עבודת גמר לקבלת תואר

טכנאי תוכנה

סמל מוסד: 47102

שם מכללה: מכללת אורט הרמלין נתניה

שם הסטודנט: דותן רייפר

ת"ז הסטודנט: 214471120

שם הפרויקט: משחק שחמט

שמות המנחים : מיכאל צ'רנובילסקי, ירון מזרחי

אפריל 2022 תשפ"ב

תוכן

תקציר

שחמט הוא משחק לוח אסטרטגי מופשט וענף ספורט המיועד לשני שחקנים. זהו אחד מהמשחקים השכיחים והמורכבים ביותר הקיימים בתרבות האנושית; המשחק מקובל ברחבי העולם כתחביב וכספורט תחרותי כאחד.

במהלך הפרויקט המוצג בספר זה, אממש משחק שחמט, ובו שחקן ממוחשב, בשפת התכנות java.

ההערכות למספר המצבים החוקיים בשחמט נעות בין 10^43 לבין 10^47, ולכן כפי שניתן להבין לא ניתן לפתור את המשחק בצורה מלאה משום שיש יותר מדי מהלכים ולא ניתן בכל רגע נתון לבחור את המהלך הטוב ביותר לחלוטין, לכן בפרויקט הזה ננסה לבנות בינה מלאכותית שמסוגלת לשחק את המשחק בצורה טובה ככל הניתן.

לשם ביצוע השחקן הממוחשב נעזר בהערכת עמדה-היכולת להעריך עמדה נתונה על פי טיבה עבור כל שחקן, ובאמצעות כך לבחור את העמדות הטובות ביותר.

בשחמט-הערכת העמדה היא תורה שלמה, ויש אינספור צורות ודרכים להעריך עמדה, ובמהלך פרויקט זה אנסה לממש כמה שיותר מהן-על מנת לקבל הערכת עמדה מוצלחת ככל שיותר.

בנוסף, אבנה משחק שחמט של שחקן אנושי כנגד שחקן אנושי, אשר יהווה הבסיס עליו נבנה את הבינה המלאכותית- שתדע לשחק שחמט עבור השחקן הממוחשב.

בנימה אישית, שיחקתי כ-10 שנים שחמט תחרותי, השתתפתי בתחרויות ובחוגים, ואני מאמין ומקווה כי הידע הקודם שלי במשחק יוכל לעזור לי בבניית הפרויקט המורכב.

בספר פרויקט זה- יוצגו הדרכים לפתרון הבעיה, האסטרטגיות שננקטו, האלגוריתמים שמומשו, והסברים מפורטים על דרך פתרון הבעיה- שהיא בניית שחקן ממוחשב שישחק בצורה טובה כנגד שחקן אנושי, וישאף לנצחו.

מושגים

- HashMap- מבנה נתונים מילוני אשר נותן גישה לרשומה באמצעות המפתח המתאים לה. המבנה הזה עובד באמצעות הפיכת המפתח על ידי פונקציית הגיבוב, למספר המייצג מיקום במערך שמפנה אל הרשומה המבוקשת. מפת הגיבוב היא מבנה נתונים אשר מאפשר שליפה של ערכים על פי מפתח ביעילות של o(1), מה שיאפשר שליפת ערכים מהירה עבור מיקומים בלוח, ולכן אשתמש במפת גיבוב על מנת לשמור את מצב המשחק הנוכחי.
- עומק- כמות המהלכים שהבינה הממוחשבת מחשבת קדימה- לדוגמא בעומק 4, הבינה המלאכותית תחשב עמדות שיתקבלו ממהלכים אפשריים בעוד 4 מהלכים.
- עלה בעץ משחק- עמדה סופית אותה מעריך המחשב, עמדה זו היא עמדה שנגיע אליה כאשר נגיע לעומק הרצוי בעץ, או כאשר נגיע למצב של סיום משחק.
- היוריסטיקה – גישה לפתרון בעיות שמפעילה שיטה פרקטית לפתרון שאינו בהכרח אופטימלי, לרוב נשתמש בהיוריסטיקה על מנת ליעל את האלגוריתמים הקיימים שלנו ולהאיץ אותם.
- effect horizon the - אפקט שמתרחש בבינה מלאכותית שמשתמש בעץ מהלכים. מכיוון והבינה שלנו מוגבלת בעומק מסוים של מהלכים שהיא מסוגלת לחשוב קדימה בהם, אנו יכולים להגיע למצב בו ה"אופק" שאנחנו מסוגלים לראות לא מתאר נכון את המצב של המשחק והציון שאנו ניתן למהלכים מסוימים יהיה שגוי. במשחק שחמט – אם נסתכל רק מהלך קדימה נראה שלדוגמא המלכה שלנו מסוגלת לאכול רגלי אך מכיוון ואנו לא מחפשים את המהלך הבא לא נראה שהמלכה שלנו תאבד במהלך הבא – אנו נחשוב שהרווחנו רגלי אחד ולא הפסדנו מלכה ולכן נשתמש באלגוריתם search Quiescence .
- מהלך שקט- מהלך יציב יחסית שנוכל לחשב אותו בעומק ההתחלתי שנבחר ולא נצטרך להוסיף לו עומק חישוב באלגוריתם ה quiescence.
- Piece-Square-Table- מערכים בגודל הלוח, אשר מתארים את הערך אשר יש להוסיף להערכת עמדה עבור כל כלי המיקום מסוים.

תיאור הפרויקט

שחמט

שחמט הוא משחק אסטרטגיה לשני שחקנים, המשוחק על לוח משבצות בגודל 8 על שמונה:

הסידור ההתחלתי של שחמט

תיאור הכלים

מלך - הכלי החשוב ביותר על הלוח. ביכולתו לנוע לכל כיוון (ישר או אלכסון) משבצת אחת בכל כיוון למעט "הצרחה" (ראו בהמשך).

כאשר המלך נמצא במצב של "שח", כלומר במצב שבו היריב יוכל "להכותו" בתורו הבא, השחקן המאוים חייב לשנות מצב זה על ידי הזזת המלך, חסימת קו ההתקפה של הכלי המאיים, או על ידי הכאה של הכלי המאיים. על מנת לנצח במשחק, על השחקן להגיע למצב שבו יריבו לא יוכל למלט את מלכו מהשח, להכות את הכלי המאיים או לחסום את האיום על ידי כלי משלו. מצב זה נקרא "מט".

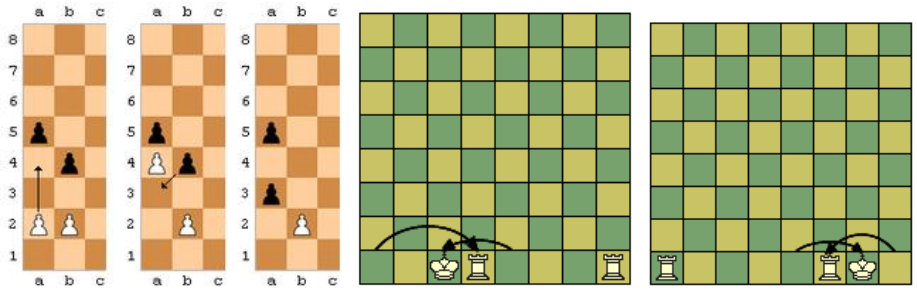
מלכה - הכלי החזק ביותר על הלוח. יתרונה הוא בגמישות התנועה שלה, שכן היא יכולה לנוע בכל קו ישר (טור, שורה או אלכסון) ולכמה מספר משבצות שהיא רוצה באותה שורה או אלכסון.

צריח - הצריח יכול לנוע רק לאורך הטורים או השורות. תפקיד מיוחד נועד לצריח בהגנת המלך במהלך המכונה הצרחה.

רץ - הרץ יכול לנוע רק באלכסונים.

פרש - לפרש תנועה מיוחדת שמשלבת תנועה ישר ובאלכסון. לחלופין ניתן לתאר את תנועתו כאות 'ך' המסובבת בכיוונים שונים.

רגלי - לנוע רק משבצת אחת בכל פעם ורק לכיוון אחד ("קדימה", ז"א, הכיוון הנגדי לצד הלוח בו הוא מתחיל), במהלכו הראשון של הרגלי, ורק במהלך זה, הוא יכול לנוע שתי משבצות קדימה או משבצת אחת בהתאם לרצון השחקן. אופן ההכאה של הרגלי מתבצע באופן שונה מאופן תנועתו הרגיל, על ידי מעבר משבצת אחת באלכסון, אך ורק בכיוון התקדמותו. אם הוא מגיע לשורה האחרונה של הלוח הוא הופך לכל כלי אחר, פרט למלך, באותו הצבע, על-פי בחירת השחקן (גם אם הכלי הנבחר טרם יצא מהמשחק). - מצב זה קרוי "הכתרה". מהלך מיוחד של הרגלי הוא הכאה דרך הילוכו.



הצרחה קטנה הצרחה גדולה הכאת דרך הילוכו

מטרת המשחק

מטרת המשחק היא לתת מט למלך היריב, בהינתן מט, השחקן אשר נתן את המת מנצח, והמשחק מסתיים. כמו כן, ניתן האופציה להיכנע, בה השחקן הנכנע מפסיד, וגם ניתנת האופציה להציע תיקו לשחקן היריב.

סיום המשחק

משחק מגיע לסיום בהכרעה עבור אחד הצדדים, כאשר אחד השחקנים עושה מט למלך היריב או כאשר אחד השחקנים נכנע, כמו כן, ייתכן מצב של תיקו במקרים הבאים:

אם הצד שתורו לשחק אינו יכול לבצע אף מסע חוקי אך אינו נמצא באיום של שח. תיקו כזה נקרא פט.

כאשר לא נותר לאף אחד משני השחקנים חומר מספיק כדי לתת מט - מלך מול מלך; מלך ורץ/פרש מול מלך.

אם אותה העמדה מופיעה בפעם השלישית (אפילו לא רצוף, ובתנאי שבכל פעם תורו של אותו צד לשחק).

אם נעשו 50 מסעים רצופים ללא כל הכאה או הזזת רגלי.

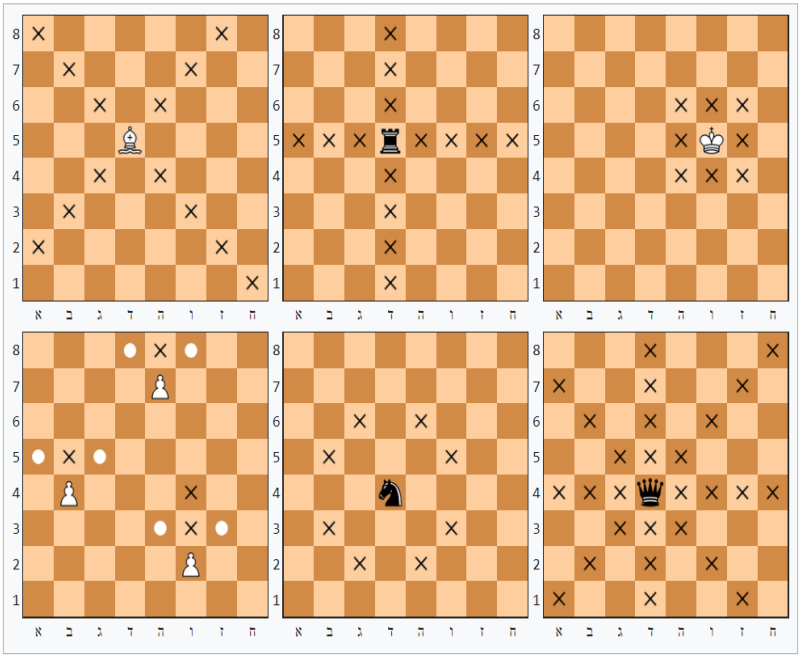
כאשר שני הצדדים מסכימים על תיקו.

נחשב לתיקו גם מצב של שח נצחי, היינו כאשר אחד הצדדים מאיים בשח שוב ושוב באופן בלתי פוסק.

יש לציין-מצב של שח נצחי ושל חזרה על אותה עמדה בפעם השלישית הם הרחבות של חוב 50 המהלכים, שכן אם מתבצע אחד מאלו שלוש פעמים ניתן לקבוע כי שני השחקנים ימשיכו בחזרה על העמדות ולבסוף יגיעו ל 50 מהלכים ללא שינוי.

סכימת תזוזות הכלים האפשריות

לכל אחד מכלי השחמט יש את סגנון התנועה האופייני לו. בסכימה זו מצוינים ב-X המהלכים האפשריים של כל כלי. המהלכים אפשריים רק בתנאי ששום כלי של אותו צד לא נמצא במקום אליו נעים. אם יש כלי של הצד היריב במיקום אליו הכלי יכול להגיע, השחקן יכול להכות את הכלי של היריב. גם הימצאותם של כלים במסלול של הכלי בתנועה יכול למנוע את התנועה, פרט לתנועה של פרש.



רקע תיאורטי בתחום הפרויקט

תוכנת שחמט היא תוכנה המסוגלת לשחק שחמט. כיוון ששחמט נחשב למשחק אסטרטגיה המביא לשיא את יכולתו של המוח האנושי, פיתוח תוכנת שחמט ברמה גבוהה, כזו המסוגלת לנצח כל שחקן אנושי, הוא אתגר רב-שנים. בעבר נחשב אתגר זה כדרך להתקדמות בפיתוחה של בינה מלאכותית, אך כעת אין הוא נחשב ככזה, משום שתוכנת שחמט מגיעה להישגים גבוהים בתחום ספציפי זה בלבד, ואין בה את היכולות הכלליות של המוח האנושי.

המאמר הראשון בנושא נכתב על ידי קלוד שאנון, פורסם בשנת 1950 לפני שאיש תכנת מחשב לשחק שחמט, וחזה בהצלחה את שתי האסטרטגיות בהן ייעשה שימוש שאותן כינה סוג א' וסוג ב'.

תוכנות סוג א' אמורות להשתמש בגישת כוח גס ולבחון כל עמדה אפשרית לעומק מוגדר של מסעים תוך שימוש באלגוריתם מינימקס. שאנון סבר שגישה זו לא מעשית משתי סיבות:

ישנם (בקירוב) בממוצע 30 מסעים חוקיים אפשריים בכל עמדת שחמט וחישוב לעומק 3 מסעים (6 חצאי מסעים) יניב 700 מיליון (306) עמדות שיש לבחון. גם בגישה האופטימית של ניתוח מיליון עמדות בשנייה, נדרש ל-16 דקות לניתוח לעומק של 3 מסעים.

החישוב הקודם התעלם ממסעים "שקטים" וניסה להעריך רק עמדות שבסיום החלפת כלים או לאחר סדרת מסעים משמעותית. שאנון הניח שהוספת מסעים שקטים, תגדיל מאוד את מספר העמדות שיהיה צורך לבחון ולכן תאט את התוכנית עוד יותר.

במקום לבזבז את עוצמת המחשב לניתוח מסעים רעים או טריוויאליים, הציע שאנון שתוכנות מסוג ב' ישתמשו בגישת אינטליגנציה מלאכותית כדי לפתור את הבעיה על ידי בחינת מסעים מועטים אך חזקים בכל עמדה. כך יתפנו לתוכנה משאבי זמן להעמיק בווריאנטיים המשמעותיים יותר.

אדריאן דה גרוט ראיין מספר שחמטאים מרמות שונות והסיק שהן אמנים והן מתחילים בוחנים 40 - 50 עמדות לפני שהם מחליטים איזה מסע לשחק. מה שהופך את האמנים לשחקנים חזקים בהרבה הוא בכך שהם עושים שימוש בכישורי זיהוי תבניות שנבנו מניסיונם האישי. כך הם יכולים לבחון הסתעפויות אחדות בעומק רב יותר ולהתעלם ממסעים שהם סבורים שהם חלשים.

הבעיה בתוכנות מסוג ב' נעוצה בכך שהתוכנה אמורה להחליט איזה מסע טוב מספיק כדי שיישקל בכל עמדה נתונה. התברר שזו בעיה קשה בהרבה לפתרון מאשר האצת תוכנות מסוג א' בעזרת חומרה משופרת.

בלִבה של תוכנת השחמט נמצאת פונקציית הערכת העמדה. פונקציה זו נותנת ערך מספרי לכל עמדה במשחק, המתקבלת על ידי ניתוח המסעים, על פי פרמטרים קבועים מראש כגון מספר הכלים, חוזקם היחסי ומיקומם. כך, מסוגלת התוכנה לבנות מעל עמדה נתונה עץ משחק (אשר ענפיו הם כל העמדות המתקבלות מכל המסעים החוקיים), ובהינתן עומקו לחשב את ה"עלה" או המסע האידיאלי, על פי אלגוריתם רקורסיבי, המסתייע בעיקרון פשוט של מציאת המקסימום מבין המינימום (שהרי יש לקחת בחשבון יריב שמשחק היטב). ככל שהחומרה שעליה פועלת תוכנת השחמט תהיה חזקה יותר כך יהיה באפשרותה של תוכנת השחמט לסרוק מספר רב יותר של מסעים (על כל האפשרויות הנגררות מהם) קדימה בפחות זמן.

הגדרת הבעיה האלגוריתמית

פיתוח משחק שחמט: נושא הפרויקט כמובן הוא פיתוח שחקן ממוחשב, אך לשם כך נצטרך ליצור תשתית יעילה בעת פיתוח משחק שחמט רגיל ששחקנים אנושיים ישתמשו בו, ולאחר מכן לנצל את תשתית זו לבניית שחקן ממוחשב.

מציאת כל המהלכים האפשריים: מציאת כל המהלכים האפשריים עבור שחקן- חלק זה ישמש אותנו גם עבור השחקנים האנושיים(נוכל לבדוק האם מהלך הוא חוקי אם הוא קיים ברשימת המהלכים האפשריים), וגם בבניית השחקן הממוחשב, כיוון שברקורסיה של אלגוריתם המינימקס נצטרך למצוא כל פעם את כל המהלכים החוקיים עבור שחקן.

חשוב מאוד שפונקציה זו תהיה יעילה ומהירה ככל הניתן, כיוון שאנו עוברים עליה מספר רב של פעמים באלגוריתם המינימקס, ואי יעילות של הפונקציה הזו תדרוש מאיתנו זמן ריצה יקר עבור כל מהלך של השחקן הממוחשב.

מימוש חוקי המשחק: מימוש החוקים של המשחק, הגבלת התזוזה של הכלים רק למהלכים חוקיים, מימוש מצבי שח, מט ופט, מימוש מצבי התיקו השונים ומעבר בין תורי השחקנים, ובעצם בניית המשחק.

מימוש פונקציית הערכת עמדה מדויקת ככל שניתן: מימוש פונקציית הערכת עמדה מספיק טובה, כך שתדע לתת הערכה מספרית טובה לעמדה במשחק נתון- ובאמצעות שילובה באלגוריתם המינימקס והאלפא-ביתא, תוכל לדחוף את השחקן הממוחשב לבצע מהלכים טובים ככל הניתן.

חשוב מאוד שהערכת העמדה תהיה מהירה ומדויקת ככל הניתן, הערכת עמדה לא יעילה תדרוש מאיתנו זמן יקר, שכן היא נקראת כל פעם במהלך ריצת הרקורסיה, ותאט משמעותית את הזמן שייקח לשחקן הממוחשב שלבצע מהלך בודד.

בניית השחקן הממוחשב: בניית שחקן ממוחשב שידע לשחק בצורה הגיונית ולשחק את המהלך האופטימלי עבורו, באמצעות פונקציית הערכת עמדה מהירה ומדוייקת, ומודל הכרעת המצבים, ומימוש עץ המשחק והאלגוריתמים השונים על מנת שיוכל לשחק בצורה הטובה ביותר והמהירה ביותר כנגד משתמש אנושי.

סקירת אלגוריתמים בתחום הבעיה

הכרעת מצבים

היכולת של המחשב להעריך את טיבן של עמדות שונות במשחק כלשהו, ובאמצעות כך להכריע בין מצבים שונים אליהם הוא יכול להגיע בתור מה המצב הטוב ביותר עבורו, ולבחור במהלך אשר יוביל אותו למצב זה, ובאמצעות כך להשיג את היכולת לשחק מהלכים לפי כמה הם מועילים לו.

הערכת עמדה(Evaluation)

היכולת של המחשב להעריך בצורה מספרית עמדת משחק, באמצעות חישוב מספר הכלים שיש לכל שחקן ומיקום כלי המשחק. מצב של שוויון בטיב עמדות השחקנים, יקבל את ההערכה 0, בעוד עבור כל השגת יתרון של אחד השחקנים תשתנה ההערכה(נהוג שיתרון לבן נע לכיוון + ויתרון שחור נע לכיוון -)באמצעות הערכת עמדה חזקה, יוכל המחשב להכריע בין מצבים ולבחור במצב בו הוא מקבל הערכת עמדה גבוהה יותר.

עץ משחק

עץ המשחק מתאר את כל אפשרויות התפתחות המשחק. "שורש" העץ הינו המצב הנוכחי, ממנו מתפצל ענף עבור כל צעד חוקי של המחשב. מכל ענף כזה מתפצלים ענפים עבור על התגובות החוקיות של היריב. ההתפצלות נפסקת במצבי סיום המסומנים ב"נצחון" (למחשב), "תיקו", או "הפסד".

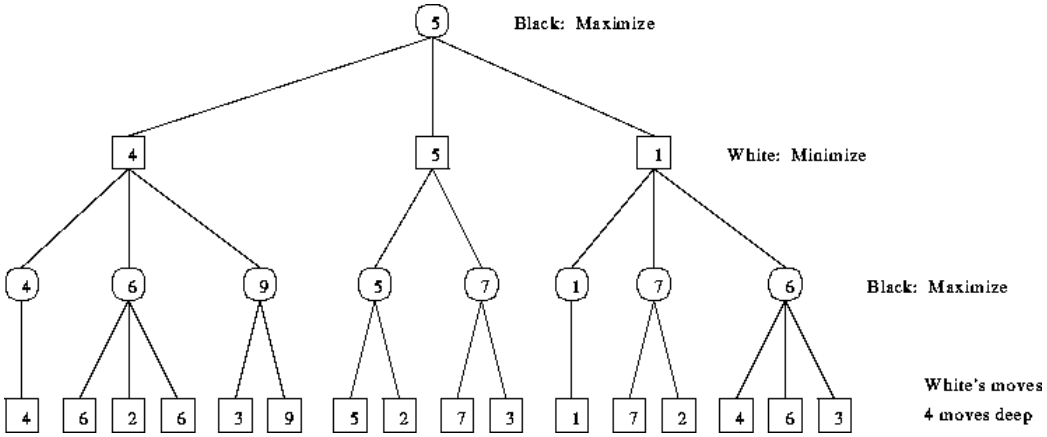
עץ מינימקס

עץ מינימקס הוא עץ (סוג של מבנה נתונים) הפורס את האפשרויות למשחק של שחקן א', את התגובות של שחקן ב' לכל פעולה של שחקן א', את תגובותיו של א' לתגובותיו של ב' וכך הלאה.

העלים בעץ שנוצר הם מצבים סטטיים שנגיע אליהם לאחר רצף של מהלכים (הנתיבים בעץ הם למעשה תרחישים אפשריים). ניתן ציון לכל מצב סטטי שכזה (מצב סטטי בלוח שחמט לדוגמה), שישקף כמה המצב טוב מבחינתנו.

נסרוק את העץ החל בעלים, דרך הקודקודים (קודקוד הוא צומת פנימי בעץ) שמעליהם עד השורש (מלמטה למעלה) הציון שניתן לכל קודקוד הוא הערך הגבוה ביותר של העלים/הקודקודים שתחתיו, אם אותו קודקוד מסמל מהלך שלי, והערך הנמוך ביותר של העלים/קודקודים שתחתיו אם אותו קודקוד מסמל מהלך של היריב (כי ברור שהוא יבחר באפשרות הטובה ביותר בשבילו - הכי גרועה בשבילי).

כשנגיע לשורש, נבחר בקודקוד שמתחת לשורש עם הציון הטוב ביותר.



אלפא-ביתא

גיזום אלפא-ביתא היא שיטת אופטימיזציה עבור עצי-חיפוש מסוג מינימקס. מטרת השיטה לצמצם את מספר תתי העצים עליהם יש להלך בעת הערכת מהלך אפשרי בעץ מינימקס ובכך לקצר משמעותית את זמן אלגוריתם המינימקס. באופן פרקטי, מספק האלגוריתם את התוצאה המקורית של מינימקס, בזמן קצר בהרבה. הגיזום פועל בצורה שבה במהלך החיפוש לעומק בעץ המינימקס ניתן לזנוח פתרונות חלקיים ברגע שברור שהם גרועים מפתרונות שכבר ראינו.

search Quiescence – על מנת להימנע effect horizon them נשתמש באלגוריתם הנתון.
אלגוריתם זה למעשה חוקר עלים בעץ שלנו שנחשבים "לא יציבים" – כלומר מהלכים שיכולים ליצור מצב בו מצבנו יהיה רע יותר (במקרה שלנו בעיקר שח ואכילה) מכיוון ואנו רוצים להימנע ממהלכים שיובילו להפסד במהלכים עתידיים, נחקור בעומק מסוים נוסף מהלכים מסוכנים, דבר זה מאפשר לנו להבין נכון יותר את הציון שיש למהלך מסוים בעלה מסוים.

Move Ordering– מכיוון וגיזום אלפא-ביתא פועל טוב יותר כאשר המהלכים הטובים ביותר נבדקים קודם יש יתרון רב בהשקעת זמן בסידור המהלכים לפני בדיקתם. ישנו איזון עדין בין כמות הזמן שמושקעת בסידור המהלכים לבין כמות הזמן שנגזמת כתוצאה מכך. במשחק שלנו נמיין את המהלכים בצורה הבאה, כך שמהלכי הכתרה של חייל יופיעו ראשונים, לאחר מכן מהלכי אכילה, אחר כך מהלכי הצרחה, ולבסוף נמיין את שאר המהלכים על פי הכלי המבצע אותם, כך שמהלכים של כלים שערכם גבוה יותר יופיעו קודם. עוד אציין כי סידור המהלכים תרם לשיפור זמן הריצה של התוכנית ברוב המצבים במשחק.

"אחת משיטות האופטימיזציה של תוכנות שונות העושות שימוש בחיפוש א"ב, היא יצירת מנגנון בחירת מהלכים חכם. מנגנון זה מנסה לחזות מה הם המהלכים ה"מעניינים" על לוח המשחק, ולבחון אותם ראשונים. זוהי היוריסטיקה שלא מובטח כי תסייע כלל, אך לעיתים רבות היא מסייעת בגיזום מהיר של העץ. כך למשל, תוכנת שח תנסה לבחון תחילה מהלכים של הכאת חייל יריב, או מהלכים המציבים את המלך היריב בשח. מהלכים אלו נחשבים "מעניינים" בעיני התכנה, וייתכן כי הם מגלמים בחובם פוטנציאל לערך מיטבי בעיני הצד המחפש. אם כן, הרי שהדבר יסייע מאוד בהליך הגיזום."-מתוך ויקיפדיה

אסטרטגיה

1. הערכת מצב משחק

על מנת להעריך נכונה מצב משחק החלטתי לדגול במספר שיטות שונות להערכת עמדה:

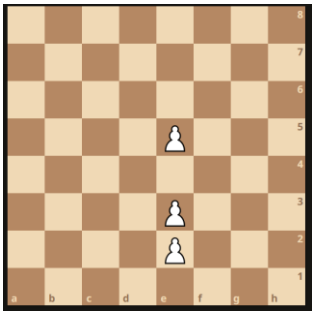
- חומר: הערכה בסיסית של החומר-כמה כלם יש לו ומה סוגם-של כל שחקן, חישוב ערכם של החיילים שנותרו לו והערכת חומרו על פי כך, כאשר ערכה של מלכה הוא 9, ערכו של צריח הוא 5, ערכו של פרש 3.2 ורץ הוא 3.3, וערכו של חייל הוא 1.
  ערכים אלו נבחרו כדי לקיים מספר תוצאות:
  1. פרש/רץ שווים יותר משלושה חיילים
  2. רץ שווה יותר מפרש
  3. 2 כלים מינוריים(רץ/פרש) שווים יותר מצריח וחייל
  4. מלכה וחייל שווים לשני צריחים
  5. כלי מינורי ושני חיילים שווים יותר מצריח
  6. צריח, כלי מינורי וחייל שווים יותר ממלכה

- ניידות: באמצעות מספר המהלכים האפשריים לכל שחקן, והערכת כל מהלך לפי הכלי המבצע אותו ומיקומו הסופי נגד מיקומו ההתחלתי, נוכל להעריך את הניידות של שחקן- כמה הכלים שלו מפותחים וניידים.
  כלים ניידים ומפותחים מאוד חשובים בשחמט- שכן כלים אלו מאיימים איומים רבים יותר ויכולים להוות פתח למתקפות שונות ורבות יותר כנגד היריב.

- מבנה חיילים(PAWNS): באמצעות מערך אשר ימספר את מספר החיילים בכל עמודה בלוח, נוכל למצוא חיילים אשר יוצרים סוללת חיילים(חיילים כפולים למשל) חיילים מבודדים, ואיי חיילים, ו"להעניש" שחקן על כך.

