ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΛΟΠΟΝΝΗΣΟΥ

ΣΧΟΛΗ ΜΗΧΑΝΙΚΩΝ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

“Σχεδιασμός και ανάπτυξη πλατφόρμας για παρτίδες σκάκι σε Java**”**

Σταμάτιος Κουλούρης

ΕΠΙΒΛΕΠΩΝ: Χριστοδούλου Σωτήρης

ΠΑΤΡΑ 2021

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή

Πάτρα, Ημερομηνία

ΕΠΙΤΡΟΠΗ ΑΞΙΟΛΟΓΗΣΗΣ

1. Ονοματεπώνυμο, Υπογραφή

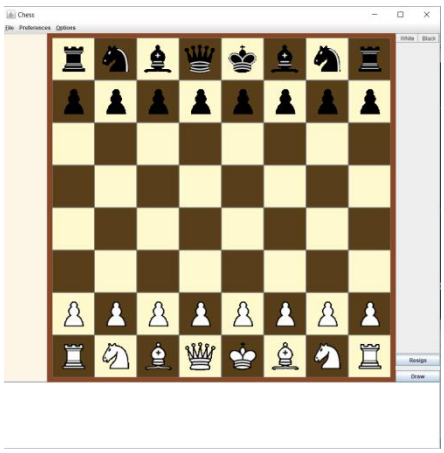
2. Ονοματεπώνυμο, Υπογραφή

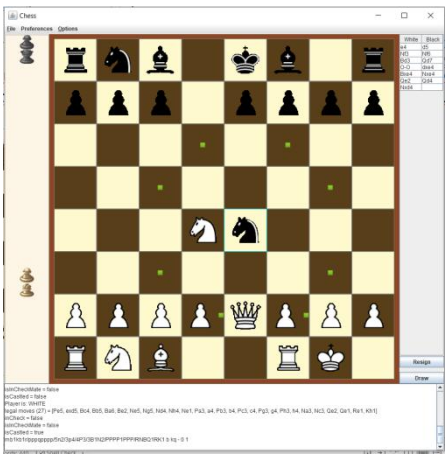
3. Ονοματεπώνυμο, Υπογραφή

Περιγραφή για τις ιδιότητες και τις λειτουργείς της Desktop εφαρμογής της Σκακιέρας

- Η εφαρμογή είναι ενα Board (ταμπλό) το οποίο έχει μέγεθος 64 Tiles (τετράγωνα).
- Έχουμε τα Pieces (πιόνια) του παιχνιδιού τα οποία έχουν Moves για το καθένα ξεχωριστά.
  - Pawn
  - Rook
  - Knight
  - Bishop
  - Queen
  - King

- Διαθέτει τους Player (οπού έχουμε τον Άσπρο και αντίστοιχα τον Μαύρο).
  - WhitePlayer
  - BlackPlayer
- Τις κινήσεις των Piece (πιονιών) άλλα και των Player (παιχτών) που δικαιούται ο καθένας.
- Για τις Move (κινήσεις) έχουμε μια συλλογή από την κίνηση άλλα και το ροκέ από την μεριά του βασιλιά είτε την μεριά της βασίλισσας αν το δικαιούται εκείνη την σειρά το EnPassant στα πιόνια άλλα και όταν ένα πιόνι καταφέρει να φτάσει στην άλλη μεριά της σκακιέρας για την αναβάθμιση και τέλος αν μπορεί να επιτεθεί σε ένα αντίπαλο Piece.
- Γραφικό περιβάλλον εφαρμογής όπως φαίνεται παρακάτω έχουμε το ταμπλό της σκακιέρας ένα ιστορικό στα αριστερά και δέξια οι κινήσεις που έχουν παιχτεί καθώς και από κάτω ένα log για παραμετρικά του συστήματος.





- Υλοποίηση αλγόριθμου minMax ώστε ένας χρήστης να μπορεί να παίξει μια παρτίδα με το computer έως 6 επίπεδα.
- Έχει την ικανότητα να γνωρίζει και να οργανώνει την σειρά των παιχτών.
- Μπορεί ανάπαυσα στιγμή να αναφέρει ποια είναι τα ενεργά πιόνια.
- Σε ένα γραφικό πάνελ καταγράφονται οι κινήσεις που έχουν διεκπεραιωθεί.
- Παρόμοιος καταγράφονται και τα πιόνια που έχουν αιχμαλωτιστεί.

- Είναι σε θέση να γνωρίζει ποτέ εφαρμόζεται η ισοπαλία το λεγόμενο Pat.
- Αντιλαμβάνεται τα “Ρουά” του αντιπάλου παίχτη.
- Σταματάει το παιχνίδι αν ένας από τους δυο παίχτες βρεθεί σε “Ρουά Ματ” και ανακοινώνει τον νικητή.

ΙΣΤΟΡΙΚΗ ΑΝΑΔΡΟΜΗ

Οι James Gosling, Mike Sheridan και Patrick Naughton ξεκίνησαν το πρόγραμμα γλώσσας Java τον Ιούνιο του 1991. [22] Η Java είχε αρχικά σχεδιαστεί για διαδραστική τηλεόραση, αλλά ήταν πολύ προηγμένη για τη βιομηχανία ψηφιακής καλωδιακής τηλεόρασης εκείνη την εποχή. Η γλώσσα ονομάστηκε αρχικά Oak μετά από μια βελανιδιά που βρισκόταν έξω από το γραφείο του Gosling Αργότερα, το έργο πήρε το όνομα Green και τελικά μετονομάστηκε Java, από Java Java, έναν τύπο καφέ από την Ινδονησία. Η Gosling σχεδίασε Java με σύνταξη τύπου C / C ++ που οι προγραμματιστές συστημάτων και εφαρμογών θα βρίσκουν οικείες.

Η Sun Microsystems κυκλοφόρησε την πρώτη δημόσια εφαρμογή ως Java 1.0 το 1996. Υποσχέθηκε τη λειτουργία Write Once, Run Anywhere (WORA), παρέχοντας χρόνο εκτέλεσης χωρίς κόστος σε δημοφιλείς πλατφόρμες. Αρκετά ασφαλής και διαθέτει διαμορφώσιμη ασφάλεια, επέτρεψε περιορισμούς πρόσβασης σε δίκτυο και αρχεία. Τα μεγάλα προγράμματα περιήγησης στο Web ενσωμάτωσαν σύντομα τη δυνατότητα εκτέλεσης εφαρμογών Java σε ιστοσελίδες και η Java γρήγορα έγινε δημοφιλής. Ο μεταγλωττιστής Java 1.0 ξαναγράφηκε στην Java από τον Arthur van Hoff για να συμμορφωθεί αυστηρά με τις προδιαγραφές γλώσσας Java 1.0. Με την έλευση του Java 2 (κυκλοφόρησε αρχικά ως J2SE 1.2 τον Δεκέμβριο 1998 - 1999), οι νέες εκδόσεις είχαν πολλές διαμορφώσεις που έχουν δημιουργηθεί για διαφορετικούς τύπους πλατφορμών. Το J2EE περιελάμβανε τεχνολογίες και API για εταιρικές εφαρμογές που εκτελούνται συνήθως σε περιβάλλοντα διακομιστή, ενώ το J2ME διαθέτει API βελτιστοποιημένα για εφαρμογές για κινητά. Η έκδοση της επιφάνειας εργασίας μετονομάστηκε σε J2SE. Το 2006, για σκοπούς μάρκετινγκ, η Sun μετονόμασε νέες εκδόσεις J2 σε Java EE, Java ME και Java SE, αντίστοιχα.

Το 1997, η Sun Microsystems πλησίασε το πρότυπο ISO / IEC JTC 1 και αργότερα το Ecma International για να επισημοποιήσει την Java, αλλά σύντομα αποσύρθηκε από τη διαδικασία. Η Java παραμένει ένα de facto πρότυπο, που ελέγχεται μέσω της διαδικασίας κοινοτικής Java. [31] Κάποια στιγμή, η Sun έκανε τις περισσότερες από τις εφαρμογές Java διαθέσιμες χωρίς χρέωση, παρά την ιδιόκτητη κατάσταση λογισμικού. Η Sun δημιούργησε έσοδα από την Java μέσω της πώλησης αδειών για εξειδικευμένα προϊόντα όπως το Java Enterprise System.

Στις 13 Νοεμβρίου 2006, η Sun κυκλοφόρησε μεγάλο μέρος της εικονικής μηχανής Java (JVM) ως δωρεάν και λογισμικό ανοιχτού κώδικα (FOSS), σύμφωνα με τους όρους της GNU General Public License (GPL). Στις 8 Μαΐου 2007, η Sun ολοκλήρωσε τη διαδικασία, καθιστώντας όλο τον βασικό κώδικα της JVM διαθέσιμο υπό όρους διανομής ελεύθερου λογισμικού / ανοιχτού κώδικα, εκτός από ένα μικρό τμήμα κώδικα στο οποίο η Sun δεν είχε τα πνευματικά δικαιώματα.

Ο αντιπρόεδρος της Sun Rich Green είπε ότι ο ιδανικός ρόλος της Sun σε σχέση με την Java ήταν ως ευαγγελιστής. [33] Μετά την εξαγορά της Sun Microsystems από την Oracle Corporation το 2009–10, η Oracle χαρακτήρισε τον εαυτό της ως διαχειριστής της τεχνολογίας Java με αδιάκοπη δέσμευση να προωθήσει μια κοινότητα συμμετοχής και διαφάνειας. Αυτό δεν εμπόδισε την Oracle να υποβάλει αγωγή εναντίον της Google λίγο μετά από αυτήν για τη χρήση Java στο Android SDK (δείτε την ενότητα Android).
Στις 2 Απριλίου 2010, ο Τζέιμς Γκόσλινγκ παραιτήθηκε από την Oracle.
Τον Ιανουάριο του 2016, η Oracle ανακοίνωσε ότι τα περιβάλλοντα χρόνου εκτέλεσης Java που βασίζονται στο JDK 9 θα διακόψουν την προσθήκη του προγράμματος περιήγησης.
Το λογισμικό Java λειτουργεί σε όλα, από φορητούς υπολογιστές έως κέντρα δεδομένων, κονσόλες παιχνιδιών έως επιστημονικούς υπερυπολογιστές.

ΣΧΕΔΙΑΣΜΟΣ ΚΑΙ ΥΛΟΙΗΣΗ ΕΦΑΡΜΟΓΗΣ

Εργαλεία που χρησιμοποιήθηκαν κατά την ανάπτυξη Java 7 σε IDEA NetBeans με Java Development Kit 1.7 και για την υλοποίηση του του γραφικού μέρους εκμεταλλεύτηκα την υπάρχων βιβλιοθήκη Swing , ακολουθήθηκε η δομή του MVC για την αρχιτεκτονική της εφαρμογής. Για την οργάνωση του προτζεκτ δούλεψα Jira και git hub για την συντήρηση του κωδικά και το Confluence για την ανάλυση των απαιτήσεων.

Η Java είναι μια γλώσσα προγραμματισμού υψηλού επιπέδου, βασισμένη σε αντικείμενα, η οποία έχει σχεδιαστεί για να έχει όσο το δυνατόν λιγότερες εξαρτήσεις υλοποίησης. Πρόκειται για μια γλώσσα προγραμματισμού γενικού σκοπού που επιτρέπει στους προγραμματιστές εφαρμογών να γράφουν μία φορά, να εκτελούνται οπουδήποτε, που σημαίνει ότι ο μεταγλωττισμένος κώδικας Java μπορεί να εκτελεστεί σε όλες τις πλατφόρμες που υποστηρίζουν Java χωρίς την ανάγκη επανασυγκέντρωσης. Οι εφαρμογές Java συνήθως συντάσσονται σε bytecode που μπορούν να εκτελεστούν σε οποιαδήποτε εικονική μηχανή Java (JVM) ανεξάρτητα από την υποκείμενη αρχιτεκτονική του υπολογιστή. Η σύνταξη της Java είναι παρόμοια με C και C ++, αλλά έχει λιγότερες εγκαταστάσεις χαμηλού επιπέδου από οποιαδήποτε από αυτές. Ο χρόνος εκτέλεσης Java παρέχει δυναμικές δυνατότητες (όπως τροποποίηση κωδικού προβληματισμού και χρόνου εκτέλεσης) που συνήθως δεν είναι διαθέσιμες σε παραδοσιακές μεταγλωττισμένες γλώσσες. Από το 2019, η Java ήταν μια από τις πιο δημοφιλείς γλώσσες προγραμματισμού που χρησιμοποιούνται σύμφωνα με το GitHub, ειδικά για εφαρμογές ιστού-διακομιστή-πελάτη, με 9 εκατομμύρια προγραμματιστές.

Το Java αναπτύχθηκε αρχικά από τον James Gosling στο Sun Microsystems (το οποίο έκτοτε εξαγοράστηκε από την Oracle) και κυκλοφόρησε το 1995 ως βασικό συστατικό της πλατφόρμας Java της Sun Microsystems. Οι αρχικοί επεξεργαστές Java και οι υλοποιητές αναφοράς, οι εικονικές μηχανές και οι βιβλιοθήκες τάξεων κυκλοφόρησαν αρχικά από τη Sun με αποκλειστικές άδειες. Από τον Μάιο του 2007, σύμφωνα με τις προδιαγραφές της Κοινοτικής Διαδικασίας Java, η Sun παραιτήθηκε από τις περισσότερες από τις τεχνολογίες της Java βάσει της άδειας GNU General

Public License. Η Oracle προσφέρει το δικό της HotSpot Java Virtual Machine, ωστόσο η επίσημη εφαρμογή αναφοράς είναι το OpenJDK JVM που είναι δωρεάν λογισμικό ανοιχτού κώδικα και χρησιμοποιείται από τους περισσότερους προγραμματιστές και είναι το προεπιλεγμένο JVM για σχεδόν όλες τις διανομές Linux.

Από τον Μάρτιο του 2021, η τελευταία έκδοση είναι η Java 16, με την Java 11, μια τρέχουσα υποστηριζόμενη έκδοση μακροπρόθεσμης υποστήριξης (LTS), που κυκλοφόρησε στις 25 Σεπτεμβρίου 2018. Η Oracle κυκλοφόρησε την τελευταία δημόσια ενημέρωση μηδενικού κόστους για την παλαιά έκδοση Java 8 Το LTS τον Ιανουάριο του 2019 για εμπορική χρήση, αν και διαφορετικά θα υποστηρίζει το Java 8 με δημόσιες ενημερώσεις για προσωπική χρήση επ 'αόριστον. Άλλοι προμηθευτές έχουν αρχίσει να προσφέρουν εκδόσεις μηδενικού κόστους των OpenJDK 8 και 11 που εξακολουθούν να λαμβάνουν ασφάλεια και άλλες αναβαθμίσεις.

Η Oracle (και άλλοι) συνιστούν ανεπιφύλακτα την απεγκατάσταση παλαιών εκδόσεων Java λόγω σοβαρών κινδύνων που οφείλονται σε ανεπίλυτα ζητήματα ασφαλείας. Δεδομένου ότι τα Java 9, 10, 12, 13, 14 και 15 δεν υποστηρίζονται πλέον, η Oracle συμβουλεύει τους χρήστες της να μεταβούν αμέσως στην τελευταία έκδοση (επί του παρόντος Java 16) ή σε μια έκδοση LTS.

Ο κώδικας γράφτηκε σε Java με βάση την JVM. Η Java Virtual Machine (JVM) είναι ένα σύνολο προγραμμάτων λογισμικού υπολογιστή και δομών δεδομένων που χρησιμοποιούν ένα μοντέλο εικονικής μηχανής για την εκτέλεση άλλων προγραμμάτων και σεναρίων υπολογιστών. Το μοντέλο που χρησιμοποιείται από μια JVM δέχεται μια μορφή ενδιάμεσης γλώσσας υπολογιστή που αναφέρεται συνήθως ως Java bytecode. Αυτή η γλώσσα αντιπροσωπεύει εννοιολογικά το σύνολο εντολών μιας αρχιτεκτονικής ικανότητας προσανατολισμένη σε στοίβα.Τα Java Virtual Machines λειτουργούν σε Java bytecode, ο οποίος συνήθως δημιουργείται από τον πηγαίο κώδικα Java. Ένα JVM μπορεί επίσης να χρησιμοποιηθεί για την εφαρμογή γλωσσών προγραμματισμού εκτός από την Java.Το JVM είναι ένα κρίσιμο στοιχείο της πλατφόρμας Java. Επειδή τα JVM είναι διαθέσιμα για πολλές πλατφόρμες υλικού και λογισμικού, η Java μπορεί να είναι ταυτόχρονα μεσαίο λογισμικό και μια πλατφόρμα από μόνη της - ως εκ τούτου, το εμπορικό σήμα γράφει μία φορά, εκτελείται οπουδήποτε. Η χρήση του ίδιου bytecode για όλες τις πλατφόρμες επιτρέπει στην Java να περιγραφεί ως "μεταγλώττιση μία φορά, εκτέλεση οπουδήποτε", σε αντίθεση με την "εγγραφή μία φορά, μεταγλώττιση οπουδήποτε", η οποία περιγράφει τις γλώσσες που έχουν μεταγλωττιστεί μεταξύ πλατφορμών Το JVM επιτρέπει επίσης τέτοιες μοναδικές δυνατότητες όπως ο αυτοματοποιημένος χειρισμός εξαίρεσης που παρέχει πληροφορίες εντοπισμού σφαλμάτων «root-αιτίας» για κάθε σφάλμα λογισμικού (εξαίρεση) ανεξάρτητα από τον πηγαίο κώδικα.

Το JVM διανέμεται μαζί με ένα σύνολο τυπικών βιβλιοθηκών τάξεων που εφαρμόζουν το Java API (Application Programming Interface). Μια διεπαφή προγραμματισμού εφαρμογών είναι αυτό που παρέχει ένα σύστημα υπολογιστή, μια βιβλιοθήκη ή μια εφαρμογή προκειμένου να επιτρέπεται η ανταλλαγή

δεδομένων μεταξύ τους. Δεδομένου ότι το Web API είναι η έκδοση Web αυτής της διεπαφής, το JVM και το API πρέπει να είναι συνεπή μεταξύ τους.

Περιβάλλον εκτέλεσης
Τα προγράμματα που προορίζονται να εκτελεστούν σε ένα JVM πρέπει να μεταγλωττιστούν σε μια τυποποιημένη φορητή δυαδική μορφή, η οποία συνήθως διατίθεται με τη μορφή αρχείων .class. Ένα πρόγραμμα μπορεί να αποτελείται από πολλές κατηγορίες σε διαφορετικά αρχεία. Για ευκολότερη διανομή μεγάλων προγραμμάτων, αρχεία πολλαπλών τάξεων μπορούν να συσκευαστούν μαζί σε ένα αρχείο .jar (συντομογραφία για το αρχείο Java).Ο χρόνος εκτέλεσης JVM εκτελεί αρχεία .class ή .jar, προσομοιώνοντας το σύνολο εντολών JVM ερμηνεύοντάς το, ή χρησιμοποιώντας έναν μεταγλωττιστή just-in-time (JIT) όπως το HotSpot της Sun. Η σύνταξη JIT, χωρίς ερμηνεία, χρησιμοποιείται στα περισσότερα JVM σήμερα για να επιτύχει μεγαλύτερη ταχύτητα. Υπάρχουν επίσης μεταγλωττιστές εκ των προτέρων που επιτρέπουν στον προγραμματιστή να προ-μεταγλωττίσει αρχεία κλάσης σε εγγενή κώδικα για μια συγκεκριμένη πλατφόρμα.Όπως και οι περισσότερες εικονικές μηχανές, η Java Virtual Machine διαθέτει αρχιτεκτονική στοίβας που μοιάζει με μικροελεγκτή / μικροεπεξεργαστή.Το JVM, το οποίο είναι το παράδειγμα του JRE (Java Runtime Environment), τίθεται σε λειτουργία όταν εκτελείται ένα πρόγραμμα Java. Όταν ολοκληρωθεί η εκτέλεση, αυτή η παρουσία συλλέγεται σκουπίδια. Το JIT είναι το μέρος του JVM που χρησιμοποιείται για την επιτάχυνση του χρόνου εκτέλεσης. Το JIT συγκεντρώνει τμήματα του κώδικα byte που έχουν παρόμοια λειτουργικότητα ταυτόχρονα, και ως εκ τούτου μειώνει το χρόνο που απαιτείται για τη συλλογή.

Bytecode verifier

Μια βασική φιλοσοφία της Java είναι ότι είναι εγγενώς «ασφαλές» από τη σκοπιά ότι κανένα πρόγραμμα χρήστη δεν μπορεί να «συντρίψει» το μηχάνημα κεντρικού υπολογιστή ή να παρεμβαίνει με άλλο τρόπο ακατάλληλα σε άλλες λειτουργίες του κεντρικού υπολογιστή και ότι είναι δυνατή η προστασία ορισμένων λειτουργιών και δεδομένων δομές που ανήκουν σε "αξιόπιστο" κώδικα από την πρόσβαση ή καταστροφή από "μη αξιόπιστο" κώδικα που εκτελείται εντός του ίδιου JVM. Επιπλέον, δεν επιτρέπονται κοινά σφάλματα προγραμματιστή που συχνά οδηγούν σε καταστροφή δεδομένων ή απρόβλεπτη συμπεριφορά, όπως η πρόσβαση στο τέλος ενός πίνακα ή η χρήση ενός μη αρχικοποιημένου δείκτη. Πολλά χαρακτηριστικά της Java συνδυάζονται για να παρέχουν αυτήν την ασφάλεια, όπως το μοντέλο της κατηγορίας, ο σωρός που συλλέγεται σκουπίδια και ο επαληθευτής.

Το JVM επαληθεύει όλους τους κωδικούς bytec πριν να εκτελεστεί. Αυτή η επαλήθευση αποτελείται κυρίως από τρεις τύπους ελέγχων:

Τα υποκαταστήματα είναι πάντα σε έγκυρες τοποθεσίες
Τα δεδομένα πάντα αρχικοποιούνται και οι αναφορές είναι πάντα ασφαλείς για τον τύπο
Η πρόσβαση σε "ιδιωτικά" ή "ιδιωτικά πακέτα" δεδομένα και μεθόδους ελέγχεται αυστηρά.

Οι δύο πρώτοι από αυτούς τους ελέγχους πραγματοποιούνται κυρίως κατά τη διάρκεια του βήματος "επαλήθευσης" που συμβαίνει όταν φορτώνεται μια κλάση και καθίσταται κατάλληλη για χρήση. Το τρίτο εκτελείται πρωτίστως δυναμικά, όταν τα στοιχεία δεδομένων ή οι μέθοδοι μιας κλάσης έχουν πρώτα πρόσβαση σε άλλη τάξη.

Ο επαληθευτής επιτρέπει μόνο ορισμένες ακολουθίες bytecode σε έγκυρα προγράμματα, π.χ. μια εντολή άλματος (κλάδος) μπορεί να στοχεύει μόνο μια εντολή μέσα στην ίδια λειτουργία ή μέθοδο. Εξαιτίας αυτού, το γεγονός ότι το JVM είναι αρχιτεκτονική στοίβας δεν συνεπάγεται ποινή ταχύτητας για εξομοίωση σε αρχιτεκτονικές που βασίζονται σε καταχωρητές κατά τη χρήση ενός μεταγλωττιστή JIT. Αντιμέτωποι με την αρχιτεκτονική JVM που έχει επαληθευτεί με κωδικό, δεν έχει καμία σημασία για έναν μεταγλωττιστή JIT εάν παίρνει ονόματα φανταστικών καταχωρητών ή φανταστικών θέσεων στοίβας που πρέπει να εκχωρηθούν στους καταχωρητές της αρχιτεκτονικής στόχου. Στην πραγματικότητα, η επαλήθευση κώδικα κάνει το JVM διαφορετικό από μια κλασική αρχιτεκτονική στοίβας του οποίου η αποτελεσματική εξομοίωση με έναν μεταγλωττιστή JIT είναι πιο περίπλοκη και συνήθως πραγματοποιείται από έναν πιο αργό διερμηνέα.

Η επαλήθευση κώδικα διασφαλίζει επίσης ότι τα αυθαίρετα μοτίβα bit δεν μπορούν να χρησιμοποιηθούν ως διεύθυνση. Η προστασία της μνήμης επιτυγχάνεται χωρίς την ανάγκη για μονάδα διαχείρισης μνήμης (MMU). Έτσι, το JVM είναι ένας αποτελεσματικός τρόπος προστασίας της μνήμης σε απλές αρχιτεκτονικές που δεν διαθέτουν MMU. Αυτό είναι ανάλογο με τον διαχειριζόμενο κώδικα στο .NET Common Language Runtime της Microsoft και εννοιολογικά παρόμοιο με δυνατότητες αρχιτεκτονικής όπως το Plessey 250 και το IBM System / 38.

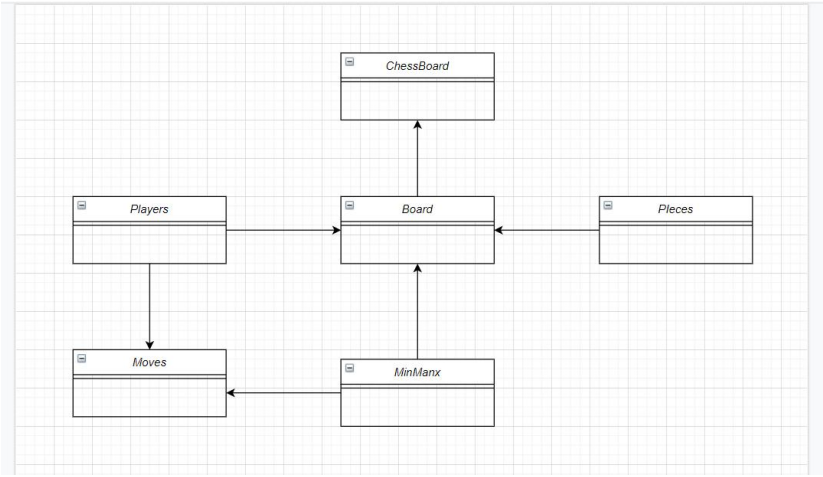
Για την δημιουργεία του γραφικού περιβάλλοντος εφαρμόστηκεSwing.
Η Swing είναι ένα κιτ εργαλείων widget GUI για Java. Είναι μέρος των τάξεων Java Foundation Oracle (JFC) - ένα API για την παροχή γραφικής διεπαφής χρήστη (GUI) για προγράμματα Java.

