

מיקום ביצוע העבודה: המכללה הטכנולוגית אורט בראודה (כרמיאל)

העבודה בוצעה בהנחיית : שמעון פיטלסון

שם הסטודנט : יובל בנימין

שם הפרויקט : שחמט דו מוקדי

לתואר הנדסאי במגמת : אלקטרוניקה: אלקטרוניקה

תאריך הגשה : 21.7.2019

תוכן העניינים:

**הצעת הפרויקט..................................................................................3 הצהרת הסטודנט...............................................................................6**

**הבעת תודה.......................................................................................7**

**פרק 1 – מבוא**

**1.1 סקירה מקצועית...........................................................................8**

**1.2 הוראות הפעלה............................................................................10**

**פרק 2 – מבנה עקרוני  
2.1 המוטיבציה לביצוע הפרויקט........................................................12**

**2.2 הסבר כללי – Top View.............................................................12**

**2.3 סכמת מלבנים עקרוני..................................................................13**

**2.4 תרשים זרימה עקרוני..................................................................14**

**פרק 3 – חומרה**

**3.1 שרטוט חשמלי...........................................................................16  
3.2 פירוט רכיבים.............................................................................18**

**פרק 4 – תוכנה**

**4.1 תרשים זרימה של ה PC.............................................................19  
4.2 תרשים זרימה של ה Arduino....................................................21**

**4.3 פירוט מחלקות..........................................................................23  
4.4 קובץ List של התוכנה במחשב...................................................42**

**4.5 קובץ List של התוכנה בארדואינו................................................85**

**פרק 5 – סיכום ומסקנות**

**5.1 מסקנות....................................................................................87  
5.2 יומן עבודה................................................................................88**

**פרק 6 – נספחים**

**5.2 דפי יצרן...................................................................................91**

**הצעת נושא לפרוייקט**

# **למילוי חלקי של הדרישות לקבלת**

**תואר הנדסאי במגמת אלקטרוניקה**

## בהתמחות : מערכות תקשורת

**שם הנושא**: **שחמט דו-מוקדי**

**שם המנחה**: שמעון פיטלסון **תואר**: BCs

### בהתייחס לנאמר בחוברת " פרוייקט ועבודת גמר במסלול על תיכוני (כיתות י"ג, י"ד) במגמת

### חשמל - אלקטרוניקה (תמוז התשנ"ד - יוני 1994) ".

**אופי הפרוייקט** :

חקר הנדסי ע"פ הנאמר בפרק ד', סעיף 2.1

בדיקת התכנות, ע"פ הנאמר בפרק ד', סעיף 2.2

תכנון ע"פ הנאמר בפרק ד', סעיף 2.3

X

תכנון ופיתוח, ע"פ הנאמר בפרק ד', סעיף 2.4

תכנון, פיתוח ומימוש מערך בדיקה, ע"פ הנאמר בפרק ד', סעיף 2.5

תכנון מערך החזקה, ע"פ הנאמר בפרק ד', סעיף 2.6

**מקום הביצוע** :

X

מכללה

צה"ל

תעשייה

מוסד מחקר

**תאריך הגשת ההצעה**: \_\_\_\_\_ . **חתימת המנחה**: שמעון פיטלסון

**שם מרכז המגמה**: דיויד מור . **חתימת המרכז**

**וחותמת המכללה**: דויד מור

הצהרת הסטודנט**: לאחר שעיינתי בחוברת נוהלי ביצוע של עבודות גמר / פרוייקטים לטכנאים והנדסאים ובהצעה, ולאחר הסברי המנחה, הנני מאשר בזאת שההצעה על חלקיה מובנת לי ומחייבת אותי.**

חתימה: יובל בנימין תאריך:\_6.4.2019\_

**תיאור הנושא** **שחמט דו-מוקדי**

**שני מסכים משני מוקדים שנמצאים במרחק מסוים אחד מהשני המתקשרים לפי בקר, לכל מוקד יש מסך ועכבר. שני המוקדים משחקים שחמט לפי החוקים הקבועים של המשחק. כך, שכל רכיב יזוז לפי צורת התזוזה שלו ואפשרות אכילה שלו על רכיבים יריבים. כל מוקד יראה את הלוח שלו לפי הצבע שלו (שחור/לבן). המשחק יהיה מסונכרן לפי הבקר שיתנהג כשופט ולפי כל תזוזה של רכיב.**

**מפרט טכני** :

1. **בקר ממשפחת ARDUINO המשמש כשופט המערכת**
2. **שעון זמן אמיתי**
3. **מחשב שחקן I**
4. **מקשים וצג לבקר השופט (תיתכן מטריצת תצוגה במקום צג)**
5. **מחשב PC שחקן II**
6. **תקשורת אלחוטית (BT או XBEE)**
7. **ספק**

**פירוט הדרישות מהמבצע** :

1. הכרת המערכות הקיימות.
2. תכנון עקרוני ומפורט של יחידת הבקרה והשליטה.
3. בחירת רכיבים וציוד.
4. פיתוח חומרה/תוכנה והכרת כלי הפיתוח.
5. בניית המערכת.
6. ביצוע אינטגרציה למערכת.
7. כתיבת ספר הפרוייקט.

**ביבליוגרפיה** :

1. ***נתוני שח, רכיב זמן אמיתי***
2. ***נתוני ספרות ARDUINO של חברת ATMEL .***

**סכמת מלבנים** :

**ספק**

**תצוגת הזמן הקצוב לשחקנים**

**בקר שופט השחמט**

**תקשורת אלחוטית**

 תקשורת ל **PC**

**רכיב שעון זמן אמיתי**

**תקשורת טורית** לבקר היריב

**לוח מקשים**

**לבדיקתיות**

**צג**

**סכמת מלבנים** :

**ספק**

**מחשב PC לשחמט**

**+**

**מחשב PC**

**ליריב**

**תקשורת אלחוטית**

 תקשורת ל **PC**

**תקשורת טורית** לבקר היריב

**עכבר**

**לוח מקשים**

**עכבר**

**שם הסטודנט** יובל בנימין **ת.ז.** 207709882

**החלטת הצוות המאשר** : \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**\_\_\_\_\_\_\_\_\_** דויד מור

**תאריך** **שם וחתימת ראש הצוות המאשר**

**הצהרת הסטודנט**

אני: יובל בנימין ת.ז: 207709882

החתום מטה, מצהיר בזאת, שכל עבודת הגמר המוגשת בחוברת זו הנה פרי עבודתי בלבד, על בסיס הנחייתו של המנחה.

אני מודע לאחריות שאני מקבל על עצמי ע"י חתימתי על הצהרה זו שכל הנאמר בה הינו אמת ורק אמת.

