεδιαστικό πρότυπο όλες οι λειτουργίες ενσωματώνονται διάσπαρτα στις κλάσεις των αντικειμένων, καθιστώντας δύσκολη την κατανόηση του κώδικα και του ελέγχου των λειτουργιών αυτών. Ένα άλλο βασικό πρόβλημα του ανωτέρου σχεδιαστικού προτύπου προκύπτει όταν επιχειρείται μεταγενέστερα η τροποποίηση ή η προσθήκη μίας επιπλέον λειτουργίας, όπου και απαιτείται η τροποποίηση σχεδόν όλων των κλάσεων της δομής. Για την εξουδετέρωση του προβλήματος συνήθως επιχειρείται η υλοποίηση των αφαιρετικών λειτουργιών (methods) σε όσο το δυνατό ποιο κοινές (αφαιρετικές) κλάσεις της ιεραρχίας της δομής. Ωστόσο έτσι προκύπτει το πρόβλημα της απώλειας του ελέγχου των επιμέρους υλοποιήσεων ανά τύπο κλάσης για κάθε λειτουργία. Δηλαδή αν ένα συγκεκριμένος υπό τύπος (κλάση) αντικειμένου χρειάζεται μια διαφορετική υλοποίηση, η οποία διέφυγε του προγραμματιστή, αυτό δεν εντοπίζεται κατά τη μεταγλώττιση (εφόσον εκτελείται η αμέσως επόμενη virtual method της ιεραρχίας) και πρέπει να διενεργηθεί ενδελεχής έλεγχος (debugging) του μεταγλωττιστή.

Ωστόσο το πρότυπο Inheritance υπερτερεί στην περίπτωση που απαιτείται η μεταγενέστερη τροποποίηση της δομής των κλάσεων. Στην περίπτωση αυτή απαιτείται μόνο η τροποποίηση των συγκεκριμένων κλάσεων της δομής και των αντίστοιχων μεθόδων των λειτουργιών για τους συγκεκριμένους τύπους (κλάσεις) της δομής.

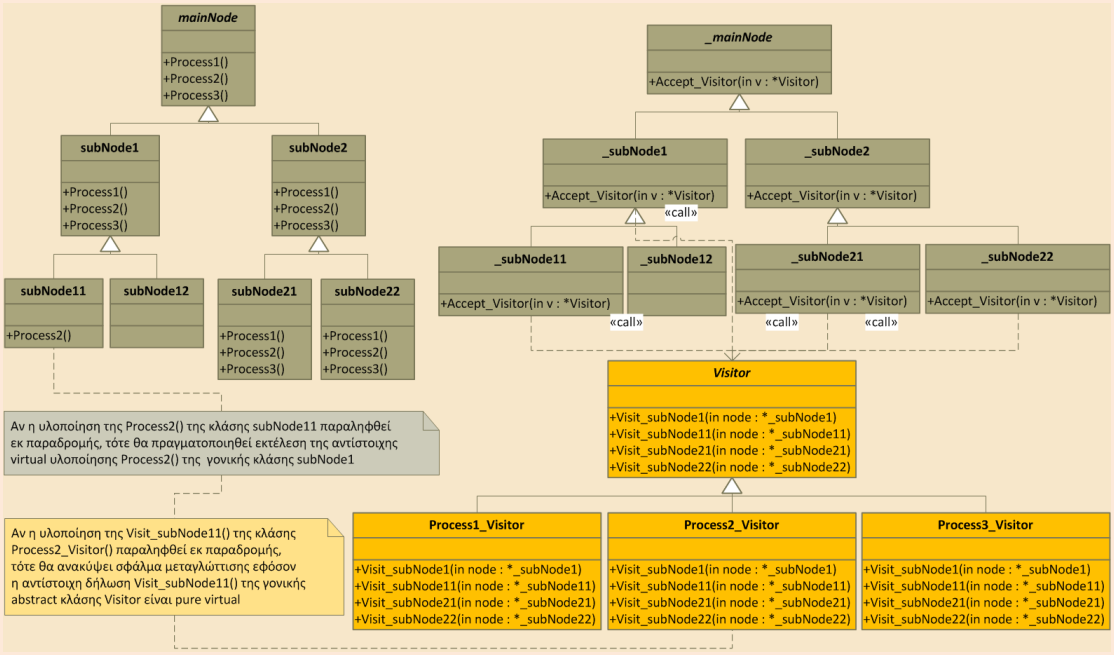
8.1.2 Σχεδιαστική προσέγγιση μέσω Visitors

Στο σχεδιαστικό πρότυπο **Visitor**, το οποίο και εφαρμόζεται στην υλοποίηση της εργασίας,

- δηλώνεται ένας τύπος κόμβου για κάθε σύνθετο αντικείμενο της γλώσσας εξειδικεύοντας τα αντικείμενα μέχρι τους στοιχειώδεις τύπους αντικειμένων, έτσι ώστε όλοι οι τύποι να κληρονομούν την κύρια κλάση (τύπο) κόμβου,
- δηλώνεται μία abstract κλάση visitor, η οποία περιλαμβάνει pure virtual μεθόδους για όλους τους τύπους κόμβων της δομής,
- ορίζεται μια διαδικασία διαπέρασης (π.χ. iterator) που επιδρά στην κύρια κλάση (τύπο) κόμβου και διαπερνά τα αντικείμενα της δομής με συγκεκριμένη σειρά, όπου κάθε αντικείμενο πραγματοποιεί κλήσεις (callbacks) σε ένα αντικείμενο visitor που παρέχεται μέσω παραμέτρου,
- δημιουργούνται πολλαπλές (μία για κάθε λειτουργία) κλάσεις που κληρονομούν την abstract κλάση visitor, όπου κάθε μία υλοποιεί τις μεθόδους, για όλους τους τύπους κόμβων της δομής, για τη συγκεκριμένη λειτουργία

Στο συγκεκριμένο σχεδιαστικό πρότυπο κάθε λειτουργία δηλώνεται συγκεντρωτικά ως ξεχωριστή υπο-κλάση visitor με μεθόδους με τις επιμέρους λειτουργίες για όλους τους τύπους κόμβων της δομής. Οι κλάσεις των αντικειμένων δε επηρεάζονται, καθιστώντας έτσι εύκολη την κατανόηση του κώδικα και του ελέγχου των λειτουργιών αυτών. Το βασικό πλεονέκτημα του ανωτέρου σχεδιαστικού προτύπου προκύπτει όταν επιχειρείται μεταγενέστερα η τροποποίηση ή η προσθήκη μίας επιπλέον λειτουργίας, όπου και απαιτείται η τροποποίηση ή η δημιουργία μόνο της υπο-κλάσης του συγκεκριμένου visitor. Επιπλέον παρέχεται καλύτερος έλεγχος των επιμέρους υλοποιήσεων ανά τύπο κλάσης για κάθε λειτουργία. Δηλαδή αν ένα συγκεκριμένος υπό τύπος (κλάση) αντικειμένου χρειάζεται μια διαφορετική υλοποίηση, η οποία διέφυγε του προγραμματιστή, αυτό εντοπίζεται ως σφάλμα κατά τη μεταγλώττιση (εφόσον απαιτείται η υλοποίηση κάθε pure virtual method για κάθε υπο-κλάση visitor) διευκολύνοντας και προλαμβάνοντας την εμφάνιση σφαλμάτων στο μεταγλωττιστή.

Ωστόσο το πρότυπο του visitor υστερεί στην περίπτωση που απαιτείται η μεταγενέστερη τροποποίηση της δομής των κλάσεων. Στην περίπτωση αυτή εκτός της τροποποίησης των συγκεκριμένων κλάσεων της δομής απαιτείται και η τροποποίηση όλων των κλάσεων των visitor που επιδρούν σε αυτή τη δομή και για όλες τις μεθόδους που αντιστοιχούν στους τροποποιούμενους τύπους κλάσεων της δομής.



Εικόνα 8-1 : Διάγραμμα κλάσεων προτύπων Inheritance - Visitor

Η Εικόνα 8-1 παρουσιάζει ένα ενδεικτικό διάγραμμα κλάσεων των σχεδιαστικών προτύπων Inheritance και Visitor σε αντιπαράθεση.

8.2 Aspect-Oriented Programming

Το πρότυπο Inheritance καθίσταται προβληματικό όταν χρειάζεται η προσθήκη νέων λειτουργιών και το πρότυπο Visitor καθίσταται προβληματικό όταν χρειάζεται ένας νέος τύπος κόμβου ή γενικά η τροποποίηση της δομής. Τα δύο πρότυπα έχουν αντιφατικά πλεονεκτήματα που δεν μπορούν να ικανοποιηθούν και υλοποιηθούν ταυτόχρονα κυρίως λόγω έλλειψης γλωσσών προγραμματισμού που να υποστηρίζουν και τα δύο στυλ σχεδίασης προγράμματος. Ένα νέο πεδίο στη σχεδίαση γλωσσών προγραμματισμού που έχει προταθεί τα τελευταία χρόνια είναι το **Aspect-Oriented Programming** (AOP).

> *«Ο AOP βασίζεται στην ιδέα ότι τα συστήματα υπολογιστών προγραμματίζονται καλύτερα προσδιορίζοντας ξεχωριστά τις διάφορες υποθέσεις (ιδιότητες ή πεδία ενδιαφέροντος) ενός συστήματος και κάποια περιγραφή των σχέσεών τους, και στη συνέχεια στηριζόμενοι στον βασικό μηχανισμό του AOP περιβάλλοντος, να τα συνθέτει σε ένα συνεκτικό πρόγραμμα … Ενώ η τάση στον ΟΟ προγραμματισμό είναι να εντοπίζει ομοιότητες μεταξύ των κλάσεων και να τις προωθεί (προς τα πάνω) στο δέντρο της μεταξύ τους ιεραρχίας, ο ΟΑΡ προσπαθεί να συλλάβει τις διάσπαρτες υποθέσεις ως first-class στοιχεία και να τα εξάγει οριζόντια από τη δομή των αντικειμένων» (Elrad , Filman, & Bader, 2001).*

Ο AOP δίνει κάποιες προοπτικές για την επίλυση όλων των προβλημάτων και των δύο σχεδιαστικών προτύπων. Γενικά κατά την μετακίνηση από το πρότυπο Inheritance στο πρότυπο του Visitor επιχειρούμε να εξάγουμε τις λειτουργίες που επιδρούν στου κόμβους της δομής οριζόντια. Ωστόσο αυτό δεν λειτουργεί όπως θα περιμέναμε εφόσον η εγγενής πολυπλοκότητα της δομής των κόμβων εξακολουθεί να εμπλέκεται και στους visitors (αλλαγή της δομής των κόμβων επιφέρει τροποποίηση όλων των visitor). Ο AOP μας

επιτρέπει να προγραμματίζουμε χωρίς τα μειονεκτήματα που εισάγει το πρότυπο του Visitor.

Η εφαρμογή του AOP αντί να εισάγει νέες περίπλοκες γλώσσες προγραμματισμού συνήθως υλοποιείται από ενδιάμεσα εργαλεία (λογισμικά), παρόμοια με αυτά του Flex και Bison. Τα εργαλεία αυτά δέχονται ως είσοδο μία aspect oriented γλώσσα προγραμματισμού για την δήλωση και την διαχείριση των κόμβων και των λειτουργιών και χρησιμοποιεί μια βασική γλώσσα (π.χ. C, C++, C#, Java) για την υλοποίηση (παραγωγή κώδικα) των λειτουργιών αυτών. Ένα AOP εργαλείο προσανατολισμένο στην υλοποίηση μεταγλωττιστών είναι το είναι το **Treecc**[21].

8.3 Αξιολόγηση – μέτρηση ποιότητας προτύπων και λογισμικού

8.3.1 Αξιολόγηση σχεδιαστικών προτύπων

8.3.1.1 Εκτίμηση ασυμπτωτικών δεικτών

Η αξιολόγηση των σχεδιαστικών προτύπων στους μεταγλωττιστές (αλλά και γενικά) μπορεί να γίνει με βάση μια εκτίμηση των παρεμβάσεων / χρόνου που απαιτείται για μια σειρά ενεργειών όπως προσθήκη κόμβου, προσθήκη λειτουργίας καθώς και με την καταγραφή χαρακτηριστικών όπως συγκέντρωση λειτουργιών, έλεγχος επιμέρους υλοποιήσεων ενεργειών. Η αξιολόγηση ουσιαστικά επικεντρώνεται στα σχεδιαστικά πρότυπα Inheritance και Visitor εφόσον τα πρότυπα Composite και Iterator είναι ανεξάρτητα και κατά κανόνα αναβαθμίζουν την ποιότητα του λογισμικού.

Δεδομένης μίας ιεραρχικής δομής **K** διαφορετικών κόμβων (συνολικών κλάσεων ιεραρχίας δομής) και **N** διαφορετικών τύπων αντικειμένων (κλάσεις αντικειμένων στο χαμηλότερο επίπεδο ιεραρχίας - leafs) όπου κατά μέσο όρο δηλώνονται **M** διαφορετικές λειτουργίες (methods) που εφαρμόζονται στους κόμβους της δομής, προκύπτουν οι παρακάτω ασυμπτωτικές[22] εκτιμήσεις (Πίνακας 8-1).

Πίνακας 8-1 : Αξιολόγηση σχεδιαστικών προτύπων Inheritance – Visitor - Iterator

Δείκτες / Χαρακτηριστικά	Visitor	Inheritance	Iterator
Απαιτήσεις τροποποιήσεων / προσθηκών **κλάσεων** σε περίπτωση **προσθήκης νέου τύπου κόμβου**	1+M = O(M)	1 = **O(1)**	max 1 = **O(1)**
Απαιτήσεις τροποποιήσεων / προσθηκών **μεθόδων** σε περίπτωση **προσθήκης νέου τύπου κόμβου**	1+M = O(M)	M = O(M)	max 4 = **O(1)**
Απαιτήσεις τροποποιήσεων / προσθηκών **κλάσεων** σε περίπτωση **προσθήκης νέας λειτουργίας**	1 = **O(1)**	K = O(K)	---
Απαιτήσεις τροποποιήσεων / προσθηκών **μεθόδων** σε περίπτωση **προσθήκης νέας λειτουργίας**	N = O(N)	K*1 = O(K)	---

[21] On line οδηγίες σχετικά με το λογισμικό Treecc (Tree Compiler-Compiler) παρέχονται στην τοποθεσία http://www.gnu.org/software/dotgnu/treecc/treecc.html

[22] Αφορά αποτίμηση επιπτώσεων (τροποποιήσεων / προσθηκών), εκφραζόμενη κατά αντιστοιχία με τον ασυμπτωτικό συμβολισμό της θεωρίας ασυμπτωτικής απόδοσης αλγορίθμων (Cormen, Leiserson, Rivest, & Stein, 2009)

Δείκτες / Χαρακτηριστικά	Visitor	Inheritance	Iterator
Κατανομή υλοποιήσεων λειτουργιών ανά κλάση κόμβου	ΟΧΙ	ΝΑΙ	---
Συγκέντρωση υλοποιήσεων λειτουργιών ανά κλάση λειτουργίας	**ΝΑΙ**	ΟΧΙ	---
Έλεγχος επιμέρους υλοποιήσεων ενεργειών ανά τύπο κόμβου από τη γλώσσα προγραμματισμού	**ΝΑΙ**	ΟΧΙ	---
Ευελιξία στην υλοποίηση και εφαρμογή λειτουργιών τόσο στην γονική κλάση όσο και στις επιμέρους κλάσεις της ιεραρχίας	ΟΧΙ	ΝΑΙ	---

Οι δείκτες και τα χαρακτηριστικά του ανωτέρου πίνακα επιβεβαιώνουν τις προδιαγραφές των σχεδιαστικών προτύπων, όπου κυρίαρχο κριτήριο μεταξύ Inheritance και Visitor αναδεικνύεται η σταθερότητα της δομής (κλάσεις κόμβων) έναντι της σταθερότητας των λειτουργιών (μέθοδοι κλάσεων) κατά τη διάρκεια υλοποίησης και συντήρησης της αναπτυσσόμενης εφαρμογής. Στους μεταγλωττιστές και ειδικά στην περίπτωση των parse/abstract trees, η δομή των οποίων βασίζεται στη γραμματική της γλώσσας εισόδου, εκτιμάται ότι είναι σπάνιο να αλλάξει η δομή της αναπαράστασης και περισσότερο πιθανό να τροποποιηθούν ή προστεθούν νέες λειτουργίες τόσο στο στάδιο της υλοποίησης όσο και της συντήρησης του μεταγλωττιστή. Συνεπώς το πρότυπο του Visitor φαίνεται να κερδίζει έδαφος στην σχεδίαση μεταγλωττιστών έναντι του προτύπου Inheritance. Επιπλέον το πρότυπο Visitor συνδυάζει και τα πλεονεκτήματα (ποιοτικά χαρακτηριστικά), α) του ελέγχου των επιμέρους υλοποιήσεων ενεργειών ανά τύπο κόμβου από τη γλώσσα προγραμματισμού, και β) της συγκέντρωσης των υλοποιήσεων λειτουργιών ανά κλάση λειτουργίας (visitor) και ανεξάρτητα της δομής που επιδρούν.

8.3.1.2 Γραφική αναπαράσταση (περίπτωση M=0.2N)

Προκειμένου να αποτυπωθούν με σαφή τρόπο οι επιπτώσεις των παρεμβάσεων στα δύο πρότυπα, παρακάτω τυποποιούνται τα κριτήρια (Πίνακας 8-1) με τον εξής συμβολισμό:

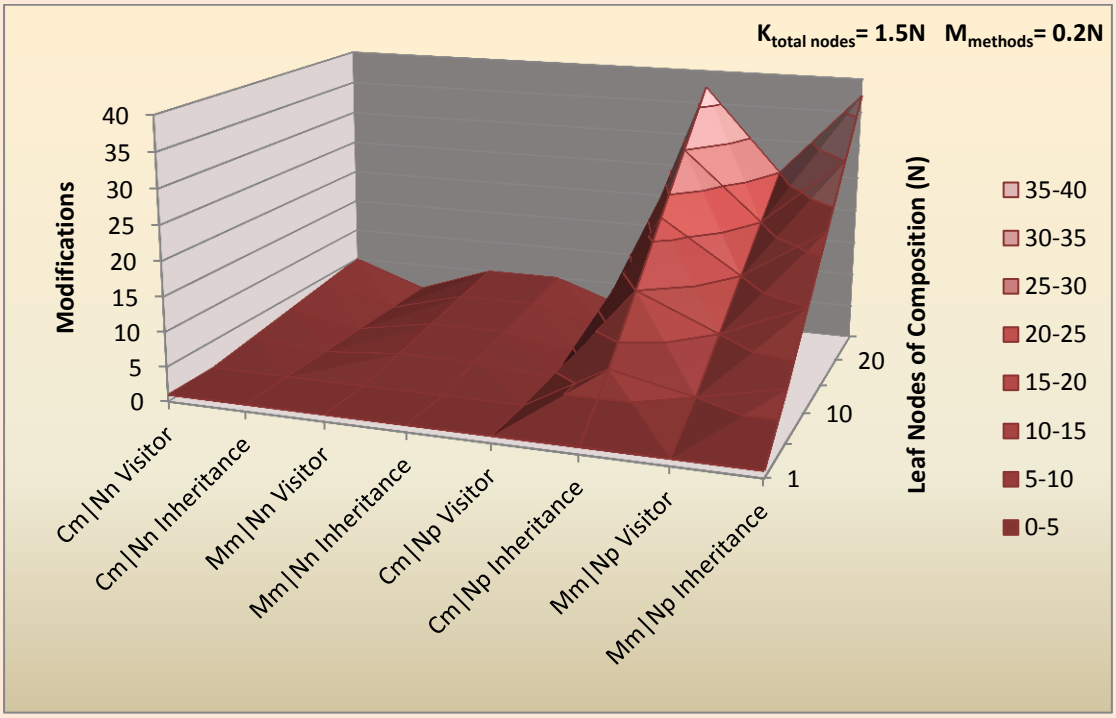
- **Cm|Nn** = Class modifications for New node = **Απαιτήσεις τροποποιήσεων / προσθηκών** κλάσεων σε περίπτωση **προσθήκης νέου τύπου κόμβου**
- **Mm|Nn** = Method modifications for New node = **Απαιτήσεις τροποποιήσεων / προσθηκών** μεθόδων σε περίπτωση **προσθήκης νέου τύπου κόμβου**
- **Cm|Np** = Class modifications for New process = **Απαιτήσεις τροποποιήσεων / προσθηκών** κλάσεων σε περίπτωση **προσθήκης νέας λειτουργίας**
- **Mm|Np** = Method modifications for New process = **Απαιτήσεις τροποποιήσεων / προσθηκών** μεθόδων σε περίπτωση **προσθήκης νέας λειτουργίας**

Σε μία σύνθεση (Composite) κλάσεων με σχέσεις κληρονομικότητας, η ύπαρξη μίας κλάσης με έναν μόνο απόγονο (αν και μπορεί να δηλωθεί) δεν έχει νόημα, και άρα μπορούμε να πούμε ότι σε μία Composite δομή κάθε κλάση έχει τουλάχιστον δύο απογόνους.