Η Swing αναπτύχθηκε για να παρέχει ένα πιο εξελιγμένο σύνολο στοιχείων GUI από το προηγούμενο Abstract Window Toolkit (AWT). Η Swing παρέχει μια εμφάνιση και αίσθηση που προσομοιώνει την εμφάνιση και την αίσθηση πολλών πλατφορμών και υποστηρίζει επίσης μια βολική εμφάνιση και αίσθηση που επιτρέπει στις εφαρμογές να έχουν μια εμφάνιση και αίσθηση που δεν σχετίζεται με την υποκείμενη πλατφόρμα. Έχει πιο ισχυρά και ευέλικτα εξαρτήματα από το AWT. Εκτός από τα γνωστά στοιχεία, όπως κουμπιά, πλαίσια ελέγχου και ετικέτες, το Swing παρέχει πολλά προηγμένα στοιχεία, όπως πάνελ με καρτέλες, παράθυρα κύλισης, δέντρα, πίνακες και λίστες.

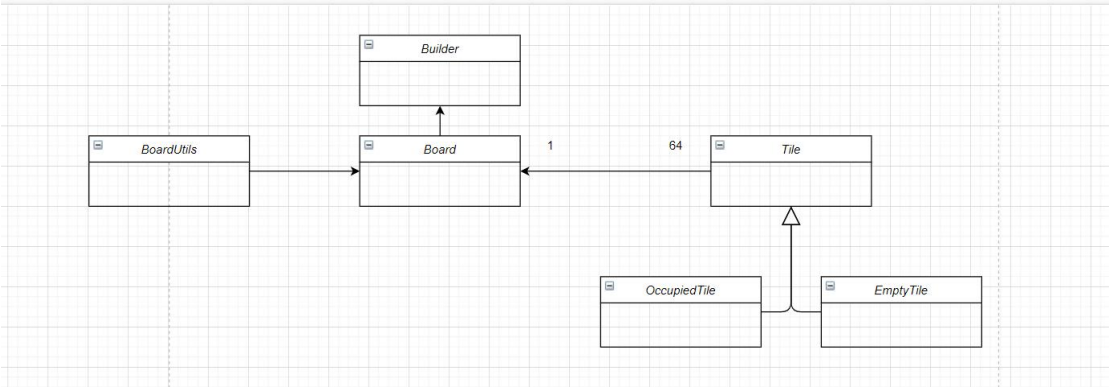
Σε αντίθεση με τα στοιχεία AWT, τα στοιχεία Swing δεν εφαρμόζονται με κώδικα για πλατφόρμα. Αντ 'αυτού, είναι γραμμένα εξ ολοκλήρου στην Java και επομένως είναι ανεξάρτητες από την πλατφόρμα.

Τον Δεκέμβριο του 2008, η Sun Microsystems (ο προκάτοχός της Oracle) κυκλοφόρησε το πλαίσιο που βασίζεται σε CSS / FXML, το οποίο σκόπευε να είναι ο διάδοχος του Swing, που ονομάζεται JavaFX.

Μια αφαιρετική εικόνα των ισχυρών κλάσεων που απεικονίζει τις σχέσεις που υπάρχουν στο project :

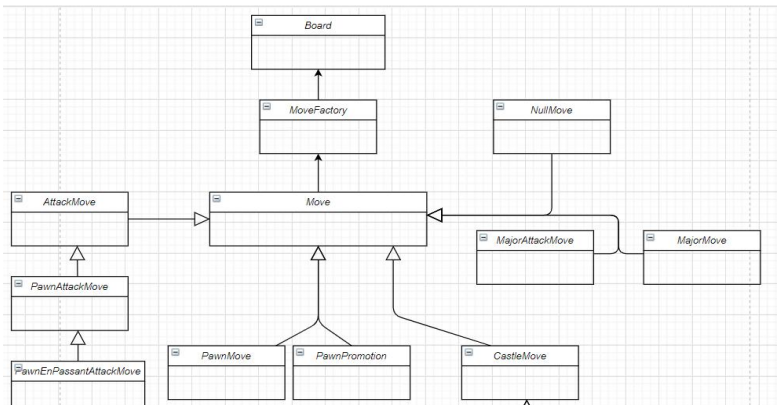


Πιο συγκεκριμένα οι σχέσεις ανα πακέτο του Board με τα Tiles:
Που αναπαριστά τα ιδιαίτερα χαρακτηριστικά και την σχέση που έχει με τα 64 Tiles

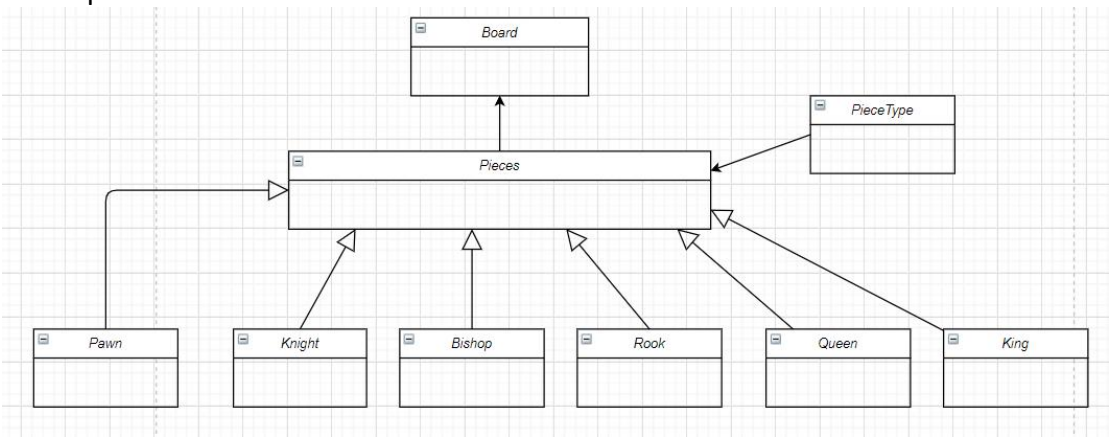


Η σχέση του Board με τις Moves :

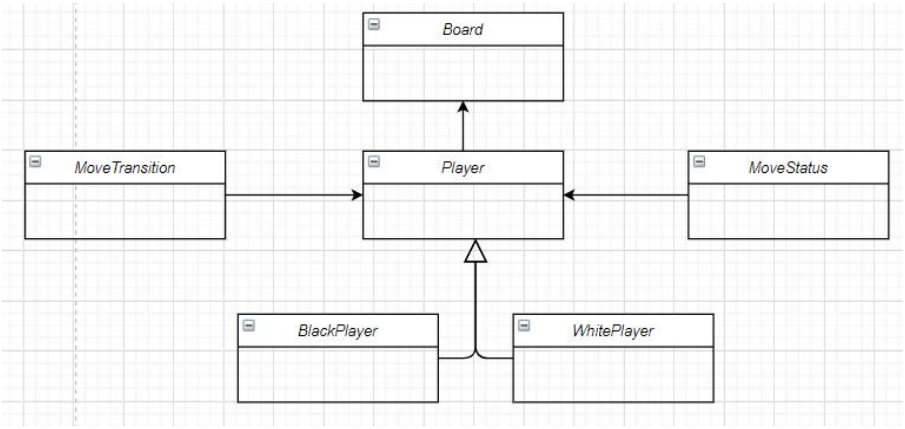
Το Board είναι υπεύθυνο να γνωρίζει τι τύπος κινήσεις έγινε στην σκακιέρα και από ποιον παίχτη και ποιο κομμάτι μετακίνησε παρακάτω η Moves θα κρίνει αν αυτή η κίνηση είναι διαθέσιμη και μπορεί να πραγματοποιηθεί. Σε κάθε περίπτωση διασφαλίζεται η ομαλή διεξαγωγή της κίνησης ακόμα και αν είναι μια ακραία περίπτωση.



Board με Pieces



Board με Player:



Μεθοδολογία για το χτίσιμο της εφαρμογής και περιγραφή κλάσεων

Tile (abstract)
#Integer tileCoordinate -Map<Integer, EmptyTile> EMPTY_TILES_CASHE
+boolean isTileOccupied() +Piece getPiece() +Integer getTileCoordinate() +String toString()

tileCoordinate : Συντεταγμένες του τετραγώνου
EMPTY_TILES_CASHE : Ο χάρτης με τα κενά τετράγωνα

EmptyTile
+boolean isTileOccupied() +Piece getPiece() +String toString()

OccupiedTile
-Piece pieceOnTile
+boolean isTileOccupied() +Piece getPiece() +String toString()

pieceOnTile : Ο τύπος του πιονιού πάνω στο τετράγωνο (Στρατιώτης , Ίππος , Αξιωματικός, Πύργος, Βασίλισσα, Βασιλιάς.)

Alliance (enumaration)
+WHITE +BLACK
+ int getDirection() + int getOppositeDirection() + boolean isWhite() + boolean isBlack() + boolean isPawnPromotionSquare(int position) + Player choosePlayer(WhitePlayer whitePlayer, BlackPlayer blackPlayer) + int pawnBonus(int position) +int knightBonus(int position) +int bishopBonus(int position) +int queenBonus(int position) +int kingBonus(int position)

Αναπαριστά τα τετράγωνα τις σκακιέρας,
για κάθε Tile γνωρίζουμε την συντεταγμένη του και έχουμε ένα
HashMap που αποτελείτε από Integer και ένα EmptyTile για την αρχηκοποίηση των 64 EmtyTile. Το κάθε Tile γνωρίζει αν είναι EmptyTile συνεπώς δεν έχει κάποιο Piece πάνω του ή αν είναι OccupiedTile που το οποίο έχει ένα Piece στην συντεταγμένη του.

Board
-List<Tile> -Collection<Piece> whitePieces -Collection<Piece> blackPieces -Int2ObjectMap<Piece> boardConfig -WhitePlayer whitePlayer -BlackPlayer blackPlayer -Player currentPlayer -Move transitionMove -Pawn enPassantPawn
+Player whitePlayer() +Player blackPlayer() +Pawn getEnPassantPawn() +Player getCurrentPlayer() +Collection<Piece> getBlackPieces() +Collection<Piece> getWhitePieces() +Piece getPiece(int coordinate) +Move getTransitionMove() +Iterable<Piece> getAllPieces() +Collection<Move>calculateLegalMoves(Collection<Piece> pieces) +Collection<Piece>calculateActivePieces(List<Tile> gameBoard, Alliance alliance) +Tile getTile(int tileCoordinate) +List<Tile> createGameBoard(Builder builder) +Board createStandardBoard() +Iterable<Move> getAllLegalMove()

gameBoard : Το ταμπλό της παρτίδας.
whitePieces : Μια λίστα με τα λεύκα πιόνια.
blackPieces : Μια λίστα με τα μάυρα πιόνια
whitePlayer : Λέυκος παίχτης
blackPlayer : Μάυρος παίχτης
currentPlayer : Καθορίζει την σειρά
transitionMove : Κίνηση του χρήστη

Board (ταμπλό) το οποίο αποτελείτε από εξήντα τέσσερα Tile (τετράγωνα) (κενά), είναι υπεύθυνο να βάλει τα πιόνια στα Tile (τετράγωνα) και αφου τοποθετηθουν αυτοματος γνωρίζει ποια τετράγωνα είναι απασχολημένα και με ποια πιονια. Να υπολογίσει τα ενεργά πιόνια και για τους δυο παίχτες και αντίστοιχα να υπολογίσει και τις επιτρεπτές κινήσεις που είναι διαθέσιμες σε εκείνο το στιγμιότυπο και να δώσει την σειρά στον παίχτη που είναι να κάνει κίνηση.

BoardUtils
+List<> FIRST_COLUM +List<> SECOND_COLUMN +List<> THIRD_COLUMN +List<> FOURTH_COLUMN +List<> FIFTH_COLUMN +List<> SIXTH_COLUMN +List<> SEVENTH_COLUMN +List<> EIGHTH_COLUMN +List<> FIRST_ROW +List<> SECOND_ROW +List<> THIRD_ROW +List<> FOURTH_ROW +List<> FIFTH_ROW +List<> SIXTH_ROW +List<> SEVENTH_ROW +List<> EIGHTH_ROW +Map<String,Integer> POSITION_TO_COORDINATE +final int NUM_TILES +final int NUM_TILES_PER_ROW +final int START_TILE_INDEX
+List<> initColumn(int columnNumber) +List<> initRow(int rowNumber) +Map<String, Integer> initializePositionToCoordinateMap() + boolean isThreatenedBoardImmediate(Board board) +boolean kingThreat(Move move) +int getCoordinateAtPosition(String position) +boolean isEndGame(Board board)

Λιστα με της γραμμες και τις στηλες της σκακιέρας

Builder
+ Int2ObjectMap<Piece> boardConfig +Alliance nextMoveMaker +Pawn enPassantPawn +Move transitionMove
+Builder setPiece(Piece piece) +Builder setMoveMaker(Alliance nextMoveMaker) +void setEnPassantPawn(Pawn enPassantPawn) +Builder setMoveTransition(final Move transitionMove)

Move (abstract)
#Board board #Piece movedPiece #int destinationCoordinate #boolean isFirstMove +Move NULL_MOVE
+Board getBoard() +int getCurrentCoordinate() +int getDestinationCoordinate() +Piece getMovedPiece() +boolean isAttack() +boolean isCastlingMove() +Piece getAttackedPiece() +Board execute() +Board undo()

board : το στιγμιότυπο της σκακιέρας .
movedPiece : το πιόνι που θα κινηθεί.
destinationCoordinate : ο προορισμός της κίνησης.
isFirstMove : Αν είναι η πρώτη κίνηση ενός πιονιού - πύργου - βασιλιά
NULL_MOVE

MajorMove

AttackMove
+Piece attackedPiece
+boolean isAttack() +Piece getAttackedPiece() +String toString()

PawnMove
+Boolean equals(Object o)

PawnAttackMove
+String toString()

MajorAttackMove

+String toString()

PawnEnPassantAttackMove
+String toString() +Board execute() +Board undo()

PawnPromotion
+promotionMove +Pawn promotedPawn
+String toString() +Board execute() + boolean isAttack() +Piece getAttackedPiece()

PawnJump
+String toString() +Board execute()

CastleMove
#Rook castleRook #int castleRookStart #int castleRookDestination
+Rook getCastleRook() +boolean isCastlingMove() +Board execute()

KingSideCastle
+Boolean equals(Object o)

QueenSideCastle
+Boolean equals(Object o)

NullMove
+getCurrentCoordinate() +Board execute()

MoveFactory
-Move NULL_MOVE

+ Move createMove(Board board , int currentCoordinate , int destinationCoodinate) +Move getNullMove()

Move (κινήσεις) αποτελείτε από το ταμπλό ώστε να γνωρίζει πιο πιόνι θα κινηθεί και που θα καταλήξει να βρίσκετε έπειτα από την κίνηση που θα γίνει.

Έχουμε την απλή κίνηση να αναπτύξω ένα πιόνι είτε αν υπάρχει η δυνατότητα να αιχμαλωτίσω κάποιο εχθρικό καθώς υπάρχουν οι κάνονες για το Pawn (πιόνι) , enPassnt, αναβάθμιση (promotion) και διπλό άνοιγμα. Τέλος υλοποιείται το ροκέ.

Piece (abstract)
#Integer piecePosition #Alliance pieceAlliance #Boolean isFirstMove #PieceType pieceType
+Integer getPiecePosition() +Alliance getPieceAlliance() +Boolean isFirstMove() +PieceType getPieceType() +Integer getPieceValue() +Collection<Move> calculateLegalMoves(Board board) +Piece movePiece(Move move)

piecePosition : θέση στην σκακιέρα του πιονιού
pieceAlliance : με ποια συμμαχία είναι Άσπρα/Μαύρα
isFirstMove : αν είναι η πρώτη κίνηση του
pieceType : ο τύπος του κομματιού

PieceType (enum)
-String pieceName; -Integer pieceValue;
+Boolean equals(Object o) +PAWN("P",100) +KNIGHT("N",320) +BISHOP("B",350) +ROOK("R",500) +QUEEN("Q",900) +KING("K",20000) +String toString() +Integer getPieceValue()

Pawn
-Integer[] CANDIDATE_MOVE_COORDINATES
+String toString() +Collection<Move> calculateLegalMoves(Board board) +Pawn movePiece(Move move) +Piece getPromotionPiece() +Integer locationBonus()

Knight
-Integer[] CANDIDATE_MOVE_COORDINATES
+String toString() +Collection<Move> calculateLegalMoves(Board board) +Knight movePiece(Move move) +Boolean isFirstColumnExlcusion(int currentPosition,int candidateOffset) +Boolean isSecondColumnExlcusion(int currentPosition,int candidateOffset) +Boolean isSevethColumnExlcusion(int currentPosition,int candidateOffset) +Boolean isEighthColumnExlcusion(int currentPosition, int candidateOffset) +Integer locationBonus()

Bishop
-Integer[] CANDIDATE_MOVE_COORDINATES
+String toString() +Collection<Move> calculateLegalMoves(Board board) +Bishop movePiece(Move move) +Boolean isFirstColumnExlcusion(int currentPosition,int candidateOffset) +Boolean isEighthColumnExlcusion(int currentPosition, int candidateOffset) +Integer locationBonus()

Rook
-Integer[] CANDIDATE_MOVE_COORDINATES
+String toString() +Collection<Move> calculateLegalMoves(Board board) +Rook movePiece(Move move) +Boolean isFirstColumnExclusion(int currentPosition,int candidateOffset) +Boolean isEighthColumnExlcusion(int currentPosition, int candidateOffset) +Integer locationBonus()

Queen
-Integer[] CANDIDATE_MOVE_COORDINATES
+String toString() +Collection<Move> calculateLegalMoves(Board board) +Queen movePiece(Move move) +Boolean isFirstColumnExclusion(int currentPosition,int candidateOffset) +Boolean isEighthColumnExlcusion(int currentPosition,int candidateOffset) +Integer locationBonus()

King
-Integer[] CANDIDATE_MOVE_COORDINATES -Boolean kingSideCastleCapable -Boolean queenSideCastleCapable -Boolean isCastled
+String toString() +Collection<Move> calculateLegalMoves(Board board) +Boolean isCastled() +Boolean isKingSideCastleCapable() +Boolean isQueenSideCastleCapable() +Boolean isFirstColumnExlcusion(int currentPosition, int candidateOffset) +Boolean isEighthColumnExlcusion(int currentPosition, int candidateOffset) +King movePiece(Move move) +Integer locationBonus()

Pieces (πιόνια) κάθε πιόνι γνωρίζει τι τύπου είναι (Pawn, Rook , Knight, Bishop, Queen , King). Σε πια θέση είναι στο ταμπλό, το χρώμα του και αν είναι η πρώτη κίνηση που έχει κάνει στο παιχνίδι και την άξια του.

Pawn (πιόνι) έχει μια λίστα με τις κινήσεις που μπορεί να κάνει στο ταμπλό οριζόντια κατά ένα ή διαγώνια όταν πρόκειται να αιχμαλωτιστεί άλλο πιόνι. Όπως το αν είναι η πρώτη του κίνηση να κάνει ένα διπλό άνοιγμα στα τετράγωνα , την απλή κίνηση ένα τετράγωνο, επίθεση διαγώνια εάν υπάρχει άλλο Piece και είναι του αντιθέτου χρώματος , το en passant (Το πιόνι που μπορεί να αιχμαλωτιστεί από εχθρικό πιόνι αν κινηθεί κατά ένα τετράγωνο μπορεί να συλληφθεί με τον ίδιο τρόπο αν κινηθεί δύο τετράγωνα όπως αν βρισκόταν στο ενδιάμεσο τετράγωνο) και τέλος όταν ένα πιόνι φτάσει στο τετράγωνο αναβαθμίσεις που μπορεί να γίνει Queen - Rook - Bishop - Knight.

Knight (ίππος) έχει μια λίστα με τις κινήσεις που μπορεί να κάνει στο ταμπλό σαν ένα κεφάλαιο γάμα (δυο μπροστά και ένα αριστερά - δέξια) σε οποία κατεύθυνση θέλει καθώς έχει και την ιδιαιτερότητα μεταξύ των άλλων πιονιών να μπορεί να αγνοεί τα εμπόδια που είναι γύρω του.

Bishop (αξιωματικός) έχει μια λίστα με τις κινήσεις που μπορεί να κάνει στο ταμπλό διαγώνια στο χρώμα του τετραγώνου που αρχικοποιείται.

Rook (πύργος) έχει μια λίστα με τις κινήσεις που μπορεί να κάνει στο ταμπλό οριζόντια ή κάθετα στην στυλή - γραμμή που βρίσκεται

Queen (βασίλισσα) έχει μια λίστα με τις κινήσεις που μπορεί να κάνει στο ταμπλό που στην ουσια ειναι ενας συνδιασμος ενος Bishop (αξιοματικου) και ενος Rook (πυργου).

King (βασιλιάς) έχει μια λίστα με τις κινήσεις που μπορεί να κάνει στο ταμπλό που είναι σαν να έχουμε μια Queen (βασίλισσα) που μπορεί να κινείται σε όλες τις κατευνάσεις απλός κατά ένα τετράγωνο.

Player (abstract)
#Board board #King playerKing #Collection<Move> legalMoves #Boolean isInCheck
#Collection<Move>calculateAttacksOnTile(int,piecePosition,Collection<Move>moves) -King establishKing() +Boolean isMoveLegal(Move move) +Boolean isInCheck() +Boolean isInCheckMate() +Boolean isInStalemate() +Boolean isKingSideCastleCapable() +Boolean isQueenCastleCapable() #Boolean hasEscapesMoves() +Boolean isCastled() +MoveTransition makeMove(Move move) +MoveTransition unMakeMove(Move move) +Collection<Piece> getActivePieces() +Alliance getAlliance() +Player getOpponent() +Collection<Move>calculateKingCastles(Collection<Move>playerLegals, Collection<Move> opponentsLegals)

board : Η σκακιέρα για να γίνει αναφορά.
playerKing : πρέπει να γνωρίζω αν υπάρχει ο Βασιλιάς του κάθε χρήστη
legalMoves : εύρος επιτρεπτών κινήσεων
isInCheck : αν βρίσκεται ο βασιλιάς υπό κάποια απειλή

WhitePlayer
+Collection<Piece> getActivePieces() +Alliance getAlliance() +Player getOpponent() +Collection<Move> calculateKingCastles(Collection<Move>playerLegals, Collection<Move> opponentsLegals)

BlackPlayer
+Collection<Piece> getActivePieces() +Alliance getAlliance() +Player getOpponent() +Collection<Move> calculateKingCastles(Collection<Move>playerLegals, Collection<Move> opponentsLegals)

Για τους Player (παίχτες) κάθε παιχτείς γνωρίζει τις επιτρεπτές τους κινήσεις, έχουν άμεση σχέση με το Board (ταμπλό) και αναγκαστικά πρέπει να υπάρχει ένας King (βασιλιάς) στο παιχνίδι και να μην βρίσκετε σε σαχ ή ρουά (check , το αναφέρουμε όταν ο αντίπαλος βασιλιάς βρίσκετε υπό απειλεί). Κάθε παίχτης έχει δικαίωμα μονό για μια κίνηση που αυτή μπορεί να θεωρηθεί ως έγκυρη (Done), μη έγκυρη (ILLEGAL_MOVE), είτε να τον αφήσει σε ρουά (LEAVES_PLAYER_IN_CHECK). Σε κάθε περίπτωση το σύστημα αντιμετωπίζει τις κινήσεις διαφορετικά στην περίπτωση του Done την εκτελεί, στην ILLEGAL_MOVE τον παραπέμπει να ξανά κάνει κάποια κίνηση και στην LEAVES_PLAYER_IN_CHECK τον αναγκάζει να μετακινήσει τον βασιλιά του είτε να μπλοκάρει την επίθεση με ένα άλλο πιόνι. Ακόμα γνωρίζει αν βρίσκετε σε ρουά , ματ ή αδιέξοδο (stalemate) , με την λέξη ρουά εννοούμε ότι ένας βασιλιάς βρίσκεται υπό την απειλή επίθεσης και ταυτόχρονα έχει κάποιο τετράγωνο διαφύγεις ή κάποιο πιόνι μπορεί να μπλοκάρει την απειλή, ματ ονομάζουμε όταν ο βασιλιάς δέχεται απειλή και δεν έχει κάποιο τετράγωνο διαφυγές άλλα ούτε κάποιο πιόνι μπορεί να μπλοκάρει την απειλή και αδιέξοδο (stalemate) όταν ένας βασιλιάς δεν δέχεται κάποια άμεση απειλή άλλα ταυτόχρα ο παίχτης δεν έχει κάποια έγκυρη κίνηση να εκτέλεση. Μπορεί να εκτελέσει μίκρο ροκέ που βρίσκετε από την μεριά του βασιλιά μονό και μονό αν ο βασιλιά δεν έχει κάνει κάποια κίνηση το ίδιο ισχύει και για τον αντίστοιχο πύργο και τα τετράγωνα που είναι ανάμεσα σε πύργο και βασιλιά πρέπει να είναι διαθέσιμα ώστε να γίνει αλλαγή τετραγώνων ανάμεσα σε βασιλιά και πύργο “O-O-O” και δεν δέχεται κάποια άμεση απειλή από τον αντίπαλο στα τετράγωνα που θα πραγματοποιηθεί το ροκέ ή το μεγάλο ροκέ από την μεριά της βασίλισσας.