יובל בנימין

**חתימת מגיש העבודה**

אישור המנחה:

הריני מאשר הגשת החוברת להערכה שמעון פיטלסון

**חתימת המנחה**

אישור ראש המגמה:

הריני מאשר הגשת החוברת להערכה דויד מור

**חתימת ראש המגמה**

הבעת תודה

**המנחה לפרויקט שלי שמעון פיטלסון אשר הרצאה לי בקורס 'מעבדה בהנחייה' שהעשירה לי את המידע לגבי בניית פרויקטים גדולים. בנוסף, המנחה עזר לי לתכנן ולממש את הפרויקט ודחף אותי להשקיע בפרויקט על ידי כך שבחר איתי את הרכיבים ונראות של הפרויקט ועזר לי בבעיות בפרויקט. בנוסף על כך, יש את יצחק שלמה שדאג ועזר להבין ולהביא את הרכיבים הדרושים לפרויקט.**

פרק 1: מבוא

**1.1 סקירה מקצועית**

השוואה בין 3 מערכות שחמט אחרות למשחק שחמט שלי

**1** **Lichess**

**הוא שרת שחמט באינטרנט(כלומר, הוא אתר שחמט שחמט). כל אחד יכול לשחק בעילום שם באתר למרות שאפשר להירשם לחשבון באתר כדי לשחק משחקים מדורגים. כל התכונות והמאפיינים של אתר השחמט זמינות בחינם כאשר האתר ממומן על יד תרומות.**

**אתר זה מאפשר למשתמשים לשחק משחקים בזמן אמת ובהתכתבות(**שחמט בהתכתבות הוא שחמט המשוחק על ידי התכתבות בין שחקנים מרוחקים, בדרך כלל על פני תקופה ממושכת.**) נגד שחקנים בפקדי זמן שונים(כלומר בכל משחק אפשר לבחור כמה זמן לכל שחקן יש לשחק) ותומכת ויראציות שונות של משחק שחמט(למשחק שחמט המציאו גרסאות רבות ומגוונות המתבססות על חוקי השחמט אך נוספו חוקיים מיוחדים על מנת לתבל את המשחק).**

**תכונות השונים בין המערכת לפרויקט**

**בעוד שהמערכת היא שרת שחמט שעובדת דרך אתר אינטרנט, יש לה שיטת דירוג שחקנים ותומכת ויראציות שונות של משחק שחמט. המשחק שחמט שלי עובד דרך תוכנה שצריך להוריד דרך שני מחשבים וחומרה שצריך לקנות רק כדי לשחק שחמט עם שני אנשים. בנוסף, אין לשחמט שלי שיטת דירוג וויראציות שונות.**

**תכונות הדומים בין המערכת לפרויקט**

**בדומה לפרויקט שלי המערכת מאפשרת משחקי שחמט בין שני אנשים מרוחקים ומאפשרת משחקים בזמן אמת ובהתכתבות(כל עוד התוכנה עדיין מופעלת). בנוסף, המערכת והפרויקט מאפשרים למשתמשים לשחק נגד שחקנים בפקדי זמן מסוימים.**

**2 Chess.com**

**הוא שרת שחמט באינטרנט, פורום שחמט ורשת חברתית(וגם שם החברה). זהו האתר הנפוץ ביותר לשחמט שמשתמשים מבקרים. המבקרים באתר יכולים לשחק על שרת שחמט בזמן אמת ומשחקים בסגנון התכתבות. השחקנים יכולים לשחק נגד מנועי שחמט(מחשב שמשחק נגדם). בנוסף לכך, המערכת תומכת בויראצות שונות של משחק שחמט(למשל משחק שחמט של 4 שחקנים). החברה מארחת עשרות טורנירי שחמט(שהזוכה מקבל כסף) של שחקנים מכל העולם.**

**תכונות השונים בין המערכת לפרויקט**

**בעוד שהמערכת היא שרת שחמט, פורום שחמט ורשת חברתית וגם חברה. בנוסף, המערכת תומכת בויראציות שונות של משחק שחמט ומארחת טורנירים עם פרסים. למשחק שחמט שלי אין פורום ואין רשת חברתית ולא תומכת במשחק שחמט של 4 שחקנים.**

**תכונות הדומים בין המערכת לפרויקט**

**בדומה לפרויקט שלי המערכת מאפשרת משחקים בזמן אמת ובסגנון התכתבות. בנוסף, יש תכונה העוזרת לשחקן לבחור כלי שחמט מסוים ולראות איזה מהלכים חוקיים יש לו בעזרת סימן של נקודות ירוקות על מיקום היעד שבו כלי השחמט יכול לזוז בהתאם למצב המשחק.**

**3 Automated chess board**

**כמו משחק השחמט 'Wizard Chess' מסדרת הארי פוטר. מערכת זו היא משחק שחמט שעובדת על לוח שחמט פיזי. כלי המשחק מבצעים מהלכים באמצעות מחשב אחד ועל הלוח הפיזי הם זזים לפי מגנטים מתחת ללוח שמזיזים אותם למיקום היעד שנבחר במחשב.**

**תכונות השונים בין המערכת לפרויקט**

**בעוד שמערכת זו פועל עם לוח שחמט וכלי משחק "אמיתיים" ומבצעת מהלכים בעזרת מערכת אלקטרומגנטית ומחשב אחד לבחירת המהלך. הפרויקט שלי עובד עם שני מחשבים ותוכנה על שניהם, אין לי לוח שחמט פיזי וכלי משחק פיזיים.**

**תכונות הדומים בין המערכת לפרויקט**

**בדומה למערכת ולפרויקט שלי בעזרת המחשב אפשר לבצע את מהלכים ללוח השחמט. בנוסף, שניהם מאפשרים משחקים בסגנון התכתבות ובזמן אמת.**