  סוללת חיילים-מצב בו מספר חיילים של אותו שחקן נמצאים על אותו הטור על הלוח- מצב זה נחשב לרע יחסית, וככל שיש יותר חיילים בסוללה נחשב רע עוד יותר, כיוון שחיילים הנמצאים על אותו הטור חוסמים זה את זה ונחשבים פגיעים
  חיילים מבודדים-חיילים מבודדים הם חיילים אשר משני הטורים שלצידם אין חיילים, ומצב זה נחשב רע, כיוון שהחייל המבודד נחשב לפגיע מאוד כיוון שאין חייל השומר עליו.
  איי חיילים-אי חיילים הוא קבוצה של חיילים אשר נמצאים על טורים סמוכים. ככל שלשחקן ישנם יותר איי חיילים, עמדתו נחשבת לרעה יותר שכן כמות רבה של איים הופכת את חיילי השחקן לפגיעים.

1. מצב של סוללת חיילים הכוללת שלושה חיילים



2.מצב של חייל מבודד-החייל המסומן בירוק מבודד כיוון שאין חיילים בטורים הסמוכים

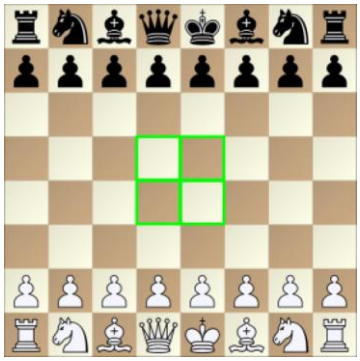
3.ללבן יש שני איי חיילים, בעוד לשחור שלושה

- בטיחות מלך: באמצעות בדיקה של איומים על המלך ובקרבתו, ומצבו של המלך, נוכל לתת בונוס עבור מלך בטוח יותר, שכן מלך בטוח יותר סביר פחות שיקבל מט.
בנוסף, נבדוק עבור מלך את "מגן החיילים" שלו-מגן החיילים מתארים את מבנה החיילים המגנים על המלך-ישנם מספר מבנים אידיאלים עבור מגן החיילים, וניתן בונוס לשחקן אשר למלכו מגן חיילים טוב.

4.אחד המגני חיילים האידאלים למלך הוא שלושה חיילים המסודרים מפניו, כמו המלך הלבן או המלך השחור

- שליטה על המרכז: באמצעות מציאת האיומים על מרכז הלוח(בעיקר ארבעת המשבצות האמצעיות), וספירת מספר החיילים אשר על משבצות המרכז נוכל להעריך את השליטה של כל שחקן על המרכז, דבר אשר נחשב חשוב מאוד בשלב הפתיחה.

5.השליטה על המרכז חשובה מאוד, ובפתיחה ירצה כל שחקן כמה שיותר כלים המאיימים על המרכז



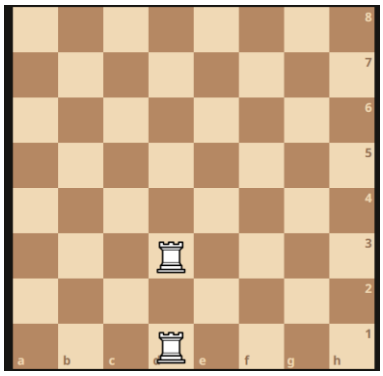
- מבנה צריחים: באמצעות מציאת מיקומם של הצריחים, נוכל לתגמל שחקן עבור עמדת צריחים טובה, לדוגמה צריחים מחוברים על שורה, או על טור.

חיבור בטור-צריחים הנמצאים באותו הטור ואין ביניהם כלים אשר חוסמים אותם. מבנה זה נחשב לחזק שכן הצריחים שומרים אחד השני מפני איומים, וניידים מאוד מבחינת התקפותיהם על היריב.

חיבור בשורה-צריחים הנמצאים באותה השורה ואין ביניהם כלים אשר חוסמים אותם. מבנה זה נחשב גם הוא לחזק אך פחות ביחס לחיבור בטור, שכן הצריחים שומרים זה על זה, אך במקרה זה לרוב ניידותם מוגבלת.

6.צריחי השחקן הלבן מחוברים ביניהם בשורה, כמו גם צריחי השחור

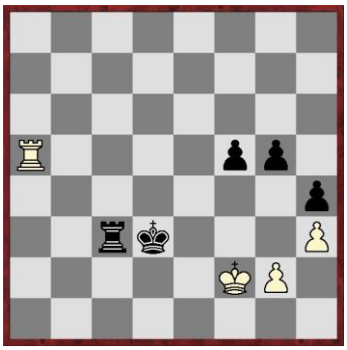
7.הצריחים הלבנים מחוברים ביניהם בטור



- שלב המשחק: באמצעות מציאת שלב המשחק(פתיחה, אמצע או סיום) על ידי חישוב החומר אשר נשאר על הלוח, נוכל לתת הערכה רלוונטית לכל אחת מהשיטות.
לדוגמא: ניתן לשליטה על המרכז ערך רק במצב הפתיחה, שכן היא לא רלוונטית למצב אמצע משחק וסיומו.

8.מצב פתיחה אפשרי

9.מצב סיום אפשרי



- מיקום החיילים: באמצעות Piece-square-table עליהם הוסבר קודם נוכל להעריך את טיבם של מיקומי הכלים של שחקן, ולתת ערך למיקומי החיילים.
PST מתבססות על מיקומים שידוע שבהם עמדתו של שחקן טובה יותר או פחות.
ניתן לקרוא עוד גם כאן:
https://www.chessprogramming.org/Piece-Square_Tables

2. יצירת מהלכים אפשריים ובדיקות ניצחון

על מנת לייצר את כל המהלכים האפשריים עבור שחקן מסוים בזמן מינימאלי ככל שאפשר החלטתי להשתמש ברשימה של כל הכלים הפעילים של שחקן, אשר עבור כל כלי נחשב את המיקומים האפשריים לו.

בנוסף, נבדוק עבור כל מהלך האם הוא מוביל לניצחון או לתיקו.

3. הערכת עץ המשחק והמהלך הטוב היותר

על מנת להעריך נכונה את עץ המשחק שלנו החלטתי ליישם את כל האלגוריתמים שציינתי למעלה, כאשר ניתנת הערכה עבור כל מהלך אפשרי בעמדה הנוכחית, ומוחזר מאלגוריתם המינימקס AB המהלך החזק ביותר של השחקן.

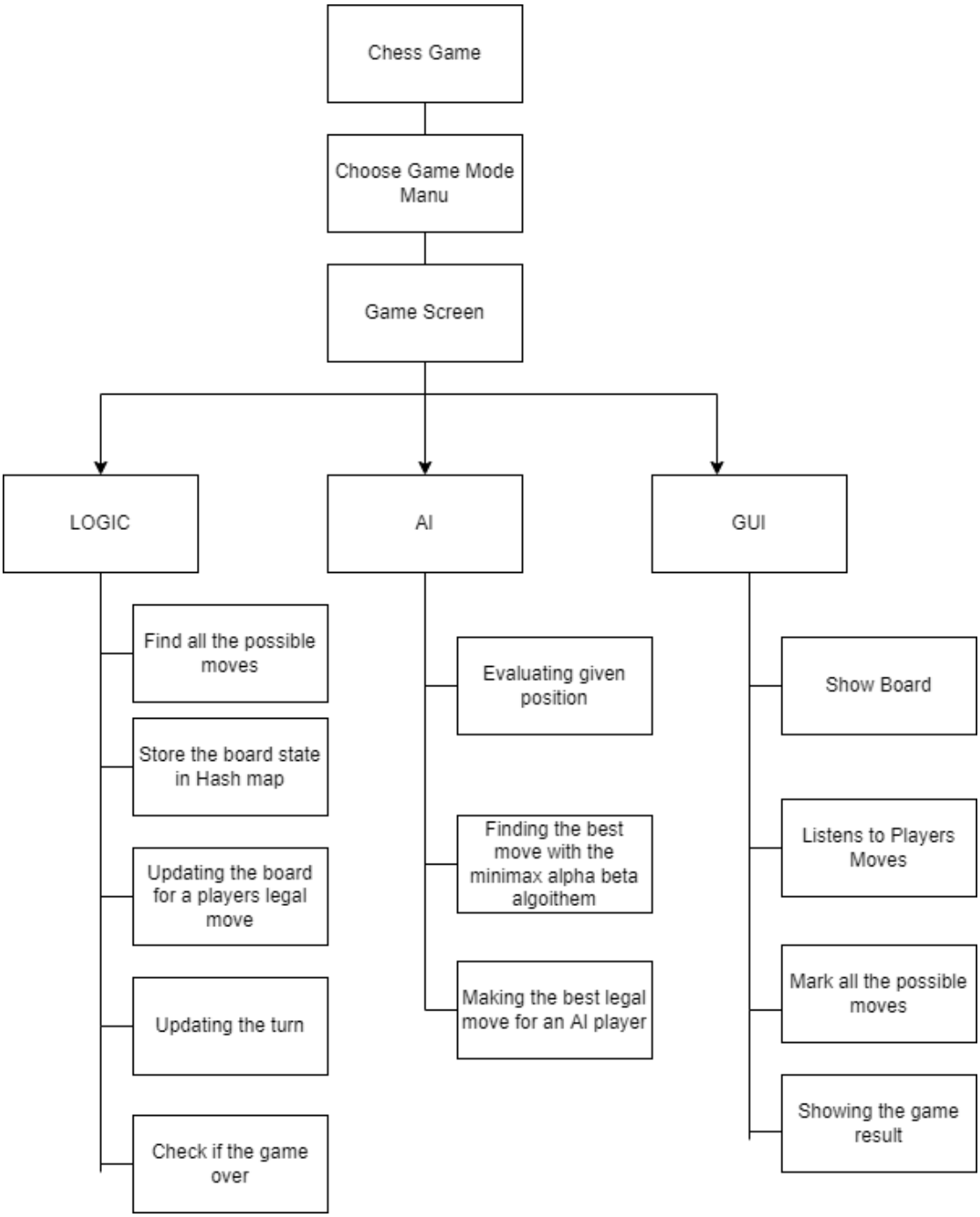
בנוסף, נעשה Move Ordering לכל המהלכים האפשריים, על מנת לסדר קודם מהלכים אשר סיכוייהם להיות טובים יחסית ולבצע גיזום גבוהים, בתקווה לקצר את זמן חשיבת המחשב. המהלכים שיופיעו ראשונים ברשימת המהלכים יהיו בעיקר מהלכי אכילה ושחים.

בנוסף נוסיף עומק חישוב למהלכים לא יציבים, כמו אכילות, באמצעות חיפוש ה quiescence, אשר יאפשר לנו להתגבר על Horizon effect.

בצורה זו נוכל למצוא את המהלך הטוב ביותר, ולבצע אותו עבור המחשב במצב של משחק נגד AI.

Top-down level design

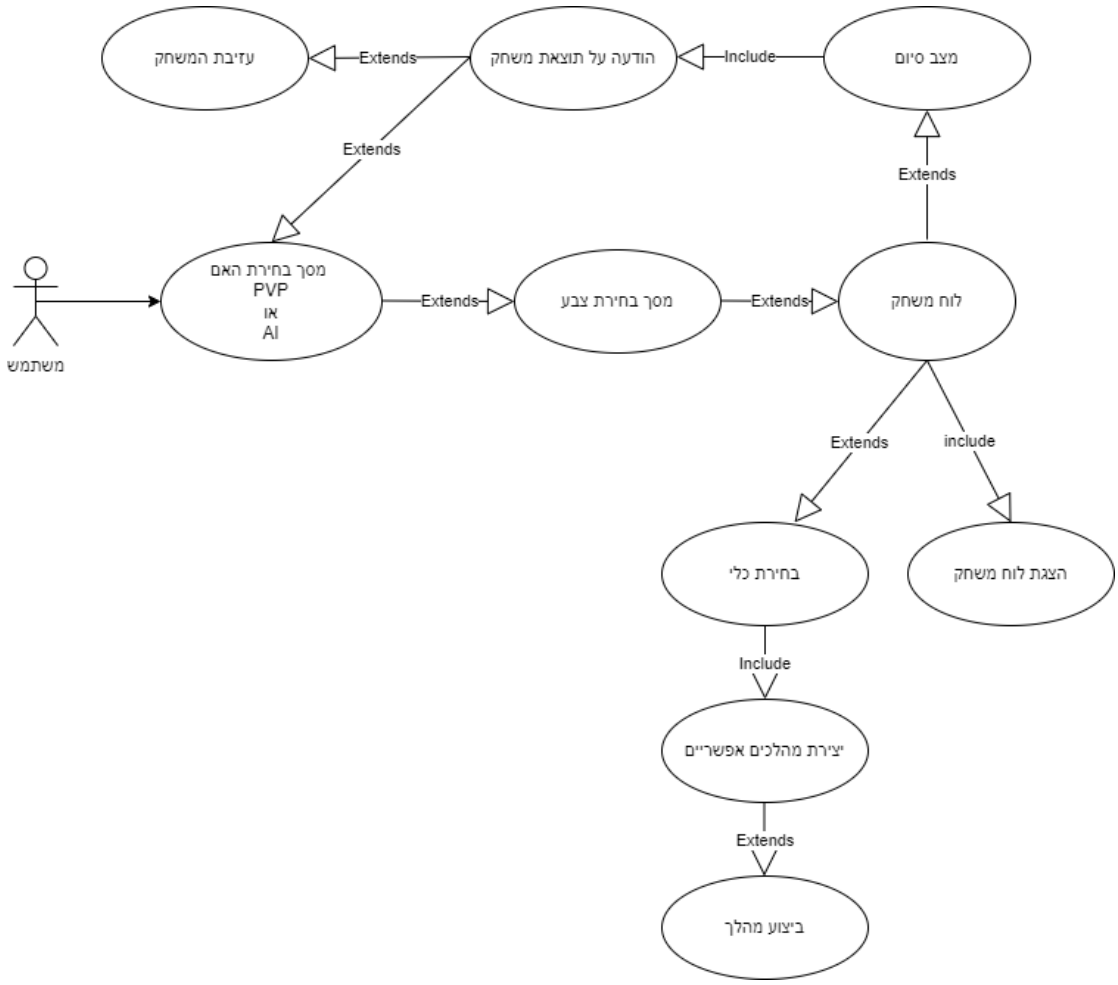
תרשים



תיאור טקסטואלי

1. משחק שחמט.
1.1. תפריט בחירת מצב משחק: בחירה האם לשחק שחקן מול שחקן או מול מחשב ובאיזה צבע.
1.2. Game: ממשק המשתמש במשחק עצמו.
1.2.1. Logic: אחראית על מימוש חוקי המשחק ועדכונו עבור כל מהלך.
1.2.1.1. מציאת כל המהלכים האפשריים: מציאת כל המהלכים האפשריים במצב.
1.2.1.2. אחסון מצב המשחק במבני הנתונים HashMap.
1.2.1.3. עדכון המשחק ומצב הלוח עבור כל מהלך של שחקן.
1.2.1.4. עדכון תור.
1.2.1.5. בדיקה האם הגענו לסיום המשחק.
1.2.2. AI: אחראית על מימוש השחקן הממוחשב.
1.2.2.1. הערכת עמדה נתונה.
1.2.2.2. מציאת המהלך הטוב ביותר מבין המהלכים החוקיים באמצעות אלגוריתם המינימקס והאלפא ביתא-ועל ידי הערכת עמדות.
1.2.2.3. ביצוע מהלך עבור שחקן ממוחשב כאשר נבחר באופציה של משחק נגד שחקן ממוחשב.
1.2.3. GUI: ממשק המשתמש הגרפי: אחראי על הצגת הלוח וקבלת מהלכים מהעכבר.
1.2.3.1. הצגת הלוח והכלים בו.
1.2.3.2. הקשבה למהלכי השחקנים.
1.2.3.3. סימון המהלכים האפשריים עבור שחקן בעת בחירת כלי על הלוח.
1.2.3.4. הצגת תוצאת המשחק בסיומו.

Use Case Diagram



מבני נתונים

מבני הנתונים בהם בחרתי להשתמש בפרויקט הם:

Hash Map- מפה, אשר באמצעותה אשמור מפתח וערך, כאשר המפתח הוא מספר בין 0 ל 63 אשר ייצג אחת מ 64 המשבצות על לוח השחמט, והערך הוא עצם מסוג Piece המייצג את הכלי במשבצת(או לא).

בצורה זו אוכל ביעילות של o(1) להעריך מצבים על הלוח, לשלוט בהיכן כל כלי נמצא על הלוח, ולבדוק האם מהלכים המשוחקים חוקיים.

מיקומי המשבצות ייוצגו בצורה הבאה:

```
00 01 02 03 04 05 06 07
08 09 10 11 12 13 14 15
16 17 18 19 20 21 22 23
24 25 26 27 28 29 30 31
32 33 34 35 36 37 38 39
40 41 42 43 44 45 46 47
48 49 50 51 52 53 54 55
56 57 58 59 60 61 62 63
```

רשימה- רשימות אשר ישמשו אותי לשמירת נתונים שונים במהלך המשחק, בגון רשימה של עצמים מסוג Move של המהלכים האפשריים של שחקן, רשימה של עצמים מסוג Piece לשמירה על רשימת החיילים הפעילים של שחקן, ועוד...

מערך מונים-נשתמש במערך מונים על מנת לשמור את מספר החיילים בכל עמודה על הלוח, בצורה זו, נוכל לנתח את מבנה החיילים של שחקן, ולנתח חיילים כפולים, חיילים מבודדים, ואיי חיילים.

מערכי Piece-Square-Table, על מנת לתת ערך למיקום של כלי, נשמור עבור כל כלי מערך בו עם ערכים מספריים על כמה טוב מיקום שלו על הלוח. המספר שנצטרך להוסיף לערך המיקום הוא בעצם הערך שבמיקום של מיקום השחקן.

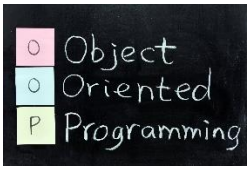
```
private final static double[] WHITE_KNIGHT_PREFERRED_COORDINATES = {
        -0.5,-0.4,-0.3,-0.3,-0.3,-0.3,-0.4,-0.5,
        -0.4,-0.2,  0,  0,  0,  0,-0.2,-0.4,
        -0.3,  0, 0.1, 0.15, 0.15, 0.1,  0,-0.3,
        -0.3,  0.05, 0.15, 0.2, 0.2, 0.15,  0.05,-0.3,
        -0.3,  0, 0.15, 0.2, 0.2, 0.15,  0,-0.3,
        -0.3,  0.05, 0.1, 0.15, 0.15, 0.1,  0.05,-0.3,
        -0.4,-0.2,  0,  0.05,  0.05,  0,-0.2,-0.4,
        -0.5,-0.4,-0.3,-0.3,-0.3,-0.3,-0.4,-0.5
};
```

כך למשל יראה Piece-Square-Table של פרש שחור, לכן, לדוגמא עבור פרש שחור הנמצא במשבצת מספר 63, נוסיף להערכת העמדה **-0.5**

תיאור הטכנולוגיה

אפליקציית DESKTOP

תכנות מונחה עצמים(OOP)



שפת תכנות

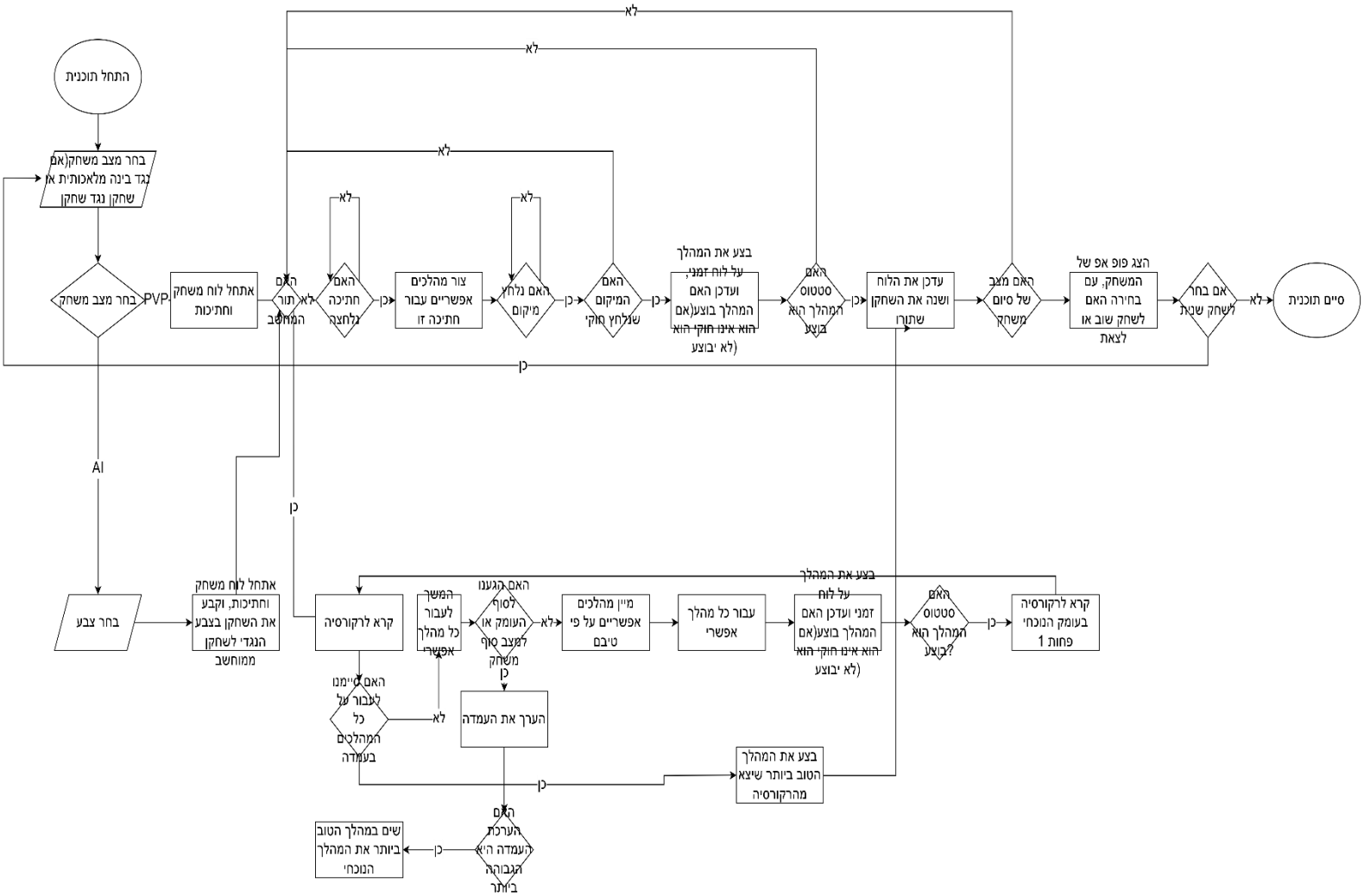
JAVA

סביבת עבודה:

Intellij

IntelliJ IDEA Community Edition 2021.3.1

אלגוריתם ראשי



ממשקים

- JAVA SWING- ג'אווה סוינג הוא ממשק ליצירת ממשק משתמש גרפי(GUI) באפליקציות ג'אווה.
במהלך הפרויקט, השתמשתי בסוינג עבור כל החלק הגרפי בפרויקט.

- Comparator Interface- ממשק המשמש להשוואה בין שני עצמים מאותה מחלקה, באמצעות ממשק זה נוכל למיין את המהלכים האפשריים, כך שמהלכים בעלי סבירות יותר גבוה להיות טובים יופיעו קודם ברשימה.

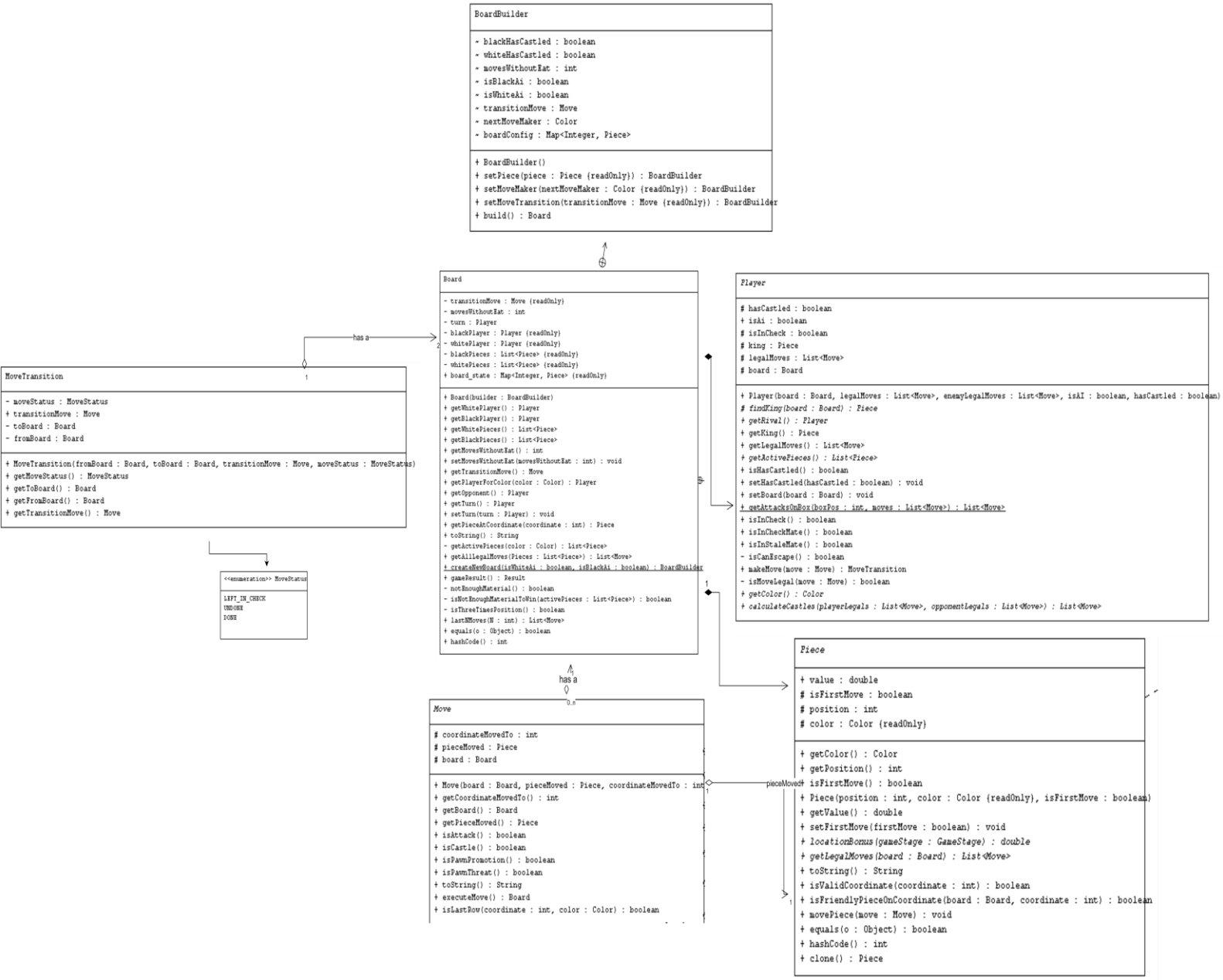
- Package com.google.common.collect- ממשק של גוגל עבור שפת ג'אווה, אשר סיפק לי מספר כלים נוחים, בעיקר עבור מיון המהלכים, כמו התחלת רצף השוואות עבור ההשוואות בין המהלכים, וסידור המהלכים לרשימה ממוינת.

- java.util - חבילת הכוללת בתוכה מימושים שונים למבני נתונים בשפת ג'אווה.
באמצעות חבילה זו אוכל להשתמש במבני נותנים ופעולות עליהם מוכנים מראש, כמו סוגי הרשימות השונות, HashMap וכדומה.