Από τη θεωρία των γράφων, ένα δυαδικό δέντρο, όπου κάθε κόμβος μπορεί να είναι τερματικός (leaf) ή να έχει υποχρεωτικά δύο παιδιά (εσωτερικός κόμβος), γνωρίζουμε ότι

για N τερματικούς κόμβους το δέντρο έχει N-1 εσωτερικούς κόμβους. Επίσης αν κάθε εσωτερικός κόμβος μπορεί να περιέχει δύο ή περισσότερα παιδία, τότε ο αριθμός των εσωτερικών κόμβων κυμαίνεται από 1 έως και N-1, και ο συνολικός αριθμός των κόμβων (leaf + εσωτερικών) από N+1 έως και N+N-1. Άρα σε μία μέση περίπτωση μια Composite δομή κλάσεων για N τερματικές κλάσεις (αντικειμένων) έχει N/2 εσωτερικές κλάσεις (συνήθως abstract class) και άρα N+N/2 ή 1.5N συνολικές κλάσεις.

Έστω μια γενική περίπτωση, όπου η ιεράρχηση της δομής (composite) είναι ένα δέντρο μέσης περίπτωσης, τότε για N διαφορετικά αντικείμενα (κλάσεις leafs) υπάρχουν **K=1.5N** συνολικές κλάσεις (inner nodes + leafs). Έστω επίσης ότι δημιουργούμε μία νέα λειτουργία για κάθε πέντε διαφορετικά αντικείμενα, δηλαδή **M=0.2N**. Για την συγκεκριμένη γενική περίπτωση οι ασυμπτωτικές εκτιμήσεις των παρεμβάσεων / αλλαγών που προκύπτουν ανά κριτήριο (Πίνακας 8-2) αποτυπώνονται στο παρακάτω γράφημα (Εικόνα 8-2).



Εικόνα 8-2 : Γράφημα ασυμπτωτικών εκτιμήσεων (M=0.2N)

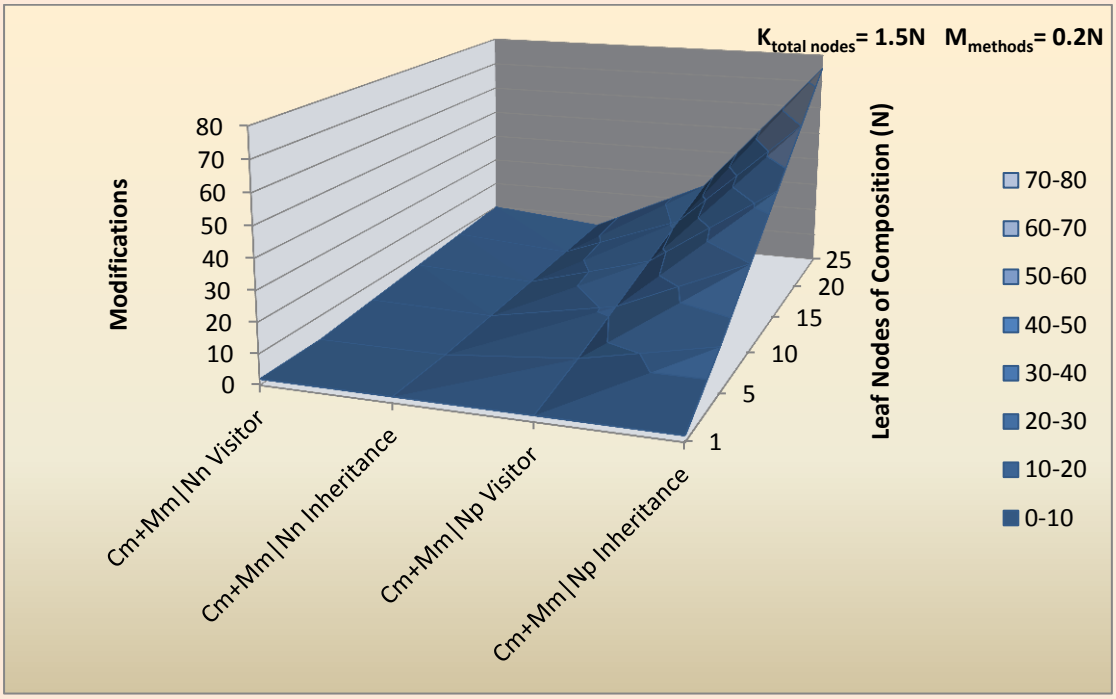
Στο ανωτέρω γράφημα στον οριζόντιο άξονα (x) παρατίθενται τα επιμέρους κριτήρια, στον κάθετο άξονα (y) οι τιμές των ασυμπτωτικών εκτιμήσεων, και στο άξονα (z) ενδεικτικές αυξητικές τιμές για το N. Για παράδειγμα για N=20, τότε K=30 και M=4. Προφανώς και το ζητούμενο είναι η κατά το δυνατό λιγότερες παρεμβάσεις και τροποποιήσεις, άρα τα βέλτιστα σημεία του γραφήματος εντοπίζονται στις χαμηλότερες ασυμπτωτικές τιμές ή αλλιώς στη χαμηλότερη στάθμη της επιφάνειας του γραφήματος.

Προκειμένου να αποτυπωθούν τα μεγέθη με βάση τα κριτήρια νέου κόμβου (Nn) και νέας λειτουργίας (Np), ομαδοποιούμε τα επιμέρους κριτήρια ως εξής:

- **Cm+Mm|Nn** = Class and Method modifications for New node = **Απαιτήσεις τροποποιήσεων / προσθηκών** κλάσεων και μεθόδων σε περίπτωση **προσθήκης νέου τύπου κόμβου**

- **Cm+Mm|Np** = Class and Method modifications for New process = **Απαιτήσεις τροποποιήσεων / προσθηκών** κλάσεων και μεθόδων σε περίπτωση **προσθήκης νέας λειτουργίας**

Από όπου και προκύπτει το σχετικό γράφημα (Εικόνα 8-3).

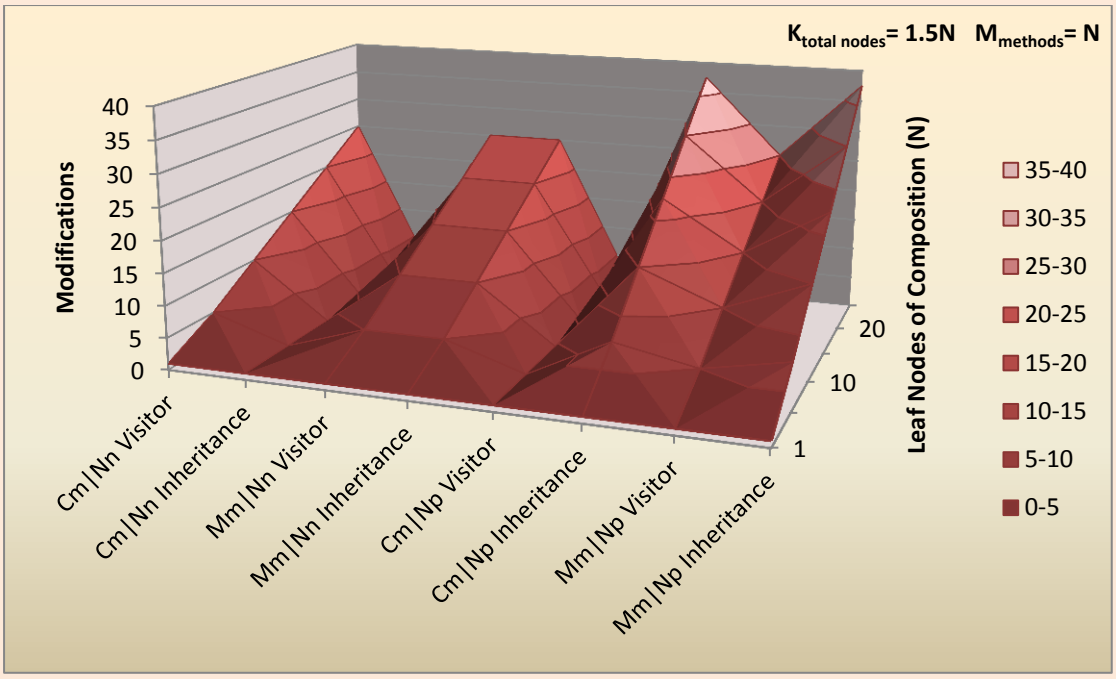


Εικόνα 8-3 : Γράφημα ομάδων ασυμπτωτικών εκτιμήσεων (M=0.2N)

Από το ανωτέρω γράφημα (Εικόνα 8-3), προκύπτει σαφώς ότι στην περίπτωση προσθήκης νέων διαδικασιών (Np), το πρότυπο Visitor επιφέρει λιγότερες τροποποιήσεις μεθόδων και κλάσεων (Cm+Mm) σε σχέση με το πρότυπο Inheritance. Από την άλλη στην περίπτωση προσθήκης νέων τύπων κόμβων (Nn) τα δύο πρότυπα φαίνεται να έχουν παρεμφερείς επιπτώσεις. Αυτό οδηγεί στο γενικό συμπέρασμα, ότι για μικρό αριθμό λειτουργιών σε σχέση με το σύνολο των διαφορετικών αντικειμένων (M=0.2N), το πρότυπο Visitor υπερέχει του Inheritance ακόμα και αν υφίσταται ανάγκη για ταυτόχρονη προσθήκη νέων τύπων κόμβων και λειτουργιών.

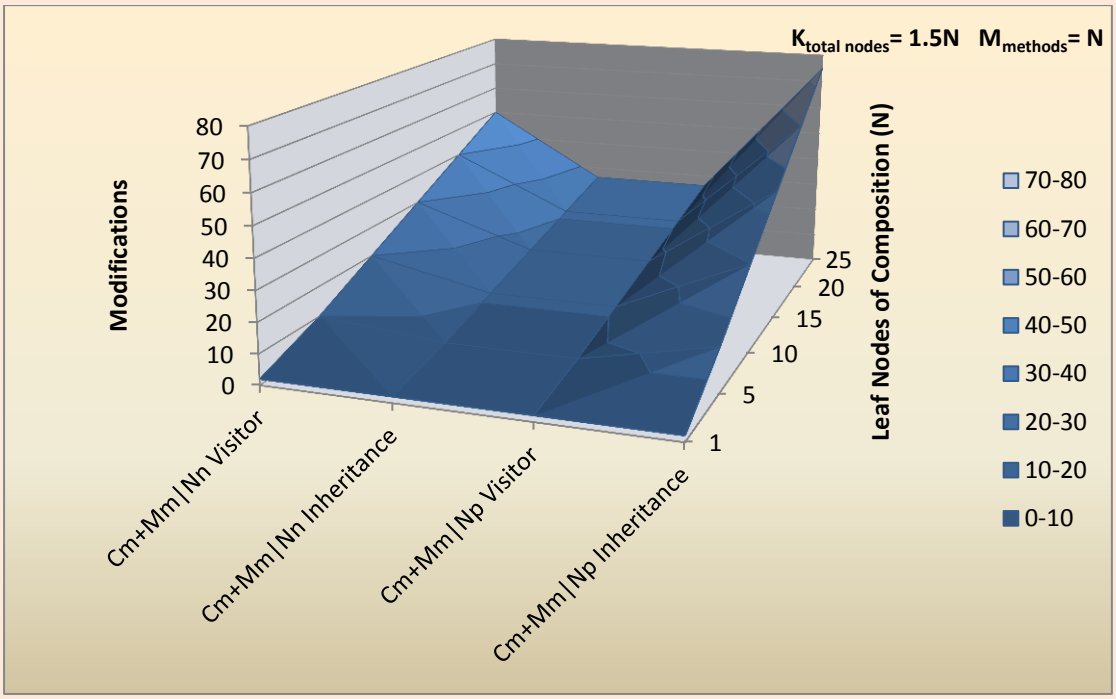
8.3.1.3 Γραφική αναπαράσταση (περίπτωση M=N)

Αντίστοιχα με την προηγούμενη υπό ενότητα, έστω τώρα μια γενική περίπτωση, όπου η ιεράρχηση της δομής είναι πάλι ένα δέντρο μέσης περίπτωσης, άρα για N διαφορετικά αντικείμενα (κλάσεις leafs) υπάρχουν **K=1.5N** συνολικές κλάσεις (inner nodes + leafs). Έστω όμως τώρα ότι δημιουργούμε μία νέα λειτουργία για κάθε διαφορετικό αντικείμενο, δηλαδή **M=N**. Για την συγκεκριμένη γενική περίπτωση οι ασυμπτωτικές εκτιμήσεις των παρεμβάσεων / αλλαγών που προκύπτουν ανά κριτήριο (Πίνακας 8-2) αποτυπώνονται στο παρακάτω γράφημα (Εικόνα 8-4).



Εικόνα 8-4 : Γράφημα ασυμπτωτικών εκτιμήσεων (M=N)

Στο ανωτέρω γράφημα αν για παράδειγμα για N=20, τότε K=30 και M=20. Προφανώς και το ζητούμενο είναι πάλι η κατά το δυνατό λιγότερες παρεμβάσεις και τροποποιήσεις, άρα τα βέλτιστα σημεία του γραφήματος εντοπίζονται στις χαμηλότερες ασυμπτωτικές τιμές ή αλλιώς στη χαμηλότερη στάθμη της επιφάνειας του γραφήματος.



Εικόνα 8-5 : Γράφημα ομάδων ασυμπτωτικών εκτιμήσεων (M=N)

Από το ανωτέρω γράφημα (Εικόνα 8-5), προκύπτει σαφώς ότι στην περίπτωση προσθήκης νέων διαδικασιών (Np), το πρότυπο Visitor επιφέρει λιγότερες τροποποιήσεις μεθόδων και κλάσεων (Cm+Mm) σε σχέση με το πρότυπο Inheritance. Από την άλλη στην περίπτωση προσθήκης νέων τύπων κόμβων (Nn) το πρότυπο Inheritance υπερέχει του Visitor. Αυτό οδηγεί στο γενικό συμπέρασμα, ότι για σημαντικό αριθμό λειτουργιών σε

σχέση με το σύνολο των διαφορετικών αντικειμένων (M=N), το πρότυπο Visitor υπερέχει του Inheritance κατά την προσθήκη νέων λειτουργιών και υπολείπεται κατά την προσθήκη νέων τύπων κόμβων.

Το τελευταίο συμπέρασμα προκύπτει και από την άμεση σύγκριση των μαθηματικών σχέσεων, από όπου η σχέση Nn (new node) του Inheritance είναι πάντα μικρότερη από την αντίστοιχη του Visitor και η σχέση Np (new process) του Visitor είναι πάντα μικρότερη από την αντίστοιχη του Inheritance.

- Cm+Mm|Nn Visitor **2M > M+1** Cm+Mm|**Nn Inheritance**
- Cm+Mm|**Np Visitor N+1 < 2K** Cm+Mm|Np Inheritance (K=κN, όπου κ=(1,…,2))

8.3.1.4 Γραφική αναπαράσταση (περίπτωση M=μN, K=κN)

Αν και τα αποτελέσματα των συγκρίσεων των προηγούμενων ενοτήτων είναι αρκετά ασφαλή σχετικά με μία σταθερή δομή parse/abstract tree ενός μεταγλωττιστή, δημιουργείται το ερώτημα «στην περίπτωση που υπάρχει πιθανότητα τροποποιήσεων τόσο στη δομή όσο και στις λειτουργίες, πιο πρότυπο μεταξύ των Inheritance και Visitor προσφέρεται ?». Δηλαδή σε μια τέτοια περίπτωση, τι επιλέγουμε ?, χωρίς να προσφύγουμε σε προσεγγίσεις όπως η aspect-oriented programming (ενότητα 8.2).

Προκειμένου να αποτυπωθούν τα μεγέθη με βάση το σύνολο των τροποποιήσεων / προσθηκών νέου κόμβου (Nn) και νέας λειτουργίας (Np), ομαδοποιούμε τα επιμέρους κριτήρια ως εξής:

- **Cm+Mm|Nn+Np** = Class and Method modifications for New node and process = **Απαιτήσεις τροποποιήσεων / προσθηκών** κλάσεων και μεθόδων σε περίπτωση **προσθήκης νέου τύπου κόμβου και νέας λειτουργίας**

Οι σχέσεις παρουσιάζονται στον Πίνακας 8-2 από όπου και προκύπτει :

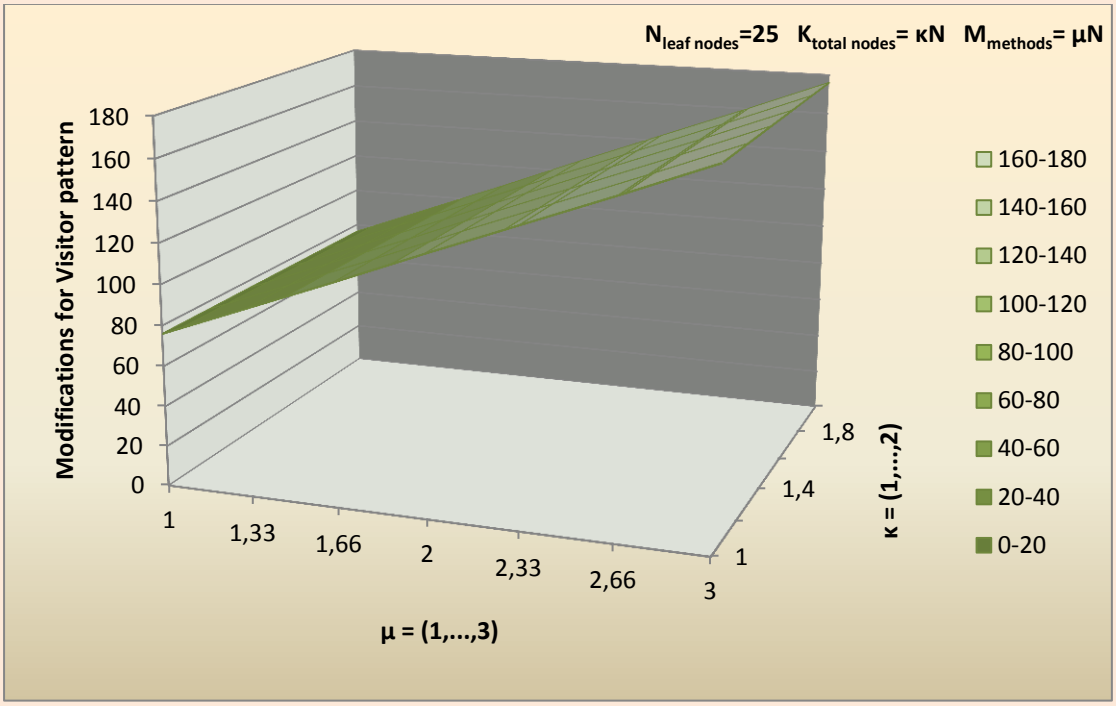
- Cm+Mm|Nn+Np Visitor = 2M+N+1 = **N(2μ+1)+1**
- Cm+Mm|Nn+Np Inheritance = M+1+2K = **N(μ+2κ)+1**
- leaf nodes = **N**
- leaf+ inner nodes **= K = κN** (κ=(1,…,2))
- process **M = μN** (μ=(0,…, +∞))
- possibility **p(Np) = p(Nn) = 0.5**

Οι ανωτέρω σχέσεις θεωρούν ότι η πιθανότητα τροποποίησης (προσθήκης) της δομής του δέντρου είναι ίδια με την πιθανότητα τροποποίησης (προσθήκης) των διαδικασιών. Διαπιστώνουμε ότι η σύγκριση μεταξύ των σχέσεων **N(2μ+1)+1** και **N(μ+2κ)+1** είναι ανεξάρτητη της τιμής του N, και αναγάγετε σε σύγκριση μεταξύ των σχέσεων **2μ+1** και **μ+2κ**, δηλαδή **μ** και **2κ-1** (κ=(1,…,2), μ=(0,…, +∞)). Τώρα με δεδομένα τα όρια τιμών που μπορεί να πάρει ο συντελεστής κ, έχουμε την παρακάτω ανάλυση τιμών:

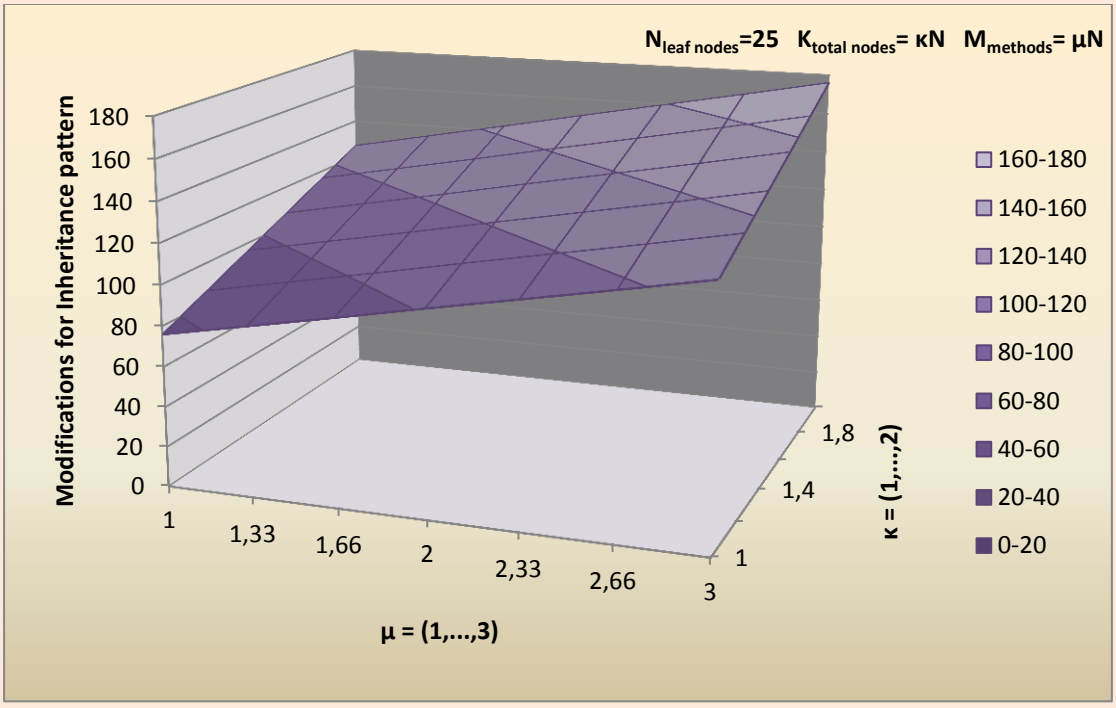
- **μ <= 1**, ισχύσει μ **<** 2k-1, άρα N(2μ+1)+1 **<** N(μ+2κ)+1, επιλογή **Visitor**
- **1<μ<3, μ < 2k-1**, ισχύει N(2μ+1)+1 **<** N(μ+2κ)+1, επιλογή **Visitor**
- **1<μ<3, μ = 2k-1**, ισχύει N(2μ+1)+1 **=** N(μ+2κ)+1, επιλογή **Visitor** ή **Inheritance**
- **1<μ<3, μ > 2k-1**, ισχύει N(2μ+1)+1 **>** N(μ+2κ)+1, επιλογή **Inheritance**

- **μ => 3**, ισχύσει μ > 2k-1, άρα N(2μ+1)+1 > N(μ+2κ)+1, επιλογή **Inheritance**

Από την ανάλυση τιμών προκύπτει ξεκάθαρη επιλογή για μ<=1 και μ>=3, ωστόσο για την κατανόηση της συμπεριφοράς των σχέσεων για 1<μ<3, παρατίθενται για (μια ανεξάρτητη) τιμή N=25, γραφήματα των ασυμπτωτικών εκτιμήσεων Cm+Mm|Nn+Np για Visitor (Εικόνα 8-6) και Inheritance (Εικόνα 8-7) αντίστοιχα.