טבלת השוואה

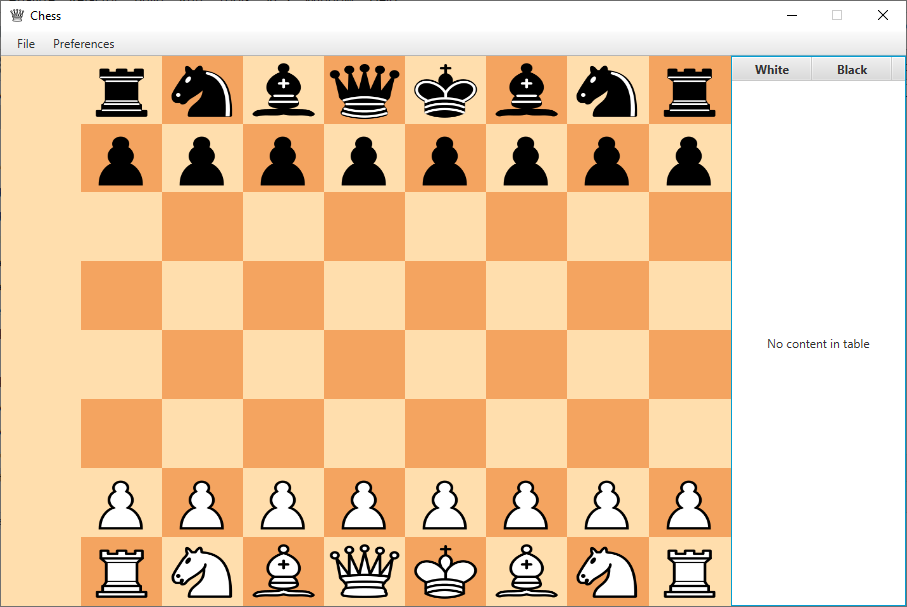
|  |  |  |  |
| --- | --- | --- | --- |
| **נושא/מערכת** | **Lichess** | **Chess.com** | **Automated chess board** |
| **הסבר כללי** | שרת שחמט הפועל באינטרנט המאפשר לשחק שחמט דרך אפליקצה או האתר ולוח השחמט מיוצג בממשק גרפי. המערכת תומכת ויראציות שונות של משחקים. | מערכת השחמט הפופולרית בעולם שפועלת בעזרת שרתי שחמט באינטרנט המאפשרת לשחק שחמט מכל העולם ובקלות רבה. בנוסף, גם לה יש אפליקציה ואתר מיוחד משלה. החברה שמפעילה את המערכת מנחה אלפי תחרויות כל שנה תמורת פרס(כסף). | מערכת שפועלת על לוח שחמט פיזי. והזזת מהלכים בעזרת מחשב בקרה השולט על לוח המשחק. למערכת זו אין אפליקציה או ממשק גרפי. |
| **משמשת ליותר משני שחקנים** | מערכת זו תומכת באמצעות השרתים שלה שיותר ממשחק אחד יתקיים באותו זמן. | מערכת זו גם תומכת במשחקים שיתקיימו באותו זמן. בנוסף, היא מפעילה גם וריאציה שבו מתקיים משחק שחמט של 4 שחקנים. | מערכת זו רק תומכת משחק אחד בזמן מסויים. |
| **שימוש בלוח שחמט פיזי** | לא משתמשת בלוח שחמט פיזי, כדי שכמה שחקנים יוכלו לשחק באותו זמן. | לא משתמשת בלוח שחמט פיזי, כדי שיהיה שימוש קל לאלפי שחקנים לשחק. | מערכת זו תומכת בלוח פיזי כדי לדמות את משחק השחמט מסדרת הארי פוטר. |
| **שימוש למטרות רווח** | מערכת זו הוקמה על בסיס תרומות כדי לספק את הצרכן. | החברה הוציאה את המערכת הזו על מנת לספק את הצרכן ולקבל רווחים ממנה. לכן לא כל תכונות האתר חינמיות. | מערכת זו היא פרויקט עצמי למען מימוש היכולות האישיות. |

**1.2 מדריך למשתמש**

**בכדי להריץ את תוכנת הפרויקט יש להיכנס למיקום הקובץ Chess.jar ולהפעיל אותו.**

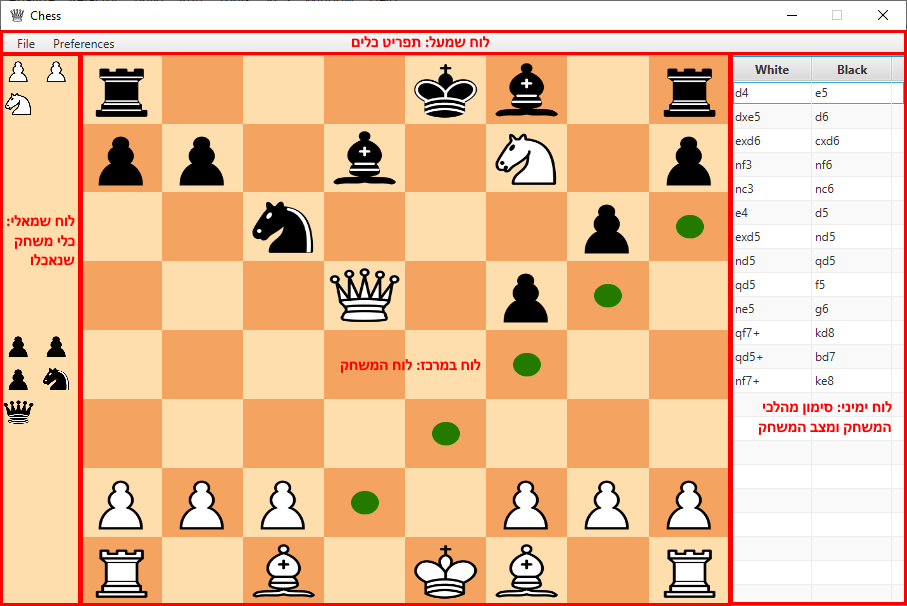
**\*תמונת המראות הגעת למיקום הקובץ**

**לאחר הפעלת תוכנת הפרויקט, יפתח הפרויקט:**



**זהו מסך המשחק והוא מורכב מכמה לוחות:**

1. **הלוח במרכז המייצג את לוח השחמט ואת כלי המשחק.**
2. **הלוח שמעל כל הלוחות הוא תפריט הכלים להוספה או שינוי משתני המשחק.**
3. **הלוח מצד ימין מייצג בצורת טקסט את סימון מהלכי המשחק ואת מצב המשחק(סימוני הזזת כלי משחק, אכילה, הגעת למצב שח/שחמט ועוד).**
4. **הלוח מצד שמאל מייצג בצורת תמונות את כלי המשחק שנאכלו בזמן המשחק לכל שחקן.**



הוראות המשחק

**שחמט הוא משחק לוח של שני שחקנים המורכב מלוח שחמט 8x8 ו16 כלי משחק מתוכם 6 סוגי כלי משחק שונים עבור כל שחקן. כל סוג של כלי משחק נע בצורה יחודית.**

**מטרת המשחק היא לעשות שחמט למלך היריב. אך, המשחק לא חייב להסתיים בשחמט הוא גם יכול להסתיים בתיקו. כלומר, כאשר תורו של השחקן לבצע מהלך והמלך שלו לא נמצא בשח**

