ΑΛΕΞΑΝΔΡΕΙΟ Τ.Ε.Ι ΘΕΣΣΑΛΟΝΙΚΗΣ

ΣΧΟΛΗ ΤΕΧΝΟΛΟΓΙΚΩΝ ΕΦΑΡΜΟΓΩΝ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Πτυχιακή εργασία

ΣΧΕΔΙΑΣΗ ΚΑΙ ΥΛΟΠΟΙΗΣΗ ΠΡΟΤΥΠΩΝ ΤΗΣ ΜΗΧΑΝΙΚΗΣ ΠΑΙΧΝΙΔΙΩΝ ΜΕ ΑΝΤΙΚΕΙΜΕΝΟΣΤΡΕΦΗ ΠΡΟΤΥΠΑ ΣΧΕΔΙΑΣΗΣ

Νικολαΐδης Ευστάθιος

Επιβλέπων Καθηγητής

Αμπατζόγλου Απόστολος

Θεσσαλονίκη 2012

Θα ήθελα να ευχαριστήσω ιδιαίτερα τον κύριο Απόστολο Αμπατζόγλου, για τον πολύτιμο χρόνο που αφιέρωσε για τη καθοδήγησή μου και τη Σοφία Σπανίδου που μου χάρισε ηρεμία και συμβουλές καθόλη τη διάρκεια εκπόνησης της πτυχιακής εργασίας.

ΠΡΟΛΟΓΟΣ

Το τελευταίο καιρό οι 3D εφαρμογές φαίνεται να είναι από βιομηχανικής άποψης ένας από τους πιο σύγχρονους και συνεχώς αυξανόμενους τομείς. Οι πιο δημοφιλείς μεταξύ αυτών φαίνεται πως είναι τα παιχνίδια στον υπολογιστή (video games). Το 2007, τα video games είχαν έσοδα περίπου 60 δισεκατομμύρια δολάρια, νούμερο που ισούται σχεδόν με το μέγεθος των δαπανών του Υπουργείου Άμυνας των ΗΠΑ για έρευνα.

Επιπλέον, η βιομηχανία παιχνιδιών είναι τόσο καινοτόμα που σε πολλές περιπτώσεις, η τεχνολογική πρόοδος σε hardware και software, εφαρμόζεται στα παιχνίδια προτού εγκριθεί σε άλλα επιστημονικά πεδία. Ακόμη, μεταξύ των νέων, τα video games είναι πιο δημοφιλή στον ελεύθερό τους χρόνο ακόμη κι απ' τη τηλεόραση και τη μουσική.

Η Δημιουργία ενός παιχνιδιού στον υπολογιστή είναι μια πολύ σύνθετη διαδικασία που απαιτεί τη συμμετοχή εξαιρετικά ειδικευμένων επαγγελματιών από ένα ευρύ φάσμα της επιστήμης των υπολογιστών.

Ο προγραμματισμός σε ένα παιχνίδι χαμηλού επιπέδου είναι τόσο πολύπλοκος που απαιτούνται εκατοντάδες χιλιάδες γραμμές κώδικα προκειμένου το παιχνίδι να είναι εμπορικά βιώσιμο. Το μέγεθος των εν λόγω προγραμμάτων, σε συνδυασμό με την εξελισσόμενη φύση του λογισμικού, απαιτεί ευελιξία στο σχεδιασμό, την υλοποίηση και διατηρήσιμη απλή τεκμηρίωση, προκειμένου να βελτιωθεί η κατανόηση μεταξύ της ομάδας ανάπτυξης και τη διευκόλυνση των μελλοντικών εξελίξεων.

Ως εκ τούτου, οι προγραμματιστές παιχνιδιών πρέπει να χρησιμοποιούν ειδικές τεχνικές μηχανικής λογισμικού, προκειμένου να επιτευχθούν υψηλά επίπεδα ποιότητας.

Τα παιχνίδια είναι προϊόντα που έχουν πολύ πιο περιορισμένη διάρκεια ζωής από ό, τι τα συμβατικά προϊόντα λογισμικού. Τα παιχνίδια συνήθως αναπτύσσονται σε ένα μικρότερο χρονικό διάστημα και όλες οι φάσεις του κύκλου ζωής πρέπει να συρρικνωθούν. Επιπλέον, η κύρια δραστηριότητα συντήρησης για τα ηλεκτρονικά παιχνίδια είναι διορθωτική συντήρηση, επειδή τα περισσότερα παιχνίδια, αφού παραδοθούν στην αγορά, έχουν μέσο όρο ζωής 6 μήνες και σε αυτό το χρονικό διάστημα η επόμενη έκδοση του προγράμματος έχει ήδη δημιουργηθεί. Κατά τη διάρκεια αυτής της περιόδου, το κύριο καθήκον είναι η διόρθωση σφαλμάτων, η οποία συνήθως παρέχεται χωρίς επιβάρυνση στους τελικούς

χρήστες και κατά συνέπεια, οι εταιρείες ανάπτυξης παιχνιδιών δεν έχουν εισόδημα για αυτή τη συντήρηση.

Ωστόσο, η επιτυχία ενός παιχνιδιού είναι συχνά η βάση για μία ή περισσότερες συνέχειες. Αν η συνέχεια περιλαμβάνει αναθεωρήσεις για το περιβάλλον εργασίας χρήστη ή ελέγχου του παιχνιδιού, ως αποτέλεσμα των παρατηρήσεων από τους χρήστες, αυτή είναι μία μορφή τέλειας συντήρησης. Τέλος, ένα άλλο ενδιαφέρον χαρακτηριστικό των παιχνιδιών είναι το γεγονός ότι σε πολλές περιπτώσεις, οι εταιρείες ανάπτυξης παιχνιδιών δίνουν στην αγορά επεκτάσεις. Αυτές οι εκδόσεις χρησιμοποιούν τον ίδιο πυρήνα του παιχνιδιού, προκειμένου αυτή η «νέα ιστορία» να μπει μέσα από την «παλιά» μηχανή του παιχνιδιού, το οποίο έχουν ήδη αγοράσει οι χρήστες. Η διαδικασία αυτή μπορεί να χαρακτηριστεί ως προσαρμοστική συντήρηση.

Μια λύση για να διεκπεραιωθεί σωστά ο σχεδιασμός ενός παιχνιδιού, είναι οι προγραμματιστές που συμμετέχουν στη βιομηχανία των παιχνιδιών, να χρησιμοποιούν μια σειρά από μεθοδολογίες και τεχνικές που δανείζονται από την ανάπτυξη λογισμικού, τη βιομηχανία του κινηματογράφου και τα παραδοσιακά παιχνίδια.

ΠΕΡΙΛΗΨΗ

Η παρούσα πτυχιακή ασχολείται με την ανάλυση προτύπων αντικειμενοστρεφούς προγραμματισμού και την εξεύρεση αντίστοιχων προτύπων μηχανικής παιχνιδιών, με στόχο τη δημιουργία προγραμμάτων που συνδυάζουν και τα δύο.

Το πρώτο κεφάλαιο αναφέρει πληροφορίες σχετικά με το τι είναι τα διαγράμματα κλάσεων UML και κάνει μια ανάλυση σχετικά με τη βιομηχανία των παιχνιδιών και την αρχιτεκτονική τους.

Στο δεύτερο κεφάλαιο παρουσιάζονται πέντε πρότυπα σχεδίασης αντικειμενοστρεφούς προγραμματισμού με επισημάνσεις στο καθένα για το σκοπό τους, την εφαρμογή τους και ένα παράδειγμα με διάγραμμα κλάσεων UML.

Αντίστοιχα στο τρίτο κεφάλαιο με τον ίδιο τρόπο παρουσιάζονται πέντε πρότυπα μηχανικής παιχνιδιών ενώ στο τέταρτο κεφάλαιο δίνεται ο κώδικας και τα διαγράμματα για πέντε παραδείγματα – προγράμματα που τα υλοποιούν με πρότυπα αντικειμενοστρεφούς προγραμματισμού.

Τέλος δίνονται τα συμπεράσματα καθώς και η βιβλιογραφία της παρούσας πτυχιακής.

ΠΙΝΑΚΑΣ ΠΕΡΙΕΧΟΜΕΝΩΝ

ΠΡΟΛΟΓΟΣ ii

ΠΕΡΙΛΗΨΗ iv

ΠΙΝΑΚΑΣ ΠΕΡΙΕΧΟΜΕΝΩΝ v

1 Εισαγωγή 1

1.1 Διαγράμματα Κλάσεων UML 1

1.2 Προγραμματισμός Παιχνιδιών 4

1.2.1 Βιομηχανία Ηλεκτρονικών παιχνιδιών 4

1.2.2 Αρχιτεκτονική παιχνιδιών 5

1.3 Παιχνίδια και Πρότυπα Σχεδίασης 7

2 Αντικειμενοστρεφή Πρότυπα Σχεδίασης 8

2.1 Visitor - Επισκέπτης 11

2.2 Builder - Κατασκευαστής 14

2.3 Observer - Παρατηρητής 16

2.4 Proxy - Πληρεξούσιο 18

2.5 Template Method – Μέθοδος Υπόδειγμα 20

3 Πρότυπα Σχεδίασης Μηχανικής Παιχνιδιών 22

3.1 Paper – Rock - Scissors 23

3.2 Analysis - Paralysis 25

3.3 Collision Detection 26

3.4 Shared Rewards 26

3.5 Game Loop 28

4 Υλοποίηση Προτύπων Μηχανικής Παιχνιδιών με Αντικειμενοστρεφή Πρότυπα 30

4.1 BattleTest (Paper-Rock-Scissors με Visitor) 30

4.2 CharBuilderDemo (Analysis – Paralysis με Builder) 40

4.3 CollisionObserverDemo (Collision Detection με Observer) ... 50

4.4 SharedTreasure (Shared Rewards με Proxy) ... 56

4.5 LoopDemo (Game Loop με Template Method) ... 60

5 Συμπεράσματα ... 64

ΒΙΒΛΙΟΓΡΑΦΙΑ ... 65

1 Εισαγωγή

Στο παρόν κεφάλαιο βρίσκουμε τις βασικές έννοιες των διαγραμμάτων κλάσεων UML τις οποίες και θα χρειαστούμε παρακάτω, ενώ δίνεται μια εικόνα για τον κόσμο των ηλεκτρονικών παιχνιδιών και κυρίως για τον τρόπο με τον οποίο προγραμματίζονται και το τι προβλήματα παρουσιάζονται.

1.1 Διαγράμματα Κλάσεων UML

Τα διαγράμματα κλάσεων περιγράφουν τις οντότητες που απαρτίζουν ένα σύστημα και τις στατικές συσχετίσεις μεταξύ τους. Αποτελούνται από:

- Κλάσεις (*classes*)
- Σχέσεις (*relationships*)

Κλάσεις

Οι κλάσεις απεικονίζονται με:

Όνομα, το οποίο είναι σε πλάγιους χαρακτήρες όταν πρόκειται για αφηρημένη κλάση (*abstract class*).

Ιδιότητες (*attributes*) οι οποίες χαρακτηρίζονται από την ορατότητά τους (*visibility*) σε:

- ✓ public (+)
- ✓ private (-)
- ✓ package (~)
- ✓ protected (#)

Οι ιδιότητες χαρακτηρίζονται επίσης από την πολλαπλότητά τους ως εξής:

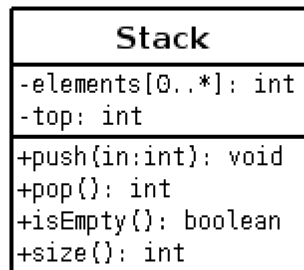
- ✓ 1 (ακριβώς μία)
- ✓ 0..1 (μία ή καμμία)
- ✓ *, ή 0..* (καμμία ή οσεσδήποτε)
- ✓ n..m (από n έως m, όπου m μπορεί να είναι και *)

Μεθόδους (*methods*) που χαρακτηρίζονται από την ορατότητά τους όπως και οι ιδιότητες.

Οι μέθοδοι και οι ιδιότητες που ανήκουν στην κλάση και όχι στα στιγμιότυπά της (*static*) αναγράφονται με πλάγιους χαρακτήρες.

Παράδειγμα Κλάσης

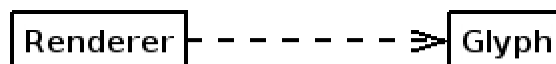
Στο παρακάτω διάγραμμα απεικονίζεται μια απλή κλάση στίβας (*stack*):



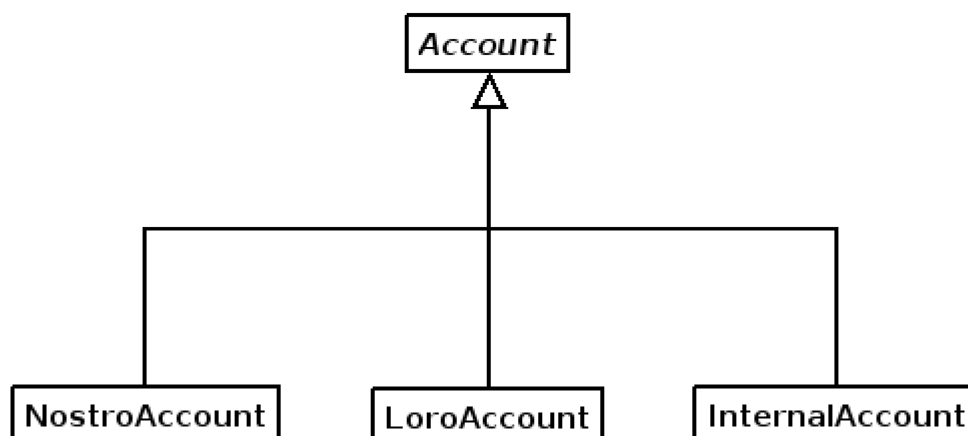
<u>**Σχέσεις**</u>

Οι σχέσεις συνδέουν μεταξύ τους τις κλάσεις ενός διαγράμματος. Η UML ορίζει τις εξής βασικές σχέσεις:

Εξάρτηση (*dependency*), η οποία δείχνει ότι μία αλλαγή σε μία κλάση επηρεάζει μία άλλη κλάση.



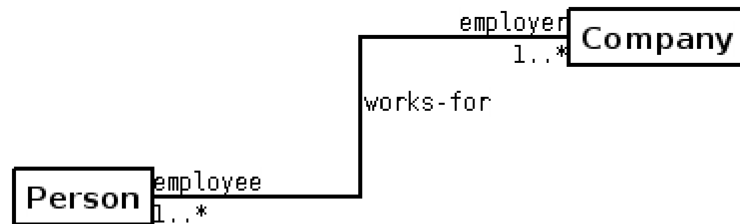
Γενίκευση (*generalisation*), η οποία δείχνει ότι μια κλάση είναι ένας πιο εξειδικευμένος τύπος μιας άλλης κλάσης.



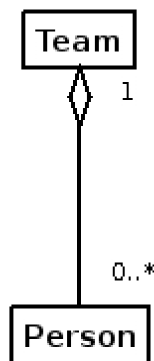
Σύνδεση (*association*), η οποία δείχνει ότι μία κλάση έχει μία δομική σύνδεση με μία άλλη κλάση.

Η σύνδεση μπορεί να έχει πολλαπλότητα όπως και οι ιδιότητες. Η πολλαπλότητα σημειώνεται στο αντίστοιχο άκρο της γραμμής. Η σύνδεση μπορεί να είναι κατευθυνόμενη (οπότε η κατεύθυνση σημειώνεται με ένα ανοιχτό βέλος) ή αμφίδρομη. Ακόμη μπορεί να έχει όνομα που αναγράφεται από πάνω της.

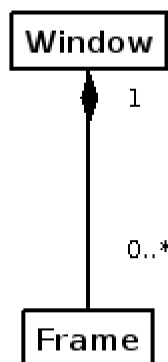
Οι εμπλεκόμενες κλάσεις συμμετέχουν στη σύνδεση με κάποιο συγκεκριμένο ρόλο ο οποίος μπορεί να αναγραφεί στο αντίστοιχο άκρο.



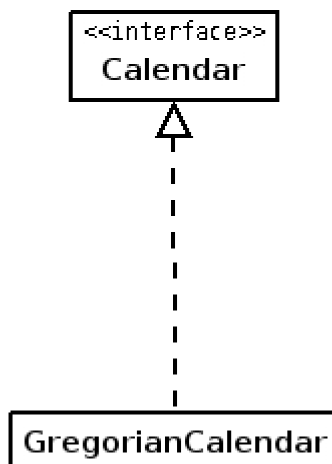
Η συγκρότηση (*aggregation*), είναι μία σύνδεση που απεικονίζει τη σχέση μίας κλάσης που απαρτίζεται από κάποιες άλλες.



Η σύνθεση (*composition*), είναι μία σύνδεση που απεικονίζει τη σχέση μίας κλάσης που συντίθεται από κάποιες έτσι ώστε κάθε στιγμιότυπο ενός συστατικού πρέπει να ανήκει σε ένα και μόνο ένα στιγμιότυπο της συντιθέμενης κλάσης.



Υλοποίηση (*realisation*), η οποία δείχνει ότι μία κλάση υλοποιεί κάποια προδιαγραφή (*specification*). Η προδιαγραφή συνήθως είναι μια διεπαφή (*interface*).



1.2 Προγραμματισμός Παιχνιδιών

Στην ενότητα 1.2 θα παρουσιαστεί μία εισαγωγή σχετικά με το προγραμματισμό παιχνιδιών. Αρχικά στην ενότητα 1.2.1 θα παρουσιαστεί η τρέχουσα κατάσταση της βιομηχανίας ηλεκτρονικών παιχνιδιών που τα τελευταία χρόνια είναι ακμάζουσα, και στην ενότητα 1.2.2 θα παρουσιαστεί μία ενδεικτική αρχιτεκτονική ενός τυπικού παιχνιδιού.

1.2.1 Βιομηχανία Ηλεκτρονικών παιχνιδιών

Η βιομηχανία ηλεκτρονικών παιχνιδιών είναι πολύ αισιόδοξη για το μέλλον της. Το 1997 το ετήσιο εισόδημα της αμερικανικής βιομηχανίας παιχνιδιών για κομπιούτερ και βιντεοπαιχνιδιών έφτασε τα 5,4 δισεκατομμύρια ευρώ, και οι παγκόσμιες πωλήσεις ανήλθαν σε τουλάχιστον 10,1 δισεκατομμύρια ευρώ. Φαίνεται ότι αυτή η τάση δεν θα ανακοπεί. Η συγκεκριμένη αγορά αναμένεται να σημειώσει άνοδο από 50 ως 75 τοις εκατό στη διάρκεια των ερχόμενων πέντε χρόνων.