MoveTransition
-Board fromBoard -Board transitionBoard -Move move -MoveStatus moveStatus
+ Board getFromBoard() +MoveStatus getMoveStatus() +Board getTransitionBoard()

Board fromBoard : από την σκακιέρα που έχω το στιγμιότυπο
Board transitionBoard : το επόμενο στιγμιότυπο βάση της κίνησης
move : Η κίνηση επιλέχτηκε
moveStatus : η κατάσταση της κίνησης

MoveStatus
DONE, ILLEGAL_MOVE, LEAVES_PLAYER_IN_CHECK

MinMax
-BoardEvaluator boardEvaluator -int searchDepth -BoardEvaluator evaluator -long boardsEvaluated -long executionTime
+Move execute(Board board) : bestMove +Boolean isEndGameScenario(Board board) +int min(Board board, int depth) +int max(Board board, int depth)

StandarBoardEvaluator
-int CHECK_MATE_BONUS -int CHECK_BONUS -int DEPTH_BONUS -int CASTLE_BONUS -int TWO_BISHOPS_BONUS -int MOBILITY_MULTIPLIER -int ATTACK_MULTIPLIER -StandarBoardEvaluator INSTANCE
+int evaluate(Board board, int depth) +String evaluationDetails(Board board, int depth) +int scorePlayer(Player player, int depth) +int attacks(Player player) +int pieceEvaluations(Player player) +int mobility(Player player) +int mobilityRatio(Player player) +int kingThreats(Player player, int depth) +int check(Player player) +int castle(Player player) +int kingSafety(Player player) +int pieceValue(Player player) +int checkMate(Player player, int depth) +int depthBonus(int depth)

<<BoardEvaluator>>
int evaluate(Board board, int depth)

<< MoveStrategy>>
Long getNumBoardsEvaluated() Move execute(Board board)

Game
#PGNGameTags tags #List<String> moves #String winner
+List<String> getMoves() +String getWinner() +String calculateWinner(String gameOutcome)

FenUtilities
+Board createGameFromFEN(String fenString) +String createFENFromGame(Board board) +String randomGenaratorGame(Board board) +Board parseFEN(final String fenString) +Alliance moveMaker(String moveMakerString) +boolean whiteKingSideCastle(String fenCastleString) +boolean whiteQueenSideCastle(String fenCastleString) +boolean blackKingSideCastle(String fenCastleString) +boolean blackQueenSideCastle(final String fenCastleString) +String calculateCastleText(Board board) +String calculateEnPassantSquare(Board board) +String calculateBoardText(Board board) + String calculateCurrentPlayerText(Board board)

InvalidGame
+String malformedGameText
+boolean isValid()

PGNGameTags
-Map<String,String> gameTags
+PGNGameTags(TagsBuilder builder)

TagsBuilder
+Map<String,String> gameTags
+TagsBuilder addTag(String tagKey, String tagValue) +PGNGameTags build()

PGNUtilities
+writeGameToPGNFile(File pgnFile,MoveLog moveLog, Player player, Board board) -String calculateEventString() -String calculateDateString() -String calculatePlyCountString(MoveLog moveLog) -String calculatePlyCountString(MoveLog moveLog) -String calculateResult(final MoveLog moveLog , final Board board)

Η FenUtilities σε συνεργασία του PGNUtilities μας δίνει την δυνατότητα μέσω της πλατφόρμας να δημιουργήσουμε σε οποίο στιγμιότυπο θέλουμε ένα Fen το οποίο είναι επεξεργάσιμο και από άλλα συστήματα - πλατφόρμες. Ακόμα μπορεί να λάβει απο κάποιο εξωσυστημικό ένα FEN και να το εφαρμόσει στην σκακιέρα εφόσον πέραση τον έλεγχο εγκυρότητας του FEN που διαθέτει. Υπάρχει και η μορφή του PGN που στην ουσία είναι μια ουρά με κινήσεις που αποτυπώνεται σε ένα text file επίσης μπορεί να το λάβει κάποιο άλλο σύστημα και να το επεξεργαστή. Πέρα από τις κινήσεις αποτυπώνεται το Event, οι παίχτες που συμμετείχαν, πόσες κινήσεις έγιναν στην παρτίδα, η ημερομηνία που παίχτηκε η παρτίδα και το αποτέλεσμα αυτής.

Table
-JFrame gameFrame -GameHistoryPanel gameHistoryPanel -TakenPiecesPanel takenPiecePanel -BoardPanel boardPanel -MoveLog moveLog -GameSetup gameSetup -Board chessBoard -DebugPanel debugPanel -Piece sourceTile -Piece humanMovedPiece -BoardDirection boardDirection -Move computerMove -boolean highLightLegalMoves
+JMenuBar createTableMenuBar() +void center(final JFrame frame) +JMenu createFileMenu() +JMenuItem openFEN() +JMenuItem saveToPGN() +JMenuItem createChess960() +JMenuItem exitMenuItem() +JButton resignButton() +JButton drawButton() +JMenu createPreferencesMenu()

+flipBoardMenuItem() +JCheckBoxMenuItem legalMoveHighlightCheckBox() +JMenuItem resetMenuItem() +JMenuItem legalMovesMenuItem() +JMenuItem undoMoveMenuItem() +JMenuItem setupGameMenuItem() +JMenuItem evaluateBoardMenuItem() +JMenuItem escapeAnalysis() +String playerInfo() +void undoAllMoves() +void undoLastMove() +void updateGameBoard(Board board) +void updateComputerMove(Move move) +void setupUpdate(GameSetup gameSetup) +void moveMadeUpdate(PlayerType playerType)

TableGameAIWatcher
+void update(Observable o, Object arg) +boolean isKingPat() +boolean isKingKnightPat() +boolean isKingBishopPat() +GameHistoryPanel getGameHistoryPanel() +TakenPiecesPanel getTakenPiecesPanel() +DebugPanel getDebugPanel() +DebugPanel getDebugPanel()

PlayerType (enum)
HUMAN, COMPUTER

AIThinkTank
#Move doInBackground() +void done()

BoardDirection (enum)
NORMAL, FLIPPED
traverse(List<TilePanel> boardTiles) BoardDirection opposite()

BoardPanel
+List<TilePanel> boardTiles
+BoardPanel()

+void drawBoard(final Board board) +void setTileDarkColor(Board board,Color darkColor) +void setTileLightColor(Board board,Color lightColor)

MoveLog
+List<Move> moves
+List<Move> getMoves() +void addMove(Move move) +int size() +void clear() +Move removeMove(int index) +boolean removeMove(Move move)

TilePanel
-int tileId
+TilePanel() +void drawTile(Board board) +void setLightTileColor(Color color) +void setDarkTileColor(Color color) +void highlightTileBorder(Board board) +void highlightAIMove() +void assignTilePieceIcon(Board board) +void highlightLegals(Board board) +Collection<Move> pieceLegalMoves(Board board) +void assignTileColor()

DebugPanel
-JTextArea jTextArea
+DebugPanel() +void redo() void update(Observable obs, Object obj)

GameHistoryPanel
-DataModel model -JScrollPane scrollPane
+GameHistoryPanel() +redo(Board board, MoveLog moveHistory) +String calculateCheckAndCheckMateHash(Board board)

DataModel
-List<Row> values final String[] NAMES = {"White","Black"}
+DataModel() +void clear() +int getRowCount() +Object getValueAt(final int row , final int column)

+void setValueAt(Object aValue, int row , int column) +Class<?> getColumnClass(int column) +String getColumnName(int column)

Row
+String whiteMove +String blackMove
+Row() +String getWhiteMove() +String getBlackMove() +void setWhiteMove(String move) void setBlackMove(String move)

GameSetup
-PlayerType whitePlayerType -PlayerType blackPlayerType -JSpinner searchDepthSpinner -String HUMAN_TEXT = "Human" -String COMPUTER_TEXT = "Computer"
+GameSetup() +boolean isAIPlayer(Player player) +PlayerType getWhitePlayerType() +PlayerType getBlackPlayerType() +JSpinner addLabeledSpinner(Container c, String label, SpinnerModel model) +int getSearchDepth()

TakenPiecesPanel
-JPanel northPanel -JPanel southPanel
+TakenPiecesPanel() +void redo(MoveLog moveLog)

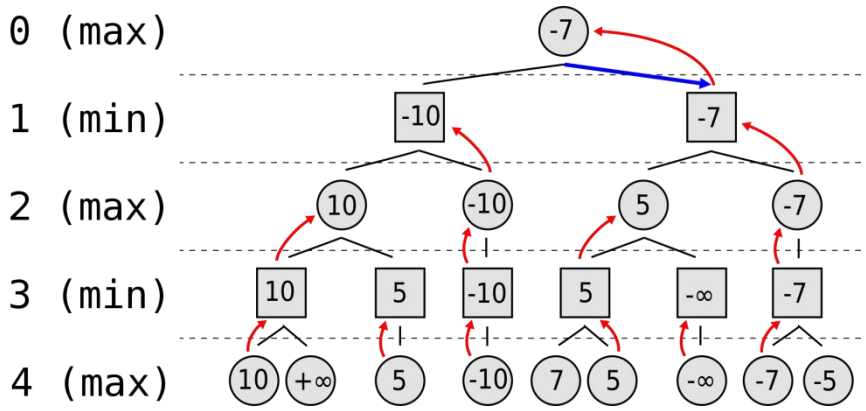
Αλγοριθμος MinMax

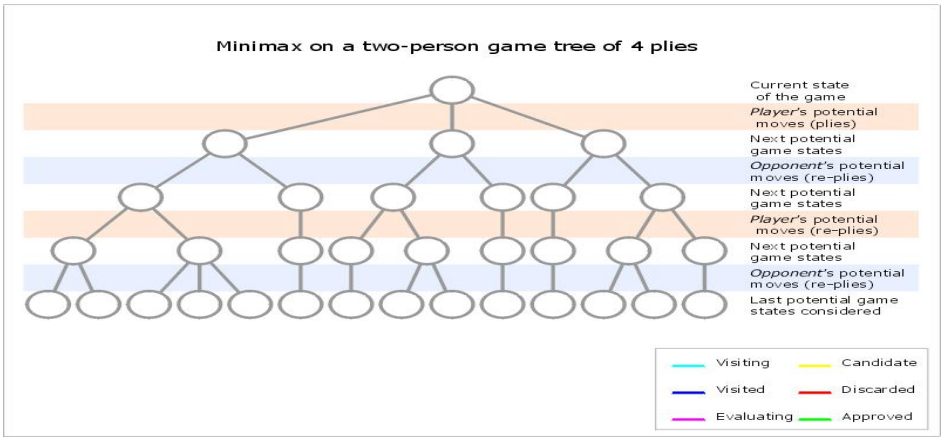
Ο Minimax είναι ένα είδος αλγορίθμου backtracking που χρησιμοποιείται στη λήψη αποφάσεων και στη θεωρία του game theory για να βρει τη βέλτιστη κίνηση για έναν παίκτη, με την προϋπόθεση ότι ο αντίπαλός σας παίζει επίσης με τον καλύτερο τρόπο. Χρησιμοποιείται ευρέως σε παιχνίδια που βασίζονται σε δύο παίκτες όπως το σκάκι.
Στον Minimax οι δύο παίκτες ονομάζονται μεγιστοποιητής και ελαχιστοποιητής. Ο μεγιστοποιητής προσπαθεί να πάρει την υψηλότερη δυνατή βαθμολογία, ενώ ο ελαχιστοποιητής προσπαθεί να κάνει το αντίθετο και να πάρει τη χαμηλότερη δυνατή βαθμολογία.

Κάθε κατάσταση στιγμιοτύπου έχει μια τιμή που σχετίζεται με αυτήν. Σε μια δεδομένη κατάσταση εάν ο μεγιστοποιητής έχει το πάνω χέρι τότε, το σκορ του πίνακα θα έχει την τάση να είναι κάποια θετική τιμή. Εάν ο ελαχιστοποιητής έχει το πάνω χέρι σε αυτήν την κατάσταση του πίνακα τότε θα έχει την τάση να έχει κάποια αρνητική τιμή. Οι τιμές του πίνακα υπολογίζονται από ορισμένες ευρετικές που είναι μοναδικές για κάθε τύπο παιχνιδιού.

Ψευδοκωδικας

```
function minimax(node, depth, maximizingPlayer) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value := −∞
        for each child of node do
            value := max(value, minimax(child, depth − 1, FALSE))
        return value
    else (* minimizing player *)
        value := +∞
        for each child of node do
            value := min(value, minimax(child, depth − 1, TRUE))
        return value
```





Περιπτώσεις χρήσης ενός actor

Στο σύστημα η επιλογή της καλύτερης κίνησης για τα λεύκα ειναι με βάση την μεγαλύτερη τιμή που θα δώσει ο αλγόριθμος με την λίστα των διαθέσιμων κινήσεων και αντίστοιχα για τα μαύρα η μικρότερη τιμή Η λίστα προκύπτει από τις διαθέσιμες κινήσεις που έχει ο κάθε παίχτης και η τιμη βγαίνει από το κόστος του κάθε κομματιού σε συνδυασμό με το πλήθος των κινήσεων που έχει. Τα βαρύ είναι σταθερά και δεν υπολογίζεται καμιά μεταβολή.

Λειτουργιές εφαρμογής σε χρήστη

Ένας actor μπορεί μέσο της εφαρμογής να :
Από το μενού “File” και το κουμπί “Load FEN File” να φορτώσει κάποιο fen ώστε να ξεκινήσει η παρτίδα από το σημείο που υποδεικνύει το fen.
Να αποθήκευση την παρτίδα σε οποίο στιγμιότυπο της βρίσκεται από το κουμπί “Save Game”.
Να δημιουργήσει μια παρτίδα τύπου chess960 από το κουμπί “Chess960”.
Και από το κουμπί “Exit” να κλείσει την εφαρμογή.
Από την δεύτερη καρτέλα του μενού “Preferences” μπορεί να:
Γυρίσει την σκακιέρα ώστε μπροστά του να έχει τα μαύρα πιόνια από το κουμπί “Flip Board”.
Μπορεί να ενεργοποίηση τις επιτρεπόμενες κινήσεις που διαθέτει κάθε πιόνι εφόσον το επιλέξει πρώτα, από το κουμπί “Highlight Legal Moves”.
Και να αλλάξει χωρά στα τετράγωνα της σκακιέρας “choose colors”.
Από την τρίτη καρτέλα του μενού “Option” μπορεί να :
Ξεκινήσει μια νέα παρτίδα από το “New Game”.
Να δει την τρέχον κατάσταση του παιχνιδιού και πόσες επιτρεπτές κινήσεις υπάρχουν από τους παίχτες και στο τέλος ένα Fen από το “Current State”.
Να πάρει μια κίνηση πίσω από το “Undo”.
Μπορεί να παίξει μια παρτίδα με τον υπολογιστή είτε να βάλει τον υπολογιστή να παίξει με τον ευτού του σε έξι διαφορετικά επίπεδα με το “Setup Game”.
Και τέλος να βάλει την μηχανή να υπολογίσει μερικά στατιστικά για το ποιος έχει το πλεονέκτημα με το “Evaluate Board”

Στατιστικά Κωδικά

Application Metrics

Note: Further Application Statistics are available.

 # Logical lines of Code
1 411
0 (NotMyCode)
Estimated Dev Effort 34d

 # Types
55
1 Projects
9 Packages
298 Methods
92 Fields
29 Source Files
119 Third-Party Elements

Comment
4.73%
70 Lines of Comment

Debt
2.75%
Rating A
Debt 7h 31min
The technical-debt is incomplete because no coverage data specified.

Coverage
N/A because no coverage data specified

Method Complexity
16 Max
1.76 Average

Quality Gates

Fail	1
Warn	0
Pass	4

Rules

Critical	1
Violated	10
Ok	198

Issues

All	61
Blocker	0
Critical	0
High	1
Medium	60
Low	0

Quality Gates summary

10 0 1

Summary of Rules or Queries with Error (syntax error, exception thrown, time-out)

Name	Group
Avoid packages dependency cycles	Project Rules \ Architecture

Showing 1 to 1 of 1 entries

Rules can be checked live at development-time, from within Visual JArchitect. Online documentation.

Name	# Issues	Elements	Group
Avoid types too big	1	type	Project Rules \ Code Smell
Avoid types with too many methods	1	type	Project Rules \ Code Smell
Avoid types with too many fields	1	type	Project Rules \ Code Smell
Avoid methods potentially poorly commented	17	methods	Project Rules \ Code Smell
Avoid types with poor cohesion	1	type	Project Rules \ Code Smell
Nested types should not be visible	18	types	Project Rules \ Object Oriented Design
Instance fields should begin with a lower character	19	fields	Project Rules \ Naming Conventions
Methods name should begin with an lower character	1	method	Project Rules \ Naming Conventions
Avoid types with name too long	1	type	Project Rules \ Naming Conventions
Avoid having different types with same name	1	type	Project Rules \ Naming Conventions

Showing 1 to 10 of 10 entries

17 methods	Percentage Comment	# lines of code (LOC)	# lines of comment	nbLinesOfCodeNotCommented	Debt	Annual Interest	Full Name
parseFEN(String)	0	53	0	53	5min	16min	stamatis.chess.pgn.(String)
execute(Board)	0	38	0	38	3min 48s	16min	stamatis.chess.engi.AlphaBetaWithMov)
execute(Board)	9.09	40	4	36	3min 36s	9min	stamatis.chess.engi.StockAlphaBeta.ex
testInvalidBoard()	0	33	0	33	3min 18s	16min	stamatis.chess.engi.testInvalidBoard()
execute(Board)	0	33	0	33	3min 18s	16min	stamatis.chess.engi.execute(Board)
testAnotherGame()	0	30	0	30	3min 0s	16min	TestCheckmate.test
testBlackburneShillingMate()	0	30	0	30	3min 0s	16min	TestCheckmate.test()
testLegalsMate()	0	28	0	28	2min 48s	16min	TestCheckmate.test
testSevenMoveMate()	0	28	0	28	2min 48s	16min	TestCheckmate.test
testBlackQueenSideCastle()	0	27	0	27	2min 42s	16min	TestCastling.testBla
Table()	0	27	0	27	2min 42s	16min	stamatis.com.gui.Ta

Παράρτημα Κώδικα

Η κλάση που αναπαριστά την σκακιέρα Board

```
public class Board {

   private final List<Tile> gameBoard;
   private final Collection<Piece> whitePieces;
   private final Collection<Piece> blackPieces;
   private final Int2ObjectMap<Piece> boardConfig;
   private final WhitePlayer whitePlayer;
   private final BlackPlayer blackPlayer;
   private final Player currentPlayer;
   private final Move transitionMove;
   private final Pawn enPassantPawn;

   private Board(final Builder builder) {
      this.gameBoard = createGameBoard(builder);
      this.whitePieces = calculateActivePieces(this.gameBoard, Alliance.WHITE);
      this.blackPieces = calculateActivePieces(this.gameBoard, Alliance.BLACK);
      this.boardConfig = Int2ObjectMaps.unmodifiable(builder.boardConfig);
      this.enPassantPawn = builder.enPassantPawn;

      final Collection<Move> whiteStandarLegalMoves = calculateLegalMoves(this.whitePieces);
      final Collection<Move> blackStandarLegalMoves = calculateLegalMoves(this.blackPieces);

      this.whitePlayer = new WhitePlayer(this, whiteStandarLegalMoves, blackStandarLegalMoves);
      this.blackPlayer = new BlackPlayer(this, whiteStandarLegalMoves, blackStandarLegalMoves);

      this.currentPlayer = builder.nextMoveMaker.choosePlayer(this.whitePlayer, this.blackPlayer);

      this.transitionMove = builder.transitionMove != null ? builder.transitionMove :
Move.MoveFactory.getNullMove();
   }

   @Override
   public String toString() {
```

```
    final StringBuilder builder = new StringBuilder();
    for (int i = 0; i < BoardUtils.NUM_TILES; i++) {
      final String tileText = this.gameBoard.get(i).toString();
      builder.append(String.format("%3s", tileText));
      if ((i + 1) % BoardUtils.NUM_TILES_PER_ROW == 0) {
        builder.append("\n");
      }
    }
    return builder.toString();
  }

  public Player whitePlayer() {
    return this.whitePlayer;
  }

  public Player blackPlayer() {
    return this.blackPlayer;
  }

  public Pawn getEnPassantPawn() {
    return this.enPassantPawn;
  }

  /**
   * Who's turn is.
   *
   * @return player
   */
  public Player getCurrentPlayer() {
    return this.currentPlayer;
  }

  public Collection<Piece> getBlackPieces() {
    return this.blackPieces;
  }

  public Collection<Piece> getWhitePieces() {
    return this.whitePieces;
  }

  public Piece getPiece(final int coordinate) {
    return this.boardConfig.get(coordinate);
  } //new
  // its the move from player.

  public Move getTransitionMove() {
    return this.transitionMove;
  }//new

  public Iterable<Piece> getAllPieces() {
    return Iterables.unmodifiableIterable(Iterables.concat(this.whitePieces, this.blackPieces));
  }

  private Collection<Move> calculateLegalMoves(final Collection<Piece> pieces) {
    final List<Move> legalMoves = new ArrayList<>();

    for (final Piece piece : pieces) {
```

```
            legalMoves.addAll(piece.calculateLegalMoves(this));
        }

        return ImmutableList.copyOf(legalMoves);
    }

    /**
     *
     * @param gameBoard
     * @param alliance
     * @return Active pieces
     */
    private static Collection<Piece> calculateActivePieces(final List<Tile> gameBoard, final Alliance alliance) {
        final List<Piece> activePieces = new ArrayList<>();

        for (final Tile tile : gameBoard) {
            if (tile.isTileOccupied()) {
                final Piece piece = tile.getPiece();
                if (piece.getPieceAlliance() == alliance) {
                    activePieces.add(piece);
                }
            }
        }
        return ImmutableList.copyOf(activePieces);
    }

    public Tile getTile(final int tileCoordinate) {
        return gameBoard.get(tileCoordinate);
    }

    /**
     * Create a Chess Board with 64 Tiles.
     *
     * @param builder
     * @return List<Tiles>
     */
    private static List<Tile> createGameBoard(final Builder builder) {
        final Tile[] tiles = new Tile[BoardUtils.NUM_TILES];
        for (int i = 0; i < BoardUtils.NUM_TILES; i++) {
            tiles[i] = Tile.createTile(i, builder.boardConfig.get(i));
        }
        return ImmutableList.copyOf(tiles);
    }

    /**
     * This the default board.
     *
     * @return
     */
    public static Board createStandardBoard() {
        final Builder builder = new Builder();
        //Black pieces

        builder.setPiece(new Rook(Alliance.BLACK, 0));
        builder.setPiece(new Knight(Alliance.BLACK, 1));
        builder.setPiece(new Bishop(Alliance.BLACK, 2));
```

```
        builder.setPiece(new Queen(Alliance.BLACK, 3));
        builder.setPiece(new King(Alliance.BLACK, 4, true, true));
        builder.setPiece(new Bishop(Alliance.BLACK, 5));
        builder.setPiece(new Knight(Alliance.BLACK, 6));
        builder.setPiece(new Rook(Alliance.BLACK, 7));
        builder.setPiece(new Pawn(Alliance.BLACK, 8));
        builder.setPiece(new Pawn(Alliance.BLACK, 9));
        builder.setPiece(new Pawn(Alliance.BLACK, 10));
        builder.setPiece(new Pawn(Alliance.BLACK, 11));
        builder.setPiece(new Pawn(Alliance.BLACK, 12));
        builder.setPiece(new Pawn(Alliance.BLACK, 13));
        builder.setPiece(new Pawn(Alliance.BLACK, 14));
        builder.setPiece(new Pawn(Alliance.BLACK, 15));

        //White pieces set Up
        builder.setPiece(new Pawn(Alliance.WHITE, 48));
        builder.setPiece(new Pawn(Alliance.WHITE, 49));
        builder.setPiece(new Pawn(Alliance.WHITE, 50));
        builder.setPiece(new Pawn(Alliance.WHITE, 51));
        builder.setPiece(new Pawn(Alliance.WHITE, 52));
        builder.setPiece(new Pawn(Alliance.WHITE, 53));
        builder.setPiece(new Pawn(Alliance.WHITE, 54));
        builder.setPiece(new Pawn(Alliance.WHITE, 55));
        builder.setPiece(new Rook(Alliance.WHITE, 56));
        builder.setPiece(new Knight(Alliance.WHITE, 57));
        builder.setPiece(new Bishop(Alliance.WHITE, 58));
        builder.setPiece(new Queen(Alliance.WHITE, 59));
        builder.setPiece(new King(Alliance.WHITE, 60, true, true));
        builder.setPiece(new Bishop(Alliance.WHITE, 61));
        builder.setPiece(new Knight(Alliance.WHITE, 62));
        builder.setPiece(new Rook(Alliance.WHITE, 63));

        //white first move
        builder.setMoveMaker(Alliance.WHITE);

        return builder.build();
    }

//get all legal moves
    public Iterable<Move> getAllLegalMove() {
        return Iterables.unmodifiableIterable(Iterables.concat(this.whitePlayer.getLegalMoves(),
                this.blackPlayer.getLegalMoves()));
    }

    /**
     * Its a builder to store position and pieces to a Hash Map. example
     * [0,King]
     */
    public static class Builder {

        Int2ObjectMap<Piece> boardConfig;//new
        // Map<Integer,Piece> boardConfig;
        Alliance nextMoveMaker;
        Pawn enPassantPawn;
        Move transitionMove;//new

        /* public Builder(){
```

```
        this.boardConfig = new HashMap<>();
    }*/
        public Builder() {
            this.boardConfig = new Int2ObjectOpenHashMap<>(33, 1.0f);
        }//new

        public Builder setPiece(final Piece piece) {
            this.boardConfig.put(piece.getPiecePosition(), piece);
            return this;
        }
        //somehow you have to initiate the first player.

        public Builder setMoveMaker(final Alliance nextMoveMaker) {
            this.nextMoveMaker = nextMoveMaker;
            return this;
        }

        public Board build() {
            return new Board(this);
        }

        public void setEnPassantPawn(Pawn enPassantPawn) {
            this.enPassantPawn = enPassantPawn;
        }

        public Builder setMoveTransition(final Move transitionMove) {
            this.transitionMove = transitionMove;
            return this;
        }
        //new

    }//Builder

}//Board


public enum BoardUtils {

    INSTANCE;
    public final List<Boolean> FIRST_COLUMN = initColumn(0);
    public final List<Boolean> SECOND_COLUMN = initColumn(1);
    public final List<Boolean> THIRD_COLUMN = initColumn(2);
    public final List<Boolean> FOURTH_COLUMN = initColumn(3);
    public final List<Boolean> FIFTH_COLUMN = initColumn(4);
    public final List<Boolean> SIXTH_COLUMN = initColumn(5);
    public final List<Boolean> SEVENTH_COLUMN = initColumn(6);
    public final List<Boolean> EIGHTH_COLUMN = initColumn(7);
    public final List<Boolean> FIRST_ROW = initRow(0);
    public final List<Boolean> SECOND_ROW = initRow(8);
    public final List<Boolean> THIRD_ROW = initRow(16);
    public final List<Boolean> FOURTH_ROW = initRow(24);
    public final List<Boolean> FIFTH_ROW = initRow(32);
    public final List<Boolean> SIXTH_ROW = initRow(40);
    public final List<Boolean> SEVENTH_ROW = initRow(48);
    public final List<Boolean> EIGHTH_ROW = initRow(56);

    public static final String[] ALGEBREIC_NOTATION = initializeAlgebreicNotation();
```

```
  public static final Map<String, Integer> POSITION_TO_COORDINATE =
initializePositionToCoordinateMap();

  public static final int NUM_TILES = 64;
  public static final int NUM_TILES_PER_ROW = 8;
  public static final int START_TILE_INDEX = 0;

  /*   private static boolean[] initColumn(int columnNumber) {
     final boolean[] column =new boolean[NUM_TILES];
     do{
        column[columnNumber]= true;
        columnNumber += NUM_TILES_ROW;
     } while(columnNumber < NUM_TILES);

     return column;
  }*/
  private static List<Boolean> initColumn(int columnNumber) {
     final Boolean[] column = new Boolean[NUM_TILES];
     for (int i = 0; i < column.length; i++) {
        column[i] = false;
     }
     do {
        column[columnNumber] = true;
        columnNumber += NUM_TILES_PER_ROW;
     } while (columnNumber < NUM_TILES);
     return Collections.unmodifiableList(Arrays.asList((column)));
  }

  private static List<Boolean> initRow(int rowNumber) {
     final Boolean[] row = new Boolean[NUM_TILES];
     for (int i = 0; i < row.length; i++) {
        row[i] = false;
     }
     do {
        row[rowNumber] = true;
        rowNumber++;
     } while (rowNumber % NUM_TILES_PER_ROW != 0);
     return Collections.unmodifiableList(Arrays.asList(row));
  }

  private static Map<String, Integer> initializePositionToCoordinateMap() {
     final Map<String, Integer> positionToCoordinate = new HashMap<>();
     for (int i = START_TILE_INDEX; i < NUM_TILES; i++) {
        positionToCoordinate.put(ALGEBREIC_NOTATION[i], i);
     }
     return ImmutableMap.copyOf(positionToCoordinate);
  }

  private static String[] initializeAlgebreicNotation() {
     return new String[]{
        "a8", "b8", "c8", "d8", "e8", "f8", "g8", "h8",
        "a7", "b7", "c7", "d7", "e7", "f7", "g7", "h7",
        "a6", "b6", "c6", "d6", "e6", "f6", "g6", "h6",
        "a5", "b5", "c5", "d5", "e5", "f5", "g5", "h5",
        "a4", "b4", "c4", "d4", "e4", "f4", "g4", "h4",
        "a3", "b3", "c3", "d3", "e3", "f3", "g3", "h3",
        "a2", "b2", "c2", "d2", "e2", "f2", "g2", "h2",
```

```
        "a1", "b1", "c1", "d1", "e1", "f1", "g1", "h1"
      };
  }

  public static boolean isThreatenedBoardImmediate(final Board board) {
    return board.whitePlayer().isInCheck() || board.blackPlayer().isInCheck();
  }

  public static boolean kingThreat(final Move move) {
    final Board board = move.getBoard();
    final MoveTransition transition = board.getCurrentPlayer().makeMove(move);
    return transition.getTransitionBoard().getCurrentPlayer().isInCheck();
  }

  public static boolean isValidTileCoordinate(final int coordinate) {
    return coordinate >= START_TILE_INDEX && coordinate < NUM_TILES;
  }

  public static int getCoordinateAtPosition(final String position) {
    return POSITION_TO_COORDINATE.get(position);
  }

  public static String getPositionAtCoordinate(final int coordinate) {
    return ALGEBREIC_NOTATION[coordinate];
  }

  public static int mvvlva(final Move move) {
    final Piece movingPiece = move.getMovedPiece();
    if (move.isAttack()) {
      final Piece attackedPiece = move.getAttackedPiece();
      return (attackedPiece.getPieceValue() - movingPiece.getPieceValue() +
Piece.PieceType.KING.getPieceValue()) * 100;
    }
    return Piece.PieceType.KING.getPieceValue() - movingPiece.getPieceValue();
  }//new

  public static List<Move> lastNMoves(final Board board, int N) {
    final List<Move> moveHistory = new ArrayList<>();
    Move currentMove = board.getTransitionMove();
    int i = 0;
    while (currentMove != MoveFactory.getNullMove() && i < N) {
      moveHistory.add(currentMove);
      currentMove = currentMove.getBoard().getTransitionMove();
      i++;
    }
    return Collections.unmodifiableList(moveHistory);
  }//NEW

  public static boolean isEndGame(final Board board) {
    return board.getCurrentPlayer().isInCheckMate()
        || board.getCurrentPlayer().isInStalemate();
  }

}//BoardUtils
```