Εικόνα 8-6 : Γράφημα ασυμπτωτικών εκτιμήσεων Nn+Np Visitor



Εικόνα 8-7 : Γράφημα ασυμπτωτικών εκτιμήσεων Nn+Np Inheritance

Πρακτικά μικρό μ, σημαίνει λιγότερες λειτουργίες και μεγάλο μ, περισσότερες λειτουργίες σε σχέση με τα διαφορετικά αντικείμενα (N) μίας ιεραρχίας κλάσεων (ή των

τερματικών κλάσεων, leaf classes). Επίσης μικρό k, σημαίνει λιγότερες ενδιάμεσες κλάσεις (inner classes) στην ιεραρχία, άρα μεγαλύτερο βαθμό συγκέντρωσης παραγόμενων κλάσεων ανά κλάση. Αντίστοιχα μεγάλο k, σημαίνει μικρότερο βαθμό συγκέντρωσης παραγόμενων κλάσεων ανά κλάση.

Συνεπώς για 1<μ<3, διαπιστώνεται,

- προτίμηση στο πρότυπο Visitor για λίγες λειτουργίες (μικρό μ) και μικρό βαθμό συγκέντρωσης (μεγάλο κ) παραγόμενων κλάσεων ανά κλάση,
- προτίμηση στο πρότυπο Inheritance για πολλές λειτουργίες (μεγάλο μ) και μεγάλο βαθμό συγκέντρωσης (μικρό κ) παραγόμενων κλάσεων ανά κλάση

Συμπερασματικά θα λέγαμε ότι, στην περίπτωση ισοπίθανης προσθήκης κόμβου και λειτουργίας, ξεκάθαρη επιλογή υπάρχει μόνο στις περιπτώσεις μ<=1, και μ=>3, επαληθεύοντας το συμπέρασμα της ενότητας 8.3.1.2 (για κ=0.2<1). Για τις περιπτώσεις 1<μ<3, η επιλογή είναι δυσδιάκριτη και εξαρτάται και από τον παράγοντα κ.

8.3.1.5 Ασυμπτωτικοί δείκτες σύγκρισης Inheritance – Visitor

Σύμφωνα με τα προηγούμενα, ο Πίνακας 8-1 διαμορφώνεται στον Πίνακας 8-2, ο οποίος περιέχει και τα νέα κριτήρια, ως συγχώνευση προηγουμένων.

Πίνακας 8-2 : Ασυμπτωτικοί δείκτες σύγκρισης Inheritance – Visitor

Δείκτες / Χαρακτηριστικά	Visitor	Inheritance
Cm|Nn Class modifications for New node	**M**	**1**
Mm|Nn Method modifications for New node	**M**	**M**
Cm|Np Class modifications for New process	**1**	**K**
Mm|Np Method modifications for New process	**N**	**K**
Cm+Mm|Nn Class and Method modifications for New node	**2M**	**M+1**
Cm+Mm|Np Class and Method modifications for New process K=κN (κ=(1,…,2))	**N+1**	**2K** or **2κN**
Cm+Mm|Nn+Np Class and Method modifications for New node and process K=κN (κ=(1,…,2)) M=μN (μ=(0,…, +∞)) possibility p(Nn) = p(Np) = 0.5	**2M+N+1** or **N(2μ+1)+1**	**M+1+2K** or **N(μ+2κ)+1**
Cm+Mm|(*p*)Nn+(*p*-1)Np Class and Method modifications for New (*p*) node and (1-*p*) process possibility p(Np) = 1- p(Nn)	**p2M+(1-p)(N+1)**	**p(M+1)+(1-p)2K**

Προχωρώντας την ανάλυση ένα βήμα παραπάνω, η τελευταία σειρά του πίνακα αναδιατυπώνει τη σχέση του δείκτη Cm+Mm|Nn+Np, ως συνάρτηση της πιθανότητας (p)

προσθήκης νέου κόμβου (έναντι πιθανότητας p-1 προθήκης νέας λειτουργίας). Έτσι, κατά την ανάλυση ενός προβλήματος με πιθανότητα μεταβολής δομής και λειτουργιών, και έχοντας μια εκτίμηση των σταθερών **N, μ, κ, p**, μπορούμε με βάση τον σχετικό πίνακα (Πίνακας 8-2), να κατανοήσουμε την απόδοση (επιπτώσεις) των σχετικών προτύπων σε μια ενδεχόμενη αλλαγή, στη βάση ασυμπτωτικών εκτιμήσεων.

8.3.2 Μέτρηση ποιότητας λογισμικού

Σε συνέχεια των αναφερομένων στο κεφάλαιο 3, τα χαρακτηριστικά και οι διαδικασίες που προσδιορίζουν την ποιότητα του λογισμικού περιγράφονται μέσω μιας σειράς προτύπων διασφάλισης – ελέγχου ποιότητας και μετρούνται μέσω μετρικών. Ωστόσο τα ποιοτικά χαρακτηριστικά που προκύπτουν από την εφαρμογή των ΟΟ προτύπων Composite, Visitor, Iterator, δεν είναι εύκολο να αποτυπωθούν μέσω αυτοματοποιημένων μετρικών. Από την άλλη ένα σύστημα το οποίο έχει σχεδιασθεί με βάση δοκιμασμένα και δομημένα σχεδιαστικά πρότυπα που εκμεταλλεύονται τα πλεονεκτήματα του ΟΟΡ, αναμένεται να παρουσιάζει βελτιωμένες επιδόσεις στην πλειονότητα των μετρικών ποιότητας.

Αναφερόμενοι στις μετρικές ποιότητας λογισμικού των προτύπων ISO (ενότητα 3.2.1) η μετρική «Change impact» (Πίνακας 3-2) που αναφέρεται στον αριθμό των δυσμενών επιπτώσεων μετά από αλλαγές σε σχέση με τον αριθμό των τροποποιήσεων που πραγματοποιήθηκαν, αναμένεται να έχει βελτιωμένη τιμή με την εφαρμογή του προτύπου visitor, κυρίως λόγω του χαρακτηριστικού του ελέγχου των επιμέρους υλοποιήσεων των ενεργειών ανά τύπο κόμβου από τη γλώσσα προγραμματισμού. Για παράδειγμα κατά την προσθήκη μιας νέας λειτουργίας σε μια δομή με τη δημιουργία νέου visitor, επιβάλλεται από τη γλώσσα η δημιουργία μεθόδων για κάθε ξεχωριστό τύπο κλάσης της δομής, μειώνοντας έτσι την πιθανότητα εμφάνισης των δυσμενών επιπτώσεων.

Αναφερόμενοι στις ποσοτικές μετρικές Halstead (ενότητα 3.2.2) και δεδομένου ότι στις διαδικασίες εφαρμογής των προτύπων iterators και visitors οι τελεστές (operators, n_1) και τα έντελα (operands, n_2) είναι περιορισμένα τόσο σε διακριτό αριθμό όσο και σε αριθμό εμφανίσεων (N_1, N_2), αναμένεται ότι όλες οι μετρικές (Πίνακας 3-3) που υπολογίζονται από τα ανωτέρω μεγέθη θα έχουν βελτιωμένες τιμές για τις συγκεκριμένες διαδικασίες εφαρμογής των προτύπων.

Τέλος αναφερόμενοι στις δομικές μετρικές McCabe (ενότητα 3.2.3) οι οποίες περιέχουν και ΟΟ μετρικές, αναμένεται ότι θα υπάρχει σχετική βελτίωση σε ένα σύνολο μετρικών (Πίνακας 3-4) όπως v(G), ev(G), pv(G), S0, OS1, gdv(G), PCTCALL, ROOTCNT, FANIN, FANOUT, QUAL, RFC, από την εφαρμογή των προτύπων composite, iterator, και visitor.

Γενικά θα λέγαμε ότι όσο αναφορά τον έλεγχο των ποιοτικών χαρακτηριστικών που προκύπτουν από την εφαρμογή των ΟΟ σχεδιαστικών προτύπων Façade, Composite, Visitor, Iterator αλλά και γενικά για οποιοδήποτε σχεδιαστικό πρότυπο, ισχύουν τα κάτωθι :

- ο ποιοτικός έλεγχος (αναφορικά με την εφαρμογή των σχεδιαστικών προτύπων) θα πρέπει να πραγματοποιείται κατά τη φάση της ανάπτυξης του μοντέλου σχεδίασης του λογισμικού και πριν την υλοποίησή του
- ο έλεγχος διενεργείται περισσότερο εμπειρικά από τον σχεδιαστή του συστήματος μέσα από τα ίδια τα διαγράμματα σχεδίασης (π.χ. διαγράμματα

κλάσεων, αντικειμένων, ακολουθίας) σε σχέση με τα πρότυπα διαγράμματα των αντίστοιχων σχεδιαστικών προτύπων

- η ίδια η προδιαγραφή των σχεδιαστικών προτύπων εγγυάται (σε μεγάλο βαθμό) την ποιότητα του παραγόμενου λογισμικού, τουλάχιστον για τα ποιοτικά χαρακτηριστικά και πλεονεκτήματα που το πρότυπο αναδεικνύει και για το λόγο αυτό προφανώς επιλέχθηκε
- η απόφαση εφαρμογής σχεδιαστικών προτύπων κατά τη φάση της υλοποίησης του λογισμικού, συνήθως συνεπάγεται την εκ βάθρων επανασχεδίαση του, γεγονός που επιφέρει σημαντική καθυστέρηση και σπατάλη πόρων
- λογισμικό το οποίο έχει σχεδιασθεί με βάση δοκιμασμένα και δομημένα σχεδιαστικά πρότυπα που εκμεταλλεύονται τα πλεονεκτήματα του OOP, αναμένεται να παρουσιάζει βελτιωμένες επιδόσεις στην πλειονότητα των μετρικών ποιότητας
- οι μετρικές ποιότητας λογισμικού είναι σε μεγάλο βαθμό υποκειμενικές και δεν είναι εύκολο να αποτυπώσουν άμεσα τα ζητούμενα ποιοτικά χαρακτηριστικά που προκύπτουν από την εφαρμογή συγκεκριμένων σχεδιαστικών προτύπων

Σκοπός των ΣΠ είναι η (δεδομένη) διασφάλιση παραγωγής ποιοτικού λογισμικού από την σχεδίαση του και όχι η εκ των υστέρων μέτρηση (επαλήθευση) της ποιότητας του.

8.4 Εκτέλεση διαπεράσεων απευθείας μέσω (recursive) Visitor

Προκειμένου η εφαρμογή των σχεδιαστικών προτύπων να είναι όσο το δυνατό πιο πλήρης θα αναφερθούμε συνοπτικά και σε μία ακόμη προσέγγιση εφαρμογής λειτουργιών μέσω visitor, όπου την ευθύνη των κλήσεων για την διαπέραση των κόμβων την έχει η ίδια η μέθοδος Accept_Visitor χωρίς τη χρήση iterators. Μία ανάλογη προσέγγιση παρουσιάζεται σε σχετική δημοσίευση (Norman, 2004), αναφορικά με την υλοποίηση ενός εκπαιδευτικού χαρακτήρα μεταγλωττιστή για προπτυχιακούς φοιτητές. Στη συνέχεια θα συγκρίνουμε τα πλεονεκτήματα και μειονεκτήματα της εν λόγω προσέγγισης με αυτήν της υλοποίησης της εργασίας.

Στην περίπτωση αυτή η υλοποίηση της μεθόδου Accept_Visitor διαμορφώνεται ανάλογα με τον τύπο της διαπέρασης που πραγματοποιεί. Για pre order διαπέραση έχει την παρακάτω γενική μορφή :

```
void Accept_Visitor (CAST_Visitor *v) {
       v->Visit_CAST_additive_expression(this); // pre order εφαρμογή visitor
       for (int i=0; i < m_NumberOfDescendants ; i++)
              // εφαρμογή visitor στους sub-nodes
              m_Descendants[i]->Accept_Visitor(v);
}
```

Η ανωτέρω pre order υλοποίηση περιέχει αναδρομική κλήση της Accept_Visitor, και τερματίζει σε κάθε τερματικό κόμβο οι οποίοι δεν έχουν sub-nodes. Διενεργεί ουσιαστικά μία top-down διαπέραση. Αντίστοιχα για post order διαπέραση έχει την παρακάτω γενική μορφή :

```
void Accept_Visitor (CAST_Visitor *v) {
       for (int i=0; i < m_NumberOfDescendants ; i++)
              // εφαρμογή visitor στους sub-nodes
```

```
            m_Descendants[i]->Accept_Visitor(v);
      v->Visit_CAST_additive_expression(this); //post order εφαρμογή visitor
}
```

Η ανωτέρω post order υλοποίηση διενεργεί ουσιαστικά μια bottom-up διαπέραση και μπορεί να χρησιμοποιηθεί για την εφαρμογή μίας SDD τύπου S-Attributed Definition. Για pre post order διαπέραση η υλοποίηση διαφοροποιείται με την παρακάτω γενική μορφή :

```
void Accept_Visitor (CAST_Visitor *v) {
      v->Visit_Pre_CAST_additive_expression(this); //post εφαρμογή visitor
      m_Descendants[1]->Accept_Visitor(v);
      v->Visit_Mid_CAST_additive_expression(this); //mid εφαρμογή visitor
      m_Descendants[3]->Accept_Visitor(v);
      v->Visit_Post_CAST_additive_expression(this); //post εφαρμογή visitor
}
```

Η ανωτέρω pre post order υλοποίηση διενεργεί ουσιαστικά μια μικτή bottom-up και top-down διαπέραση, με αριστερά προς δεξιά κλήση των sub-nodes και ενδιάμεσες επισκέψεις στον κύριο κόμβο και μπορεί να χρησιμοποιηθεί για την εφαρμογή μίας SDD τύπου L-Attributed Definition.

Η έναρξη της εφαρμογής ενός visitor μπορεί να γίνει με απλή κλήση της Accept_Visitor στον κόμβο (ρίζα) από τον οποίο και θέλουμε να εκκινήσει η διαπέραση.

```
void Postorder_CAST_Visitor_Application (CAST_SyntaxElement *_root) {
      TypeChecking_CAST_Visitor *tcastv =
                                    new TypeChecking_CAST_Visitor();

      _root->Accept_Visitor(tcastv); // post εφαρμογή visitor στην ρίζα
}
```

Σε σύγκριση με την υλοποίηση της εργασίας παρατηρούμε ότι δεν χρησιμοποιούνται iterators και οι κλήσεις της εκάστοτε διαπέρασης πραγματοποιούνται με αναδρομικές κλήσεις από την ίδια τη μέθοδο Accept_Visitor προς τους sub-nodes. Στα πλεονεκτήματα της προσέγγισης καταγράφονται ότι :

- Δεν δημιουργούνται αντικείμενα iterator (πχ vector_iterator) για κάθε κόμβο (του τρέχοντος μονοπατιού της διαπέρασης)
- δεν δημιουργούμε αντικείμενο τύπου PreOrder_Iterator. Ωστόσο η στοίβα των iterators που περιλαμβάνεται στα αντικείμενα διαπεράσεων (Preorde, Postorder, Prepostorter iterator), στην ουσία δημιουργείται από τον run-time μηχανισμό αναδρομικών κλήσεων των μεθόδων Accept_Visitor της γλώσσας

Όμως η προσέγγιση αυτή έχει και αρκετά μειονεκτήματα, όπως :

- Για διαφορετικού τύπου διαπεράσεις απαιτείτε ο ορισμός διαφορετικών μεθόδων Accept_Visitor για κάθε τύπο κόμβου της composite δομής,
- Η υλοποίηση των μεθόδων Accept_Visitor «μολύνονται» από συγκεκριμένες λεπτομέρειες σχετικά με την διαπέραση των sub-nodes. Για την αποφυγή αυτών των λεπτομερειών μια λύση είναι η αντικατάσταση του βρόγχου for από ένα αντίστοιχο βρόγχο for με χρήση ενός vector_iterator (οπότε στην ουσία επανερχόμαστε στη χρήση των iterators)

- Στην περίπτωση της pre post order διαπέρασης ο αντίστοιχος visitor περιέχει μεθόδους για κάθε διαφορετικό κόμβο και για κάθε διαφορετική επίσκεψη στον κόμβο (πχ Visit_**Pre**_CAST_additive_expression), δημιουργώντας σύγχυση στον interface της abstract class του visitor σε σχέση με άλλους visitor που διαθέτουν μόνο μία μέθοδο για κάθε τύπο κόμβου. Για την αποφυγή των πολλαπλών μεθόδων ανά τύπο κόμβου μπορεί να χρησιμοποιηθεί το πρότυπο visitor της ενότητας 7.8.1.4 όπου διαμέσου του Visitor_Table μπορούμε να έχουμε μόνο μια μέθοδο ανά τύπο κόμβου, η οποία και να συμπεριφέρεται διαφορετικά σε κάθε επίσκεψη στον ίδιο κόμβο
- Ο έλεγχος της διαπέρασης εναπόκειται στο μηχανισμό αναδρομικών κλήσεων της Accept_Visitor χωρίς ο χρήστης να μπορεί να επέμβει ενδιάμεσα. Έτσι παράλληλες και εμβόλιμες διαπεράσεις (όπως αυτή της ενότητας 7.9) δεν μπορούν να υλοποιηθούν

Γενικά είναι δυνατό οι δύο προσεγγίσεις να συνδυαστούν ή/και συνυπάρξουν ανάλογα με τις απαιτήσεις και την πολυπλοκότητα του προς επίλυση προβλήματος. Ειδικά για την υλοποίηση μεταγλωττιστών διαφαίνεται ότι η προσέγγιση της εργασίας είναι πιο ευέλικτη και ισχυρή έστω και αν απαιτεί (σε ορισμένες περιπτώσεις) τη δημιουργία επιπλέον αντικειμένων. Επίσης είναι πιο προσκολλημένη στις σχεδιαστικές αρχές των προτύπων, καθιστώντας την εφαρμογή τους και την συντήρηση του κώδικα τυποποιημένη και άρα ευκολότερη.

8.5 Πίνακας εφαρμογής Σχεδιαστικών Προτύπων

Από την σχεδίαση και υλοποίηση του (μέρους) μεταγλωττιστή και τη σχετική έρευνα που πραγματοποιήθηκε στα πλαίσια της εργασίας, προέκυψαν μια σειρά από διαγράμματα σχεδίασης λογισμικού καθώς και ενδεικτικός κώδικας σχετικά με την εφαρμογή των σχεδιαστικών προτύπων Façade, Composite, Iterator και Visitor. Επιπλέον αποτυπώθηκαν προβληματισμοί και προτάθηκαν λύσεις (σχεδίασης και υλοποίησης) σε μια σειρά πρακτικών θεμάτων όπως α) υλοποίηση και εφαρμογή λειτουργιών τόσο στην γονική κλάση όσο και στις επιμέρους κλάσεις της ιεραρχίας (ενότητα 7.7.3.1), β) υλοποίηση και εφαρμογή λειτουργιών μιας L-Attributed Definition με Synthesized και Inherited attributes με χρήση κατάλληλων visitor δια μέσου μιας pre-post order διαπέρασης (ενότητα 7.8.1.1). Η εφαρμογή των σχεδιαστικών προτύπων επικεντρώθηκε σε δομημένες δενδροειδής δομές αναπαράστασης στοιχείων (πχ parse/abstract trees) και στον τρόπο διαπέρσης των στοιχείων των δομών, καθώς και στον τρόπο επίσκεψης ή εφαρμογής επιμέρους λειτουργιών σε αυτούς. Γενικά οι περισσότερες από τις επιμέρους λειτουργίες που επιτελούνται από τους μεταγλωττιστές βασίζονται στην διαχείριση και επεξεργασίες τέτοιων δομών. Βασικά στοιχεία που εμπλέκονται στη σχεδίαση και υλοποίηση των μεταγλωττιστών είναι οι γραφικές δομές αναπαράστασης (graphics IRs) και ειδικότερα τα parse/abstract trees, οι SDDs του attribute grammar framework και οι επιμέρους ιδιότητες για την κατά περίπτωση context-sensitive analysis.