Κάθε ημέρα, σύμφωνα με τον οργανισμό *Φόρεστερ Ρισέρτς* (*Forrester Research*), πάνω από ένα εκατομμύριο άνθρωποι συμμετέχουν σε διάφορα παιχνίδια που βασίζονται στο Ιντερνέτ, και λέγεται ότι το ενδιαφέρον για τα διαδικτυακά παιχνίδια θα αυξηθεί με την εξάπλωση ενός τύπου σύνδεσης ευρείας ζώνης (broadband) με το Ιντερνέτ σε υψηλές ταχύτητες. Τα παιδιά

που έχουν μεγαλώσει παίζοντας παιχνίδια για κομπιούτερ δεν δείχνουν ότι θα σταματήσουν να παίζουν καθώς μεγαλώνουν. Κάποιος που παίζει πολύ καιρό λέει: «Με τα παιχνίδια για κομπιούτερ μπορεί κανείς να συναναστραφεί με φίλους από όλο τον κόσμο».

Σύμφωνα με άρθρο των New York Times, στον κινηματογράφο 2 από τις 10 ταινίες θα πραγματοποιήσουν κέρδη. Τα ίδια ποσοστά, σύμφωνα με το ίδιο άρθρο, αρχίζει να αποκτά και η βιομηχανία ηλεκτρονικών παιχνιδιών. Ένα παιχνίδι που κοστίζει 10 εκ. δολάρια και παίρνει άλλα 10 εκ. για την προώθησή του στην αγορά θα πρέπει να πουλήσει παραπάνω τεμάχια απ' ότι ένα αντίστοιχο παιχνίδι στα τέλη της δεκαετίας του '90 που το μέσο μπάτζετ παραγωγής ενός video game έφτανε τα 3 εκ. δολάρια. Και τα ποσά μπορεί να γίνουν πολύ υψηλότερα για μερικά παιχνίδια. Η Atari ξόδεψε 20 εκ. δολάρια για το «Enter the Matrix».Μπορεί τα ηλεκτρονικά παιχνίδια να μην έχουν φτάσει το κόστος μιας χολιγουντιανής υπερπαραγωγής, αλλά αν διατηρηθούν οι σημερινοί ρυθμοί αύξησης των εσόδων των μεγάλων παραγωγών video games όπως είναι η No.1 στον κόσμο Electronic Arts και η No.2 Activision σύντομα θα το ξεπεράσουν. Κάτι που αποδεικνύει ότι η βιομηχανία των ονείρων έχει αποκτήσει σαν καύσιμο την αισθητική των παιχνιδιών που έχουμε στην κονσόλα μας.

Η έκθεση με τίτλο «Οικοσύστημα Παιχνιδιών 2011» εκτιμά ότι οι δαπάνες για λογισμικό και hardware του τομέα το 2011 θα υπερβούν τα 74 δισ. δολάρια, από 67 δισ. δολάρια το 2010. Η μεγαλύτερη ανάπτυξη αναμένεται, όμως, να σημειωθεί στον τομέα των παιχνιδιών κινητών τηλεφώνων, λέει ο Τουόνγκ Νουγιέν, επικεφαλής ερευνών της Gartner, που προβλέπει μάλιστα ότι οι πωλήσεις και η χρήση φορητών «παιχνιδο-μηχανών», όπως αυτών της Sony και Nintendo, θα συρρικνωθούν καθώς οι νέοι χρήστες τους θα επιλέξουν «έξυπνα» κινητά τηλέφωνα ή συσκευές tablet, για να παίξουν ηλεκτρονικά παιχνίδια.

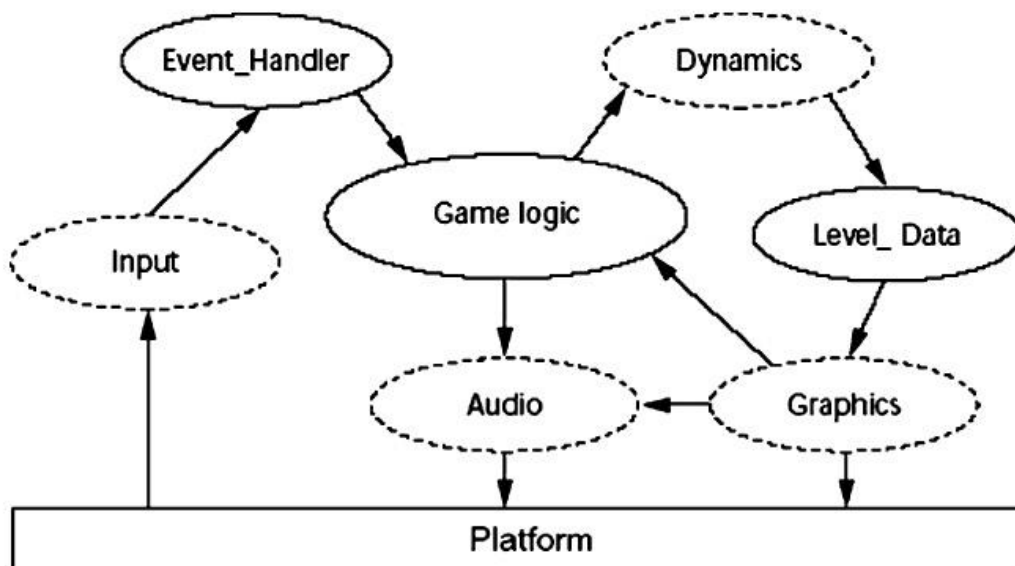
Την ίδια στιγμή, οι πωλήσεις λογισμικού παιχνιδιών έχουν σπάσει κάθε προηγούμενο ρεκόρ, με την Activision -κατασκευάστρια προηγμένων παιχνιδιών για το Xbox και το PS3, να σημειώνει κέρδη 650 εκατ. δολαρίων σε πέντε ημέρες από τις πωλήσεις του νέου της παιχνιδιού Call of Duty: Black Ops.

1.2.2 Αρχιτεκτονική παιχνιδιών

Μία από τις πιο ενδιαφέρουσες πτυχές της έρευνας του προγραμματισμού παιχνιδιών, είναι η αρχιτεκτονική που ο προγραμματιστής θα χρησιμοποιήσει. Τελευταία, γίνεται εκτεταμένη

συζήτηση για τη συντήρηση και την επαναχρησιμοποίηση του κώδικα των παιχνιδιών . Αυτά τα ζητήματα έχουν εξεταστεί λεπτομερώς στο κλασικό αντικειμενοστρεφή προγραμματισμό, αλλά αυτές οι ιδέες είναι εξαιρετικά «ανώριμες» (ακόμη) στον προγραμματισμό παιχνιδιών. Ο σχεδιασμός και προγραμματισμός μίας μεγάλης κλίμακας λογισμικού, είναι ένα πολύ περίπλοκο έργο που απαιτεί πολλές ανθρώπινες ώρες εργασίας. Ως εκ τούτου, το λογισμικό είναι συνήθως χωρισμένο σε υποπρογράμματα που έχουν σχεδιαστεί, προγραμματιστεί και ελεγθεί από ξεχωριστές ομάδες προγραμματιστών. Αυτά τα υποπρογράμματα ονομάζονται μονάδες (modules). Η αποσύνθεση σε ενότητες του λογισμικού είναι μια πολύ σημαντική απόφαση που παίζει κεντρικό ρόλο στην αρχιτεκτονική και τον περαιτέρω σχεδιασμό του παιχνιδιού.

Παρακάτω δίνεται ένα γενικό παράδειγμα μιας αρχιτεκτονικής παιχνιδιού:



Τα στοιχεία με έντονα περιγράμματα είναι ουσιαστικής σημασίας για κάθε παιχνίδι, ενώ αυτά με τις διακεκομμένες γραμμές αναφέρονται σε ενότητες που βρίσκονται σε πιο περίπλοκα και απαιτητικά παιχνίδια. Το “Game Logic” είναι το μέρος που κρατά την ιστορία (story) του παιχνιδιού. Ο ήχος(Audio) και γραφικά(Graphics) είναι οι ενότητες που βοηθούν τους συγγραφείς να αφηγούνται την ιστορία στο παίκτη. Το «Event_Handler» και οι μονάδες εισόδου (Input), παρέχουν τη “Game Logic με την επόμενη ενέργεια του παίκτη. Τέλος, η μονάδα δεδομένων επίπεδου (Level_Data) είναι μια μονάδα αποθήκευσης για λεπτομέρειες σχετικά με συμπεριφορά διαμορφώνει δυναμικά τη συμπεριφορά των χαρακτήρων του παιχνιδιού.

1.3 Παιχνίδια και Πρότυπα Σχεδίασης

Για να διεκπεραιωθεί σωστά ο σχεδιασμός ενός παιχνιδιού, οι προγραμματιστές που συμμετέχουν στη βιομηχανία των παιχνιδιών, χρησιμοποιούν μια σειρά από μεθοδολογίες και τεχνικές που δανείζονται από την ανάπτυξη λογισμικού, τη βιομηχανία του κινηματογράφου και τα παραδοσιακά παιχνίδια. Αν και αυτή η προσέγγιση φαίνεται να αποφέρει καλά αποτελέσματα, κρίνοντας από τα παιχνίδια που βγαίνουν στην αγορά, υπάρχει μια έντονη ανησυχία από τους επαγγελματίες σχεδιαστές παιχνιδιών, για έλλειψη μιας κοινής, αναπτυγμένης, τεχνικής σχεδιασμού ηλεκτρονικών παιχνιδιών. Η ανάγκη αυτή γίνεται πιο επιτακτική, σκεπτόμενοι το γεγονός ότι στην ανάπτυξη ενός παιχνιδιού, χρησιμοποιείται πολλών διαφορετικών ειδικοτήτων προσωπικό, όπως προαναφέρθηκε.

Αν και τα πιεστικά χρονικά περιθώρια που δίνονται για την ανάπτυξη των παιχνιδιών δεν ευνοούν τη διεξαγωγή έρευνας, οι επαγγελματίες προγραμματιστές έχουν ανάγκη την εύρεση νέων, καλύτερων μεθοδολογιών σχεδίασης λογισμικού για παιχνίδια.

Σχετικά με τους μηχανισμούς των παιχνιδιών, ο Bjork παρουσιάζει μια ομάδα σχεδιαστικών προτύπων που ουσιαστικά, αποτελούν περιγραφές από συχνά εμφανιζόμενές αλληλεπιδραστικές διατάξεις που αφορούν την ιστορία των παιχνιδιού και τον τρόπο παιξίματος του. Αυτό συνεπάγεται ότι τα πρότυπα αυτά δεν σχετίζονται με την αρχιτεκτονική του λογισμικού και τον κώδικα. Τα προτεινόμενα πρότυπα συλλέχθηκαν από συνεντεύξεις προγραμματιστών που ασχολούνται επαγγελματικά με τα παιχνίδια, από ανάλυση υπάρχων παιχνιδιών και από μετασχηματισμό των μηχανισμών που συναντώνται στα παιχνίδια. Όπως αναφέρεται και στο άρθρο «Ο ιδανικός τρόπος για την αναγνώριση πρότυπων είναι να παίζεις παιχνίδια, να σκέφτεσαι παιχνίδια, να ονειρεύεσαι παιχνίδια, να σχεδιάζεις παιχνίδια και να διαβάζεις για παιχνίδια».

Παρά το γεγονός ότι μέχρι τώρα δεν υπάρχου πολλά πράγματα που βρέθηκαν στη χρήση αντικειμενοστρεφούς σχεδιαστικών προτύπων στα παιχνίδια, πιστεύουμε ότι μια τέτοια χρήση μπορεί να αποδειχθεί πολύ χρήσιμη στον τομέα αυτό.

Το γεγονός αυτό μπορεί να εξεταστεί με τη διερεύνηση τον πηγαίο κώδικα των παιχνιδιών για την ύπαρξη των προτύπων σχεδιασμού.

2 Αντικειμενοστρεφή Πρότυπα Σχεδίασης

Ένας προγραμματιστής υλοποιώντας ένα σύστημα λογισμικού έχει να αντιμετωπίσει αρκετά προβλήματα, τα οποία εμφανίζονται συνεχώς (όχι μόνο μία φορά). Συνήθως έχουν εμφανιστεί και σε άλλα έργα λογισμικού σε άλλους προγραμματιστές ή και στον ίδιο και αντιμετωπίσθηκαν επιτυχώς. Η στρατηγική επίλυσης πολλών από αυτών των προβλημάτων είναι κοινή ή παρόμοια σε αρκετές περιπτώσεις, ειδικά όταν αυτή αποτυπώνεται σε στατική δομή ενός αντικειμενοστρεφούς συστήματος λογισμικού.

Για να συστηματοποιήσουμε κάποιες από αυτές τις «συνηθισμένες» λύσεις σε καθημερινά και συνηθισμένα προβλήματα λογισμικού, χρησιμοποιούμε τα πρότυπα σχεδίασης (Design Patterns). Κάθε πρότυπο σχεδίασης κατονομάζει τη συγκεκριμένη λύση, παρέχει μια περιγραφή του προβλήματος στο οποίο μπορεί να εφαρμοστεί, και προδιαγράφει τη λύση, συνήθως σε επίπεδο αρχιτεκτονικής σχεδίασης. Το όνομα κάθε προτύπου διευκολύνει την επικοινωνία μεταξύ προγραμματιστών καθώς και την εύκολη αναφορά σε κοινά είδη προβλημάτων. Το πρόβλημα σε κάθε πρότυπο προσδιορίζει ένα γενικότερο πλαίσιο όπου υπό κανονική αντιμετώπιση (χωρίς τη χρήση προτύπων) θα προέκυπταν ανεπιθύμητες συνέπειες αναφορικά με την λειτουργία, τον έλεγχο και τη συντήρηση του λογισμικού. Τέλος, η λύση περιγράφει τα στοιχεία από τα οποία πρέπει να συγκροτείται το σχέδιο, τις μεταξύ τους σχέσεις, τις ιδιότητες και τις αρμοδιότητες αυτών.

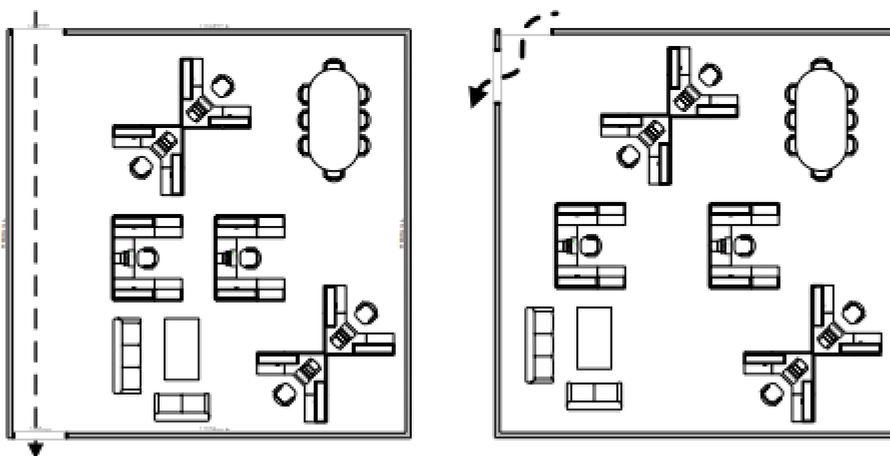
Η έμπνευση για τη χρήση προτύπων στην Τεχνολογία Λογισμικού προήλθε από το χώρο της Αρχιτεκτονικής και συγκεκριμένα από την προσπάθεια του αρχιτέκτονα Christopher Alexander ο οποίος στο βιβλίο του "A Pattern Language: Towns, Buildings, Construction (1977)", προσπάθησε να δώσει απάντηση στο ερώτημα εάν η ποιότητα (στην αρχιτεκτονική) είναι μια αντικειμενική ιδιότητα. Αν αποδεχθεί κανείς ότι είναι δυνατό να αναγνωρίσει και να περιγράψει ένα αρχιτεκτονικό σχέδιο καλής ποιότητας, τότε σύμφωνα με τον Alexander το επόμενο εύλογο ερώτημα είναι τι υπάρχει σε ένα σχέδιο καλής ποιότητας το οποίο δεν εμφανίζεται σε ένα σχέδιο κακής ποιότητας;

Μελετώντας πληθώρα αρχιτεκτονικών κατασκευασμάτων, ο Alexander παρατήρησε ότι αυτές που θεωρούνται καλές κατασκευές είχαν κοινά στοιχεία μεταξύ τους. Τα κοινά αυτά στοιχεία, συνήθως αφορούν κοινές "λύσεις" ή λύσεις σε κοινά προβλήματα. Κατανοώντας ότι οι διάφορες δομές δεν μπορούν να διαχωριστούν από το πρόβλημα το οποίο προσπαθούν

να επιλύσουν, αναζήτησε τις διαφορετικές δομές που σχεδιάστηκαν για να επιλύσουν το ίδιο πρόβλημα.

Για παράδειγμα στο "πρόβλημα" του περιορισμού των ενοχλήσεων σε ένα δωμάτιο λόγω πολλαπλών θυρών, είναι πιθανό να παρατηρηθούν διαφορετικές αρχιτεκτονικές λύσεις, όπως αυτές που παρουσιάζονται στο παρακάτω σχήμα. Στις λύσεις αυτές εμφανίζονται επαναλαμβανόμενα στοιχεία, και ο Alexander ονόμασε τα κοινά αυτά στοιχεία μεταξύ διαφορετικών σχεδίων υψηλής ποιότητας, πρότυπα. Η έννοια του προτύπου ορίστηκε ως "μια λύση ενός προβλήματος μέσα σε συγκεκριμένο πλαίσιο" σημειώνοντας ότι "κάθε πρότυπο περιγράφει ένα πρόβλημα που εμφανίζεται διαρκώς στο περιβάλλον και στη συνέχεια περιγράφει τον πυρήνα της λύσης κατά τέτοιο τρόπο ώστε η λύση να μπορεί να εφαρμοστεί εκατομμύρια φορές".