**אבל אין לו מהלכים חוקיים לבצע(מכונה בשם פט).**

הזזת כלי משחק

**כאשר התור הוא שלך באמצעות העכבר בוחרים בלחיצה הראשונה את הכלי שרוצים להזיז ובלחיצה השניה בוחרים את הנקודה שרוצים להזיז את הכלי שבחרנו לשם. אם המהלך חוקי (לפי חוקי הכלי משחק ולא מביט לשח/שחמט של המלך של השחקן) אז המשחק מעביר את הכלי לנקודה זו. בנוסף, כדי לבטל בחירת כלי משחק(לחיצה ראשונה) יש ללחוץ על מקש ימני.**

* **ישנה אופציה בתפריט הכלים** **לראות את המהלכים החוקיים של כלי מסויים בעט בחירתו והמהלכים האלו מסומנים בעיגולים ירוקים.**

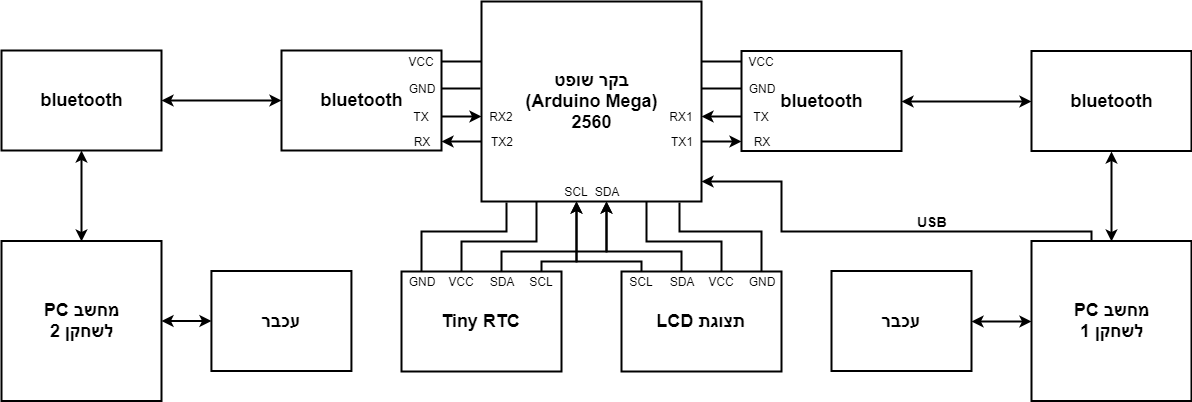
פרק 2 – מבנה עקרוני

**2.1 מוטיבציה לפרויקט**

**מאז התיכון שבו היה לי פרויקט יחסית גדול לא היה לי פרויקט כזה מאתגר וגדול. הייתי צריך לתכנן הרבה ולעבור דרך הרבה מאוד בעיות. אבל בסופו של דבר העבודה הקשה השתלמה. בסמסטר הקודם המשחק שחמט היה בין הדברים היחידים שעניינו אותי.הייתי משחק שעות ובזמן הפנוי הייתי לומד תכנות כי היה אפשר לעשות עם התוכנה הרבה פרויקטים מגניבים. לכן, כאשר היינו צריכים לבחור על נושא לפרויקט גמר, אני ידעתי איזה פרויקט אני ארצה לעשות. הפרויקט היה מאתגר מאוד והיה מאוד כיף למצוא פתרון כמעט לכל הבעיות. לסיכום, אני נהנתי לעבוד על הפרויקט וגם למדתי הרבה לגבי בניית פרויקטים ושחמט.**

**2.2 הסבר כללי -** Top View

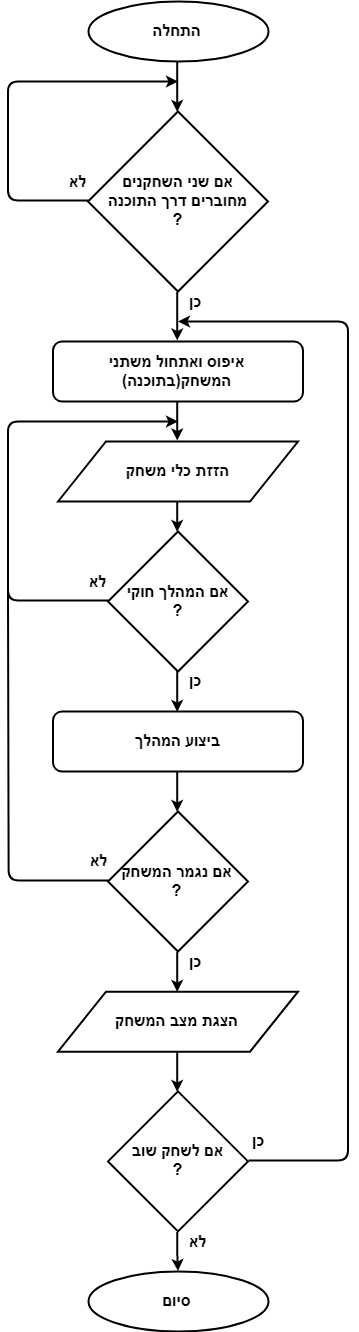
**המערכת של המשחק שחמט תפעל כך שלכל מחשב יהיה תוכנה העובדת ביחד עם המחשב השני כאשר מעבירים מידע אחד לשני באמצעות תקן תקשורת (Bluetooth) דרך הבקר. הבקר המקבל מידע מהמחשבים ומתנהג כשופט. הבקר מפעיל את השעון ומציג דרך ה LCD את הזמן שיש לכל שחקן במשחק ואת מצב המשחק. כאשר, שני המחשבים מחוברים דרך התקן התקשורת, המשחק יכול להתחיל. הממשק הגרפי והביצוע מהלכי כלי המשחק מבוצע דרך התוכנה שבמחשב. כך, שהתוכנה מתחשבת בחוקי משחק השחמט.**

**2.3 סכמת מלבנים עקרוני –** Block Diagram

הסבר לסכמת מלבנים:

**הבקר השופט(הוא ארדואינו מגה 2560) מסנכרן מידע (שליחת מידע בין שני המחשבים ומידע המכיל מהלכי כלי משחק מנקודה נוכחית למיקום היעד.)בין שני מחשבים לגבי המהלכים של כל שחקן, כותב ל LCD מידע(איזה שחקן ניצח ואת מצב המשחק), קולט מה RTC את הזמן הנוכחי (כדי לחשב ולהציג את הזמן שנאשר לשחקן לבצע מהלכים במשחק בLCD) העובר בין מחשבים באמצעות Bluetooth. בנוסף, הבקר מציג ל LCD את הזמן שיש לכל שחקן במשחק ואת מצב המשחק (מי ניצח, האם יש תיקו ועוד). רכיב הRTC שהוא שעון זמן אמת הדרוש כדי לדעת באופן מדויק את הזמן לכל שחקן לאורך כל המשחק. רכיב הBluetooth עוזר לשדר ולקבל מידע אלחוטי לטווחים קצרים בין שני המחשבים והבקר.**