Η κλάση με όλες τις κινήσεις του παιχνιδιού

```
public abstract class Move {

   protected final Board board;
   protected final Piece movedPiece;
   protected final int destinationCoordinate;
   protected final boolean isFirstMove;

   public static final Move NULL_MOVE = new NullMove();

   private Move(final Board board, final Piece movedPiece, final int destinationCoordinate) {

      this.board = board;
      this.movedPiece = movedPiece;
      this.destinationCoordinate = destinationCoordinate;
      this.isFirstMove = movedPiece.isFirstMove();
   }

   private Move(final Board board, final int destinationCoordinate) {
      this.board = board;
      this.destinationCoordinate = destinationCoordinate;
      this.movedPiece = null;
      this.isFirstMove = false;
   }
   //an id for each piece (pos)

   @Override
   public int hashCode() {
      final int prime = 31;
      int result = 1;

      result = prime * result + this.destinationCoordinate;
      result = prime * result + this.movedPiece.hashCode();
      result = prime * result + this.movedPiece.getPiecePosition();

      return result;
   }

   @Override
   public boolean equals(final Object other) {
      if (this == other) {
         return true;
      }
      if (!(other instanceof Move)) {
         return false;
      }
      final Move otherMove = (Move) other;

      return getCurrentCoordinate() == otherMove.getCurrentCoordinate()
            && getDestinationCoordinate() == otherMove.getDestinationCoordinate()
            && getMovedPiece().equals(otherMove.getMovedPiece());
   }

   public Board getBoard() {
      return this.board;
   }

   public int getCurrentCoordinate() {
```

```
    return this.getMovedPiece().getPiecePosition();
}

public int getDestinationCoordinate() {
    return this.destinationCoordinate;
}

public Piece getMovedPiece() {
    return this.movedPiece;
}

public boolean isAttack() {
    return false;
}

public boolean isCastlingMove() {
    return false;
}

public Piece getAttackedPiece() {
    return null;
}

/**
 * recall execute to rebuild the new board after a move.
 *
 * @return Board
 */
public Board execute() {
    final Builder builder = new Builder();

    for (final Piece piece : this.board.getCurrentPlayer().getActivePieces()) {
        if (!this.movedPiece.equals(piece)) {
            builder.setPiece(piece);
        }
    }
    for (final Piece piece : this.board.getCurrentPlayer().getOpponent().getActivePieces()) {
        builder.setPiece(piece);
    }
    //move the piece it should be moved
    builder.setPiece(this.movedPiece.movePiece(this));
    builder.setMoveMaker(this.board.getCurrentPlayer().getOpponent().getAlliance());

    return builder.build();
}

/**
 * It take a move back.
 *
 * @return
 */
public Board undo() {
    final Board.Builder builder = new Builder();
    for (final Piece piece : this.board.getAllPieces()) {
        builder.setPiece(piece);
    }
    builder.setMoveMaker(this.board.getCurrentPlayer().getAlliance());
```

```
        return builder.build();
    }

    /**
     * basic attack move.
     */
    public static class MajorAttackMove extends AttackMove {

        public MajorAttackMove(final Board board,
                final Piece pieceMoved,
                final int destinationCoordinate,
                final Piece pieceAttacked) {

            super(board, pieceMoved, destinationCoordinate, pieceAttacked);

        }

        @Override
        public boolean equals(final Object other) {
            return this == other || other instanceof MajorAttackMove && super.equals(other);
        }

        /*  @Override
        public String toString(){
            return                                    movedPiece.getPieceType()                                    +
BoardUtils.getPositionAtCoordinate(this.destinationCoordinate);
        }*/
        @Override
        public String toString() {
            return movedPiece.getPieceType() + "x"
                    + BoardUtils.getPositionAtCoordinate(this.destinationCoordinate);
        }

    }//MajorAttackMove

    /**
     * basic move.
     */
    public static final class MajorMove extends Move {

        public MajorMove(final Board board, final Piece movedPiece, final int destinationCoordinate) {
            super(board, movedPiece, destinationCoordinate);
        }

        @Override
        public boolean equals(final Object other) {
            return this == other || other instanceof MajorMove && super.equals(other);
        }

        @Override
        public String toString() {
            return                                movedPiece.getPieceType().toString()                                +
BoardUtils.getPositionAtCoordinate(this.destinationCoordinate);
        }

    }//MajorMove
```

```
  public static class AttackMove extends Move {

    final Piece attackedPiece;

    public AttackMove(final Board board, final Piece movedPiece, final int destinationCoordinate,
        final Piece attackedPiece) {
      super(board, movedPiece, destinationCoordinate);
      this.attackedPiece = attackedPiece;
    }

    @Override
    public int hashCode() {
      return this.attackedPiece.hashCode() + super.hashCode();
    }

    @Override
    public boolean equals(final Object other) {
      if (this == other) {
        return true;
      }
      if (!(other instanceof AttackMove)) {
        return false;
      }
      final AttackMove otherAttackMove = (AttackMove) other;

      return super.equals(otherAttackMove) &&
getAttackedPiece().equals(otherAttackMove.getAttackedPiece());
    }

    @Override
    public boolean isAttack() {
      return true;
    }

    @Override
    public Piece getAttackedPiece() {
      return this.attackedPiece;
    }

    @Override
    public String toString() {
      return movedPiece.getPieceType() + "x"
          + BoardUtils.getPositionAtCoordinate(this.destinationCoordinate);
    }

  }//AttackMove

  public static final class PawnMove extends Move {

    public PawnMove(final Board board, final Piece movedPiece, final int destinationCoordinate) {
      super(board, movedPiece, destinationCoordinate);
    }

    @Override
    public boolean equals(final Object other) {
      return this == other || other instanceof PawnMove && super.equals(other);
    }
```

```
    @Override
    public String toString() {
      return BoardUtils.getPositionAtCoordinate(this.destinationCoordinate);
    }
  }//PawnMove

  public static class PawnAttackMove extends AttackMove {

    public PawnAttackMove(final Board board, final Piece movedPiece, final int
destinationCoordinate,
        final Piece attackedPiece) {
      super(board, movedPiece, destinationCoordinate, attackedPiece);

    }

    @Override
    public boolean equals(final Object other) {
      return this == other || other instanceof PawnAttackMove && super.equals(other);
    }

    @Override
    public String toString() {
      return  BoardUtils.getPositionAtCoordinate(this.movedPiece.getPiecePosition()).substring(0,  1)
+ "x"
          + BoardUtils.getPositionAtCoordinate(this.destinationCoordinate);
    }

  }//PawnAttackMove

  public static final class PawnEnPassantAttackMove extends PawnAttackMove {

    public PawnEnPassantAttackMove(final Board board, final Piece movedPiece, final int
destinationCoordinate,
        final Piece attackedPiece) {
      super(board, movedPiece, destinationCoordinate, attackedPiece);
    }

    @Override
    public boolean equals(final Object other) {
      return this == other || other instanceof PawnEnPassantAttackMove && super.equals(other);
    }

    @Override
    public Board execute() {
      final Builder builder = new Builder();

      for (final Piece piece : this.board.getCurrentPlayer().getActivePieces()) {
        if (!this.movedPiece.equals(piece)) {
          builder.setPiece(piece);
        }
      }
      for (final Piece piece : this.board.getCurrentPlayer().getOpponent().getActivePieces()) {
        if (!piece.equals(this.getAttackedPiece())) {
          builder.setPiece(piece);
        }
      }
```

```
            builder.setPiece(this.movedPiece.movePiece(this));
            builder.setMoveMaker(this.board.getCurrentPlayer().getOpponent().getAlliance());

            return builder.build();
        }

        @Override
        public Board undo() {
            final Board.Builder builder = new Builder();
            for (final Piece piece : this.board.getAllPieces()) {
                builder.setPiece(piece);
            }
            builder.setEnPassantPawn((Pawn) this.getAttackedPiece());
            builder.setMoveMaker(this.board.getCurrentPlayer().getAlliance());
            return builder.build();
        }//new

    }// PawnAttackMove

    /**
     * When a pawn reach the final file.
     */
    public static class PawnPromotion extends Move {

        final Move decoratedMove;
        final Pawn promotedPawn;
        //  final Piece promotionPiece;

        public PawnPromotion(final Move decoratedMove) {
            super(decoratedMove.getBoard(),                                    decoratedMove.getMovedPiece(),
decoratedMove.getDestinationCoordinate());
            this.decoratedMove = decoratedMove;
            this.promotedPawn = (Pawn) decoratedMove.getMovedPiece();

        }

        @Override
        public int hashCode() {
            return this.decoratedMove.hashCode() + (31 * promotedPawn.hashCode());
        }

        @Override
        public boolean equals(final Object other) {
            return this == other || other instanceof PawnPromotion && (super.equals(other));
        }

        @Override
        public Board execute() {
            final Board pawnMoveBoard = this.decoratedMove.execute();
            final Board.Builder builder = new Builder();
            for (final Piece piece : pawnMoveBoard.getCurrentPlayer().getActivePieces()) {
                if (!this.promotedPawn.equals(piece)) {
                    builder.setPiece(piece);
                }
            }
            for (final Piece piece : pawnMoveBoard.getCurrentPlayer().getOpponent().getActivePieces()) {
                builder.setPiece(piece);
```

```
        }

        builder.setPiece(this.promotedPawn.getPromotionPiece().movePiece(this));
        builder.setMoveMaker(pawnMoveBoard.getCurrentPlayer().getAlliance());
        return builder.build();
    }

    @Override
    public boolean isAttack() {
        return this.decoratedMove.isAttack();
    }

    @Override
    public Piece getAttackedPiece() {
        return this.decoratedMove.getAttackedPiece();
    }

    @Override
    public String toString() {
        return BoardUtils.getPositionAtCoordinate(destinationCoordinate) + "=Q";
    }

}//PawnPromotion

/**
 * Pawn jump is when a pawn at First move of a pawn decide to move two tiles
 * instead of one.
 */
public static final class PawnJump extends Move {

    public PawnJump(final Board board, final Piece movedPiece, final int destinationCoordinate) {
        super(board, movedPiece, destinationCoordinate);
    }

    @Override
    public Board execute() {
        final Builder builder = new Builder();

        for (final Piece piece : this.board.getCurrentPlayer().getActivePieces()) {
            if (!this.movedPiece.equals(piece)) {
                builder.setPiece(piece);
            }
        }
        for (final Piece piece : this.board.getCurrentPlayer().getOpponent().getActivePieces()) {
            builder.setPiece(piece);
        }
        final Pawn movedPawn = (Pawn) this.movedPiece.movePiece(this);
        builder.setPiece(movedPawn);
        builder.setEnPassantPawn(movedPawn);
        builder.setMoveMaker(this.board.getCurrentPlayer().getOpponent().getAlliance());

        return builder.build();
    }

    @Override
    public String toString() {
        return BoardUtils.getPositionAtCoordinate(destinationCoordinate);
```

```
    }

  }//PawnJump

  /**
   * represent the basic castling on chess.
   */
  public static class CastleMove extends Move {

    protected final Rook castleRook;
    protected final int castleRookStart;
    protected final int castleRookDestination;

    public CastleMove(final Board board,
        final Piece movedPiece,
        final int destinationCoordinate,
        final Rook castleRook,
        final int castleRookStart,
        final int castleRookDestination) {
      super(board, movedPiece, destinationCoordinate);

      this.castleRook = castleRook;
      this.castleRookStart = castleRookStart;
      this.castleRookDestination = castleRookDestination;
    }

    public Rook getCastleRook() {
      return this.castleRook;
    }

    @Override
    public boolean isCastlingMove() {
      return true;
    }

    @Override
    public Board execute() {
      final Builder builder = new Builder();

      for (final Piece piece : this.board.getAllPieces()) {//getCurrentPlayer().getActivePieces()){
        if (!this.movedPiece.equals(piece) && !this.castleRook.equals(piece)) {
          builder.setPiece(piece);
        }
      }
      for (final Piece piece : this.board.getCurrentPlayer().getOpponent().getActivePieces()) {
        builder.setPiece(piece);
      }
      builder.setPiece(this.movedPiece.movePiece(this));
      builder.setPiece(new Rook(this.castleRook.getPieceAlliance(), this.castleRookDestination,
false));//, false));
      builder.setMoveMaker(this.board.getCurrentPlayer().getOpponent().getAlliance());

      return builder.build();
    }

    @Override
    public int hashCode() {
```

```
        final int prime = 31;
        int result = super.hashCode();
        result = prime * result + this.castleRook.hashCode();
        result = prime * result + this.castleRookDestination;

        return result;
      }

      @Override
      public boolean equals(final Object other) {
        if (this == other) {
          return true;
        }
        if (!(other instanceof CastleMove)) {
          return false;
        }

        final CastleMove otherCastleMove = (CastleMove) other;

        return super.equals(otherCastleMove) &&
this.castleRook.equals(otherCastleMove.getCastleRook());
      }

    }//CastleMove

    /**
     * represent the basic king side castling on chess.
     */
    public static final class KingSideCastle extends CastleMove {

      public KingSideCastle(final Board board,
          final Piece movedPiece,
          final int destinationCoordinate,
          final Rook castleRook,
          final int castleRookStart,
          final int castleRookDestination) {
        super(board, movedPiece, destinationCoordinate, castleRook, castleRookStart,
castleRookDestination);
      }

      @Override
      public boolean equals(final Object other) {
        if (this == other) {
          return true;
        }
        if (!(other instanceof KingSideCastle)) {
          return false;
        }
        final KingSideCastle otherKingSideCastle = (KingSideCastle) other;
        return super.equals(otherKingSideCastle) &&
this.castleRook.equals(otherKingSideCastle.getCastleRook());
      }

      @Override
      public String toString() {
        return "O-O";
      }
```

```
  }//KingSideCastle

  /**
   * represent the basic queen side castling on chess.
   */
  public static final class QueenSideCastle extends CastleMove {

    public QueenSideCastle(final Board board,
        final Piece movedPiece,
        final int destinationCoordinate,
        final Rook castleRook,
        final int castleRookStart,
        final int castleRookDestination) {
      super(board, movedPiece, destinationCoordinate, castleRook, castleRookStart,
castleRookDestination);
    }

    @Override
    public boolean equals(final Object other) {
      return this == other || other instanceof QueenSideCastle && super.equals(other);
    }

    @Override
    public String toString() {
      return "O-O-O";
    }
  }//QueenSideCastle

  /**
   * is just a null move in case of a mistake of a player , we don't want the
   * program stack
   */
  public static final class NullMove extends Move {

    public NullMove() {
      super(null, 65);
    }

    @Override
    public Board execute() {
      throw new RuntimeException("Your Move Succefully failed to move !  I cant execute a null
move dummy");
    }

    @Override
    public int getCurrentCoordinate() {
      return -1;
    } //double click in Pieces got Null problem exception but with this we solve it !
  }//NullMove

  public static class MoveFactory {

    private static final Move NULL_MOVE = new NullMove();//new

    private MoveFactory() {
      throw new RuntimeException("Instantiate failed Succefully ! Not be instantiable!");
    }
```

```
    public static Move getNullMove() {
      return NULL_MOVE;
    }//new

    public static Move createMove(final Board board, final int currentCoordinate, final int
destinationCoodinate) {
      for (final Move move : board.getAllLegalMove()) {
        if (move.getCurrentCoordinate() == currentCoordinate
            && move.getDestinationCoordinate() == destinationCoodinate) {

          return move;

        }
      }
      return NULL_MOVE;
    }

  }//MoveFactory

}//Move
```