Στο συγκεκριμένο πρόβλημα διαπιστώνεται ότι αν οι θύρες σε ένα δωμάτιο προκαλούν "ανησυχία" λόγω της θέσης τους, τότε το δωμάτιο αυτό δεν θα είναι ποτέ άνετο για διαβίωση ή εργασία. Η λύση του συγκεκριμένου προτύπου επιβάλλει, "με εξαίρεση τους πολύ μεγάλους χώρους, την τοποθέτηση των θυρών όσο το δυνατόν πλησιέστερα προς τα άκρα του δωματίου".



Μέχρι στιγμής ίσως φαίνεται παράλογος ο συσχετισμός μεταξύ αρχιτεκτονικής και τεχνολογίας λογισμικού. Παρόλο που επαναλαμβανόμενες λύσεις χρησιμοποιούνταν στην ανάπτυξη λογισμικού από τις πρώτες ημέρες του προγραμματισμού, στις αρχές της δεκαετίας του 1990 τέθηκαν τα εξής δύο ερωτήματα

- αν υπάρχουν προβλήματα στο λογισμικό τα οποία επαναλαμβάνονται
- αν υπάρχει η δυνατότητα σχεδίασης λογισμικού αξιοποιώντας πρότυπα

Τα πρότυπα σχεδίασης έχουν γνωρίσει πολύ μεγάλη αποδοχή, παρόλη τη σχετικά σύντομη διάρκεια ζωής τους. Αυτό οφείλεται κυρίως στο ότι τα αποτελέσματα της ακαδημαϊκής έρευνας αλλά και η εφαρμογή τους στην πράξη απέδειξαν, πώς η συστηματική χρήση τους οδηγεί στην ανάπτυξη καλά δομημένου, συντηρήσιμου και επαναχρησιμοποιήσιμου λογισμικού.

Σε γενικές γραμμές μπορούμε να κατηγοριοποιήσουμε τα πρότυπα ανάλογα με τις λειτουργίες τους ως εξής:

Δομικά Πρότυπα:

- μειώνουν τη σύζευξη ανάμεσα σε δυο ή περισσότερες κλάσεις
- εισάγουν μία abstract κλάση για μελλοντικές επεκτάσεις
- εμπεριέχουν σύνθετες δομές

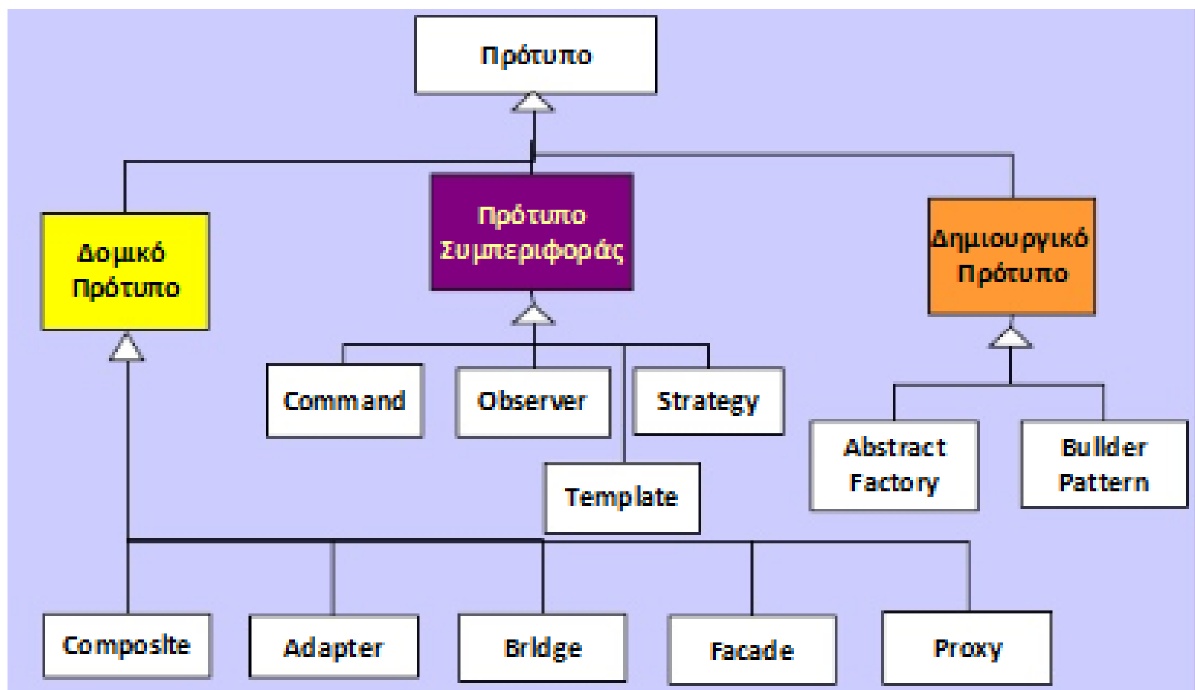
Πρότυπα Συμπεριφοράς:

- επιτρέπουν επιλογή ανάμεσα σε αλγορίθμους και ορισμό ευθυνών ανάμεσα σε αντικείμενα (ποιος κάνει τι;)
- χαρακτηρίζουν σύνθετες ροές ελέγχου οι οποίες είναι δύσκολο να παρακολουθηθούν κατά την εκτέλεση

Δημιουργικά Πρότυπα:

- επιτρέπουν την αφαίρεση από σύνθετες διαδικασίες αρχικοποίησης
- κάνουν το σύστημα ανεξάρτητο από τον τρόπο με τον οποίο δημιουργούνται, συνθέτονται και αναπαριστώνται τα αντικείμενα

Στο παρακάτω σχήμα βλέπουμε 11 πρότυπα κατηγοριοποιημένα σύμφωνα με τα παραπάνω.



Συνοπτικά τα πρότυπα σχεδίασης είναι σύνολα από κλάσεις (ή από αντικείμενα) που αλληλεπιδρούν μεταξύ τους, έτσι ώστε να χαρακτηρίζονται από μια συγκεκριμένη συμπεριφορά. Αποτελούν τρόπους για την περιγραφή άριστων πρακτικών και καλών προσεγγίσεων σχεδίασης και ενσωματώνουν (σχεδιαστική) πείρα με τέτοιο τρόπο ώστε αυτή να είναι δυνατό να επαναχρησιμοποιηθεί από άλλους.

2.1 Visitor - Επισκέπτης

Το πρότυπο σχεδίασης «Επισκέπτης»(Visitor) έχει ως στόχο την αναπαράσταση μίας λειτουργίας που πρόκειται να πραγματοποιηθεί στα στοιχεία μίας δομής αντικειμένων. Το πρότυπο επιτρέπει την προσθήκη μίας νέας λειτουργίας στις υπάρχουσες κλάσεις του συστήματος χωρίς όμως την τροποποίηση του κώδικα στις κλάσεις στις οποίες επιδρά. Έτσι νέες λειτουργικότητες μπορούν να προστεθούν εύκολα στην αρχική ιεραρχία, χωρίς την

τροποποίηση των ιδίων των κλάσεων, μόνο με τη δημιουργία μιας νέας υποκλάσης στο πρότυπο Επισκέπτης.

Το πρότυπο «Επισκέπτης» έχει αρκετά μεγάλη πολυπλοκότητα στη λειτουργία του καθώς υλοποιεί τη λεγόμενη διπλή αποστολή (double ή dual dispatch). Τα μηνύματα στον αντικειμενοστρεφή προγραμματισμό κατά κανόνα επιδεικνύουν συμπεριφορά που αντιστοιχεί στην απλή αποστολή, δηλαδή ότι η λειτουργία που εκτελείται εξαρτάται από το όνομα της αίτησης και τον τύπο του αποδεκτή. Στη διπλή αποστολή που αποβλέπει το πρότυπο «Επισκέπτης» η λειτουργία που εκτελείται εξαρτάται από το όνομα της αίτησης και τον τύπο δύο αποδεκτών (από τον τύπο του «Επισκέπτη» και από τον τύπο του στοιχείου που επισκέπτεται).

Η χρήση του συγκεκριμένου προτύπου σχεδίασης είναι ωφέλιμη κατά την προσθήκη λειτουργιών στα αντικείμενα μιας ιεραρχίας χωρίς να προκαλέσει «μόλυνση» στις κλάσεις τους. Το πρότυπο σχεδίασης «Επισκέπτης» επιτρέπει να διατηρούνται οι σχετιζόμενες λειτουργίες σε ένα μέρος, ορίζοντας αυτές σε μια νέα κλάση.

Γενικός σκοπός:

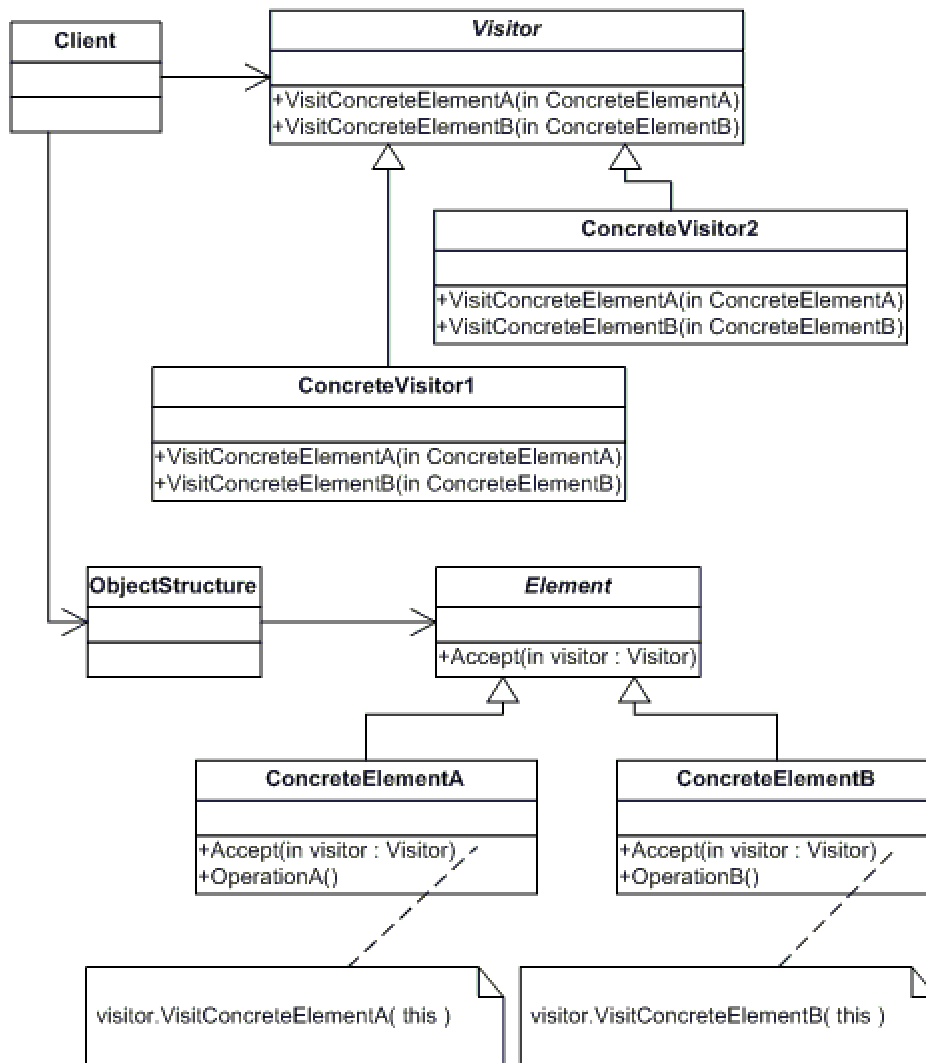
Να εξασφαλίσει έναν εύκολο τρόπο να εκτελούνται εντολές για μια οικογένεια κλάσεων. Συγκεντρώνει όλες τις συμπεριφορές και επιτρέπει αλλαγές σε αυτές χωρίς να μεταβάλλονται οι κλάσεις στις οποίες ενεργούν.

Εφαρμογές:

Το πρότυπο «Επισκέπτης» χρησιμοποιείται όταν:

- Ένα σύστημα περιέχει μία ομάδα σχετικών αντικειμένων.
- Συγκεκριμένες (μη τετριμμένες) ενέργειες πρέπει να εκτελεστούν σε μερικές από αυτές ή σε όλες τις κλάσεις.
- Οι ενέργειες αυτές δεν είναι ίδιες για όλες τις κλάσεις.

Υλοποίηση:



Πλεονεκτήματα:

- Λόγω της δομής του καθιστά την διαδικασία πρόσθεσης νέων συμπεριφορών στο σύστημα εύκολη.
- Τα «Visitors» είναι χρήσιμα γιατί συγκεντρώνουν ολόκληρο τον κώδικα, που αφορά μία δράση – ενέργεια (συμπεριφορά). Επίσης, το ίδιο αντικείμενο «Visitor» μπορεί να χρησιμοποιηθεί για να «επισκεφθεί» οποιοδήποτε «Element» σε μία δομή.

Μειονεκτήματα:

- Κάθε προσθήκη ή αλλαγή στην κλάση «Element» μπορεί να αποτελέσει αιτία αλλαγής της κλάσης «Visitor» με αποτέλεσμα πολλές φορές να πρέπει να ξαναγραφεί. Περαιτέρω, κάθε επιπλέον κλάση απαιτεί την προσθήκη μίας επιπλέον μεθόδου στο Interface «Visitor» και κάθε Concrete «Visitor» πρέπει να παρέχει μια υλοποίηση γι αυτήν την μέθοδο.
- Το πρότυπο «Επισκέπτης» αποκλίνει από μια βασική αρχή του αντικειμενοστρεφή προγραμματισμού (encapsulation principle), καθώς παίρνει κώδικα που χρησιμοποιεί ένα αντικείμενο (μιας κλάσης αντικειμένων) και το μεταφέρει σε ένα άλλο αντικείμενο.

2.2 Builder - Κατασκευαστής

Το πρότυπο «Κατασκευαστής» (Builder) χρησιμοποιείται για τη δημιουργία αντικειμένων. Σκοπός του είναι η κατασκευή αυτών των αντικειμένων σε «αφηρημένα» βήματα έτσι ώστε διαφορετικές εφαρμογές να μπορούν να κατασκευάσουν διαφορετικές αναπαραστάσεις των αντικειμένων από αυτά τα βήματα. Συχνά το χρησιμοποιούμε για την κατασκευή προϊόντων σε συμφωνία με το πρότυπο «Σύνθετο» (Composite).

Η κατασκευή τέτοιων σύνθετων αντικειμένων είναι συχνή. Για παράδειγμα θέλουμε να υπάρχει η δυνατότητα του μετασχηματισμού ενός εγγράφου σε άλλους τύπους. Τα βήματα εγγραφής του εγγράφου παραμένουν τα ίδια ενώ οι λεπτομέρειες κάθε βήματος εξαρτώνται από τον τύπο.

Προσέγγιση:

- Ο αλγόριθμος κατασκευής καθορίζεται από μια κλάση (τον «Director»)
- Τα αφηρημένα βήματα του αλγορίθμου (ένα για κάθε βήμα) καθορίζονται από μια διεπαφή (τον «Builder»)
- Κάθε αναπαράσταση παρέχει μια στιβαρή υλοποίηση της διεπαφής (οι «Concrete Builders»)

Γενικός σκοπός:

Η πρόθεση του προτύπου Builder είναι να διαχωρίσουμε την κατασκευή ενός σύνθετου αντικειμένου από την αναπαράσταση του. Με αυτόν τον τρόπο, η ίδια διαδικασία κατασκευής μπορεί να δημιουργήσει διαφορετικές αναπαραστάσεις.

Εφαρμογές:

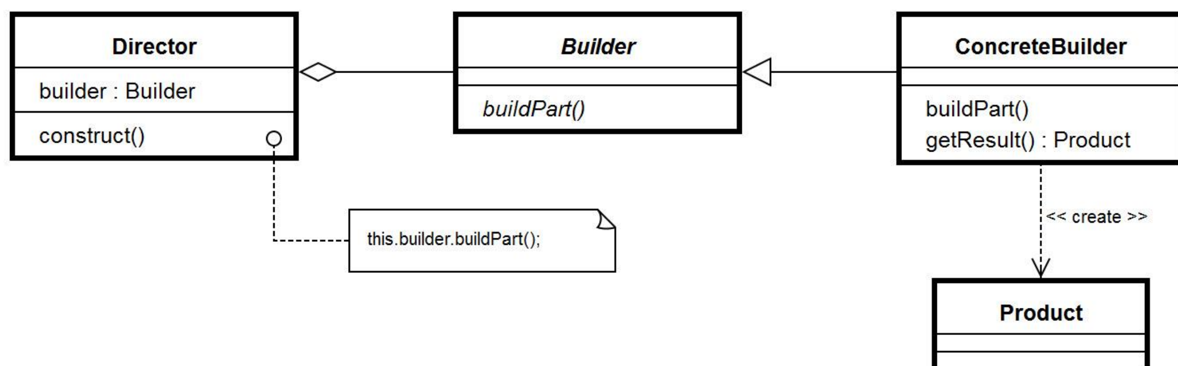
Η δημιουργία ενός πολύπλοκου προϊόντος πρέπει να είναι ανεξάρτητη των τμημάτων που αποτελούν το προϊόν. Συγκεκριμένα:

- Η διαδικασία δημιουργίας δεν πρέπει να γνωρίζει πληροφορίες για τη διαδικασία συναρμολόγησης
- Η διαδικασία δημιουργίας πρέπει να επιτρέπει διαφορετικές αναπαραστάσεις του αντικειμένου που κατασκευάζεται

Παράδειγμα:

- Ένα σπίτι με 1 όροφο, 3 δωμάτια, 2 διαδρόμους, 1γκαράζ και 3 πόρτες.
- Ένας ουρανοξύστης με 50 ορόφους, 15 γραφεία και 5 διαδρόμους σε κάθε όροφο.
- Η διαρρύθμιση των γραφείων διαφέρει ανά όροφο.

Υλοποίηση:



Χρήσιμες πληροφορίες:

- Το πρότυπο Builder επικεντρώνεται στην κατασκευή ενός πολύπλοκου αντικειμένου βήμα προς βήμα
- Συχνά χρησιμοποιείται για να «χτίσει» ένα πρότυπο Composite
- Προσφέρει μια πολύ άνετη διεπαφή
- Πιο ευέλικτο και πολύπλοκο από το πρότυπο Factory

2.3 Observer - Παρατηρητής

Το πρότυπο σχεδίασης «Παρατηρητής» (Observer) ορίζει μία σχέση εξάρτησης ένα – προς - πολλά μεταξύ αντικειμένων έτσι ώστε όταν μεταβάλλεται η κατάσταση ενός αντικειμένου, όλα τα εξαρτώμενα αντικείμενα να ενημερώνονται και να τροποποιούνται αυτόματα. Είναι επίσης γνωστό και ως πρότυπο «Δημοσίευση – Εγγραφή» (Publish-Subscribe).