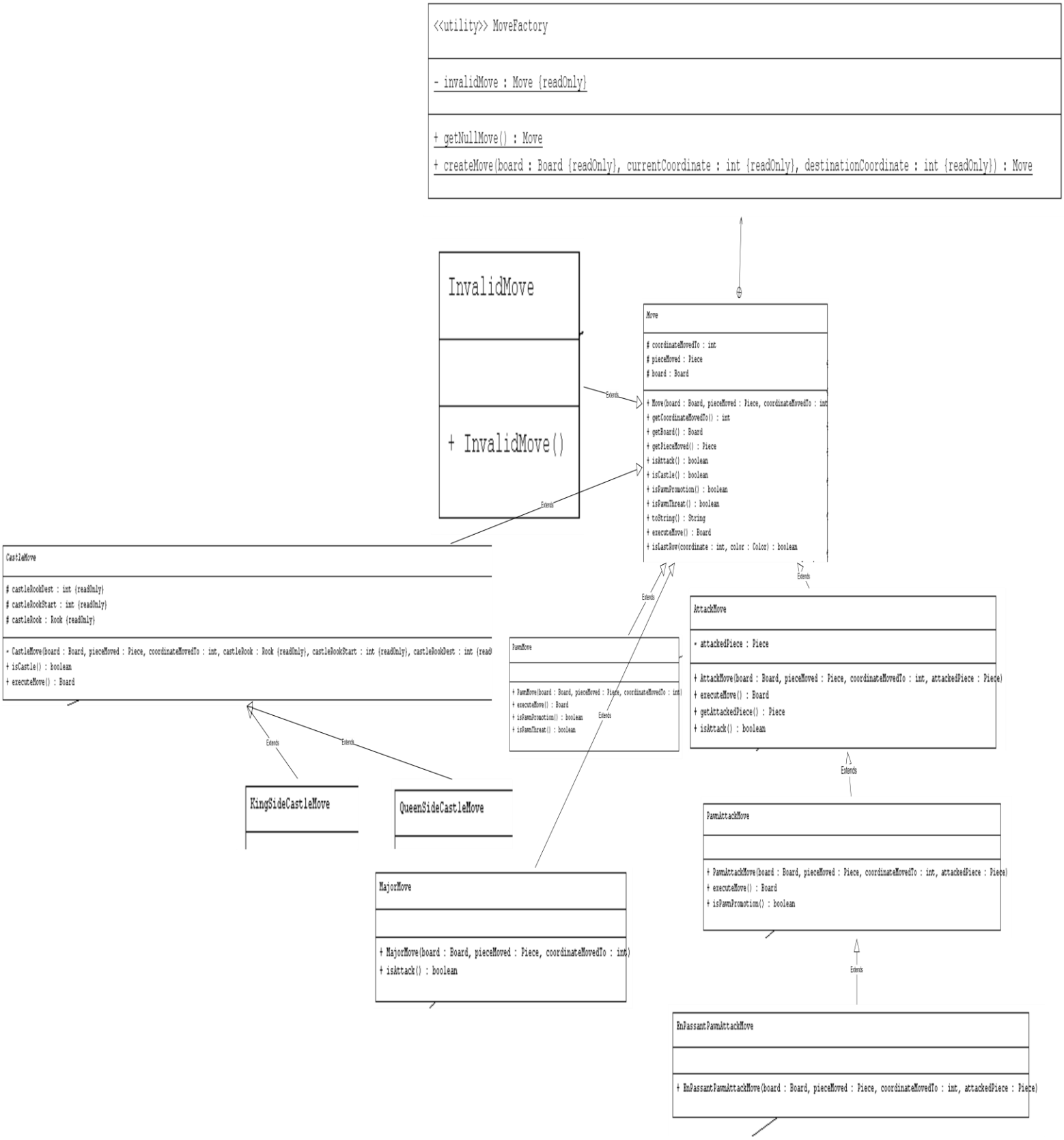
UML Class Diagram

***התרשים מחולק למספר תרשימים, כדי להימנע מתרשים גדול מדי או בלתי קריא ***

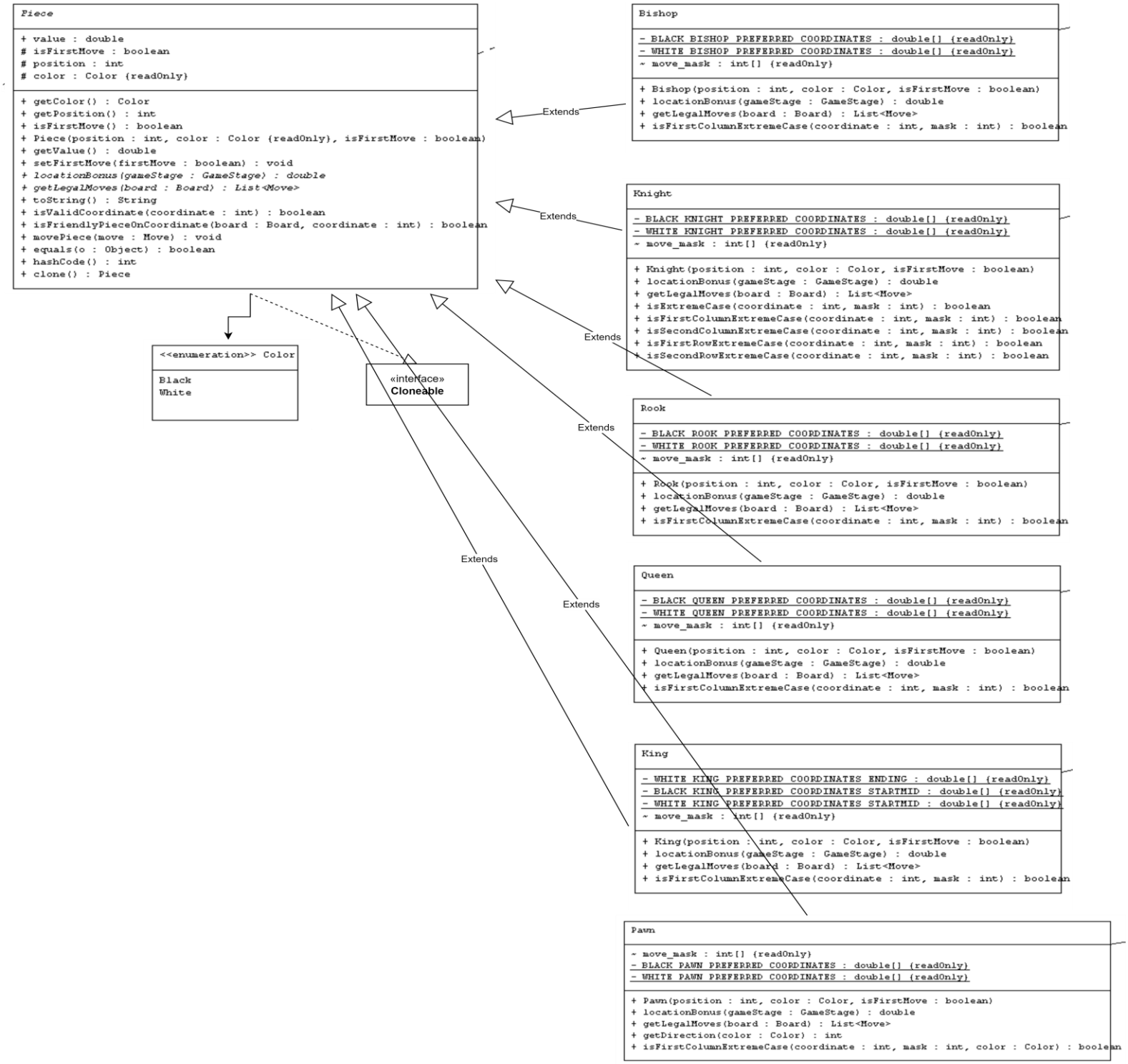
המחלקות הראשיות



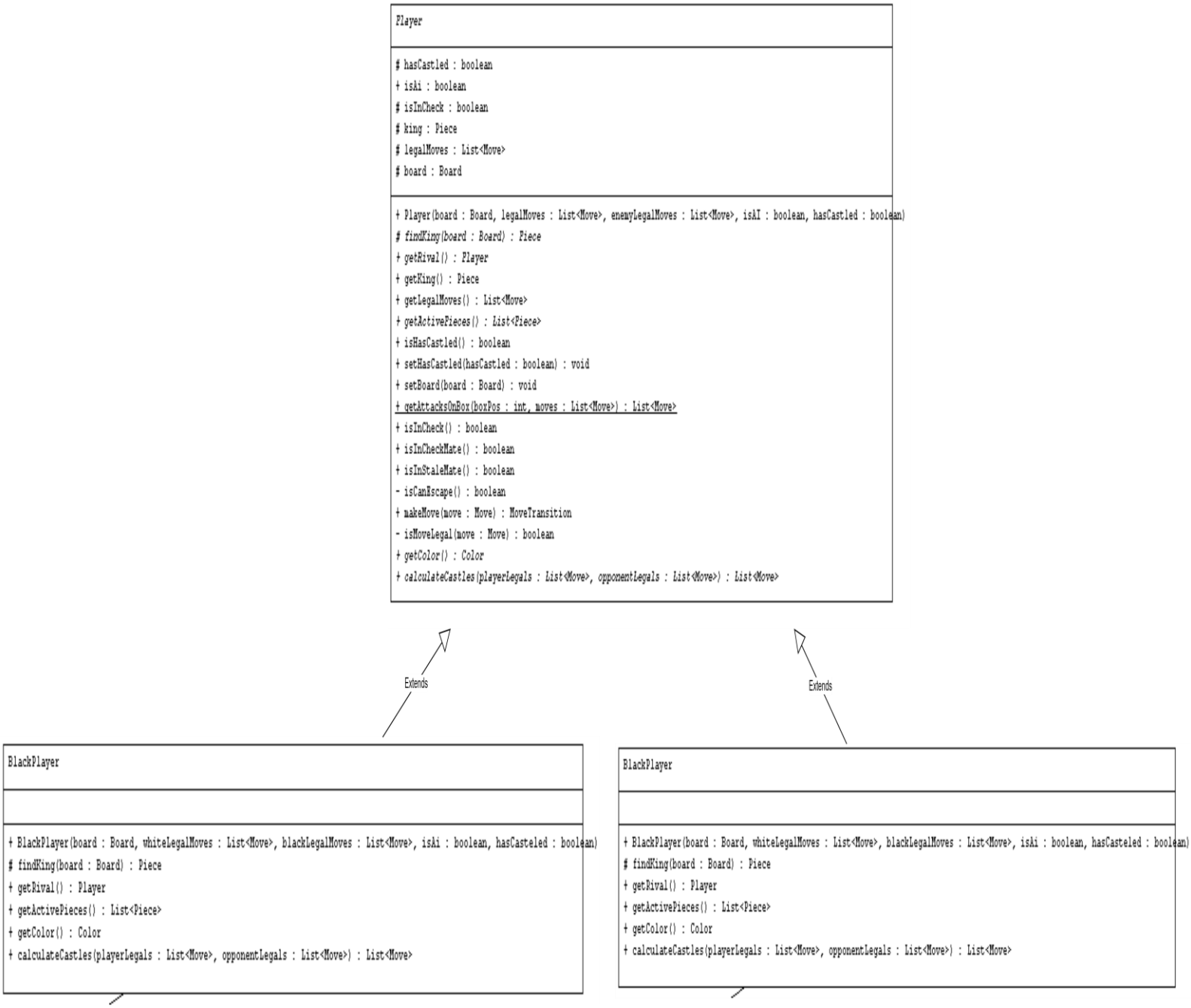
מחלקות ה Move השונות



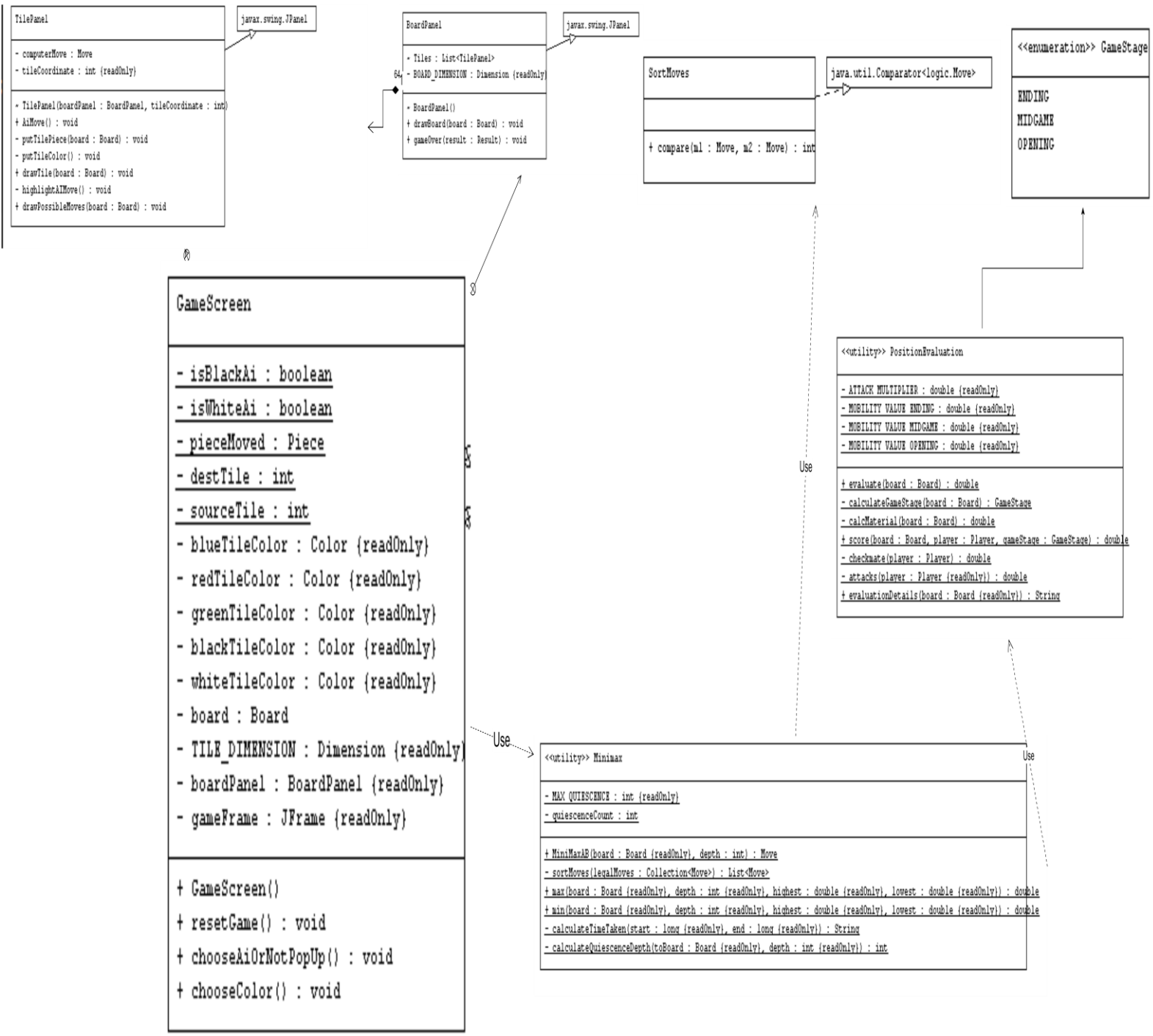
מחלקות ה Piece השונות



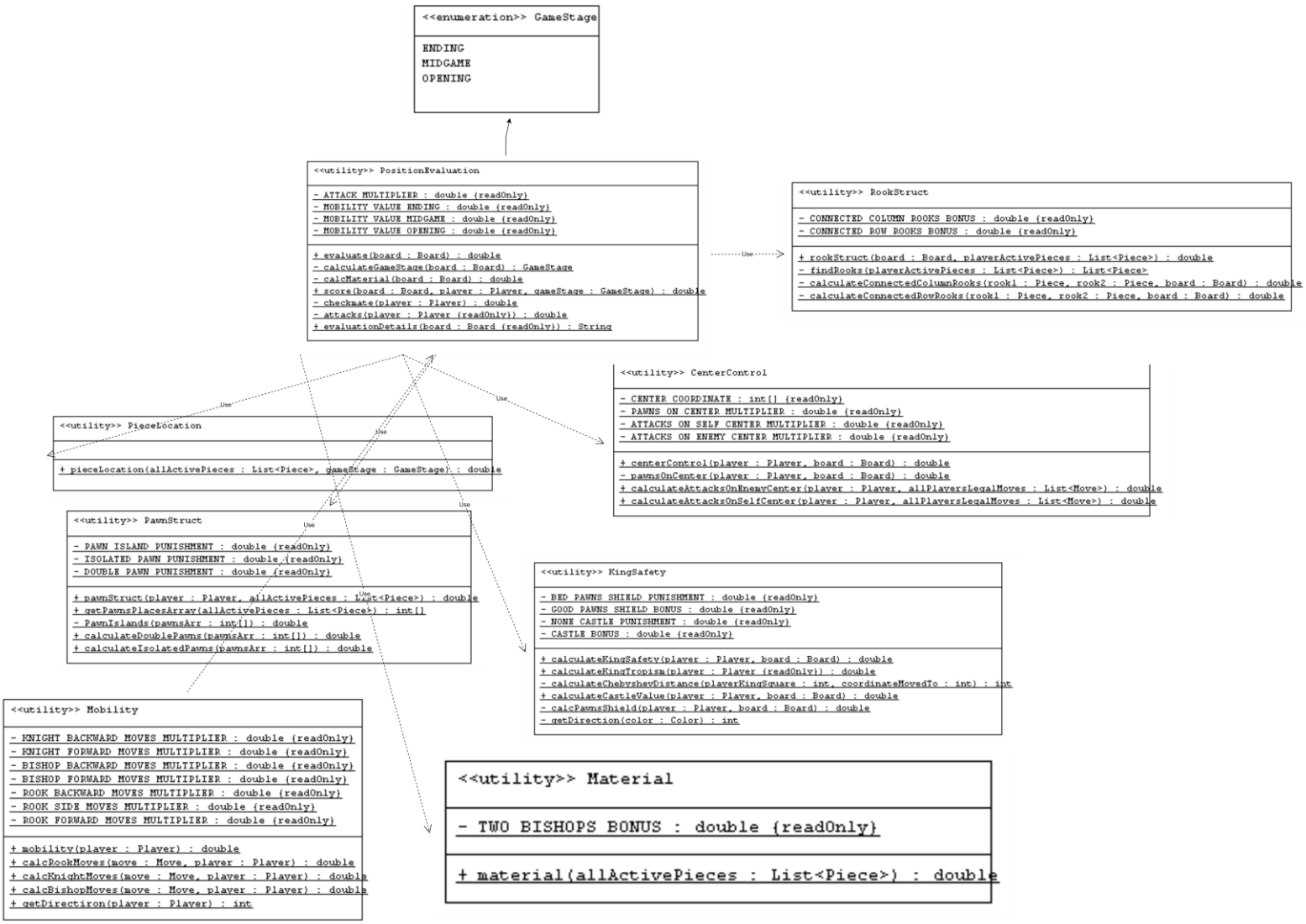
מחלקות ה Player השונות



מחלקת ה Gui וזיקתה לבינה



מחלקות הערכת העמדה שונות



הפונקציות הראשיות בפרויקט / תיאור המחלקות הראשיות בפרויקט

1. שם מחלקה: Board

תיאור מחלקה: מחלקה ראשית אשר מייצגת לוח שחמט ותכונותיו, ואת עצמי השחקנים המשחקים.
המחלקה אחראית על מציאת הכלים הפעילים, מציאת המהלכים האפשריים ועל אחסון של מצב לוח נוכחי באמצעות תכונה של HashMap המייצג את הלוח.

פונקציה	תיאור הפונקציה
`public Piece getPieceAtCoordinate(int coordinate)`	פונקציה אשר מקבלת ערך של מיקום בלוח, ומחזירה את הכלי הנמצא במיקום זה. יעילות: O(1)
`private List<Piece> getActivePieces(Map<Integer, Piece> board_state, Color color)`	פונקציה המקבלת HashMap של מצב לוח וצבע, ומחזירה את כל רשימה של כל הכלים הפעילים של השחקן בצבע הזה ב HashMap הנתון:O(1) , כיוון שמספר הערכים המקסימלי ב HashMap הינו 16.
`public List<Move> getAllLegalMoves(List<Piece> Pieces)`	פונקציה המקבלת רשימה של כלים, ומחזירה רשימה של כל המהלכים החוקיים עבור כלים אלו. יעילות: O(n) כאשר n הוא מספר המהלכים האפשריים.
`public static BoardBuilder createNewBoard(boolean isWhiteAi, boolean isBlackAi)`	פונקציה המקבלת ערכים בוליאניים של האם כל שחקן הוא AI, ובונה בנאי עבור לוח חדש-במצבו ההתחלתי. בנאי הלוח משמש לשמירת ערכים עבור לוח חדש, והוא הפרמטר היחיד של פונקציית הבנאי של הלוח. יעילות: O(1)
`public Result gameResult()`	פעולה המחזירה Enum של תוצאת המשחק-לבן, שחור, תיקו או משחק לא נגמר. יעילות: O(1)

```
public List<Move> getAllLegalMoves(List<Piece> Pieces)
```

תיאור פסאודו קוד של הפונקציה:

1. הגדרת רשימת מהלכים חדשה
2. עבור כל כלי:
   a. קרא לפונקציה למציאת כל מהלכי הכלי
   b. הוסף את כל מהלכי הכלי לרשימת המהלכים
3. החזר את רשימת המהלכים

2. שם מחלקה:Player

תיאור מחלקה: מחלקה אבסטרקטית אשר מייצגת שחקן על הלוח, וממנה יורשות שתי מחלקות WhitePlayer, ו BlackPlayer.

המחלקה תשמור בתוגה את לוח המשחק הנוכחי של השחקן, המהלכים האפשריים לו, ותכונות נוספות החשובות לשחקן כמו האם הוא הצריח, האם הוא Ai או לא, מיקום המלך שלו, ועוד...

המחלקה אחראית על בדיקות של מצבים כמו האם שחקן בשח או במט וכדומה, מציאת כלים פעילים של השחקן ומהלכיו האפשריים ועל ביצוע מהלך עבור שחקן.

`protected abstract Piece findKing(Board board);`	פעולה אשר במימושה במחלקות היורשות מקבלת לוח, ומחזירה את העצם של מלך השחקן. יעילות: O(1)
`public static List<Move> getAttacksOnBox(int boxPos, List<Move> moves)`	פעולה אשר מקבלת מיקום של משבצת ורשימה של מהלכים, ומחזירה רשימה של מהלכים התוקפים את המשבצת הנתונה. יעילות: O(n) כאשר n הוא מספר המהלכים האפשריים.
`public MoveTransition makeMove(Move move)`	פעולה אשר מקבלת עצם מהלך ומחזירה את עצם מעבר-מהלך של ביצוע המהלך-היא מבצעת את המהלך אם הוא חוקי ושמה את הלוח החדש בלוח היעד, אחרת היא מחזירה מעבר-מהלך שבו לוח היעד ולוח המקור זהים ושמה בעצם המעבר-מהלך את סטטוס המהלך-האם בוצע, האם אינו חוקי או האם משאיר מלך בשח. יעילות : O(n) כאשר n מספר המהלכים האפשריים של היריב
`public abstract List<Move> calculateCastles(List<Move> playerLegals, List<Move> opponentLegals)`	פעולה אשר במימושה במחלקות היורשות מקבל רשימה של מהלכים אפשריים של השחקן וכזו של מהלכים אפשריים של יריבו, ומחזירה רשימה של מהלכי ההצרחה האפשריים לשחקן. יעילות: O(n) כאשר n מספר המהלכים האפשריים של היריב
`public boolean isInCheck(){ return isInCheck; }`	Getter של התכונה המאותחלת בפעולה הבונה, מחזירה אמת אם השחקן בשח ושקר אם לא. יעילות: מכיוון ש getter היא O(1), אך חשוב לציין שאתחול התכונה isInCheck יעילותה O(n)
`private boolean isCanEscape()`	פעולה אשר בודקת האם לשחקן יש מהלכים חוקיים לשחק ומחזירה אמת אם כן ושקר אם לא, ומשמשת לבדיקת מט ופט. יעילות: O(n)

	כאשר n מספר המהלכים האפשריים של השחקן.
`public boolean isInCheckMate()`	פעולה אשר מחזירה האם השחקן נמצא בשח-מט, אמת אם כן ושקר אם לא. יעילות O(n)
`public boolean isInStaleMate()`	פעולה אשר מחזירה האם השחקן נמצא בפט, אמת אם כן ושקר אם לא. יעילות- O(n)

```
public MoveTransition makeMove(Move move)
```

תיאור פסאודו קוד של הפונקציה:

1. קרא לפונקציה הבודקת האם המהלך חוקי עבור המהלך
   a. אם לא חוקי:
      i. החזר עצם מעבר מהלך עם הלוח הנוכחי בלוח היעד ובלוח המקור, המהלך, וסטטוס לא בוצע
   b. אחרת:
      i. קרא לפעולת ביצוע המהלך ושים את החזרתה בעצם לוח
      ii. קרא לפעולה המוצאת התקפות על משבצת עבור המלך שלנו, ומהלכי היריב, ושים את החזרתה ברשימת מהלכים.
      iii. אם רשימת המהלכים ריקה:
         1. החזר עצם מעבר מהלך עם הלוח הנוכחי בלוח היעד ובלוח המקור, המהלך, וסטטוס נשאר בשח
      iv. אחרת:
         1. החזר עצם מעבר מהלך עם הלוח החדש ובלוח היעד והלוח הנוכחי בלוח המקור, המהלך, וסטטוס בוצע.

3. שם מחלקה: Move
תיאור מחלקה: מחלקה אבסטרקטית אשר מייצגת מהלך.
המחלקה תשמור בתוכה את לוח המשחק בו התבצע המהלך, עצם הכלי שזז, והמיקום אליו זז הכלי, וממנה ירשו סוגי המהלכים השונים, כמו מהלך אכילה, מהלך הצרחה, מהלך של חייל ועוד...

המחלקה אחראית על ייצוג מהלך משחק ועל ביצוע מהלך, וזוהי הפעולה החשובה ביותר בה.

```
public Board executeMove()
```

הפעולה תיצור לוח חדש, תשנה את תכונותיו על פי המהלך החדש, ותחזיר את הלוח החדש.

הפעולה תידרס על פי הצורך במחלקות היורשות ממחלקת המהלך.

יעילות:

O(n)

כיוון שבבנייה של board חדש נקראת הפעולה getAllLegalMoves שיעילותה O(n), יעילות הפונקציה גם היא כזאת, כאשר n מספר המהלכים האפשריים לשני השחקנים.

תיאור פסאודו קוד של הפונקציה במחלקה הראשית:

1. צור עצם בנאי-לוח חדש
2. אתחל את ערכי עצם הבנאי לערכי תכונת הלוח, והעלה את מספר המהלכים ללא אכילה או תזוזת חייל 1.
3. עבור כל כלי שאינו הכלי שזז, שים אותו במבנה הנתונים של הלוח שבבנאי-הלוח
4. בצע שכפול רדוד לכלי שזז
5. הזז את הכלי החדש שנוצר
6. הכנס את הכלי החדש במיקומו החדש למבנה הנונים של הלוח שבבנאי-הלוח
7. שנה את התור
8. קבע את מהלך המעבר של בנאי הלוח למהלך הנוכחי
9. בנה את הלוח החדש והחזר אותו

4. שם מחלקה: Piece

תיאור מחלקה: מחלקה אבסטרקטית אשר מייצגת כלי משחק.
המחלקה תשמור בתכונותיה את מיקום הכלי, צבעו, ערכו והאם הוא כבר זז.
את המחלקה ירשו מחלקות המייצגות את סוגי הכלים השונים.

המחלקה אחראית על מציאת כל המהלכים האפשריים עבור כלי, וייצוג של כלי על הלוח.

`public abstract List<Move> getLegalMoves(Board board);`	פעולה אבסטרקטית אשר מקבלת לוח משחק, ובמימושה במחלקות היורשות מוצאת את המהלכים האפשריים עבור כלי, ומחזירה רשימה של מהלכים אלו. יעילות: O(n) כאשר n הוא מספר המהלכים האפשריים של הכלי
`public boolean isFriendlyPieceOnCoordinate(Board board, int coordinate)`	פעולה המקבלת לוח ומיקום, ובודקת האם יש כלי מאותו צבע של הכלי על המיקום, ומחזירה אמת אם כן, ושקר אם לא. באמצעות כך נוכל לבדוק האם משבצת היא חוקית לזוז אליה. יעילות: O(1)
`public void movePiece(Move move)`	פעולה המקבלת מהלך ומזיזה את הכלי למיקום החדש. יעילות: O(1)
`public Piece clone()`	פעולה המבצעת Shllow Cloning לכלי ומחזירה את הכלי החדש. יעילות: O(1)

5. שם מחלקה: GameScreen

תיאור מחלקה: המחלקה הגרפית בפרויקט, שומרת תכונות כמו פאנל המשחק, רוחב וגודל מסך המסך ועוד...
המחלקה אחראית על כל ממשק המשתמש הגרפי, הצגת משחק המשחק, הקשבה ללחיצות השחקנים ועוד...

`public GameScreen()`	פעולה בונה המאתחלת את תכונות המסך
`public void drawBoard(Board board)`	פעולה המקבלת לוח וציירת אותו על המסך
`public void resetGame()`	פעולה המאתחלת את המשחק
`public void chooseAiOrNotPopUp()`	פעולה המעלה את מסך הפופ אפ שמבקש מהמשתמש לבחור האם או רוצה לשחק נגד AI או לא
`public void chooseColor()`	פעולה המעלה את מסך הפופ אפ שמבקש מהמשתמש לבחור צבע
`public void gameOver(Result result)`	פעולה המקבלת תוצאת משחק ומציגה את פופ אפ סיום המשחק.
`TilePanel(BoardPanel boardPanel, int tileCoordinate)`	פעולה מאתחלת של Tile, קוראת לצביעת הלוח ומקשיבה ללחיצות השחקנים על העכבר, ומזיזה את הכלים לפי הלוגיקה בהתאם.
`public void AiMove()`	פעולה הנקראת כאשר תור AI לשחק, היא קוראת למינימקס מהלוגיקה ומשחקת עבור ה AI את המהלך שחושב שהוא הטוב לו ביותר.