**2.4 תרשים זרימה עקרוני –** Flow Chart



הסבר לתרשים זרימה:

**בהתחלה שני המשתמשים נכנסים לתוכנה במחשב ולוחצים על כפתור להתחבר לשחקן השני. כלומר, התוכנה בודקת אם שני המחשבים מסוגלים לשדר לבקר ולשדר אחד לשני ומראה אם הם מחוברים. לכן, אם הם מחוברים נאפס את משתני המשחק בתוכנה(לעשות איפוס ללוח המשחק בשני המחשבים) ונציג את לוח המשחק בממשק הגרפי של כל מחשב. ואם הם לא מחוברים אז נמשיך לנסות לשדר לבקר ולשני המחשבים.**

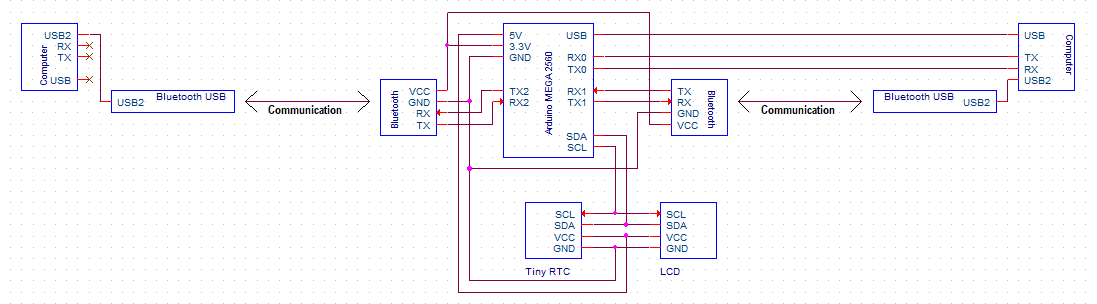
**כאשר שחקן מנסה להזיז כלי משחק, המהלך נבדק במחשב PC 1 ונשלח דרך הבקר ונבדק במחשב PC 2 אם לשני המחשבים המהלך חוקי כך שנבדוק אותו בארדואינו אז נבצע את המהלך בשני המחשבים. ואם המהלך לא חוקי לא נבצע את המהלך והשחקן יצטרך לבצע את המהלך החוקי שירצה שוב. לפי זאת, המשחק ממשיך להתקיים כאשר שחקן בתורו מבצע עוד מהלך, התוכנה בשני המחשבים בודקת שהוא חוקי ואם הוא חוקי התוכנה מבצעת אותו בשני המחשבים עד שמסתיים המשחק.**

**וכאשר ישנו מצב בו אחד מהשחקנים ניצח במשחק נשלח לארדואינו את השחקן המנצח(או אם המצב הוא תיקו נשלח זאת לארדואינו) והוא יציג את המנצח בLCD.**

**לאחר מכן, נבדוק אם שני השחקנים ירצו לשחק שוב אז נשלח את התשובות שלהם לארדואינו ונבדוק בארדואינו אם להתחיל את המשחק שוב. ואם שני השחקנים רוצים לשחק שוב נבצע שוב איפוס למשחק בשני המחשבים והשחקנים יוכלו לשחק שוב. לעומת זאת, אם לא ירצו לשחק שוב לא נבצע איפוס והמשחק ישאר במצב IDLE(מצב בו מחכים לתגובת המשתמש) עד שהשחקן יצא מהתוכנית או ירצה לשחק שוב.**

פרק 3 – החומרה

**3.1 שרטוט חשמלי מלא**



הסבר סרטוט חשמלי:

**המערכת מעבירה מידע בין מחשב למחשב דרך הארדואינו. כלומר, המידע נשלח באמצעות תוכנה דרך רכיב Bluetooth USB המחובר ישירות למחשב. מידע זה נשלח בתקשורת אלחוטית, הנשלחים לרכיב Bluetooth. לאחר מכן, המידע עובר מרכיב Bluetooth ל Arduino דרך פינים TX וRX לפינים TX2 ו RX2 של ה Arduino. הדבר קורה גם עבור המחשב השני. אך, המידע העובר מ Blueooth לArduino הוא דרך פינים TX1 וRX1 של ה Arduino. בנוסף, מחובר רכיב TinyRTC שמטרתו לקלוט את הזמן הנוכחי כדי לבדוק כמה זמן עבר מתחילת המשחק. יתר על כן, מחובר רכיב הLCD כדי להציג את הזמנים שנשאר לכל שחקן למשחק ואת מצב המשחק.**

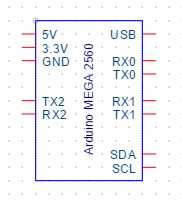
חיבור הרכיבים:

**מחשב 1(הימיני) מחובר לבקר Arduino ולBluetoothUSB דרך הפינים הבאים:**

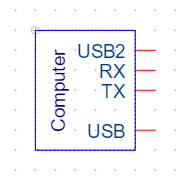
1. **USB כד לספק מתח לבקר.**
2. **TX וRX כדי לשדר ולקלוט מידע מהמחשב לבקר.**
3. **USB2 כדי להתחבר להתקן BluetoothUSB.**

**למחשב 2(השמאלי) מחובר רק USB2 כד להתחבר להתקן BluetoothUSB מכיוון שאין צורך בשאר הפינים.**

**הבקר:**



**המחשב:**

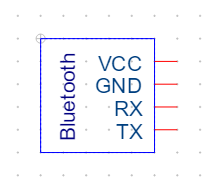


**התקן Bluetooth USB Mini**



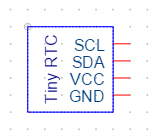
**Bluetooth RN-41 מחובר לבקר דרך הפינים הבאים:**

1. **TX וRX כדי לשדר ולקלוט מידע הנשלח דרך הBluetooth ממחשב לבקר.**
2. **GND ו VCC כדי לספק מתח V3.3 ואדמה לרכיב.**



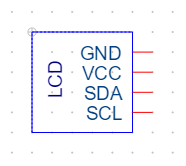
**TinyRTC מחובר לבקר דרך הפינים הבאים:**

1. **SCL שעון I2C עבור שעון זמן אמת.**
2. **SDL מידע I2C עבור שעון זמן אמת.**
3. **GND ו VCC כדי לספק מתח V5 ואדמה לרכיב.**