Γενικός σκοπός:

Επειδή στον αντικειμενοστρεφή προγραμματισμό έχουμε ως στόχο τη δημιουργία ενός συνόλου αλληλεπιδρώντων αντικειμένων, ένα από τα συχνά προβλήματα που προκύπτουν είναι η αναγκαιότητα συνεργασίας μεταξύ των κλάσεων, γεγονός που οδηγεί σε υψηλή σύζευξη. Ορισμένα από τα πρότυπα σχεδίασης, όπως χαρακτηριστικά είναι το πρότυπο Observer, προσπαθούν να μειώσουν τη σύζευξη μεταξύ των αντικειμένων, παρέχοντας αυξημένη δυνατότητα επαναχρησιμοποίησης και τροποποίησης του συστήματος. Το συγκεκριμένο πρότυπο, επιτρέπει την αυτόματη ειδοποίηση και ενημέρωση ενός συνόλου αντικειμένων τα οποία "αναμένουν" ένα γεγονός, που εκδηλώνεται ως αλλαγή στην κατάσταση ενός αντικειμένου. Ο στόχος είναι η αποσύζευξη των παρατηρητών από το παρακολουθούμενο αντικείμενο (υποκείμενο), έτσι ώστε κάθε φορά που προστίθεται ένας νέος παρατηρητής (με διαφορετική διασύνδεση ενδεχομένως), να μην απαιτούνται αλλαγές στο παρακολουθούμενο αντικείμενο.

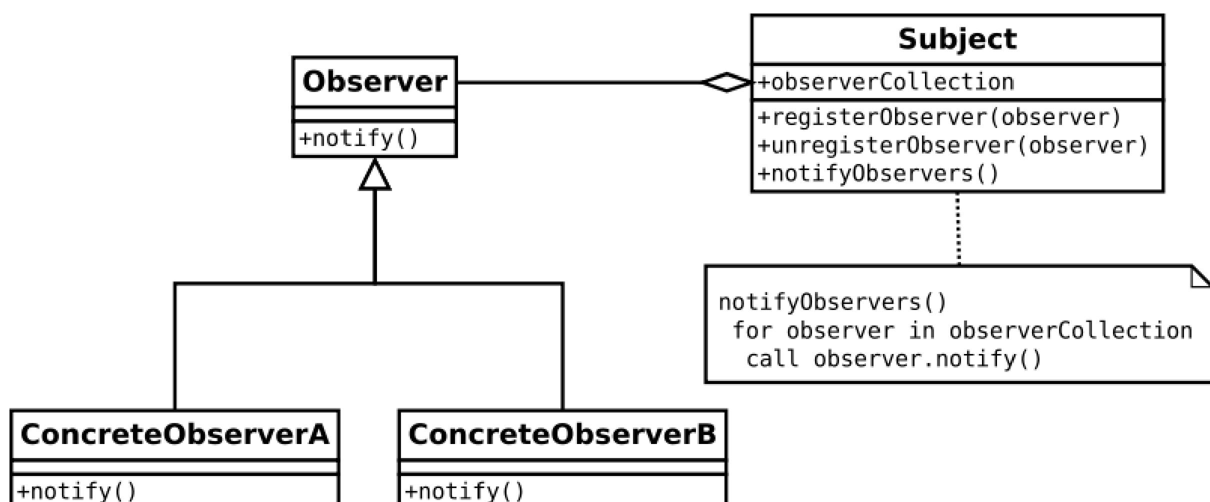
Η εφαρμογή του προτύπου προϋποθέτει τον εντοπισμό των εξής δύο τμημάτων: του υποκειμένου και του παρατηρητή. Μεταξύ των δύο τμημάτων υπάρχει μια σχέση ένα – προς – πολλά. Το υποκείμενο θεωρείται ότι διατηρεί το μοντέλο των δεδομένων και η

λειτουργικότητα που αφορά στην παρατήρηση των δεδομένων κατανέμεται σε διακριτά αντικείμενα-παρατηρητές. Οι παρατηρητές καταχωρούνται στο υποκείμενο κατά τη δημιουργία τους. Οποτεδήποτε αλλάζει, ''ανακοινώνει'' προς άλλους καταχωρημένους παρατηρητές το γεγονός της αλλαγής, και κάθε παρατηρητής ερωτά το υποκείμενο για το υποσύνολο της κατάστασης του υποκειμένου που το ενδιαφέρει. Το πρότυπο «Παρατηρητής» είναι από τα ευρέως χρησιμοποιούμενα και υλοποιείται με σχετική ευκολία σε διάφορες γλώσσες προγραμματισμού. Επιπλέον, εφαρμόζεται για χρόνια στην ευρέως γνωστή αρχιτεκτονική MVC (Model-View-Controller).

Εφαρμογές:

Επειδή η χρήση του προτύπου «Παρατηρητής» μπορεί να προσθέσει περιττή πολυπλοκότητα στο σύστημα, θα πρέπει να χρησιμοποιείται μόνο σε κατάλληλες περιπτώσεις. Πιο συγκεκριμένα, θα πρέπει να χρησιμοποιείται πρώτον, όταν η αλλαγή της κατάστασης ενός αντικειμένου (υποκειμένου) απαιτεί την ειδοποίηση άλλων αντικειμένων (παρατηρητών), αλλά το υποκείμενο δεν θα πρέπει να κάνει καμιά υπόθεση για το ποια είναι αυτά τα αντικείμενα. Δεύτερον, όταν η λίστα των παρατηρητών μπορεί να αλλάζει κατά τη διάρκεια εκτέλεσης του προγράμματος.

Υλοποίηση:



2.4 Proxy - Πληρεξούσιο

Όταν η πρόσβαση ενός εξωτερικού προγράμματος σε αντικείμενα κάποιας συγκεκριμένης κλάσης (έστω της Α) πρέπει να είναι ελεγχόμενη και να πληροί ορισμένες προϋποθέσεις (π.χ., για λόγους εξοικονόμησης μνήμης, να μη φορτώνεται πραγματικά η Α μέχρι να κληθεί μία μέθοδος της), το πρότυπο Proxy παρέχει έναν τυποποιημένο τρόπο ώστε αυτό να γίνεται διατηρώντας την κλειστότητα του εξωτερικού προγράμματος ως προς την εν λόγω κλάση και τον τρόπο λειτουργίας της, ακόμα και αν η υλοποίηση της αλλάξει, ο πελάτης είναι αδιάφορος απέναντι σε αυτές τις αλλαγές (δε χρειάζεται τροποποίηση) χάρη σε ένα επίπεδο αφαίρεσης που του παρέχεται από μία ενδιάμεση κλάση, τον Proxy, η οποία παίζει το ρόλο του μεσολαβητή πρόσβασης.

Ο Proxy είναι αυτός που εκτελεί όλους τους απαραίτητους ελέγχους και προσπελαύνει πραγματικά τα αντικείμενα, ενώ ταυτόχρονα υλοποιεί την ίδια διασύνδεση με την Α κι έτσι ο πελάτης, ο οποίος κατέχει μία αναφορά προς το στιγμιότυπο του Proxy αντί της Α, δεν αντιλαμβάνεται καν τη μεσολάβησή του. Ο Proxy ουσιαστικώς δίνει πρόσβαση στη ζητούμενη κλάση. Συνήθως κατασκευάζεται, σε αντίθεση με την Α, με πρότυπο «Factory» και εντελώς διαφανώς για το εξωτερικό πρόγραμμα, ενώ δεν είναι σπάνιο να περιέχει μία αναφορά στο πραγματικό στιγμιότυπο της Α ως ιδιωτικό πεδίο.

Γενικός σκοπός:

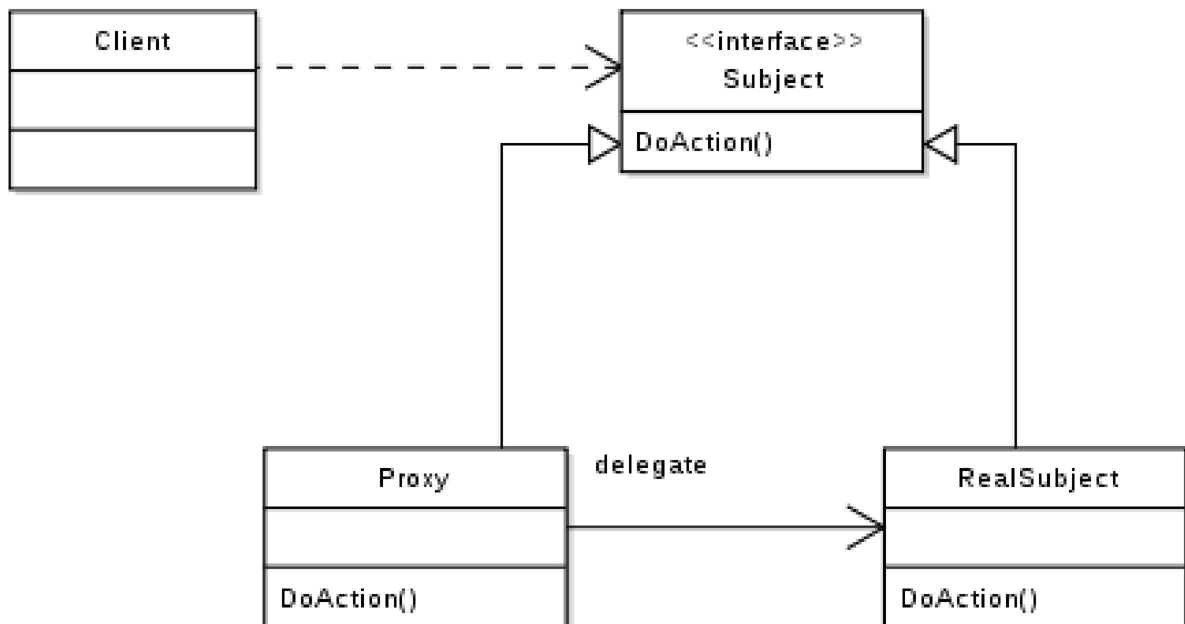
Το σχεδιαστικό αυτό πρότυπο αποτελεί ειδική εφαρμογή της γενικότερης έννοιας του «Proxy» στην πληροφορική. Με αυτήν τη γενικότερη έννοια ο «Proxy» μπορεί να μεσολαβεί μεταξύ ενός πελάτη και ενός οποιουδήποτε ακριβού, σπάνιου ή περίπλοκου πόρου. Μία ιδιαίτερη περίπτωση εφαρμογής της έννοιας του «Proxy» αποτελεί ο «απομακρυσμένος Proxy», όπως τα στελέχη (stubs) πελάτη και διακομιστή στο υπόδειγμα απομακρυσμένης κλήσης διαδικασιών στα συστήματα ενδιάμεσου λογισμικού.

Εφαρμογές:

Γενικά το συγκεκριμένο πρότυπο προτείνει μια λύση στο πρόβλημα της ανεξαρτητοποίησης του "Client" και των κλάσεων που χρησιμοποιεί, από τη διαδικασία πρόσβασης στα αντικείμενα αυτών των κλάσεων. Ο "Client" δεν έχει αναφορά στο πραγματικό αντικείμενο.

Αντίθετα έχει αναφορά σε ένα αντικείμενο που προσφέρει το ίδιο "interface" με το πραγματικό αντικείμενο και υλοποιεί διαδικασίες ελέγχου και πρόσβασης στο πραγματικό αντικείμενο.

Υλοποίηση:



Περιπτώσεις χρήσης:

Υπάρχουν τέσσερις κοινές περιπτώσεις κατά τις οποίες το πρότυπο «Proxy» είναι εφαρμόσιμο:

- *Εικονικό Proxy (Virtual Proxy)*: το πραγματικό αντικείμενο δημιουργείται μόνο όταν το ζητήσει/προσπελάσει ο πελάτης
- *Απομακρυσμένο Proxy (Remote Proxy)*: παρέχει μια τοπική προσέγγιση του αντικειμένου, ενώ αυτό βρίσκεται σε χώρο διαφορετικής διεύθυνσης
- *Προστατευτικό Proxy (Protective Proxy)*: ελέγχει την πρόσβαση σε ένα «ευαίσθητο» κύριο αντικείμενο
- *Έξυπνο Proxy (Smart Proxy)*: παρεμβάλει επιπλέον ενέργειες κατά την πρόσβαση ενός αντικειμένου

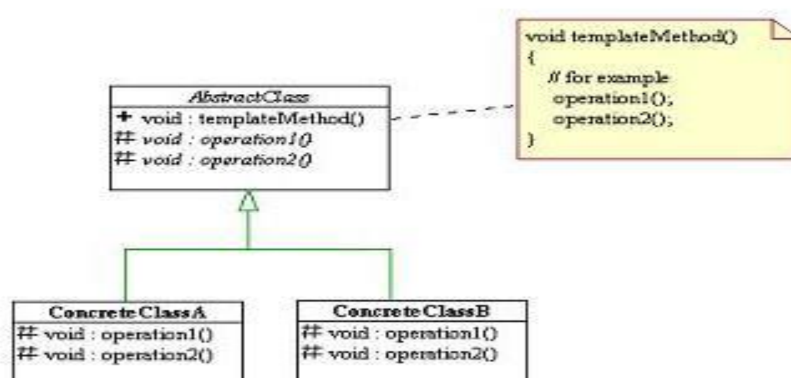
2.5 Template Method – Μέθοδος Υπόδειγμα

Το πρότυπο σχεδίασης "Μέθοδος Υπόδειγμα" (Template Method) ορίζει το περίγραμμα ενός αλγορίθμου σε μια λειτουργία, αφήνοντας ορισμένα βήματα στις παράγωγες κλάσεις. Το πρότυπο επιτρέπει στις παράγωγες κλάσεις να επαναορίσουν ορισμένα βήματα του αλγορίθμου χωρίς να αλλάξουν τη δομή του. Το πρότυπο επιλύει ένα παρόμοιο πρόβλημα όπως το πρότυπο "Στρατηγική". Στόχος είναι ο διαχωρισμός ενός γενικού αλγορίθμου από συγκεκριμένες υλοποιήσεις. Ωστόσο, ενώ το πρότυπο "Στρατηγική" αξιοποιεί τη δυνατότητα μεταφοράς μιας αρμοδιότητας σε ένα άλλο αντικείμενο μέσω της διαβίβασης μηνυμάτων (delegation), το πρότυπο "Μέθοδος Υπόδειγμα" εκμεταλλεύεται το μηχανισμό της κληρονομικότητας. Σημειώνεται ότι και αυτό το πρότυπο εφαρμόζεται πολύ συχνά από τους σχεδιαστές αντικειμενοστρεφούς λογισμικού, ακόμα και αν δεν γίνεται αντιληπτό ως ξεχωριστή τεχνική.

Γενικός σκοπός:

Στόχος του προτύπου είναι ο διαχωρισμός ενός γενικού αλγορίθμου από συγκεκριμένες υλοποιήσεις, καθώς και η πλήρης εκμετάλλευση του μηχανισμού της κληρονομικότητας. Το σχεδιαστικό πρότυπο «Μέθοδος Υπόδειγμα» χρησιμοποιείται για τον ορισμό των αμετάβλητων τμημάτων και τη μετάθεση της υλοποίησης των μεταβλητών τμημάτων του αλγορίθμου σε παραγωγές κλάσης.

Υλοποίηση:



Πλεονεκτήματα:

- Δεν επαναλαμβάνεται κώδικας μεταξύ των κλάσεων.
- Κληρονομικότητα και όχι Σύνθεση. Κάθε φορά που ένα πρότυπο σχεδίασης χρησιμοποιεί την κληρονομικότητα ως βασικό στοιχείο αντί της σύνθεσης, θα πρέπει να εξετάσουμε την αιτία. Επειδή το σχεδιαστικό πρότυπο «Μέθοδος Υπόδειγμα» χρησιμοποιεί κάποιες, αλλά όχι όλες τις μεθόδους που μπορούν να παρακαμφθούν για να επιτευχθεί η ευελιξία, ενσωματώνει ένα βασικό πλεονέκτημα της κληρονομικότητας αντί της Σύνθεσης.
- Με την αξιοποίηση του πολυμορφισμού, η υπερκλάση καλεί αυτόματα τις μεθόδους των σωστών υποκλάσεων.

Μειονεκτήματα:

- Ανεπαρκής επικοινωνία
- Δυσκολία να συνθέσει λειτουργικότητα
- Δύσκολό να κατανοήσει τη ροή του προγράμματος
- Δύσκολη συντήρηση

3 Πρότυπα Σχεδίασης Μηχανικής Παιχνιδιών

Τα πρότυπα μηχανικής παιχνιδιών εισήχθησαν αρχικά από τον Bjork ο οποίος παρουσιάσε μια ομάδα σχεδιαστικών προτύπων που ουσιαστικά, αποτελούν περιγραφές από συχνά εμφανιζόμενές αλληλεπιδραστικές διατάξεις που αφορούν την ιστορία των παιχνιδιού και τον τρόπο παιξίματος του. Αυτό συνεπάγεται ότι τα πρότυπα αυτά δεν σχετίζονται με την αρχιτεκτονική του λογισμικού και τον κώδικα. Τα προτεινόμενα πρότυπα συλλέχθηκαν από συνεντεύξεις προγραμματιστών που ασχολούνται επαγγελματικά με τα παιχνίδια, από ανάλυση υπάρχων παιχνιδιών και από μετασχηματισμό των μηχανισμών που συναντώνται στα παιχνίδια.

Η αναπαράσταση των προτύπων σχεδίασης παιχνιδιών περιλαμβάνει τέσσερα βασικά στοιχεία:

Όνομα : κάθε προτύπου σχεδίασης είναι ο τρόπος με τον οποίο μπορούμε να περιγράψουμε το πρόβλημα, τη λύση του και τις συνέπειες της, με μία ή δύο λέξεις. Δίνοντας ένα όνομα σε ένα πρότυπο σχεδίασης, άμεσα δημιουργούμε ένα εξειδικευμένο λεξικό (design vocabulary) που διευκολύνει την επικοινωνία μεταξύ των προγραμματιστών. Τα ονόματα θα πρέπει να είναι μνημονικά ώστε να μη πέσουμε στη παγίδα του «γνωστό και ως…» (also known as).