שם החבילה:AI
תיאור החבילה: חבילה זו כוללת בתוכה את כל המחלקות הסטטיות שנדרשות לשחקן הממוחשב, מחלקת הערכת העמדה, מחלקה אלגוריתם המינימקס והאלפא-ביתא, ומחלקות שונות האחראיות על ניתוח של אספקטים שונים במצב המשחק, כגון מחלקת Material, CenterControl, Mobility ועוד.

שם המחלקה: PositionEvaluation
תיאור המחלקה: מחלקה סטטית האחראית על הערכת העמדה, והיא כוללת בתוכה קריאות שונות לפונקציות חישוב אשר בסופו של דבר מתנקזות להערכה מספרית של עמדה שאותה תחזיר פעולה ה Evaluate.

`private static double checkmate(Player player)`	פעולה סטטית המקבלת שחקן ומחזירה ערך מספרי מסוג double אם יריבו של השחקן במצב מט, או 0 אם לא. יעילות: O(n)
`private static GameStage calculateGameStage(Board board)`	פעולה המקבלת לוח ומחזירה את מצב המשחק הנוכחי באמצעות חיושוב כמות הכלים על הלוח. יעילות: O(1)
`public static double score(Board board, Player player, GameStage gameStage)`	פעולה חשובה מאוד, אשר מקבלת את לוח המשחק, שחקן, ושלב משחק, ומחזירה ערך מספרי להערכת העמדה עבור שחקן זה-באמצעות קריאות שונות לפונקציות ההערכה השונות בחבילת ה AI. יעילות: O(n) כאשר n מספר המהלכים החוקיים של השחקן
`public static double evaluate(Board board)`	פעולת הערכת העמדה-לב ליבה של השחקן הממוחשב, תקבל לוח, ותעריך את ערכו המספרי באמצעות חיסור הערכת עמדת הלבן עם הערכת עמדת השחור. יעילות: O(n)-כאשר n הוא מספר המהלכים האפשריים בעמדה(נובע בשל פונקציית ההערכה השונות המגיעות מ score)

```
public static double score(Board board, Player player,
GameStage gameStage)
```

תיאור פסאודו קוד של הפונקציה:

1. אם מצב התחלה, החזר:
   a. חישוב כל הפעולות הסטטיות של ניתוח הלוח במצב התחלה עבור שחקן זהθ
2. אם מצב אמצע, החזר:
   a. חישוב כל הפעולות הסטטיות של ניתוח הלוח במצב אמצע עבור שחקן זה
3. אם מצב סיום, החזר:
   a. חישוב כל הפעולות הסטטיות של ניתוח הלוח במצב סוף עבור שחקן זה

```
public static double evaluate(Board board)
```

תיאור פסאודו קוד של הפונקציה:

1. חשב את מצב המשחק ושמור אותו
2. החזר את תוצאת השחקן הלבן פחות תוצאת השחקן השחור

שם המחלקה: Minimax

תיאור המחלקה: מחלקה סטטית האחראית על מימוש אלגוריתם המינימקס ואלגוריתם ה Quiescence search.

`public static Move MiniMaxAB(final Board board, int depth)`	פונקציה המקבלת לוח ועומק חישוב, ובאמצעות אלגוריתם המינימקס אלפא-ביתא, ובאמצעות ההערכת העמדה שהגדרנו מקודם, מוצאת את המהלך הטוב ביותר בעמדה הנוכחית, ומחזירה אותו. יעילות הפונקציה: **O(nlogn*b^(d/2))** n - מספר המהלכים האפשריים b-פקטור החיתוך d-עומק החישוב
`private static List<Move> sortMoves(Collection<Move> legalMoves)`	פונקציה המקבלת רשימה של מהלכים, וממיינת אותם על פי סבירותם להיות טובים עבור השחקן, על מנת להקל על אלגוריתם האלפא-ביתא לבצע יותר גיזומים. יעילות: O(nlogn) היא מספר המהלכים האפשריים n כאשר
`private static int calculateQuiescenceDepth(final Board toBoard, final int depth)`	פונקציה המקבלת את הלוח ואת העומק, ומחזירה את העומק החדש לפי כמה והאם המהלכים הקודמים היו מהלכים שקטים או לא. יעילות הפוקנציה: O(1)
`public static double max(final Board board, final int depth, final double highest, final double lowest)`	פונקציה המקבלת לוח, עומק חישוב, ערך גבוה ביותר, וערך נמוך ביותר, ומחזירה את הערכת העמדה עם חישוב העומק קדימה לשחקן המקסימום.(פונקציית המקסימום של המינימקס)
`public static double min(final Board board, final int depth, final double highest, final double lowest)`	פונקציה המקבלת לוח, עומק חישוב, ערך גבוה ביותר, וערך נמוך ביותר, ומחזירה את הערכת העמדה עם חישוב העומק קדימה לשחקן המינימום.(פונקציית המינימום של המינימקס)

```
public static Move MiniMaxAB(final Board board, int depth)
```

1. קבע ערך ההערך הגבוה ביותר שנראה למינוס אינסוף, והערך הנמוך ביותר שנראה לאינסוף
2. מיין את המהלכים האפשריים
3. עבור כל מהלך אפשרי:
    a. בצע את המהלך לעצם מעבר-מהלך חדש
    b. אם סטטוס ביצוע המהלך הוא בוצע
        i. בדוק האם המהלך הוא מהלך מט, אם כן:
            1. החזר את המהלך
        ii. אחרת:
            1. אם צבע השחקן לבן, הערך הנוכחי שווה לקריאה לפוקנציית המינימום, אחרת לקריאה לפונקציית המקסימום.
            2. אם צבע השחקן לבן, והערך הנוכחי גדול מהערך הגבוה ביותר שראינו-שים את הערך הנוכחי בערך הכי גבוה שראינו, ובמהלך הטוב ביותר את המלך הנוכחי

3. אם צבע השחקן שחור, והערך הנוכחי קטן מהערך הנמוך ביותר שראינו-שים את הערך הנוכחי בערך הכי נמוך שראינו, ובמהלך הטוב ביותר את המהלך הנוכחי.
4. החזר את המהלך הטוב ביותר

```
private static int calculateQuiescenceDepth(final Board
toBoard, final int depth)
```

תיאור פסאודו קוד של הפונקציה:

1. אם עומק שווה ל 1, ולא הגענו לגבול החיפושים האפשרי, בצע
   a. אתחל את מד הפעילות ל 0
   b. אם המהלך הוא שח, העלה את מד הפעילות ב-1
   c. אם המהלך הוא קידום חייל, העלה את מד הפעילות ב-2
   d. אם המהלך הוא אכילה-העלה את מד הפעילות ב-2
   e. אם המהלך הוא איום-חייל, העלה את מד הפעילות ב 1
   f. עבור כל אחד משלושת המהלכים האחרונים:
      i. אם המהלך הוא אכילה-העלה את מד הפעילות ב 1
2. אם מד הפעילות גדול מ-2, החזר שלוש
3. החזר את העומק פחות 1.

*חשוב לציין, פונקציה זו היא דינמית וייתכן ושתשתנה קלות עד הגשת הפרויקט.

<u>מחלקות הערכה:</u>
שם מחלקה: Material
תיאור המחלקה: המחלקה אחראית על ניתוח הפאן החומרי בלוח, כמות הכלים שיש לכל שחקן ערכם וכדומה.

```
public static double material(List<Piece> allActivePieces
```

פעולה זו מקבלת רשימה של החתיכות הפעילות של שחקן ואחראית על חישוב הערך החומרי של החתיכות הנתונות והחזרתו.
על חומר בשחמט ניתן לקרוא כאן:
https://www.chessprogramming.org/Material
יעילות הפונקציה:
O(1)

שם מחלקה: Mobility
תיאור המחלקה: המחלקה אחראית על ניתוח הניידות של השחקן בלוח, ולהעריך את עמדתו על פי כך.

```
public static double mobility(Player player)
```

פעולה זו מקבלת את העצם של השחקן שלו אנו מחשבים את הניידות ואחראית על חישוב ניידותו.
על ניידות ניתן לקרוא כאן:
https://www.chessprogramming.org/Mobility
יעילות הפונקציה:
O(n)
כאשר n הוא מספר המהלכים האפשריים עבור השחקן.

שם מחלקה: PawnStruct
תיאור המחלקה: המחלקה אחראית על ניתוח מבנה החיילים של השחקן בלוח, ולהעריך את עמדתו על פי כך.

הפעולה מקבלת רשימה של כל הכלים הפעילים של השחקן, ומחזירה מערך בגדול 8 של מספר החיילים כל עמודה(כאשר האינדקס במערך מייצג עמודה) יעילות הפונקציה: O(1)	`public static int[] getPawnsPlacesArray(List<Piece> allActivePieces)`
פעולה המקבלת את עצם השחקן ואת חייליו הפעילים, ומחזירה הערכת עמדה מספרים עבור מבנה החיילים שלו. יעילות הפונקציה: O(1)	`public static double pawnStruct(Player player, List<Piece> allActivePieces)`
הפונקציה מקבלת את מערך החיילים ומחזירה ערך עבור הערכת איי החיילים. על איי חיילים בשחמט ניתן לקרוא כאן: https://en.wiktionary.org/wiki/pawn_island	`private static double PawnIslands(int[] pawnsArr)`

	יעילות הפונקציה: O(1)
`public static double calculateIsolatedPawns(int[] pawnsArr)`	הפונקציה מקבלת את מערך החיילים ומחזירה ערך עבור הערכת החיילים המבודדים. על חיילים מבודדים בשחמט ניתן לקרוא כאן: https://en.wikipedia.org/wiki/Isolated_pawn יעילות הפונקציה: O(1)
`public static double calculateDoublePawns(int[] pawnsArr)`	הפונקציה מקבלת את מערך החיילים ומחזירה ערך עבור הערכת החיילים הכפולים. על חיילים כפולים בשחמט ניתן לקרוא כאן: https://en.wikipedia.org/wiki/Doubled_pawns יעילות הפונקציה: O(1)

שם מחלקה: CenterControl

תיאור המחלקה: המחלקה אחראית על ניתוח השליטה במרכז של השחקן בלוח, ולהעריך את עמדתו על פי כך.

על שליטה על המרכז בשחמט ניתן לקרוא כאן:

https://en.wikipedia.org/wiki/Chess_strategy#Control_of_the_center

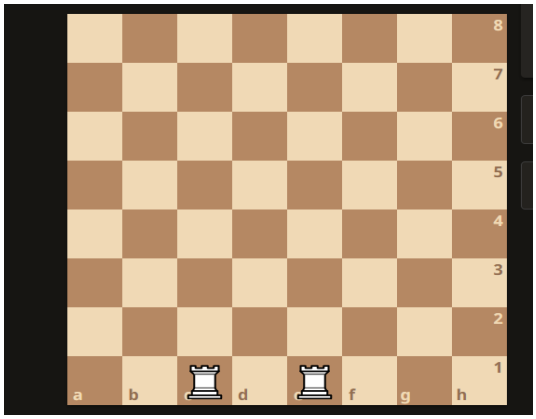
`private static double pawnsOnCenter(Player player, Board board)`	הפונקציה מקבלת את השחקן והלוח ומחזירה ערך עבור הערכת חייליו במרכז. על חיילים במרכז בשחמט ניתן לקרוא כאן: https://en.wikipedia.org/wiki/Doubled_pawns יעילות הפונקציה: O(1)
`public static double calculateAttacksOnEnemyCenter(Player player, List<Move> allPlayersLegalMoves)`	הפונקציה מקבלת את השחקן ורשימת מהלכיו האפשריים ומחזירה ערך עבור הערכת המקפות על חצי המרכז של היריב. יעילות הפונקציה: O(n) כאשר n מספר המהלכים האפשריים של השחקן
`public static double calculateAttacksOnSelfCenter(Player player, List<Move> allPlayersLegalMoves)`	הפונקציה מקבלת את השחקן ורשימת מהלכיו האפשריים ומחזירה ערך עבור הערכת המקפות על חצי המרכז שלו. יעילות הפונקציה: O(n) כאשר n מספר המהלכים האפשריים של השחקן
`public static double centerControl(Player player, Board board)`	פעולה המקבלת שחקן ולוח ומחזירה הערכת עמדה עבור השליטה על המרכז של השחקן, באמצעות הפונקציות הסטטיות המתוארות מעלה. יעילות הפונקציה: O(n) כאשר n מספר המהלכים האפשריים של השחקן

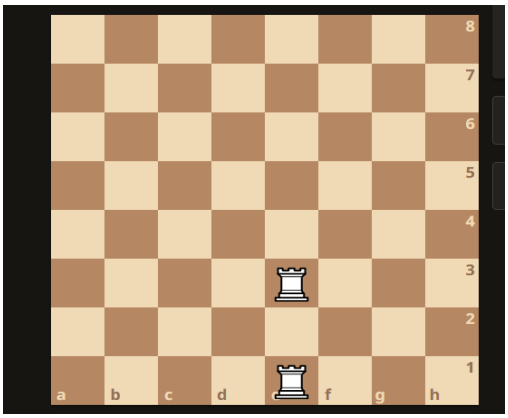
שם מחלקה: RookStruct

תיאור המחלקה: המחלקה אחראית על ניתוח מבנה הצריחים של השחקן בלוח, ולהעריך את עמדתו על פי כך.

`private static List<Piece> findRooks(List<Piece> playerActivePieces)`	פעולה המקבלת רשימה של הכלים הפעילים של שחקן ומחזירה רשימה של הצריחים שלו. יעילות: O(1)
`private static double calculateConnectedColumnRooks(Piece rook1, Piece rook2, Board board)`	פעולה המקבלת עצמים מסוג Piece שהם שני הצריחים של השחקן ולוח, ונותנת הערכת עמדה עבור חיבור על עמודה של שני צריחים. יעילות: O(1)
`private static double calculateConnectedRowRooks(Piece rook1, Piece rook2, Board board)`	פעולה המקבלת עצמים מסוג Piece שהם שני הצריחים של השחקן ולוח, ונותנת הערכת עמדה עבור חיבור על שורה של שני צריחים. יעילות: O(1)
`public static double rookStruct(Board board, List<Piece> playerActivePieces)`	הפונקציה מקבלת את השחקן וחייליו הפעילים ומחזירה ערך עבור הערכת מבנה צריחיו. יעילות הפונקציה: O(1)

תמונות להמחשה-מימין: חיבור על עמודה של שני צריחים, משמאל: חיבור על שורה של שני צריחים.





שם מחלקה: PieceLocation
תיאור המחלקה: המחלקה אחראית על ניתוח מיקומי החיילים של השחקן בלוח, ולהעריך את עמדתו על פי כך.

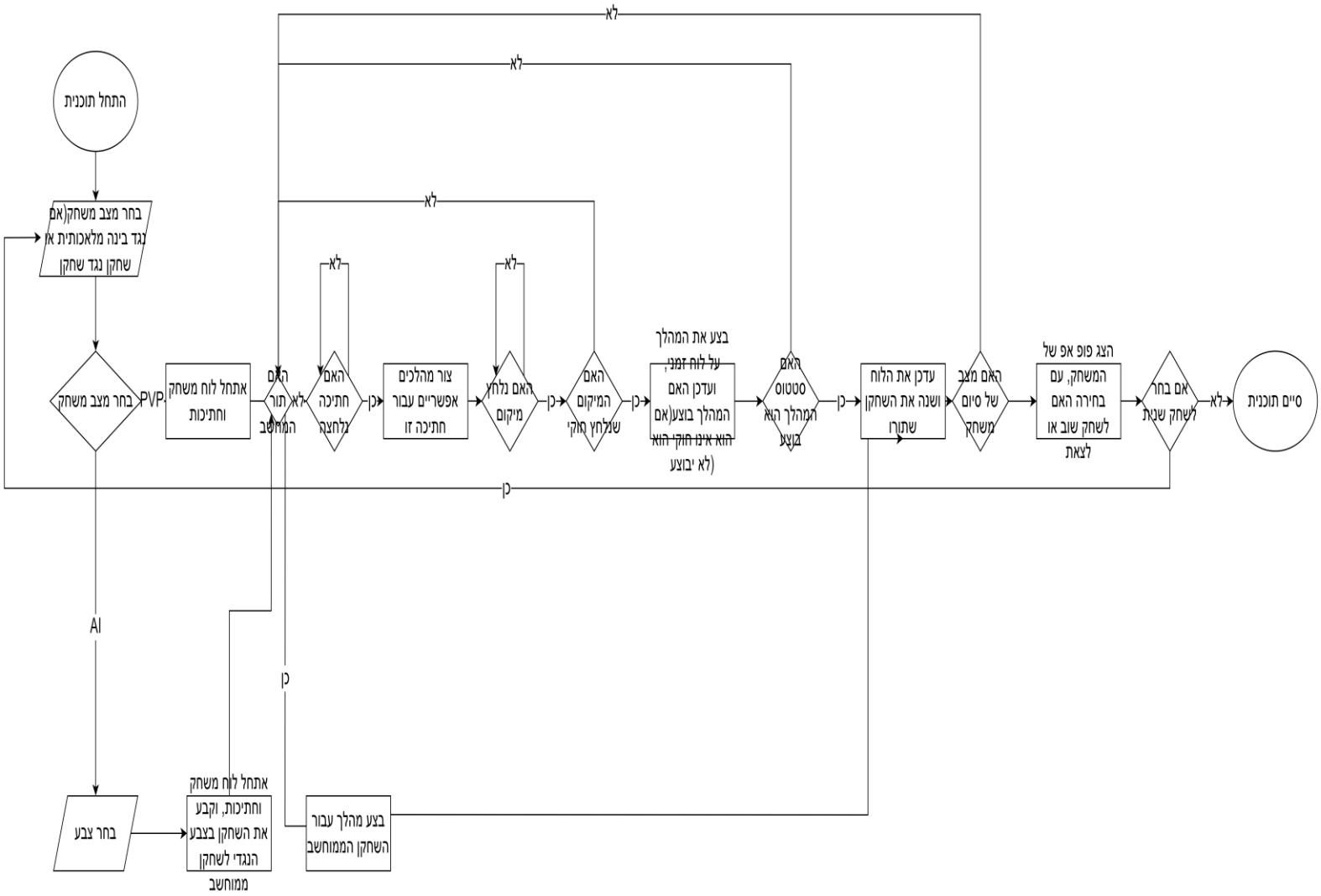
```
public static double pieceLocation(List<Piece>
allActivePieces, GameStage gameStage)
```

פעולה זו מקבלת את רשימת כלי השחקן הפעילים ובה אנו מחשבים את מיקומי חייליו ומחזירים את ערך עמדתו לפיכך.
עוד על מיקומי חיילים ו Piece-Square-Tabales ניתן לקרוא כאן:

https://www.chessprogramming.org/Piece-Square_Tables

יעילות הפונקציה:
O(1)

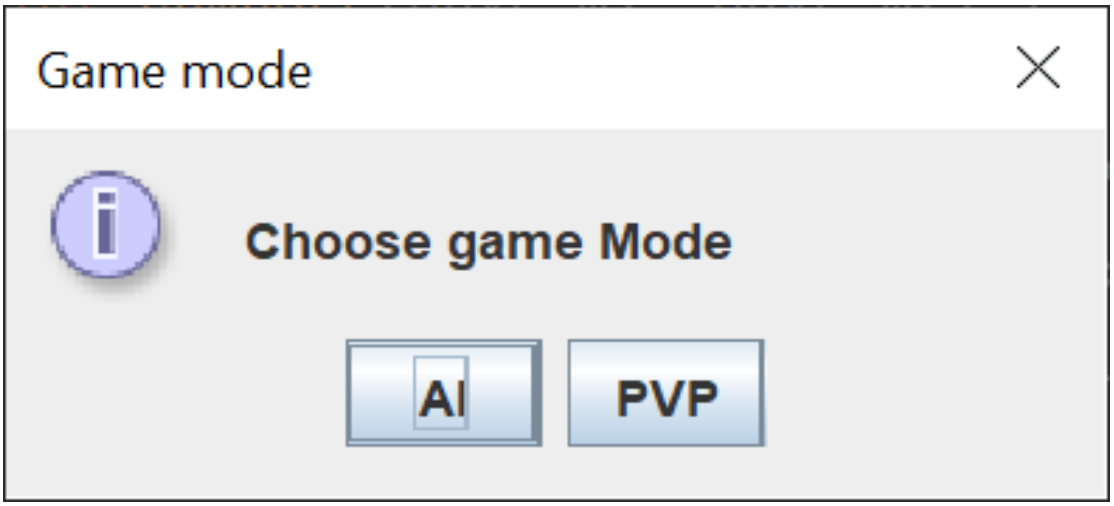
התוכנית הראשית



מדריך למשתמש

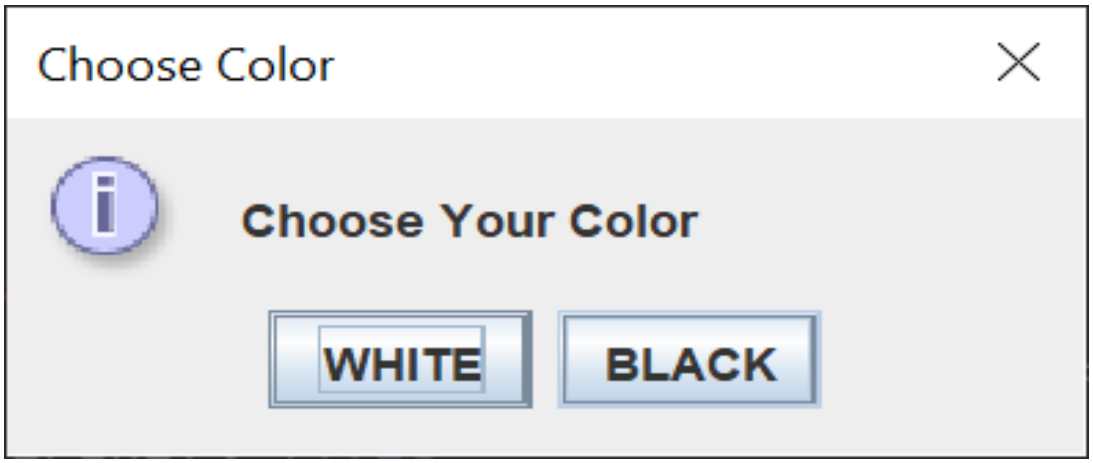
- בעת הפעלת התוכנית, יוצג בפנינו ה Pop up הבא, המבקש מאיתנו לבחור מצב משחק, כאשר לחיצה על X תוציא אותנו מהמשחק, לחיצה על PvP תשלח אותנו למצב של שחקן אנושי נגד שחקן אנושי, ולחיצה נוספת תוביל אותנו ל Pop up בחירת הצבע.

10.מסך בחירת מצב משחק



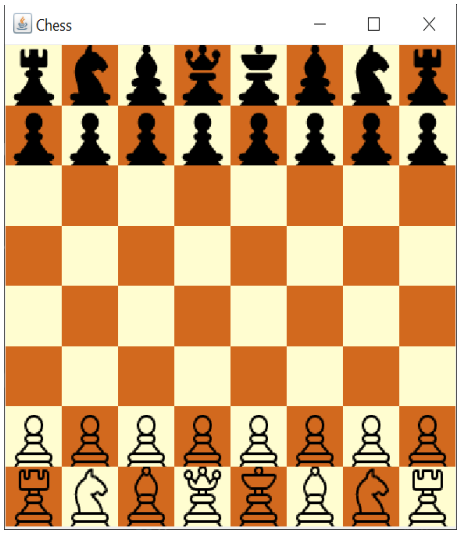
- אם נבחר ב AI יוצג בפנינו ה Pop up הבא, ביבקש מאיתנו לבחור צבע, כמו ב Pop up הקודם, לחיצה על X תוציא אותנו מהמשחק, ולחיצה על אחד הצבעים תוביל אותנו למצב משחק נגד המחשב בצבע שבחרנו.

11.מצב בחירת צבע

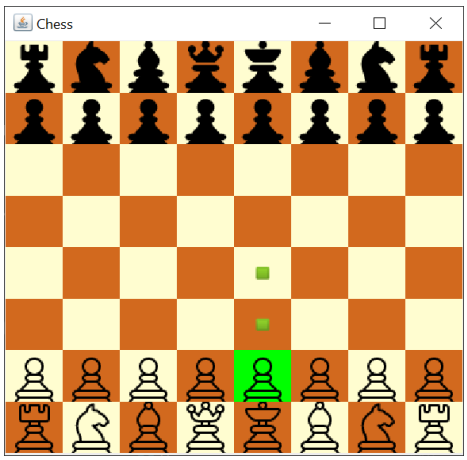


- מסך המשחק- במצב המשחק PvP נשחק שחקן נגד שחקן, וב AI ישחק נגדנו המחשב, כדי לבצע מהלך, עלינו ללחוץ באמצעות העכבר על הכלי איתו נרצה לבצע את המהלך, לאחר מכן יצבע הכלי שבחרנו בירוק ועל המהלכים האפשריים לכלי שבחרנו תופיע נקודה ירוקה.
במצב של שח משבצתו של המלך המותקף תיצבע באדום.
כאשר נלחץ פעם נוספת על מקום חוקי, הכלי יזוז אליו, ואם נלחץ על מקום לא חוקי תתבטל בחירת הכלי.

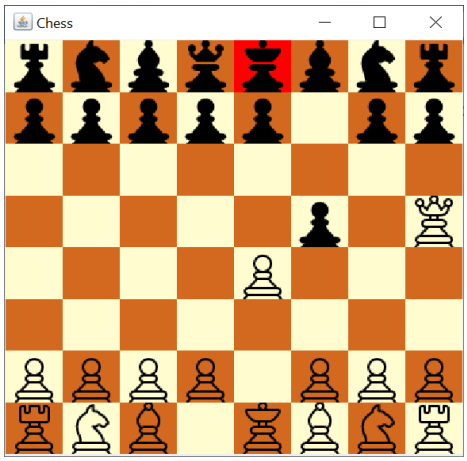
12.לוח המשחק



13.ולאחר בחירת כלי



14.בעת שח



- מצב סיום- בעת סיום המשחק יופיע לנו Pop up המציג את תוצאת המשחק, ושואל אותנו אם נרצה לנסות שנית, לחיצה על OK תחזיר אותנו לתחילת התוכנית, בעוד לחיצה על Cancel או X תוציא אותנו מהתוכנית.

15.מצב סיום משחק

סיכום אישי

בפרויקט הגמר שלפניך הושקעו שעות רבות של עבודה ומחשבה, במטרה לפתח פרויקט מוצלח שעונה על פתרון בעיות אלגוריתמיות שונות ומסובכות. אני מרגיש שהפרויקט תרם לי הרבה, שכן הוא שילב בתוכו פרקטיקה רבה, בשילוב נושאים תיאורטיים חשובים, ותכנון ועשייה רבה.

לא אשקר-לא תמיד לאורך העבודה על הפרויקט "ליקקתי דבש" וכן, העבודה על הפרויקט גם הביאה איתה רגעים של תסכול מבאגים טורדניים או לחץ ממועדי הגשה הולכים ומתקרבים, אך בסיכום כולל של העבודה על הפרויקט, אני יכול להגיד בפה מלא שלגמרי נהניתי.

העבודה על הפרויקט אתגרה אותי, דרשה ממני לחשוב מחוץ ובתוך הקופסא, ונתנה לי כלים שרק פרויקט בסדר גודל כזה יכול לתת.

היא נתנה לי כלים חשובים להמשך החיים, כמו עמידה בלוח זמנים, שבחיים הבוגרים לעיתים אינה מתפשרת, תכנון של פרויקט לפני העבודה עליו, כדי לעמוד בסטנדרטים שקבעתי לעצמי, ועוד כלים רבים וחשובים שאני בטוח שיעזרו לי בהמשך הדרך.

אתגר מרכזי שעמד בפני בעבודה על הפרויקט הוא בניית הערכת העמדה של המשחק-כמו שציינתי בתחילת הספר, הערכת עמדה בשחמט היא תורה שלמה, יש אינספור סוגים, ווריאנטים, דרכים וצורות להעריך עמדה, שכמובן לא את כולם ואף לא את רובם הייתי מספיק לממש.

לכן, הייתי צריך להחליט עם עצמי על הדרכים שהן אנקוט, ולשים בצד שיטות שנראו לי פחות רלוונטיות או פחות נחוצות כדי באמת להעריך עמדת שחמט בצורה טובה ככל האפשר.

אתגר נוסף, הוא העמידה בזמנים- שנת הי"ג היא שנה עמוסה, מלבד הפרויקט, יש לנו עבודות שיעורים ומבחנים רבים, שאת כולם גם ארצה לשלב עם חיי חברה, בילויים, טיסה עם המשפחה וכדומה, שלא תמיד מתאפשרים בזמן האידאלי בשבילי(לדוגמא: חברים חוזרים מהצבא או משנת השירות רק פעם בשבועיים או משפחה שלא יכולה לטוס במועדים המושלמים עבורי).

לכן העמידה בלוחות הזמנים של הפרויקט היוותה אתגר עבורי, שדרש ממנו תכנון מוצלח של לוח הזמנים, ועבודה על הפרויקט פעמים רבות במקום סתם "להתבטל".