Η αναπαράσταση των πιονιών

```
public abstract class Piece {

  protected final int piecePosition;
  protected final Alliance pieceAlliance;
  protected final boolean isFirstMove;
  protected final PieceType pieceType;
  private final int cachedHasCode;

  Piece(final PieceType pieceType,
      final int piecePosition,
      final Alliance pieceAlliance,
      final boolean isFirstMove) {

    this.pieceType = pieceType;
    this.piecePosition = piecePosition;
    this.pieceAlliance = pieceAlliance;

    this.isFirstMove = isFirstMove;
    this.cachedHasCode = computeHashCode();
  }

  /**
   * It compares two objects (Pieces)
   *
   * @param other
   * @return boolean
   */
  @Override
  public boolean equals(final Object other) {
    if (this == other) {
      return true;
    }
    if (!(other instanceof Piece)) {
```

```
            return false;
        }
        //from this point We know is a Piece cause of the if (Other instaceof Piece) pass the check
        final Piece otherPiece = (Piece) other;
        return piecePosition == otherPiece.getPiecePosition() && pieceType == otherPiece.getPieceType()
                && pieceAlliance == otherPiece.getPieceAlliance() && isFirstMove ==
otherPiece.isFirstMove();
    }

    /**
     * an integer value represent piece types
     *
     * @return int
     */
    private int computeHashCode() {
        int result = pieceType.hashCode();
        result = 31 * result + pieceAlliance.hashCode();
        result = 31 * result + piecePosition;
        result = 31 * result + (isFirstMove ? 1 : 0);

        return result;
    }

    @Override
    public int hashCode() {
        return this.cachedHasCode;
    }

    public int getPiecePosition() {
        return this.piecePosition;
    }

    public Alliance getPieceAlliance() {
        return this.pieceAlliance;
    }

    public boolean isFirstMove() {
        return this.isFirstMove;
    }

    public PieceType getPieceType() {
        return this.pieceType;
    }

    public int getPieceValue() {
        return this.pieceType.getPieceValue();
    }

    /**
     * need work here.
     *
     * @return int
     */
    public abstract int locationBonus();

    /**
     * Calculates the Legal moves.
```

```
 *
 * @param board
 * @return Collection<Move>
 */
public abstract Collection<Move> calculateLegalMoves(final Board board);

/**
 * Is makes the next move happened.
 *
 * @param move
 * @return
 */
public abstract Piece movePiece(Move move);

/**
 * this enum represent the piece type and the values in the game.
 */
public enum PieceType {
   PAWN("P", 100) {//100
      @Override
      public boolean isKing() {
         return false;
      }

      @Override
      public boolean isRook() {
         return false;
      }

      @Override
      public boolean isPawn() {
         return true;
      }

      @Override
      public boolean isBishop() {
         return false;
      }

      @Override
      public boolean isKnight() {
         return false;
      }

      @Override
      public boolean isQuuen() {
         return false;
      }
   },
   KNIGHT("N", 320) {//320
      @Override
      public boolean isKing() {
         return false;
      }

      @Override
      public boolean isRook() {
```

```
        return false;
    }

    @Override
    public boolean isPawn() {
        return false;
    }

    @Override
    public boolean isBishop() {
        return false;
    }

    @Override
    public boolean isKnight() {
        return true;
    }

    @Override
    public boolean isQuuen() {
        return false;
    }
},
BISHOP("B", 350) {//350
    @Override
    public boolean isKing() {
        return false;
    }

    @Override
    public boolean isRook() {
        return false;
    }

    @Override
    public boolean isPawn() {
        return false;
    }

    @Override
    public boolean isBishop() {
        return true;
    }

    @Override
    public boolean isKnight() {
        return false;
    }

    @Override
    public boolean isQuuen() {
        return false;
    }
},
ROOK("R", 500) {//500
    @Override
    public boolean isKing() {
```

```
        return false;
    }

    @Override
    public boolean isRook() {
        return true;
    } // yes you are the Might ROOK

    @Override
    public boolean isPawn() {
        return false;
    }

    @Override
    public boolean isBishop() {
        return false;
    }

    @Override
    public boolean isKnight() {
        return false;
    }

    @Override
    public boolean isQuuen() {
        return false;
    }
},
QUEEN("Q", 900) {//900
    @Override
    public boolean isKing() {
        return false;
    }

    @Override
    public boolean isRook() {
        return false;
    }

    @Override
    public boolean isPawn() {
        return false;
    }

    @Override
    public boolean isBishop() {
        return false;
    }

    @Override
    public boolean isKnight() {
        return false;
    }

    @Override
    public boolean isQuuen() {
        return true;
```

```
    }
},
KING("K", 20000) {
    @Override
    public boolean isKing() {
        return true;
    } // Yes you are the center of the game

    @Override
    public boolean isRook() {
        return false;
    }

    @Override
    public boolean isPawn() {
        return false;
    }

    @Override
    public boolean isBishop() {
        return false;
    }

    @Override
    public boolean isKnight() {
        return false;
    }

    @Override
    public boolean isQuuen() {
        return false;
    }
};

private String pieceName;
private int pieceValue;

PieceType(final String pieceName, final int pieceValue) {
    this.pieceName = pieceName;
    this.pieceValue = pieceValue;
}

@Override
public String toString() {
    return this.pieceName;
}

public int getPieceValue() {
    return this.pieceValue;
}

public abstract boolean isPawn();

public abstract boolean isBishop();

public abstract boolean isKing();
```

```
    public abstract boolean isRook();

    public abstract boolean isQuuen();

    public abstract boolean isKnight();
  }//PieceType

}//Piece

public class Pawn extends Piece {

  private final static int[] CANDIDATE_MOVE_COORDINATES = {8, 16, 7, 9};

  public Pawn(final Alliance pieceAlliance, final int piecePosition) {
    super(PieceType.PAWN, piecePosition, pieceAlliance, true);
  }

  public Pawn(final Alliance pieceAlliance, final int piecePosition, final boolean isFirstMove) {
    super(PieceType.PAWN, piecePosition, pieceAlliance, isFirstMove);
  }

  /**
   * we need to calculate the legal moves and the attacks which bishop have in
   * the instance of the game.
   *
   * @param board
   * @return Collection<Move>
   */
  @Override
  public Collection<Move> calculateLegalMoves(final Board board) {

    final List<Move> legalMoves = new ArrayList<>();
    for (final int currentCandidateOffset : CANDIDATE_MOVE_COORDINATES) {

      final int candidateDestinationCoordinate = this.piecePosition + (this.pieceAlliance.getDirection()
* currentCandidateOffset);
      if (!BoardUtils.isValidTileCoordinate(candidateDestinationCoordinate)) {
        continue;
      }
      if (currentCandidateOffset == 8
&& !board.getTile(candidateDestinationCoordinate).isTileOccupied()) {
        // promo!
        if (this.pieceAlliance.isPawnPromotionSquare(candidateDestinationCoordinate)) {
//          legalMoves.add(new PawnPromotion(
//              new PawnMove(board, this, candidateDestinationCoordinate),
PieceUtils.INSTANCE.getMovedQueen(this.pieceAlliance, candidateDestinationCoordinate)));
          legalMoves.add(new PawnPromotion(new PawnMove(board, this,
candidateDestinationCoordinate)));
        } else {
          legalMoves.add(new MajorMove(board, this, candidateDestinationCoordinate));
        }

      } else if (currentCandidateOffset == 16 && this.isFirstMove()
          && ((BoardUtils.INSTANCE.SECOND_ROW.get(this.piecePosition)
this.getPieceAlliance().isBlack())
          || (BoardUtils.INSTANCE.SEVENTH_ROW.get(this.piecePosition)
this.getPieceAlliance().isWhite()))) {
```

```
            final int behindCandidateDestinationCoordinate = this.piecePosition +
(this.pieceAlliance.getDirection() * 8);

            if (!board.getTile(behindCandidateDestinationCoordinate).isTileOccupied()
                && !board.getTile(candidateDestinationCoordinate).isTileOccupied()) {

              legalMoves.add(new PawnJump(board, this, candidateDestinationCoordinate));
            }

          } else if (currentCandidateOffset == 7
              &&          !((BoardUtils.INSTANCE.EIGHTH_COLUMN.get(this.piecePosition)          &&
this.pieceAlliance.isWhite()
              ||          (BoardUtils.INSTANCE.FIRST_COLUMN.get(this.piecePosition)          &&
this.pieceAlliance.isBlack()))))) {
            if (board.getTile(candidateDestinationCoordinate).isTileOccupied()) {
              final Piece pieceOnCandidate = board.getTile(candidateDestinationCoordinate).getPiece();
              if (this.pieceAlliance != pieceOnCandidate.getPieceAlliance()) {
                if (this.pieceAlliance.isPawnPromotionSquare(candidateDestinationCoordinate)) {
                  legalMoves.add(new      PawnPromotion(new      PawnAttackMove(board,      this,
candidateDestinationCoordinate, pieceOnCandidate)));
//                  legalMoves.add(new PawnPromotion(
//                      new PawnAttackMove(board, this, candidateDestinationCoordinate),
//                                                  board.getPiece(candidateDestinationCoordinate)),
PieceUtils.INSTANCE.getMovedQueen(this.pieceAlliance,
candidateDestinationCoordinate)));//legalMoves.add(new                  PawnPromotion(new
PawnAttackMove(board, this, candidateDestinationCoordinate, pieceOnCandidate)));
                } else {
                  legalMoves.add(new  PawnAttackMove(board,  this,  candidateDestinationCoordinate,
pieceOnCandidate));
                }
              }
            } else if (board.getEnPassantPawn() != null) {
              if      (board.getEnPassantPawn().getPiecePosition()      ==      (this.piecePosition      +
(this.pieceAlliance.getOppositeDirection()))) {
                final Piece pieceOnCandidate = board.getEnPassantPawn();
                if (this.pieceAlliance != pieceOnCandidate.getPieceAlliance()) {
                  legalMoves.add(new          Move.PawnEnPassantAttackMove(board,          this,
candidateDestinationCoordinate, pieceOnCandidate));
                }
              }
            }

          } else if (currentCandidateOffset == 9
              &&          !((BoardUtils.INSTANCE.FIRST_COLUMN.get(this.piecePosition)          &&
this.pieceAlliance.isWhite()
              ||          (BoardUtils.INSTANCE.EIGHTH_COLUMN.get(this.piecePosition)          &&
this.pieceAlliance.isBlack()))))) {
            if (board.getTile(candidateDestinationCoordinate).isTileOccupied()) {
              final Piece pieceOnCandidate = board.getTile(candidateDestinationCoordinate).getPiece();
              if (this.pieceAlliance != pieceOnCandidate.getPieceAlliance()) {
                if (this.pieceAlliance.isPawnPromotionSquare(candidateDestinationCoordinate)) {
                  legalMoves.add(new      PawnPromotion(new      PawnAttackMove(board,      this,
candidateDestinationCoordinate, pieceOnCandidate)));
//                  legalMoves.add(new PawnPromotion(
//                      new PawnAttackMove(board, this, candidateDestinationCoordinate,
```

```
//                                                  board.getPiece(candidateDestinationCoordinate)),
PieceUtils.INSTANCE.getMovedQueen(this.pieceAlliance,          candidateDestinationCoordinate)));//
legalMoves.add(new          PawnPromotion(new          PawnAttackMove(board,          this,
candidateDestinationCoordinate, pieceOnCandidate)));
                } else {
                  legalMoves.add(new  PawnAttackMove(board,  this,  candidateDestinationCoordinate,
pieceOnCandidate));
                }
              }
            } else if (board.getEnPassantPawn() != null) {
              if       (board.getEnPassantPawn().getPiecePosition()       ==       (this.piecePosition       -
(this.pieceAlliance.getOppositeDirection()))) {
                final Piece pieceOnCandidate = board.getEnPassantPawn();
                if (this.pieceAlliance != pieceOnCandidate.getPieceAlliance()) {
                  legalMoves.add(new          Move.PawnEnPassantAttackMove(board,          this,
candidateDestinationCoordinate, pieceOnCandidate));
                }
              }
            }

          }
      }

      return ImmutableList.copyOf(legalMoves);
    }

    @Override
    public String toString() {
      return Piece.PieceType.PAWN.toString();
    }

    /**
     * it initiated a new bishop to the next candidate Destination.
     *
     * @param move
     * @return Pawn
     */
    //Piece
    @Override
    public Pawn movePiece(Move move) {
      return new Pawn(move.getMovedPiece().getPieceAlliance(), move.getDestinationCoordinate());
    }

    /**
     * when a pawn reach the final rank at the game it can be promote.
     *
     * @return
     */
    public Piece getPromotionPiece() {
      return new Queen(this.pieceAlliance, this.piecePosition, false);
    }

    @Override
    public int locationBonus() {
      return this.pieceAlliance.pawnBonus(this.piecePosition);
       }
    }//Pawn
```

```
public class Bishop extends Piece {

  private final static int[] CANDIDATE_VECTOR_MOVE_COORDINATES = {-9, -7, 7, 9};

  public Bishop(final Alliance pieceAlliance, final int piecePosition) {
    super(PieceType.BISHOP, piecePosition, pieceAlliance, true);
  }

  public Bishop(final Alliance pieceAlliance, final int piecePosition, final boolean isFirstMove) {
    super(PieceType.BISHOP, piecePosition, pieceAlliance, isFirstMove);
  }

  /**
   * we need to calculate the legal moves and the attacks which bishop have in
   * the instance of the game.
   *
   * @param board
   * @return Collection<Move>
   */
  @Override
  public Collection<Move> calculateLegalMoves(Board board) {
    final List<Move> legalMoves = new ArrayList<>();

    for (final int candidateCoordinateOffset : CANDIDATE_VECTOR_MOVE_COORDINATES) {
      int candidateDestinationCoordinate = this.piecePosition;

      while (BoardUtils.isValidTileCoordinate(candidateDestinationCoordinate)) {
        if (isFirstColumnExclusion(candidateDestinationCoordinate, candidateCoordinateOffset)
            || isEighthColumnExlcusion(candidateDestinationCoordinate,
candidateCoordinateOffset)) {
          break;
        }
        candidateDestinationCoordinate += candidateCoordinateOffset;

        if (BoardUtils.isValidTileCoordinate(candidateDestinationCoordinate)) {

          final Tile candidateDestinationTile = board.getTile(candidateDestinationCoordinate);

          if (!candidateDestinationTile.isTileOccupied()) {
            legalMoves.add(new MajorMove(board, this, candidateDestinationCoordinate));
          } else {
            final Piece pieceAtDestination = candidateDestinationTile.getPiece();
            final Alliance pieceAlliance = pieceAtDestination.getPieceAlliance();

            if (this.pieceAlliance != pieceAlliance) {
              legalMoves.add(new  MajorAttackMove(board,  this,  candidateDestinationCoordinate,
pieceAtDestination));
            }
            break;
          }

        }
      }
    }
    return ImmutableList.copyOf(legalMoves);
  }
```

```
    @Override
    public String toString() {
        return Piece.PieceType.BISHOP.toString();
    }

    /**
     * we set some exclusion to set the borders of the board.
     *
     * @param currentPosition
     * @param candidateOffset
     * @return
     */
    private static boolean isFirstColumnExclusion(final int currentPosition, final int candidateOffset) {
        return BoardUtils.INSTANCE.FIRST_COLUMN.get(currentPosition) && (candidateOffset == -9 ||
candidateOffset == 7);
    }

    /**
     * we set some exclusion to set the borders of the board.
     *
     * @param currentPosition
     * @param candidateOffset
     * @return
     */
    private static boolean isEighthColumnExlcusion(final int currentPosition, final int candidateOffset) {
        return BoardUtils.INSTANCE.EIGHTH_COLUMN.get(currentPosition) && (candidateOffset == -7 ||
candidateOffset == 9);
    }

    /**
     * it initiated a new bishop to the next candidate Destination.
     *
     * @param move
     * @return Bishop
     */
    //Piece
    @Override
    public Bishop movePiece(Move move) {
        return new Bishop(move.getMovedPiece().getPieceAlliance(), move.getDestinationCoordinate());

        //return            PieceUtils.INSTANCE.getMovedBishop(move.getMovedPiece().getPieceAlliance(),
move.getDestinationCoordinate());
    }

    @Override
    public int locationBonus() {
        return this.pieceAlliance.bishopBonus(this.piecePosition);
    }//new

}//Bishop

public class King extends Piece {

    private final static int[] CANDIDATE_MOVE_COORDINATES = {-9, -8, -7, -1, 1, 7, 8, 9};
    private final boolean kingSideCastleCapable, queenSideCastleCapable;
    private final boolean isCastled;
```

```
  public King(final Alliance pieceAlliance, final int piecePosition, final boolean kingSideCastleCapable,
      final boolean queenSideCastleCapable) {
    super(PieceType.KING, piecePosition, pieceAlliance, true);
    this.isCastled = false;
    this.kingSideCastleCapable = kingSideCastleCapable;
    this.queenSideCastleCapable = queenSideCastleCapable;
  }

  public King(final Alliance pieceAlliance, final int piecePosition, final boolean isFirstMove, final
boolean isCastled,
       final boolean kingSideCastleCapable, final boolean queenSideCastleCapable) {
    super(PieceType.KING, piecePosition, pieceAlliance, isFirstMove);

    this.isCastled = isCastled;
    this.kingSideCastleCapable = kingSideCastleCapable;
    this.queenSideCastleCapable = queenSideCastleCapable;
  }

  public boolean isCastled() {
    return this.isCastled;
  }

  public boolean isKingSideCastleCapable() {
    return this.kingSideCastleCapable;
  }

  public boolean isQueenSideCastleCapable() {
    return this.queenSideCastleCapable;
  }

  /**
   * we need to calculate the legal moves and the attacks which bishop have in
   * the instance of the game.
   *
   * @param board
   * @return Collection<Move>
   */
  @Override
  public Collection<Move> calculateLegalMoves(Board board) {

    final List<Move> legalMoves = new ArrayList<>();

    for (final int currentCandidateOffset : CANDIDATE_MOVE_COORDINATES) {

      final int candidateDestinationCoordinate = this.piecePosition + currentCandidateOffset;

      if (isFirstColumnExlcusion(this.piecePosition, currentCandidateOffset)
          || isEighthColumnExlcusion(this.piecePosition, currentCandidateOffset)) {
        continue;
      }

      if (BoardUtils.isValidTileCoordinate(candidateDestinationCoordinate)) {

        final Tile candidateDestinationTile = board.getTile(candidateDestinationCoordinate);

        if (!candidateDestinationTile.isTileOccupied()) {
```

```
                legalMoves.add(new MajorMove(board, this, candidateDestinationCoordinate));
            } else {
                final Piece pieceAtDestination = candidateDestinationTile.getPiece();
                final Alliance pieceAlliance = pieceAtDestination.getPieceAlliance();

                if (this.pieceAlliance != pieceAlliance) {
                    legalMoves.add(new MajorAttackMove(board, this, candidateDestinationCoordinate,
pieceAtDestination));
                }
            }
        }

    }

    return ImmutableList.copyOf(legalMoves);
  }

  @Override
  public String toString() {
    return Piece.PieceType.KING.toString();
  }

  /**
   * we set some exclusion to set the borders of the board.
   *
   * @param currentPosition
   * @param candidateOffset
   * @return
   */
  private static boolean isFirstColumnExlcusion(final int currentPosition, final int candidateOffset) {
    return BoardUtils.INSTANCE.FIRST_COLUMN.get(currentPosition) && (candidateOffset == -9 ||
candidateOffset == -1
        || candidateOffset == 7);
  }

  /**
   * we set some exclusion to set the borders of the board.
   *
   * @param currentPosition
   * @param candidateOffset
   * @return
   */
  private static boolean isEighthColumnExlcusion(final int currentPosition, final int candidateOffset) {
    return BoardUtils.INSTANCE.EIGHTH_COLUMN.get(currentPosition) && (candidateOffset == -7 ||
candidateOffset == 1
        || candidateOffset == 9);
  }

  /**
   * it initiated a new bishop to the next candidate Destination.
   *
   * @param move
   * @return King
   */
  //Piece
  @Override
  public King movePiece(Move move) {
```

```
        return new King(move.getMovedPiece().getPieceAlliance(), move.getDestinationCoordinate(),
                false, move.isCastlingMove(), false, false);
    }

    @Override
    public int locationBonus() {
        return this.pieceAlliance.kingBonus(this.piecePosition);
    }//new

}//King

public class Knight extends Piece {

    private final static int[] CANDIDATE_MOVE_COORDINATES = {-17, -15, -10, -6, 6, 10, 15, 17};

    public Knight(final Alliance pieceAlliance, final int piecePosition) {
        super(PieceType.KNIGHT, piecePosition, pieceAlliance, true);
    }

    public Knight(final Alliance pieceAlliance, final int piecePosition, final boolean isFirstMove) {
        super(PieceType.KNIGHT, piecePosition, pieceAlliance, isFirstMove);
    }

    /**
     * we need to calculate the legal moves and the attacks which bishop have in
     * the instance of the game.
     *
     * @param board
     * @return Collection<Move>
     */
    @Override
    public Collection<Move> calculateLegalMoves(final Board board) {

        final List<Move> legalMoves = new ArrayList<>();

        for (final int currentCandidateOffset : CANDIDATE_MOVE_COORDINATES) {

            final int candidateDestinationCoordinate = this.piecePosition + currentCandidateOffset;

            if (BoardUtils.isValidTileCoordinate(candidateDestinationCoordinate)) {

                if (isFirstColumnExlcusion(this.piecePosition, currentCandidateOffset)
                        || isSecondColumnExlcusion(this.piecePosition, currentCandidateOffset)
                        || isSevethColumnExlcusion(this.piecePosition, currentCandidateOffset)
                        || isEighthColumnExlcusion(this.piecePosition, currentCandidateOffset)) {
                    continue;
                }

                final Tile candidateDestinationTile = board.getTile(candidateDestinationCoordinate);

                if (!candidateDestinationTile.isTileOccupied()) {
                    legalMoves.add(new MajorMove(board, this, candidateDestinationCoordinate));
                } else {
                    final Piece pieceAtDestination = candidateDestinationTile.getPiece();
                    final Alliance pieceAlliance = pieceAtDestination.getPieceAlliance();

                    if (this.pieceAlliance != pieceAlliance) {
```

```
                legalMoves.add(new MajorAttackMove(board, this, candidateDestinationCoordinate,
pieceAtDestination));
              }
            }
          }
        }

      return ImmutableList.copyOf(legalMoves);
    }

    @Override
    public String toString() {
      return Piece.PieceType.KNIGHT.toString();
    }

    /**
     * we set some exclusion to set the borders of the board.
     *
     * @param currentPosition
     * @param candidateOffset
     * @return
     */
    private static boolean isFirstColumnExlcusion(final int currentPosition, final int candidateOffset) {
      return BoardUtils.INSTANCE.FIRST_COLUMN.get(currentPosition) && (candidateOffset == -17 ||
candidateOffset == -10
          || candidateOffset == 6 || candidateOffset == 15);
    }

    /**
     * we set some exclusion to set the borders of the board.
     *
     * @param currentPosition
     * @param candidateOffset
     * @return
     */
    private static boolean isSecondColumnExlcusion(final int currentPosition, final int candidateOffset) {
      return BoardUtils.INSTANCE.SECOND_COLUMN.get(currentPosition) && (candidateOffset == -10
|| candidateOffset == 6);
    }

    /**
     * we set some exclusion to set the borders of the board.
     *
     * @param currentPosition
     * @param candidateOffset
     * @return
     */
    private static boolean isSevethColumnExlcusion(final int currentPosition, final int candidateOffset) {
      return BoardUtils.INSTANCE.SEVENTH_COLUMN.get(currentPosition) && (candidateOffset == -6
|| candidateOffset == 10);
    }

    /**
     * we set some exclusion to set the borders of the board.
     *
     * @param currentPosition
     * @param candidateOffset
```

```
     * @return
     */
    private static boolean isEighthColumnExlcusion(final int currentPosition, final int candidateOffset) {
        return  BoardUtils.INSTANCE.EIGHTH_COLUMN.get(currentPosition)  &&  (candidateOffset  ==  -15
|| candidateOffset == -6
                || candidateOffset == 10 || candidateOffset == 17);
    }

    /**
     * it initiated a new bishop to the next candidate Destination.
     *
     * @param move
     * @return Knight
     */
    //Piece
    @Override
    public Knight movePiece(Move move) {
        return new Knight(move.getMovedPiece().getPieceAlliance(), move.getDestinationCoordinate());
    }

    @Override
    public int locationBonus() {
        return this.pieceAlliance.knightBonus(this.piecePosition);
    }//new

}//Knight

public class Queen extends Piece {

    private final static int[] CANDIDATE_VECTOR_MOVE_COORDINATES = {-9, -8, -7, -1, 1, 7, 8, 9};

    public Queen(final Alliance pieceAlliance, final int piecePosition) {
        super(PieceType.QUEEN, piecePosition, pieceAlliance, true);
    }

    public Queen(final Alliance pieceAlliance, final int piecePosition, final boolean isFirstMove) {
        super(PieceType.QUEEN, piecePosition, pieceAlliance, isFirstMove);
    }

    /**
     * we need to calculate the legal moves and the attacks which bishop have in
     * the instance of the game.
     *
     * @param board
     * @return Collection<Move>
     */
    @Override
    public Collection<Move> calculateLegalMoves(Board board) {
        final List<Move> legalMoves = new ArrayList<>();

        for (final int candidateCoordinateOffset : CANDIDATE_VECTOR_MOVE_COORDINATES) {
            int candidateDestinationCoordinate = this.piecePosition;

            while (BoardUtils.isValidTileCoordinate(candidateDestinationCoordinate)) {
                if (isFirstColumnExclusion(candidateDestinationCoordinate, candidateCoordinateOffset)
                        ||                                  isEighthColumnExlcusion(candidateDestinationCoordinate,
candidateCoordinateOffset)) {
```

```
            break;
          }
          candidateDestinationCoordinate += candidateCoordinateOffset;

          if (BoardUtils.isValidTileCoordinate(candidateDestinationCoordinate)) {

            final Tile candidateDestinationTile = board.getTile(candidateDestinationCoordinate);

            if (!candidateDestinationTile.isTileOccupied()) {
              legalMoves.add(new Move.MajorMove(board, this, candidateDestinationCoordinate));
            } else {
              final Piece pieceAtDestination = candidateDestinationTile.getPiece();
              final Alliance pieceAlliance = pieceAtDestination.getPieceAlliance();

              if (this.pieceAlliance != pieceAlliance) {
                legalMoves.add(new MajorAttackMove(board, this, candidateDestinationCoordinate,
pieceAtDestination));
              }
              break;
            }

          }
        }
      }
      return ImmutableList.copyOf(legalMoves);
    }

    @Override
    public String toString() {
      return Piece.PieceType.QUEEN.toString();
    }

    /**
     * we set some exclusion to set the borders of the board.
     *
     * @param currentPosition
     * @param candidateOffset
     * @return
     */
    private static boolean isFirstColumnExclusion(final int currentPosition, final int candidateOffset) {
      return BoardUtils.INSTANCE.FIRST_COLUMN.get(currentPosition) && (candidateOffset == -1
          || candidateOffset == -9 || candidateOffset == 7);
    }

    /**
     * we set some exclusion to set the borders of the board.
     *
     * @param currentPosition
     * @param candidateOffset
     * @return
     */
    private static boolean isEighthColumnExlcusion(final int currentPosition, final int candidateOffset) {
      return BoardUtils.INSTANCE.EIGHTH_COLUMN.get(currentPosition) && (candidateOffset == -7 ||
candidateOffset == 1
          || candidateOffset == 9);
    }
```

```
  /**
   * it initiated a new bishop to the next candidate Destination.
   *
   * @param move
   * @return Queen
   */
  //Piece
  @Override
  public Queen movePiece(Move move) {
    return new Queen(move.getMovedPiece().getPieceAlliance(), move.getDestinationCoordinate());
  }

  @Override
  public int locationBonus() {
    return this.pieceAlliance.queenBonus(this.piecePosition);
  }//new

}//Queen

public class Rook extends Piece {

  private final static int[] CANDIDATE_VECTOR_MOVE_COORDINATES = {-8, -1, 1, 8};

  public Rook(final Alliance pieceAlliance, final int piecePosition) {
    super(PieceType.ROOK, piecePosition, pieceAlliance, true);
  }

  public Rook(final Alliance pieceAlliance, final int piecePosition, final boolean isFirstMove) {
    super(PieceType.ROOK, piecePosition, pieceAlliance, isFirstMove);
  }

  /**
   * we need to calculate the legal moves and the attacks which bishop have in
   * the instance of the game.
   *
   * @param board
   * @return Collection<Move>
   */
  @Override
  public Collection<Move> calculateLegalMoves(Board board) {
    final List<Move> legalMoves = new ArrayList<>();
    int candidateDestinationCoordinate;
    Tile candidateDestinationTile;
    Piece pieceAtDestination;
    Alliance pieceAlliance;

    for (final int candidateCoordinateOffset : CANDIDATE_VECTOR_MOVE_COORDINATES) {
      candidateDestinationCoordinate = this.piecePosition;

      while (true) {
        if (isFirstColumnExclusion(candidateDestinationCoordinate, candidateCoordinateOffset)
            ||                                    isEighthColumnExlcusion(candidateDestinationCoordinate,
candidateCoordinateOffset)) {
          break;
        }

        candidateDestinationCoordinate += candidateCoordinateOffset;
```

```
            if (BoardUtils.isValidTileCoordinate(candidateDestinationCoordinate) == false) {
                break;
            }

            candidateDestinationTile = board.getTile(candidateDestinationCoordinate);

            if (!candidateDestinationTile.isTileOccupied()) {
                legalMoves.add(new Move.MajorMove(board, this, candidateDestinationCoordinate));
            } else {
                pieceAtDestination = candidateDestinationTile.getPiece();
                pieceAlliance = pieceAtDestination.getPieceAlliance();

                if (this.pieceAlliance != pieceAlliance) {
                    legalMoves.add(new MajorAttackMove(board, this, candidateDestinationCoordinate,
pieceAtDestination));
                }
                break;
            }

        }
    }
    return ImmutableList.copyOf(legalMoves);
  }

  @Override
  public String toString() {
    return Piece.PieceType.ROOK.toString();
  }

  /**
   * we set some exclusion to set the borders of the board.
   *
   * @param currentPosition
   * @param candidateOffset
   * @return
   */
  private static boolean isFirstColumnExclusion(final int currentPosition, final int candidateOffset) {
    return BoardUtils.INSTANCE.FIRST_COLUMN.get(currentPosition) && (candidateOffset == -1);
  }

  /**
   * we set some exclusion to set the borders of the board.
   *
   * @param currentPosition
   * @param candidateOffset
   * @return
   */
  private static boolean isEighthColumnExlcusion(final int currentPosition, final int candidateOffset) {
    return BoardUtils.INSTANCE.EIGHTH_COLUMN.get(currentPosition) && (candidateOffset == 1);
  }

  /**
   * it initiated a new bishop to the next candidate Destination.
   *
   * @param move
   * @return Rook
```

```
     */

    @Override
    public Rook movePiece(Move move) {
        return new Rook(move.getMovedPiece().getPieceAlliance(), move.getDestinationCoordinate());
    }

    @Override
    public int locationBonus() {
        return this.pieceAlliance.rookBonus(this.piecePosition);
    }//new

}//Rook
```