Το πρόβλημα: σε κάθε πρότυπο περιγράφει ένα γενικότερο πλαίσιο όπου υπό κανονική αντιμετώπιση – χωρίς τη χρήση προτύπων – θα προέκυπταν ανεπιθύμητες συνέπειες αναφορικά με την λειτουργία, τον έλεγχο και τη συντήρηση του λογισμικού. Μερικές φορές το πρόβλημα μπορεί να περιλαμβάνει κάποιες προϋποθέσεις που θα πρέπει να συναντώνται προκειμένου να έχει αποτέλεσμα η χρήση ενός προτύπου σχεδίασης.

Η λύση: περιγράφει τα στοιχεία από τα οποία πρέπει να συγκροτείται η σχεδίαση, τις μεταξύ τους σχέσεις, τις ιδιότητες και τις αρμοδιότητες αυτών. Ωστόσο, η λύση δεν περιγράφει μια συγκεκριμένη σχεδίαση ή υλοποίηση, επειδή το πρότυπο είναι σαν "πλαίσιο" και μπορεί να

εφαρμοστεί σε πολλές διαφορετικές περιπτώσεις. Περιγράφει μια πιο γενική λύση περικλείοντας όσο το δυνατόν περισσότερες από όλη την «οικογένεια» των (ίδιων) προβλημάτων.

Οι συνέπειες: είναι το αποτέλεσμα της εφαρμογής των προτύπων σχεδίασης. Αν και δεν αναφέρονται ρητά, όταν περιγράφουμε τρόπους σχεδίασης, είναι κρίσιμες για την εκτίμηση των εναλλακτικών που υπάρχουν στη σχεδίαση και για την κατανόηση του κόστους και της ωφέλειας από τη χρήση του προτύπου. Οι συνέπειες συχνά αφορούν ζητήματα χώρου μνήμης και χρόνου εκτέλεσης. Επιπλέον, επειδή η επαναχρησιμοποίηση ενός κώδικα είναι συχνά ένας παράγοντας που δηλώνει καλή αντικειμενοστραφή σχεδίαση, οι συνέπειες συχνά αναφέρονται στην επεκτασιμότητα και την ευελιξία του συστήματος.

Στη συνέχεια παρουσιάζονται ενδεικτικά πέντε πρότυπα μηχανικής παιχνιδιών.

3.1 Paper – Rock - Scissors

Πρόβλημα: Αποφυγή της στρατηγικής που κάνει τις αποφάσεις του παίκτη να φαίνονται ως ασήμαντες επιλογές (πατρονάρισμα). Με άλλα λόγια κάνει το παίκτη να αισθάνεται ότι οι αποφάσεις του είναι προφανείς.

Λύση: Παρουσιάζει μια μη-μεταβατική σχέση μεταξύ των εναλλακτικών επιλογών, όπως ακριβώς γίνεται στο γνωστό παιδικό παιχνίδι πέτρα-ψαλίδι-χαρτί.

Συνέπεια**:** Ο παίκτης δεν είναι πλέον σε θέση να βρει μια ενιαία στρατηγική η οποία θα είναι βέλτιστη σε όλες τις περιπτώσεις και κάτω από όλες τις συνθήκες. Θα πρέπει να επανεξετάσει τις αποφάσεις του, και ανάλογα με τους περιορισμούς που επιβάλλονται από το παιχνίδι, είτε να προσαρμοστεί στις μεταβαλλόμενες καταστάσεις, είτε να υποστεί τις συνέπειες της προηγούμενης απόφασής του.

Παραδείγματα: Ένα παράδειγμα δίνεται από τον Andrew Rollings στο παιχνίδι The Ancient Art of War (Broderbund 1984) αναφερόμενος στη σχέση πολεμιστή-βάρβαρου-τοξότη. Επίσης ένα ακόμη παράδειγμα βρίσκεται στο παιχνίδι Quake και στη σχέση μεταξύ των όπλων (ποιο όπλο κερδίζει ποιο).

Αναφορά: Ο Chris Crawford μας παρέχει μια πρώτη εκτενή περιγραφή ως αναφορά τη χρήση της μη-μεταβατικής σχέσης (Triangularity).

Γενικά: Η γενική φιλοσοφία του συγκεκριμένου προτύπου είναι βασισμένη στο εξής παράδειγμα: «Η πέτρα κερδίζει το ψαλίδι, το ψαλίδι κερδίζει το χαρτί, όμως η πέτρα ΔΕΝ κερδίζει (μεταβατικά) το χαρτί».

Το πρότυπο μπορεί να υλοποιηθεί με επιλογές που έχουν είτε άμεσες συνέπειες (όπως το παιχνίδι από το οποίο πήρε το όνομά του), είτε με μακροπρόθεσμες επιπτώσεις. Σε κάθε περίπτωση προωθεί την Ένταση, μέχρι τη στιγμή κατά την οποία αποκαλύπτονται οι επιλογές των παικτών ή μέχρι η επιτυχία/αποτυχία της επιλεγμένης στρατηγικής γίνει πια προφανής.

Το πρότυπο εισάγει την τυχαιότητα (Randomness) εκτός και αν οι παίκτες μπορούν να κερδίσουν γνώση για τις τρέχουσες επιλογές των άλλων παικτών ή να καταγράψουν τις συμπεριφορές τους. Διαφορετικά δεν υπάρχει τρόπος να προβλεφθεί μια συμφέρουσα τακτική. Όταν το παιχνίδι δίνει την δυνατότητα χρήσης σωστών στρατηγικών (βάση συλλογής γνώσεων), επιτρέπει στον παίκτη να γίνει «αυθεντία» (Game Mastery).

Είναι από τα πρώτα πρότυπα παιχνιδιών που έχουν ποτέ καταγραφεί και μπορεί να περιγραφεί και ως γενικό πρότυπο σχεδίασης. Σε περιπλοκότερο επίπεδο είναι μια περιγραφή ενός παραδείγματος με πάνω από τρεις σχέσεις η οποία όμως βασίζεται στην (ελάχιστη) βασική.

3.2 Analysis - Paralysis

Πρόβλημα: Όταν ένας παίκτης έρχεται αντιμέτωπος με πολλές δυνατότητες, οι διάφορες συνέπειες των επιλογών του τον επηρεάζουν αρνητικά. Όλες αυτές οι δυνατότητες δεν είναι επαρκείς για να έχει ο παίκτης την αίσθηση της εμβάθυνσης. Ο παίκτης πρέπει να έχει την αίσθηση ότι η ανάλυση μιας κατάστασης είναι εφικτή και όταν τη κάνει σωστά, αυτό θα του δώσει ένα πλεονέκτημα απέναντι στους άλλους παίκτες.

Λύση: Το πρότυπο ωθεί το παίκτη στο να ξοδέψει χρόνο για να αποφασίσει τι θα κάνει, αντί να αλληλεπιδράσει με το σύστημα. Αυτό τον οδηγεί στο να μαζέψει όλες εκείνες τις εμπειρίες που θα τον οδηγήσουν στο να γίνει «αυθεντία» (Master) στο συγκεκριμένο παιχνίδι και να του γίνει αντιληπτό ότι η νίκη του δεν προήλθε από καθαρή τύχη.

Συνέπεια: Το πρότυπο συνήθως αποφεύγεται από τους προγραμματιστές παιχνιδιών καθώς θεωρείται ότι είναι ένα σημάδι, πως το παιχνίδι δεν έχει το σωστό επίπεδο πολυπλοκότητας για τους παίκτες. Σε παιχνίδια με εναλλασσόμενη σειρά (Turn Taking), οδηγούμαστε αναγκαστικά σε μεγάλους χρόνους αναμονής για τους παίκτες που δεν είναι η σειρά τους. Αυτή η αρνητική συνέπεια μπορεί εύκολα να αποφευχθεί θέτοντας χρονικούς περιορισμούς (Time Limits).

Παραδείγματα: Θέτοντας όριο στο χρόνο τον οποίο υπάρχει η δυνατότητα να ολοκληρωθεί μία αποστολή, δίνεται η ανάγκη στο παίκτη να πάρει γρήγορα αποφάσεις, ενώ μια συνεχής κίνηση για παράδειγμα αναγκάζει το παίκτη να σταθμίζει συνεχώς τα οφέλη του κατά τη διάρκεια ενός συμβάντος. Το σκάκι μπορεί να χαρακτηριστεί ως ένα χαρακτηριστικό παράδειγμα των παιχνιδιών που ο παίκτης μπορεί να αναλύσει σχεδόν ασταμάτητα τις πιθανές μελλοντικές του ενέργειες. Αυτό επιτυγχάνεται με δέντρα αποφάσεων τα οποία αυξάνονται με γεωμετρική πρόοδο στην πάροδο του χρόνου παιχνιδιού.

Γενικά: Το πρότυπο μπορεί να χρησιμοποιηθεί όχι μόνο σε παιχνίδια για πολλούς παίκτες αλλά και σε οποιοδήποτε παιχνίδι, όπου ο παίκτης έχει μια επιλογή (η κατάσταση του

παιχνιδιού έχει παγώσει μέχρι να γίνει αυτή η επιλογή) για να σχεδιάσει τις μελλοντικές κινήσεις του με σκοπό να αυξήσει την πιθανότητα να κερδίσει.

3.3 Collision Detection

Πρόβλημα: Πολλά παιχνίδια (και ειδικά τα arcade) πρέπει να χειριστεί διάφορες συγκρούσεις μεταξύ αντικειμένων. Με απλά λόγια, θα πρέπει να έχουν τη δυνατότητα να ανιχνεύσουν όταν δύο αντικείμενα συγκρούονται μεταξύ τους.

Λύση: Δεδομένου ότι δεν έχουμε πραγματική φυσική για να καθορίσει πότε θα υπάρξει μια σύγκρουση, όπως έχουμε στην πραγματική ζωή, πρέπει να δημιουργήσουμε ένα σύνολο από κανόνες για το πότε μια σύγκρουση έχει συμβεί μετά την εφαρμογή των εν λόγω κανόνων στον κώδικα μας. Ένας τρόπος για να χειριστούμε την ανίχνευση σύγκρουσης είναι να δούμε αν συγκρούονται δύο δυνητικά αντικείμενα τέμνονται καθόλου μέσα στη πίστα. Δηλαδή, εάν ο ένας επικαλύπτει τον άλλο.

Παραδείγματα**:** Η ανίχνευση σύγκρουσης είναι ένα βασικό στοιχείο των παιχνιδιού. Είναι ένα από τα κύρια μέσα για τον προσδιορισμό του πως και πότε τα αντικείμενα σε ένα παιχνίδι πρέπει να αλληλεπιδρούν. Στο παιχνίδι Asteroids, υπάρχουν συγκρούσεις μεταξύ του σκάφους του παίκτη και αυτών των εξωγήινων. Υπάρχουν συγκρούσεις μεταξύ πυραύλων του παίκτη προς τα εξωγήινα σκάφοι ενώ υπάρχουν και συγκρούσεις μεταξύ των πυραύλων των εξωγήινων προς το σκάφος του παίκτη. Κάθε ένα από αυτά είναι ζωτικής σημασίας για το παιχνίδι.

3.4 Shared Rewards

Πρόβλημα: Οι παίκτες που έχουν εμπλακεί με κάποιο τρόπο στην επίτευξη ενός στόχου του παιχνιδιού, μοιράζονται συνήθως κάποια μορφή ανταμοιβής. Δεν είναι απαραίτητο για όλους τους παίκτες που μοιράζονται τα οφέλη να συνεργαστούν πραγματικά καθώς σε ορισμένες

περιπτώσεις, οι παίκτες μπορούν για παράδειγμα να στοιχηματίσουν για το αποτέλεσμα μιας ομάδας σε κάποια περιπέτεια.

Λύση: Η χρήση του προτύπου εξαρτάται σε μεγάλο βαθμό από το αν το μέγεθος και η κατανομή της ανταμοιβής του έχει οριστεί στην αρχή ή κατά τη διάρκεια του παιχνιδιού. Εάν ένας παίκτης παίρνει την ίδια ποσοτική επιβράβευση, ανεξάρτητα από τον αριθμό των άλλων παικτών που μοιράζονται την ανταμοιβή ο παίκτης δεν χάνει τίποτα με το να συνεργαστεί. Αυτό, ωστόσο, απαιτεί Απεριόριστο χώρο συγκέντρωσης πόρων, τουλάχιστον κατά τη διάρκεια της πραγματικής κατανομής της ανταμοιβή και μπορεί να οδηγήσει σε ένα είδος πληθωρισμού.

Εάν το μέγεθος της αμοιβής καθορίζεται (Περιορισμένη συγκέντρωση πόρων), οι παίκτες έχουν να ανταγωνιστούν ένας ενάντια στον άλλο για το ποιος θα λάβει το μεγαλύτερο δυνατό μερίδιο. Το πώς ορίζεται η διανομή επηρεάζει σε μεγάλο βαθμό το παιχνίδι για αυτά τα είδη των ανταμοιβών: μια first-come-first-served (ο πρώτος «πληρώνεται» πρώτος) λειτουργία δημιουργεί μια κούρσα μεταξύ των παικτών, ενώ η κατανομή τυχαίων επιβραβεύσεων προωθεί τις επενδύσεις. Στις προκαθορισμένες σχέσεις μεταξύ μεριδίων και ανταμοιβών, ενθαρρύνεται ο ανταγωνισμός κατά τη διανομή της ανταμοιβής.

Οι παίκτες λαμβάνουν τα μερίδια μιας ανταμοιβής, είτε αξιοκρατικά (by merit), είτε με σύμβαση παραχώρησης. Για τους παίκτες που λαμβάνουν μερίδιο αξιοκρατικά απλά σημαίνει ότι θα πρέπει να πληρούν ορισμένα κριτήρια για να το δικαιούνται. Μερίδιο από παραχώρηση παίρνουν όταν ένας άλλος παίκτης παραχωρεί μέρος του δικού του μεριδίου.

Συνέπεια: Το «Shared Rewards» ανοίγει μια σειρά από δυνατότητες για την αλληλεπίδραση μεταξύ των παικτών. Η κοινή ανταμοιβή δίνει στους παίκτες ένα κοινό στόχο και προωθεί την ώθηση της κοινωνικής αλληλεπίδρασης.

Στην περίπτωση όπου η ανταμοιβή θα πρέπει να μοιραστεί μεταξύ των παικτών το πρότυπο ενθαρρύνει τον ανταγωνισμός μεταξύ των παικτών. Επιπλέον, όταν τα οφέλη μπορούν να μοιραστούν μεταξύ ανταγωνιστών η ανταμοιβή δεν ωφελεί μόνο έναν παίκτη. Αυτό είναι ιδιαίτερα αληθές σε καταστάσεις όπου οι παίκτες πρέπει να επιλέξουν με ποιον να μοιραστούν το βραβείο.

Παραδείγματα: Για παράδειγμα, το επιτραπέζιο παιχνίδι Kohle, Kies & Knete (Sid Sackson, Schmidt Spiele 1994) είναι ένα παιχνίδι στο οποίο κάνοντας deals (συμφωνίες) δίνετε ένα ορισμένο ποσό χρημάτων. Ωστόσο, deals σπάνια μπορούν να πραγματοποιηθούν από ένα μόνο παίκτη και ο κύριος παίκτης του deal θα πρέπει να ζητήσει βοήθεια από τους άλλους παίκτες, με αντάλλαγμα ένα κομμάτι αυτής.

Γενικά: Το πραγματικό περιεχόμενο του Shared Reward μπορεί να είναι αποτελέσματα της διαδικασίας ασυμφωνίας μεταξύ παικτών. Ένα παράδειγμα αυτού είναι to Trading, στο οποίο οι παίκτες που συμμετέχουν σε μια ανταλλαγή (trade) , πλεονεκτούν σε σχέση με εκείνους που δε συμμετείχαν.

Τέλος, στους κανόνες ανταλλαγής που επιτρέπουν την μπλόφα, το πραγματικό περιεχόμενο του Shared Reward είναι αβέβαιο δημιουργώντας Uncertain Shared Reward

3.5 Game Loop

Πρόβλημα: Στο πρότυπο Game Loop αρχικά προσδιορίστηκε από τη μελέτη περίπτωσης, ένα αντικείμενο ελέγχου (controller object) με ένα "while (!game is over)" βρόχο ο οποίος παρακολουθεί συνεχώς τις κινήσεις των παικτών και στέλνει μηνύματα σε sprites, ζητώντας από αυτά να ενημερώσουν τη κατάστασή τους αναλόγως.

Λύση: Με τη συγκέντρωση όλου του κώδικα εισόδου-επεξεργασίας και τη «χαρτογράφηση» των μηνυμάτων sprite σε ένα ενιαίο αντικείμενο ελεγκτή (controller object), το πρότυπο καθιστά εύκολο το να βρεθεί και να διατηρηθεί αυτός ο κώδικας αργότερα.

Παραδείγματα**:** Αρκετά περίπλοκα παιχνίδια εφαρμόζουν αυτό το πρότυπο, κάνοντας το Stage sprite να δουλεύει ως Ελεγκτής. Το Stage είναι ένα Scratch αντικείμενο που δουλεύει ως ένα ψεύτικο λευκό φόντο σε όλα τα πρότζεκτ. Όπως και τα άλλα sprites, το Stage μπορεί να έχει ένα σενάριο. Μέσα από αυτό το σενάριο, ο προγραμματιστής μπορεί να τοποθετήσει μια "επανάληψη μέχρι το παιχνίδι τελειώνει" ως βρόγχο, και στο εσωτερικό του βρόγχου,

κώδικα που να μπορεί να διαβάσει τις εισαγόμενες πληροφορίες, να μπορεί να ορίσει γενικές μεταβλητές, και στη συνέχεια να στείλει μηνύματα σε όλα τα sprites.

Για παράδειγμα, όταν ένα συγκεκριμένο Stage του παιχνιδιού λαμβάνει από το πληκτρολόγιο του χρήστη τη κίνηση ενός χαρακτήρα του παιχνιδιού, στέλνει σε όλα τα απαραίτητα sprites ένα μήνυμα ώστε να προχωρήσουν αναλόγως το χαρακτήρα.

Γενικά: Σε γενικές γραμμές το Game Loop είναι ένα απλό αρχιτεκτονικό πρότυπο για την οργάνωση του κώδικα ενός παιχνιδιού. Το κύριο χαρακτηριστικό του είναι ένας κύκλος ενημέρωσης της κατάσταση του παιχνιδιού, η αλλαγή αυτής της κατάστασης, και στη συνέχεια η αναμονή για ένα μικρό χρονικό διάστημα μέχρι να έρθει η ώρα να πάει και πάλι στη ζητούμενη κατάσταση.