**LCD 2004a מחובר לבקר דרך הפינים הבאים:**

1. **SCL שעון I2C עבור שעון זמן אמת.**
2. **SDL מידע I2C עבור שעון זמן אמת.**
3. **GND ו VCC כדי לספק מתח V5 ואדמה לרכיב.**



**3.2 פירוט רכבים**

מיקרו בקר ארדואינו מגה 2560:

**בקר הארדואינו מובנה לפרויקטים אשר צריכים יותר פינים של כניסות ויציאות ויותר זיכרון של סקיצת תוכנית הארדואינו. כלומר, לבקר יש 54 כניסות ויציאות דיגיטליות ויש לו 16 כניסות ויציאות אנלוגיות. בנוסף, המיקרו בקר מתוכנת באמצעות התוכנה ארדואינו.**

**TinyRTC**:

**מודל RTC זה מבוסס על שבב השעון ds1307 התומך בפרוטוקול I2C. בנוסף, השעון/לוח שנה מספק שניות, דקות, שעות ותאריך הכולל יום חודש ושנה.**

**LCD-2004a**:

**צג גביש נוזלי(LCD) הוא מודל עם ממשק מהירות גבוהה I2C. בנוסף, הוא מסוגל להציג 20x4 תווים(4 שורות ו20 עמודות).**

**לכל התקן LCD המתשמש בפרוטוקול I2C יש כתובת הקסדצימלי שהוא בין (0x20-0x27).**

**I2C**:

**הוא פרוטוקול המיועד לאפשר למספר שבבי "slaves" לתקשר עם אחד או יותר שבבי "Master". ואפשרות לבחירת slave באמצעות כתובת שהוא ערך הקסדצימלי שלו.**

**כאשר scl הוא קו השעון ו sda הוא קו הנתונים.**

**Bluetooth RN-41**:

**רכיב זה מסוגל לשלוח ולקבל נתונים לכל רכיב Bluetooth RN-41 אחר. כלומר, הוא מושלם עבור החלפה ישירה של ממשק טורי אסינכרוני טורי. אך, תקשורת אלחוטית לא תעבוד כאשר אלא אם כן יש לך שני התקנים המסוגלים לדבר אחד עם השני מודמי Bluetooth יכולים לדבר עם כל התקן Bluetooth אחר התומך spp(רשימה ארוכה של רכיבים Bluetooth שיכולים להתחבר להתקני Bluetooth). כלומר, רכיבים אלו יכולים להיות התקני Bluetooth RN-41 או RN-42 או אפילו פלאפון חכם. אם למחשב אין התקן Bluetooth משלו אפשר לחבר Bluetooth USB Module(רכיב שהשתמשנו בפרויקט).**

**לכל התקן Bluetooth יש כתובת מיוחדת המוצגת בדרך כלל כערך הקסדצימלי.**

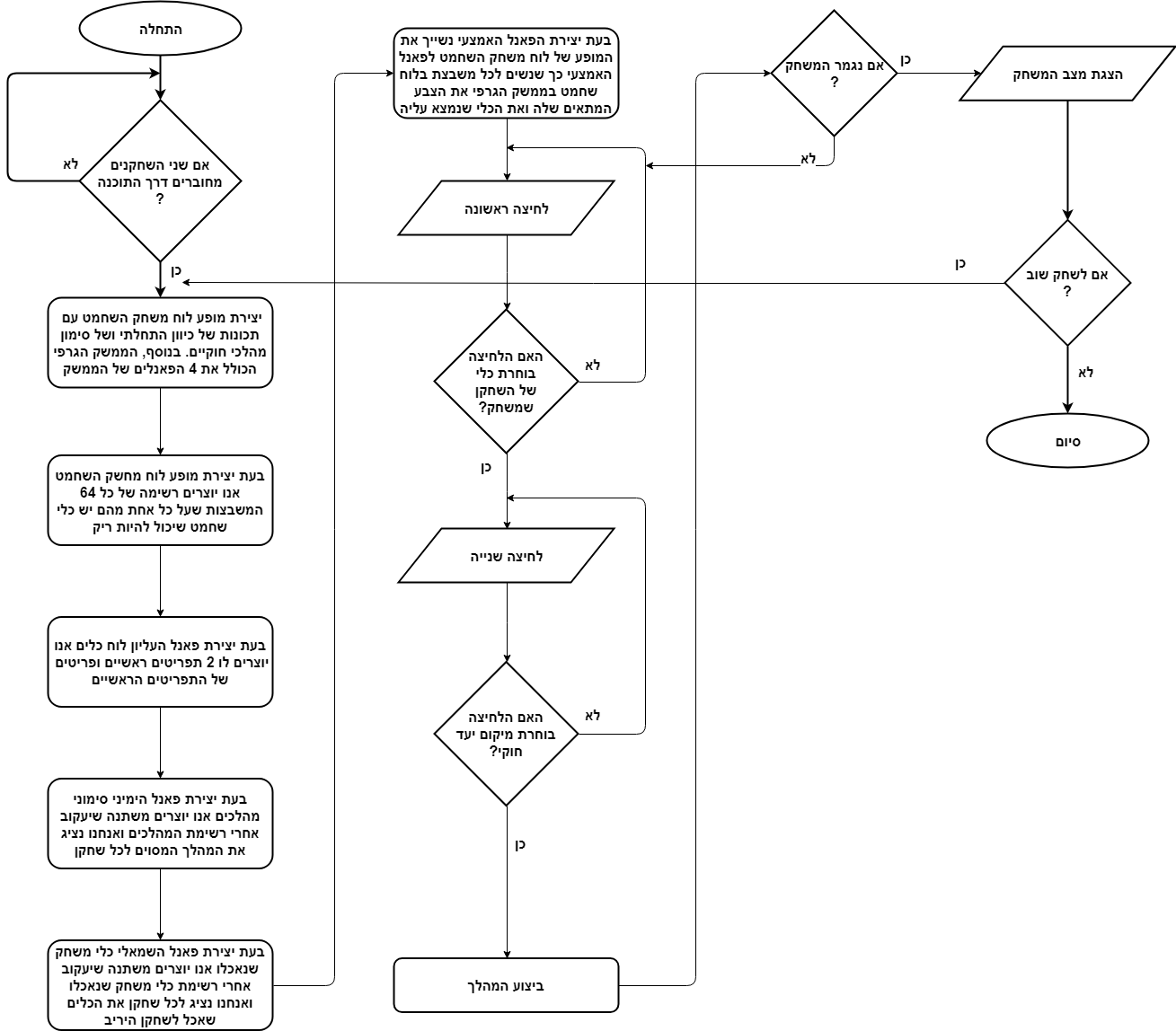
**BluetoothUSB**:

**התקן Bluetooth המאפשר להתקני Bluetooth-RN41 או RN-42 להתחבר אליהם.**

פרק 4 – תכנה

שפת שימוש: **Java**

שימוש בספריות: **JavaFX לממשק הגרפי, וב Guava.**

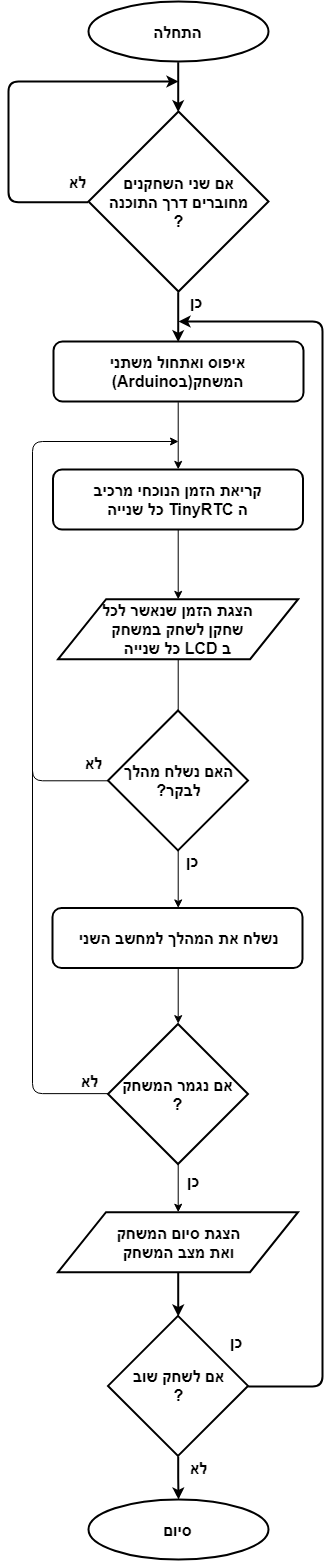
**4.1 תרשים זרימה של התוכנית בPC**

הסבר תרשים זרימה של ה**PC** :

**בהתחלה המערכת בודקת שני השחקנים מחוברים ומצליחים להעביר תקשורת ביניהם. לאחר מכן, נאתחל ונאפס את נתוני המערכת וניצור את 4 הפאנלים של הממשק הגרפי ואת מופע לוח המשחק המדומה(אוביקט שיש עליו את כלי המשחק המבצעים מהלכים). כעת לאחר שהתחלנו משחק כאשר שחקן מנסה לבצע מהלך על כלי המסויים הוא צריך בלחיצה הראשונה ללחוץ על כלי שחק מסויים שלו ובלחיצה השנייה על מיקום יעד חוקי(מיקום היעד שכלי המשחק יכול לזוז בהתאם לחוקי כלי המשחק ובהתאם למצב המשחק).**

**לאחר ביצוע המהלך נשלח את המהלך לבקר שישלח למחשב השני ונבדוק האם המשחק נגמר. אם המשחק לא נגמר נחכה ללחיצה ראשונה שוב של השחקן היריב. לעומת זאת, אם המשחק נגמר נציג את מצב המשחק ונבדוק אם השחקנים רוצים לשחק שוב. אם כן נאתחל את המשחק שוב וניצור שוב את 4 הפאנלים של הממשק הגרפי ואת מופע לוח המשחק המודמה. ואם לא נסיים את המשחק.**

**4.2 תרשים זרימה של התוכנית בArduino**



הסבר תרשים זרימה של ה**Arduino** :

**בהתחלה נבדוק ששני השחקנים מחוברים ומתקשרים ביניהם דרך הבקר. לאחר מכן, נאפס ונאתחל את משתני המשחק. כעת, כאשר מתחילים את המשחק כל שנייה נבדוק ב TinyRTC את הזמן הנוכחי, נחשב את הזמן שנשאר לכל שחקן לשחק במשחק ונציג על מסך ה LCD את הזמן הזה. במקביל, נבדוק אם התבצע מהלך בלוח ונשלח לבקר. אם כן נשלח את המהלך למחשב השני שיבצע אותו בלוח שלו. לאחר מכן, נבדוק אם נגמר המשחק(אם שני הלוחות הגיע למצב של שחמט או פט). אם לא, נחזור על התהליך של בדיקת הזמן שנשר לשחקן והצגתו ובדיקה אם התבצע מהלך. אם כן, נציג על המסך את סיום המשחק ואת מצב המשחק. לפי זאת, נבדוק אם שני השחקנים ירצו לשחק שוב. אם כן, נאפס ונאתחל את משתני המשחק ונתחיל את המשחק מהתחלה. אם לא נסיים את המשחק.**