מסקנה אחת שהגעתי אליה מהעבודה על הפרויקט היא החשיבות של תכנון לפני עבודה על פרויקטים, בעבודה על הפרויקט, נעזרתי מאוד בתכנון המחלקות שכתבתי בתחילת השנה, ובמחקר אישי שביצעתי עוד לפני תחילת העבודה על הפרויקט. נכון-מחלקות נוספו ומחלקות נמחקו, פונקציות נוספו ופונקציות נשארו על רצפת חדר העריכה אך אני מאמין כי התכנון הכללי והחשיבה המוקדמת על מה אני רוצה להשיג מהפרויקט, ואיך אני רוצה שיעבוד הובילה אותי ליצירת תוצר שאני אישית מאוד מרוצה ומסופק ממנו.

אילו הייתי מתחיל לעבוד היום על הפרויקט- דבר אחד שהייתי משנה הוא הסדר של העבודה שלי על הפרויקט, במהלך העבודה על הפרויקט, תכנונו תרם לי במיקוד על אילו דברים אני צריך ורוצה שיהיו בפרויקט, אך לצערי לא תכננתי את סדר העבודה, ומצאתי את עצמי "מקפץ" בין הקודים השונים, כך שלא התמקדתי על מימוש כל דבר בנפרד, לדעתי, חוסר הסדר הזה שרף לי זמן יקר, שכן לא הייתי ממוקד מטרה על דבר ספציפי כל פעם, ועברתי לדבר חדש מבלי לסיים דבר קודם.

אני מאמין כי יותר סדר בעבודה היה הופך את העבודה על הפרויקט ליעילה יותר עבורי, ואולי הייתי מספיק לממש דבר נוסף או שניים😉.

לסיכום, אני רוצה להגיד שהעבודה על הפרויקט הייתה מעשירה ומהנה, וברוב הזמן(אך לא תמיד) נהניתי לשבת ולבנות את הפרויקט. אני מאמין כי העבודה על הפרויקט תרמה לי כלים רבים, ותמשיך לתרום לי בהמשך, לדוגמה בתפקידי הצבאי.

ביבליוגרפיה

Jeroen W.T. Carolus(2006), **Alpha-Beta with Sibling Prediction Pruning in Chess**, Masters thesis Computer Science, Faculteit der Natuurwetenschappen, Wiskunde en Informatica University of Amsterdam Netherlands

https://www.chessprogramming.org/Main_Page

https://www.chessprogramming.org/Evaluation

https://www.youtube.com/watch?v=V_2-LOvr5E8&list=PLQV5mozTHmacMeRzJCW_8K3qw2miYqd0c

קוד הפרוייקט

מחלקה:Piece

```
package logic.Pieces;

import logic.Board;
import logic.Color;
import logic.Move;
import logic.player.AI.GameStage;

import java.util.List;
import java.util.Objects;

/**
 * this class is an abstract class of a piece on the board, and it
extends by the spastic class of every piece
 * @author dotanraif
 */
public abstract class Piece implements Cloneable {
    protected final Color color;
    protected int position;
    protected boolean isFirstMove;
    public double value;

    // getter
    public Color getColor() {
        return color;
    }

    // getter
    public int getPosition() {
        return position;
    }

    // getter
    public boolean isFirstMove() {
        return isFirstMove;
    }

    public Piece(int position, final Color color, boolean
isFirstMove) {
        this.position = position;
        this.color = color;
        this.isFirstMove = isFirstMove;
        this.value = getValue();
    }

    // getter
    public double getValue() {
        return value;
    }

    // setter
    public void setFirstMove(boolean firstMove) {
        isFirstMove = firstMove;
    }

    /**
     * get the bonus for the piece location according to her Piece-
Square Tables
```

```
     * @return the bonus to add
     */
    public abstract double locationBonus(GameStage gameStage);

    /**
     * finds all the legal moves for a piece in a given board
     * @param board the board we check in
     * @return List of all the legal moves for a piece(some special
moves not include)
     */
    public abstract List<Move> getLegalMoves(Board board);

    @Override
    public String toString() {
        return " " + this.getClass();
    }

    /**
     * check if a given coordinate is valid(in the board, between 0
and 63)
     * @param coordinate coordinate as number
     * @return true if the coordinate is valid, false if not
     */
    public boolean isValidCoordinate(int coordinate) {
        return coordinate >= 0 && coordinate <= 63;
    }

    /**
     *  check if the coordinate is not empty, and it has friendly
piece on it
     * @param board the board we check on
     * @param coordinate the coordinate we check on
     * @return true if the piece is friendly, else otherwise
     */
    public boolean isFriendlyPieceOnCoordinate(Board board, int
coordinate) {
        return board.board_state.get(coordinate) != null &&
board.board_state.get(coordinate).color == this.color;
    }

    /**
     * changes the piece that been moved to the current attributes
     * @param move the move we make
     */
    public void movePiece(Move move)
    {
        this.position = move.getCoordinateMovedTo();
        this.isFirstMove = false;
    }

    /**
     * equals override, check if object o is equal to this piece
     * @param o object to check if equal
     * @return true if all attributes equal, else false
     */
    @Override
    public boolean equals(Object o) {
        // if same
        if (this == o) return true;
        // if not the right class
        if (o == null || getClass() != o.getClass()) return false;
```

```
        // casting
        Piece piece = (Piece) o;
        // check if all attributes equal
        return position == piece.position && color == piece.color;
    }

    @Override
    public int hashCode() {
        return Objects.hash(color, position, isFirstMove);
    }

    /**
     * shallow cloning for piece object
     * @return return clone of the object
     */
    @Override
    public Piece clone() {
        try {
            // TODO: copy mutable state here, so the clone can't
change the internals of the original
            return (Piece) super.clone();
        } catch (CloneNotSupportedException e) {
            throw new AssertionError();
        }
    }
}
```

מחלקה:Pawn

```
package logic.Pieces;

import logic.Board;
import logic.Color;
import logic.Move;
import logic.player.AI.GameStage;

import java.util.ArrayList;
import java.util.List;
/**
 * this class represent a pawn, and it extends Piece class
 * @author dotanraif
 * @see logic.Pieces.Piece
 */
public class Pawn extends Piece {

    // piece square table for white pawn
    private final static double[] WHITE_PAWN_PREFERRED_COORDINATES =
{
            0,  0,  0,  0,  0,  0,  0,  0,
            0.75, 0.75, 0.75, 0.75, 0.75, 0.75, 0.75, 0.75,
            0.25, 0.25, 0.29, 0.29, 0.29, 0.29, 0.25, 0.25,
            0.05,  0.05, 0.1, 0.55, 0.55, 0.1,  0.05,  0.05,
            0,  0,  0, 0.2, 0.2,  0,  0,  0,
            0.05, -0.05,-0.1,  0,  0,-0.1, -0.05,  0.05,
            0.05, 0.1, 0.1,-0.2,-0.2, 0.1, 0.1,  0.05,
            0,  0,  0,  0,  0,  0,  0,  0
    };
    // piece square table for black pawn
    private final static double[] BLACK_PAWN_PREFERRED_COORDINATES =
{
            0,  0,  0,  0,  0,  0,  0,  0,
            0.05, 0.1, 0.1,-0.2,-0.2, 0.1, 0.1,  0.05,
            0.05, -0.05,-0.1,  0,  0,-0.1, -0.05,  0.05,
            0,  0,  0, 0.2, 0.2,  0,  0,  0,
            0.05,  0.05, 0.1, 0.55, 0.55, 0.1,  0.05,  0.05,
            0.25, 0.25, 0.29, 0.29, 0.29, 0.29, 0.25, 0.25,
            0.75, 0.75, 0.75, 0.75, 0.75, 0.75, 0.75, 0.75,
            0,  0,  0,  0,  0,  0,  0,  0
    };

    final int[] move_mask = {8, 16, 7 , 9};

    public Pawn(int position, Color color, boolean isFirstMove) {
        super(position, color, isFirstMove);
        this.value = 1;
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public double locationBonus(GameStage gameStage) {
        return this.color == Color.White ?
WHITE_PAWN_PREFERRED_COORDINATES[this.position] :
BLACK_PAWN_PREFERRED_COORDINATES[this.position];
    }

    /**
```

```
     * {@inheritDoc}
     */
    @Override
    public List<Move> getLegalMoves(Board board) {
        int possible_coordinate;
        List<Move> legalMoves = new ArrayList<>();
        for (int mask : move_mask) {
            possible_coordinate = position + (mask *
getDirection(color));
            if (isValidCoordinate(possible_coordinate)) {
                if(isFirstColumnExtremeCase(this.position, mask,
color))
                    continue;
                if (mask == 8 &&
board.board_state.get(possible_coordinate) == null) {
                    legalMoves.add(new Move.PawnMove(board, this,
possible_coordinate));
                }
                else if (mask == 16 &&
board.board_state.get(possible_coordinate) == null &&
                        board.board_state.get(possible_coordinate - 8
* getDirection(color)) == null && isFirstMove)
                {
                    legalMoves.add(new Move.PawnMove(board, this,
possible_coordinate));
                }
                else if(mask == 7 &&
board.board_state.get(possible_coordinate) != null &&
!isFriendlyPieceOnCoordinate(board, possible_coordinate))
                {
                    legalMoves.add(new Move.PawnAttackMove(board,
this, possible_coordinate,
board.getPieceAtCoordinate(possible_coordinate)));
                }
                else if(mask == 9 &&
board.board_state.get(possible_coordinate) != null &&
!isFriendlyPieceOnCoordinate(board, possible_coordinate))
                {
                    legalMoves.add(new Move.PawnAttackMove(board,
this, possible_coordinate,
board.getPieceAtCoordinate(possible_coordinate)));
                }
            }
        }
        return legalMoves;
    }

    public int getDirection(Color color) {
        if (color == Color.White)
            return -1;
        return 1;
    }

    /**
     * this function use to check that a piece is not going out from
one side of the board to the other
     * because it's on first column
     * @param coordinate the coordinate the piece is in
     * @param mask the mask we check in
     * @return true if the move is illegal-false otherwise
     */
```

```
    public boolean isFirstColumnExtremeCase(int coordinate, int mask,
Color color) {
        // TODO possible problem with direction
        if (coordinate % 8 == 0 && ((mask == 7 && color ==
Color.Black)||(mask == 9 && color == Color.White)))
            return true;
        else return ((coordinate + 1) % 8 == 0 && ((mask == 9 &&
color == Color.Black)||(mask == 7 && color == Color.White)));
    }


}
```

מחלקה: Bishop

```
package logic.Pieces;

import logic.Board;
import logic.Color;
import logic.Move;
import logic.player.AI.GameStage;

import java.util.ArrayList;
import java.util.List;

/**
 * this class represent a bishop, and it extends Piece class
 * @author dotanraif
 * @see logic.Pieces.Piece
 */
public class Bishop extends Piece {
    final int[] move_mask = {7, 9, -7, -9};

    // piece square table for white bishop
    private final static double[] WHITE_BISHOP_PREFERRED_COORDINATES
= {
            -0.2,-0.1,-0.1,-0.1,-0.1,-0.1,-0.1,-0.2,
            -0.1,  0,  0,  0,  0,  0,  0,-0.1,
            -0.1,  0,  0.05, 0.1, 0.1,  0.05,  0,-0.1,
            -0.1,  0.05,  0.05, 0.1, 0.1,  0.05,  0.05,-0.1,
            -0.1,  0, 0.1, 0.1, 0.1, 0.1,  0,-0.1,
            -0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1,-0.1,
            -0.1,  0.05,  0,  0,  0,  0,  0.05,-0.1,
            -0.2,-0.1,-0.1,-0.1,-0.1,-0.1,-0.1,-0.2
    };

    // piece square table for black bishop
    private final static double[] BLACK_BISHOP_PREFERRED_COORDINATES
= {
            -0.2,-0.1,-0.1,-0.1,-0.1,-0.1,-0.1,-0.2,
            -0.1,  0.05,  0,  0,  0,  0,  0.05,-0.1,
            -0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1,-0.1,
            -0.1,  0, 0.1, 0.1, 0.1, 0.1,  0,-0.1,
            -0.1,  0.05,  0.05, 0.1, 0.1,  0.05,  0.05,-0.1,
            -0.1,  0,  0.05, 0.1, 0.1,  0.05,  0,-0.1,
            -0.1,  0,  0,  0,  0,  0,  0,-0.1,
            -0.2,-0.1,-0.1,-0.1,-0.1,-0.1,-0.1,-0.2
    };

    public Bishop(int position, Color color, boolean isFirstMove) {
        super(position, color, isFirstMove);
        this.value = 3.3;
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public double locationBonus(GameStage gameStage) {
        return this.color == Color.White ?
WHITE_BISHOP_PREFERRED_COORDINATES[this.position] :
BLACK_BISHOP_PREFERRED_COORDINATES[this.position];
    }
```

```
    /**
     * {@inheritDoc}
     */
    @Override
    public List<Move> getLegalMoves(Board board) {
        int possible_coordinate;
        List<Move> legalMoves = new ArrayList<>();
        for (int mask : move_mask) {
            possible_coordinate = position;
            while (isValidCoordinate(possible_coordinate)) {
                if (isFirstColumnExtremeCase(possible_coordinate,
mask))
                    break;
                possible_coordinate += mask;
                if (isValidCoordinate(possible_coordinate)) {
                    if (board.board_state.get(possible_coordinate) ==
null)
                        // regular move
                        legalMoves.add(new Move.MajorMove(board,
this, possible_coordinate));
                    else if (!isFriendlyPieceOnCoordinate(board,
possible_coordinate)) {
                        // attack move
                        // TODO check if eating move needed
                        legalMoves.add(new Move.AttackMove(board,
this, possible_coordinate,
board.getPieceAtCoordinate(possible_coordinate)));
                        break;
                    } else
                        // if friendly on dest
                        break;
                } else
                    // if out of board
                    break;
            }
        }
        return legalMoves;
    }

    /**
     * this function use to check that a piece is not going out from
one side of the board to the other
     * because it's on first column
     * @param coordinate the coordinate the piece is in
     * @param mask the mask we check in
     * @return true if the move is illegal-false otherwise
     */
    public boolean isFirstColumnExtremeCase(int coordinate, int mask)
{
        if (coordinate % 8 == 0 && (mask == -9 || mask == 7))
            return true;
        else return ((coordinate + 1) % 8 == 0 && (mask == -7 || mask
== 9));
    }
}
```

מחלקה:Knight

```
package logic.Pieces;

import logic.Board;
import logic.Color;
import logic.Move;
import logic.player.AI.GameStage;

import java.util.ArrayList;
import java.util.List;

/**
 * this class represent a knight, and it extends Piece class
 * @author dotanraif
 * @see logic.Pieces.Piece
 */
public class Knight extends Piece {
    final int[] move_mask = {-17, -15, -10, -6, 17, 15, 10, 6};

    // piece square table for white knight
    private final static double[] WHITE_KNIGHT_PREFERRED_COORDINATES
= {
            -0.5,-0.4,-0.3,-0.3,-0.3,-0.3,-0.4,-0.5,
            -0.4,-0.2,  0,  0,  0,  0,-0.2,-0.4,
            -0.3,  0, 0.1, 0.15, 0.15, 0.1,  0,-0.3,
            -0.3,  0.05, 0.15, 0.2, 0.2, 0.15,  0.05,-0.3,
            -0.3,  0, 0.15, 0.2, 0.2, 0.15,  0,-0.3,
            -0.3,  0.05, 0.1, 0.15, 0.15, 0.1,  0.05,-0.3,
            -0.4,-0.2,  0,  0.05,  0.05,  0,-0.2,-0.4,
            -0.5,-0.4,-0.3,-0.3,-0.3,-0.3,-0.4,-0.5
    };

    // piece square table for black knight
    private final static double[] BLACK_KNIGHT_PREFERRED_COORDINATES
= {
            -0.5,-0.4,-0.3,-0.3,-0.3,-0.3,-0.4,-0.5,
            -0.4,-0.2,  0,  0.05,  0.05,  0,-0.2,-0.4,
            -0.3,  0.05, 0.1, 0.15, 0.15, 0.1,  0.05,-0.3,
            -0.3,  0, 0.15, 0.20, 0.20, 0.15,  0,-0.30,
            -0.3,  0.05, 0.15, 0.20, 0.20, 0.15,  0.05,-0.30,
            -0.30,  0, 0.10, 0.15, 0.15, 0.10,  0,-0.30,
            -0.4,-0.2,  0,  0,  0,  0,-0.2,-0.4,
            -0.5,-0.4,-0.3,-0.3,-0.3,-0.3,-0.4,-0.5,
    };


    public Knight(int position, Color color, boolean isFirstMove) {
        super(position, color, isFirstMove);
        this.value = 3.2;
    }


    /**
     * {@inheritDoc}
     */
    @Override
    public double locationBonus(GameStage gameStage) {
        return this.color == Color.White ?
WHITE_KNIGHT_PREFERRED_COORDINATES[this.position] :
BLACK_KNIGHT_PREFERRED_COORDINATES[this.position];
```

```
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public List<Move> getLegalMoves(Board board) {
        // TODO check if eating move needed
        int possible_coordinate;
        List<Move> legalMoves = new ArrayList<>();
        for (int mask : move_mask) {
            if (isExtremeCase(position, mask))
                continue;
            possible_coordinate = mask + position;
            if (isValidCoordinate(possible_coordinate))
                if (board.board_state.get(possible_coordinate) ==
null)
                    // regular move
                    legalMoves.add(new Move.MajorMove(board, this,
possible_coordinate));
                else if (!isFriendlyPieceOnCoordinate(board,
possible_coordinate)) {
                    legalMoves.add(new Move.AttackMove(board, this,
possible_coordinate,
board.getPieceAtCoordinate(possible_coordinate)));
                }
        }
        return legalMoves;
    }

    /**
     * this function use to check that a piece is not going out from
one side of the board to the other
     * because it's on first row or column or second row or column
     * @param coordinate the coordinate the piece is in
     * @param mask the mask we check in
     * @return true if the move is illegal-false otherwise
     */
    public boolean isExtremeCase(int coordinate, int mask) {
        return (isFirstColumnExtremeCase(coordinate, mask) ||
isSecondColumnExtremeCase(coordinate, mask)
                || isFirstRowExtremeCase(coordinate, mask) ||
isSecondRowExtremeCase(coordinate, mask));
    }

    /**
     * this function use to check that a piece is not going out from
one side of the board to the other
     * because it's on first column
     * @param coordinate the coordinate the piece is in
     * @param mask the mask we check in
     * @return true if the move is illegal-false otherwise
     */
    public boolean isFirstColumnExtremeCase(int coordinate, int mask)
{
        if (coordinate % 8 == 0 && (mask == -10 || mask == -17 ||
mask == 6 || mask == 15))
            return true;
        else return (coordinate + 1) % 8 == 0 && (mask == -15 || mask
== -6 || mask == 17 || mask == 10);
    }
```

```
    /**
     * this function use to check that a piece is not going out from
one side of the board to the other
     * because it's on second column
     * @param coordinate the coordinate the piece is in
     * @param mask the mask we check in
     * @return true if the move is illegal-false otherwise
     */
    public boolean isSecondColumnExtremeCase(int coordinate, int
mask) {
        if ((coordinate - 1) % 8 == 0 && (mask == -10 || mask == 6))
            return true;
        else return (coordinate + 2) % 8 == 0 && (mask == -6 || mask
== 10);
    }

    /**
     * this function use to check that a piece is not going out from
one side of the board to the other
     * because it's on first row
     * @param coordinate the coordinate the piece is in
     * @param mask the mask we check in
     * @return true if the move is illegal-false otherwise
     */
    public boolean isFirstRowExtremeCase(int coordinate, int mask) {
        if ((coordinate >= 0 && coordinate <= 7) && (mask == -17 ||
mask == -15 || mask == -10 || mask == -6))
            return true;
        else return ((coordinate >= 56 && coordinate <= 63) && (mask
== 17 || mask == 15 || mask == 10 || mask == 6));
    }

    /**
     * this function use to check that a piece is not going out from
one side of the board to the other
     * because it's on second row
     * @param coordinate the coordinate the piece is in
     * @param mask the mask we check in
     * @return true if the move is illegal-false otherwise
     */
    public boolean isSecondRowExtremeCase(int coordinate, int mask) {
        if ((coordinate >= 8 && coordinate <= 15) && (mask == -17 ||
mask == -15))
            return true;
        else return ((coordinate >= 48 && coordinate <= 55) && (mask
== 17 || mask == 15));
    }
}
```

מחלקה: Rook

```
package logic.Pieces;

import logic.Board;
import logic.Color;
import logic.Move;
import logic.player.AI.GameStage;

import java.util.ArrayList;
import java.util.List;

/**
 * this class represent a Rook, and it extends Piece class
 * @author dotanraif
 * @see logic.Pieces.Piece
 */
public class Rook extends Piece{
    final int[] move_mask = {1, 8, -8, -1};

    // piece square table for white rook
    private final static double[] WHITE_ROOK_PREFERRED_COORDINATES =
{
            0,  0,  0,  0,  0,  0,  0,  0,
            0.05, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2,  0.05,
            -0.05,  0,  0,  0,  0,  0,  0, -0.05,
            -0.05,  0,  0,  0,  0,  0,  0, -0.05,
            -0.05,  0,  0,  0,  0,  0,  0, -0.05,
            -0.05,  0,  0,  0,  0,  0,  0, -0.05,
            -0.05,  0,  0,  0,  0,  0,  0, -0.05,
            0,  0,  0,  0.05,  0.05,  0,  0,  0
    };

    // piece square table for black rook
    private final static double[] BLACK_ROOK_PREFERRED_COORDINATES =
{
            0,  0,  0,  0.05,  0.05,  0,  0,  0,
            -0.05,  0,  0,  0,  0,  0,  0, -0.05,
            -0.05,  0,  0,  0,  0,  0,  0, -0.05,
            -0.05,  0,  0,  0,  0,  0,  0, -0.05,
            -0.05,  0,  0,  0,  0,  0,  0, -0.05,
            -0.05,  0,  0,  0,  0,  0,  0, -0.05,
            0.05, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2,  0.05,
            0,  0,  0,  0,  0,  0,  0,  0,
    };

    public Rook(int position, Color color, boolean isFirstMove) {
        super(position, color, isFirstMove);
        this.value = 5;
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public double locationBonus(GameStage gameStage) {
        return this.color == Color.White ?
WHITE_ROOK_PREFERRED_COORDINATES[this.position] :
BLACK_ROOK_PREFERRED_COORDINATES[this.position];
    }
```

```
    /**
     * {@inheritDoc}
     */
    @Override
    public List<Move> getLegalMoves(Board board) {
        int possible_coordinate;
        List<Move> legalMoves = new ArrayList<>();
        for (int mask : move_mask) {
            possible_coordinate = position;
            while (isValidCoordinate(possible_coordinate))
            {
                if(isFirstColumnExtremeCase(possible_coordinate,
mask))
                    break;
                possible_coordinate += mask;
                if(isValidCoordinate(possible_coordinate))
                {
                    if(board.board_state.get(possible_coordinate) ==
null)
                        // regular move
                        legalMoves.add(new Move.MajorMove(board,
this, possible_coordinate));
                    else if(!isFriendlyPieceOnCoordinate(board,
possible_coordinate)) {
                        // attack move
                        // TODO check if eating move needed
                        legalMoves.add(new Move.AttackMove(board,
this, possible_coordinate,
board.getPieceAtCoordinate(possible_coordinate)));
                        break;
                    }
                    else
                        // if friendly on dest
                        break;
                }
                else
                    // if out of board
                    break;
            }
        }
        return legalMoves;
    }

    /**
     * this function use to check that a piece is not going out from
one side of the board to the other
     * because it's on first column
     * @param coordinate the coordinate the piece is in
     * @param mask the mask we check in
     * @return true if the move is illegal-false otherwise
     */
    public boolean isFirstColumnExtremeCase(int coordinate, int mask)
{
        if (coordinate % 8 == 0 && (mask == -1))
            return true;
        else return ((coordinate + 1) % 8 == 0 && (mask == 1));
    }

}
```

מחלקה: Queen

```
package logic.Pieces;

import logic.Board;
import logic.Color;
import logic.Move;
import logic.player.AI.GameStage;

import java.util.ArrayList;
import java.util.List;

/**
 * this class represent a Queen, and it extends Piece class
 * @author dotanraif
 * @see logic.Pieces.Piece
 */
public class Queen extends Piece {
    final int[] move_mask = {7, 9, -7, -9, 1, 8, -8, -1};


    // piece square table for white queen
    private final static double[] WHITE_QUEEN_PREFERRED_COORDINATES =
{
            -0.2,-0.1,-0.1, -0.05, -0.05,-0.1,-0.1,-0.2,
            -0.1,  0,  0,  0,  0,  0,  0,-0.1,
            -0.1,  0,  0.05,  0.05,  0.05,  0.05,  0,-0.1,
            -0.05,  0,  0.05,  0.05,  0.05,  0.05,  0, -0.05,
            0,  0,  0.05,  0.05,  0.05,  0.05,  0, -0.05,
            -0.10,  0.05,  0.05,  0.05,  0.05,  0.05,  0,-0.1,
            -0.1,  0,  0.05,  0,  0,  0,  0,-0.1,
            -0.2,-0.1,-0.1, -0.05, -0.05,-0.1,-0.1,-0.2
    };

    // piece square table for black queen
    private final static double[] BLACK_QUEEN_PREFERRED_COORDINATES =
{
            -0.2,-0.1,-0.1, -0.05, -0.05,-0.1,-0.1,-0.2,
            -0.1,  0,  0.05,  0,  0,  0,  0,-0.1,
            -0.1,  0.05,  0.05,  0.05,  0.05,  0.05,  0,-0.1,
            0,  0,  0.05,  0.05,  0.05,  0.05,  0, -0.05,
            0,  0,  0.05,  0.05,  0.05,  0.05,  0, -0.05,
            -0.1,  0,  0.05,  0.05,  0.05,  0.05,  0,-0.1,
            -0.1,  0,  0,  0,  0,  0,  0,-0.1,
            -0.2,-0.1,-0.1, -0.05, -0.05,-0.1,-0.1,-0.2
    };
    public Queen(int position, Color color, boolean isFirstMove) {
        super(position, color, isFirstMove);
        this.value = 9;
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public double locationBonus(GameStage gameStage) {
        return this.color == Color.White ?
WHITE_QUEEN_PREFERRED_COORDINATES[this.position] :
BLACK_QUEEN_PREFERRED_COORDINATES[this.position];
    }
    /**
     * {@inheritDoc}
     */
```

```
    @Override
    public List<Move> getLegalMoves(Board board) {
        int possible_coordinate;
        List<Move> legalMoves = new ArrayList<>();
        for (int mask : move_mask) {
            possible_coordinate = position;
            while (isValidCoordinate(possible_coordinate)) {
                if (isFirstColumnExtremeCase(possible_coordinate,
mask))
                    break;
                possible_coordinate += mask;
                if (isValidCoordinate(possible_coordinate)) {
                    if (board.board_state.get(possible_coordinate) ==
null)
                        // regular move
                        legalMoves.add(new Move.MajorMove(board,
this, possible_coordinate));
                    else if (!isFriendlyPieceOnCoordinate(board,
possible_coordinate)) {
                        // attack move
                        // TODO check if eating move needed
                        legalMoves.add(new Move.AttackMove(board,
this, possible_coordinate,
board.getPieceAtCoordinate(possible_coordinate)));
                        break;
                    } else
                        // if friendly on dest
                        break;
                } else
                    // if out of board
                    break;
            }
        }
        return legalMoves;
    }

    /**
     * this function use to check that a piece is not going out from
one side of the board to the other
     * because it's on first column
     * @param coordinate the coordinate the piece is in
     * @param mask the mask we check in
     * @return true if the move is illegal-false otherwise
     */
    public boolean isFirstColumnExtremeCase(int coordinate, int mask)
{
        if (coordinate % 8 == 0 && (mask == -1 || mask == -9 || mask
== 7))
            return true;
        else return ((coordinate + 1) % 8 == 0 && (mask == 1 || mask
== -7 || mask == 9));
    }
}
```