Μπορείτε να σκεφτείτε το πρότυπο σαν ένα καρδιακό παλμό του παιχνιδιού σας που πιέζεται συνεχώς μέσω του κύκλου σε τακτά χρονικά διαστήματα, μέχρι το παιχνίδι να τελειώνει. Όπως και τα περισσότερα πρότυπα σχεδίασης, έτσι και το Game Loop είναι μια εννοιολογική ιδέα που μπορεί να εφαρμοστεί σε πολλές περιπτώσεις. Σχεδόν σε κάθε παιχνίδι θα βρείτε κάποια εφαρμογή του. Σε υψηλό επίπεδο, μπορείτε να το σκεφτείτε σαν να σπάτε το παιχνίδι σας σε πέντε στάδια: την εκκίνηση, την ενημέρωση, την σχεδίαση, την αναμονή, και τον καθαρισμό.

4 Υλοποίηση Προτύπων Μηχανικής Παιχνιδιών με Αντικειμενοστρεφή Πρότυπα

Στο συγκεκριμένο κεφάλαιο παρουσιάζεται ο τρόπος με τον οποίο θα μπορούσαν να υλοποιηθούν τα πρότυπα μηχανικής παιχνιδιών μέσω αντικειμενοστρεφών προτύπων σχεδίασης με στόχο τη βελτίωση της σχεδιαστικής ποιότητας των παιχνιδιών. Σε κάθε μία από τις ενότητες παρουσιάζεται το σχέδιο και η υλοποίηση ενός προτύπου μηχανικής παιχνιδιών μέσω ενός αντικειμενοστρεφούς προτύπου σχεδίασης.

4.1 BattleTest (Paper-Rock-Scissors με Visitor)

Στο συγκεκριμένο παιχνίδι ο παίκτης έχει τη δυνατότητα να διαλέξει ένα από τα τέσσερα διαφορετικά είδη όπλων (σπαθί, τόξο, λόγχη και μαχαίρι) και να μονομαχήσει με άλλα για να βγει νικητής και να «ανεβάσει» το επίπεδο του όπλου του.

Το πρότυπο μηχανικής παιχνιδιών «Paper - Rock - Scissors» εντοπίζεται στη σχέση των όπλων μεταξύ τους, και πιο συγκεκριμένα στο ποιο νικάει ποιο, χωρίς να υπάρχει η μεταβατική ιδιότητα όπως αναφέρθηκε και στη περιγραφή του προτύπου. Παρακάτω φαίνονται οι σχέσεις των όπλων:

Bow < Sword

Bow > Spear

Bow < Knife

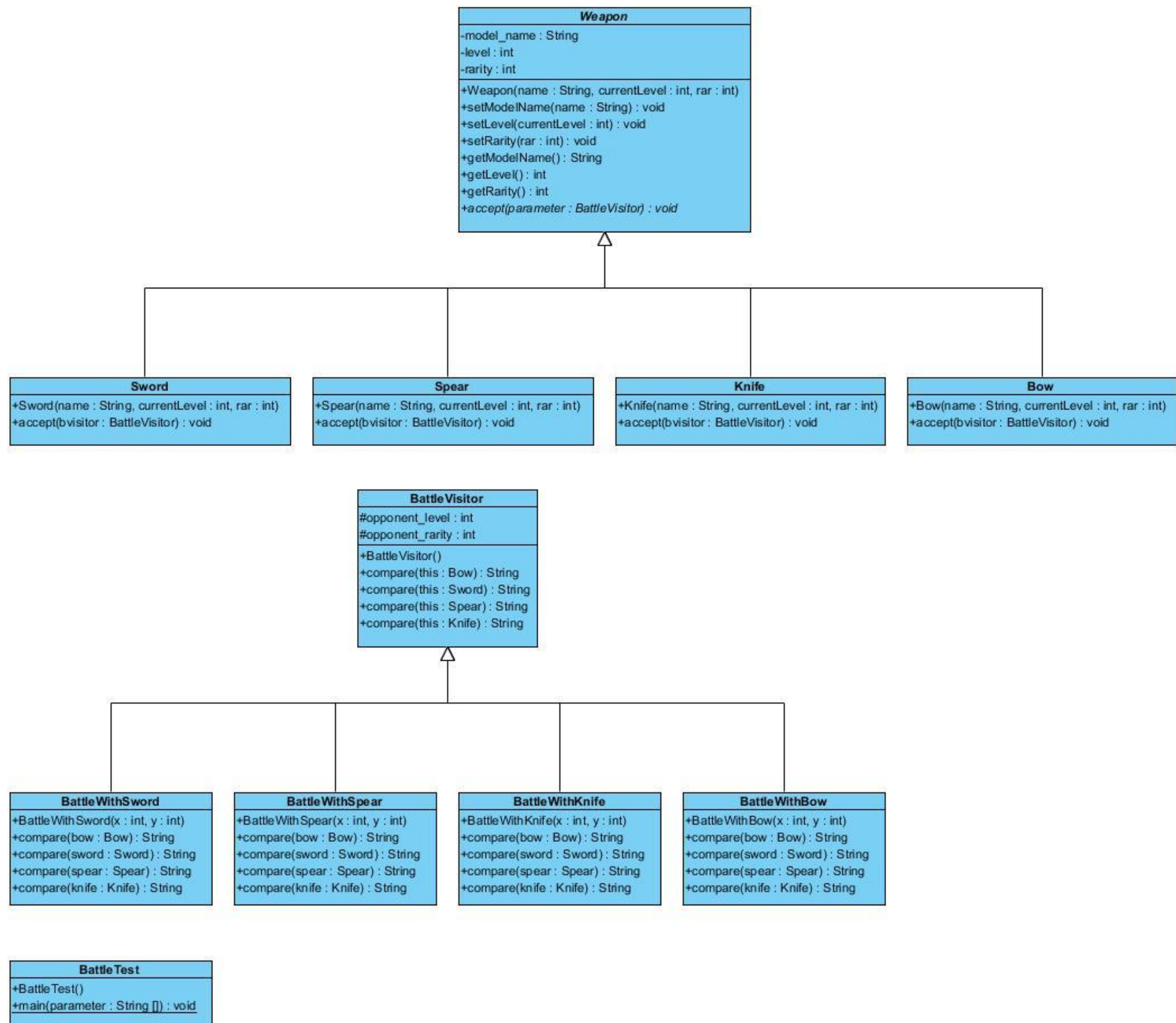
Sword < Spear

Sword > Knife

Spear > Knife

Φυσικά αν ένα όπλο είναι πολύ μεγαλύτερου επιπέδου από το άλλο, κερδίζει ανεξαρτήτως των παραπάνω.

<u>UML Diagram</u>:



<u>Κώδικας</u>:

```
public abstract class Weapon {
	private String model_name;//Το όνομα του μοντέλου
	private int level;//Το επίπεδο δυνατοτήτων του όπλου
	private int rarity;//Η σπανιότητα του όπλου με το 0 να δηλώνει το "συνηθισμένο"

	public Weapon(String name, int currentLevel, int rar){
		model_name=name;
		level=currentLevel;
		rarity=rar;
	}

	public void setModelName(String name){ model_name=name; }
```

```
	public void setLevel(int currentLevel) { level=currentLevel; }
	public void setRarity(int rar) { rarity=rar; }
	public String getModelName(){ return model_name;}
	public int getLevel(){ return level;}
	public int getRarity(){ return rarity;}

	public void weaponInfo(){
		System.out.println(	"---------------------\nMy	weapon
Info:\nModel Name:" + model_name +
					"\nWeapon Level:" + level +
					"\nWeapon Rarity:" + rarity +
					"\n---------------------");
	}

	public abstract String accept(BattleVisitor bvisitor);
}

public class Sword  extends Weapon{
	public Sword(String name, int currentLevel, int rar){
		super(name,currentLevel,rar);
	}

	public	String	accept(BattleVisitor	bvisitor)	{return
bvisitor.compare(this);}
}

public class Spear extends Weapon{
	public Spear(String name, int currentLevel, int rar){
		super(name,currentLevel,rar);
	}

	public	String	accept(BattleVisitor	bvisitor)	{return
bvisitor.compare(this);}
}

public class Knife extends Weapon{
	public Knife(String name, int currentLevel, int rar){
		super(name,currentLevel,rar);
	}
```

```
	public String accept(BattleVisitor bvisitor) {return bvisitor.compare(this);}
}

public class Bow extends Weapon{
	public Bow(String name, int currentLevel, int rar){
		super(name,currentLevel,rar);
	}

	public String accept(BattleVisitor bvisitor) {return bvisitor.compare(this);}
}

public interface BattleVisitor {
	public String compare(Bow bow);
	public String compare(Sword sword);
	public String compare(Spear spear);
	public String compare(Knife knife);
}

//κλάση BattleWithBow (μάχη όπλου με τόξο)
public class BattleWithBow implements BattleVisitor{

	protected int opponent_level;
	protected int opponent_rarity;

	public BattleWithBow(int x, int y) {
		this.opponent_level = x;
		this.opponent_rarity = y;
	}

	//Ισοπαλία Bow(mine)=Bow(opponent)
	public String compare(Bow bow){
		//Ίδια όπλα, οπότε θα λύσουν τις διαφορές του αρχικά στο Level και σε περίπτωση και εκεί ισοπαλίας
		//θα λύσουν τις διαφορές τους στο rarity (αν είναι ολα ίδια κερδίζει το δικό μου όπλο)
		if(bow.getLevel() > opponent_level){
			bow.setLevel(bow.getLevel() + 1);//Μετά τη νίκη το όπλο ανεβαίνει επίπεδο
			return "wins";
```

```
        }else if(bow.getLevel() == opponent_level){
            if(bow.getRarity() >= opponent_rarity){
                bow.setLevel(bow.getLevel() + 1);//Μετά τη νίκη το
όπλο ανεβαίνει επίπεδο
                return "wins";
            }else{
                return "loses";
            }
        }else{
            return "loses";
        }
    }

    //Νίκη Sword(mine)>Bow(opponent)
    public String compare(Sword sword){
        //Μόνο αν το όπλο μου είναι 20 Level χειρότερο χάνει, σε κάθε
άλλη περίπτωση κερδίζει
        if(opponent_level - sword.getLevel() >= 20){
            return "loses";
        }else{
            sword.setLevel(sword.getLevel() + 1);//Μετά τη νίκη το
όπλο ανεβαίνει επίπεδο
            return "wins";
        }
    }

    //Ήττα Spear(mine)<Bow(opponent)
    public String compare(Spear spear){
        //Μόνο αν το όπλο μου είναι 20 Level καλύτερο κερδίζει, σε κάθε
άλλη περίπτωση χάνει
        if(spear.getLevel() - opponent_level >= 20){
            spear.setLevel(spear.getLevel() + 1);//Μετά τη νίκη το
όπλο ανεβαίνει επίπεδο
            return "wins";
        }else{
            return "loses";
        }
    }

    //Νίκη Sword(mine)>Bow(opponent)
    public String compare(Knife knife){
```

```
			//Μόνο αν το όπλο μου είναι 20 Level χειρότερο χάνει, σε κάθε
άλλη περίπτωση κερδίζει
			if(opponent_level - knife.getLevel() >= 20){
				return "loses";
			}else{
				knife.setLevel(knife.getLevel() + 1);//Μετά τη νίκη το
όπλο ανεβαίνει επίπεδο
				return "wins";
			}
		}
}

//κλάση BattleWithKnife (μάχη όπλου με μαχαίρι)
public class BattleWithKnife implements BattleVisitor{

		protected int opponent_level;
		protected int opponent_rarity;

		public BattleWithKnife(int x, int y) {
			this.opponent_level = x;
			this.opponent_rarity = y;
		}

		public String compare(Bow bow){
			if(bow.getLevel() - opponent_level >= 20){
				bow.setLevel(bow.getLevel() + 1);
				return "wins";
			}else{
				return "loses";
			}
		}
		public String compare(Sword sword){
			if(opponent_level - sword.getLevel() >= 20){
				return "loses";
			}else{
				sword.setLevel(sword.getLevel() + 1);
				return "wins";
			}
		}
		public String compare(Spear spear){
```

```java
        if(spear.getLevel() - opponent_level >= 20){
            spear.setLevel(spear.getLevel() + 1);
            return "wins";
        }else{
            return "loses";
        }
    }
    public String compare(Knife knife){
        if(knife.getLevel() > opponent_level){
            knife.setLevel(knife.getLevel() + 1);
            return "wins";
        }else if(knife.getLevel() == opponent_level){
            if(knife.getRarity() >= opponent_rarity){
                knife.setLevel(knife.getLevel() + 1);
                return "wins";
            }else{
                return "loses";
            }
        }else{
            return "loses";
        }
    }
}

//κλάση BattleWithSpear (μάχη όπλου με λόγχη)
public class BattleWithSpear implements BattleVisitor{

    protected int opponent_level;
    protected int opponent_rarity;

    public BattleWithSpear(int x, int y) {
        this.opponent_level = x;
        this.opponent_rarity = y;
    }
    public String compare(Bow bow){
        if(opponent_level - bow.getLevel() >= 20){
            return "loses";
        }else{
            bow.setLevel(bow.getLevel() + 1);
```

```
				return "wins";
			}
		}
		public String compare(Sword sword){
			if(sword.getLevel() - opponent_level >= 20){
				sword.setLevel(sword.getLevel() + 1);
				return "wins";
			}else{
				return "loses";
			}
		}
		public String compare(Spear spear){
			if(spear.getLevel() > opponent_level){
				spear.setLevel(spear.getLevel() + 1);
				return "wins";
			}else if(spear.getLevel() == opponent_level){
				if(spear.getRarity() >= opponent_rarity){
					spear.setLevel(spear.getLevel() + 1);
					return "wins";
				}else{
					return "loses";
				}
			}else{
				return "loses";
			}
		}
		public String compare(Knife knife){
			if(knife.getLevel() - opponent_level >= 20){
				knife.setLevel(knife.getLevel() + 1);
				return "wins";
			}else{
				return "loses";
			}
		}
}

//κλάση BattleWithSword (μάχη όπλου με σπαθί)
public class BattleWithSword implements BattleVisitor {
```

```java
protected int opponent_level;
protected int opponent_rarity;

public BattleWithSword(int x, int y) {
	this.opponent_level = x;
	this.opponent_rarity = y;
}

public String compare(Bow bow){
	if((bow.getLevel() - opponent_level) >= 20){
		bow.setLevel(bow.getLevel() + 1);
		return "wins";
	}else{
		return "loses";
	}
}
public String compare(Sword sword){
	if(sword.getLevel() > opponent_level){
		sword.setLevel(sword.getLevel() + 1);
		return "wins";
	}else if(sword.getLevel() == opponent_level){
		if(sword.getRarity() >= opponent_rarity){
			sword.setLevel(sword.getLevel() + 1);
			return "wins";
		}else{
			return "loses";
		}
	}else{
		return "loses";
	}
}
public String compare(Spear spear){
	if(opponent_level - spear.getLevel() >= 20){
		return "loses";
	}else{
		spear.setLevel(spear.getLevel() + 1);
		return "wins";
	}
}
```

```
	public String compare(Knife knife){
		if(knife.getLevel() - opponent_level >= 20){
			knife.setLevel(knife.getLevel() + 1);
			return "wins";
		}else{
			return "loses";
		}
	}
}

public class BattleTest {
	public static void main(String[] args) {

				//Δημιουργία ενός σπαθιού
				Weapon w = new Sword("Excalibur",1,100);
				System.out.println("Weapon created...");



				//Προβολή πληροφοριών του όπλου μας
				w.weaponInfo();

				//Μονομαχία του όπλου μας με ένα τόξο επιπέδου 5
και σπανιότητας 10
				BattleVisitor b1 = new BattleWithBow(5,10);
				System.out.println("My " + w.getModelName() + " vs
" + "a bow, outcome: " + w.accept(b1));

				//Μονομαχία του όπλου μας με ένα σπαθί επιπέδου 3
και σπανιότητας 5
				BattleVisitor b2 = new BattleWithSword(3,5);
				System.out.println("My " + w.getModelName() + " vs
" + "a sword, outcome: " + w.accept(b2));

				//Μονομαχία του όπλου μας με ένα μαχαίρι επιπέδου
22 και σπανιότητας 1
				BattleVisitor b3 = new BattleWithKnife(32,1);
				System.out.println("My " + w.getModelName() + " vs
" + "a knife, outcome: " + w.accept(b3));

				//Μονομαχία του όπλου μας με μία λόγχη επιπέδου 5
και σπανιότητας 17
				BattleVisitor b4 = new BattleWithSpear(5,17);
```