Πίνακας 8-3: Πίνακας εφαρμογής Σχεδιαστικών Προτύπων

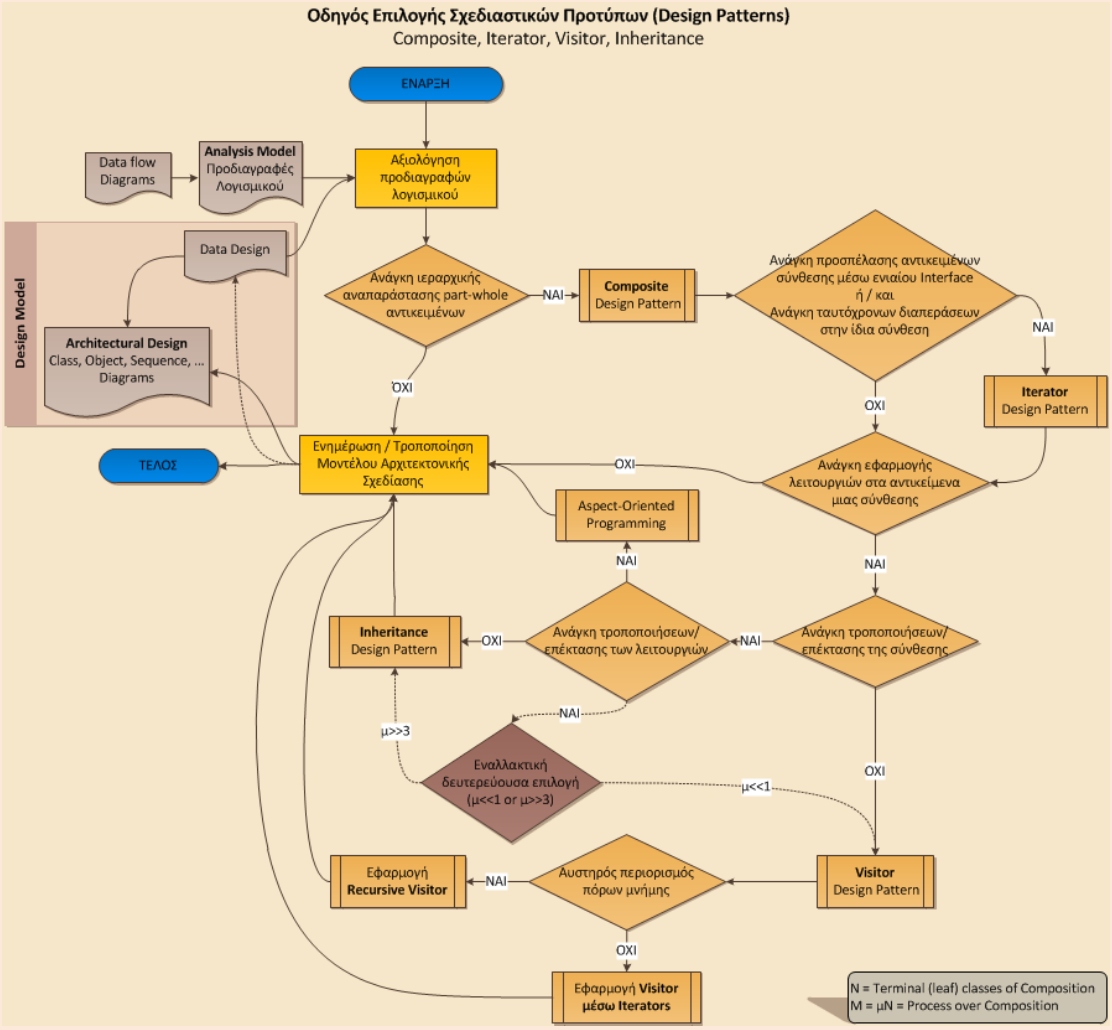
Είδος δομής / λειτουργιών **Design pattern**	**Τύπος ιδιοτήτων** (syntax directed definition, attribute grammar framework)	**Σχεδίαση** (ενότητα)	**Υλοποίηση** (ενότητα)	**Εφαρμογή** (ενότητα)
Σταθερή δομή, μεταβαλλόμενες λειτουργίες	Εφαρμογή ΟΟ σχεδιαστικών προτύπων (design patterns) Composite, Iterator, Visitor (ενότητες 4.3.1, 4.3.2, 4.3.3, σελίδα 46)			
	Single node (vector) iteration			
Iterator		7.5.2.1	7.5.2.2	7.5.2.3, 7.7.2.3
	Synthesized attributes, **S-Attributed definitions** (bottom-up, post-order iteration)			
Composite		7.5.1, 7.7.1, 7.10.2	---	---
Iterator		7.7.2.1	7.7.2.2	7.7.2.4
Visitor		7.7.3.1, 7.8.2.1	---	7.7.3.4, 7.8.2.2
	Inherited attributes (top-down, pre-order iteration)			
Composite		7.5.1, 7.7.1, 7.10.2	---	---
Iterator		7.5.2.1	7.5.2.2	7.5.2.4
Visitor		7.5.3.1, 7.7.3.1	---	7.5.3.3, 7.7.3.3
	Synthesized και **Inherited** attributes, **L-Attributed definitions** (pre-post-order iteration)			
Composite		7.5.1, 7.7.1, 7.10.2	---	---
Iterator		7.8.1.1	7.8.1.2	7.8.1.3
Visitor		7.8.1.4	7.8.1.5	7.8.1.6
Μεταβαλλόμενη δομή, σταθερές λειτουργίες	Εφαρμογή ΟΟ σχεδιαστικού προτύπου Inheritance (ενότητα 8.1.1, σελίδα 135)			
Μεταβαλλόμενη δομή, μεταβαλλόμενες λειτουργίες	Εφαρμογή Aspect-Oriented Programming (ενότητα 8.2, σελίδα 137) *Εναλλακτική εφαρμογή Visitor ή Inheritance (υπό συγκεκριμένες συνθήκες, ενότητα 8.3.1.4, σελίδα 143)*			

Ο Πίνακας 8-3, παρουσιάζει μια συνεκτική ιεράρχηση των σχεδιαστικών προτύπων της εργασίας, ανάλογα με τον τύπο της δομής και τον τύπο των λειτουργιών καθώς και τον τύπο των ιδιοτήτων των εκάστοτε λειτουργιών. Ο πίνακας ανάλογα με την περίπτωση, παραπέμπει στις σχετικές ενότητες σχεδίασης, υλοποίησης και εφαρμογής του εκάστοτε σχεδιαστικού προτύπου και μπορεί να χρησιμοποιηθεί ως αναλυτικός οδηγός για την

σχεδίαση, εφαρμογή και υλοποίηση των σχετικών ΟΟ σχεδιαστικών προτύπων κατά περίπτωση.

8.6 Οδηγός επιλογής Σχεδιαστικών Προτύπων

Με βάση τα διαγράμματα, τις υλοποιήσεις, τα παραδείγματα εφαρμογής, τις συγκρίσεις μεταξύ εναλλακτικών σχεδιαστικών προτύπων (ενότητα 8.1), την αξιολόγηση των σχεδιαστικών προτύπων (ενότητα 8.3.1), τις ενδεικτικές μετρήσεις ποιότητας, τα επιμέρους συμπεράσματα και ευρήματα της έρευνας, καθώς και τον πίνακα εφαρμογής σχεδιαστικών προτύπων (ενότητα 8.6), συντάχθηκε ο παρακάτω διαγραμματικός Οδηγός Επιλογής Σχεδιαστικών Προτύπων (Εικόνα 8-8), αναφορικά με τα Composite, Iterator, Visitor, και Inheritance σχεδιαστικά πρότυπα.



Εικόνα 8-8 : Διαγραμματικός Οδηγός Επιλογής Σχεδιαστικών Προτύπων

9 Σύνοψη

Η ανάπτυξη ενός πλήρη μεταγλωττιστή αποτελεί ένα από τα πιο εκτενή και περίπλοκα πεδία της αρχιτεκτονικής λογισμικού, τόσο στο στάδιο του σχεδιασμού όσο και της υλοποίησης. Απαιτούνται δεξιότητες και γνώσεις από όλα σχεδόν τα επιμέρους πεδία της επιστήμης των υπολογιστών τόσο από την πλευρά του λογισμικού όσο και από την πλευρά του υλικού. Η υλοποίηση του μεταγλωττιστή είναι κατά γενική παραδοχή ένα πολύ δύσκολο και περίπλοκο πρόβλημα, και για το λόγο αυτό επιλέχθηκε ως υλοποίηση στην παρούσα εργασία, ακριβώς γιατί η εφαρμογή σχεδιαστικών προτύπων (design patterns) αναμένεται να έχει σημαντική επίδραση στη βελτίωση της ποιότητα του παραγόμενου λογισμικού σε δύσκολα προβλήματα. Παράλληλα η υλοποίηση ενός μεταγλωττιστή αποτελεί πάντα μια πρόκληση για τον σχεδιαστή – προγραμματιστή, δεδομένης της περιπλοκότητας του. Επίσης διαπιστώνεται ότι παρά την πληθώρα βιβλιογραφίας σχετικά με τεχνικές, δομές και αλγόριθμους, εντούτοις, είναι πολύ δύσκολη η εύρεση των ίδιων των υλοποιήσεων (αναλυτικά σχεδιαστικά μοντέλα), κυρίως ίσως για εμπορικούς λόγους. Πόσο μάλλον οι υλοποιήσεις αυτές να είναι ποιοτικές και κατανοητές από τρίτους.

Μελετώντας τη σχετική βιβλιογραφία, είναι δυνατό να εντοπιστούν κάποιες βασικές αρχές με κοινότυπες ή παρεμφερείς δομές αναπαράστασης και στάδια επεξεργασίας που χρησιμοποιούνται αρκετά συχνά (όπως οι Graphical IRs, ενότητα 5.6.1). Η σχεδίαση και υλοποίηση του front-end του μεταγλωττιστή μαζί με ένα ενδεικτικό υποσύνολο των κυριότερων δομών και λειτουργιών του επαρκεί, προκειμένου να εξάγουμε τα συμπεράσματά μας σχετικά με την αποτελεσματικότητα και τα οφέλη της εφαρμογής συγκριμένων αντικειμενοστραφών σχεδιαστικών προτύπων (design patterns) σε ένα δύσκολο πρόβλημα όπως είναι ένας μεταγλωττιστής. Άλλωστε, όπως έχει αναφερθεί και στους στόχους της εργασίας, δεν επικεντρωνόμαστε στην επί της ουσίας εξαντλητική ανάλυση, λύση και υλοποίηση των επιμέρους προβλημάτων και λειτουργιών ενός μεταγλωττιστή, αλλά στην εφαρμογή κατάλληλων σχεδιαστικών προτύπων που συνδράμουν θετικά στην υλοποίηση αυτών των λύσεων, ενισχύοντας παράλληλα τα ποιοτικά χαρακτηριστικά του παραγόμενου λογισμικού. Η εφαρμογή των σχεδιαστικών προτύπων επικεντρώθηκε σε δομημένες δενδροειδής δομές αναπαράστασης στοιχείων (πχ parse/abstract trees) και στον τρόπο διαπέρσης των στοιχείων των δομών, καθώς και στον τρόπο επίσκεψης ή εφαρμογής επιμέρους λειτουργιών σε αυτούς.

9.1 Ευρήματα

Από την σχεδίαση και υλοποίηση του (μέρους) μεταγλωττιστή και τη σχετική έρευνα που πραγματοποιήθηκε στα πλαίσια της εργασίας, προέκυψαν μια σειρά από διαγράμματα σχεδίασης λογισμικού καθώς και ενδεικτικός κώδικας σχετικά με την εφαρμογή των σχεδιαστικών προτύπων Façade, Composite, Iterator και Visitor. Επιπλέον αποτυπώθηκαν προβληματισμοί και προτάθηκαν λύσεις (σχεδίασης και υλοποίησης) σε μια σειρά πρακτικών θεμάτων. Παρακάτω διατυπώνονται συγκεντρωτικά τα σημαντικότερα ευρήματα της έρευνας.

Το σχεδιαστικό πρότυπο Façade παρέχει ένα ενιαίο interface για ένα σύνολο επιμέρους interface ενός υποσυστήματος και η εφαρμογή του συμβάλει στη μείωση της πολυπλοκότητας της λύσης, στην επαναχρησιμοποίηση τμημάτων του λογισμικού καθώς

και στην ελαχιστοποίηση των αλληλεξαρτήσεων μεταξύ των υποσυστημάτων που ενοποιεί. Η χρήση του καθιστά πιο ξεκάθαρη την αλληλεπίδραση μεταξύ υποσυστημάτων, αποκρύπτοντας τις εσωτερικές διεργασίες των επιμέρους συστατικών τους και επηρεάζει θετικά τα αντίστοιχα ποιοτικά χαρακτηριστικά του παραγόμενου λογισμικού.

Το σχεδιαστικό πρότυπο Composite είναι μια σύνθεση αντικειμένων σε δομές δέντρων για την αναπαράσταση της part-whole ιεραρχίας ώστε να επιτρέπει τον χειρισμό ανεξάρτητων αντικειμένων και συνθέσεων με ενιαίο τρόπο. Η χρήση του στις εσωτερικές δομές αναπαράστασης ενός μεταγλωττιστή και ειδικά των parse και abstract trees κρίνεται απαραίτητη ανεξάρτητα από την χρήση των σχεδιαστικών προτύπων Iterator και Visitor.

Το σχεδιαστικό πρότυπο Iterator παρέχει ένα τρόπο πρόσβασης (διαπέρασης) στα στοιχεία μίας σύνθεσης αντικειμένων χωρίς να αποκαλύπτει στον χρήση τον τρόπο που αυτά αναπαρίστανται (οργανώνονται). Η χρήση του προτύπου τόσο στους μεταγλωττιστές όσο και σε αντίστοιχα ή παρόμοια προβλήματα κρίνεται ιδιαιτέρως ωφέλιμη ειδικά σε συνδυασμό με το πρότυπο Composite. Αποκρύπτει τον τρόπο διαπέρασης για διαφορετικούς τύπους αντικειμένων από τον χρήστη, παρέχοντας ένα τυποποιημένο και ενιαίο interface. Μάλιστα η σχεδίαση που παρουσιάζεται στην εργασία δίνεται μέσω ενός template για άμεση χρήση με οποιονδήποτε τύπο αντικειμένου. Επίσης παρουσιάζεται η υλοποίηση μιας σειράς από iterators για την πραγματοποίηση γνωστών τύπων διαπεράσεων δενδροειδών δομών δεδομένων, όπως pre order και post order διαπέραση. Επιπρόσθετα παρουσιάζεται η υλοποίηση μίας μεικτής pre post order διαπέρασης (ενότητα 7.8.1.2) για την εφαρμογή λειτουργιών L-Attributed Definitions, με βάση το attribute grammar framework και τις syntax directed definitions, στα πλαίσια της context sensitive analysis των μεταγλωττιστών. Γενικά τόσο η σχεδίαση των templates όσο και η υλοποίηση των Iterator κλάσεων που παρατίθενται στο κεφάλαιο 7, μπορούν να χρησιμοποιηθούν αυτούσια σε παρόμοια προβλήματα και γενικά για την διαπέραση δομών αντικειμένων τύπου composite.

Το σχεδιαστικό πρότυπο Visitor αναπαριστά μία λειτουργία που επιδρά στα στοιχεία μίας δομής αντικειμένων επιτρέποντάς μας να τροποποιούμε ή να ορίζουμε μια νέα λειτουργία χωρίς να τροποποιούμε την κλάση των αντικειμένων στα οποία αυτή επιδρά. Οι βασικές προϋποθέσεις για την επιλογή του προτύπου είναι όταν α) πολλές διακριτές και ανεξάρτητες λειτουργίες χρειάζεται να εφαρμοσθούν σε αντικείμενα μίας δομής, χωρίς την συμπερίληψή τους σε αυτές τις κλάσεις, β) οι κλάσεις που ορίζουν τα αντικείμενα της δομής σπάνια αλλάζουν, αλλά συχνά ορίζονται νέες λειτουργίες για τα στοιχειά αυτής της δομής.

Με βάση τις προϋποθέσεις και όπως προκύπτει από την έρευνα το πρότυπο Visitor προτιμάται για τους μεταγλωττιστές έναντι της εναλλακτικής σχεδιαστικής προσέγγισης Inheritance με αντίστροφες προϋποθέσεις επιλογής. Ο Πίνακας 8-1 (ενότητα 8.3.1) περιέχει ένα σύνολο ασυμπτωτικών εκτιμήσεων για επιμέρους σχεδιαστικά χαρακτηριστικά και ιδιοτήτων. Στους μεταγλωττιστές και ειδικά στην περίπτωση των parse/abstract trees, η δομή των οποίων βασίζεται στη γραμματική της γλώσσας εισόδου, εκτιμάται ότι είναι σπάνιο να αλλάξει η δομή της αναπαράστασης και περισσότερο πιθανό να τροποποιηθούν ή προστεθούν νέες λειτουργίες τόσο στο στάδιο της υλοποίησης όσο και της συντήρησης του μεταγλωττιστή. Συνεπώς το πρότυπο του Visitor προτιμάται στην σχεδίαση

μεταγλωττιστών έναντι του προτύπου Inheritance (ενότητα 8.1.1). Ειδικότερα, α) για μικρό αριθμό λειτουργιών σε σχέση με το σύνολο των διαφορετικών αντικειμένων, το πρότυπο Visitor υπερέχει του Inheritance ακόμα και αν υφίσταται ανάγκη για ταυτόχρονη προσθήκη νέων τύπων κόμβων και νέων λειτουργιών (ενότητα 8.3.1.2), β) για σημαντικό αριθμό λειτουργιών σε σχέση με το σύνολο των διαφορετικών αντικειμένων, το πρότυπο Visitor υπερέχει του Inheritance κατά την προσθήκη νέων λειτουργιών και υπολείπεται κατά την προσθήκη νέων τύπων κόμβων (ενότητα 8.3.1.3).

Επιπλέον το πρότυπο Visitor συνδυάζει και τα πλεονεκτήματα (ποιοτικά χαρακτηριστικά), α) του ελέγχου των επιμέρους υλοποιήσεων ενεργειών ανά τύπο κόμβου από τη γλώσσα προγραμματισμού, και β) της συγκέντρωσης των υλοποιήσεων λειτουργιών ανά κλάση λειτουργίας (visitor) και ανεξάρτητα της δομής που επιδρούν.

Η εφαρμογή των visitors στην υλοποίηση και τα παραδείγματα της εργασίας πραγματοποιείται μέσω των Iterators, ωστόσο στα πλαίσια της έρευνας (ενότητα 8.4) παρουσιάζεται και η εναλλακτική προσέγγιση απευθείας διαπέρασης μέσω visitors, τα πλεονεκτήματα και μειονεκτήματα, ομοιότητες και διαφορές καθώς και σύγκριση μεταξύ τους και οι λόγοι επιλογής της διαπέρασης μέσω iterators. Τέλος στην ενότητα 8.2, γίνεται μια αναφορά στην εναλλακτική προσέγγιση Aspect-Oriented Programming.

Κατά τη αρχιτεκτονική σχεδίαση της υλοποίησης της εργασίας προέκυψαν προβληματισμοί και προτάθηκαν σχεδιαστικές λύσεις σε θέματα όπως α) υλοποίηση και εφαρμογή λειτουργιών (μέσω Visitor) τόσο στην γονική κλάση όσο και στις επιμέρους κλάσεις της ιεραρχίας (ενότητα 7.7.3.1), β) υλοποίηση και εφαρμογή λειτουργιών μιας L-Attributed Definition με Synthesized και Inherited attributes με χρήση iterators και κατάλληλων visitors δια μέσου μιας pre-post order διαπέρασης (ενότητες 7.8.1.1, 7.8.1.4, 7.8.1.5, 7.8.1.6).

Επίσης από την έρευνα διαπιστώθηκε ότι τα ποιοτικά χαρακτηριστικά που προκύπτουν από την εφαρμογή των ΟΟ προτύπων Composite, Visitor, Iterator, δεν είναι εύκολο να αποτυπωθούν άμεσα μέσω αυτοματοποιημένων μετρικών.