מחלקה:King

```
package logic.Pieces;

import logic.Board;
import logic.Color;
import logic.Move;
import logic.player.AI.GameStage;

import java.util.ArrayList;
import java.util.List;

/**
 * this class represent a king, and it extends Piece class
 * @author dotanraif
 * @see logic.Pieces.Piece
 */
public class King extends Piece {
    final int[] move_mask = {-9, -8, -7, -1, 1, 7, 8, 9};

    // piece square table for white king in start and midgame
    private final static double[]
WHITE_KING_PREFERRED_COORDINATES_STARTMID = {
            -0.3,-0.4,-0.4,-0.5,-0.5,-0.4,-0.4,-0.3,
            -0.3,-0.4,-0.4,-0.5,-0.5,-0.4,-0.4,-0.3,
            -0.3,-0.4,-0.4,-0.5,-0.5,-0.4,-0.4,-0.3,
            -0.3,-0.4,-0.4,-0.5,-0.5,-0.4,-0.4,-0.3,
            -0.2,-0.3,-0.3,-0.4,-0.4,-0.3,-0.3,-0.2,
            -0.1,-0.2,-0.2,-0.2,-0.2,-0.2,-0.2,-0.1,
            0.2, 0.2,  0,  0,  0,  0, 0.2, 0.2,
            0.2, 0.3, 0.1,  0,  0, 0.1, 0.3, 0.2
    };

    // piece square table for black king in start and midgame
    private final static double[]
BLACK_KING_PREFERRED_COORDINATES_STARTMID = {
            0.2, 0.3, 0.1,  0,  0, 0.1, 0.3, 0.2,
            0.2, 0.2,  0,  0,  0,  0, 0.2, 0.2,
            -0.1,-0.2,-0.2,-0.2,-0.2,-0.2,-0.2,-0.1,
            -0.2,-0.3,-0.3,-0.4,-0.4,-0.3,-0.3,-0.2,
            -0.3,-0.4,-0.4,-0.5,-0.5,-0.4,-0.4,-0.3,
            -0.3,-0.4,-0.4,-0.5,-0.5,-0.4,-0.4,-0.3,
            -0.3,-0.4,-0.4,-0.5,-0.5,-0.4,-0.4,-0.3,
            -0.3,-0.4,-0.4,-0.5,-0.5,-0.4,-0.4,-0.3
    };

    // piece square table for white king in ending
    private final static double[]
WHITE_KING_PREFERRED_COORDINATES_ENDING =
            {
                    -0.5, -0.4, -0.3, -0.2, -0.2, -0.3, -0.4, -0.5,
                    -0.3, -0.2, -0.1, 0, 0, -0.1, -0.2, -0.3,
                    -0.3, -0.1, 0.2, 0.3, 0.3, 0.2, -0.1, -0.3,
                    -0.3, -0.1, 0.3, 0.4, 0.4, 0.3, -0.1, -0.3,
                    -0.3, -0.1, 0.3, 0.4, 0.4, 0.3, -0.1, -0.3,
                    -0.3, -0.1, 0.2, 0.3, 0.3, 0.2, -0.1, -0.3,
                    -0.3, -0.3, 0, 0, 0, 0, -0.3, -0.3,
                    -0.5, -0.3, -0.3, -0.3, -0.3, -0.3, -0.3, -0.5,
            };

    public King(int position, Color color, boolean isFirstMove) {
```

```
        super(position, color, isFirstMove);
        this.value = 10000;
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public double locationBonus(GameStage gameStage) {
        if(gameStage == GameStage.OPENING || gameStage ==
GameStage.MIDGAME)
            return this.color == Color.White ?
WHITE_KING_PREFERRED_COORDINATES_STARTMID[this.position] :
BLACK_KING_PREFERRED_COORDINATES_STARTMID[this.position];
        return this.color == Color.White ?
WHITE_KING_PREFERRED_COORDINATES_ENDING[this.position] :
WHITE_KING_PREFERRED_COORDINATES_ENDING[WHITE_KING_PREFERRED_COORDINA
TES_ENDING.length - this.position - 1];

    }

    /**
     * {@inheritDoc}
     */
    @Override
    public List<Move> getLegalMoves(Board board) {
        int possible_coordinate;
        List<Move> legalMoves = new ArrayList<>();
        for (int mask : move_mask) {
            if (isFirstColumnExtremeCase(position, mask))
                continue;
            possible_coordinate = mask + position;
            if (isValidCoordinate(possible_coordinate)) {
                if (board.board_state.get(possible_coordinate) ==
null)
                    // regular move
                    legalMoves.add(new Move.MajorMove(board, this,
possible_coordinate));
                else if (!isFriendlyPieceOnCoordinate(board,
possible_coordinate)) {
                    legalMoves.add(new Move.AttackMove(board, this,
possible_coordinate,
board.getPieceAtCoordinate(possible_coordinate)));
                }
            }
        }
        return legalMoves;

    }

    /**
     * this function use to check that a piece is not going out from
one side of the board to the other
     * because it's on first column
     * @param coordinate the coordinate the piece is in
     * @param mask the mask we check in
     * @return true if the move is illegal-false otherwise
     */
    public boolean isFirstColumnExtremeCase(int coordinate, int mask)
{
        if (coordinate % 8 == 0 && (mask == -1 || mask == -9 || mask
```

```
== 7))
            return true;
        else return ((coordinate + 1) % 8 == 0 && (mask == 1 || mask
== -7 || mask == 9));
    }
}
```

מחלקה: Board

```
package logic;

import gui.Result;
import logic.Pieces.*;
import logic.player.BlackPlayer;
import logic.player.Player;
import logic.player.WhitePlayer;

import java.util.*;
import java.util.Collections;

/**
 * this class represent a board of chess, and its attributes
 */
public class Board {
    /**
     * an HahMap of the board state
     */
    public final Map<Integer, Piece> board_state;
    private final List<Piece> whitePieces;
    private final List<Piece> blackPieces;
    private final Player whitePlayer;
    private final Player blackPlayer;
    private Player turn;
    private int movesWithoutEat;
    private final Move transitionMove;

    /**
     * Constructor for the Board class
     * @param builder a builder that contains the needed values for
the new Board
     */
    public Board(BoardBuilder builder) {
        this.board_state =
Collections.unmodifiableMap(builder.boardState);
        this.whitePieces = getActivePieces(Color.White);
        this.blackPieces = getActivePieces(Color.Black);
        List<Move> whiteLegalMoves =
getAllLegalMoves(this.whitePieces);
        List<Move> blackLegalMoves =
getAllLegalMoves(this.blackPieces);
        this.whitePlayer = new WhitePlayer(this, whiteLegalMoves,
blackLegalMoves, builder.isWhiteAi, builder.whiteHasCastled);
        this.blackPlayer = new BlackPlayer(this, whiteLegalMoves,
blackLegalMoves, builder.isBlackAi, builder.blackHasCastled);
        this.turn = getPlayerForColor(builder.turn);
        this.movesWithoutEat = builder.movesWithoutEat;
        this.transitionMove = builder.transitionMove != null ?
builder.transitionMove : Move.MoveFactory.getNullMove();

    }

    // getter
    public Player getWhitePlayer() {
        return whitePlayer;
```

```
    }

    // getter
    public Player getBlackPlayer() {
        return blackPlayer;
    }

    // getter
    public List<Piece> getWhitePieces() {
        return whitePieces;
    }

    // getter
    public List<Piece> getBlackPieces() {
        return blackPieces;
    }

    public int getMovesWithoutEat() {
        return movesWithoutEat;
    }

    public void setMovesWithoutEat(int movesWithoutEat) {
        this.movesWithoutEat = movesWithoutEat;
    }

    public Move getTransitionMove() {
        return transitionMove;
    }

    /**
     * return the player object for the given color
     * @param color color Enum
     * @return the player object of this color
     */
    public Player getPlayerForColor(Color color)
    {
        if(color == Color.White)
            return this.whitePlayer;
        else
            return this.blackPlayer;
    }

    /**
     * return the opponent of the player who it's his turn
     * @return the opponent player object
     */
    public Player getOpponent()
    {
        if(this.turn == null)
            return this.whitePlayer;
        if(this.turn.getClass() == WhitePlayer.class)
            return this.blackPlayer;
        return this.whitePlayer;
    }

    // getter
    public Player getTurn() {
        return turn;
    }

    // setter
```

```
    public void setTurn(Player turn) {
        this.turn = turn;
    }

    /**
     * look for the piece object on a given coordinate
     * @param coordinate the coordinate we look in
     * @return the piece object on the given coordinate, or null if
no piece there
     */
    public Piece getPieceAtCoordinate(int coordinate)
    {
        if(board_state.containsKey(coordinate))
            return board_state.get(coordinate);
        return null;
    }
    @Override
    public String toString() {
        return "Board{" +
                "board_state=" + board_state +
                ", whitePieces=" + whitePieces +
                ", blackPieces=" + blackPieces +
                '}';
    }

    /**
     * get a list of all the active pieces for a given color
     * @param color the color we want to find the pieces for
     * @return list of all the pieces from this color
     */
    private List<Piece> getActivePieces(Color color)
    {
        List<Piece> activePieces = new ArrayList<>();
        for (Piece piece : this.board_state.values())
        {
            if (piece != null && piece.getColor() == color)
                activePieces.add(piece);
        }
        return activePieces;
    }


    /**
     * find all the possible moves for a given list of pieces
     * @param Pieces the pieces we want to get all their legal moves
     * @return all the possible moves for this group of pieces
     */
    public List<Move> getAllLegalMoves(List<Piece> Pieces)
    {
        List<Move> possibleMoves = new ArrayList<>();
        for(Piece piece : Pieces)
        {
            possibleMoves.addAll(piece.getLegalMoves(this));
        }
        return possibleMoves;
    }

    /**
     * create the stating board
     * @return HashMap of the starting board
     */
```

```
    public static BoardBuilder createNewBoard(boolean isWhiteAi,
boolean isBlackAi)
    {
        BoardBuilder builder = new BoardBuilder();
        HashMap<Integer, Piece> board_state = new HashMap<Integer,
Piece>();
        // Black Layout
        builder.setPiece(new Rook(0, Color.Black, true));
        builder.setPiece(new Knight(1, Color.Black, true));
        builder.setPiece(new Bishop(2, Color.Black, true));
        builder.setPiece(new Queen(3, Color.Black, true));
        builder.setPiece(new King(4, Color.Black, true));
        builder.setPiece(new Bishop(5, Color.Black, true));
        builder.setPiece(new Knight(6, Color.Black, true));
        builder.setPiece(new Rook(7, Color.Black, true));
        builder.setPiece(new Pawn(8, Color.Black, true));
        builder.setPiece(new Pawn(9, Color.Black, true));
        builder.setPiece(new Pawn(10, Color.Black, true));
        builder.setPiece(new Pawn(11, Color.Black, true));
        builder.setPiece(new Pawn(12, Color.Black, true));
        builder.setPiece(new Pawn(13, Color.Black, true));
        builder.setPiece(new Pawn(14, Color.Black, true));
        builder.setPiece(new Pawn(15, Color.Black, true));
        // Empty boxes
        builder.setPiece(new Pawn(48, Color.White, true));
        builder.setPiece(new Pawn(49, Color.White, true));
        builder.setPiece(new Pawn(50, Color.White, true));
        builder.setPiece(new Pawn(51, Color.White, true));
        builder.setPiece(new Pawn(52, Color.White, true));
        builder.setPiece(new Pawn(53, Color.White, true));
        builder.setPiece(new Pawn(54, Color.White, true));
        builder.setPiece(new Pawn(55, Color.White, true));
        builder.setPiece(new Rook(56, Color.White, true));
        builder.setPiece(new Knight(57, Color.White, true));
        builder.setPiece(new Bishop(58, Color.White, true));
        builder.setPiece(new Queen(59, Color.White, true));
        builder.setPiece(new King(60, Color.White, true));
        builder.setPiece(new Bishop(61, Color.White, true));
        builder.setPiece(new Knight(62, Color.White, true));
        builder.setPiece(new Rook(63, Color.White, true));

        // turn to white
        builder.setTurn(Color.White);
        // AI values
        builder.isWhiteAi = isWhiteAi;
        builder.isBlackAi = isBlackAi;
        // has castled values
        builder.whiteHasCastled = false;
        builder.blackHasCastled = false;
        //moves without eat to 0
        builder.movesWithoutEat = 0;
        return builder;
    }

    /**
     * check if the game has been finished, and in what score if does
     * @return the Result of the game, NOT_FINISHED if the game still
going
     * @see Result Enum
     */
    public Result gameResult()
```

```
    {
        // if black gave white checkmate
        if(this.whitePlayer.isInCheckMate())
            return Result.BLACK;
        // if white gave black checkmate
        if(this.blackPlayer.isInCheckMate())
            return Result.WHITE;
        // if one of the draw conditions happens
        if(this.getTurn().isInStaleMate() ||
this.getMovesWithoutEat() == 50 || notEnoughMaterial()
        || isThreeTimesPosition())
            return Result.DRAW;
        // else-the game still going
        return Result.NOT_FINISHED;
    }

    /**
     * check if both of the players has not enough material to
win(draw case)
     * @exemple king against king, king and knight against king and
so on...
     * @return true if both can't win
     */
    private boolean notEnoughMaterial() {
        return isNotEnoughMaterialToWin(this.blackPieces) &&
isNotEnoughMaterialToWin(this.whitePieces);
    }

    /**
     * check if the active pieces are enough to win
     * @param activePieces the active pieces of a player
     * @return true if the player has not enough material to win,
else if does
     */
    private boolean isNotEnoughMaterialToWin(List<Piece>
activePieces)
    {
        boolean isPawn = false;
        int material = 0;
        for(Piece piece : activePieces)
        {
            if(piece.getClass() != King.class)
                material += piece.value;
            if(piece.getClass() == Pawn.class)
                isPawn = true;
        }
        return (material <= 3 && !isPawn);
    }

    /**
     * this function checks if we go back at the same position 3
times
     * @see <a
href="https://en.wikipedia.org/wiki/Threefold_repetition">
     * @return true if the same position returned 3 times-false
otherwise
     */
    private boolean isThreeTimesPosition()
    {
        int samePositionCounter = 0;
        // for every move in the last N moves without eat
```

```
        for(Move move : lastNMoves(movesWithoutEat))
        {
            // check if the move board was equal to the current board
            if (this.equals(move.getBoard())) {
                samePositionCounter++;
            }
        }
        // if we get back 3 times
        return samePositionCounter >= 3;
    }

    /**
     * this function return a list of the last N Moves in the game
     * @param N the number of last moves to find
     * @return list of Move of the last N moves in the game
     */
    public List<Move> lastNMoves(int N) {
        final List<Move> moveHistory = new ArrayList<>();
        Move currentMove = this.getTransitionMove();
        int i = 0;
        // while the not a nullMove and still not in N
        while(currentMove != Move.MoveFactory.getNullMove() && i < N)
{
            // add the move
            moveHistory.add(currentMove);
            // go back another board
            currentMove = currentMove.getBoard().getTransitionMove();
            i++;
        }
        // return the list
        return Collections.unmodifiableList(moveHistory);
    }

    /**
     * this Inner class is responsible for building a new board' with
his attributes
     */
    public static class BoardBuilder {

        Map<Integer, Piece> boardState;
        Color turn;
        Move transitionMove;
        boolean isWhiteAi;
        boolean isBlackAi;
        int movesWithoutEat;
        boolean whiteHasCastled;
        boolean blackHasCastled;

        public BoardBuilder() {
            this.boardState = new HashMap<>();
        }

        /**
         * put piece in the board state
         * @param piece the piece we want to set
         * @return the board builder after the piece set
         */
        public BoardBuilder setPiece(final Piece piece) {
            this.boardState.put(piece.getPosition(), piece);
            return this;
        }
```

```
        /**
         * set the turn
         * @param nextMoveMaker the color of the player we want to
change the turn to
         * @return the board builder after the turn set
         */
        public BoardBuilder setTurn(final Color nextMoveMaker) {
            this.turn = nextMoveMaker;
            return this;
        }

        /**
         * set the move transition
         * @param transitionMove the last move we made the brought us
to the current board
         * @return the board builder after the set
         */
        public BoardBuilder setMoveTransition(final Move
transitionMove) {
            this.transitionMove = transitionMove;
            return this;
        }

        /**
         * build the new board
         * @return the new board
         */
        public Board build() {
            return new Board(this);
        }
    }

    /**
     * equals override-checks if 2 board are equal
     * @param o the board we want the check if equal to this board
     * @return true if they are equal, false if not
     */
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Board b = (Board) o;
        return board_state.equals(b.board_state) &&
turn.getColor().equals(b.turn.getColor());
    }

    /**
     * hash code for the board equals
     * @return the hash of the board
     */
    @Override
    public int hashCode() {
        return Objects.hash(board_state, turn);
    }
}
```

מחלקה:Move

```
package logic;

import logic.Pieces.Piece;
import logic.Pieces.Queen;
import logic.Pieces.Rook;

/**
 * this class represent a Move on the board, and its attributes
 */
public abstract class Move {
    protected Board board;
    protected Piece pieceMoved;
    protected int coordinateMovedTo;

    /**
     * Constructor for the Move class
     * @param board the board we make the move in
     * @param pieceMoved the piece that moved in this Move
     * @param coordinateMovedTo the coordinate on the board that the
Piece moved to
     * @see Board
     */
    public Move(Board board, Piece pieceMoved, int coordinateMovedTo)
{
        this.board = board;
        this.pieceMoved = pieceMoved;
        this.coordinateMovedTo = coordinateMovedTo;
    }
    public enum MoveStatus {
        DONE, UNDONE, LEFT_IN_CHECK
    }

    // getter
    public int getCoordinateMovedTo() {
        return coordinateMovedTo;
    }

    // getter
    public Board getBoard() {
        return board;
    }

    // getter
    public Piece getPieceMoved() {
        return pieceMoved;
    }

    // is attack move
    public boolean isAttack()
    {
        return false;
    }

    // is castle move
    public boolean isCastle()
    {
        return false;
    }
```

```
    // is pawn promotion
    public boolean isPawnPromotion(){return false;}

    // is pawn threat
    public boolean isPawnThreat(){return false;}
    @Override
    public String toString() {
        return "Move{" +
                ", pieceMoved=" + pieceMoved +
                ", coordinateMovedTo=" + coordinateMovedTo +
                '}';
    }

    /**
     * this function return a board after to execute of this move
     * @return the Board object after making this move
     */
    public Board executeMove() {
        // create new builder
        final Board.BoardBuilder builder = new Board.BoardBuilder();
        // put back the unchanged attributes
        builder.isWhiteAi = board.getWhitePlayer().isAi;
        builder.isBlackAi = board.getBlackPlayer().isAi;
        builder.whiteHasCastled =
board.getWhitePlayer().isHasCastled();
        builder.blackHasCastled =
board.getBlackPlayer().isHasCastled();
        // add to the moves without eat 1
        builder.movesWithoutEat = board.getMovesWithoutEat() + 1;
        // set turn player pieces
        this.board.getTurn().getActivePieces().stream().filter(piece
-> !this.pieceMoved.equals(piece)).forEach(builder::setPiece);
        // set enemy pieces

this.board.getOpponent().getActivePieces().forEach(builder::setPiece)
;
        // clone the moved piece
        Piece piece = pieceMoved.clone();
        // move the piece
        piece.movePiece(this);
        builder.setPiece(piece);
        // change turn
        builder.setTurn(this.board.getOpponent().getColor());
        // set the move transition
        builder.setMoveTransition(this);
        return builder.build();
    }

    /**
     * this class represent a regular move
     */
    public static class MajorMove extends Move{
        public MajorMove(Board board, Piece pieceMoved, int
coordinateMovedTo) {
            super(board, pieceMoved, coordinateMovedTo);
        }

        @Override
        public boolean isAttack() {
            // not an attack
            return false;
```

```
        }
    }

    /**
     * this class represent an Attack move
     */
    public static class AttackMove extends Move{
        Piece attackedPiece;
        public AttackMove(Board board, Piece pieceMoved, int
coordinateMovedTo, Piece
                          attackedPiece) {
            super(board, pieceMoved, coordinateMovedTo);
            this.attackedPiece = attackedPiece;
        }

        /**
         * {@inheritDoc}
         */
        @Override
        public Board executeMove() {
            Board board = super.executeMove();
            // moves without eat to 0
            board.setMovesWithoutEat(0);
            return board;
        }

        public Piece getAttackedPiece() {
            return attackedPiece;
        }

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

    /**
     * this class represent a Pawn move
     */
    public static class PawnMove extends Move{

        public PawnMove(Board board, Piece pieceMoved, int
coordinateMovedTo) {
            super(board, pieceMoved, coordinateMovedTo);

        }

        /**
         * {@inheritDoc}
         */
        @Override
        public Board executeMove()
        {
            // create new board builder
            final Board.BoardBuilder builder = new
Board.BoardBuilder();
            // put back the unchanged attributes
            builder.isWhiteAi = board.getWhitePlayer().isAi;
            builder.isBlackAi = board.getBlackPlayer().isAi;
            builder.whiteHasCastled =
board.getWhitePlayer().isHasCastled();
```

```
            builder.blackHasCastled =
board.getBlackPlayer().isHasCastled();
            // pawn move initialized moves without eat
            builder.movesWithoutEat = 0;
            // set the turn player pieces

this.board.getTurn().getActivePieces().stream().filter(piece ->
!this.pieceMoved.equals(piece)).forEach(builder::setPiece);
            // set the enemy player pieces

this.board.getOpponent().getActivePieces().forEach(builder::setPiece)
;
            // clone the moved piece
            Piece piece = pieceMoved.clone();
            // move the piece
            piece.movePiece(this);
            // set the piece in the builder
            builder.setPiece(piece);
            // check if pawn promotion
            if(isLastRow(coordinateMovedTo, piece.getColor()))
                builder.setPiece(new Queen(this.coordinateMovedTo,
piece.getColor(), true));
            else
                builder.setPiece(piece);
            // change turn
            builder.setTurn(this.board.getOpponent().getColor());
            // set the move transition
            builder.setMoveTransition(this);
            // build the new board
            return builder.build();
        }

        /**
         * check if the move is pawn promotion
         * @return true if pawn promotion, false otherwise
         */
        @Override
        public boolean isPawnPromotion() {
            return isLastRow(coordinateMovedTo,
pieceMoved.getColor());
        }

        /**
         * check if the pawn threat other pieces
         */
        @Override
        public boolean isPawnThreat() {
            return board.getPieceAtCoordinate(coordinateMovedTo + (9
* pieceMoved.getColor().getDirection())) != null ||
                    board.getPieceAtCoordinate(coordinateMovedTo + (7
* pieceMoved.getColor().getDirection())) != null;

        }
    }

    /**
     * this class represent a pawn's attack move
     */
    public static class PawnAttackMove extends AttackMove{

        public PawnAttackMove(Board board, Piece pieceMoved, int
```

```
coordinateMovedTo, Piece attackedPiece) {
            super(board, pieceMoved,
coordinateMovedTo,attackedPiece);
        }
        /**
         * {@inheritDoc}
         */
        @Override
        public Board executeMove()
        {
            // create new board builder
            final Board.BoardBuilder builder = new
Board.BoardBuilder();
            // put back the unchanged attributes
            builder.isWhiteAi = board.getWhitePlayer().isAi;
            builder.isBlackAi = board.getBlackPlayer().isAi;
            // pawn move initialized moves without eat
            builder.movesWithoutEat = 0;
            builder.whiteHasCastled =
board.getWhitePlayer().isHasCastled();
            builder.blackHasCastled =
board.getBlackPlayer().isHasCastled();
            // set the turn player pieces

this.board.getTurn().getActivePieces().stream().filter(piece ->
!this.pieceMoved.equals(piece)).forEach(builder::setPiece);
            // set the enemy player pieces

this.board.getOpponent().getActivePieces().forEach(builder::setPiece)
;
            // clone the moved piece
            Piece piece = pieceMoved.clone();
            // move the piece
            piece.movePiece(this);
            // set the piece in the builder
            builder.setPiece(piece);
            // check if pawn promotion
            if(isLastRow(coordinateMovedTo, pieceMoved.getColor()))
                builder.setPiece(new Queen(this.coordinateMovedTo,
piece.getColor(), true));
            else
                builder.setPiece(piece);
            // change turn
            builder.setTurn(this.board.getOpponent().getColor());
            // set the move transition
            builder.setMoveTransition(this);
            // build the new board
            return builder.build();
        }

        @Override
        public boolean isPawnPromotion() {
            return isLastRow(coordinateMovedTo,
pieceMoved.getColor());
        }
    }
    public static final class EnPassantPawnAttackMove extends
PawnAttackMove{

        public EnPassantPawnAttackMove(Board board, Piece pieceMoved,
int coordinateMovedTo, Piece attackedPiece) {
```

```
            super(board, pieceMoved, coordinateMovedTo,
attackedPiece);
        }
    }

    /**
     * this class represent a castle move
     */
    static abstract class CastleMove extends Move{
        protected final Rook castleRook;
        protected final int castleRookStart;
        protected final int castleRookDest;

        CastleMove(Board board, Piece pieceMoved, int
coordinateMovedTo, final Rook castleRook,
                           final int castleRookStart, final int
castleRookDest) {
            super(board, pieceMoved, coordinateMovedTo);
            this.castleRook = castleRook;
            this.castleRookStart = castleRookStart;
            this.castleRookDest = castleRookDest;
        }

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

        /**
         * {@inheritDoc}
         */
        @Override
        public Board executeMove() {
            // create new board builder
            final Board.BoardBuilder builder = new
Board.BoardBuilder();
            // put back the unchanged attributes
            builder.isWhiteAi = board.getWhitePlayer().isAi;
            builder.isBlackAi = board.getBlackPlayer().isAi;
            builder.whiteHasCastled =
board.getWhitePlayer().isHasCastled();
            builder.blackHasCastled =
board.getBlackPlayer().isHasCastled();
            // put castled value
            if(board.getTurn().getColor() == Color.White)
                builder.whiteHasCastled = true;
            else
                builder.blackHasCastled = true;
            // moves without eat or pawns move + 1
            builder.movesWithoutEat = board.getMovesWithoutEat() + 1;
            // set the turn payer pieces
            for (final Piece piece :
this.board.getTurn().getActivePieces()) {
                if (!this.pieceMoved.equals(piece) &&
!this.castleRook.equals(piece)) {
                    builder.setPiece(piece);
                }
            }
            // set the enemy pieces

this.board.getOpponent().getActivePieces().forEach(builder::setPiece)
```

```
;
            // clone the moved piece
            Piece piece = pieceMoved.clone();
            // move the piece
            piece.movePiece(this);
            // set the piece to the board
            builder.setPiece(piece);
            // create new rook with the new attributes
            builder.setPiece(new Rook(this.castleRookDest,
this.castleRook.getColor(), true));
            // set the rook first move to false

builder.boardState.get(castleRookDest).setFirstMove(false);
            // change turn
            builder.setTurn(this.board.getOpponent().getColor());
            // set the move transition
            builder.setMoveTransition(this);
            // build and return the new board
            return builder.build();
        }
    }

    /**
     * this class represent a king castle move
     */
    public static final class KingSideCastleMove extends CastleMove {

        public KingSideCastleMove(Board board, Piece pieceMoved, int
coordinateMovedTo, final Rook castleRook,
                                  final int castleRookStart, final
int castleRookDest) {
            super(board, pieceMoved, coordinateMovedTo, castleRook,
castleRookStart,
                    castleRookDest);
        }

    }
    /**
     * this class represent a queen castle move
     */
    public static final class QueenSideCastleMove extends CastleMove{

        public QueenSideCastleMove(Board board, Piece pieceMoved, int
coordinateMovedTo, final Rook castleRook,
                                   final int castleRookStart, final
int castleRookDest) {
            super(board, pieceMoved, coordinateMovedTo, castleRook,
castleRookStart,
                     castleRookDest);
        }
    }
    /**
     * this class represent a Invalid Move move
     */
    public final static class InvalidMove extends Move{

        public InvalidMove() {
            super(null, null, -1);
        }

    }
```

```
    /**
     * this class is making a move objects
     */
    public static class MoveFactory {

        private static final Move invalidMove = new InvalidMove();

        public static Move getNullMove() {
            return invalidMove;
        }

        public static Move createMove(final Board board,
                                      final int currentCoordinate,
                                      final int
destinationCoordinate) {
            // check if the move exists
            for (final Move move : board.getTurn().getLegalMoves()) {
                if (move.pieceMoved.getPosition() ==
currentCoordinate &&
                        move.getCoordinateMovedTo() ==
destinationCoordinate) {
                    return move;
                }
            }
            return invalidMove;
        }
    }

    /**
     * this function check if a given coordinate is the last row for
a given color
     * @param coordinate the coordinate to check
     * @param color the color we check for
     * @return true if it is the last row, else otherwise
     * @see PawnMove
     * @see PawnAttackMove
     */
    public boolean isLastRow(int coordinate, Color color)
    {
        if(coordinate >= 0 && coordinate <= 7 && color ==
Color.White)
            return true;
        else return coordinate >= 56 && coordinate <= 63 && color ==
Color.Black;
    }



}
```

מחלקה:MoveTransition

```
package logic;

/**
 * this class represent a move transition, and the influences of a
move
 */
public class MoveTransition {
```

```
    private Board fromBoard;
    private Board toBoard;
    public Move transitionMove;
    private Move.MoveStatus moveStatus;

    /**
     * a constructor for the MoveTransition class
     * @param fromBoard the board we came from before the move
     * @param toBoard the board we go to after the move
     * @see Board
     * @param transitionMove the Move we made
     * @see Move
     * @param moveStatus the move status of the MoveTransition
     * @see logic.Move.MoveStatus
     */
    public MoveTransition(Board fromBoard, Board toBoard, Move
transitionMove, Move.MoveStatus moveStatus) {
        this.fromBoard = fromBoard;
        this.toBoard = toBoard;
        this.transitionMove = transitionMove;
        this.moveStatus = moveStatus;
    }

    // getter
    public Move.MoveStatus getMoveStatus() {
        return moveStatus;
    }

    // getter
    public Board getToBoard() {
        return toBoard;
    }

    // getter
    public Board getFromBoard() {
        return fromBoard;
    }

    // getter
    public Move getTransitionMove() {
        return transitionMove;
    }
}
```