Κλάση για τους παίχτες της σκακίερας

```
public abstract class Player {

    protected final Board board;
    protected final King playerKing;
    protected final Collection<Move> legalMoves;
    private final boolean isInCheck;

    public Player(final Board board,
            final Collection<Move> legalMoves,
            final Collection<Move> opponentMoves) {
        this.board = board;
        this.playerKing = establishKing();
        /*
            gia na ginei auto prepei na kserw kai ti kiniseis exei o adipalos
            wste na kserw an kapoio tetragono dexete epithesi!
         */

        //Combines multiple iterables into a single iterable ,we must know if castle is possible.
        this.legalMoves = ImmutableList.copyOf(Iterables.concat(legalMoves,
                calculateKingCastles(legalMoves, opponentMoves)));
        this.isInCheck = !Player.calculateAttacksOnTile(this.playerKing.getPiecePosition(),
opponentMoves).isEmpty();
    }

    public King getPlayerKing() {
        return this.playerKing;
    }

    public Collection<Move> getLegalMoves() {
        return this.legalMoves;
    }

    /**
     * calculates the threats
     *
     * @param piecePosition
     * @param moves
     * @return attackMoves
     */
    protected static Collection<Move> calculateAttacksOnTile(int piecePosition, Collection<Move>
moves) {
```

```
        final List<Move> attackMoves = new ArrayList<>();
        for (final Move move : moves) {
            if (piecePosition == move.getDestinationCoordinate()) {
                attackMoves.add(move);
            }
        }

        return ImmutableList.copyOf(attackMoves);
    }

    /**
     * We need to know where is our king
     *
     * @return piece (king)
     */
    private King establishKing() {

        for (final Piece piece : getActivePieces()) {
            if (piece.getPieceType().isKing()) {
                return (King) piece;
            }
        }
        throw new RuntimeException("Should not reach here ! not a valid board" +
this.getActivePieces());
    }

    /**
     * if we a move is legal.
     *
     * @param move
     * @return boolean
     */
    public boolean isMoveLegal(final Move move) {
        return this.legalMoves.contains(move);
    }

    /**
     * If our king is under attack
     *
     * @return boolean
     */
    public boolean isInCheck() {
        return this.isInCheck;
    }

    /**
     * If our king is under a attack and has not escape moves.
     *
     * @return boolean
     */
    public boolean isInCheckMate() {
        return this.isInCheck && !hasEscapesMoves();
    }

    /**
     * when our king is not under attack and at the same instance king has not
```

```
     * escape moves that contains other pieces can block the attack or capture
     * the enemy attacker.
     *
     * @return boolean
     */
    public boolean isInStalemate() {
        return !this.isInCheck && !hasEscapesMoves();
    }

    /**
     * is used to cover the rules for auto pat .
     *
     * @return boolean
     */
    public boolean isInRuleStalemate() {
        return !this.isInCheck && hasEscapesMoves();
    }

    public boolean isKingSideCastleCapable() {
        return this.playerKing.isKingSideCastleCapable();
    }

    public boolean isQueenCastleCapable() {
        return this.playerKing.isQueenSideCastleCapable();
    }

    /**
     * looking for a move that can be done under attack this means blocking
     * moves or capture the attacker or king escape.
     *
     * @return boolean
     */
    protected boolean hasEscapesMoves() {
        for (final Move move : this.legalMoves) {
            final MoveTransition transition = makeMove(move);
            if (transition.getMoveStatus().isDone()) {
                return true;
            }
        }
        return false;
    }

    public boolean isCastled() {
        return this.playerKing.isCastled();
    }

    /**
     * We need to cover the states of players like the moves that can be done on
     * the game or to protect him for mistakes.
     *
     * @param move
     * @return MoveTransition
     */
    public MoveTransition makeMove(final Move move) {
        if (!isMoveLegal(move)) {
            return new MoveTransition(this.board, this.board, move, MoveStatus.ILLEGAL_MOVE);
        }
```

```
      final Board transitionBoard = move.execute();

      final Collection<Move> kingAttacks =
Player.calculateAttacksOnTile(transitionBoard.getCurrentPlayer().getOpponent().getPlayerKing().getPi
ecePosition(),
          transitionBoard.getCurrentPlayer().getLegalMoves());

      if (!kingAttacks.isEmpty()) {
        return new MoveTransition(this.board, this.board, move,
MoveStatus.LEAVES_PLAYER_IN_CHECK);
      }
      return new MoveTransition(this.board, transitionBoard, move, MoveStatus.DONE);
    }

    /**
     * We undo a move of a player
     *
     * @param move
     * @return MoveTransition
     */
    public MoveTransition unMakeMove(final Move move) {
      return new MoveTransition(this.board, move.undo(), move, MoveStatus.DONE);
    }//new

    /**
     * read the board for the active pieces
     *
     * @return activePieces
     */
    public abstract Collection<Piece> getActivePieces();

    /**
     * with which alliance we are (White or Black)
     *
     * @return Alliance
     */
    public abstract Alliance getAlliance();

    /**
     * we need to store information for each player
     *
     * @return Player
     */
    public abstract Player getOpponent();

    /**
     * Calculate the king Castle sides
     *
     * @param playerLegals
     * @param opponentsLegals
     * @return Moves
     */
    protected abstract Collection<Move> calculateKingCastles(Collection<Move> playerLegals,
Collection<Move> opponentsLegals);

}//Player
```

```
public class WhitePlayer extends Player {

   public WhitePlayer(final Board board,
         final Collection<Move> whiteStandarLegalMoves,
         final Collection<Move> blackStandarLegalMoves) {

      super(board, whiteStandarLegalMoves, blackStandarLegalMoves);

   }

   /**
    * get all whites active pieces
    *
    * @return activePieces
    */
   @Override
   public Collection<Piece> getActivePieces() {
      return this.board.getWhitePieces();
   }

   /**
    * Set the alliance
    *
    * @return Alliance
    */
   @Override
   public Alliance getAlliance() {
      return Alliance.WHITE;
   }

   /**
    * take the info from opponent
    *
    * @return Player
    */
   @Override
   public Player getOpponent() {
      return this.board.blackPlayer();
   }

   /**
    * calculate king castles if possible , king side and queen side.
    *
    * @param playerLegals
    * @param opponentsLegals
    * @return
    */
   @Override
   protected Collection<Move> calculateKingCastles(final Collection<Move> playerLegals,
         final Collection<Move> opponentsLegals) {

      final List<Move> kingCastles = new ArrayList<>();

      if (this.playerKing.isFirstMove() && !isInCheck()) {
         //white King casle
         if (!this.board.getTile(61).isTileOccupied() && !this.board.getTile(62).isTileOccupied()) {
```

```
            final Tile rookTile = this.board.getTile(63);

            if (rookTile.isTileOccupied() && rookTile.getPiece().isFirstMove()) {
               if (Player.calculateAttacksOnTile(61, opponentsLegals).isEmpty()
                     && Player.calculateAttacksOnTile(62, opponentsLegals).isEmpty()
                     && rookTile.getPiece().getPieceType().isRook()) {
                  kingCastles.add(new KingSideCastle(this.board, this.playerKing, 62,
                        (Rook) rookTile.getPiece(), rookTile.getTileCoordinate(), 61));
               }

            }
         }
         //White King casle big
         if (!this.board.getTile(59).isTileOccupied() && !this.board.getTile(58).isTileOccupied()
               && !this.board.getTile(57).isTileOccupied()) {

            final Tile rookTile = this.board.getTile(56);

            if (rookTile.isTileOccupied() && rookTile.getPiece().isFirstMove()
                  && Player.calculateAttacksOnTile(58, opponentsLegals).isEmpty()
                  && Player.calculateAttacksOnTile(59, opponentsLegals).isEmpty()
                  && rookTile.getPiece().getPieceType().isRook()) {

               kingCastles.add(new Move.QueenSideCastle(this.board, this.playerKing, 58,
                     (Rook) rookTile.getPiece(), rookTile.getTileCoordinate(), 59));
            }

         }

      }

      return ImmutableList.copyOf(kingCastles);
   }

}//WhitePlayer


public class BlackPlayer extends Player {

   public BlackPlayer(final Board board,
         final Collection<Move> whiteStandarLegalMoves,
         final Collection<Move> blackStandarLegalMoves) {
      super(board, blackStandarLegalMoves, whiteStandarLegalMoves);
   }

   /**
    * get all black active pieces
    *
    * @return activePieces
    */
   @Override
   public Collection<Piece> getActivePieces() {
      return this.board.getBlackPieces();
   }

   /**
    * Set the alliance
```

```
 *
 * @return Alliance
 */
@Override
public Alliance getAlliance() {
   return Alliance.BLACK;
}

/**
 * take the info from opponent
 *
 * @return Player
 */
@Override
public Player getOpponent() {
   return this.board.whitePlayer();
}

/**
 * calculate king castles if possible , king side and queen side.
 *
 * @param playerLegals
 * @param opponentsLegals
 * @return
 */
@Override
protected Collection<Move> calculateKingCastles(final Collection<Move> playerLegals,
      final Collection<Move> opponentsLegals) {
   final List<Move> kingCastles = new ArrayList<>();

   if (this.playerKing.isFirstMove() && !isInCheck()) {
      //blacks King casle small
      if (!this.board.getTile(5).isTileOccupied() && !this.board.getTile(6).isTileOccupied()) {
         final Tile rookTile = this.board.getTile(7);

         if (rookTile.isTileOccupied() && rookTile.getPiece().isFirstMove()) {
            if (Player.calculateAttacksOnTile(5, opponentsLegals).isEmpty()
                  && Player.calculateAttacksOnTile(6, opponentsLegals).isEmpty()
                  && rookTile.getPiece().getPieceType().isRook()) {
               kingCastles.add(new KingSideCastle(this.board, this.playerKing, 6,
                     (Rook) rookTile.getPiece(), rookTile.getTileCoordinate(), 5));
            }
         }
      }
      //blacks King casle big
      if (!this.board.getTile(1).isTileOccupied() && !this.board.getTile(2).isTileOccupied()
            && !this.board.getTile(3).isTileOccupied()) {

         final Tile rookTile = this.board.getTile(0);

         if (rookTile.isTileOccupied() && rookTile.getPiece().isFirstMove()
               && Player.calculateAttacksOnTile(2, opponentsLegals).isEmpty()
               && Player.calculateAttacksOnTile(3, opponentsLegals).isEmpty()
               && rookTile.getPiece().getPieceType().isRook()) {
            kingCastles.add(new QueenSideCastle(this.board, this.playerKing, 2,
                  (Rook) rookTile.getPiece(), rookTile.getTileCoordinate(), 3));
         }
```

```
        }
      }

      return ImmutableList.copyOf(kingCastles);
    }

}//BlackPlayer


public class MoveTransition {

    private final Board fromBoard;//new
    private final Board transitionBoard;
    private final Move move;
    private final MoveStatus moveStatus;

    public MoveTransition(final Board fromBoard,
          final Board transitionBoard,
          final Move move,
          final MoveStatus moveStatus) {

      this.fromBoard = fromBoard;//new
      this.transitionBoard = transitionBoard;
      this.move = move;
      this.moveStatus = moveStatus;
    }

    public Board getFromBoard() {
      return this.fromBoard;
    }

    public MoveStatus getMoveStatus() {
      return this.moveStatus;
    }

    public Board getTransitionBoard() {
      return this.transitionBoard;
    }

}//MoveTransition

public enum MoveStatus {

    DONE {
      @Override
      public boolean isDone() {
        return true;
      }
    },
    ILLEGAL_MOVE {
      @Override
      public boolean isDone() {
        return false;
      }
    },
    LEAVES_PLAYER_IN_CHECK {
```

```
        @Override
        public boolean isDone() {
            return false;
        }

    };

    public abstract boolean isDone();
}//MoveStatus
```

MINMAX

```
public class MinMax implements MoveStrategy {

    private final BoardEvaluator boardEvaluator;
    private final int searchDepth;
    private final BoardEvaluator evaluator; //new
    private long boardsEvaluated; //new
    private long executionTime;//new
    private FreqTableRow[] freqTable;//new clss
    private int freqTableIndex;//new

    public MinMax(final int searchDepth) {
        this.evaluator = StandarBoardEvaluator.get();//new
        this.boardEvaluator = new StandarBoardEvaluator();
        this.searchDepth = searchDepth;
    }

    @Override
    public String toString() {
        return "MinMax";
    }

    @Override
    public long getNumBoardsEvaluated() {
        return this.boardsEvaluated;
    }//new

    @Override
    public Move execute(Board board) {
        final long startTime = System.currentTimeMillis();
        Move bestMove = MoveFactory.getNullMove();//new null;
        int highestSeenValue = Integer.MIN_VALUE;
        int lowestSeenValue = Integer.MAX_VALUE;
        int currentValue;

        System.out.println(board.getCurrentPlayer() + " I am thinking , Depth =" + this.searchDepth);
        this.freqTable = new FreqTableRow[board.getCurrentPlayer().getLegalMoves().size()];//new
        this.freqTableIndex = 0;//new
        int moveCounter = 1;
        int numMoves = board.getCurrentPlayer().getLegalMoves().size();

        for (final Move move : board.getCurrentPlayer().getLegalMoves()) {
            final MoveTransition moveTransition = board.getCurrentPlayer().makeMove(move);

            if (moveTransition.getMoveStatus().isDone()) {
                final FreqTableRow row = new FreqTableRow(move);
```

```
            this.freqTable[this.freqTableIndex] = row;
            currentValue = board.getCurrentPlayer().getAlliance().isWhite()
                  ? /*WhitePlayer*/ min(moveTransition.getTransitionBoard(), this.searchDepth - 1)
                  : /*BlackPlayer*/ max(moveTransition.getTransitionBoard(), this.searchDepth - 1);
            System.out.println("\t" + toString() + " analyzing move (" + moveCounter + "/" + numMoves +
") " + move
                  + " scores " + currentValue);//new + " " +this.freqTable[this.freqTableIndex]
            this.freqTableIndex++;//new

            if (board.getCurrentPlayer().getAlliance().isWhite() && currentValue >= highestSeenValue) {
               highestSeenValue = currentValue;
               bestMove = move;
            } else if (board.getCurrentPlayer().getAlliance().isBlack() && currentValue <=
lowestSeenValue) {
               lowestSeenValue = currentValue;
               bestMove = move;
            } else {
               System.out.println("\t" + toString() + " can't execute move (" + moveCounter + "/" +
numMoves + ") " + move);
            }//new
            moveCounter++;//new
         }
      }
      final long executionTime = System.currentTimeMillis() - startTime;
      System.out.printf("%s SELECTS %s [#boards = %d time taken = %d ms, rate = %.1f\n",
board.getCurrentPlayer(),
            bestMove, this.boardsEvaluated, this.executionTime, (1000 * ((double) this.boardsEvaluated
/ this.executionTime)));
      long total = 0;
      for (final FreqTableRow row : this.freqTable) {
         if (row != null) {
            total += row.getCount();
         }
      }
      if (this.boardsEvaluated != total) {
         System.out.println("somethings wrong with the # of boards evaluated!");
      }
      return bestMove;
   }

   private static boolean isEndGameScenario(final Board board) {
      return board.getCurrentPlayer().isInCheckMate()
            || board.getCurrentPlayer().isInStalemate();
   }

   //recursive mix call max and max call min !
   public int min(final Board board, final int depth) {
      if (depth == 0) {
         this.boardsEvaluated++;
         this.freqTable[this.freqTableIndex].increment();
         return this.evaluator.evaluate(board, depth);
      }//new
      if (isEndGameScenario(board)) {
         return this.boardEvaluator.evaluate(board, depth);
      }

      int lowestSeenValue = Integer.MAX_VALUE;
```

```
    for (final Move move : board.getCurrentPlayer().getLegalMoves()) {
      final MoveTransition moveTransition = board.getCurrentPlayer().makeMove(move);

      if (moveTransition.getMoveStatus().isDone()) {
        final int currentValue = max(moveTransition.getTransitionBoard(), depth - 1);
        if (currentValue <= lowestSeenValue) {
          lowestSeenValue = currentValue;
        }
      }
    }
    return lowestSeenValue;
  }

  public int max(final Board board, final int depth) {
    if (depth == 0) {
      this.boardsEvaluated++;
      this.freqTable[this.freqTableIndex].increment();
      return this.evaluator.evaluate(board, depth);
    }
    if (isEndGameScenario(board)) {
      return this.boardEvaluator.evaluate(board, depth);
    }

    int highestSeenValue = Integer.MIN_VALUE;

    for (final Move move : board.getCurrentPlayer().getLegalMoves()) {
      final MoveTransition moveTransition = board.getCurrentPlayer().makeMove(move);

      if (moveTransition.getMoveStatus().isDone()) {
        final int currentValue = min(moveTransition.getTransitionBoard(), depth - 1);
        if (currentValue >= highestSeenValue) {
          highestSeenValue = currentValue;
        }
      }
    }
    return highestSeenValue;
  }

  private static class FreqTableRow {

    private final Move move;
    private final AtomicLong count;

    FreqTableRow(final Move move) {
      this.count = new AtomicLong();
      this.move = move;
    }

    long getCount() {
      return this.count.get();
    }

    void increment() {
      this.count.incrementAndGet();
    }

    @Override
```

```
        public String toString() {
            return BoardUtils.getPositionAtCoordinate(this.move.getCurrentCoordinate())
                    + BoardUtils.getPositionAtCoordinate(this.move.getDestinationCoordinate()) + " : " +
this.count;
        }
    }
}

public final class StandarBoardEvaluator implements BoardEvaluator {

    private static final int CHECK_MATE_BONUS = 10000; //10000
    private static final int CHECK_BONUS = 50; //50
    private static final int DEPTH_BONUS = 250; //100
    private static final int CASTLE_BONUS = 300; //25
    private final static int TWO_BISHOPS_BONUS = 25; //25
    private final static int MOBILITY_MULTIPLIER = 100;//5
    private final static int ATTACK_MULTIPLIER = 2;//2
    private static final StandarBoardEvaluator INSTANCE = new StandarBoardEvaluator();

    public StandarBoardEvaluator() {
    }

    /*white for score >0 for black score <0 */
    @Override
    public int evaluate(final Board board, int depth) {
        return scorePlayer(board.whitePlayer(), depth)
                - scorePlayer(board.blackPlayer(), depth);

    }

    public static StandarBoardEvaluator get() {
        return INSTANCE;
    }

    public String evaluationDetails(final Board board, final int depth) {
        return ("White Mobility : " + mobility(board.whitePlayer()) + "\n")
                + "White kingThreats : " + kingThreats(board.whitePlayer(), depth) + "\n"
                + "White attacks : " + attacks(board.whitePlayer()) + "\n"
                + "White castle : " + castle(board.whitePlayer()) + "\n"
                + "White pieceEval : " + pieceEvaluations(board.whitePlayer()) + "\n"
                + //"White pawnStructure : " + pawnStructure(board.whitePlayer()) + "\n" +
                "---------------------\n"
                + "Black Mobility : " + mobility(board.blackPlayer()) + "\n"
                + "Black kingThreats : " + kingThreats(board.blackPlayer(), depth) + "\n"
                + "Black attacks : " + attacks(board.blackPlayer()) + "\n"
                + "Black castle : " + castle(board.blackPlayer()) + "\n"
                + "Black pieceEval : " + pieceEvaluations(board.blackPlayer()) + "\n"
                + //"Black pawnStructure : " + pawnStructure(board.blackPlayer()) + "\n\n" +
                "Final Score = " + evaluate(board, depth);
    }

    private int scorePlayer(final Player player, final int depth) {

        /*  return pieceValue(player) + mobility(player) + check(player) +checkMate(player,depth) +
                castled(player);*/
        // + Checkmate +++ check ++ castle Etc*/
        return mobility(player)
```

```
            + kingThreats(player, depth)
            + attacks(player)
            + castle(player)
            + pieceEvaluations(player)
            + checkMate(player, depth)
            + kingSafety(player)
            + pieceValue(player);
        //pawnStructure(player);
    }

    private static int attacks(final Player player) {
        int attackScore = 0;
        for (final Move move : player.getLegalMoves()) {
            if (move.isAttack()) {
                final Piece movedPiece = move.getMovedPiece();
                final Piece attackedPiece = move.getAttackedPiece();
                if (movedPiece.getPieceValue() <= attackedPiece.getPieceValue()) {
                    attackScore++;
                }
            }
        }
        return attackScore * ATTACK_MULTIPLIER;
    }//new

    private static int pieceEvaluations(final Player player) {
        int pieceValuationScore = 0;
        int numBishops = 0;
        for (final Piece piece : player.getActivePieces()) {
            pieceValuationScore += piece.getPieceValue() + piece.locationBonus();
            if (piece.getPieceType().isBishop()) {
                numBishops++;
            }
        }
        return pieceValuationScore + (numBishops == 2 ? TWO_BISHOPS_BONUS : 0);
    }//new

    private static int mobility(final Player player) {
        return MOBILITY_MULTIPLIER * mobilityRatio(player);
    }//new

    private static int mobilityRatio(final Player player) {
        return (int) ((player.getLegalMoves().size() * 100.0f) /
player.getOpponent().getLegalMoves().size());
    }//new

    private static int kingThreats(final Player player,
            final int depth) {
        return player.getOpponent().isInCheckMate() ? CHECK_MATE_BONUS * depthBonus(depth) :
check(player);
    }//new

    private static int check(final Player player) {
        return player.getOpponent().isInCheck() ? CHECK_BONUS : 0;
    }//new

    private static int castle(final Player player) {
        return player.isCastled() ? CASTLE_BONUS : 0;
```

```
    }//new
    //Create KingSafetyAnalyzer

    private static int kingSafety(final Player player) {
        final KingDistance kingDistance = KingSafetyAnalyzer.get().calculateKingTropism(player);
        return ((kingDistance.getEnemyPiece().getPieceValue() / 100) * kingDistance.getDistance());
    }
    //RookStructureAnalyzer

    private static int rookStructure(final Board board, final Player player) {
        return RookStructureAnalyzer.get().rookStructureScore(board, player);
    }//new

    private static int pieceValue(Player player) {
        int pieceValueScore = 0;
        for (final Piece piece : player.getActivePieces()) {
            pieceValueScore += piece.getPieceValue();
        }
        return pieceValueScore;
    }

    /* how many this.player legal moves got ?
      how many option i got as a player? /moves || attack || def?
     */
    private int checkMate(final Player player, int depth) {
        return player.getOpponent().isInCheckMate() ? CHECK_MATE_BONUS * depthBonus(depth) : 0;
    }

    private static int depthBonus(int depth) {

        return depth == 0 ? 1 : DEPTH_BONUS * depth;
    }

}
```