Γενικά προκύπτει ότι στους μεταγλωττιστές παρατηρείται συχνά η ανάγκη εφαρμογής των σχεδιαστικών προτύπων composite, iterator και visitor σε αρκετές φάσεις μεταγλώττισης που έχουν να κάνουν κυρίως με την διαχείριση (εφαρμογή λειτουργιών) σε δομές / συλλογές δεδομένων γραφικής αναπαράστασης (graphics IRs).

9.2 Κατευθύνσεις για μελλοντική έρευνα

Με αφετηρία το υλικό της παρούσας εργασίας είναι δυνατή η περεταίρω διερεύνηση σε μια σειρά από επιμέρους θέματα όπως:

- εφαρμογή των προτύπων και των τεχνικών που παρουσιάζονται στην υλοποίηση του μεταγλωττιστή της εργασίας σε παρόμοια προβλήματα που ικανοποιούν τις προϋποθέσεις των συγκεκριμένων προτύπων
- περεταίρω βελτιστοποίηση, τροποποίηση η υιοθέτηση εναλλακτικών ή επιπρόσθετων σχεδιαστικών προτύπων, προκειμένου να παραχθεί λογισμικό με καλύτερα ποιοτικά χαρακτηριστικά, κυρίως αναφορικά με την εφαρμογή λειτουργιών σε δομές δεδομένων γραφικής αναπαράστασης

- κατασκευή εργαλείων για αυτόματη / τυποποιημένη παραγωγή κώδικα (π.χ. parse-tree) με βάση τα σχεδιαστικά πρότυπα της εργασίας

Αναλυτικότερα με βάση την σύγχρονη τάση για ανάπτυξη εργαλείων για αυτόματη, τυποποιημένη και γρήγορη παραγωγή έτοιμου κώδικα και δεδομένου ότι υπάρχουν σχετικά εργαλεία ήδη για γραμματικές compiler, υπάρχει ένα ανοιχτό πεδίο διερεύνησης σχετικά με την ανάπτυξη αντίστοιχων εργαλείων με βάση την αποκτηθείσα εμπειρία, το υλικό και τα αποτελέσματα της παρούσας εργασίας. Ενδεικτικά αναφέρεται η ανάπτυξη εργαλείου το οποίο με είσοδο ένα αρχείο δηλώσεων (κανόνες γραμματικής σε μορφή LR(1), συνοδευόμενοι με κατάλληλη notation για τον τύπο – κλάση και την ιεραρχία του κάθε κανόνα και token) να παράγει αρχεία *.h και *.cpp με ορισμούς βασικών α) κλάσεων Composition για κάθε κανόνα (με βάση την ορισμένη ομαδοποίηση και ιεραρχία) καθώς και για ομάδες tokens του parse/abstract tree, β) template κλάσεων Iterator για όλους τους τύπους διαπεράσεων, γ) πρότυπων κλάσεων Visitor, συμπεριλαμβανομένων των pre post order μεικτών διαπεράσεων, με interface μεθόδων για όλους τους επιμέρους τύπους αντικειμένων της composition και δ) ταυτόχρονα να παράγει αρχεία *.y και *.l για τον λεκτικό και συντακτικό αναλυτή Flex & Bison, όπου μαζί με τον ορισμό των tokens και της γραμματικής να ορίζονται, ως ενέργειες των κανόνων, κατάλληλες κλήσεις των constructors των κλάσεων της δομής για τη δημιουργία του parse/abstract tree. Στην ίδια κατεύθυνση και παράλληλα μπορεί να εξετασθεί και η παραγωγή έτοιμων C++ templates για εναλλακτικά ή επιπλέον σχεδιαστικά πρότυπα που εξυπηρετούν τη σχεδίαση και υλοποίηση μεταγλωττιστών, αντίστοιχα με την προσέγγιση «Design Patterns Automation with Template Library»[23] (Sergiu, Ning, & Narayan, 2005).

9.3 Συμπεράσματα

Η εφαρμογή δοκιμασμένων σχεδιαστικών προτύπων (design pattern) αποσκοπεί στην παραγωγή ποιοτικού λογισμικού που επικεντρώνεται κυρίως στα επιμέρους χαρακτηριστικά της επαναχρησιμοποίησης, της ευκολίας κατανόησης και συντήρησης του σχεδιαστικού μοντέλου και του κώδικα, τόσο από τον σχεδιαστή όσο και από τον προγραμματιστή, μη επηρεάζοντας την αποδοτικότητά του. Η επιλογή κατάλληλων και δοκιμασμένων σχεδιαστικών προτύπων αποτελεί μέρος του ευρύτερου πεδίου της Μηχανικής Λογισμικού και ειδικότερα της αρχιτεκτονικής σχεδίασης, ένας από τους σκοπούς της οποίας είναι η παραγωγή ποιοτικού λογισμικού. Διενεργείται κυρίως με βάση τις απαιτήσεις των προδιαγραφών (ποιότητα, ταχύτητα, μέγεθος, κλ.π) καθώς και την εμπειρία της ομάδας έργου και αποτυπώνεται στο μοντέλο αρχιτεκτονικής σχεδίασης του λογισμικού.

Ο ποιοτικός έλεγχος (αναφορικά με την εφαρμογή των σχεδιαστικών προτύπων) θα πρέπει να πραγματοποιείται κατά τη φάση της ανάπτυξης του μοντέλου σχεδίασης του λογισμικού και πριν την υλοποίησή του. Διενεργείται περισσότερο εμπειρικά από τον σχεδιαστή του συστήματος μέσα από τα ίδια τα διαγράμματα σχεδίασης (π.χ. διαγράμματα κλάσεων, αντικειμένων, ακολουθίας) σε σχέση με τα πρότυπα διαγράμματα των αντίστοιχων σχεδιαστικών προτύπων. Η ίδια η προδιαγραφή των σχεδιαστικών προτύπων

[23] Μέθοδος για την επίτευξη αυτοματοποιημένης παραγωγής σχεδιαστικών προτύπων (design patterns), δια μέσου της προσέγγισης Design Pattern Automation (DPA) για την εφαρμογή σχεδιαστικών προτύπων κατά το στάδιο της υλοποίησης του κύκλου ανάπτυξης λογισμικού

εγγυάται (σε μεγάλο βαθμό) την ποιότητα του παραγόμενου λογισμικού, τουλάχιστον για τα ποιοτικά χαρακτηριστικά και πλεονεκτήματα που το πρότυπο αναδεικνύει και για το λόγο αυτό προφανώς επιλέχθηκε. Λογισμικό το οποίο έχει σχεδιασθεί με βάση δοκιμασμένα και δομημένα σχεδιαστικά πρότυπα που εκμεταλλεύονται τα πλεονεκτήματα του OOP, αναμένεται να παρουσιάζει βελτιωμένες επιδόσεις στην πλειονότητα των μετρικών ποιότητας. Γενικά οι μετρικές ποιότητας λογισμικού είναι σε μεγάλο βαθμό υποκειμενικές και δεν είναι εύκολο να αποτυπώσουν άμεσα τα ζητούμενα ποιοτικά χαρακτηριστικά που προκύπτουν από την εφαρμογή σχεδιαστικών προτύπων. Σκοπός των σχεδιαστικών προτύπων, είναι η (δεδομένη) διασφάλιση παραγωγής ποιοτικού λογισμικού από την σχεδίαση του και όχι η εκ των υστέρων μέτρηση (επαλήθευση) της ποιότητας του.

Η υλοποίηση ενός μεταγλωττιστή συνιστά ένα δύσκολο πρόβλημα και πρέπει να δίνεται ιδιαίτερη βαρύτητα κατά την ανάλυση, σχεδίαση, και τεκμηρίωση του υπό ανάπτυξη λογισμικού (μοντελοποίηση με τυποποιημένα διαγράμματα UML), εφαρμόζοντας όπου είναι δυνατό κατάλληλα και δοκιμασμένα σχεδιαστικά πρότυπα.

Από τις επιμέρους υλοποιήσεις της εργασίας προκύπτει το συμπέρασμα ότι η εφαρμογή των σχεδιαστικών προτύπων composite και iterator έχουν αδιαμφισβήτητο όφελος δεδομένου ότι κάνουν χρήση των ΟΟ ιδιοτήτων της ιεραρχίας, της κληρονομικότητας και του πολυμορφισμού, προσφέροντας τυποποιημένα interface για την διαπέραση των αντικειμένων της δομής. Μάλιστα οι iterators δια μέσου του σχετικού template που παρουσιάσθηκε, έχουν ευρεία χρήση σε διαφορετικές δομές ανεξάρτητα από των τύπο και την ιεράρχηση των επιμέρους στοιχείων τους. Επίσης προκύπτει ότι η χρήση του προτύπου Visitor ενδείκνυται στους μεταγλωττιστές για την εφαρμογή λειτουργιών σε σύνθετες (Composite) δομές εσωτερικής αναπαράστασης (π.χ. Graphics IRs), όπου σε συνδυασμό με το πρότυπο Iterator, παρέχει στο χρήση ένα ισχυρό και πολύ απλό πλαίσιο για την εφαρμογή πολλαπλών λειτουργιών για όλους τους γνωστούς τύπους διαπεράσεων (συμπεριλαμβανομένης της pre post order για L-Attributed Definitions). Διαπιστώθηκε ότι ακόμη και μια γραμμική αναπαράσταση, όπως ο tree-address code, μπορεί και ενδείκνυται να σχεδιαστεί με τα ίδια πρότυπα.

Μπορούμε να πούμε ότι η παρούσα εργασία, ως αποτέλεσμα της σχετικής έρευνας, αποτελεί μια «Close to code» περιγραφή για τη σχεδίαση, προσαρμογή, υλοποίηση και εφαρμογή συγκεκριμένων σχεδιαστικών προτύπων σε μεταγλωττιστές, ως μία κάθετη προσέγγιση που εκκινεί από την μοντελοποιημένη αρχιτεκτονική σχεδίαση μέχρι και την ενδεικτική υλοποίηση κώδικα εφαρμογής τους. Ειδικότερα ο συγκεντρωτικός πίνακας (Πίνακας 8-3) καθώς και ο διαγραμματικός οδηγός εφαρμογής προτύπων της εργασίας (Εικόνα 8-8), μπορούν να χρησιμοποιηθούν συνδυαστικά ως αναλυτικός οδηγός για την σχεδίαση, εφαρμογή και υλοποίηση των σχεδιαστικών προτύπων Composite, Iterator και Visitor στους μεταγλωττιστές (συμπεριλαμβανομένων των παραλλαγών τους). Επίσης η παρούσα εργασία μπορεί να χρησιμοποιηθεί ως ένα κείμενο εναλλακτικής αναφοράς ή εκκίνησης για όσους επιθυμούν να ασχοληθούν με την κατανόηση και ανάπτυξη μεταγλωττιστών ή παρόμοιων προβλημάτων ή γενικά για την κατανόηση και εφαρμογή των γνωστών σχεδιαστικών προτύπων Façade, Composite, Iterator και Visitor.

Η παρούσα έρευνα αποτελεί κυρίως μία προσέγγιση από την οπτική της αρχιτεκτονικής σχεδίασης λογισμικού, μέρος της οποίας είναι η συνεχή προσπάθεια

βελτίωσης και εύρεσης νέων σχεδιαστικών προτύπων. Η επιτυχία και ευχρηστία των σχεδιαστικών προτύπων της εργασίας σε μεταγλωττιστές, θα αναδειχθεί μέσα από την εκτεταμένη χρήση τους.

Βιβλιογραφία / Αναφορές

Aho, A. V., & Ullman, J. D. (1995). *Foundations of Computer Science, C Edition.* W. H. Freeman.

Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2007). *Compilers Principles, Techniques, & Tools* (Second ed.). Pearson Education, Inc.

Bruno, C. O., Meng, W., & Jeremy, G. (2008, September 10). The visitor pattern as a reusable, generic, type-safe component. *ACM SIGPLAN Notices*, σσ. 439-456.

Cooper, K. D., & Torczon, L. (2012). *Engineering a Compiler* (Second ed.). Elsevier, Inc.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms (3d edition).* Massachusetts Institute of Technology.

Derek, M. J. (2005). The New C Standard (An Economic and Cultural Commentary).

Donnelly, C., & Stallman, R. (2008, 11 19). *Bison (the Yacc-compatible parser generator), version 2.4.1.* Ανάκτηση 03 2013, από GnuWin : Bison for Windows: http://gnuwin32.sourceforge.net/packages/bison.htm

Elrad , T., Filman, R. E., & Bader, A. (2001, 10). Aspect-oriented programming. *Communications of the ACM* , σσ. 29-32.

Feigenbaum, A. V. (1983). *Quality control (3d edition).* McGraw-Hill.

Filman, E. R., Tzilla, E., Siobhán, C., & Mehmet, A. (2004). *Aspect-Oriented Software Development.* Addison-Wesley Professional.

Gamma, E., Helm, R., Johnson, R., & Vlisssides, J. (1994). *Design Patterns : Elements of Reusable Object-Oriented Software.* Addison-Wesley.

Halstead, H. M. (1977). *Elements of Software Science.* Amsterdam: Elsevier North-Holland, Inc.

Herbert, S. (1998). *C++ : The Complete Reference (Third Edition).* Osborne McGraw-Hill.

Ho, S. (2000). *Practical UML:A Hands-On Introduction for Developers.* Ανάκτηση 03 2014, από Course Material from Dr. Sean Ho: http://twu.seanho.com/11spr/cmpt166/uml/

Intel Corporation. (2013, January). *Intel® 64 and IA-32 Architectures Software Developer's Manual (Combined Volumes: 1,2A,2B,2C,3A,3B,3C).* Retrieved April 2013, from Intel: http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

Irvine, K. R. (2011). *Assembly Language for x86 Processors* (6th ed.). Pearson Education, Inc.

ISO. (2002, 03). ISO/IEC 9126-3:2003 - Software engineering – Product quality – Part 3: Internal metrics.

ISO. (2012, 04). ISO/IEC 19505-1:2012 - Information technology - Object Management Group Unified Modeling Language (OMG UML), Infrastructure.

ISO. (2012, 04). ISO/IEC 19505-2:2012 - Information technology - Object Management Group Unified Modeling Language (OMG UML), Superstructure.

ISO. (2012, 11). ISO/IEC 25021:2012 - Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Quality measure elements.

ISO Organization. (2014). *Standards catalogue: ISO/IEC 14882:2011 Programming languages -- C++.* Ανάκτηση 03 2014, από ISO (International Organization for Standardization).

ISO Organization. (2014). *Standards catalogue: ISO/IEC 9899:1990 Programming languages -- C*. Ανάκτηση 3 11, 2014, από ISO (International Organization for Standardization): http://www.iso.org/iso/iso_catalogue/catalogue_ics/catalogue_detail_ics.htm?csnumber=17782

ISO Organization. (2014). *Standards catalogue: ISO/IEC 9899:2011 Programming languages -- C*. Ανάκτηση 3 10, 2014, από ISO (International Organization for Standardization): http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=57853

Kernighan, B. W., & Ritchie, D. M. (1988). *The C programming Language.* Prentice-Hall.

Khan, M. A. (2005). Introduction to UML.

Larman, C. (2004). *Appling UML and Patterns : An introduction to Object-Oriented Analysis and Design and Iterative Development (3rd edition).* Addison Wesley Professional.

Levine, J. R. (2009). *flex & bison.* O'Reilly Media, Inc.

McCabe Software. (2014). *McCabe: Software Metrics Glossary*. Ανάκτηση 3 28, 2014, από McCabe Software: http://www.mccabe.com/iq_research_metrics.htm

Norman, N. (2004, 03 1). Attribute Based Compiler Implemented Using Visitor Pattern. *ACM SIGCSE*, σσ. 130-134.

OMG. (2011, 08). OMG Unified Modeling LanguageTM (OMG UML), Infrastructure (version 2.4.1).

OMG. (n.d.). *Unified Modeling Language™ (UML®)*. Ανάκτηση 03 2014, από OMG (Object Management Group): http://www.omg.org/spec/UML/

Paxson, V. (1995, 03). *Flex (a fast scanner generator), version 2.5.* Ανάκτηση 08 2013, από GnwWin: Flex for Windows: http://gnuwin32.sourceforge.net/packages/flex.htm

Pressman, S. R. (2001). *Software Engineering : A practitioner's approach (5th edition).* McGraw-Hill.

Sergiu, D., Ning, H., & Narayan, D. (2005). Design patterns automation with template library. *Signal Processing and Information Technology, 2005. Proceedings of the Fifth IEEE International Symposium on* (σσ. 699-705). Athens: IEEE.

Sommerville, I. (2007). *Software Engineering (8th Edition).* Pearson Education.

Soulie, J. (2007, 06). *C++ Language Tutorial.* Ανάκτηση 2014, από cplusplus.com : http://www.cplusplus.com/doc/tutorial/

Stephen, C. N., & Emden, R. G. (2014, 2). *Cgraph Tutorial.* Ανάκτηση 3 2014, από Graphviz - Graph Visualization Software: http://www.graphviz.org/pdf/Agraph.pdf

TechstreetStore. (2014). *INCITS/ISO/IEC 9899 - Programming Languages - C*. Ανάκτηση 3 11, 2014, από TechstreetStore: http://www.techstreet.com/products/232462

Treichel, K. (2009, 01 18). *Tree Compiler-Compiler.* Ανάκτηση 04 2014, από GNU Operating System: http://www.gnu.org/software/dotgnu/treecc/treecc.html

Van Der Linden, P. (1994). *Expert C Programming : Deep C Secrets.* Sun Microsystems, Inc.

Watson, H. A., & McCabe, J. T. (1996, 09). *Structured Testing: A Testing Methodology, Using the Cyclomatic Complexity Metric.* Ανάκτηση 3 27, 2014, από McCabe Software: http://www.mccabe.com/pdf/mccabe-nist235r.pdf

Wikipedia. (2014, 3 26). *Halstead complexity measures*. Ανάκτηση 3 28, 2014, από Wikipedia: http://en.wikipedia.org/wiki/Halstead_Complexity_Measures

Wikipedia. (2014, 2 8). *Procedural programming*. Ανάκτηση 3 10, 2014, από Wikipedia: http://en.wikipedia.org/wiki/Procedural_programming#cite_ref-2

Ευρετήριο Όρων

10 Παράρτημα Ι : C89 – Flex & Bison Grammar

10.1 Flex Tokens Regular Expressions (REs)

```
BLANK                   [ \t]
DEC_DIGIT               [0-9]
DEC_DIGIT_NZ            [1-9]
OCTAL_DIGIT             [0-7]
LETTER_                 [a-zA-Z_]
HEX_DIGIT               [a-fA-F0-9]
EXP                     [Ee][+-]?{DEC_DIGIT}+
BIN_EXP                 [Pp][+-]?{DEC_DIGIT}+
FLOAT_SUFFIX            (f|F|l|L)
INT_SUFFIX              ((u|U)(l|L|ll|LL)?)|((l|L|ll|LL)(u|U)?)
SIMPLE_ESC_SEQ          ((\\')|(\\")|(\\?)|(\\\)|(\\n)|(\\a)|(\\b)
                        |(\\f)|(\\r)|(\\t)|(\\v)|(\\0)|(\\({LETTER_}|{DEC_DIGIT}))
OCTAL_ESC_SEQ           \\({OCTAL_DIGIT}){1,3}
HEX_ESC_SEQ             \\x({HEX_DIGIT}){1,4}
HEX_QUAD                ({HEX_DIGIT}){4}
U_CHAR                  ((\\u{HEX_QUAD})|(\\U{HEX_QUAD}{HEX_QUAD}))
```

```
({LETTER_}|{U_CHAR})({U_CHAR}|{LETTER_}|{DEC_DIGIT})* {  /* identifier or type name*/}

0[xX]{HEX_DIGIT}+{INT_SUFFIX}?              { /* hexadecimal integer constant */}

0{OCTAL_DIGIT}+{INT_SUFFIX}?                { /* octal integer constant */}

{DEC_DIGIT}+{INT_SUFFIX}?                   { /* decimal integer constant */}

{DEC_DIGIT}+{EXP}{FLOAT_SUFFIX}?            { /* integral floating point with exponent */}

{DEC_DIGIT}*"."{DEC_DIGIT}+({EXP})?{FLOAT_SUFFIX}?
{ /* floating with optional integral part and mandatory decimal part and optional exponent*/}

{DEC_DIGIT}+"."{DEC_DIGIT}*({EXP})?{FLOAT_SUFFIX}?
{/* floating with mandatory integral part and optional decimal part and optional exponent*/}

0[xX](({HEX_DIGIT}*"."{HEX_DIGIT}+)|({HEX_DIGIT}+(".")?)){BIN_EXP}{FLOAT_SUFFIX}?
                                            {/* hexadecimal floating */}

L?'({SIMPLE_ESC_SEQ}|{OCTAL_ESC_SEQ}|{HEX_ESC_SEQ}|{U_CHAR}|[^\\'\n]){1,4}'
                        { /* character constant - also supports wide characters */}

L?(\"({SIMPLE_ESC_SEQ}|{OCTAL_ESC_SEQ}|{HEX_ESC_SEQ}|{U_CHAR}|[^\\"\n])*\"){1}
                        {/* string literal - also supports wide characters */}
```