Enum:Color

```
package logic;

/**
 * Enum of color
 */
public enum Color {
    White
            {
                public int getDirection(){return -1;}
            }
```

```
    , Black
            {
                public int getDirection(){return 1;}
            };
    abstract int getDirection();

}
```

מחלקה: Player

```
package logic.player;

import com.google.common.collect.ImmutableList;
import com.google.common.collect.Iterables;
import gui.Result;
import logic.*;
import logic.Pieces.King;
import logic.Pieces.Pawn;
import logic.Pieces.Piece;

import java.util.ArrayList;
import java.util.List;

/**
 * this is an abstract class that represent a player, and it extended
by the WhitePlayer Class
 * and the BlackPlayerClass
 * @author dotanraif
 */
public abstract class Player {
    protected Board board;
    protected List<Move> legalMoves;
    protected Piece king;
    protected boolean isInCheck;
     // is the player is AI
    public boolean isAi;
    protected boolean hasCastled;

    /**
     * A constructor for the Player class
     * @param board the board the player is in
     * @param legalMoves the legal moves of the player
     * @param enemyLegalMoves the legal moves of the player's enemy
     * @param isAI is the user choose this Player to be AI
     */
    public Player(Board board, List<Move> legalMoves, List<Move>
enemyLegalMoves, boolean isAI, boolean hasCastled) {
        this.board = board;
        this.king = findKing(board);
        this.legalMoves =
ImmutableList.copyOf(Iterables.concat(legalMoves,
calculateCastles(legalMoves, enemyLegalMoves)));
        this.isInCheck = !getAttacksOnBox(king.getPosition(),
enemyLegalMoves).isEmpty();
        this.isAi = isAI;
        this.hasCastled = hasCastled;
    }
    /**
     * find the king of the player
     * @param board the board to look on
     * @return the king of the player
     */
    protected abstract Piece findKing(Board board);

    /**
     * get the rival player
     * @return the rival player object
     */
    public abstract Player getRival();
```

```
    // getter
    public Piece getKing() {
        return king;
    }

    // getter
    public List<Move> getLegalMoves() {
        return legalMoves;
    }



    /**
     * will get every active piece for the player
     * @return list of the active pieces of the player
     */
    public abstract List<Piece> getActivePieces();


    public boolean isHasCastled() {
        return hasCastled;
    }

    public void setHasCastled(boolean hasCastled) {
        this.hasCastled = hasCastled;
    }

    public void setBoard(Board board) {
        this.board = board;
    }

    /**
     * find all the possible attack moves on a given box, and return
list of it
     * @param boxPos the coordinate of the box
     * @param moves list of moves to check
     * @return list of the possible attack moves to a box on the
board
     */
    public static List<Move> getAttacksOnBox(int boxPos, List<Move>
moves){
        List<Move> attackMoves = new ArrayList<>();
        for (Move move : moves)
        {
            if(boxPos == move.getCoordinateMovedTo())
                attackMoves.add(move);
        }
        return attackMoves;
    }
    // getter
    public boolean isInCheck(){
        return isInCheck;
    }

    /**
     * check if this player is in checkmate position
     * @return true if in check mate, false otherwise
     */
    public boolean isInCheckMate() {
        // if in check and has no moves
```

```
        return isInCheck() && !isCanEscape();
    }

    /**
     * check if this player is in stalemate position
     * @return true if in stalemate, false otherwise
     */
    public boolean isInStaleMate(){
        // if not in check but has no moves
        return !isInCheck() && !isCanEscape();
    }

    // TODO : implement method that check if the king can escape

    /**
     * check if the player has legal moves
     * @return true if it has legal moves, false otherwise
     */
    private boolean isCanEscape() {
        for (Move move : this.legalMoves){
            MoveTransition transition = makeMove(move);
            // if the move is done
            if (transition.getMoveStatus() == Move.MoveStatus.DONE)
                return true;
        }
        return false;
    }

    /**
     * the function make a given move, if the move is illegal it will
return move transition with undone or left_in_check
     * status and won't change the destination board, if the move is
legal it will return the move transition of the move.
     * @see MoveTransition
     * @param move the move to make
     * @return move transition of after the move has done
     */
    public MoveTransition makeMove(Move move) {
        // if the move is illegal
        if (!isMoveLegal(move)) {
            return new MoveTransition(this.board, this.board, move,
Move.MoveStatus.UNDONE);
        }
        // execute the move to temp board
        Board transitionBoard = move.executeMove();
        // get attacks on king
        List<Move> attacksOnKing =
getAttacksOnBox(findKing(transitionBoard).getPosition(),
transitionBoard.getTurn().legalMoves);
        // if in the new board there are any attacks on the king, the
king was left in check' and the move is illegal
        if(!attacksOnKing.isEmpty()) {
            return new MoveTransition(this.board, this.board, move,
Move.MoveStatus.LEFT_IN_CHECK);
        }
        // if the move legal, return the new Transition move
        return new MoveTransition(this.board, transitionBoard, move,
Move.MoveStatus.DONE);
    }

    /**
```

```
     * check if the move is legal
     * @param move the move to check if legal
     * @return true if move legal, false otherwise
     */
    private boolean isMoveLegal(Move move) {
        return this.legalMoves.contains(move);
    }

    /**
     * get the color of the player
     * @return the color of the player
     */
    public abstract Color getColor();

    /**
     * this find the possible castles for the player
     * @param playerLegals list of the player legal moves
     * @param opponentLegals list of the rival player legal moves
     * @return list of the possible caste moves
     */
    public abstract List<Move> calculateCastles(List<Move>
playerLegals, List<Move> opponentLegals);
}
```

מחלקה: WhitePlayer

```
package logic.player;

import logic.Board;
import logic.Color;
import logic.Move;
import logic.Pieces.King;
import logic.Pieces.Piece;
import logic.Pieces.Rook;

import java.util.ArrayList;
import java.util.List;

/**
 * this class represent a black player, and it extends the Player
class
 * @author dotanraif
 * @see logic.player.Player
 */
public class WhitePlayer extends Player{

    /**
     * {@inheritDoc}
     */
    public WhitePlayer(Board board, List<Move> whiteLegalMoves,
List<Move> blackLegalMoves, boolean isAi, boolean hasCastled) {
        super(board, whiteLegalMoves, blackLegalMoves, isAi,
hasCastled);
        this.king = findKing(board);
    }


    /**
     * {@inheritDoc}
     */
    @Override
    protected Piece findKing(Board board) {
        for (Piece piece : board.getWhitePieces()) {
            if(piece.getClass() == King.class)
                return piece;
        }
        return null;
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public Player getRival() {
        return board.getBlackPlayer();
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public List<Piece> getActivePieces() {
        return this.board.getWhitePieces();
    }
```

```
    /**
     * {@inheritDoc}
     */
    @Override
    public Color getColor() {
        return Color.White;
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public List<Move> calculateCastles(List<Move> playerLegals,
List<Move> opponentLegals) {
        List<Move> Castles = new ArrayList<>();
        if(this.king.isFirstMove() && !this.isInCheck)
        {
            // if not occupied
            if(this.board.getPieceAtCoordinate(61) == null &&
this.board.getPieceAtCoordinate(62) == null)
            {
                Piece rook = this.board.getPieceAtCoordinate(63);
                // if rook and first move
                if(rook != null && rook.getClass() == Rook.class &&
rook.isFirstMove())
                {
                    // if tiles are not under attack
                    if(Player.getAttacksOnBox(61,
opponentLegals).isEmpty()
                    && Player.getAttacksOnBox(62,
opponentLegals).isEmpty())
                    {
                        Castles.add(new
Move.KingSideCastleMove(this.board, this.king, 62, (Rook) rook, 63,
61));
                    }
                }
            }
            // same for queen castle
            if(this.board.getPieceAtCoordinate(59) == null &&
                    this.board.getPieceAtCoordinate(58) == null &&
                    this.board.getPieceAtCoordinate(57) == null)
            {
                Piece rook = this.board.getPieceAtCoordinate(56);
                if(rook != null && rook.getClass() == Rook.class &&
rook.isFirstMove()) {
                    if (Player.getAttacksOnBox(59,
opponentLegals).isEmpty()
                            && Player.getAttacksOnBox(58,
opponentLegals).isEmpty()) {
                        Castles.add(new
Move.QueenSideCastleMove(this.board, this.king, 58, (Rook)rook, 56,
59));
                    }
                }
            }
        }
        return Castles;
    }
```

```
}
```

מחלקה:BlackPlayer

```
package logic.player;

import logic.Board;
import logic.Color;
import logic.Move;
import logic.Pieces.King;
import logic.Pieces.Piece;
import logic.Pieces.Rook;

import java.util.ArrayList;
import java.util.List;

/**
 * this class represent a black player, and it extends the Player
class
 * @author dotanraif
 * @see logic.player.Player
 */
public class BlackPlayer extends Player{
    /**
     * {@inheritDoc}
     */
    public BlackPlayer(Board board, List<Move> whiteLegalMoves,
List<Move> blackLegalMoves, boolean isAi, boolean hasCasteled) {
        super(board, blackLegalMoves, whiteLegalMoves, isAi,
hasCasteled);
        this.king = findKing(board);

    }

    /**
     * {@inheritDoc}
     */
    @Override
    protected Piece findKing(Board board) {
        for (Piece piece : board.getBlackPieces()) {
            if(piece.getClass() == King.class)
                return piece;
        }
        return null;
    }

    /**
     * {@inheritDoc}
     * @return
     */
    @Override
    public Player getRival() {
        return board.getWhitePlayer();
    }

    /**
     * {@inheritDoc}
     */
    @Override
```

```
    public List<Piece> getActivePieces() {
        return this.board.getBlackPieces();
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public Color getColor() {
        return Color.Black;
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public List<Move> calculateCastles(List<Move> playerLegals,
List<Move> opponentLegals) {
        List<Move> Castles = new ArrayList<>();
        if(this.king.isFirstMove() && this.king.getPosition() == 4 &&
!this.isInCheck)
        {
            // if not occupied
            if(this.board.getPieceAtCoordinate(5) == null &&
this.board.getPieceAtCoordinate(6) == null)
            {
                Piece rook = this.board.getPieceAtCoordinate(7);
                // if rook and first move
                if(rook != null && rook.getClass() == Rook.class &&
rook.isFirstMove())
                {
                    // if tiles are not under attack
                    if(Player.getAttacksOnBox(5,
opponentLegals).isEmpty()
                            && Player.getAttacksOnBox(6,
opponentLegals).isEmpty())
                    {
                        Castles.add(new
Move.KingSideCastleMove(this.board, this.king, 6, (Rook)rook, 7, 5));
                    }
                }
            }
            // same for queen castle
            if(this.board.getPieceAtCoordinate(1) == null &&
                    this.board.getPieceAtCoordinate(2) == null &&
                    this.board.getPieceAtCoordinate(3) == null)
            {
                Piece rook = this.board.getPieceAtCoordinate(0);
                if(rook != null && rook.getClass() == Rook.class &&
rook.isFirstMove()) {
                    if (Player.getAttacksOnBox(2,
opponentLegals).isEmpty()
                            && Player.getAttacksOnBox(3,
opponentLegals).isEmpty()) {
                        Castles.add(new
Move.QueenSideCastleMove(this.board, this.king, 2, (Rook)rook, 0,
3));
                    }
                }
            }
        }
```

```
        return Castles;
    }

}
```

Package:AI

מחלקה: CenterControl

```
package logic.player.AI;

import logic.Board;
import logic.Color;
import logic.Move;
import logic.player.Player;

import java.util.List;

/**
 * this class contains static methods for evaluating a Center control
of a player
 */
public class CenterControl {
    private static final double ATTACKS_ON_ENEMY_CENTER_MULTIPLIER =
0.06;
    private static final double ATTACKS_ON_SELF_CENTER_MULTIPLIER =
0.025;
    private static final double PAWNS_ON_CENTER_MULTIPLIER = 0.08;
    private static final int[] CENTER_COORDINATE = {27, 28, 35, 36};


    public static double centerControl(Player player, Board board)
    {
        List<Move> allPlayersLegalMoves = player.getLegalMoves();
        //System.out.println("\n center: " + "\n Enemycenter:" +
calculateAttacksOnEnemyCenter(player, allPlayersLegalMoves) +
               // "\nself center: " +
calculateAttacksOnSelfCenter(player, allPlayersLegalMoves) + "\n
pawns on center: " +
               // pawnsOnCenter(player, board));
        return calculateAttacksOnEnemyCenter(player,
allPlayersLegalMoves) +
                calculateAttacksOnSelfCenter(player,
allPlayersLegalMoves) +
                pawnsOnCenter(player, board);
    }

    /**
     * this function calculate the pawns on the center evaluation
     * @param player the player we calculate his pawns on the center
evaluation
     * @param board the current board
     * @return the evaluation of the pawns in the center for the
player
     */
    private static double pawnsOnCenter(Player player, Board board) {
        int numberOfPawnsOnCenter = 0;
        for(int i : CENTER_COORDINATE)
        {
            if(board.getPieceAtCoordinate(i) != null &&
board.getPieceAtCoordinate(i).getColor() == player.getColor())
                numberOfPawnsOnCenter++;
        }
        return numberOfPawnsOnCenter * PAWNS_ON_CENTER_MULTIPLIER;
    }
```

```
    /**
     * this function calculate the attacks on the enemy center
evaluation
     * @param player the player we calculate his attacks on the enemy
center on the center evaluation
     * @param allPlayersLegalMoves all the player's legal moves
     * @return the evaluation of the attacks on enemy center the
player
     */
    public static double calculateAttacksOnEnemyCenter(Player player,
List<Move> allPlayersLegalMoves)
    {
        double attacksOnBoxVal = 0;
        if(player.getColor() == Color.White) {
            // calculate attacks on the white enemy center
            attacksOnBoxVal = Player.getAttacksOnBox(27,
allPlayersLegalMoves).size() + Player.getAttacksOnBox(28,
allPlayersLegalMoves).size();

        }
        else if(player.getColor() == Color.Black) {
            // calculate attacks on the black enemy center
            attacksOnBoxVal = Player.getAttacksOnBox(35,
allPlayersLegalMoves).size() + Player.getAttacksOnBox(36,
allPlayersLegalMoves).size();
        }
        return attacksOnBoxVal * ATTACKS_ON_ENEMY_CENTER_MULTIPLIER;
    }

    /**
     * this function calculate the attacks on the self center
evaluation
     * @param player the player we calculate his attacks on the self
center on the center evaluation
     * @param allPlayersLegalMoves all the player's legal moves
     * @return the evaluation of the attacks on self center the
player
     */
    public static double calculateAttacksOnSelfCenter(Player player,
List<Move> allPlayersLegalMoves)
    {
        double attacksOnBoxVal = 0;
        if(player.getColor() == Color.White) {
            // calculate attacks on the white self center
            attacksOnBoxVal = Player.getAttacksOnBox(35,
allPlayersLegalMoves).size() + Player.getAttacksOnBox(36,
allPlayersLegalMoves).size();

        }
        else if(player.getColor() == Color.Black) {
            // calculate attacks on the black self center
            attacksOnBoxVal = Player.getAttacksOnBox(27,
allPlayersLegalMoves).size() + Player.getAttacksOnBox(28,
allPlayersLegalMoves).size();
        }
        return attacksOnBoxVal * ATTACKS_ON_SELF_CENTER_MULTIPLIER;
    }


}
```

מחלקה: KingSafety

```
package logic.player.AI;

import logic.Board;
import logic.Color;
import logic.Move;
import logic.Pieces.King;
import logic.Pieces.Pawn;
import logic.Pieces.Piece;
import logic.player.Player;

import java.util.Collection;
import java.util.List;

/**
 * this class contains static methods for evaluating a King safety of
a player
 */
public class KingSafety {
    private static final double CASTLE_BONUS = 0.2;
    private static final double NONE_CASTLE_PUNISHMENT = -0.1;
    private static final double GOOD_PAWNS_SHIELD_BONUS = 0.3;
    private static final double BED_PAWNS_SHIELD_PUNISHMENT = -0.3;


    /**
     * evaluate the board according to king safety
     * @param player the player we evaluate for
     * @param board the current board
     * @return the value of the player's king safety if the current
board
     */
    public static double calculateKingSafety(Player player, Board
board)
    {
        return calculateCastleValue(player, board) +
calculateKingTropism(player);
    }

    /**
     * this function calculate the king tropism for the player
     * @param player the player we calculate his king tropism
     * @return the king tropism evaluation
     */
    public static double calculateKingTropism(final Player player) {
        final int playerKingSquare = player.getKing().getPosition();
        final List<Move> enemyMoves =
player.getRival().getLegalMoves();
        // the closest distance move
        int currentDistance;
        // the closest piece
        Piece closestPiece = null;
        int closestDistance = Integer.MAX_VALUE;
        for(final Move move : enemyMoves) {
            // calculate the distance between the move to the king
            currentDistance =
calculateChebyshevDistance(playerKingSquare,
move.getCoordinateMovedTo());
            if(move.getPieceMoved().getClass() != King.class &&
currentDistance < closestDistance) {
```

```
                closestDistance = currentDistance;
                closestPiece = move.getPieceMoved();
            }
        }
        if(closestPiece != null)
            return -1*(closestPiece.value / 200 * (10 -
closestDistance));
        return 0;
    }

    /**
     * this function calculate the Chebyshev Distance between the
king coordinate to a piece move coordinate
     * @param playerKingSquare the king coordinate
     * @param coordinateMovedTo the piece coordinate
     * @return return the Chebyshev distance to the king
     */
    private static int calculateChebyshevDistance(int
playerKingSquare, int coordinateMovedTo) {
        int kingColumn =  playerKingSquare % 8;
        int kingRow = playerKingSquare / 8;
        int enemyColumn = coordinateMovedTo % 8;
        int enemyRow = coordinateMovedTo / 8;

        int columnDistance = Math.abs(kingColumn - enemyColumn);
        int rowDistance = Math.abs(kingRow - enemyRow);

        return Math.max(columnDistance, rowDistance);


    }

    /**
     * this function evaluate the castle and possibility to castle of
a player
     * @param player the player we calculate castling values
     * @param board the current board
     * @return the value of castling for the player
     */
    public static double calculateCastleValue(Player player, Board
board) {
        if (player.isHasCastled()) {
            return CASTLE_BONUS + calcPawnsShield(player, board);
        }
        else if(!player.getKing().isFirstMove())
            return NONE_CASTLE_PUNISHMENT;
        return 0;
    }

    /**
     * this function calculate the king position evaluation according
to his pawn shield
     * @param player the player we calculate his king pos
     * @param board thr current board
     * @return the evaluation of the pawn shield
     */
    private static double calcPawnsShield(Player player, Board board)
{
        final int[] pawnsBestPos1 = {9, 8, 7};
        final int[] pawnsBestPos2 = {9, 8, 15};
        final int[] pawnsBestPos3 = {9, 16, 7};
```

```
        final int[] pawnsBestPos4 = {17, 8, 7};
        final int[][] pawnsBestShield = {pawnsBestPos1,
pawnsBestPos2, pawnsBestPos3, pawnsBestPos4};
        int kingCoordinate = player.getKing().getPosition();
        boolean isAllShieldTrue = true;
        // for every possible shield
        for(int[] shield : pawnsBestShield)
        {
            isAllShieldTrue = true;
            // for every spot in the shield
            for(int i : shield)
            {
                // if there is no pawn in the spot, and the king is
not if first column case with the mask
                if(board.getPieceAtCoordinate(kingCoordinate + i *
getDirection(player.getColor())) == null ||
                !(board.getPieceAtCoordinate(kingCoordinate + i *
getDirection(player.getColor())).getClass() ==
                        Pawn.class) &&

((King)player.getKing()).isFirstColumnExtremeCase(kingCoordinate, i))
                    isAllShieldTrue = false;
            }
            if(isAllShieldTrue)
                return GOOD_PAWNS_SHIELD_BONUS;
        }
        return BED_PAWNS_SHIELD_PUNISHMENT;

    }

    /**
     * get the movement direction
     * @param color the color
     * @return the direction multiplier
     */
    private static int getDirection(Color color)
    {
        if(color == Color.White)
            return -1;
        return 1;
    }
}
```

מחלקה: Material

```
package logic.player.AI;

import logic.Pieces.Bishop;
import logic.Pieces.Piece;

import java.util.List;

/**
 * this class contains static methods for evaluating a Material of a
player
 */
public class Material {
    private final static double TWO_BISHOPS_BONUS = 0.05;

    /**
     * this function evaluate the material value of the given pieces
     * @param allActivePieces the pieces we want to evaluate their
values
     * @return material evaluation for the given pieces
     */
    public static double material(List<Piece> allActivePieces)
    {
        int numberOfBishops = 0;
        double materialValue = 0;
        for(Piece piece : allActivePieces) {
            materialValue += piece.getValue();
            if (piece.getClass() == Bishop.class)
                numberOfBishops++;
        }
        return materialValue
                + (numberOfBishops == 2 ? TWO_BISHOPS_BONUS : 0);
    }
}
```

מחלקה: Mobility

```
package logic.player.AI;

import logic.Color;
import logic.Move;
import logic.Pieces.Bishop;
import logic.Pieces.Knight;
import logic.Pieces.Rook;
import logic.player.Player;

/**
 * this class contains static methods for evaluating a Mobility of a
player
 */
public class Mobility {
    private final static double ROOK_FORWARD_MOVES_MULTIPLIER = 0.02;
    private final static double ROOK_SIDE_MOVES_MULTIPLIER = 0.012;
    private final static double ROOK_BACKWARD_MOVES_MULTIPLIER =
0.007;
    private final static double BISHOP_FORWARD_MOVES_MULTIPLIER =
0.014;
    private final static double BISHOP_BACKWARD_MOVES_MULTIPLIER =
0.009;
    private final static double KNIGHT_FORWARD_MOVES_MULTIPLIER =
0.012;
    private final static double KNIGHT_BACKWARD_MOVES_MULTIPLIER =
0.005;
    /**
     * this function evaluate player mobility
     * @param player the player we calculate his mobility
     * @return evaluation of the player's mobility
     */
    public static double mobility(Player player) {
        double mobilityValue = 0;
        for(Move move : player.getLegalMoves()) {
            // according to the piece
            if (Rook.class.equals(move.getPieceMoved().getClass())) {
                mobilityValue += calcRookMoves(move, player);
            } else if
(Bishop.class.equals(move.getPieceMoved().getClass())) {
                mobilityValue += calcBishopMoves(move, player);
            } else if
(Knight.class.equals(move.getPieceMoved().getClass())) {
                mobilityValue += calcKnightMoves(move, player);
            } else
                mobilityValue += 0.001;
        }
        return mobilityValue;
    }

    /**
     * this function calculate the rook moves evaluation, according
to the rook moves forward, backward or to sides
     * @param move the move we investigate
     * @param player the player we evaluate for
     * @return the evaluation for the rook mobility
     */
    public static double calcRookMoves(Move move, Player player)
    {
        int sideMove = 0;
```

```
        int forwardMove = 0;
        int backwardMove = 0;
        if(Math.abs(move.getCoordinateMovedTo() -
move.getPieceMoved().getPosition()) <8 )
            sideMove++;
        else if((move.getCoordinateMovedTo() -
move.getPieceMoved().getPosition()) * getDirection(player) >= 8)
            forwardMove++;
        else if((move.getCoordinateMovedTo() -
move.getPieceMoved().getPosition()) * getDirection(player) <= -8)
            backwardMove++;
        return sideMove * ROOK_SIDE_MOVES_MULTIPLIER + forwardMove *
ROOK_FORWARD_MOVES_MULTIPLIER + backwardMove *
ROOK_BACKWARD_MOVES_MULTIPLIER;
    }

    /**
     * this function calculate the knight moves evaluation, according
to the knight moves forward and backward
     * @param move the move we investigate
     * @param player the player we evaluate for
     * @return the evaluation for the knight mobility
     */
    public static double calcKnightMoves(Move move, Player player)
    {
        int forwardMove = 0;
        int backwardMove = 0;
        if((move.getCoordinateMovedTo() -
move.getPieceMoved().getPosition()) * getDirection(player) >= 6)
            forwardMove++;
        else if((move.getCoordinateMovedTo() -
move.getPieceMoved().getPosition()) * getDirection(player) <= -6)
            backwardMove++;
        return forwardMove * KNIGHT_FORWARD_MOVES_MULTIPLIER +
backwardMove * KNIGHT_BACKWARD_MOVES_MULTIPLIER;
    }

    /**
     * this function calculate the bishop moves evaluation, according
to the bishop moves forward and backward
     * @param move the move we investigate
     * @param player the player we evaluate for
     * @return the evaluation for the bishop mobility
     */
    public static double calcBishopMoves(Move move, Player player)
    {
        int forwardMove = 0;
        int backwardMove = 0;
        if((move.getCoordinateMovedTo() -
move.getPieceMoved().getPosition()) * getDirection(player) >= 7)
            forwardMove++;
        else if((move.getCoordinateMovedTo() -
move.getPieceMoved().getPosition()) * getDirection(player) <= -7)
            backwardMove++;
        return forwardMove * BISHOP_FORWARD_MOVES_MULTIPLIER +
backwardMove * BISHOP_BACKWARD_MOVES_MULTIPLIER;
    }

    /**
     * get the movement direction for the player
     * @param player the player
```

```
     * @return the direction of the player
     */
    public static int getDirection(Player player)
    {
        if(player.getColor() == Color.White)
            return -1;
        return 1;
    }
}
```

מחלקה: PawnStruct

```
package logic.player.AI;

import logic.Pieces.Pawn;
import logic.Pieces.Piece;
import logic.player.Player;

import java.util.List;

/**
 * this class contains static methods for evaluating a Pawn structure
of a player
 */
public class PawnStruct {
    private static final double DOUBLE_PAWN_PUNISHMENT = -0.2;
    private static final double ISOLATED_PAWN_PUNISHMENT = -0.15;
    private static final double PAWN_ISLAND_PUNISHMENT = -0.1;

    /**
     * this function evaluate the player pawn struct, according to
pawns stack and isolated pawns
     * @param player the player we want to analyze his pawn structure
     * @param allActivePieces list of all the active pieces of the
player
     * @return double that represent the pawn structure evaluation of
the player
     */
    public static double pawnStruct(Player player, List<Piece>
allActivePieces)
    {
        int[] pawnsArr = getPawnsPlacesArray(allActivePieces);
        return calculateDoublePawns(pawnsArr) +
calculateIsolatedPawns(pawnsArr) + PawnIslands(pawnsArr);

    }

    /**
     * this function count the number of pawns of the player in every
column, and put the number in the place in the array
     * it returns
     * @param allActivePieces all the active pieces of the player
     * @return an array that represent the number of pawns in every
column
     */
    public static int[] getPawnsPlacesArray(List<Piece>
allActivePieces)
    {
        int [] pawnsArr = {0, 0, 0, 0, 0, 0, 0, 0};
        for(Piece piece: allActivePieces)
```

```
        {
            if(piece.getClass() == Pawn.class)
                pawnsArr[piece.getPosition() % 8]++;
        }
        return pawnsArr;
    }
    /**
     * this function count the pawns islands on the board, and punish
the evaluation according to it
     * @param pawnsArr array of the number of pawns in every column
of the player in the board
     * @return evaluation of the pawns Islands on the board for this
pawns
     * @see <a href="https://www.chessprogramming.org/Pawn_Islands">
     */
    private static double PawnIslands(int[] pawnsArr) {
        int pawnsIslands = 0;
        for(int i = 0;i < pawnsArr.length; i++)
        {
            if(pawnsArr[i] > 0)
            {
                while( i < pawnsArr.length && pawnsArr[i] > 0)
                    i++;
                pawnsIslands++;
            }
        }
        return pawnsIslands * PAWN_ISLAND_PUNISHMENT;
    }


    /**
     * this function count the pawns stack on the board, and punish
the evaluation according to it
     * @param pawnsArr array of the number of pawns in every column
of the player in the board
     * @return evaluation of the pawns stack on the board for this
pawns
     * @see <a
href="https://www.chessprogramming.org/Doubled_Pawn"></a>
     */
    public static double calculateDoublePawns(int[] pawnsArr)
    {
        int pawnDoubledTotal = 0;
        for (int i : pawnsArr) {
            // if more then one, for every couple add 1
            if (i > 1)
                pawnDoubledTotal += i - 1;
        }
        return pawnDoubledTotal * DOUBLE_PAWN_PUNISHMENT;
    }

    /**
     * this function count the isolated pawns on the board, and
punish the evaluation according to it
     * @param pawnsArr array of the number of pawns in every column
of the player in the board
     * @return evaluation of the isolated pawns on the board for this
pawns
     * @see <a
href="https://www.chessprogramming.org/Isolated_Pawn"></a>
     */
```

```
    public static double calculateIsolatedPawns(int[] pawnsArr)
    {
        int numIsolatedPawns = 0;
        // if isolated on rightest column
        if(pawnsArr[0] > 0 && pawnsArr[1] == 0) {
            numIsolatedPawns += pawnsArr[0];
        }
        // if isolated on leftest column
        if(pawnsArr[7] > 0 && pawnsArr[6] == 0) {
            numIsolatedPawns += pawnsArr[7];
        }
        for(int i = 1; i < pawnsArr.length - 1; i++) {
            // if isolated from both sides
            if((pawnsArr[i-1] == 0 && pawnsArr[i+1] == 0)) {
                numIsolatedPawns += pawnsArr[i];
            }
        }
        return numIsolatedPawns * ISOLATED_PAWN_PUNISHMENT;
    }

}
```