Και το UI

```
public class Table extends Observable {

    private final JFrame gameFrame;
    private final GameHistoryPanel gameHistoryPanel;
    private final TakenPiecesPanel takenPiecePanel;
    private final BoardPanel boardPanel;
    private final MoveLog moveLog;
    private final GameSetup gameSetup;
    private Board chessBoard;
    private final DebugPanel debugPanel;

    private Piece sourceTile;
    private Piece humanMovedPiece;
    private BoardDirection boardDirection;

    private Move computerMove;

    private boolean highLightLegalMoves;
```

```
  private final static Dimension OUTER_FRAME_DIMENSION = new Dimension(1000, 1000);//800,800
//1000 , 1000
  private final static Dimension BOARD_PANEL_DIMENSION = new Dimension(600, 450); //400,350
//600,450
  private final static Dimension TILE_PANEL_DIMENSION = new Dimension(10, 10);
  private Color lightTileColor = Color.decode("#FFFACD");
  private Color darkTileColor = Color.decode("#593E1A");

  private static String iconPiecesPath = "resources\\PiecesSet\\"; //"resources\\pieceIcons\\"; //src

  private int counter = 0;

  private static final Table INSTANCE = new Table();

  private Table() {
    this.gameFrame = new JFrame("Chess");
    this.gameFrame.setLayout(new BorderLayout());
    final JMenuBar tableMenuBar = createTableMenuBar();
    this.gameFrame.setJMenuBar(tableMenuBar);
    this.gameFrame.setSize(OUTER_FRAME_DIMENSION);
    this.chessBoard = Board.createStandardBoard();
    this.gameHistoryPanel = new GameHistoryPanel();
    this.debugPanel = new DebugPanel(); //new
    this.takenPiecePanel = new TakenPiecesPanel();
    this.boardPanel = new BoardPanel();
    this.moveLog = new MoveLog();
    this.addObserver(new TableGameAIWatcher());
    this.gameSetup = new GameSetup(this.gameFrame, true);
    this.boardDirection = BoardDirection.NORMAL;
    this.highLightLegalMoves = false;
    this.gameFrame.add(this.takenPiecePanel, BorderLayout.WEST);
    this.gameFrame.add(this.boardPanel, BorderLayout.CENTER);
    this.gameFrame.add(this.gameHistoryPanel, BorderLayout.EAST);
    this.gameFrame.add(debugPanel, BorderLayout.SOUTH); // new
    gameHistoryPanel.add(jButtonJPanel, BorderLayout.SOUTH);
    jButtonJPanel.setLayout(new BorderLayout(70, 5));
    jButtonJPanel.add(resignButton(), BorderLayout.NORTH);
    jButtonJPanel.add(drawButton(), BorderLayout.SOUTH);
    setDefaultLookAndFeelDecorated(true); //new

    center(this.gameFrame);
    this.gameFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    this.gameFrame.setVisible(true);
  }

  public static Table get() {
    return INSTANCE;
  }

  private GameSetup getGameSetup() {
    return this.gameSetup;
  }

  public Board getGameBoard() {
    return this.chessBoard;
  }
```

```
private JFrame getGameFrame() {
   return this.gameFrame;
}

private MoveLog getMoveLog() {
   return this.moveLog;
}

private boolean getHighlightLegalMoves() {
   return this.highLightLegalMoves;
}

public void show() {
   Table.get().getMoveLog().clear();
   Table.get().getGameHistoryPanel().redo(chessBoard, Table.get().getMoveLog());
   Table.get().getTakenPiecesPanel().redo(Table.get().getMoveLog());
   Table.get().getBoardPanel().drawBoard(Table.get().getGameBoard());
   Table.get().getDebugPanel().redo();//new

}

JPanel jButtonJPanel = new JPanel();

private JMenuBar createTableMenuBar() {
   final JMenuBar tableMenuBar = new JMenuBar();
   tableMenuBar.add(createFileMenu()); //new
   tableMenuBar.add(createPreferencesMenu());
   tableMenuBar.add(createOptionsMenu());

   return tableMenuBar;
}

private static void center(final JFrame frame) {
   final Dimension dim = Toolkit.getDefaultToolkit().getScreenSize();
   final int w = frame.getSize().width;
   final int h = frame.getSize().height;
   final int x = (dim.width - w) / 2;
   final int y = (dim.height - h) / 2;
   frame.setLocation(x, y);
}

private JMenu createFileMenu() {
   final JMenu filesMenu = new JMenu("File");
   filesMenu.setMnemonic(KeyEvent.VK_F);

   final JMenuItem openFEN = new JMenuItem("Load FEN File", KeyEvent.VK_F);
   openFEN.addActionListener(new ActionListener() {
      @Override
      public void actionPerformed(final ActionEvent e) {
         String fenString = JOptionPane.showInputDialog("Input FEN");
         undoAllMoves();
         chessBoard = FenUtilities.createGameFromFEN(fenString);
         Table.get().getBoardPanel().drawBoard(chessBoard);
      }
   });
   filesMenu.add(openFEN);
```

```
    final JMenuItem saveToPGN = new JMenuItem("Save Game", KeyEvent.VK_S);
    saveToPGN.addActionListener(new ActionListener() {
      @Override
      public void actionPerformed(final ActionEvent e) {
        final JFileChooser chooser = new JFileChooser();
        chooser.setFileFilter(new FileFilter() {
          @Override
          public String getDescription() {
            return ".pgn";
          }

          @Override
          public boolean accept(final File file) {
            return file.isDirectory() || file.getName().toLowerCase().endsWith("pgn");
          }
        });
        final int option = chooser.showSaveDialog(Table.get().getGameFrame());
        if (option == JFileChooser.APPROVE_OPTION) {
          savePGNFile(chooser.getSelectedFile());
        }
      }
    });
    filesMenu.add(saveToPGN);

    final JMenuItem createChess960 = new JMenuItem("Chess960");
    createChess960.addActionListener(new ActionListener() {
      @Override
      public void actionPerformed(final ActionEvent e) {
        undoAllMoves();
        chessBoard = FenUtilities.createGameFromFEN(
            FenUtilities.randomGenaratorGame(Table.get().getGameBoard()));
        Table.get().getBoardPanel().drawBoard(chessBoard);

      }
    });
    filesMenu.add(createChess960);

    final JMenuItem exitMenuItem = new JMenuItem("Exit", KeyEvent.VK_X);
    exitMenuItem.addActionListener(new ActionListener() {
      @Override
      public void actionPerformed(final ActionEvent e) {
        Table.get().getGameFrame().dispose();
        System.exit(0);
      }
    });
    filesMenu.add(exitMenuItem);

    return filesMenu;
  }

  private JButton resignButton() {
    JButton resign = new JButton("Resign");
    resign.addActionListener(new ActionListener() {
      @Override
      public void actionPerformed(ActionEvent e) {
        int dialog = 0;
```

```
            dialog = JOptionPane.showConfirmDialog(null,
Table.get().getGameBoard().getCurrentPlayer() + "You wana Resign the game", "Warning", dialog);
            if (dialog == JOptionPane.YES_OPTION) {

              JOptionPane.showMessageDialog(Table.get().getBoardPanel(),
                   "Game Over: Player " + Table.get().getGameBoard().getCurrentPlayer() + " Resign the
match", "Game Over",
                   JOptionPane.INFORMATION_MESSAGE);

              savePGNFile(new File("resources\\lastGame.txt"));

              int dialogResult = 0;
              dialogResult = JOptionPane.showConfirmDialog(null,
Table.get().getGameBoard().getCurrentPlayer() + "New game?", "Warning", dialogResult);
              if (dialogResult == JOptionPane.YES_OPTION) {
                undoAllMoves();
              }//if

            }//if

          }//actionPerformed
      });
      return resign;
    }

    public JButton drawButton() {
      JButton draw = new JButton("Draw");
      draw.addActionListener(new ActionListener() {
        @Override
        public void actionPerformed(ActionEvent e) {
          int dialogResult = 0;
          dialogResult = JOptionPane.showConfirmDialog(null,
Table.get().getGameBoard().getCurrentPlayer() + "Offers Draw", "Warning", dialogResult);
            if (dialogResult == JOptionPane.YES_OPTION) {
              JOptionPane.showMessageDialog(Table.get().getBoardPanel(),
                   "Game Over: Player " + Table.get().getGameBoard().getCurrentPlayer().getOpponent()
+ " Draw the match", "Game Over",
                   JOptionPane.INFORMATION_MESSAGE);
            }
          }

      });
      return draw;
    }

    public JMenu createPreferencesMenu() {
      final JMenu preferencesMenu = new JMenu("Preferences");
      final JMenuItem flipBoardMenuItem = new JMenuItem("Flip Board");

      flipBoardMenuItem.addActionListener(new ActionListener() {
        @Override
        public void actionPerformed(final ActionEvent e) {
          boardDirection = boardDirection.opposite();
          boardPanel.drawBoard(chessBoard);

        }
      });
```

```
    preferencesMenu.add(flipBoardMenuItem);

    preferencesMenu.addSeparator();
    final JCheckBoxMenuItem legalMoveHighlightCheckBox = new JCheckBoxMenuItem("HighLight Legal Moves", false);
    legalMoveHighlightCheckBox.addActionListener(new ActionListener() {
      @Override
      public void actionPerformed(ActionEvent e) {
        highLightLegalMoves = legalMoveHighlightCheckBox.isSelected();
      }
    });
    preferencesMenu.add(legalMoveHighlightCheckBox);

    final JMenu colorChooserSubMenu = new JMenu("Choose Colors");
    colorChooserSubMenu.setMnemonic(KeyEvent.VK_S);

    final JMenuItem chooseDarkMenuItem = new JMenuItem("Choose Dark Tile Color");
    colorChooserSubMenu.add(chooseDarkMenuItem);

    final JMenuItem chooseLightMenuItem = new JMenuItem("Choose Light Tile Color");
    colorChooserSubMenu.add(chooseLightMenuItem);

    preferencesMenu.add(colorChooserSubMenu);

    chooseDarkMenuItem.addActionListener(new ActionListener() {
      @Override
      public void actionPerformed(final ActionEvent e) {
        final Color colorChoice = JColorChooser.showDialog(Table.get().getGameFrame(), "Choose Dark Tile Color",
            Table.get().getGameFrame().getBackground());
        if (colorChoice != null) {
          Table.get().getBoardPanel().setTileDarkColor(chessBoard, colorChoice);
        }
      }
    });

    chooseLightMenuItem.addActionListener(new ActionListener() {
      @Override
      public void actionPerformed(final ActionEvent e) {
        final Color colorChoice = JColorChooser.showDialog(Table.get().getGameFrame(), "Choose Light Tile Color",
            Table.get().getGameFrame().getBackground());
        if (colorChoice != null) {
          Table.get().getBoardPanel().setTileLightColor(chessBoard, colorChoice);
        }
      }
    });

    return preferencesMenu;
  }

  private JMenu createOptionsMenu() {
    final JMenu optionsMenu = new JMenu("Options");
    optionsMenu.setMnemonic(KeyEvent.VK_O);

    final JMenuItem resetMenuItem = new JMenuItem("New Game", KeyEvent.VK_P);
```

```
        resetMenuItem.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(final ActionEvent e) {
                undoAllMoves();
            }

        });
        optionsMenu.add(resetMenuItem);

        final JMenuItem legalMovesMenuItem = new JMenuItem("Current State", KeyEvent.VK_L);
        legalMovesMenuItem.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(final ActionEvent e) {

                System.out.println(chessBoard.getWhitePieces());
                System.out.println(chessBoard.getBlackPieces());
                System.out.println(playerInfo(chessBoard.getCurrentPlayer()));
                System.out.println(playerInfo(chessBoard.getCurrentPlayer().getOpponent()));

                Table.get().debugPanel.update(INSTANCE, chessBoard.getWhitePieces() + "\n");
                Table.get().debugPanel.update(INSTANCE, chessBoard.getBlackPieces());
                Table.get().debugPanel.update(INSTANCE, playerInfo(chessBoard.getCurrentPlayer()));
                Table.get().debugPanel.update(INSTANCE,
playerInfo(chessBoard.getCurrentPlayer().getOpponent()));
                Table.get().debugPanel.update(INSTANCE,
FenUtilities.createFENFromGame(Table.get().getGameBoard()));

            }
        });
        optionsMenu.add(legalMovesMenuItem);

        final JMenuItem undoMoveMenuItem = new JMenuItem("Undo last move", KeyEvent.VK_M);
        undoMoveMenuItem.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(final ActionEvent e) {
                if (Table.get().getMoveLog().size() > 0) {
                    undoLastMove();
                }
            }
        });
        optionsMenu.add(undoMoveMenuItem);

        final JMenuItem setupGameMenuItem = new JMenuItem("Setup Game", KeyEvent.VK_S);
        setupGameMenuItem.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(final ActionEvent e) {
                Table.get().getGameSetup().promptUser();
                Table.get().setupUpdate(Table.get().getGameSetup());
            }
        });
        optionsMenu.add(setupGameMenuItem);

        final JMenuItem evaluateBoardMenuItem = new JMenuItem("Evaluate Board", KeyEvent.VK_E);
        evaluateBoardMenuItem.addActionListener(e                                                    ->
System.out.println(StandarBoardEvaluator.get().evaluationDetails(chessBoard,
gameSetup.getSearchDepth())));
        optionsMenu.add(evaluateBoardMenuItem);
```

```
        final JMenuItem escapeAnalysis = new JMenuItem("Escape Analysis Score", KeyEvent.VK_S);
        escapeAnalysis.addActionListener(e -> {
            final Move lastMove = moveLog.getMoves().get(moveLog.size() - 1);
            if (lastMove != null) {
                System.out.println(MoveUtils.exchangeScore(lastMove));
            }

        });
        optionsMenu.add(escapeAnalysis);

        return optionsMenu;
    }

    private static void savePGNFile(final File pgnFile) {
        try {
            writeGameToPGNFile(pgnFile,
                    Table.get().getMoveLog(),
                    Table.get().getGameBoard().getCurrentPlayer(),
                    Table.get().getGameBoard());
        } catch (final IOException e) {
            e.printStackTrace();
        }
    }

    private static String playerInfo(final Player player) {
        return ("Player is: " + player.getAlliance() + "\nlegal moves (" + player.getLegalMoves().size() + ")
= " + player.getLegalMoves() + "\ninCheck = "
                + player.isInCheck() + "\nisInCheckMate = " + player.isInCheckMate()
                + "\nisCastled = " + player.isCastled()) + "\n";
    }//new

    private void undoAllMoves() {
        for (int i = Table.get().getMoveLog().size() - 1; i >= 0; i--) {
            final Move lastMove = Table.get().getMoveLog().removeMove(Table.get().getMoveLog().size() -
1);
            this.chessBoard                                                                                =
this.chessBoard.getCurrentPlayer().unMakeMove(lastMove).getTransitionBoard();
        }
        this.computerMove = null;
        Table.get().getMoveLog().clear();
        Table.get().getGameHistoryPanel().redo(chessBoard, Table.get().getMoveLog());
        Table.get().getTakenPiecesPanel().redo(Table.get().getMoveLog());
        Table.get().getBoardPanel().drawBoard(chessBoard);
        Table.get().getDebugPanel().redo();
    }//new

    private void undoLastMove() {
        final Move lastMove = Table.get().getMoveLog().removeMove(Table.get().getMoveLog().size() -
1);
        this.chessBoard                                                                                =
this.chessBoard.getCurrentPlayer().unMakeMove(lastMove).getTransitionBoard();
        this.computerMove = null;
        Table.get().getMoveLog().removeMove(lastMove);
        Table.get().getGameHistoryPanel().redo(chessBoard, Table.get().getMoveLog());
        Table.get().getTakenPiecesPanel().redo(Table.get().getMoveLog());
        Table.get().getBoardPanel().drawBoard(chessBoard);
```

```
      Table.get().getDebugPanel().redo();
    }//new

    public void updateGameBoard(final Board board) {
      this.chessBoard = board;
    }

    public void updateComputerMove(final Move move) {
      this.computerMove = move;
    }

    private void setupUpdate(final GameSetup gameSetup) {
      setChanged();
      notifyObservers(gameSetup);
    }

    private void moveMadeUpdate(final PlayerType playerType) {
      setChanged();
      notifyObservers(playerType);
    }

    private static class TableGameAIWatcher implements Observer {

      @Override
      public void update(final Observable o, final Object arg) {

        if (Table.get().getGameSetup().isAIPlayer(Table.get().getGameBoard().getCurrentPlayer())
            && !Table.get().getGameBoard().getCurrentPlayer().isInCheckMate()
            && !Table.get().getGameBoard().getCurrentPlayer().isInStalemate())/* &&
            !Table.get().fiftyMoveRule(Table.get().getMoveLog()))*/ {//test
          //create AI thread!
          System.out.println(Table.get().getGameBoard().getCurrentPlayer() + " is set to AI,
thinking....");
          //Table.get().debugPanel.update(INSTANCE," is set to AI, thinking....");
          final AIThinkTank thinkTank = new AIThinkTank();
          thinkTank.execute();

        }

        if (Table.get().isKingPat() || Table.get().isKingKnightPat() || Table.get().isKingBishopPat()
            || Table.get().replayRule()) {

          JOptionPane.showMessageDialog(Table.get().getBoardPanel(),
              "Game Over: is PAT", "Game Over",
              JOptionPane.INFORMATION_MESSAGE);
          savePGNFile(new File("resources\\autoPatTestGame.txt"));
        }

        if (Table.get().getGameBoard().getCurrentPlayer().isInCheckMate()) {
          JOptionPane.showMessageDialog(Table.get().getBoardPanel(),
              "Game Over: Player " + Table.get().getGameBoard().getCurrentPlayer() + " is in
checkmate!", "Game Over",
              JOptionPane.INFORMATION_MESSAGE);
          savePGNFile(new File("resources\\lastTestGame.txt"));

        }
        if (Table.get().getGameBoard().getCurrentPlayer().isInStalemate()) {
```

```
            JOptionPane.showMessageDialog(Table.get().getBoardPanel(),
                    "Game Over: Player " + Table.get().getGameBoard().getCurrentPlayer() + " is in
stalemate!", "Game Over",
                    JOptionPane.INFORMATION_MESSAGE);
          }

          /* if(Table.get().fiftyMoveRule()){
          JOptionPane.showMessageDialog(Table.get().getBoardPanel(),
                    "Game Over: Player " + Table.get().getGameBoard().getCurrentPlayer() + " is to much
Moves!", "Game Over",
                JOptionPane.INFORMATION_MESSAGE);
        }*/
        }//new

    }

    /*private boolean fiftyMoveRule(){
      if(moveLog.getMoves().get(getMoveLog().size()-1).toString().contains("x")){
           counter = 0;
           System.out.println("Counter in IF :"+counter);

         }

        else if(!(moveLog.getMoves().get(getMoveLog().size()-1).toString().contains("x"))){
          counter++;
          System.out.println("Counter in else :"+counter);
          }
        if(counter > 4){
           return true;
        }

       return false;
    }*/
    private boolean replayRule() {
      for (int i = 2; i < moveLog.size(); i++) {
         if  (moveLog.getMoves().get(i).toString().compareTo(moveLog.getMoves().get(i  -  2).toString())
== 0) {
            return true;
         }
      }
      return false;
    }

    private boolean isKingPat() {
      if ((Table.get().chessBoard.getBlackPieces().toString().compareTo("[K]") == 0)
            && (Table.get().chessBoard.getWhitePieces().toString().compareTo("[K]") == 0)) {
         return true;

      } else {
         return false;
      }
    }
    //check this again
    // [K,N] [K] [N, K]

    private boolean isKingKnightPat() {
      if ((((Table.get().chessBoard.getBlackPieces().toString().compareTo("[K, N]") == 0)
```

```
        || (Table.get().chessBoard.getBlackPieces().toString().compareTo("[N, K]") == 0)
        || (Table.get().chessBoard.getBlackPieces().toString().compareTo("[K]") == 0)))
        && (Table.get().chessBoard.getWhitePieces().toString().compareTo("[K, N]") == 0
        || (Table.get().chessBoard.getWhitePieces().toString().compareTo("[N, K]") == 0)
        || (Table.get().chessBoard.getWhitePieces().toString().compareTo("[K]") == 0))) {
      /*
       (Table.get().chessBoard.getWhitePieces().toString().compareTo("[K, N]") == 0)) ||
      ((Table.get().chessBoard.getBlackPieces().toString().compareTo("[N, K]") == 0)  ||
      (Table.get().chessBoard.getBlackPieces().toString().compareTo("[K]") == 0)    &&
      (Table.get().chessBoard.getWhitePieces().toString().compareTo("[N, K]") == 0) ||
      ((Table.get().chessBoard.getBlackPieces().toString().compareTo("[K, N]") == 0) &&
      (Table.get().chessBoard.getWhitePieces().toString().compareTo("[N, K]") == 0) ||
      (Table.get().chessBoard.getBlackPieces().toString().compareTo("[N, K]") == 0) &&
      (Table.get().chessBoard.getWhitePieces().toString().compareTo("[K, N]") == 0)){*/
      return true;

    } else {
      return false;
    }

}

private boolean isKingBishopPat() {
    if ((Table.get().chessBoard.getBlackPieces().toString().compareTo("[K, B]") == 0)
          && (Table.get().chessBoard.getWhitePieces().toString().compareTo("[K, B]") == 0)
          || ((Table.get().chessBoard.getBlackPieces().toString().compareTo("[B, K]") == 0)
          && (Table.get().chessBoard.getWhitePieces().toString().compareTo("[B, K]") == 0))
          || ((Table.get().chessBoard.getBlackPieces().toString().compareTo("[K, B]") == 0)
          && (Table.get().chessBoard.getWhitePieces().toString().compareTo("[B, K]") == 0))
          || (((Table.get().chessBoard.getBlackPieces().toString().compareTo("[B, K]") == 0)
          && (Table.get().chessBoard.getWhitePieces().toString().compareTo("[K, N]") == 0)))) {
      return true;

    } else {
      return false;
    }

}

private GameHistoryPanel getGameHistoryPanel() {
    return this.gameHistoryPanel;
}

private TakenPiecesPanel getTakenPiecesPanel() {
    return this.takenPiecePanel;
}

private DebugPanel getDebugPanel() {
    return this.debugPanel;
} //new

private BoardPanel getBoardPanel() {
    return this.boardPanel;
}

public enum PlayerType {
    HUMAN,
```

```
      COMPUTER

    }

    private static class AIThinkTank extends SwingWorker<Move, String> {

      private AIThinkTank() {

      }

      @Override
      protected Move doInBackground() throws Exception {
        final MoveStrategy minMax = new MinMax(Table.get().gameSetup.getSearchDepth()); //new
MinMax(4);
        final Move bestMove = minMax.execute(Table.get().getGameBoard());
        System.out.println("best move :" + bestMove);
        /*  final AlphaBetaWithMoveOrdering strategy =
              new     AlphaBetaWithMoveOrdering(Table.get().getGameSetup().getSearchDepth()     +
bonusDepth);
          strategy.addObserver(Table.get().getDebugPanel());
          bestMove = strategy.execute(
              Table.get().getGameBoard());*/
        return bestMove;
      }

      @Override
      public void done() {

        try {
          final Move bestMove = get();

          Table.get().updateComputerMove(bestMove);

Table.get().updateGameBoard(Table.get().getGameBoard().getCurrentPlayer().makeMove(bestMove).
getTransitionBoard());
          Table.get().getMoveLog().addMove(bestMove);
          Table.get().getGameHistoryPanel().redo(Table.get().getGameBoard(),
Table.get().getMoveLog());
          Table.get().getTakenPiecesPanel().redo(Table.get().getMoveLog());
          Table.get().getBoardPanel().drawBoard(Table.get().getGameBoard());
          Table.get().moveMadeUpdate(PlayerType.COMPUTER);

        } catch (InterruptedException ex) {
          ex.printStackTrace();
        } catch (ExecutionException ex) {
          ex.printStackTrace();
        }

      }

    }

    public enum BoardDirection {
      NORMAL {
        @Override
        List<TilePanel> traverse(List<TilePanel> boardTiles) {
          return boardTiles;
```

```
      }

      @Override
      BoardDirection opposite() {
        return FLIPPED;
      }
    },
    FLIPPED {
      @Override
      List<TilePanel> traverse(List<TilePanel> boardTiles) {
        return Lists.reverse(boardTiles);
      }

      @Override
      BoardDirection opposite() {
        return NORMAL;
      }
    };

    abstract List<TilePanel> traverse(final List<TilePanel> boardTiles);

    abstract BoardDirection opposite();
  }//BoardDirection

  private class BoardPanel extends JPanel {

    final List<TilePanel> boardTiles;

    public BoardPanel() {
      super(new GridLayout(8, 8));

      this.boardTiles = new ArrayList<>();
      for (int i = 0; i < BoardUtils.NUM_TILES; i++) {
        final TilePanel tilePanel = new TilePanel(this, i);
        this.boardTiles.add(tilePanel);
        add(tilePanel);
      }
      setPreferredSize(BOARD_PANEL_DIMENSION);
      setBorder(BorderFactory.createEmptyBorder(10, 10, 10, 10));
      setBackground(Color.decode("#8B4726"));
      validate();
    }

    public void drawBoard(final Board board) {
      removeAll();
      for (final TilePanel tilePanel : boardDirection.traverse(boardTiles)) {
        tilePanel.drawTile(board);
        add(tilePanel);
      }

      validate();
      repaint();
    }

    void setTileDarkColor(final Board board,
        final Color darkColor) {
      for (final TilePanel boardTile : boardTiles) {
```

```
            boardTile.setDarkTileColor(darkColor);
        }
        drawBoard(board);
    }

    void setTileLightColor(final Board board,
            final Color lightColor) {
        for (final TilePanel boardTile : boardTiles) {
            boardTile.setLightTileColor(lightColor);
        }
        drawBoard(board);
    }

}//BoardPanel

public static class MoveLog {

    private final List<Move> moves;

    public MoveLog() {
        this.moves = new ArrayList<>();
    }

    public List<Move> getMoves() {
        return this.moves;
    }

    public void addMove(final Move move) {
        this.moves.add(move);
    }

    public int size() {
        return this.moves.size();
    }

    public void clear() {
        this.moves.clear();
    }

    public Move removeMove(int index) {
        return this.moves.remove(index);
    }

    public boolean removeMove(final Move move) {
        return this.moves.remove(move);
    }

}//MoveLog

private class TilePanel extends JPanel {

    private final int tileId;

    public TilePanel(final BoardPanel boardPanel, final int tileId) {
        super(new GridBagLayout());
        this.tileId = tileId;
        setPreferredSize(TILE_PANEL_DIMENSION);
```

```
        assignTileColor();
        assignTilePieceIcon(chessBoard);

        addMouseListener(new MouseListener() {
          @Override
          public void mouseClicked(final MouseEvent e) {

            if (Table.get().getGameSetup().isAIPlayer(Table.get().getGameBoard().getCurrentPlayer())
                || BoardUtils.isEndGame(Table.get().getGameBoard())) {
              return;
            }

            if (SwingUtilities.isRightMouseButton(e)) {
              sourceTile = null;
              humanMovedPiece = null;
            } else if (SwingUtilities.isLeftMouseButton(e)) {
              if (sourceTile == null) {
                sourceTile = chessBoard.getPiece(tileId);
                humanMovedPiece = sourceTile;
                if (humanMovedPiece == null) {
                  sourceTile = null;
                }
              } else {
                final Move move = MoveFactory.createMove(chessBoard,
sourceTile.getPiecePosition(),
                    tileId);
                final MoveTransition transition = chessBoard.getCurrentPlayer().makeMove(move);
                if (transition.getMoveStatus().isDone()) {
                  chessBoard = transition.getTransitionBoard();
                  moveLog.addMove(move);
                }
                sourceTile = null;
                humanMovedPiece = null;
              }
            }
            SwingUtilities.invokeLater(new Runnable() {
              public void run() {
                gameHistoryPanel.redo(chessBoard, moveLog);
                takenPiecePanel.redo(moveLog);
                //if (gameSetup.isAIPlayer(chessBoard.getCurrentPlayer())) {
                Table.get().moveMadeUpdate(PlayerType.HUMAN);
                //}
                boardPanel.drawBoard(chessBoard);
                debugPanel.redo();
              }
            });

          }

          @Override
          public void mousePressed(final MouseEvent e) {
            if (e.isAltDown()) {
              setBackground(Color.red);
            }
            //  System.oSyut.println("MousePressed");
          }
```

```
        @Override
        public void mouseReleased(final MouseEvent e) {
          // if(tileId == tileId){
          if (e.isAltDown()) {
            setBackground(Color.red);
          }
          //  }

          // System.out.println("MouseRealeased");
        }

        @Override
        public void mouseEntered(final MouseEvent e) {

        }

        @Override
        public void mouseExited(final MouseEvent e) {

        }

      });
      validate();
    }

    public void drawTile(final Board board) {
      assignTileColor();
      assignTilePieceIcon(board);
      highlightTileBorder(board);
      highlightLegals(board);

      highlightAIMove();
      validate();
      repaint();
    }

    void setLightTileColor(final Color color) {
      lightTileColor = color;
    }

    void setDarkTileColor(final Color color) {
      darkTileColor = color;
    }

    private void highlightTileBorder(final Board board) {
      if (humanMovedPiece != null
          && humanMovedPiece.getPieceAlliance() == board.getCurrentPlayer().getAlliance()
          && humanMovedPiece.getPiecePosition() == this.tileId) {
        setBorder(BorderFactory.createLineBorder(Color.cyan));
      } else {
        setBorder(BorderFactory.createLineBorder(Color.GRAY));
      }
    }//new

    private void highlightAIMove() {
      if (computerMove != null) {
        if (this.tileId == computerMove.getCurrentCoordinate()) {
```

```
                setBackground(Color.pink);
            } else if (this.tileId == computerMove.getDestinationCoordinate()) {
                setBackground(Color.red);
            }
        }
    }//new

    private void assignTilePieceIcon(final Board board) {
        this.removeAll();
        if (board.getTile(this.tileId).isTileOccupied()) {

            try {