10.2 C89 Bison Token definitions

%nonassoc LOWER_THAN_ELSE	%token TILDE	%token MOD_ASSIGN	%token CONST
%nonassoc ELSE	%token EXCLAMATIONMARK	%token ADD_ASSIGN	%token VOLATILE
%token END	%token SLASH	%token SUB_ASSIGN	%token VOID
%token IDENTIFIER	%token PERCENT	%token LEFT_ASSIGN	%token STRUCT
%token CONSTANT	%token LANGLEBRACKET	%token RIGHT_ASSIGN	%token UNION
%token STRING_LITERAL	%token RANGLEBRACKET	%token AND_ASSIGN	%token ENUM
%token RBRACE	%token CARET	%token XOR_ASSIGN	%token ELLIPSIS
%token LBRACE	%token VERTICALBAR	%token OR_ASSIGN	%token RESTRICT
%token LPARENTHESIS	%token SEMICOLON	%token TYPE_NAME	%token INLINE
%token RPARENTHESIS	%token QUESTIONMARK	%token TYPEDEF	%token CASE
%token LBRACKET	%token EQUALSIGN	%token EXTERN	%token DEFAULT
%token RBRACKET	%token COLON	%token STATIC	%token IF
%token PERIOD	%token LEFT_OP	%token AUTO	%token SWITCH
%token PTR_OP	%token RIGHT_OP	%token REGISTER	%token WHILE
%token INC_OP	%token LE_OP	%token CHAR	%token DO
%token DEC_OP	%token GE_OP	%token SHORT	%token FOR
%token SIZEOF	%token EQ_OP	%token INT	%token GOTO
%token COMMA	%token NE_OP	%token LONG	%token CONTINUE
%token AMBERSAND	%token AND_OP	%token SIGNED	%token BREAK
%token ASTERISK	%token OR_OP	%token UNSIGNED	%token RETURN
%token PLUS	%token MUL_ASSIGN	%token FLOAT	
%token HYPHEN	%token DIV_ASSIGN	%token DOUBLE	

Εικόνα 10-1 : C89 – Bison Token definitions

10.3 C89 Bison LR(1) Grammar

primary_expression :IDENTIFIER
| CONSTANT | STRING_LITERAL
| LPARENTHESIS expression RPARENTHESIS

postfix_expression :primary_expression
| postfix_expression LBRACKET expression RBRACKET | postfix_expression LPARENTHESIS RPARENTHESIS
| postfix_expression LPARENTHESIS argument_expression_list RPARENTHESIS
| postfix_expression PERIOD IDENTIFIER | postfix_expression PTR_OP IDENTIFIER
| postfix_expression INC_OP | postfix_expression DEC_OP

argument_expression_list :assignment_expression
| argument_expression_list COMMA assignment_expression

unary_expression :postfix_expression
| INC_OP unary_expression | DEC_OP unary_expression
| unary_operator cast_expression | SIZEOF unary_expression
| SIZEOF LPARENTHESIS type_name RPARENTHESIS

unary_operator :AMBERSAND
| ASTERISK | PLUS
| HYPHEN | TILDE
| EXCLAMATIONMARK

cast_expression :unary_expression
| LPARENTHESIS type_name RPARENTHESIS cast_expression

multiplicative_expression :cast_expression | multiplicative_expression ASTERISK cast_expression
| multiplicative_expression SLASH cast_expression | multiplicative_expression PERCENT cast_expression

additive_expression :multiplicative_expression | additive_expression PLUS multiplicative_expression
| additive_expression HYPHEN multiplicative_expression

shift_expression :additive_expression | shift_expression LEFT_OP additive_expression
| shift_expression RIGHT_OP additive_expression

relational_expression :shift_expression
| relational_expression LANGLEBRACKET shift_expression | relational_expression RANGLEBRACKET shift_expression
| relational_expression LE_OP shift_expression | relational_expression GE_OP shift_expression

```
equality_expression :relational_expression
| equality_expression EQ_OP relational_expression          | equality_expression NE_OP relational_expression

and_expression :equality_expression                        | and_expression AMBERSAND equality_expression

exclusive_or_expression :and_expression                    | exclusive_or_expression CARET and_expression

inclusive_or_expression :exclusive_or_expression           | inclusive_or_expression VERTICALBAR exclusive_or_expression

logical_and_expression :inclusive_or_expression            | logical_and_expression AND_OP inclusive_or_expression

logical_or_expression :logical_and_expression              | logical_or_expression OR_OP logical_and_expression

conditional_expression :logical_or_expression
| logical_or_expression QUESTIONMARK expression COLON conditional_expression

assignment_expression :conditional_expression              | unary_expression assignment_operator assignment_expression

assignment_operator :EQUALSIGN
| MUL_ASSIGN                | DIV_ASSIGN                | MOD_ASSIGN
| ADD_ASSIGN                | SUB_ASSIGN                | LEFT_ASSIGN
| RIGHT_ASSIGN              | AND_ASSIGN                | XOR_ASSIGN
| OR_ASSIGN

expression :assignment_expression                          | expression COMMA assignment_expression

constant_expression :conditional_expression

declaration :declaration_specifiers SEMICOLON              | declaration_specifiers init_declarator_list SEMICOLON

declaration_specifiers :storage_class_specifier            | storage_class_specifier declaration_specifiers
| type_specifier                                           | type_specifier declaration_specifiers
| type_qualifier                                           | type_qualifier declaration_specifiers
| function_specifier                                       | function_specifier declaration_specifiers

init_declarator_list :init_declarator                      | init_declarator_list COMMA init_declarator

init_declarator :declarator                                | declarator EQUALSIGN initializer

storage_class_specifier :TYPEDEF
| EXTERN                   | STATIC
| AUTO                     | REGISTER

unction_specifier :INLINE

type_specifier :VOID
| CHAR                     | SHORT                     | INT
| LONG                     | FLOAT                     | DOUBLE
| SIGNED                   | UNSIGNED                  | struct_or_union_specifier
| enum_specifier           | TYPE_NAME

struct_or_union_specifier :struct_or_union IDENTIFIER LBRACE struct_declaration_list RBRACE
| struct_or_union LBRACE struct_declaration_list RBRACE    | struct_or_union IDENTIFIER
| struct_or_union IDENTIFIER LBRACE RBRACE

struct_or_union :STRUCT                                    | UNION

struct_declaration_list :struct_declaration                | struct_declaration_list struct_declaration

struct_declaration :specifier_qualifier_list struct_declarator_list SEMICOLON
| specifier_qualifier_list SEMICOLON

specifier_qualifier_list :type_specifier specifier_qualifier_list   | type_specifier
| type_qualifier specifier_qualifier_list                          | type_qualifier

struct_declarator_list :struct_declarator                  | struct_declarator_list COMMA struct_declarator

struct_declarator :declarator
| COLON constant_expression                                | declarator COLON constant_expression

enum_specifier :ENUM LBRACE enumerator_list RBRACE
| ENUM IDENTIFIER LBRACE enumerator_list RBRACE            | ENUM IDENTIFIER
```

```
enumerator_list :enumerator                                    | enumerator_list COMMA enumerator

enumerator :IDENTIFIER                                         | IDENTIFIER EQUALSIGN constant_expression

type_qualifier :CONST                  | VOLATILE              | RESTRICT

declarator :pointer direct_declarator                          | direct_declarator

direct_declarator :IDENTIFIER                                  | LPARENTHESIS declarator RPARENTHESIS
| direct_declarator LBRACKET constant_expression RBRACKET      | direct_declarator LBRACKET RBRACKET
| direct_declarator LPARENTHESIS parameter_type_list RPARENTHESIS
| direct_declarator LPARENTHESIS identifier_list RPARENTHESIS
| direct_declarator LPARENTHESIS RPARENTHESIS

pointer :ASTERISK                                              | ASTERISK type_qualifier_list
| ASTERISK pointer                                            | ASTERISK type_qualifier_list pointer

type_qualifier_list :type_qualifier                            | type_qualifier_list type_qualifier

parameter_type_list :parameter_list                            | parameter_list COMMA ELLIPSIS

parameter_list :parameter_declaration                          | parameter_list COMMA parameter_declaration

parameter_declaration :declaration_specifiers declarator
| declaration_specifiers abstract_declarator                   | declaration_specifiers

identifier_list :IDENTIFIER                                    | identifier_list COMMA IDENTIFIER

type_name :specifier_qualifier_list                            | specifier_qualifier_list abstract_declarator

abstract_declarator :pointer
| direct_abstract_declarator                                   | pointer direct_abstract_declarator

direct_abstract_declarator :LPARENTHESIS abstract_declarator RPARENTHESIS
| LBRACKET RBRACKET                                           | LBRACKET constant_expression RBRACKET
| direct_abstract_declarator LBRACKET RBRACKET
| direct_abstract_declarator LBRACKET constant_expression RBRACKET
| LPARENTHESIS RPARENTHESIS                                   | LPARENTHESIS parameter_type_list RPARENTHESIS
| direct_abstract_declarator LPARENTHESIS RPARENTHESIS
| direct_abstract_declarator LPARENTHESIS parameter_type_list RPARENTHESIS

initializer :assignment_expression
| LBRACE initializer_list RBRACE                             | LBRACE initializer_list COMMA RBRACE

initializer_list :initializer                                  | initializer_list COMMA initializer

statement :labeled_statement                                   | compound_statement
| expression_statement                                        | selection_statement
| iteration_statement                                         | jump_statement

labeled_statement :IDENTIFIER COLON statement
| CASE constant_expression COLON statement                    | DEFAULT COLON statement

left_brace_compound_statement :LBRACE

compound_statement :left_brace_compound_statement RBRACE
| left_brace_compound_statement statement_list RBRACE        | left_brace_compound_statement declaration_list RBRACE
| left_brace_compound_statement declaration_list statement_list RBRACE

declaration_list :declaration                                  | declaration_list declaration

statement_list :statement                                      | statement_list statement

expression_statement :SEMICOLON                                | expression SEMICOLON

selection_statement :IF LPARENTHESIS expression RPARENTHESIS statement %prec LOWER_THAN_ELSE
| IF LPARENTHESIS expression RPARENTHESIS statement ELSE statement
| SWITCH LPARENTHESIS expression RPARENTHESIS statement

statement_annotation :object_descriptor_list
```

object_descriptor_list :object_descriptor | object_descriptor_list object_descriptor

object_descriptor :CDB_FORLOOP COLON CDB_IDENTIFIER | CDB_ITERATOR COLON CDB_IDENTIFIER
| CDB_WHILELOOP COLON CDB_IDENTIFIER | CDB_DOWHILELOOP COLON CDB_IDENTIFIER
| CDB_ITERATIONS COLON CDB_IDENTIFIER

iteration_statement :WHILE LPARENTHESIS expression RPARENTHESIS statement
| DO statement WHILE LPARENTHESIS expression RPARENTHESIS SEMICOLON
| FOR LPARENTHESIS expression_statement expression_statement RPARENTHESIS statement
| FOR LPARENTHESIS expression_statement expression_statement expression RPARENTHESIS statement
| statement_annotation FOR LPARENTHESIS expression_statement expression_statement RPARENTHESIS statement
| statement_annotation FOR LPARENTHESIS expression_statement expression_statement expression RPARENTHESIS statement

jump_statement :GOTO IDENTIFIER SEMICOLON
| CONTINUE SEMICOLON | BREAK SEMICOLON
| RETURN SEMICOLON | RETURN expression SEMICOLON

translation_unit :external_declaration | translation_unit external_declaration

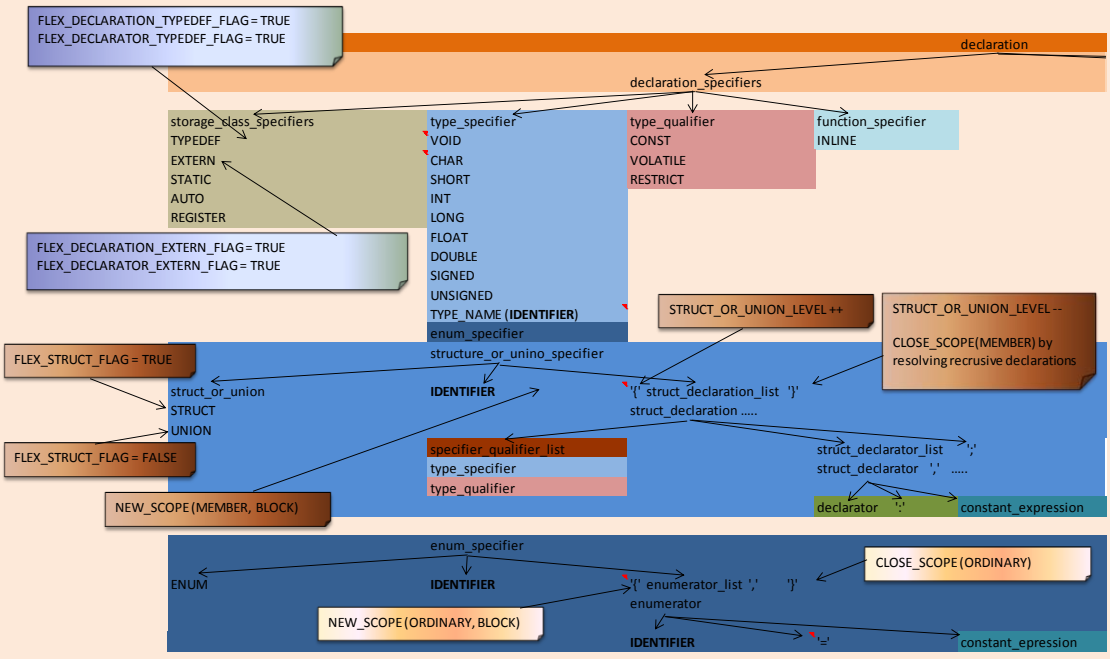
external_declaration :function_definition | declaration

function_definition :declaration_specifiers declarator declaration_list compound_statement
| declaration_specifiers declarator compound_statement | declarator declaration_list compound_statement
| declarator compound_statement

10.4 C89 Ad-Hoc Semantic analysis on Bison LR(1) Grammar

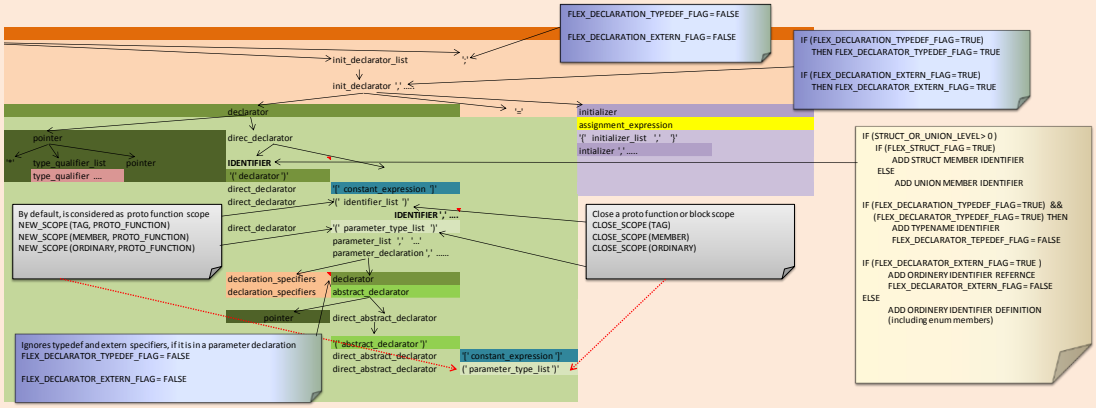
Στην παρούσα ενότητα παρουσιάζονται σχηματικές δενδροειδής αναπαραστάσεις της C89 LR(1) Grammar, με επισήμανση των κανόνων στους οποίους προσαρτώνται ενέργειες (κώδικας) για την πραγματοποίηση της σημασιολογικής ανάλυσης του scope checking των identifiers μέσω ενός symbol table.

10.4.1 Declaration specifier Rule



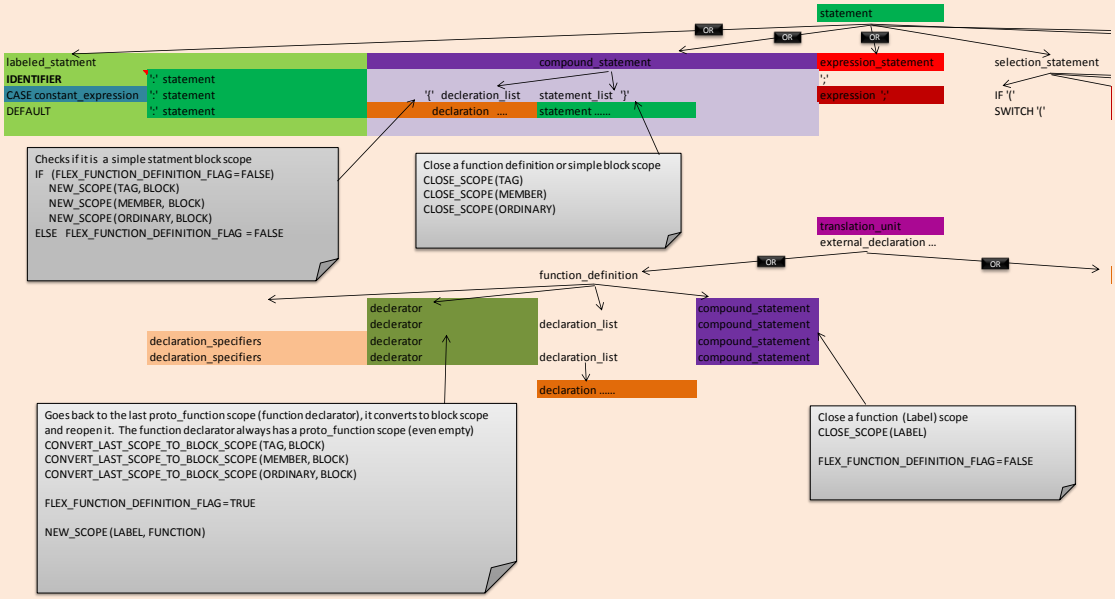
Εικόνα 10-2 : C89 ad-hoc semantic analysis on declaration_specifiers rule

10.4.2 Init declarator Rule



Εικόνα 10-3 : C89 ad-hoc semantic analysis on init_declarator rule

10.4.3 Statement & Translation unit Rules



Εικόνα 10-4 : C89 ad-hoc semantic analysis on statement and translation_unit rules

11 Παράρτημα II : C89 Identifiers

11.1 Name Spaces of Identifiers

C Identifier		Name Space			
class		Label names	Tags	Members (separate name space for each sructure or union)	Ordinary indetifiers
Object	Scalar types, Enumeration constants, etc, Function parameters				ordinary object declaretors
Function standard & prototype	(returned type) **name** (parameters) {...}				ordinary function declaretors
Structure Tag	Structure **name** {elements...}		by the keyword **struct**		
Union Tag	Union **name** {elements...}		by the keyword **union**		
Enumeration Tag	Enum **name** {enumeration constants...}		by the keyword **enum**		
Structure Member	Structure name {type **element...**}			by the operators **.** or **->**	
Union Member	Union name {type **element...**}			by the operators **.** or **->**	
Enumeration Member	Enum name {**enumeration constant...**}				enumeration constants
Typedef Name	Typedef type **name**				ordinary type declaretors
Label Name	**name**: C statement	by the syntax and use of label declaretions			

Εικόνα 11-1 : Name Spaces of Identifiers

11.2 Scopes of Identifiers

<table>
<tr><th>C Identifier</th><th></th><th colspan="4">Scope</th></tr>
<tr><th>class</th><th></th><th>Function</th><th>File</th><th>Block</th><th>Function prototype</th></tr>
<tr><td>Object</td><td>Scalar types, Enumeration constants, etc, Function parameters</td><td rowspan="9"></td><td colspan="3">καθορίζεται από την θέση της declaration
(μέσα σε έναν declarator ή type specifier)</td></tr>
<tr><td>Function
standard & prototype</td><td>(returned type) name (parameters) {...}</td><td>καθορίζεται από την σημασιολογική του εμφάνιση</td><td colspan="2"></td></tr>
<tr><td>Structure Tag</td><td>Structure name {elements...}</td><td colspan="3" rowspan="8">καθορίζεται από τη θέση της (declaration) δήλωσης
(μέσα σε έναν declarator ή type specifier)

1) αν ο declarator ή ο type specifier που δηλώνει τον identifier εμφανίζεται εκτός οποιουδήποτε block ή λίστας παραμέτρων (list of parameters), τότε ο identifier έχει file scope, ο οποίος και εκτείνεται μέχρι το τέλος του τμήματος κώδικα (translation unit)
2) αν ο declarator ή ο type specifier που δηλώνει τον identifier εμφανίζεται εντός ενός block η μέσα σε μία λίστα παραμέτρων (list of parameter) δηλώσεων εντός ενός ορισμού συνάρτησης (function definition), τότε ο identifier έχει block scope, ο οποίος και εκτείνεται μέχρι το τέλος του σχετικού block
3) αν ο declarator ή ο type specifier που δηλώνει τον identifier εμφανίζεται εντός μίας λίστας παραμέτρων (list of parameter) δηλώσεων μέσα σε μία πρωτότυπη δήλωση συνάρτησης (function prototype) (όχι ως μέρος ορισμού συνάρτησης - function definition), τότε ο identifier έχει function prototype scope, ο οποίος και εκτείνεται μέχρι το τέλος της δήλωσης της συνάρτησης (function declarator)