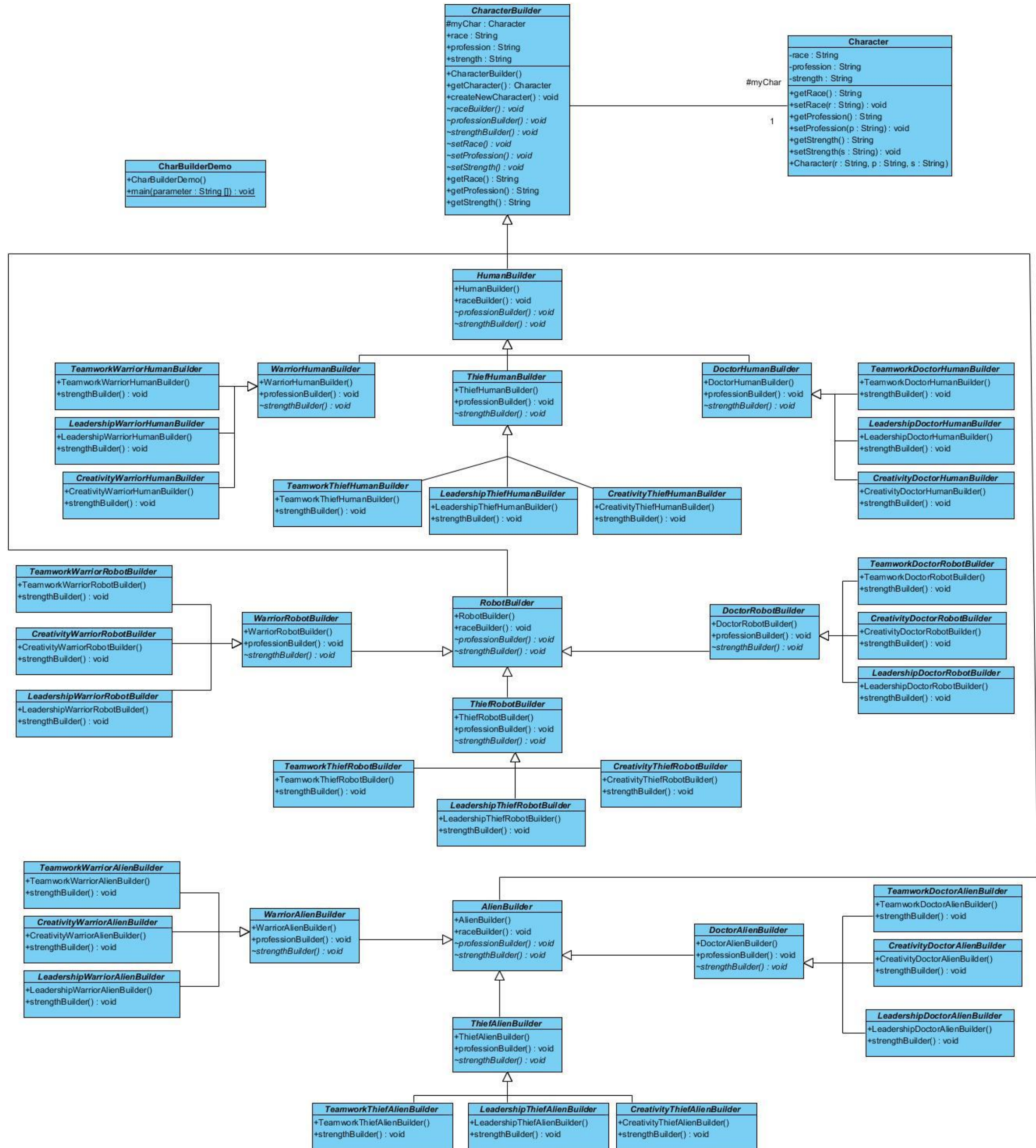
```
                        System.out.println("My " + w.getModelName() + " vs
" + "a spear, outcome: " + w.accept(b4));

                        //Προβολή πληροφοριών του όπλου μας μετά τις
παραπάνω μάχες
                        w.weaponInfo();

      }
}
```

4.2 CharBuilderDemo (Analysis – Paralysis με Builder)

Στο συγκεκριμένο παιχνίδι ο παίκτης δημιουργεί ένα χαρακτήρα επιλέγοντας τα χαρακτηριστικά του. Μ' αυτό τον τρόπο του δίνεται η δυνατότητα να έχει την αίσθηση της εμβάθυνσης στο «χτίσιμο» του χαρακτήρα του αλλά και στο μέλλον του παιχνιδιού, καθώς αν διαλέξει τα χαρακτηριστικά της αρεσκείας του, θα μπορεί να τα εφαρμόσει παίζοντας. Έτσι φαίνεται ξεκάθαρα το πρότυπο μηχανικής παιχνιδιών «Analysis – Paralysis».

Οι επιλογές του είναι σε τρεις κατηγορίες: Race (φυλή), Profession(ειδικότητα) και Strength (δύναμη) με αντίστοιχες επιλογές (human-alien-robot, warrior-thief-engineer, leadership-creativity-teamwork).

UML Diagram:



Κώδικας:

```
public class CharBuilderDemo {

	public static void main(String[] args) {
		Character myCharacter1 = new Character("","","");
		CharacterBuilder cb1 = new    CreativityWarriorHumanBuilder();
```

```
		Character myCharacter2 = new Character("","","");
		CharacterBuilder cb2 = new    LeadershipDoctorAlienBuilder();

		Character myCharacter3 = new Character("","","");
		CharacterBuilder cb3 = new    TeamworkThiefRobotBuilder();

		myCharacter1 = cb1.getCharacter();
		myCharacter2 = cb2.getCharacter();
		myCharacter3 = cb3.getCharacter();

		System.out.println(myCharacter1.CharInfo());
		System.out.println(myCharacter2.CharInfo());
		System.out.println(myCharacter3.CharInfo());
	}
}

public abstract class CharacterBuilder {
	protected Character myChar;
	public String race;
	public String profession;
	public String strength;


	public CharacterBuilder(){
		this.createNewCharacter();
	}

	public Character getCharacter(){
		return myChar;
	}

	public void createNewCharacter(){
		myChar = new Character("","","");
		this.raceBuilder();
		this.professionBuilder();
		this.strengthBuilder();
	}

	public abstract void raceBuilder();
```

```
	public abstract void professionBuilder();

	public abstract void strengthBuilder();


	public void setRace(String race){
		this.race=race;
	}
	public void setProfession(String profession){
		this.profession = profession;
	}
	public void setStrength(String strength){
		this.strength = strength;
	}

	public String getRace(){
		return race;
	}
	public String getProfession(){
		return profession;
	}
	public String getStrength(){
		return strength;
	}
}

public class Character {
	private String race = "";
	private String profession = "";
	private String strength = "";

	public String getRace(){
		return race;
	}
	public void setRace(String r){
		race=r;
	}
	public String getProfession(){
```

```
		return profession;
	}
	public void setProfession(String p){
		profession=p;
	}
	public String getStrength(){
		return strength;
	}
	public void setStrength(String s){
		strength=s;
	}

	public Character(String r, String p, String s){
		race=r;
		profession=p;
		strength=s;
	}
	public String CharInfo(){
		return "-------------------------\n" +
				"My Character's Info:\n" + "Race:" + race +
				"\nProfession:" + profession +
				"\nStrength:" + strength;
	}
}


/////////////////////////////////////////////////////////////////
public abstract class HumanBuilder extends CharacterBuilder{
	public void raceBuilder() {
		myChar.setRace("human");
	}

	public abstract void professionBuilder();

	public abstract void strengthBuilder();
}
public abstract class AlienBuilder extends CharacterBuilder{
	public void raceBuilder() {
		myChar.setRace("alien");
```

```
	}

	public abstract void professionBuilder();

	public abstract void strengthBuilder();
}
public abstract class RobotBuilder extends CharacterBuilder{
	public void raceBuilder() {
		myChar.setRace("robot");
	}

	public abstract void professionBuilder();

	public abstract void strengthBuilder();
}

/////////////////////////////////////////////////////////////////////////
public abstract class ThiefAlienBuilder extends AlienBuilder {
	public void professionBuilder() {
		myChar.setProfession("thief");
	}

	public abstract void strengthBuilder();
}
public abstract class ThiefHumanBuilder extends HumanBuilder{
	public void professionBuilder() {
		myChar.setProfession("thief");
	}

	public abstract void strengthBuilder();
}
public abstract class ThiefRobotBuilder extends RobotBuilder {
	public void professionBuilder() {
		myChar.setProfession("thief");
	}

	public abstract void strengthBuilder();
}
```

```
public abstract class WarriorAlienBuilder extends AlienBuilder{
	public void professionBuilder() {
		myChar.setProfession("warrior");
	}

	public abstract void strengthBuilder();
}
public abstract class WarriorHumanBuilder extends HumanBuilder{
	public void professionBuilder() {
		myChar.setProfession("warrior");
	}

	public abstract void strengthBuilder();
}
public abstract class WarriorRobotBuilder extends RobotBuilder{
	public void professionBuilder() {
		myChar.setProfession("robot");
	}

	public abstract void strengthBuilder();
}
public abstract class DoctorRobotBuilder extends RobotBuilder {
	public void professionBuilder() {
		myChar.setProfession("doctor");
	}

	public abstract void strengthBuilder();
}
public abstract class DoctorHumanBuilder extends HumanBuilder {
	public void professionBuilder() {
		myChar.setProfession("doctor");
	}

	public abstract void strengthBuilder();
}
public abstract class DoctorAlienBuilder extends AlienBuilder{
	public void professionBuilder() {
		myChar.setProfession("doctor");
	}
```

```
	public abstract void strengthBuilder();
}
//////////////////////////////////////////////////////////////////////////
////
public class LeadershipDoctorRobotBuilder extends DoctorRobotBuilder{
	public void strengthBuilder(){
		myChar.setStrength("leadership");
	}
}
public class LeadershipThiefAlienBuilder extends ThiefAlienBuilder {
	public void strengthBuilder(){
		myChar.setStrength("leadership");
	}
}
public class LeadershipThiefHumanBuilder extends ThiefHumanBuilder{
	public void strengthBuilder(){
		myChar.setStrength("leadership");
	}
}
public class LeadershipThiefRobotBuilder extends ThiefRobotBuilder{
	public void strengthBuilder(){
		myChar.setStrength("leadership");
	}
}
public class LeadershipWarriorAlienBuilder extends WarriorAlienBuilder{
	public void strengthBuilder(){
		myChar.setStrength("leadership");
	}
}
public class LeadershipWarriorHumanBuilder extends WarriorHumanBuilder{
	public void strengthBuilder(){
		myChar.setStrength("leadership");
	}
}
public class LeadershipWarriorRobotBuilder extends WarriorRobotBuilder{
	public void strengthBuilder(){
		myChar.setStrength("leadership");
	}
}
```

```
public class TeamworkDoctorAlienBuilder extends DoctorAlienBuilder{
	public void strengthBuilder(){
		myChar.setStrength("teamwork");
	}
}
public class TeamworkDoctorHumanBuilder extends DoctorHumanBuilder{
	public void strengthBuilder(){
		myChar.setStrength("teamwork");
	}
}
public class TeamworkDoctorRobotBuilder extends DoctorRobotBuilder{
	public void strengthBuilder(){
		myChar.setStrength("teamwork");
	}
}
public class TeamworkThiefAlienBuilder extends ThiefAlienBuilder {
	public void strengthBuilder(){
		myChar.setStrength("teamwork");
	}
}
public class TeamworkThiefHumanBuilder extends ThiefHumanBuilder{
	public void strengthBuilder(){
		myChar.setStrength("teamwork");
	}
}
public class TeamworkThiefRobotBuilder extends ThiefRobotBuilder  {
	public void strengthBuilder(){
		myChar.setStrength("teamwork");
	}
}
public class TeamworkWarriorAlienBuilder extends WarriorAlienBuilder{
	public void strengthBuilder(){
		myChar.setStrength("teamwork");
	}
}
public class TeamworkWarriorHumanBuilder extends WarriorHumanBuilder{
	public void strengthBuilder(){
		myChar.setStrength("teamwork");
	}
```

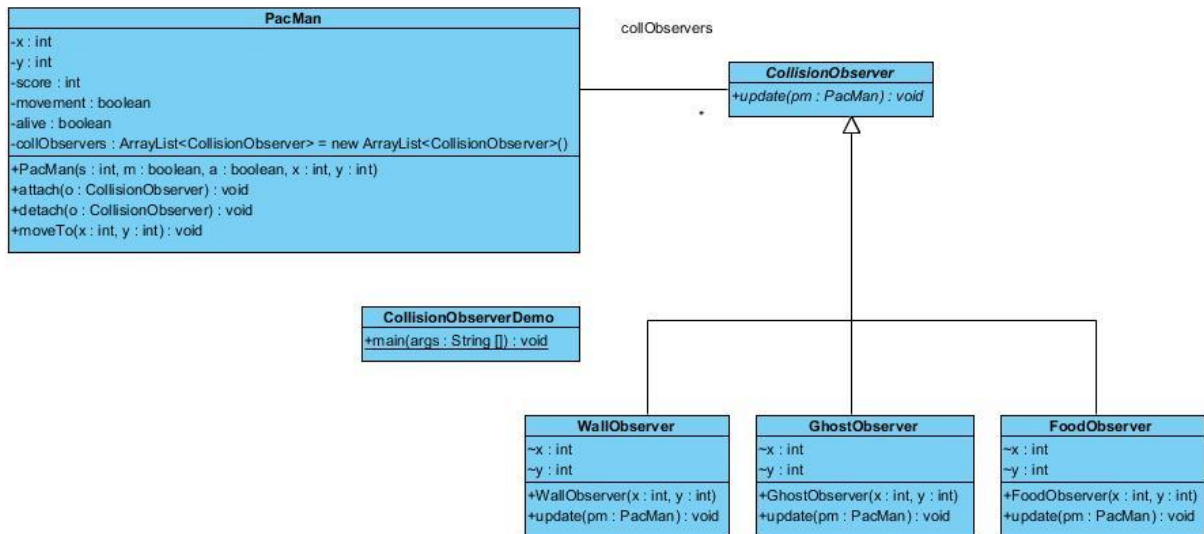
```
}
public class TeamworkWarriorRobotBuilder extends WarriorRobotBuilder {
	public void strengthBuilder(){
		myChar.setStrength("teamwork");
	}
}
public class LeadershipDoctorHumanBuilder extends DoctorHumanBuilder{
	public void strengthBuilder(){
		myChar.setStrength("leadership");
	}
}
public class LeadershipDoctorAlienBuilder extends DoctorAlienBuilder {
	public void strengthBuilder(){
		myChar.setStrength("leadership");
	}
}
public class CreativityWarriorRobotBuilder extends WarriorRobotBuilder {
	public void strengthBuilder(){
		myChar.setStrength("creativity");
	}
}
public class CreativityWarriorHumanBuilder extends WarriorHumanBuilder{
	public void strengthBuilder(){
		myChar.setStrength("creativity");
	}
}
public class CreativityWarriorAlienBuilder extends WarriorAlienBuilder {
	public void strengthBuilder(){
		myChar.setStrength("creativity");
	}
}
public class CreativityThiefRobotBuilder extends ThiefRobotBuilder {
	public void strengthBuilder(){
		myChar.setStrength("creativity");
	}
}
public class CreativityThiefHumanBuilder extends ThiefHumanBuilder{
	public void strengthBuilder(){
		myChar.setStrength("creativity");
```

```
	}
}
public class CreativityThiefAlienBuilder extends ThiefAlienBuilder {
	public void strengthBuilder(){
		myChar.setStrength("creativity");
	}
}
public class CreativityDoctorRobotBuilder extends DoctorRobotBuilder  {
	public void strengthBuilder(){
		myChar.setStrength("creativity");
	}
}
public class CreativityDoctorHumanBuilder extends DoctorHumanBuilder{
	public void strengthBuilder(){
		myChar.setStrength("creativity");
	}
}
public class CreativityDoctorAlienBuilder extends DoctorAlienBuilder{
	public void strengthBuilder(){
		myChar.setStrength("creativity");
	}
}
```

4.3 CollisionObserverDemo (Collision Detection με Observer)

Σε αυτό το παιχνίδι δημιουργείτε ένα PacMan το οποίο κινείται τυχαία στο χώρο με σκοπό να φάει. Το πρόγραμμα ελέγχει πότε το PacMan βρίσκεται σε θέση που έχει φαγητό, πότε χτυπάει σε τοίχο και πότε τον «πιάνουν» τα φαντασματάκια. Στο τελευταίο προφανώς και εντοπίζουμε την ανάγκη του προτύπου μηχανικής παιχνιδιών Collision Detection.

UML Diagram:



Κώδικας:

```
//import javax.swing.*;
import java.util.*;

public class PacMan {
	private int x,y;
	private int score;
	private boolean movement;
	private boolean alive;

	public PacMan(int s,boolean m,boolean a,int x,int y){
		score=s;
		movement=m;
		alive=a;
		this.x=x;
		this.y=y;
	}

	private ArrayList<CollisionObserver> collObservers = new
ArrayList<CollisionObserver>();

	public void attach(CollisionObserver o){
		collObservers.add(o); //καταχώρηση παρατηρητή
```

```
	}

	public void detach(CollisionObserver o){
		collObservers.remove(o); //διαγραφή παρατηρητή
	}

	public void moveTo(int x, int y) {
		this.setX(x);
		this.setY(y);
		System.out.println("Pacman's Position: " + x + "," + y);
		for(int i=0;i<collObservers.size();i++)
			((CollisionObserver)collObservers.get(i)).update(this);
// ανακοίνωση αλλαγών στους παραλήπτες
	}

	public int getScore(){ return score; }
	public void setScore(int s){
		score=s;
		//Notify(); //Κάθε φορά που αλλάζει η κατάσταση καλείται η
Notify
	}
	public boolean getMovement(){ return movement; }
	public void setMovement(boolean m){
		movement=m;
		//Notify(); //Κάθε φορά που αλλάζει η κατάσταση καλείται η
Notify
	}
	public boolean getAlive(){ return alive; }
	public void setAlive(boolean a){
		alive=a;
		//Notify(); //Κάθε φορά που αλλάζει η κατάσταση καλείται η
Notify
	}
	public void setX(int x){
		this.x = x;
	}
	public int getX(){
		return x;
	}
	public void setY(int y){
		this.y = y;
```

```
	}
	public int getY(){
		return y;
	}
	/*
	public void MoveTo(int x,int y){
		this.x=x;
		this.y=y;
	}
	*/
}

public abstract class CollisionObserver {
	public abstract void update(PacMan pm);
}

public class FoodObserver extends CollisionObserver {
	int x,y;
	public FoodObserver(int x,int y){
		this.x=x;
		this.y=y;
	}
	public void update(PacMan pm){
		if (pm.getX()==this.x && pm.getY()==this.y)  {
			System.out.println("PacMan eats food"); //Οι παρατηρητές
αντλούν πληροφορία
			pm.setScore(pm.getScore() + 1);
			pm.setMovement(true);
			//pm.attach (this); //Οι παρατηρητές καταχωρούν τον εαυτό
τους
		}
	}
}

public class GhostObserver extends CollisionObserver {
	int x,y;

	public GhostObserver(int x,int y){
		this.x=x;
		this.y=y;
```