                final BufferedImage image = ImageIO.read(new File(iconPiecesPath
                        + board.getTile(this.tileId).getPiece().getPieceAlliance().toString().substring(0, 1)
                        + board.getTile(tileId).getPiece().toString() + ".gif"));
                add(new JLabel(new ImageIcon(image)));
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }//new

    private void highlightLegals(final Board board) {
        if (Table.get().getHighlightLegalMoves()) { //if(highLightLegalMoves){
            for (final Move move : pieceLegalMoves(board)) {
                if (move.getDestinationCoordinate() == this.tileId) {
                    try {
                        add(new JLabel(new ImageIcon(ImageIO.read(new
File("resources//dots//green_dot.png"))))); //"src//misc//green_dot.png"))));
                    } catch (Exception e) {
                        e.printStackTrace();
                    }
                }
            }
        }
    }

    private Collection<Move> pieceLegalMoves(final Board board) {
        if (humanMovedPiece != null && humanMovedPiece.getPieceAlliance() ==
board.getCurrentPlayer().getAlliance()) {
            return humanMovedPiece.calculateLegalMoves(board);

        }
        return Collections.emptyList();
    }

    private void assignTileColor() {
        if (BoardUtils.INSTANCE.EIGHTH_ROW.get(this.tileId)
                || BoardUtils.INSTANCE.SIXTH_ROW.get(this.tileId)
                || BoardUtils.INSTANCE.FOURTH_ROW.get(this.tileId)
                || BoardUtils.INSTANCE.SECOND_ROW.get(this.tileId)) {
            setBackground(this.tileId % 2 == 0 ? darkTileColor : lightTileColor);
//setBackground(this.tileId % 2 == 0 ? lightTileColor : darkTileColor);

        } else if (BoardUtils.INSTANCE.SEVENTH_ROW.get(this.tileId)
                || BoardUtils.INSTANCE.FIFTH_ROW.get(this.tileId)
                || BoardUtils.INSTANCE.THIRD_ROW.get(this.tileId)
```

```
                || BoardUtils.INSTANCE.FIRST_ROW.get(this.tileId)) {
            setBackground(this.tileId % 2 != 0 ? darkTileColor : lightTileColor);
            //setBackground(this.tileId % 2 != 0 ? lightTileColor : darkTileColor);
          }
        }

    }//TilePanel

}//Table

class DebugPanel extends JPanel implements Observer {

    private static final Dimension DEBUGGING_PANEL = new Dimension(600, 150); //500,150
    private final JTextArea jTextArea;

    public DebugPanel() {
        super(new BorderLayout());
        this.jTextArea = new JTextArea("");
        add(new JScrollPane(jTextArea));
        setPreferredSize(DEBUGGING_PANEL);
        validate();
        setVisible(true);
    }

    public void redo() {
        validate();
    }

    @Override
    public void update(final Observable obs,
            final Object obj) {
        this.jTextArea.append(obj.toString().trim() + "\n");
        //this.jTextArea.setText(obj.toString().trim() + "\n");
        redo();
    }

}

public class GameHistoryPanel extends JPanel {

    private final DataModel model;
    private final JScrollPane scrollPane;

    private static final Dimension HISTORY_PANEL_DIMENSION = new Dimension(100, 400); //check
100,400 , 120,400

    GameHistoryPanel() {
        this.setLayout(new BorderLayout());
        this.model = new DataModel();
        final JTable table = new JTable(model);
        table.setRowHeight(15);
        this.scrollPane = new JScrollPane(table);
        scrollPane.setColumnHeaderView(table.getTableHeader());
        scrollPane.setPreferredSize(HISTORY_PANEL_DIMENSION);
        this.add(scrollPane, BorderLayout.CENTER);
        this.setVisible(true);
    }
```

```
  void redo(final Board board, final MoveLog moveHistory) {
    int currentRow = 0;
    this.model.clear();
    for (final Move move : moveHistory.getMoves()) {
      final String moveText = move.toString();
      if (move.getMovedPiece().getPieceAlliance().isWhite()) {
        this.model.setValueAt(moveText, currentRow, 0);
      } else if (move.getMovedPiece().getPieceAlliance().isBlack()) {
        this.model.setValueAt(moveText, currentRow, 1);
        currentRow++;
      }
    }
    if (moveHistory.getMoves().size() > 0) {
      final Move lastMove = moveHistory.getMoves().get(moveHistory.size() - 1);
      final String moveText = lastMove.toString();

      if (lastMove.getMovedPiece().getPieceAlliance().isWhite()) {
        this.model.setValueAt(moveText + calculateCheckAndCheckMateHash(board), currentRow,
0);
      } else if (lastMove.getMovedPiece().getPieceAlliance().isBlack()) {
        this.model.setValueAt(moveText + calculateCheckAndCheckMateHash(board), currentRow -
1, 1);
      }
    }

    final JScrollBar vertical = scrollPane.getVerticalScrollBar();
    vertical.setValue(vertical.getMaximum());
  }

  private String calculateCheckAndCheckMateHash(final Board board) {
    if (board.getCurrentPlayer().isInCheckMate()) {
      return "#";
    } else if (board.getCurrentPlayer().isInCheck()) {
      return "+";
    }
    return "";

  }

  private static class DataModel extends DefaultTableModel {

    private final List<Row> values;
    private static final String[] NAMES = {"White", "Black"};

    DataModel() {
      this.values = new ArrayList<>();
    }

    public void clear() {
      this.values.clear();
      setRowCount(0);
    }

    @Override
    public int getRowCount() {
      if (this.values == null) {
```

```
            return 0;
        }
        return this.values.size();
    }

    @Override
    public int getColumnCount() {
        return NAMES.length;
    }

    @Override
    public Object getValueAt(final int row, final int column) {
        final Row currentRow = this.values.get(row);
        if (column == 0) {
            return currentRow.getWhiteMove();
        } else if (column == 1) {
            return currentRow.getBlackMove();
        }
        return null;
    }

    @Override
    public void setValueAt(final Object aValue, final int row, final int column) {
        final Row currentRow;
        if (this.values.size() <= row) {
            currentRow = new Row();
            this.values.add(currentRow);
        } else {
            currentRow = this.values.get(row);
        }
        if (column == 0) {
            currentRow.setWhiteMove((String) aValue);
            fireTableRowsInserted(row, row);
        } else if (column == 1) {
            currentRow.setBlackMove((String) aValue);
            fireTableCellUpdated(row, column);
        }
    }

    @Override
    public Class<?> getColumnClass(final int column) {
        return Move.class;
    }

    @Override
    public String getColumnName(final int column) {
        return NAMES[column];
    }

} //DataModel

private static class Row {

    private String whiteMove;
    private String blackMove;

    Row() {
```

```
        }

        public String getWhiteMove() {
            return this.whiteMove;
        }

        public String getBlackMove() {
            return this.blackMove;
        }

        public void setWhiteMove(final String move) {
            this.whiteMove = move;
        }

        public void setBlackMove(final String move) {
            this.blackMove = move;
        }

    }//Row

}//GameHistoryPanel
class GameSetup extends JDialog {

    private PlayerType whitePlayerType;
    private PlayerType blackPlayerType;
    private JSpinner searchDepthSpinner;

    private static final String HUMAN_TEXT = "Human";
    private static final String COMPUTER_TEXT = "Computer";

    GameSetup(final JFrame frame,
            final boolean modal) {
        super(frame, modal);
        final JPanel myPanel = new JPanel(new GridLayout(0, 1));
        final JRadioButton whiteHumanButton = new JRadioButton(HUMAN_TEXT);
        final JRadioButton whiteComputerButton = new JRadioButton(COMPUTER_TEXT);
        final JRadioButton blackHumanButton = new JRadioButton(HUMAN_TEXT);
        final JRadioButton blackComputerButton = new JRadioButton(COMPUTER_TEXT);
        whiteHumanButton.setActionCommand(HUMAN_TEXT);
        final ButtonGroup whiteGroup = new ButtonGroup();
        whiteGroup.add(whiteHumanButton);
        whiteGroup.add(whiteComputerButton);
        whiteHumanButton.setSelected(true);

        final ButtonGroup blackGroup = new ButtonGroup();
        blackGroup.add(blackHumanButton);
        blackGroup.add(blackComputerButton);
        blackHumanButton.setSelected(true);

        getContentPane().add(myPanel);
        myPanel.add(new JLabel("White"));
        myPanel.add(whiteHumanButton);
        myPanel.add(whiteComputerButton);
        myPanel.add(new JLabel("Black"));
        myPanel.add(blackHumanButton);
        myPanel.add(blackComputerButton);
```

```
    myPanel.add(new JLabel("Search"));
    this.searchDepthSpinner = addLabeledSpinner(myPanel, "Search Depth", new
SpinnerNumberModel(6, 0, Integer.MAX_VALUE, 1));

    final JButton cancelButton = new JButton("Cancel");
    final JButton okButton = new JButton("OK");

    okButton.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        whitePlayerType = whiteComputerButton.isSelected() ? PlayerType.COMPUTER :
PlayerType.HUMAN;
        blackPlayerType = blackComputerButton.isSelected() ? PlayerType.COMPUTER :
PlayerType.HUMAN;
        GameSetup.this.setVisible(false);
      }
    });

    cancelButton.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        System.out.println("Cancel");
        GameSetup.this.setVisible(false);
      }
    });

    myPanel.add(cancelButton);
    myPanel.add(okButton);

    setLocationRelativeTo(frame);
    pack();
    setVisible(false);
  }

  void promptUser() {
    setVisible(true);
    repaint();
  }

  boolean isAIPlayer(final Player player) {
    if (player.getAlliance() == Alliance.WHITE) {
      return getWhitePlayerType() == PlayerType.COMPUTER;
    }
    return getBlackPlayerType() == PlayerType.COMPUTER;
  }

  PlayerType getWhitePlayerType() {
    return this.whitePlayerType;
  }

  PlayerType getBlackPlayerType() {
    return this.blackPlayerType;
  }

  private static JSpinner addLabeledSpinner(final Container c,
      final String label,
      final SpinnerModel model) {
    final JLabel l = new JLabel(label);
    c.add(l);
```

```
        final JSpinner spinner = new JSpinner(model);
        l.setLabelFor(spinner);
        c.add(spinner);
        return spinner;
    }

    public int getSearchDepth() {
        return (Integer) this.searchDepthSpinner.getValue();
    }
}


class TakenPiecesPanel extends JPanel {

    private final JPanel northPanel;
    private final JPanel southPanel;

    private static final long serialVersionUID = 1L;
    private static final Color PANEL_COLOR = Color.decode("0xFDF5E6");
    private static final Dimension TAKEN_PIECES_PANEL_DIMENSION = new Dimension(100, 80);
    private static final EtchedBorder PANEL_BORDER = new EtchedBorder(EtchedBorder.RAISED);

    public TakenPiecesPanel() {
        super(new BorderLayout());
        setBackground(Color.decode("0xFDF5E6"));
        setBorder(PANEL_BORDER);
        this.northPanel = new JPanel(new GridLayout(8, 2));
        this.southPanel = new JPanel(new GridLayout(8, 2));
        this.northPanel.setBackground(PANEL_COLOR);
        this.southPanel.setBackground(PANEL_COLOR);
        add(this.northPanel, BorderLayout.NORTH);
        add(this.southPanel, BorderLayout.SOUTH);
        setPreferredSize(TAKEN_PIECES_PANEL_DIMENSION);
    }

    public void redo(final MoveLog moveLog) {
        southPanel.removeAll();
        northPanel.removeAll();

        final List<Piece> whiteTakenPieces = new ArrayList<>();
        final List<Piece> blackTakenPieces = new ArrayList<>();

        for (final Move move : moveLog.getMoves()) {
            if (move.isAttack()) {
                final Piece takenPiece = move.getAttackedPiece();
                if (takenPiece.getPieceAlliance().isWhite()) {
                    whiteTakenPieces.add(takenPiece);
                } else if (takenPiece.getPieceAlliance().isBlack()) {
                    blackTakenPieces.add(takenPiece);
                } else {
                    throw new RuntimeException("Should not reach here!");
                }
            }
        }

        Collections.sort(whiteTakenPieces, new Comparator<Piece>() {
            @Override
```

```
            public int compare(final Piece p1, final Piece p2) {
                return Ints.compare(p1.getPieceValue(), p2.getPieceValue());
            }
        });

        Collections.sort(blackTakenPieces, new Comparator<Piece>() {
            @Override
            public int compare(final Piece p1, final Piece p2) {
                return Ints.compare(p1.getPieceValue(), p2.getPieceValue());
            }
        });

        for (final Piece takenPiece : whiteTakenPieces) {
            try {
                final BufferedImage image = ImageIO.read(new File("resources\\pieceIcons\\"
                        + takenPiece.getPieceAlliance().toString().substring(0, 1) + "" + takenPiece.toString()
                        + ".gif"));
                final ImageIcon ic = new ImageIcon(image);
                final JLabel imageLabel = new JLabel(new ImageIcon(ic.getImage().getScaledInstance(
                        ic.getIconWidth() - 5, ic.getIconWidth() - 5, Image.SCALE_SMOOTH)));
                this.southPanel.add(imageLabel);
            } catch (final IOException e) {
                e.printStackTrace();
            }
        }

        for (final Piece takenPiece : blackTakenPieces) {
            try {
                final BufferedImage image = ImageIO.read(new File("resources\\pieceIcons\\"
                        + takenPiece.getPieceAlliance().toString().substring(0, 1) + "" + takenPiece.toString()
                        + ".gif"));
                final ImageIcon ic = new ImageIcon(image);
                final JLabel imageLabel = new JLabel(new ImageIcon(ic.getImage().getScaledInstance(
                        ic.getIconWidth() - 5, ic.getIconWidth() - 5, Image.SCALE_SMOOTH)));
                this.northPanel.add(imageLabel);

            } catch (final IOException e) {
                e.printStackTrace();
            }
        }
        repaint();
        validate();
    }
}

UTILS
public enum Alliance {
    WHITE {
        /**
         * It determine the pawn direction for White player.
         * @return
         */
        @Override
        public int getDirection() {
            return -1;
        }
        /**
```

```
 * It determine the pawn direction for White player.
 * @return
 */
@Override
public int getOppositeDirection() {
   return 1;
}

@Override
public boolean isWhite() {
   return true;
}

@Override
public boolean isBlack() {
   return false;
}
/**
 * It used to cover the rule when a pawn reach the final rank.
 * @param position
 * @return
 */
 @Override
public boolean isPawnPromotionSquare(int position) {
   return BoardUtils.INSTANCE.FIRST_ROW.get(position);
}

/**
 * It determine the turns for the players.
 * @param whitePlayer
 * @param blackPlayer
 * @return
 */
@Override
public Player choosePlayer(final WhitePlayer whitePlayer, final BlackPlayer blackPlayer) {
   return whitePlayer;
}

  @Override
public int pawnBonus(final int position) {
   return WHITE_PAWN_PREFERRED_COORDINATES[position];
}

@Override
public int knightBonus(final int position) {
   return WHITE_KNIGHT_PREFERRED_COORDINATES[position];
}

@Override
public int bishopBonus(final int position) {
   return WHITE_BISHOP_PREFERRED_COORDINATES[position];
}

@Override
public int rookBonus(final int position) {
   return WHITE_ROOK_PREFERRED_COORDINATES[position];
}
```

```
    @Override
    public int queenBonus(final int position) {
        return WHITE_QUEEN_PREFERRED_COORDINATES[position];
    }

    @Override
    public int kingBonus(final int position) {
        return WHITE_KING_PREFERRED_COORDINATES[position];
    }



},
BLACK {
    /**
     * It determine the pawn direction for Black player.
     * @return
     */
    @Override
   public int getDirection() {
        return 1;
    }
   /**
     * It determine the pawn direction for Black player.
     * @return
     */
   @Override
    public int getOppositeDirection() {
        return -1;
    }

    @Override
    public boolean isWhite() {
        return false;
    }

    @Override
    public boolean isBlack() {
        return true;
    }
    /**
     * It determine the turns for the players.
     * @param whitePlayer
     * @param blackPlayer
     * @return
     */
    @Override
    public boolean isPawnPromotionSquare(int position) {
        return BoardUtils.INSTANCE.EIGHTH_ROW.get(position);
    }
     /**
     * It determine the turns for the players.
     * @param whitePlayer
     * @param blackPlayer
     * @return
     */
```

```
    @Override
    public Player choosePlayer(final WhitePlayer whitePlayer, final BlackPlayer blackPlayer) {
      return blackPlayer;
    }

    @Override
    public int pawnBonus(final int position) {
      return BLACK_PAWN_PREFERRED_COORDINATES[position];
    }

    @Override
    public int knightBonus(final int position) {
      return BLACK_KNIGHT_PREFERRED_COORDINATES[position];
    }

    @Override
    public int bishopBonus(final int position) {
      return BLACK_BISHOP_PREFERRED_COORDINATES[position];
    }

    @Override
    public int rookBonus(final int position) {
      return BLACK_ROOK_PREFERRED_COORDINATES[position];
    }

    @Override
    public int queenBonus(final int position) {
      return BLACK_QUEEN_PREFERRED_COORDINATES[position];
    }

    @Override
    public int kingBonus(final int position) {
      return BLACK_KING_PREFERRED_COORDINATES[position];
    }



  };

public abstract int getDirection();
public abstract int getOppositeDirection();
public abstract boolean isWhite();
public abstract boolean isBlack();
public abstract boolean isPawnPromotionSquare(int position);
public abstract Player choosePlayer(WhitePlayer whitePlayer, BlackPlayer blackPlayer);

  public abstract int pawnBonus(int position);//new

  public abstract int knightBonus(int position);//new

  public abstract int bishopBonus(int position);//new

  public abstract int rookBonus(int position);//new

  public abstract int queenBonus(int position);//new

  public abstract int kingBonus(int position); //new
```

```
private final static int[] WHITE_PAWN_PREFERRED_COORDINATES = {
      0,  0,  0,  0,  0,  0,  0,  0,
      75, 75, 75, 75, 75, 75, 75, 75,
      25, 25, 29, 29, 29, 29, 25, 25,
      5,  5, 10, 25, 25, 10,  5,  5,
      0,  0,  0, 20, 20,  0,  0,  0,
      5, -5,-10,  0,  0,-10, -5,  5,
      5, 10, 10,-20,-20, 10, 10,  5,
      0,  0,  0,  0,  0,  0,  0,  0
};

private final static int[] BLACK_PAWN_PREFERRED_COORDINATES = {
      0,  0,  0,  0,  0,  0,  0,  0,
      5, 10, 10,-20,-20, 10, 10,  5,
      5, -5,-10,  0,  0,-10, -5,  5,
      0,  0,  0, 20, 20,  0,  0,  0,
      5,  5, 10, 25, 25, 10,  5,  5,
      25, 25, 29, 29, 29, 29, 25, 25,
      75, 75, 75, 75, 75, 75, 75, 75,
      0,  0,  0,  0,  0,  0,  0,  0
};

private final static int[] WHITE_KNIGHT_PREFERRED_COORDINATES = {
      -50,-40,-30,-30,-30,-30,-40,-50,
      -40,-20,  0,  0,  0,  0,-20,-40,
      -30,  0, 10, 15, 15, 10,  0,-30,
      -30,  5, 15, 20, 20, 15,  5,-30,
      -30,  0, 15, 20, 20, 15,  0,-30,
      -30,  5, 10, 15, 15, 10,  5,-30,
      -40,-20,  0,  5,  5,  0,-20,-40,
      -50,-40,-30,-30,-30,-30,-40,-50
};

private final static int[] BLACK_KNIGHT_PREFERRED_COORDINATES = {
      -50,-40,-30,-30,-30,-30,-40,-50,
      -40,-20,  0,  5,  5,  0,-20,-40,
      -30,  5, 10, 15, 15, 10,  5,-30,
      -30,  0, 15, 20, 20, 15,  0,-30,
      -30,  5, 15, 20, 20, 15,  5,-30,
      -30,  0, 10, 15, 15, 10,  0,-30,
      -40,-20,  0,  0,  0,  0,-20,-40,
      -50,-40,-30,-30,-30,-30,-40,-50,
};

private final static int[] WHITE_BISHOP_PREFERRED_COORDINATES = {
      -20,-10,-10,-10,-10,-10,-10,-20,
      -10,  0,  0,  0,  0,  0,  0,-10,
      -10,  0,  5, 10, 10,  5,  0,-10,
      -10,  5,  5, 10, 10,  5,  5,-10,
      -10,  0, 10, 10, 10, 10,  0,-10,
      -10, 10, 10, 10, 10, 10, 10,-10,
      -10,  5,  0,  0,  0,  0,  5,-10,
      -20,-10,-10,-10,-10,-10,-10,-20
};

private final static int[] BLACK_BISHOP_PREFERRED_COORDINATES = {
```

```
        -20,-10,-10,-10,-10,-10,-10,-20,
        -10,  5,  0,  0,  0,  0,  5,-10,
        -10, 10, 10, 10, 10, 10, 10,-10,
        -10,  0, 10, 10, 10, 10,  0,-10,
        -10,  5,  5, 10, 10,  5,  5,-10,
        -10,  0,  5, 10, 10,  5,  0,-10,
        -10,  0,  0,  0,  0,  0,  0,-10,
        -20,-10,-10,-10,-10,-10,-10,-20,
};

private final static int[] WHITE_ROOK_PREFERRED_COORDINATES = {
        0,  0,  0,  0,  0,  0,  0,  0,
        5, 20, 20, 20, 20, 20, 20,  5,
        -5,  0,  0,  0,  0,  0,  0, -5,
        -5,  0,  0,  0,  0,  0,  0, -5,
        -5,  0,  0,  0,  0,  0,  0, -5,
        -5,  0,  0,  0,  0,  0,  0, -5,
        -5,  0,  0,  0,  0,  0,  0, -5,
        0,  0,  0,  5,  5,  0,  0,  0
};

private final static int[] BLACK_ROOK_PREFERRED_COORDINATES = {
        0,  0,  0,  5,  5,  0,  0,  0,
        -5,  0,  0,  0,  0,  0,  0, -5,
        -5,  0,  0,  0,  0,  0,  0, -5,
        -5,  0,  0,  0,  0,  0,  0, -5,
        -5,  0,  0,  0,  0,  0,  0, -5,
        -5,  0,  0,  0,  0,  0,  0, -5,
        5, 20, 20, 20, 20, 20, 20,  5,
        0,  0,  0,  0,  0,  0,  0,  0,
};

private final static int[] WHITE_QUEEN_PREFERRED_COORDINATES = {
        -20,-10,-10, -5, -5,-10,-10,-20,
        -10,  0,  0,  0,  0,  0,  0,-10,
        -10,  0,  5,  5,  5,  5,  0,-10,
         -5,  0,  5,  5,  5,  5,  0, -5,
          0,  0,  5,  5,  5,  5,  0, -5,
        -10,  5,  5,  5,  5,  5,  0,-10,
        -10,  0,  5,  0,  0,  0,  0,-10,
        -20,-10,-10, -5, -5,-10,-10,-20
};

private final static int[] BLACK_QUEEN_PREFERRED_COORDINATES = {
        -20,-10,-10, -5, -5,-10,-10,-20,
        -10,  0,  5,  0,  0,  0,  0,-10,
        -10,  5,  5,  5,  5,  5,  0,-10,
          0,  0,  5,  5,  5,  5,  0, -5,
          0,  0,  5,  5,  5,  5,  0, -5,
        -10,  0,  5,  5,  5,  5,  0,-10,
        -10,  0,  0,  0,  0,  0,  0,-10,
        -20,-10,-10, -5, -5,-10,-10,-20
};

private final static int[] WHITE_KING_PREFERRED_COORDINATES = {
        -30,-40,-40,-50,-50,-40,-40,-30,
        -30,-40,-40,-50,-50,-40,-40,-30,
```

```
        -30,-40,-40,-50,-50,-40,-40,-30,
        -30,-40,-40,-50,-50,-40,-40,-30,
        -20,-30,-30,-40,-40,-30,-30,-20,
        -10,-20,-20,-20,-20,-20,-20,-10,
         20, 20,  0,  0,  0,  0, 20, 20,
         20, 30, 10,  0,  0, 10, 30, 20
    };

    private final static int[] BLACK_KING_PREFERRED_COORDINATES = {
         20, 30, 10,  0,  0, 10, 30, 20,
         20, 20,  0,  0,  0,  0, 20, 20,
        -10,-20,-20,-20,-20,-20,-20,-10,
        -20,-30,-30,-40,-40,-30,-30,-20,
        -30,-40,-40,-50,-50,-40,-40,-30,
        -30,-40,-40,-50,-50,-40,-40,-30,
        -30,-40,-40,-50,-50,-40,-40,-30,
        -30,-40,-40,-50,-50,-40,-40,-30
    }; //new

}//Alliance

public class FenUtilities {

    private FenUtilities() {
        throw new RuntimeException("Not Instantiable!");
    }

    public static Board createGameFromFEN(final String fenString) {
        return parseFEN(fenString);
    }

    public static String createFENFromGame(final Board board) {
        return calculateBoardText(board) + " "
                + calculateCurrentPlayerText(board) + " "
                + calculateCastleText(board) + " "
                + calculateEnPassantSquare(board) + " "
                + "0 1";
    }

    public static String randomGenaratorGame(final Board board) {
        //pppppppp/8/8/8/8/PPPPPPPP rnbqkbnr

        String[] chessData = new String[]{"r", "r", "n", "n", "b", "b", "q", "k"};
        String chess960 = "/pppppppp/8/8/8/8/PPPPPPPP/";
        String fen = "";

        Random rand = new Random();

        for (int i = 0; i < chessData.length; i++) {
            int randomIndexToSwap = rand.nextInt(chessData.length);
            String temp = chessData[randomIndexToSwap];
            chessData[randomIndexToSwap] = chessData[i];
            chessData[i] = temp;
        }

        for (int j = 0; j < chessData.length; j++) {
            fen += chessData[j];
```

```
    }

    return fen + chess960 + fen.toUpperCase() + " w KQkq - 0 1";
  }

  private static Board parseFEN(final String fenString) {
    final String[] fenPartitions = fenString.trim().split(" ");
    final Builder builder = new Builder();
    final boolean whiteKingSideCastle = whiteKingSideCastle(fenPartitions[2]);
    final boolean whiteQueenSideCastle = whiteQueenSideCastle(fenPartitions[2]);
    final boolean blackKingSideCastle = blackKingSideCastle(fenPartitions[2]);
    final boolean blackQueenSideCastle = blackQueenSideCastle(fenPartitions[2]);
    final String gameConfiguration = fenPartitions[0];
    final char[] boardTiles = gameConfiguration.replaceAll("/", "")
          .replaceAll("8", "--------")
          .replaceAll("7", "-------")
          .replaceAll("6", "------")
          .replaceAll("5", "-----")
          .replaceAll("4", "----")
          .replaceAll("3", "---")
          .replaceAll("2", "--")
          .replaceAll("1", "-")
          .toCharArray();
    int i = 0;
    while (i < boardTiles.length) {
      switch (boardTiles[i]) {
        case 'r':
          builder.setPiece(new Rook(Alliance.BLACK, i));
          i++;
          break;
        case 'n':
          builder.setPiece(new Knight(Alliance.BLACK, i));
          i++;
          break;
        case 'b':
          builder.setPiece(new Bishop(Alliance.BLACK, i));
          i++;
          break;
        case 'q':
          builder.setPiece(new Queen(Alliance.BLACK, i));
          i++;
          break;
        case 'k':
          final boolean isCastled = !blackKingSideCastle && !blackQueenSideCastle;
          builder.setPiece(new King(Alliance.BLACK, i, blackKingSideCastle, blackQueenSideCastle));
          i++;
          break;
        case 'p':
          builder.setPiece(new Pawn(Alliance.BLACK, i));
          i++;
          break;
        case 'R':
          builder.setPiece(new Rook(Alliance.WHITE, i));
          i++;
          break;
        case 'N':
```

```
            builder.setPiece(new Knight(Alliance.WHITE, i));
            i++;
            break;
          case 'B':
            builder.setPiece(new Bishop(Alliance.WHITE, i));
            i++;
            break;
          case 'Q':
            builder.setPiece(new Queen(Alliance.WHITE, i));
            i++;
            break;
          case 'K':
            builder.setPiece(new King(Alliance.WHITE, i, whiteKingSideCastle, whiteQueenSideCastle));
            i++;
            break;
          case 'P':
            builder.setPiece(new Pawn(Alliance.WHITE, i));
            i++;
            break;
          case '-':
            i++;
            break;
          default:
            throw new RuntimeException("Invalid FEN String " + gameConfiguration);
        }
      }
      builder.setMoveMaker(moveMaker(fenPartitions[1]));
      return builder.build();
    }

    private static Alliance moveMaker(final String moveMakerString) {
      if (moveMakerString.equals("w")) {
        return Alliance.WHITE;
      } else if (moveMakerString.equals("b")) {
        return Alliance.BLACK;
      }
      throw new RuntimeException("Invalid FEN String " + moveMakerString);
    }

    private static boolean whiteKingSideCastle(final String fenCastleString) {
      return fenCastleString.contains("K");
    }

    private static boolean whiteQueenSideCastle(final String fenCastleString) {
      return fenCastleString.contains("Q");
    }

    private static boolean blackKingSideCastle(final String fenCastleString) {
      return fenCastleString.contains("k");
    }

    private static boolean blackQueenSideCastle(final String fenCastleString) {
      return fenCastleString.contains("q");
    }

    private static String calculateCastleText(final Board board) {
      final StringBuilder builder = new StringBuilder();
```

```
        if (board.whitePlayer().isKingSideCastleCapable()) {
            builder.append("K");
        }
        if (board.whitePlayer().isQueenCastleCapable()) {
            builder.append("Q");
        }
        if (board.blackPlayer().isKingSideCastleCapable()) {
            builder.append("k");
        }
        if (board.blackPlayer().isQueenCastleCapable()) {
            builder.append("q");
        }
        final String result = builder.toString();

        return result.isEmpty() ? "-" : result;
    }

    private static String calculateEnPassantSquare(final Board board) {
        final Pawn enPassantPawn = board.getEnPassantPawn();
        if (enPassantPawn != null) {
            return BoardUtils.getPositionAtCoordinate(enPassantPawn.getPiecePosition()
                    + (8) * enPassantPawn.getPieceAlliance().getOppositeDirection());
        }
        return "-";
    }

    private static String calculateBoardText(final Board board) {
        final StringBuilder builder = new StringBuilder();
        for (int i = 0; i < BoardUtils.NUM_TILES; i++) {
            final String tileText = board.getPiece(i) == null ? "-"
                    : board.getPiece(i).getPieceAlliance().isWhite() ? board.getPiece(i).toString()
                    : board.getPiece(i).toString().toLowerCase();
            builder.append(tileText);
        }
        builder.insert(8, "/");
        builder.insert(17, "/");
        builder.insert(26, "/");
        builder.insert(35, "/");
        builder.insert(44, "/");
        builder.insert(53, "/");
        builder.insert(62, "/");
        return builder.toString()
                .replaceAll("--------", "8")
                .replaceAll("-------", "7")
                .replaceAll("------", "6")
                .replaceAll("-----", "5")
                .replaceAll("----", "4")
                .replaceAll("---", "3")
                .replaceAll("--", "2")
                .replaceAll("-", "1");
    }

    private static String calculateCurrentPlayerText(final Board board) {
        return board.getCurrentPlayer().toString().substring(29, 30).toLowerCase();
    }

}
```