Αν ένας identifier προσδιορίζει δύο διαφορετικές οντότητες εντός του ιδίου name space, τότε οι scopes μπορεί να υπερκαλύπτονται. Εντός του εσωτερικού scope, ο identifier προσδιορίζει την οντότητα που δηλώθηκε (declared) εντός του εσωτερικού scope και η οντότητα που δηλώθηκε (declared) εντός του εξωτερικού scope είναι αφανής (και μη ορατή) στο εσωτερικό scope</td></tr>
<tr><td>Union Tag</td><td>Union name {elements...}</td></tr>
<tr><td>Enumeration Tag</td><td>Enum name {enumeration constants...}</td></tr>
<tr><td>Structure Member</td><td>Structure name {type element...}</td></tr>
<tr><td>Union Member</td><td>Union name {type element...}</td></tr>
<tr><td>Enumeration Member</td><td>Enum name {enumeration constant...}</td></tr>
<tr><td>Typedef Name</td><td>Typedef type name</td></tr>
<tr><td>Label Name</td><td>name: C statement</td><td>function scope
δηλώνεται εμμέσως από την σημασιολογική του εμφάνιση</td><td></td><td></td><td></td></tr>
</table>

Εικόνα 11-2 : Scopes of Identifiers

11.3 Linkage of Identifiers

C Identifier		Linkage		
class		External	Internal	None
Object	Scalar types, Enumeration constants, etc, Function parameters	with **extern** specifier	with **static** specifier	1. function parameters, 2. block scope objects without specifier **extern**
Function standard & prototype	(returned type) **name** (parameters) {...}	(default) with **extern** specifier	with **static** specifier	
Structure Tag	Structure **name** {elements...}			always
Union Tag	Union **name** {elements...}			always
Enumeration Tag	Enum **name** {enumeration constants...}			always
Structure Member	Structure name {type **element...**}			always
Union Member	Union name {type **element...**}			always
Enumeration Member	Enum name {**enumeration constant...**}			always
Typedef Name	Typedef type **name**			always
Label Name	**name**: C statement			always

Εικόνα 11-3 : Linkage of Identifiers

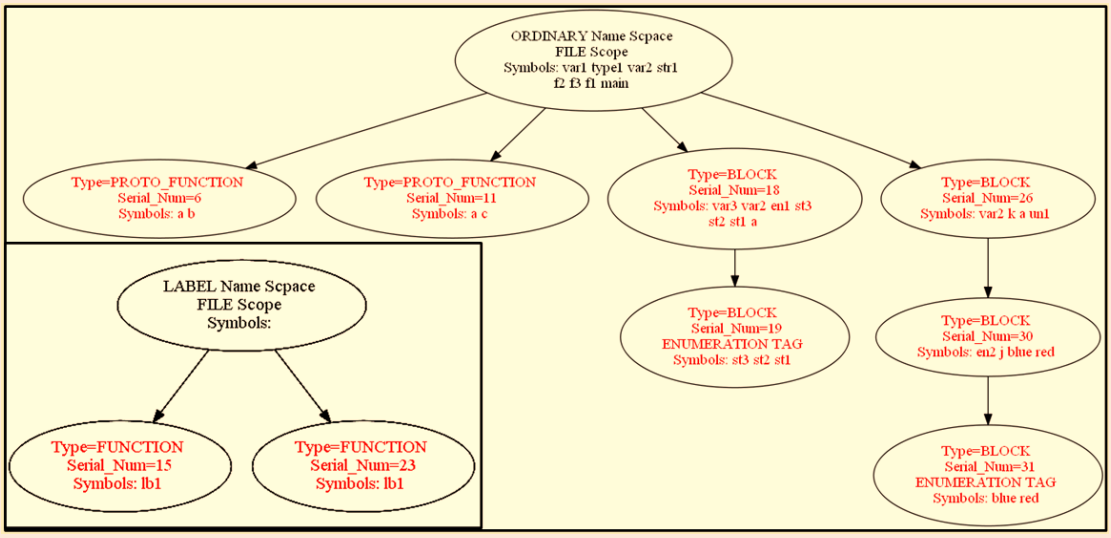
12 Παράρτημα III : Παραδείγματα

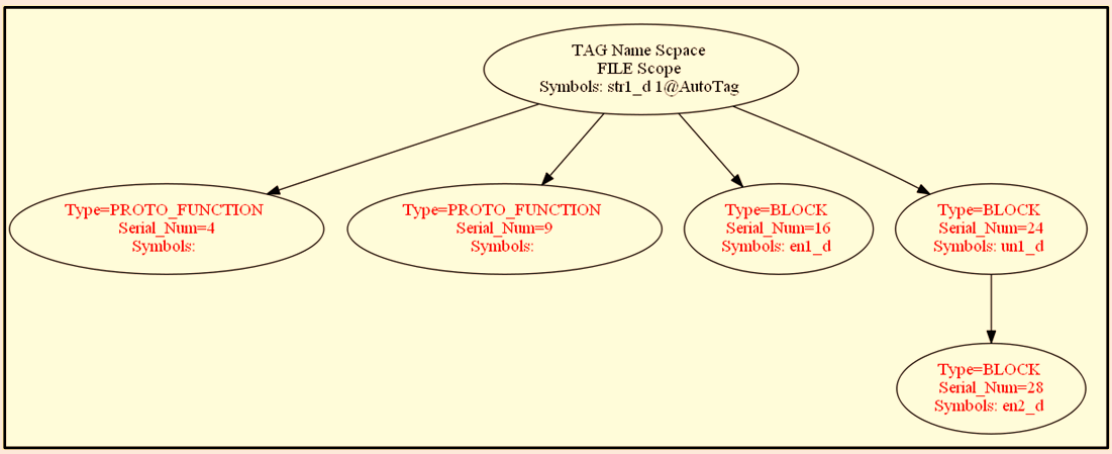
12.1 Παράδειγμα I: Γραφική αναπαράσταση Symbol Table

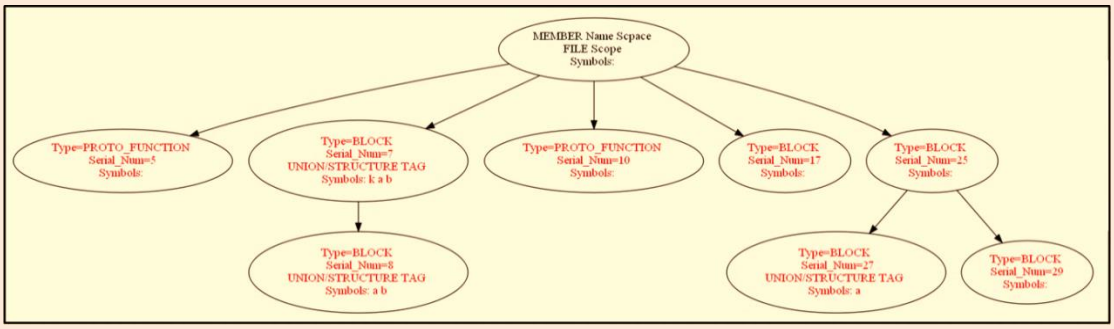
Για το παρακάτω ενδεικτικό παράδειγμα προγράμματος σε C89

```
int var1, var2, f1(int a, float b);
struct str1_d {
	int a;
	struct {
		float a;
		int b;
	} b;
	int k;
} str1;
void f2(int a, float c);
typedef struct str1 *type1;
void main() {
	int var2;
	float a;
	enum en1_d {st1=1, st2, st3} en1;
	type1 var3;
lb1:	f2(var2, a);
}
int f3(int var2, int a) {
	union un1_d {
		int a;
		struct str1;
	} un1;
	int k;
	{ int j;
		enum en2_d {red, blue} en2;
lb1:		j++;
	}
}
```

Προκύπτει η ακόλουθη γραφική αναπαράσταση των name spaces του πίνακα συμβόλων ανά scope, με βάση την σχεδιαστική προσέγγιση της εργασίας





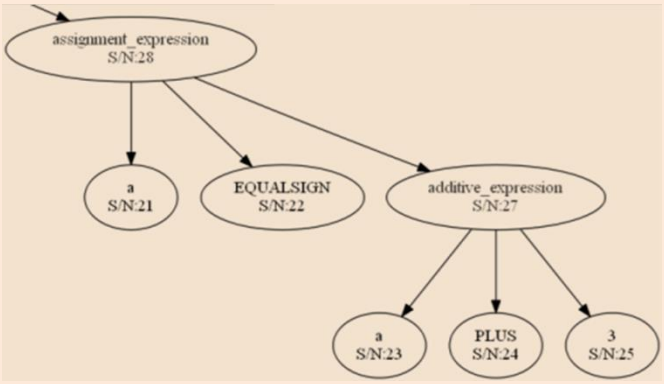


12.2 Παράδειγμα ΙΙ: Γραφική αναπαράσταση Parse / Abstract Tree

Για το παρακάτω ενδεικτικό παράδειγμα προγράμματος σε C89 και ειδικά για την έκφραση ανάθεσης

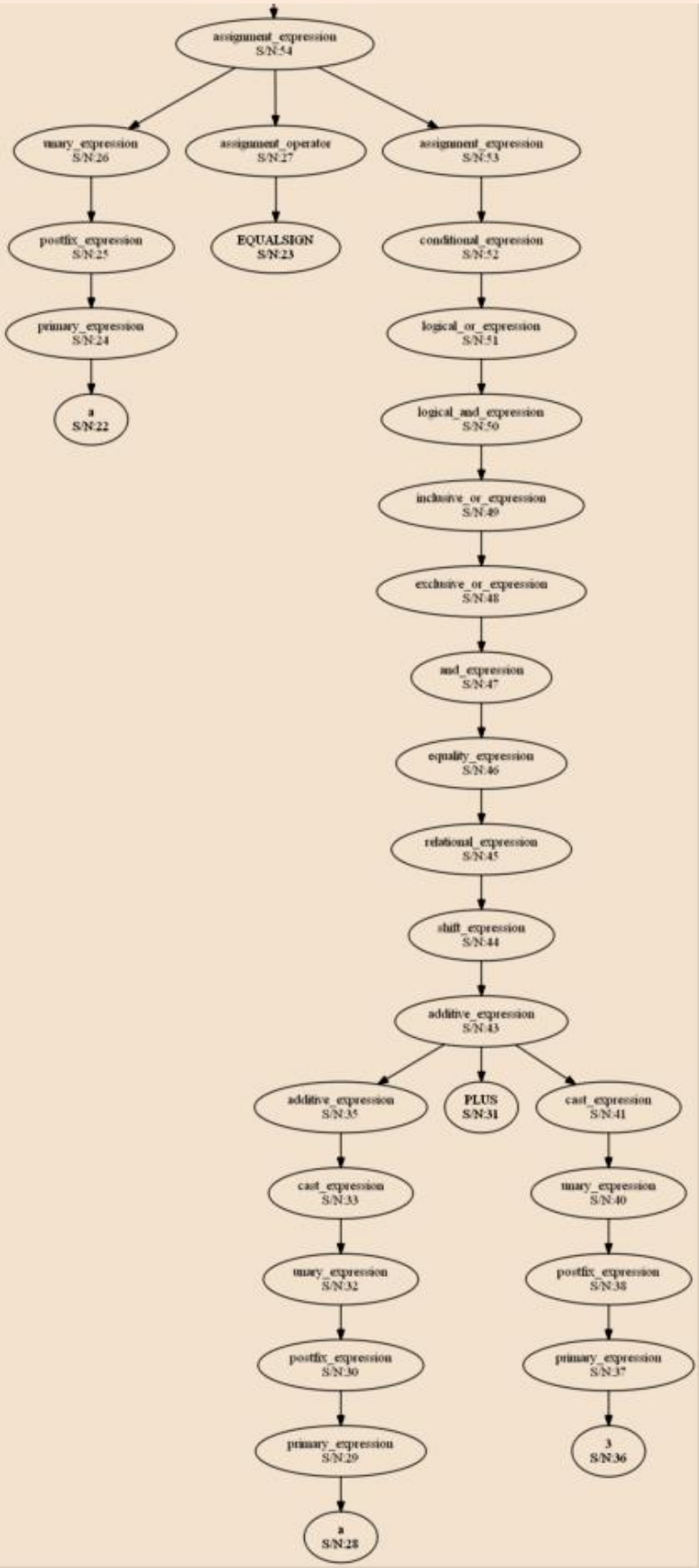
```
void main() {
      int a;
      a = a + 3 ;
}
```

Προκύπτει η ακόλουθη γραφική αναπαράσταση του **abstract tree**, με βάση την σχεδιαστική προσέγγιση της εργασίας



Εικόνα 12-1 : Γραφική αναπαράσταση Abstract-Tree της έκφρασης a=a+3

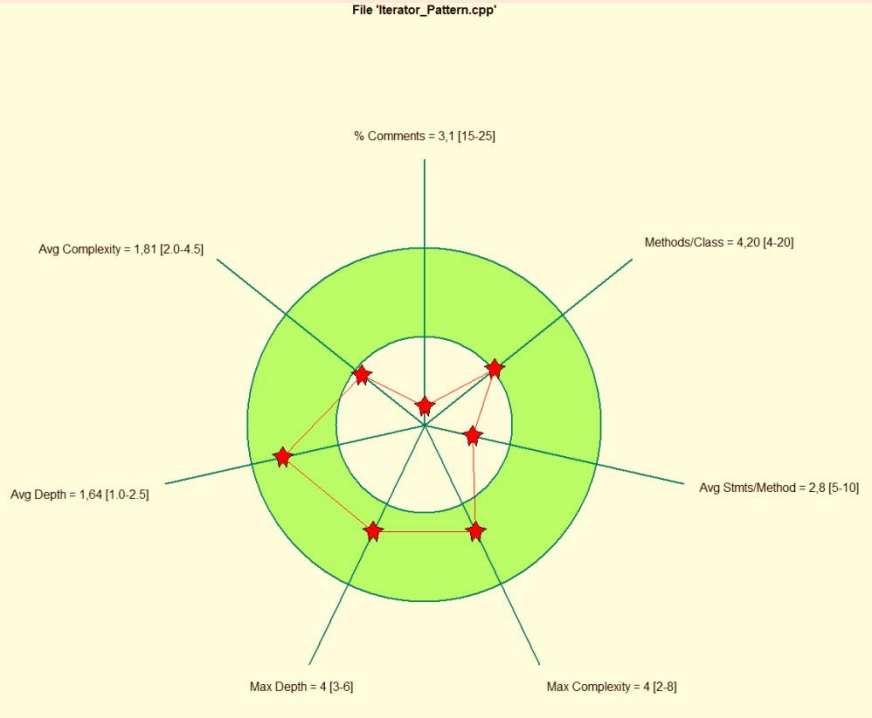
Επίσης προκύπτει η ακόλουθη γραφική αναπαράσταση του **parse tree**, με βάση την σχεδιαστική προσέγγιση της εργασίας



Εικόνα 12-2 : Γραφική αναπαράσταση Parse-Tree της έκφρασης a=a+3

13 Παράρτημα IV : Μετρήσεις ποιότητας λογισμικού

Στο παράρτημα παρουσιάζονται αναφορές μετρήσεων της υλοποίησης της εργασίας για τέσσερα ενδεικτικά module κώδικα, που έχουν παραχθεί με χρήση του ελεύθερου λογισμικού ελέγχου ποιότητας κώδικα SourceMonitor (έκδοση 3.4.6.297). Για κάθε module παρατίθεται ο γράφος μετρικών Kiviat καθώς και πίνακας με επιμέρους μετρήσεις ανά κλάση και μέθοδο. Οι μετρικές μέσης/μέγιστης κυκλωματικής πολυπλοκότητας, μέσου/μέγιστου βάθους, αριθμού μεθόδων ανά κλάση, μέσων δηλώσεων ανά κλάση, ποσοστών σχολείων κώδικα και αριθμού κλήσεων ανά μέθοδο, που εξάγονται είναι σχετικά απλές, αποτυπώνοντας βασικά ποιοτικά χαρακτηριστικά του κώδικα.

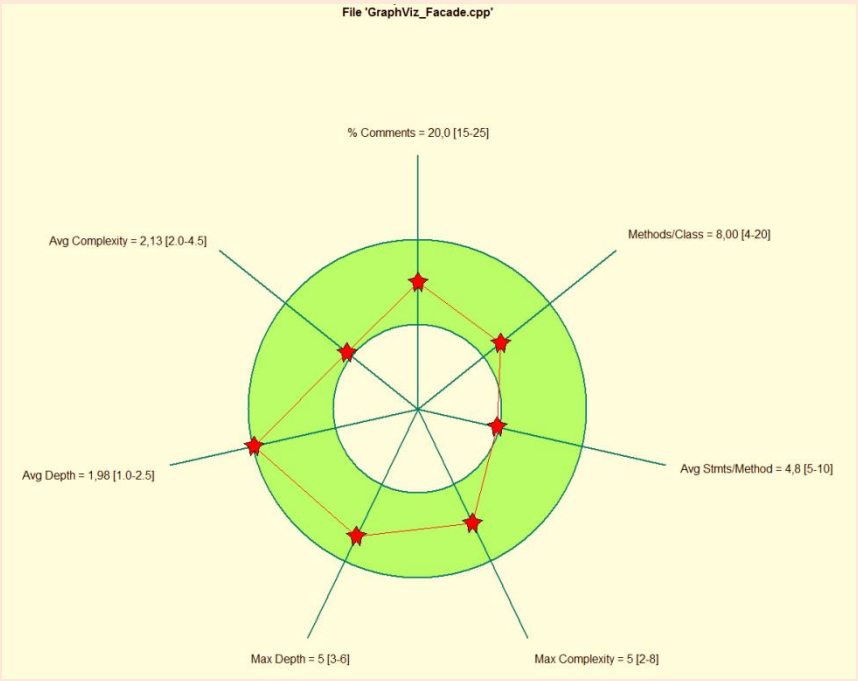


Εικόνα 13-1 : Γράφος μετρικών Kiviat του Iterator_Pattern.cpp module

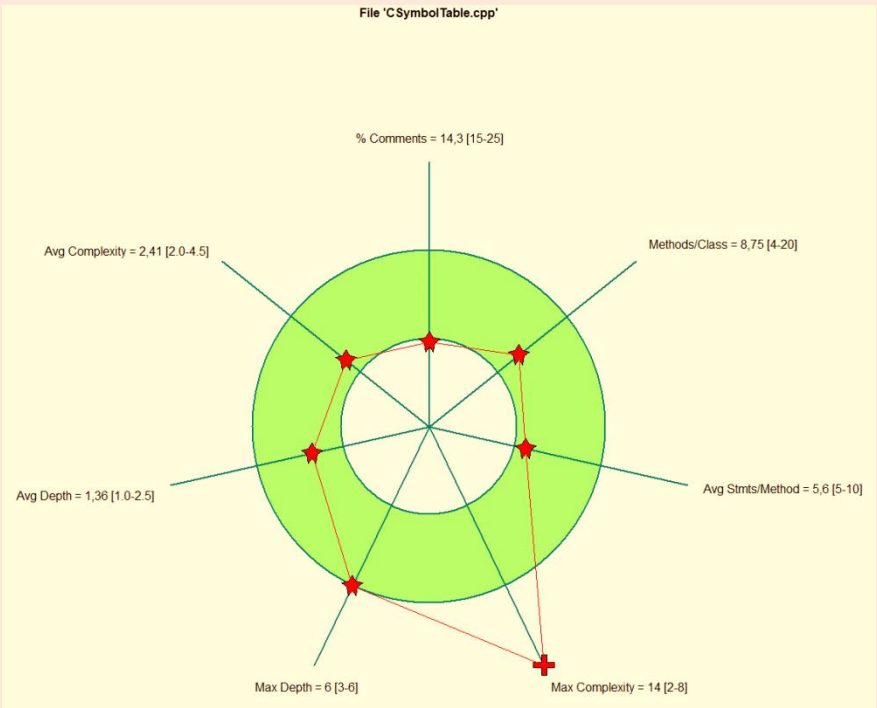
Πίνακας 13-1 : Πίνακας επιμέρους μετρικών του Iterator_Pattern.cpp module

Class	Method Name	Complexity	Statements	Maximum Depth	Calls
Iterator	Iterator()	1	0	0	0
Null_Iterator	Current_Item()	1	1	2	0
Null_Iterator	First()	1	0	0	0
Null_Iterator	Is_Done()	1	1	2	0
Null_Iterator	Next()	1	0	0	0
Null_Iterator	Null_Iterator()	1	0	0	0
Postorder_Iterator	Current_Item()	2	2	2	3
Postorder_Iterator	First()	3	8	4	8
Postorder_Iterator	Is_Done()	1	1	2	1
Postorder_Iterator	Next()	4	12	3	13
Postorder_Iterator	Postorder_Iterator()	1	2	2	0
Preorder_Iterator	Current_Item()	3	4	3	3
Preorder_Iterator	First()	2	5	3	4
Preorder_Iterator	Is_Done()	1	1	2	3
Preorder_Iterator	Next()	4	8	3	13
Preorder_Iterator	Preorder_Iterator()	1	2	2	0
Vector_Iterator	Current_Item()	3	4	3	2
Vector_Iterator	First()	1	1	2	0
Vector_Iterator	Is_Done()	4	4	3	2
Vector_Iterator	Next()	1	1	2	0
Vector_Iterator	Vector_Iterator()	1	2	2	0

Με βάση τα αποδεκτά τιθέμενα όρια ανά μετρική (πράσινη περιοχή), εξάγεται άμεσα η πληροφορία σχετικά με τα χαρακτηριστικά (μετρικές) που ξεφεύγουν των αναμενόμενων ποιοτικών χαρακτηριστικών του κώδικα. Περεταίρω από τους πίνακες των επιμέρους μετρικών είναι δυνατός ο εντοπισμός των μεθόδων (διαδικασιών) οι οποίες και χρειάζονται βελτίωση (π.χ. διάσπαση μίας μεθόδου σε επιμέρους για τη μείωση του βάθους ή/και της κυκλωματικής πολυπλοκότητας).