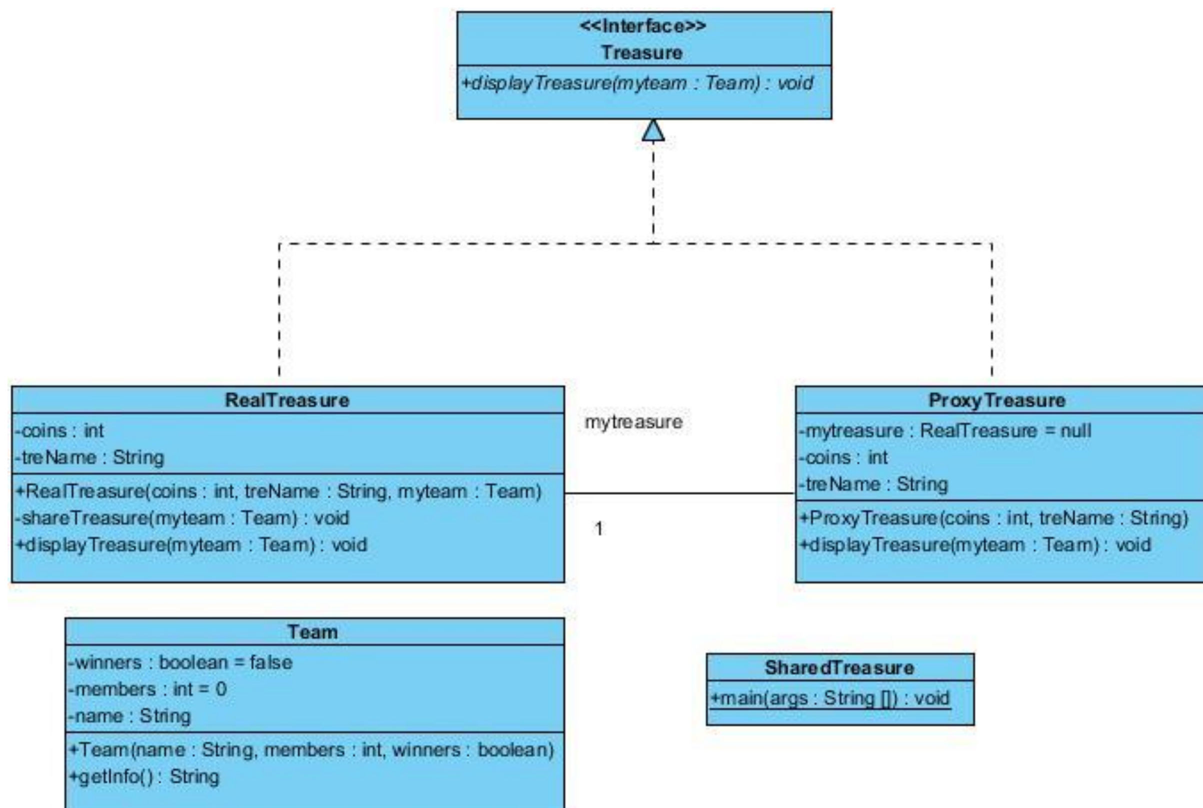
```
	}
	public void update(PacMan pm){
		if (pm.getX()==this.x && pm.getY()==this.y)  {
			System.out.println("PacMan  hits  a  Ghost");  //Οι παρατηρητές αντλούν πληροφορία
			pm.setMovement(false);
			pm.setAlive(false);
			//pm.attach (this); //Οι παρατηρητές καταχωρούν τον εαυτό τους
		}
	}
}

public class WallObserver extends CollisionObserver {
	int x,y;

	public WallObserver(int x,int y){
		this.x=x;
		this.y=y;

	}
	public void update(PacMan pm){
		if (pm.getX()==this.x && pm.getY()==this.y)  {
			System.out.println("PacMan  hits  the  Wall");  //Οι παρατηρητές αντλούν πληροφορία
			pm.setMovement(false);
			//pm.attach(this); //Οι παρατηρητές καταχωρούν τον εαυτό τους
		}
	}
}

public class CollisionObserverDemo {

	public static void main(String[] args) {

		PacMan myPacMan = new PacMan(0,true,true,2,3);

		WallObserver WO1 = new WallObserver(10,10);
		WallObserver WO2 = new WallObserver(5,5);
```

```
        GhostObserver GO1 = new GhostObserver(8,8);
        GhostObserver GO2= new GhostObserver(3,2);
        GhostObserver GO3 = new GhostObserver(7,9);
        FoodObserver FO1 = new FoodObserver(6,6);
        FoodObserver FO2 = new FoodObserver(4,3);
        FoodObserver FO3 = new FoodObserver(6,7);

        myPacMan.attach(FO1);
        myPacMan.attach(FO2);
        myPacMan.attach(FO3);
        myPacMan.attach(WO1);
        myPacMan.attach(WO2);
        myPacMan.attach(GO1);
        myPacMan.attach(GO2);
        myPacMan.attach(GO3);

        int i=0;
        while(myPacMan.getAlive() && i<100){
              int rand_x = (int) Math.round( 15*Math.random() );
              int rand_y = (int) Math.round( 15*Math.random() );
              myPacMan.moveTo(rand_x, rand_y);
              i++;
        }

        //Διαγραφή Παρατηρητών
        myPacMan.detach(FO1);
        myPacMan.detach(FO2);
        myPacMan.detach(FO3);
        myPacMan.detach(GO1);
        myPacMan.detach(GO2);
        myPacMan.detach(GO3);
        myPacMan.detach(WO1);
        myPacMan.detach(WO2);
    }
}
```

4.4 SharedTreasure (Shared Rewards με Proxy)

Στο παρακάτω παιχνίδι έχουμε δύο ομάδες που κυνηγούν δύο διαφορετικού θησαυρούς. Όταν εκπληρώσουν τις προαπαιτούμενες ενέργειες ο θησαυρός μοιράζεται (πρότυπο μηχανικής παιχνιδιών «Shared Rewards» σε όλα τα μέλη της ομάδας αλλά μόνο σε αυτής που το έκανε πρώτη (first come first served). Σε κάθε άλλη περίπτωση η ομάδα δεν έχει τη δυνατότητα να έχει ούτε καν πρόσβαση στον (αληθινό) θησαυρό με τη βοήθεια του αντικειμενοστρεφούς σχεδιαστικού προτύπου «Proxy».

UML Diagram:



Κώδικας:

```
public interface Treasure {
	public void displayTreasure(Team myteam);
}

public class RealTreasure implements Treasure {
	private int coins;
	private String treName;
```

```java
      //Δομητής του RealTreasure
    public RealTreasure(final int coins,final String treName,Team myteam) {
        this.coins = coins;
        this.treName = treName;
        shareTreasure(myteam);
    }


    //Μοίρασμα του θησαυρού
    private void shareTreasure(Team myteam) {
      System.out.println("-----------------------");
        System.out.println(myteam.getName()  +  ",  you  are  the  first  that reached the "+ treName+" treasure!\nYour team's reward: " + coins);
        System.out.println("Each     team's     member     gets:     "     + (coins/myteam.getMembers()) + " coins");
        System.out.println("-----------------------");
        myteam.setWinners(false);//επαναφορά  της  κατάστασης  "υποχρεώσεων" στην αρχική
    }

    //Προβολή στοιχείων θησαυρού
    public void displayTreasure(Team myteam) {
      System.out.println(myteam.getName()  +  ",  it's  too  late  now,  the treasure -" +treName+"- has been taken");
    }
}

public class ProxyTreasure implements Treasure {

    private RealTreasure mytreasure = null;
    private int coins;
    private String treName;

      //Δομητής του ProxyTreasure
    public ProxyTreasure(final int coins,final String treName) {
        this.coins = coins;
        this.treName = treName;
    }
```

```
    //Προβολή στοιχείων θησαυρού
    public void displayTreasure(Team myteam) {
        if (mytreasure == null){
            if(myteam.getWinners()==true){
                    mytreasure = new RealTreasure(coins,treName,myteam);
            }else{
                    System.out.println(myteam.getName()  +  ",  try  Harder  to
get the "+treName+" treasure");
            }
        }
        else{
            mytreasure.displayTreasure(myteam);
        }
    }
}

public class Team {

      private boolean winners = false;
      private int members = 0;
      private String name;

      //Δομητής Team
      public Team(String name,int members,boolean winners){
            this.name = name;
            this.members = members;
            this.winners = winners;
      }

      public void setWinners(boolean winners){
            this.winners = winners;
      }
      public boolean getWinners(){
            return winners;
      }
      public void setMembers(int members){
            this.members = members;
      }
      public int getMembers(){
```

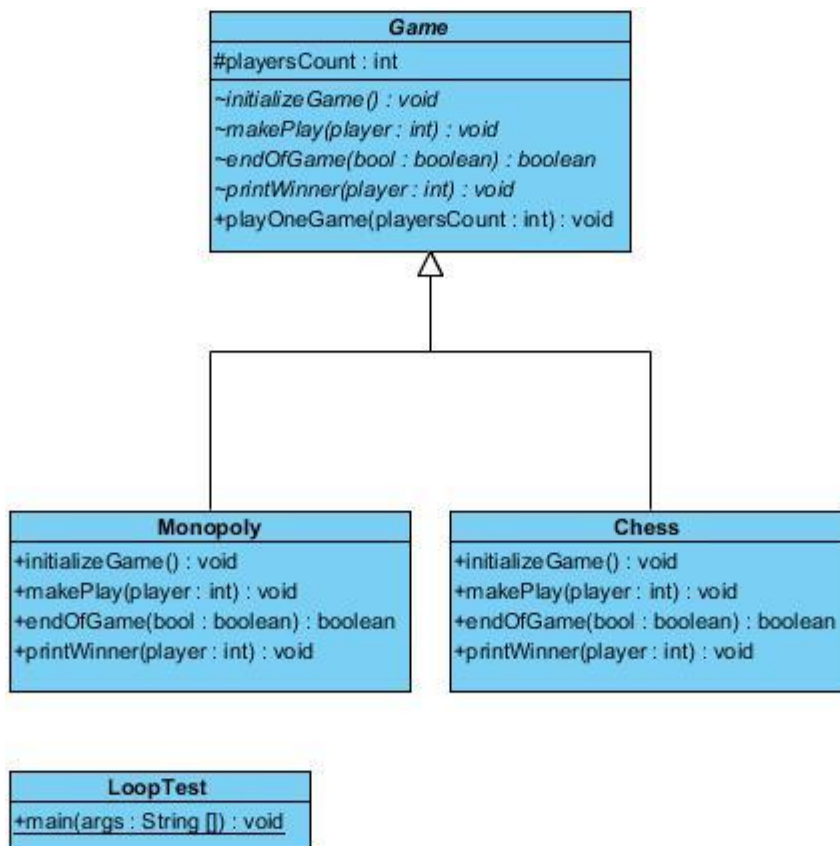
```
		return members;
	}
	public void setName(String name){
		this.name = name;
	}
	public String getName(){
		return name;
	}

	public String getInfo(){
		return "Team: " + name + "\nMembers: " + members + "\n---------
----------";
	}
}

public class SharedTreasure {

	public static void main(String[] args) {
		//Δημιουργία Θησαυρών
		final Treasure tre1 = new ProxyTreasure(100,"Da Vinci");
	  final Treasure tre2 = new ProxyTreasure(10,"Pirate");

	  //Δημιουργία Ομάδων
	  Team team1 = new Team("Team1",5,false);
	  Team team2 = new Team("Team2",3,false);

	  //Προβολή Λεπτομεριών Ομάδων
	  System.out.println(team1.getInfo());
	  System.out.println(team2.getInfo());

		//Προσπάθειες ανάκτησης θησαυρών
	  tre1.displayTreasure(team1);
	  team2.setWinners(true);//H team2 έχει ολοκληρώσει τις υποχρεώσεις
της
	  tre1.displayTreasure(team2);
	  tre2.displayTreasure(team2);
	  tre2.displayTreasure(team1);
	  tre1.displayTreasure(team1);
	  team1.setWinners(true);//H team1 έχει ολοκληρώσει τις υποχρεώσεις
της
```

```
        tre2.displayTreasure(team1);
        tre2.displayTreasure(team2);
    }
}
```

4.5 LoopDemo (Game Loop με Template Method)

Σε αυτό το πρόγραμμα δημιουργούνται διάφορα παιχνίδια της αρεσκείας μας (εδώ δημιουργήθηκαν χωρίς την συνολική τους υλοποίηση η Monopoly και το Σκάκι) στα οποία οι παίκτες δεν παίζουν ταυτόχρονα (turn based). Έτσι λοιπόν το παιχνίδι συνεχίζει να «τρέχει» έως ότου η μεταβλητή endOfGame γίνει αληθείς. Εδώ εντοπίζεται και το πρότυπο μηχανικής παιχνιδιών «Game Loop».

UML Diagram:



Κώδικας:

```
public abstract class Game {
```

```
  protected int playersCount;
abstract void initializeGame();
abstract void makePlay(int player);
abstract boolean endOfGame(boolean bool);
abstract void printWinner(int player);

/* A template method : */
public final void playOneGame(int playersCount) {
    this.playersCount = playersCount;
    initializeGame();
    int j = 0;
    double test = 0;
    boolean bool_test = false;
    while (!endOfGame(bool_test)) {
        makePlay(j);
        j = (j + 1) % playersCount;
        test = 100*Math.random();
        if(test>90)
              bool_test = true;
    }
    printWinner((j));
}
}

public class Monopoly extends Game{

    public void initializeGame() {
      // Αρχικοποίηση παιχνιδιού
        System.out.println("Initializing Monopoly Game for " + playersCount
);
    }
    public void makePlay(int player) {
      // Διαδικασία της σειράς των παικτών
      System.out.println("It's player's number " + (player+1) + " turn!");
    }
    public boolean endOfGame(boolean bool) {
      // επιστρέφει true μόνο αν έχουμε πλειοψηφία οικοπέδων
        return bool;
    }
```

```
    public void printWinner(int player) {
      // Ανακοίνωση του νικητή
      System.out.println("And the winner is player number: " + player);
    }
    /* Από εδώ και κάτω μπαίνουν οι κλάσεις που αφορούν τη Monopoly */

    // ...
}

public class Chess extends Game{

    public void initializeGame() {
        // Αρχικοποίηση παιχνιδιού
        System.out.println("Initializing Chess Game...");
    }
    public void makePlay(int player) {
        // Διαδικασία της σειράς των παικτών
      if(player==0)
            System.out.println("Now playing: blacks");
      else
            System.out.println("Now playing: whites");

    }
    public boolean endOfGame(boolean bool) {
        // επιστρέφει true μόνο αν έχουμε Ματ
        return bool;
    }
    public void printWinner(int player) {
        // Ανακοίνωση του νικητή
      if(player==1)
            System.out.println("Winner: blacks");
      else
            System.out.println("Winner: whites");
    }
    /* Από εδώ και κάτω μπαίνουν οι κλάσεις που αφορούν το Σκάκι */

    // ...
}
```

```java
public class LoopTest {
	public static void main(String[] args) {

		Game mygame = new Chess();
		mygame.playOneGame(2);

		Game secondgame = new Monopoly();
		secondgame.playOneGame(5);

	}
}
```

5 Συμπεράσματα

Η παρούσα πτυχιακή εργασία αποσκοπεί στην αξιολόγηση της χρήσης του αντικειμενοστρεφών σχεδιαστικών προτύπων στην ανάπτυξη ενός του παιχνιδιού. Προκειμένου να επιτευχθεί αυτός ο στόχος προγραμματίστηκαν πέντε παιχνίδια. Τα αποτελέσματα δείχνουν ότι τα πρότυπα μπορεί να είναι επωφελή σε σχέση με τη συντήρηση, ενώ τα παιχνίδια έχουν μειωμένη πολυπλοκότητα και σύζευξη σε σύγκριση με το προγραμματισμό τους χωρίς πρότυπο. Επιπροσθέτως, η εφαρμογή των προτύπων τείνει να αυξήσει τη συνοχή του λογισμικού. Κατά συνέπεια, και λόγω της συνεχώς μεταβαλλόμενης φύσης των παιχνιδιών, πιστεύω πως στο προγραμματισμό παιχνιδιών θα πρέπει να γίνεται χρήση προτύπων σχεδίασης.

Θεωρούμε ότι εφόσον πέντε από τα πρότυπα μηχανικής παιχνιδιών μπορούν να υλοποιηθούν με αντικειμενοστρεφή πρότυπα σχεδίασης, πιθανότατα μπορούν και τα υπόλοιπα προσφέροντας τα παραπάνω πλεονεκτήματα και κυρίως μπορούν να παρέχουν έτοιμες σχεδιαστικά λύσεις για την υλοποίηση συχνά εμφανιζόμενων απαιτήσεων στο σχεδιασμό παιχνιδιών.

ΒΙΒΛΙΟΓΡΑΦΙΑ

A. Ampatzoglou and A. Chatzigeorgiou, “Evaluation of object-oriented design patterns in game development”, Information and Software Technology, Elsevier, 49 (5), pp.445-454, May 2007.

G. Antoniol, G. Casazza, M .Di Penta and R .Fiutem, “Object-Oriented design patterns recovery”, Journal of Systems and Software, Elsevier, 59 (2), pp.181-196, November 2001

A. Ampatzoglou and Ioannis Stamelos, “Software engineering research for computer games: A systematic review”, Information and Software Technology, May 2010

Απόστολος Ζάρρας, “Πρότυπα Δόμησης (Structural Patterns)”, http://www.cs.uoi.gr/~zarras/

Staffan Björk, Sus Lundgren, “Game Design Patterns”, PLAY, Interactive Institute, May 2010

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (2001). Design Patterns – Elements of Reusable Object-Oriented Software. Addison-Wesley, 2001.

Kreimeier, B., Holopainen, J., & Björk, S. (2003). Game Design Patterns. Lecture Notes from GDC 2003, Game Developers Conference, March 4-8, 2003, San Jose, CA, USA

Björk, S. & Holopainen, J. (2003). Describing Games - An Interaction-Centric Structural Framework. Proceedings of Level Up - 1st international Digital Games Research Conference 2003, 4-6 November 2003 University of Utrecht, The Netherlands.

Boardgamegeek. http://www.boardgamegeek.com.

Church, D. (1999). Formal Abstract Design Tools. Online article available at www.gamasutra.com.

Crawford, C. (1982). The Art of Computer Game Design

Kreimeier, B. (2002). The Case For Game Design Patterns. Online publication available from http://www.gamasutra.com/features/20020313/kreimeier_03.htm.

A. Χατζηγεωργίου, “Αντικειμενοστρεφής Σχεδίαση: UML, Αρχές, Πρότυπα και Ευρε-τικοί Κανόνες”, Κλειδάριθμος, 2005

L. Herman, J. Horwitz, St. Kent et all, “The history of video games” accessed by www.gamespot.com

C. Crawford, “A taxonomy of computer games”, Washington State University, 2006

L. Bishop, D. Eberly, T. Whitted, M. Finch, M. Shantz, “Designing a PC game engine”, IEEE Computer Graphics and Application, 1998

N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, S. T. Halkidis, "Design Pattern Detection using Similarity Scoring", IEEE Transactions on Software Engineering, vol. 32, no. 11, November 2006, pp. 896-909