מחלקה: PieceLocation

```
package logic.player.AI;

import logic.Pieces.Piece;

import java.util.List;
/**
 * this class contains static methods for evaluating a Piece location
of a player
 */
public class PieceLocation {

    /**
     * this function evaluate the pieces' location on the board
     * @param allActivePieces all the player's active pieces
     * @param gameStage the game stage we are in
     * @return the piece location evaluation
     */
    public static double pieceLocation(List<Piece> allActivePieces,
GameStage gameStage)
    {
        double locationBonus = 0;
        for(Piece piece : allActivePieces) {
            locationBonus += piece.locationBonus(gameStage);
        }
        return locationBonus;
    }
}
```

מחלקה: RookStruct

```
package logic.player.AI;

import logic.Board;
import logic.Move;
import logic.Pieces.Piece;
import logic.Pieces.Rook;

import java.util.ArrayList;
import java.util.List;

/**
 * this class contains static methods for evaluating a Rook structure
of a player
 */
public class RookStruct {
    private static final double CONNECTED_ROW_ROOKS_BONUS = 0.1;
    private static final double CONNECTED_COLUMN_ROOKS_BONUS = 0.15;

    /**
     * this function evaluate the rook structure on the board for the
player
     * @param board the board we analyze
     * @param playerActivePieces all the player's active pieces
     * @return evaluation of the rook structure of the player
     */
    public static double rookStruct(Board board, List<Piece>
playerActivePieces)
    {
        List<Piece> rooks = findRooks(playerActivePieces);
        // if there are two rooks
        if(rooks.size() == 2) {
            Piece rook1 = rooks.get(0);
            Piece rook2 = rooks.get(1);
            return calculateConnectedRowRooks(rook1, rook2, board) +
calculateConnectedColumnRooks(rook1, rook2, board);
        }
        return 0;

    }

    /**
     * this function find the rooks on the board
     * @param playerActivePieces all the player's active pieces
     * @return list of the rooks of the player
     */
    private static List<Piece> findRooks(List<Piece>
playerActivePieces) {
        List<Piece> rooks = new ArrayList<>();
        for(Piece piece : playerActivePieces)
        {
            if(piece.getClass() == Rook.class)
                rooks.add(piece);
        }
        return rooks;
    }

    /**
     * this function evaluate the rook structure according to rooks
connected on the Column
```

```
     * @param rook1 the first rook
     * @param rook2 the second rook
     * @param board the board we are in
     * @return the bonus if the rooks are connected on column, 0
otherwise
     */
    private static double calculateConnectedColumnRooks(Piece rook1,
Piece rook2, Board board) {
        // if the rooks are on the same column
        if(rook1.getPosition() % 8 == rook2.getPosition() % 8)
        {
            // check if there are any pieces between the rooks
            for(int i = Math.min(rook1.getPosition() + 8,
rook2.getPosition());i < Math.max(rook1.getPosition(),
rook2.getPosition());i+=8)
            {
                if(board.getPieceAtCoordinate(i) != null)
                    return 0;
            }
            return CONNECTED_COLUMN_ROOKS_BONUS;
        }
        return 0;
    }

    /**
     * this function evaluate the rook structure according to rooks
connected on the Row
     * @param rook1 the first rook
     * @param rook2 the second rook
     * @param board the board we are in
     * @return the bonus if the rooks are connected on row, 0
otherwise
     */
    private static double calculateConnectedRowRooks(Piece rook1,
Piece rook2, Board board) {
        // if the rooks are on the same row
        if(rook1.getPosition() / 8 == rook2.getPosition() / 8)
        {
            // check if there are any pieces between the rooks
            for(int i = Math.min(rook1.getPosition() + 1,
rook2.getPosition());i < Math.max(rook1.getPosition(),
rook2.getPosition());i++)
            {
                if(board.getPieceAtCoordinate(i) != null)
                    return 0;
            }
            return CONNECTED_ROW_ROOKS_BONUS;
        }
        return 0;

    }



}
```

מחלקה: SortMoves

```
package logic.player.AI;

import com.google.common.collect.ComparisonChain;
import logic.Move;

import java.util.Comparator;
/**
 * this class implements the Comparator interface and used for
compare between moves
 * to make a Move ordering strategy
 */
public class SortMoves implements Comparator<Move> {
    /**
     * this function is used to sort the list of moves by their
possibility to be good
     * the function compare between the first move and the second
     * @param m1 first move
     * @param m2 second move
     * @return the int value of the compartment
     */
    @Override
    public int compare(Move m1, Move m2) {
        return ComparisonChain.start()
                .compareTrueFirst(m1.isPawnPromotion(),
m2.isPawnPromotion())
                .compareTrueFirst(m1.isAttack(), m2.isAttack())
                .compareTrueFirst(m1.isCastle(), m2.isCastle())
                .compare(m2.getPieceMoved().value,
m1.getPieceMoved().value)
                .compareTrueFirst(m1.getPieceMoved().getPosition() <
m1.getCoordinateMovedTo(),
                        m2.getPieceMoved().getPosition() <
m2.getCoordinateMovedTo())
                .result();
    }
}
```

Enum:GameStage

```
package logic.player.AI;

/**
 * Enum for game stage
 */
public enum GameStage {
    OPENING, MIDGAME, ENDING
}
```

מחלקה: PositionEvaluation

```
package logic.player.AI;

import logic.Board;
import logic.Move;
import logic.Pieces.King;
import logic.Pieces.Piece;
import logic.player.Player;

import java.util.List;
/**
 * this class contains static methods for evaluating a full position
of a player
 */
public class PositionEvaluation {
    private static final double MOBILITY_VALUE_OPENING = 1;
    private static final double MOBILITY_VALUE_MIDGAME = 0.8;
    private static final double MOBILITY_VALUE_ENDING = 0.4;
    private static final double ATTACK_MULTIPLIER = 0.01;

    /**
     * evaluate the given board, by subtracting the white player
evaluation with the black player evaluation
     * the biggest it will return-the better for white, the smallest-
better for black
     * @param board the board we evaluate
     * @return the evaluation of the board
     * @see #score(Board, Player, GameStage)
     */
    public static double evaluate(Board board)
    {
        GameStage gameStage = calculateGameStage(board);
        // the score of the white - the score of the black
        return (score(board, board.getWhitePlayer(), gameStage) -
score(board, board.getBlackPlayer(), gameStage
        ));
    }

    /**
     * this function calculate the game stage according to the
material left og the board
     * @param board the board we want to Determine his game stage
     * @return Enum of the game stage(OPENING,MIDGAME OR ENDING)
     */
    private static GameStage calculateGameStage(Board board) {
        double materialLeft = calcMaterial(board);
        if(materialLeft > 60)
            return GameStage.OPENING;
        else if(materialLeft <= 60 && materialLeft > 28)
            return GameStage.MIDGAME;
        else if(materialLeft <= 28)
            return GameStage.ENDING;
        return GameStage.OPENING;
    }

    /**
     * calculate the material on the board
     * @param board the board we want to calculate on
     * @return the material value on this board
     */
```

```
    private static double calcMaterial(Board board) {
        double materialSum = 0;
        for(Piece piece : board.board_state.values())
        {
            // if the piece is not king,add her value
            if(piece != null && piece.getClass() != King.class)
                materialSum += piece.value;

        }
        return materialSum;
    }

    /**
     * this function evaluate the score of the board for the given
player
     * @param board the board we calculate on
     * @param player the player we calculate the score for
     * @param gameStage Enum of the game stage
     * @return the evaluation of the player position on this board
     * @see GameStage
     */
    public static double score(Board board, Player player, GameStage
gameStage)
    {
        List<Piece> allActivePieces = player.getActivePieces();
        return switch (gameStage) {
            // if opening game stage
            case OPENING -> Material.material(allActivePieces) +
                    Mobility.mobility(player) *
MOBILITY_VALUE_OPENING+
                    PawnStruct.pawnStruct(player, allActivePieces) +
                    checkmate(player) + attacks(player) +
                    RookStruct.rookStruct(board, allActivePieces)+
                    CenterControl.centerControl(player, board)+
                    KingSafety.calculateKingSafety(player, board) +
                    PieceLocation.pieceLocation(allActivePieces,
GameStage.OPENING)
            ;
            // if midgame game stage
            case MIDGAME -> Material.material(allActivePieces) +
                    Mobility.mobility(player) *
MOBILITY_VALUE_MIDGAME+
                    PawnStruct.pawnStruct(player, allActivePieces) +
                    CenterControl.centerControl(player, board) +
                    checkmate(player) + attacks(player) +
                    RookStruct.rookStruct(board, allActivePieces)+
                    KingSafety.calculateKingSafety(player, board) +
                    PieceLocation.pieceLocation(allActivePieces,
GameStage.MIDGAME)

            ;
            // if ending game stage
            case ENDING -> Material.material(allActivePieces) +
                    Mobility.mobility(player) * MOBILITY_VALUE_ENDING
+
                    PawnStruct.pawnStruct(player, allActivePieces) +
                    checkmate(player) + attacks(player) +
                    CenterControl.centerControl(player, board) +
                    RookStruct.rookStruct(board, allActivePieces)+
                    KingSafety.calculateKingSafety(player, board) +
                    PieceLocation.pieceLocation(allActivePieces,
```

```
GameStage.ENDING)
            ;
        };
    }

    /**
     * this function checks if the player rival is in checkmate
     * @param player the player we check if gave checkmate
     * @return true if the rival on checkmate, false otherwise
     */
    private static double checkmate(Player player) {
        return (player.getRival().isInCheckMate() ?  10000 :  0);
    }

    /**
     * this function evaluate good eating moves(where smaller piece
eat a bigger piece)
     * @param player the player we look for good eating moves
     * @return the evaluation for good eating moves
     */
    private static double attacks(final Player player) {
        int attackScore = 0;
        for(final Move move : player.getLegalMoves()) {
            if(move instanceof Move.AttackMove) {
                final Piece movedPiece = move.getPieceMoved();
                final Piece attackedPiece = ((Move.AttackMove)
move).getAttackedPiece();
                if(movedPiece.value <= attackedPiece.value) {
                    attackScore++;
                }
            }
        }
        return attackScore * ATTACK_MULTIPLIER;
    }

    /**
     * this function prints the evaluation Details
     * @param board the board we evaluate
     * @return String of the evaluation details
     */
    public static String evaluationDetails(final Board board) {
        List<Piece> WallActivePieces = board.getWhitePieces();
        List<Piece> BallActivePieces = board.getBlackPieces();
        return
                "\ngame stage" + calculateGameStage(board) + "\n" +
                ("White:\n material: " +
Material.material(WallActivePieces) + " \nmobility:" +
        Mobility.mobility(board.getWhitePlayer()) *
MOBILITY_VALUE_OPENING+"\n pawns"+
                PawnStruct.pawnStruct(board.getWhitePlayer(),
WallActivePieces) +"\nchackmate:"+
                checkmate(board.getWhitePlayer())) + "\n attack: " +
attacks(board.getWhitePlayer()) +"\ncenter:"+
                CenterControl.centerControl(board.getWhitePlayer(),
board)+"\nrooks:"+
                RookStruct.rookStruct(board, WallActivePieces )
+"\n"+
                        "kingtro :" +
KingSafety.calculateKingTropism(board.getWhitePlayer()) + "\n" +
                        "saftey: " +
KingSafety.calculateKingSafety(board.getWhitePlayer(), board) + "\n"+
```

```
                        "PL:" +
PieceLocation.pieceLocation(WallActivePieces,
calculateGameStage(board)) +

                        "black +: \n material" +
Material.material(BallActivePieces) + "\nmobility:" +
                        Mobility.mobility(board.getBlackPlayer()) *
MOBILITY_VALUE_OPENING+"\n pawns"+
                        PawnStruct.pawnStruct(board.getBlackPlayer(),
BallActivePieces) +"\n checkmate"+
                        checkmate(board.getBlackPlayer()) +"\n
attack:" + attacks(board.getBlackPlayer()) +"\ncenter:"+
                CenterControl.centerControl(board.getBlackPlayer(),
board)+"\nrooks:"+
                RookStruct.rookStruct(board, BallActivePieces )
+"\n"+
                        "kingtro"+
KingSafety.calculateKingTropism(board.getBlackPlayer()) + "\n" +
                        "saftey: " +
KingSafety.calculateKingSafety(board.getBlackPlayer(), board) + "\n"
+
                        "casled" +
board.getBlackPlayer().isHasCastled() +
                        "PL:" +
PieceLocation.pieceLocation(BallActivePieces,
calculateGameStage(board)) +

                        "Final Score = " + evaluate(board);

    }



}
```

מחלקה: Minimax

```
package logic.player.AI;

import com.google.common.collect.Ordering;
import gui.Result;
import logic.Board;
import logic.Color;
import logic.Move;
import logic.MoveTransition;
import logic.player.Player;

import java.util.ArrayList;
import java.util.Collection;
import java.util.Collections;
import java.util.List;

/**
 * this class contains static methods for the Minimax with alpha-beta
pruning to find the best move for a player
 */
public class Minimax {
    private static int quiescenceCount = 0;
    private static final int MAX_QUIESCENCE = 5000 * 5;


    /**
     * this function uses Minimax with Alpha-Beta to find the best
move for a player in the given board
     * @param board the board we want to return the best move for
     * @param depth the depth we want to calculate the board
positions
     * @return the best move by the computer for the given board
     * @see <a href="https://www.youtube.com/watch?v=l-hh51ncgDI">
     */
    public static Move MiniMaxAB(final Board board, int depth) {
        final Player turn = board.getTurn();
        final Color color = turn.getColor();
        Move bestMove = null;
        double highestSeenValue = Double.NEGATIVE_INFINITY;
        double lowestSeenValue = Double.POSITIVE_INFINITY;
        double currentValue;
        quiescenceCount = 0;
        // sort the moves
        List<Move> SortedMoves = sortMoves(turn.getLegalMoves());
        // for every possible move
        for (final Move move : SortedMoves) {
            final MoveTransition moveTransition =
board.getTurn().makeMove(move);
            if (moveTransition.getMoveStatus() ==
Move.MoveStatus.DONE) {
                // if the move is checkmate-return him

if(moveTransition.getToBoard().getTurn().isInCheckMate())
                    return move;
                // if white turn, call min, else call max
                currentValue = color == Color.White ?
                        min(moveTransition.getToBoard(), depth - 1,
highestSeenValue, lowestSeenValue) :
                        max(moveTransition.getToBoard(), depth - 1,
highestSeenValue, lowestSeenValue);
```

```
                // if white and we found bigger
                if (color == Color.White && currentValue >
highestSeenValue) {
                    highestSeenValue = currentValue;
                    bestMove = move;
                    // if black and we found bigger
                } else if (color == Color.Black && currentValue <
lowestSeenValue) {
                    lowestSeenValue = currentValue;
                    bestMove = move;
                }
            }
        }
        return bestMove;
    }

    /**
     * sort the move by order
     * @param legalMoves the list of moves we want to sort
     * @return the sorted list of moves
     */
    private static List<Move> sortMoves(Collection<Move> legalMoves)
{
        SortMoves sortMoves = new SortMoves();
        // sort the moves by order
        return
Ordering.from(sortMoves).immutableSortedCopy(legalMoves);
    }

    /**
     * this function finds the best move for the max player
     * @param board the board we are in
     * @param depth the tree depth left
     * @param highest the highest value we have seen
     * @param lowest the lowest value we have seen
     * @return the highest move evaluation for the max player
     */
    public static double max(final Board board,
                    final int depth,
                    final double highest,
                    final double lowest) {
        // depth end of the depth search or game came to result
        if (depth == 0 ||  board.gameResult() != Result.NOT_FINISHED)
{
            // if the game is draw-return 0;
            if(board.gameResult() == Result.DRAW)
                return 0;
            // return the position evaluation
            return PositionEvaluation.evaluate(board);
        }
        double currentHighest = highest;
        // sort the moves
        List<Move> SortedMoves =
sortMoves(board.getTurn().getLegalMoves());
        // for every move
        for (final Move move : SortedMoves) {
            // make the move
            final MoveTransition moveTransition =
board.getTurn().makeMove(move);
            // if the move legal
            if (moveTransition.getMoveStatus() ==
```

```
Move.MoveStatus.DONE) {
                currentHighest = Math.max(currentHighest,
min(moveTransition.getToBoard(),

calculateQuiescenceDepth(moveTransition.getToBoard(), depth),
currentHighest, lowest));
                if (currentHighest >= lowest) {
                    return lowest;
                }
            }
        }
        return currentHighest;
    }

    /**
     * this function finds the best move for the min player
     * @param board the board we are in
     * @param depth the tree depth left
     * @param highest the highest value we have seen
     * @param lowest the lowest value we have seen
     * @return the minimum move evaluation for the min player
     */
    public static double min(final Board board,
                    final int depth,
                    final double highest,
                    final double lowest) {
        // depth end of the depth search or game came to result
        if (depth == 0 ||  board.gameResult() != Result.NOT_FINISHED)
{
            if(board.gameResult() == Result.DRAW)
                // if the game is draw-return 0;
                return 0;
            // return the position evaluation
            return PositionEvaluation.evaluate(board);
        }
        double currentLowest = lowest;
        // sort the moves
        List<Move> SortedMoves =
sortMoves(board.getTurn().getLegalMoves());
        // for every move
        for (final Move move : SortedMoves) {
            // make the move
            final MoveTransition moveTransition =
board.getTurn().makeMove(move);
            // if the move legal
            if (moveTransition.getMoveStatus() ==
Move.MoveStatus.DONE) {
                currentLowest = Math.min(currentLowest,
max(moveTransition.getToBoard(),

calculateQuiescenceDepth(moveTransition.getToBoard(), depth),
highest, currentLowest));
                if (currentLowest <= highest) {
                    return highest;
                }
            }
        }
        return currentLowest;
    }
```

```
    private static String calculateTimeTaken(final long start, final
long end) {
        final long timeTaken = (end - start) / 1000000;
        return timeTaken + " ms";
    }

    /**
     * this function calculate the quiescence needed depth
     * @param toBoard the board we move to
     * @param depth the current depth
     * @return the quiescence new depth
     */
    private static int calculateQuiescenceDepth(final Board toBoard,
                                                final int depth) {
        // if the depth is 1, and we didn't pass the Max quiescence
limit, check for non quit moves
        if(depth == 1 && quiescenceCount < MAX_QUIESCENCE) {
            int activityMeasure = 0;
            if (toBoard.getTurn().isInCheck()) {
                activityMeasure += 1;
            }

            if(toBoard.getTransitionMove().isPawnPromotion())
                activityMeasure += 2;
            if(toBoard.getTransitionMove().isAttack())
                activityMeasure += 2;
            if(toBoard.getTransitionMove().isPawnThreat())
                activityMeasure += 1;
            for(final Move move: toBoard.lastNMoves(3)) {
                if(move.isAttack()) {
                    activityMeasure += 1;
                }
            }
            // if the activity measure is bigger or equal to 2, add
depth
            if(activityMeasure >= 2) {
                quiescenceCount++;
                return 3;
            }
        }
        return depth - 1;
    }

}
```

מחלקה: GameScreen

```
package gui;

import logic.Board;
import logic.Move;
import logic.Move.MoveStatus;
import logic.MoveTransition;
import logic.Pieces.Piece;
import logic.player.AI.Minimax;
import logic.player.AI.PositionEvaluation;

import javax.imageio.ImageIO;
import javax.swing.*;
import java.awt.*;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

import static javax.swing.SwingUtilities.invokeLater;
import static javax.swing.SwingUtilities.isLeftMouseButton;
import static logic.player.AI.PositionEvaluation.evaluate;

/**
 * this function is responsible for all our gui
 */
public class GameScreen {
    private final JFrame gameFrame;
    private final BoardPanel boardPanel;

    private final Dimension TILE_DIMENSION = new Dimension(10, 10);

    private Board board;

    private final Color whiteTileColor = Color.decode("#FFFDD0");
    private final Color blackTileColor = Color.decode("#D2691E");
    private final Color greenTileColor = Color.decode("#00FF00");
    private final Color redTileColor = Color.decode("#FF0000");
    private final Color blueTileColor = Color.decode("#0000FF");

    private static int sourceTile = -1;
    private static int destTile = -1;
    private static Piece pieceMoved;

    private static boolean isWhiteAi;
    private static boolean isBlackAi;

    /**
     * our game screen definition
     */
    public GameScreen()
    {
        chooseAiOrNotPopUp();
        this.board = new Board(Board.createNewBoard(isWhiteAi,
isBlackAi));

        this.gameFrame = new JFrame("Chess");//creating instance of
```

```
JFrame
        this.gameFrame.setLayout(new BorderLayout());
        this.gameFrame.setSize(400, 400);

        this.boardPanel = new BoardPanel();
        this.gameFrame.add(this.boardPanel, BorderLayout.CENTER);

        this.gameFrame.setVisible(true);
    }

    /**
     * reset the game
     */
    public void resetGame()
    {
        chooseAiOrNotPopUp();
        this.board = new Board(Board.createNewBoard(isWhiteAi,
isBlackAi));
        this.gameFrame.setVisible(true);
    }

    /**
     * pop up that asks the user to choose if he wants to play
against Ai or not
     */
    public void chooseAiOrNotPopUp()
    {
        String[] options = {"AI", "PVP"};
        int choice = JOptionPane.showOptionDialog(null, "Choose game
Mode",
                "Game mode",
                JOptionPane.DEFAULT_OPTION,
JOptionPane.INFORMATION_MESSAGE, null, options, options[0]);
        if(choice == -1)
            System.exit(0);
        else if(choice == 0)
            chooseColor();
        else{
            isWhiteAi = false;
            isBlackAi = false;
        }
    }

    /**
     * pop up that asks the player to choose color
     */
    public void chooseColor()
    {
        String[] options = {"WHITE", "BLACK"};
        int choice = JOptionPane.showOptionDialog(null, "Choose Your
Color",
                "Choose Color",
                JOptionPane.DEFAULT_OPTION,
JOptionPane.INFORMATION_MESSAGE, null, options, options[0]);
        if(choice == -1)
            System.exit(0);
        else if(choice == 0)
        {
            isWhiteAi = false;
            isBlackAi = true;
        }
```

```
        else if(choice == 1)
        {
            isWhiteAi = true;
            isBlackAi = false;
        }
    }

    /**
     * this class represent the board panel
     */
    public class BoardPanel extends JPanel {
        private final Dimension BOARD_DIMENSION = new Dimension(400,
400);
        List<TilePanel> Tiles;

        BoardPanel(){
            super(new GridLayout(8, 8));
            this.Tiles = new ArrayList<>();
            // make a list of 64 tiles
            for (int i = 0; i < 64;i++)
            {
                TilePanel tilePanel = new TilePanel(this, i);
                this.Tiles.add(tilePanel);
                add(tilePanel);
            }
            setPreferredSize(BOARD_DIMENSION);
            validate();
        }



        /**
         * draw the board tiles
         * @param board the board to draw
         */
        public void drawBoard(Board board) {
            removeAll();
            for(TilePanel tilePanel : Tiles)
            {
                tilePanel.drawTile(board);
                add(tilePanel);
            }
            validate();
            repaint();
        }
        /**
         * game over pop up
         */
        public void gameOver(Result result)
        {
            String message;
            if(result == Result.DRAW)
                message = "draw, Want to try again?";
            else
                message = result + " won, Want to try Again?";
            String title = "Game Over";
            int userPressed = JOptionPane.showConfirmDialog(this,
message, title, JOptionPane.OK_CANCEL_OPTION);

            if (userPressed == JOptionPane.OK_OPTION)
            {
```

```
                resetGame();
            }
            else
            {
                System.exit(0);
            }
        }
    }

    /**
     * this class represent a single tile/box panel on the board
     */
    public class TilePanel extends JPanel {
        private final int tileCoordinate;
        private Move computerMove;

        TilePanel(BoardPanel boardPanel, int tileCoordinate){
            super(new GridBagLayout());
            this.tileCoordinate = tileCoordinate;
            setPreferredSize(TILE_DIMENSION);
            putTileColor();
            putTilePiece(board);
            if(board.getTurn().isAi)
                AiMove();
            addMouseListener(new MouseListener() {
                @Override
                public void mouseClicked(MouseEvent e) {
                    MoveTransition moveTransition = null;
                    if(isLeftMouseButton(e))
                    {
                        if(sourceTile == -1) {
                            // first click
                            sourceTile = tileCoordinate;
                            pieceMoved =
board.getPieceAtCoordinate(sourceTile);
                            if(pieceMoved == null){
                                sourceTile = -1;
                            }
                        }
                        else{
                            // second click
                            destTile = tileCoordinate;
                            if(destTile != sourceTile)
                            {
                                Move move =
Move.MoveFactory.createMove(board, pieceMoved.getPosition(),
destTile);
                                moveTransition =
board.getTurn().makeMove(move);
                                if (moveTransition.getMoveStatus() ==
MoveStatus.DONE) {
                                    // if the move legal, make it
                                    board =
moveTransition.getToBoard();
                                    if(board.getTurn().isAi) {
                                        // if it's the AI turn, make
an AI move
                                        AiMove();
                                    }
                                }
                            }
```

```
                                // initialize
                                sourceTile = -1;
                                destTile = -1;
                                pieceMoved = null;
                            }
                            SwingUtilities.invokeLater(new Runnable() {
                                @Override
                                public void run() {
                                    boardPanel.drawBoard(board);
                                    if(board.gameResult() !=
Result.NOT_FINISHED) {

boardPanel.gameOver(board.gameResult());
                                    }
                                }
                            });
                        }

                    }

                    @Override
                    public void mousePressed(MouseEvent e) {

                    }

                    @Override
                    public void mouseReleased(MouseEvent e) {
                    }

                    @Override
                    public void mouseEntered(MouseEvent e) {

                    }

                    @Override
                    public void mouseExited(MouseEvent e) {

                    }
                });

                validate();

            }

            /**
             * this function make an AI move for the current player
             */
            public void AiMove()
            {

System.out.println(PositionEvaluation.evaluationDetails(board));
                MoveTransition moveTransition;
                moveTransition =
board.getTurn().makeMove(Minimax.MiniMaxAB(board, 5));
                if (moveTransition.getMoveStatus() == MoveStatus.DONE) {
                    board = moveTransition.getToBoard();
                    computerMove = moveTransition.getTransitionMove();
                    if(board.gameResult() != Result.NOT_FINISHED)
                        boardPanel.gameOver(board.gameResult());
                }
```

```
System.out.println(PositionEvaluation.evaluationDetails(board));
        }

        /**
         * put the piece on the tile
         * @param board the board we put his tiles
         */
        private void putTilePiece(Board board) {
            this.removeAll();
            Piece piece = board.board_state.get(tileCoordinate);
            String PieceIconPath = "resources/";
            if(piece != null) {
                try {
                    BufferedImage image =
                            ImageIO.read(new File(PieceIconPath +
piece.getClass().getSimpleName() + piece.getColor() + ".gif"));
                    add(new JLabel(new ImageIcon(image)));
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }

        /**
         * choose the current color for a tile
         */
        private void putTileColor() {
            boolean isLight = ((tileCoordinate + tileCoordinate / 8)
% 2 == 0);
            setBackground(isLight ? whiteTileColor : blackTileColor);
            // if the tile of the chosen piece
            if(tileCoordinate == sourceTile && pieceMoved.getColor()
== board.getTurn().getColor())
                setBackground(greenTileColor);
            // if the tile is the tile of a king under check
            if(tileCoordinate ==
board.getTurn().getKing().getPosition() &&
board.getTurn().isInCheck())
                setBackground(redTileColor);

        }

        /**
         * draw the tile, his color, and the piece on it(or not)
         * @param board the board to draw his tiles
         */
        public void drawTile(Board board) {
            putTileColor();
            putTilePiece(board);
            drawPossibleMoves(board);
            validate();
            repaint();
        }

        /**
         * draw the possible moves for a chosen piece
         * @param board the board the piece is in
         */
        public void drawPossibleMoves(Board board)
        {
            if(pieceMoved != null) {
```

```
                for (Move move : board.getTurn().getLegalMoves()) {
                    MoveTransition moveTransition =
board.getTurn().makeMove(move);
                    if (pieceMoved == move.getPieceMoved() &&
move.getCoordinateMovedTo() == this.tileCoordinate
                    && board.getTurn().getLegalMoves().contains(move)
&& moveTransition.getMoveStatus() == MoveStatus.DONE) {
                        try {
                            BufferedImage image =
                                    ImageIO.read(new
File("resources/green_dot.png"));
                            add(new JLabel(new ImageIcon(image)));
                        } catch (IOException e) {
                            e.printStackTrace();
                        }

                    }
                }
            }
        }
    }
}
```

Enum: Result

```
package gui;

/**
 * Enum of the game result
 */
public enum Result {
        DRAW, BLACK, WHITE, NOT_FINISHED
}
```

Main:

```
import gui.GameScreen;

/**
 * Chess program
 * @author dotanraif
 * @version 1.0
 * @since April 2022
 */
public class Program {
    public static void main(String[] args){
        // make a new game screen
        GameScreen gameScreen = new GameScreen();
```