Εικόνα 13-2 : Γράφος μετρικών Kiviat του GraphViz_Facade.cpp module



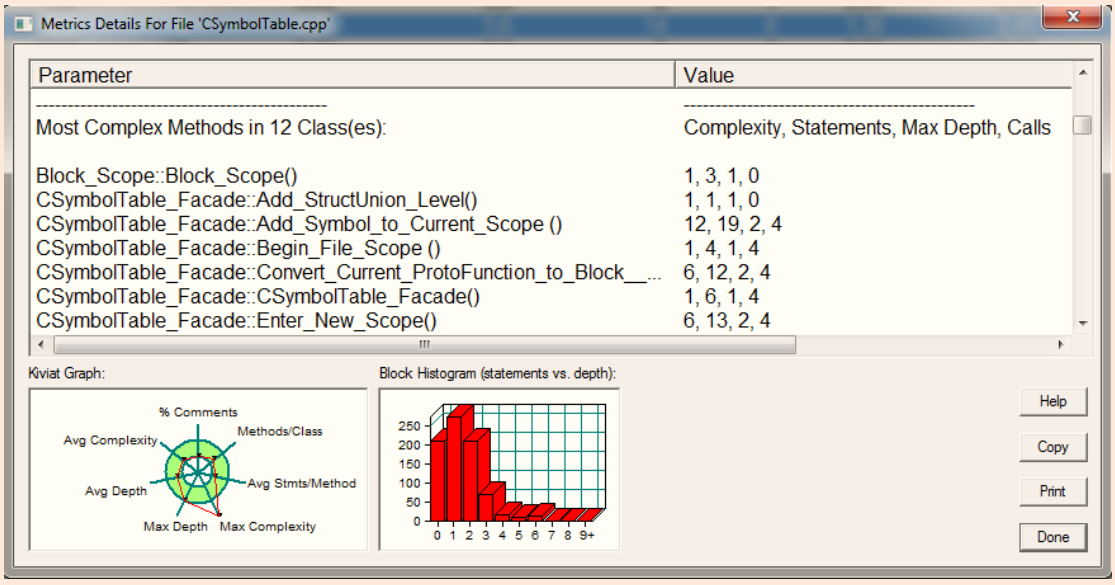
Εικόνα 13-3 : Γράφος μετρικών Kiviat του CSymbolTable.cpp module

Πίνακας 13-2 : Πίνακας επιμέρους μετρικών του CSymbolTable.cpp module 1/2

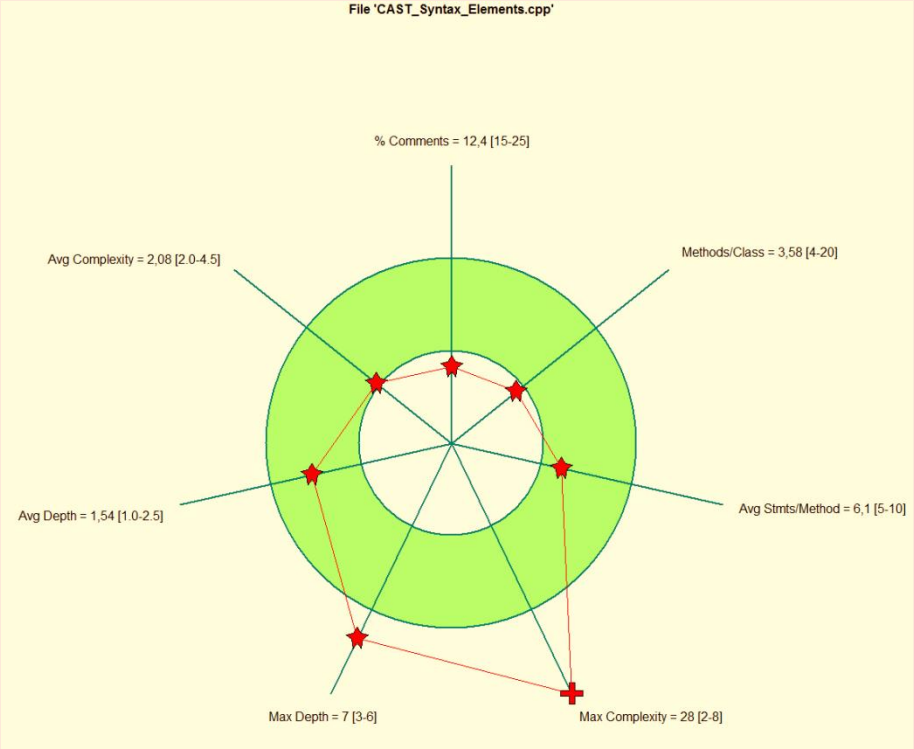
Class	Method Name	Complexity	Statements	Maximum Depth	Calls
Block_Scope	Block_Scope()	1	3	1	0
CSymbolTable_Facade	Add_StructUnion_Level()	1	1	1	0
CSymbolTable_Facade	Add_Symbol_to_Current_Scope ()	12	19	2	4
CSymbolTable_Facade	Begin_File_Scope ()	1	4	1	4
CSymbolTable_Facade	Convert_Current_ProtoFunction_to_Block__Scope()	6	12	2	4
CSymbolTable_Facade	CSymbolTable_Facade()	1	6	1	4
CSymbolTable_Facade	Enter_New_Scope()	6	13	2	4
CSymbolTable_Facade	Get_Symbol_Current_Scope ()	6	14	2	4
CSymbolTable_Facade	Get_Symbol_Name_Space ()	6	13	2	4
CSymbolTable_Facade	Is_in_StructUnion()	2	2	1	0
CSymbolTable_Facade	Is_Symbol_Type_Name ()	4	5	2	2
CSymbolTable_Facade	Leave_Current_Scope()	6	12	3	4
CSymbolTable_Facade	Make_Scopes_GraphViz_File()	5	40	2	55
CSymbolTable_Facade	Next_Auto_TagName()	1	8	1	6
CSymbolTable_Facade	ResolveCurrent_Struct_Union_Scope_andLeave()	1	1	1	1
CSymbolTable_Facade	Set_LastCurrent_to_Current_Scope()	6	12	3	4
CSymbolTable_Facade	Sub_StructUnion_Level()	1	1	1	0
CSymbolTable_Facade	~CSymbolTable_Facade()	1	4	1	0
File_Scope	File_Scope()	1	4	1	0
Function_Scope	Function_Scope()	1	4	1	0
Function_Scope	Get_Function_Symbol ()	1	1	1	0
Function_Scope	Set_Function_Symbol ()	1	1	1	0
MakeGraph_Scope_Visitor	MakeGraph_Scope_Visitor()	1	3	1	0
MakeGraph_Scope_Visitor	Visit_Scope()	3	16	1	21
Name_Space	Add_Symbol_to_Current_Scope ()	7	18	3	8
Name_Space	Convert_Current_ProtoFunction_to_Block__Scope()	3	12	2	10
Name_Space	Enter_New_Scope ()	14	41	4	9
Name_Space	Get_Child_Scope()	1	1	1	0
Name_Space	Get_Symbol_Current_Scope ()	4	9	3	4
Name_Space	Get_Symbol_Name_Space ()	6	10	3	3
Name_Space	Leave_Current_Scope()	3	8	2	4
Name_Space	Name_Space()	1	4	1	0
Name_Space	ResolveCurrent_Struct_Union_Scope_andLeave()	14	32	6	28
Name_Space	Set_LastCurrent_to_Current_Scope()	3	5	2	1
Name_Space	~Name_Space()	1	1	1	0
Proto_Function_Scope	Get_Function_Symbol ()	1	1	1	0
Proto_Function_Scope	Proto_Function_Scope()	1	4	1	0
Proto_Function_Scope	Set_Function_Symbol ()	1	1	1	0
Scope	Accept_Visitor()	1	1	1	1
Scope	Add_subScope()	6	20	3	6
Scope	Add_Symbol()	1	2	1	2
Scope	AddAll_SubScopes_from_oldScope ()	3	7	3	8
Scope	Copy_Object_Pointers_from()	1	5	1	5
Scope	Create_Iterator()	1	1	1	0
Scope	Delete_subScopePointer_from_Vector()	3	7	3	3
Scope	Get_All_Sybmols_String()	1	1	1	1
Scope	Get_All_Symbols_Current_Parent_Scopes()	7	16	6	13
Scope	Get_All_Symbols_Current_Scope()	3	6	3	4
Scope	Get_Begin()	1	1	1	0
Scope	Get_End()	1	1	1	0
Scope	Get_Enum_Tag()	1	1	1	0
Scope	Get_or_Add_Symbol()	1	1	1	1
Scope	Get_Parrent_NameSpace ()	1	1	1	0
Scope	Get_Parrent_Scope()	1	1	1	0
Scope	Get_Scope_Serial_Number()	1	1	1	0
Scope	Get_Scope_Type()	1	1	1	0
Scope	Get_Scope_Type_String()	6	16	3	5

Πίνακας 13-3 : Πίνακας επιμέρους μετρικών του CSymbolTable.cpp module 2/2

Class	Method Name	Complexity	Statements	Maximum Depth	Calls
Scope	Get_Symbol()	1	1	1	1
Scope	Get_Symbol()	1	1	1	1
Scope	Get_Symbol_Table()	1	1	1	0
Scope	Move_SubScope_from_ProtoFunctions_to_Blocks...	5	9	4	6
Scope	Replace_Symbol_Table()	1	3	1	1
Scope	Scope()	1	6	1	0
Scope	Set_Begin_Location ()	1	1	1	1
Scope	Set_End_Location ()	1	1	1	1
Scope	Set_Enum_Tag ()	1	1	1	0
Scope	Set_Object_Pointers_to_Zero()	1	4	1	0
Scope	Set_Parent_Scope_toall_Scope_SubScopes()	2	2	2	3
Scope	Set_Parent_Scope_toall_Scope_Symbols()	3	3	2	3
Scope	Set_Parrent_Scope ()	1	1	1	0
Scope	Set_Struct_Union_Tag ()	1	1	1	0
Scope	~Scope()	3	7	3	2
Scope_Visitor	~Scope_Visitor()	1	0	0	0
Symbol	Clear_Ambiguous()	1	1	1	0
Symbol	Get_Enum_Scope()	1	1	1	0
Symbol	Get_Hash_Key()	1	1	1	0
Symbol	Get_Loc()	1	1	1	0
Symbol	Get_Parent_Scope()	1	1	1	0
Symbol	Get_String()	1	1	1	0
Symbol	Get_Struct_Union_Scope()	1	1	1	0
Symbol	Is_Ambiguous()	1	1	1	0
Symbol	Is_Type_Name()	1	1	1	0
Symbol	Set_Ambiguous()	1	1	1	0
Symbol	Set_Enum_Scope()	1	1	1	0
Symbol	Set_Parent_Scope()	1	1	1	0
Symbol	Set_Struct_Union_Scope()	1	1	1	0
Symbol	Set_Type_Name()	1	1	1	0
Symbol	String_to_Hash()	2	4	2	0
Symbol	Symbol()	1	11	1	2
Symbol	Symbol()	1	11	1	1
Symbol	Symbol()	1	11	1	0
Symbol_Table	Add_Symbol()	6	11	3	4
Symbol_Table	DeleteTable_Objects()	1	0	0	0
Symbol_Table	Get_All_Sybmols_String()	4	9	4	6
Symbol_Table	Get_or_Add_Symbol()	6	14	3	5
Symbol_Table	Get_Symblo_Table_Size()	1	1	1	0
Symbol_Table	Get_Symbol()	1	1	1	0
Symbol_Table	Get_Symbol()	6	12	3	5
Symbol_Table	Initialize_Table()	2	2	2	0
Symbol_Table	Symbol_Index()	1	1	1	1
Symbol_Table	Symbol_Table()	1	2	1	1
Symbol_Table	~Symbol_Table()	1	1	1	1
Visitor	~Visitor()	1	0	0	0



Εικόνα 13-4 : Επισκόπηση μετρικών του CSymbolTable.cpp module



Εικόνα 13-5 : Γράφος μετρικών Kiviat του CAST_Syntax_Element.cpp module

Πίνακας 13-4 : Πίνακας μετρικών του CAST_Syntax_Elements.cpp module 1/2

Class	Method Name	Complexity	Statements	Maximum Depth	Calls
CAST_abstract_declarator	CAST_abstract_declarator()	2	6	2	1
CAST_abstract_declarator	CAST_abstract_declarator()	2	5	2	1
CAST_abstract_declarator	CAST_abstract_declarator()	2	5	2	1
CAST_abstract_declarator	~CAST_abstract_declarator()	1	0	0	0
CAST_additive_expression	CAST_additive_expression()	2	7	2	1
CAST_additive_expression	CAST_additive_expression()	2	5	2	1
CAST_additive_expression	~CAST_additive_expression()	1	0	0	0
CAST_and_expression	CAST_and_expression()	2	7	2	1
CAST_and_expression	CAST_and_expression()	2	5	2	1
CAST_and_expression	~CAST_and_expression()	1	0	0	0
CAST_argument_expression_list	CAST_argument_expression_list()	2	7	2	1
CAST_argument_expression_list	CAST_argument_expression_list()	2	5	2	1
CAST_argument_expression_list	~CAST_argument_expression_list()	1	0	0	0
CAST_assignment_expression	CAST_assignment_expression()	2	7	2	1
CAST_assignment_expression	CAST_assignment_expression()	2	5	2	1
CAST_assignment_expression	~CAST_assignment_expression()	1	0	0	0
CAST_assignment_operator	CAST_assignment_operator()	2	5	2	1
CAST_assignment_operator	~CAST_assignment_operator()	1	0	0	0
CAST_cast_expression	CAST_cast_expression()	2	8	2	1
CAST_cast_expression	CAST_cast_expression()	2	5	2	1
CAST_cast_expression	~CAST_cast_expression()	1	0	0	0
CAST_compound_statement	CAST_compound_statement()	2	8	2	1
CAST_compound_statement	CAST_compound_statement()	2	7	2	1
CAST_compound_statement	CAST_compound_statement()	2	7	2	1
CAST_compound_statement	CAST_compound_statement()	2	6	2	1
CAST_compound_statement	~CAST_compound_statement()	1	0	0	0
CAST_conditional_expression	CAST_conditional_expression()	2	9	2	1
CAST_conditional_expression	CAST_conditional_expression()	2	5	2	1
CAST_conditional_expression	~CAST_conditional_expression()	1	0	0	0
CAST_CONSTANT	CAST_CONSTANT()	20	81	3	27
CAST_CONSTANT	Extract_Character_From_Constant_Literal()	27	157	5	29
CAST_CONSTANT	Extract_Number_From_Constant_Literal()	10	35	4	12
CAST_CONSTANT	Get_Element_Info_String()	1	1	1	0
CAST_CONSTANT	~CAST_CONSTANT()	1	0	0	0
CAST_constant_expression	CAST_constant_expression()	2	5	2	1
CAST_constant_expression	~CAST_constant_expression()	1	0	0	0
CAST_declaration	CAST_declaration()	7	17	3	5
CAST_declaration	CAST_declaration()	2	6	2	1
CAST_declaration	~CAST_declaration()	1	0	0	0
CAST_declaration_list	CAST_declaration_list()	2	6	2	1
CAST_declaration_list	CAST_declaration_list()	2	5	2	1
CAST_declaration_list	~CAST_declaration_list()	1	0	0	0
CAST_declaration_specifiers	CAST_declaration_specifiers()	2	7	2	1
CAST_declaration_specifiers	CAST_declaration_specifiers()	2	6	2	1
CAST_declaration_specifiers	CAST_declaration_specifiers()	2	7	2	1
CAST_declaration_specifiers	CAST_declaration_specifiers()	2	6	2	1
CAST_declaration_specifiers	CAST_declaration_specifiers()	2	7	2	1
CAST_declaration_specifiers	CAST_declaration_specifiers()	2	6	2	1
CAST_declaration_specifiers	CAST_declaration_specifiers()	2	7	2	1
CAST_declaration_specifiers	CAST_declaration_specifiers()	2	6	2	1
CAST_declaration_specifiers	~CAST_declaration_specifiers()	1	0	0	0
CAST_declarator	CAST_declarator()	2	8	2	1
CAST_declarator	CAST_declarator()	4	14	2	1
CAST_declarator	~CAST_declarator()	1	0	0	0
CAST_direct_abstract_declarator	CAST_direct_abstract_declarator()	2	8	2	1
CAST_direct_abstract_declarator	CAST_direct_abstract_declarator()	2	7	2	1
CAST_direct_abstract_declarator	CAST_direct_abstract_declarator()	2	8	2	1

Πίνακας 13-5 : Πίνακας μετρικών του CAST_Syntax_Elements.cpp module 2/2

Class	Method Name	Complexity	Statements	Maximum Depth	Calls
CAST_direct_declarator	CAST_direct_declarator()	2	9	2	1
CAST_direct_declarator	CAST_direct_declarator()	2	6	2	1
CAST_direct_declarator	~CAST_direct_declarator()	1	0	0	0
CAST_Element_Type	CAST_Element_Type()	19	62	5	12
CAST_Element_Type	~CAST_Element_Type()	1	0	0	0
CAST_enum_specifier	CAST_enum_specifier()	2	6	2	1
CAST_enum_specifier	CAST_enum_specifier()	2	9	2	1
CAST_enum_specifier	CAST_enum_specifier()	2	8	2	1
CAST_enum_specifier	~CAST_enum_specifier()	1	0	0	0
CAST_enumerator	CAST_enumerator()	2	7	2	1
CAST_enumerator	CAST_enumerator()	2	5	2	1
CAST_enumerator	~CAST_enumerator()	1	0	0	0
CAST_enumerator_list	CAST_enumerator_list()	2	7	2	1
CAST_enumerator_list	CAST_enumerator_list()	2	5	2	1
CAST_enumerator_list	~CAST_enumerator_list()	1	0	0	0
CAST_equality_expression	CAST_equality_expression()	2	7	2	1
CAST_equality_expression	CAST_equality_expression()	2	5	2	1
CAST_equality_expression	~CAST_equality_expression()	1	0	0	0
CAST_exclusive_or_expression	CAST_exclusive_or_expression()	2	7	2	1
CAST_exclusive_or_expression	CAST_exclusive_or_expression()	2	5	2	1
CAST_exclusive_or_expression	~CAST_exclusive_or_expression()	1	0	0	0
CAST_expression	CAST_expression()	2	7	2	1
CAST_expression	CAST_expression()	2	5	2	1
CAST_expression	~CAST_expression()	1	0	0	0
CAST_expression_statement	CAST_expression_statement()	2	6	2	1
CAST_expression_statement	CAST_expression_statement()	2	5	2	1
CAST_expression_statement	~CAST_expression_statement()	1	0	0	0
CAST_external_declaration	CAST_external_declaration()	2	5	2	1
CAST_external_declaration	CAST_external_declaration()	2	5	2	1
CAST_external_declaration	~CAST_external_declaration()	1	0	0	0
CAST_Facade	CAST_Facade()	1	1	1	0
CAST_Facade	Get_CAST_Root()	1	1	1	0
CAST_Facade	Make_CAST_GraphViz_File()	3	13	2	16
CAST_Facade	Set_CAST_Root()	1	1	1	0
CAST_Facade	~CAST_Facade()	1	0	0	0
CAST_function_definition	CAST_function_definition()	2	6	2	1
CAST_function_definition	CAST_function_definition()	2	7	2	1
CAST_function_definition	CAST_function_definition()	2	7	2	1
CAST_function_definition	CAST_function_definition()	2	8	2	1
CAST_function_definition	~CAST_function_definition()	1	0	0	0
CAST_function_specifier	CAST_function_specifier()	2	6	2	1
CAST_function_specifier	~CAST_function_specifier()	1	0	0	0
CAST_IDENTIFIER	CAST_IDENTIFIER ()	1	2	1	0
CAST_IDENTIFIER	Get_Element_Info_String()	1	1	1	0
CAST_IDENTIFIER	Get_Symbol()	1	1	1	0
CAST_IDENTIFIER	Set_Symbol()	1	1	1	0
CAST_IDENTIFIER	~CAST_IDENTIFIER()	1	0	0	0
CAST_identifier_list	CAST_identifier_list()	2	7	2	1
CAST_identifier_list	CAST_identifier_list()	2	5	2	1
CAST_identifier_list	~CAST_identifier_list()	1	0	0	0
CAST_inclusive_or_expression	CAST_inclusive_or_expression()	2	7	2	1
CAST_inclusive_or_expression	CAST_inclusive_or_expression()	2	5	2	1
CAST_inclusive_or_expression	~CAST_inclusive_or_expression()	1	0	0	0
CAST_init_declarator	CAST_init_declarator()	2	8	2	1
CAST_init_declarator	CAST_init_declarator()	2	6	2	1
CAST_init_declarator	~CAST_init_declarator()	1	0	0	0
CAST_init_declarator_list	CAST_init_declarator_list()	2	7	2	1