**4.3 פירוט מחלקות**

|  |  |  |
| --- | --- | --- |
| שם המחלקה | פרמטרים | פירוט |
| Tile |  | הורשה: איןמחלקה המייצגת משבצת כללית של לוח השחמט. |
| שם הפונקציה |  |  |
| Tile | **int** tileCoordinate | פונקציית הבנאי המעדכנת את מיקום המשבצת. |
| getTileCoordinate | **אין** | פונקציה המחזירה את מיקום המשבצת. |
| createAllPossibleEmptyTiles | **אין** | פונקציה היוצרת מפה של 64 משבצות ריקות(Empty Tiles) מ0 ל63. |
| createTile | **int** tileCoordinate, Piece piece | פונקציה היוצרת משבצת שיש עליה כלי משחק(OccupiedTile) ואם אין כלי משחק לוקחת את המשבצת מהמפה של משבצות ריקות, במיקום מסויים בלוח המשחק. |
| EmptyTile |  | הורשה: Tileמחלקה המייצגת משבצת של לוח שחמט ללא כלי שחמט. |
| EmptyTile | **int** tileCoordinate | פונקציית הבנאי המעדכנת את מיקום המשבצת. |
| isTileOccupied | אין | פונקציה המחזירה אם למשבצת יש כלי משחק עליה(תמיד מחזירה לא, כי זו משבצת ריקה). |
| getPiece | אין | פונקציה המחזירה את הכלי משחק שעל המשבצת(תמיד תחזיר null, כי זו משבצת ריקה). |
| toString | אין | פונקציה המחזירה את הסימון של משבצת ריקה בצורת טקסט. |
| OccupiedTile |  | הורשה: Tileמחלקה המייצגת משבצת של לוח שחמט עם כלי שחמט. |
| OccupiedTile | **int** tileCoordinate, Piece piece | פונקציית הבנאי המעדכנת את מיקום המשבצת ואת כלי המשחק שעל המשבצת. |
| isTileOccupied | אין | פונקציה המחזירה אם למשבצת יש כלי משחק עליה(תמיד מחזירה כן, כי זו משבצת שתפוסה). |
| getPiece | אין | פונקציה המחזירה את הכלי משחק שעל המשבצת. |
| toString | אין | פונקציה המחזירה את הסימון של הכלי שחמט שעל המשבצת אם הוא כלי בצבע שחור יהיה בסימון אותיות קטנות ואם הוא כלי בצבע לבן יהיה בסימון באותיות גדולות ובצורת טקסט. |
| Piece |  | הורשה: איןמחלקה המייצגת כלי משחק כללי. |
| Piece | **int** piecePosition, Alliance pieceAlliance,PieceType pieceType, **boolean** isFirstMove | פונקציית הבנאי המעדכנת את מיקום כלי המשחק, את הצבע שלו ואת הסוג הכליואם הוא ביצע מהלך ראשון ואת ערך הקוד של האובייקט. |
| computeHashCode | אין | פונקציית המחשבת את ערך הקוד של כל אובייקט(כי לכל אובייקט אמור להיות ערך קוד יחודי והוא נועד לצורך בדיקה אם שני אובייקטים שווים או לא). |
| Equals | Object other | פונקציה הבודקת אם שני אובייקטים שווים לפי ערך קוד האובייקט ואם הוא ממחלקה Piece ואם יש להם אותם משתנים(תכונות) |
| hashCode | אין | פונקציית גיבוב(בעיברית) המחשבת את ערך הקוד של האובייקט. |
| getPiecePosition | אין | פונקציה המחזירה את מיקום כלי השחמט. |
| getPieceAlliance | אין | פונקציה המחזירה את צבע כלי השחמט. |
| getPieceType | אין | פונקציה המחזירה את סוג כלי השחמט. |
| isFirstMove | אין | פונקציה המחזירה אם זהו המהלך הראשון של הכלי. |
| getPieceValue | אין | פונקציה המחזירה את ערך הכלי. |
| calculateLegalMoves | Board board | פונקציה המחזירה רשימה של כל המהלכים החוקיים של כלי המשחק. |
| movePiece | Move move | פונקציה המזיזה כלי שחמט עלי ידי יצירת מופע חדש של כלי זה במיקום החדש. |
| PieceType |  | הורשה: איןEnum(קבוצה של משתנים קבועים עם מאפיינים והתנהגות) המייצגת את סוג כלי המשחק. |
| PieceType | String pieceName, **int** pieceValue | פונקציית הבנאי המעדכנת לכל משתנה קבוע את סימון השם שלו כטקסט ואת הערך שלו. |
| getPieceValue | אין | פונקציה המחזירה ערך המשתנה הקבוע. |
| toString | אין | פונקציה המחזירה את סימון שם המשתנה הקבוע כטקסט. |
| isKing | אין | פונקציה המחזירה אם המשתנה הקבוע הוא מלך. |
| isRook | אין | פונקציה המחזירה אם המשתנה הקבוע הוא צריח. |
| Alliance |  | הורשה: איןEnum(קבוצה של משתנים קבועים עם מאפיינים והתנהגות) המייצגת את צבע כלי המשחק. |
| Alliance | String color | פונקציית הבנאי המעדכנת לכל משתנה קבוע את סימון השם שלו כטקסט. |
| toString |  | פונקציה המחזירה את סימון שם המשתנה הקבוע כטקסט. |
| getDirection |  | פונקציה המחזירה את כיוון המשתנה הקבוע(פונקציית עזר לבדיקה אם המהלך של החייל חוקית). |
| isBlack |  | פונקציה המחזירה אם המשתנה הקבוע הוא שחור. |
| isWhite |  | פונקציה המחזירה אם המשתנה הקבוע הוא לבן. |
| isPawnPromotionSquare | **int** position | פונקציה המחזירה אם לחייל בהתאם למיקום שלו ובהתאם למשתנה הקבוע צריך לקבל קידום(להפוך למלכה). |
| choosePlayer | WhitePlayer whitePlayer, BlackPlayer blackPlayer | פונקציה המחזירה בהתאם למשתנה הקבוע את השחקן המסויים(השחקן הלבן או השחור, נועד כדי לבדוק מי השחקן הנוכחי). |
| Bishop |  | הורשה: Pieceמחלקה המייצגת את כלי משחק הרץ. |
| Bishop | **int** piecePosition, Alliance pieceAlliance | פונקציית הבנאי המעדכנת את מיקום כלי המשחק ואת הצבע שלו ומאתחלת אותו כאשר הוא עדיין לא ביצע מהלך ראשון. |
| Bishop | **int** piecePosition, Alliance pieceAlliance, **boolean** isFirstMove | פונקציית הבנאי המעדכנת את מיקום כלי המשחק ואת הצבע שלו ואם הוא ביצע מהלך ראשון. |
| calculateLegalMoves | Board board | פונקציה המחזירה רשימה של כל המהלכים החוקיים של כלי המשחק. לחישוב רשימת המהלכים של הכלי נעבור על כל ערכי האופסטים(מספרים קבועים שבהם אולי הכלי יכול לנוע) ונחשב את מיקום היעד ביחס למיקום הנוכחי ולעומת הפרש נחשב את מיקום היעד שוב ושוב(הוספת האופסט כל פעם) עד שמיקום היעד כבר לא נמצא על לוח המשחק ועבור כל מיקום היעד חדש אשר נמצא בעמודה הראשונה ושמינית(לחוקי ערכי האופסט יש מקרים היוצאים מן הכלל). כלומר, מיקום יעד אשר מפר את החוקים נעבור לאופסט אחר. לאחר מכן, אם במיקום היעד אין כלי משחק אז נוסיף לרשימת מהלכים את המהלך הזזה הדרוש ואם יש כלי משחק במיקום היעד נבודק שהוא לא של אותו צבע כמו של צבע הכלי הנוכחי ואם הוא לא נוסיף לרשימה את מהלך תקיפה הדרוש. |
| movePiece | Move move | פונקציה המזיזה כלי שחמט עלי ידי יצירת מופע חדש של כלי זה במיקום החדש. |
| toString | אין | פונקציה המחזירה את סימון סוג כלי המשחק כטקסט. |
| isFirstColumnExclusion | **int** currentPosition, **int** candidateOffset | פונקציה המחזירה עבור מיקום נוכחי ועבור אופסטים מסויימים אם הכלי נמצא בעמודה ראשונה. |
| isEighthColumnExclusion | **int** currentPosition, **int** candidateOffset | פונקציה המחזירה עבור מיקום נוכחי ועבור אופסטים מסויימים אם הכלי נמצא בעמודה שמינית. |
| Knight |  | הורשה: Pieceמחלקה המייצגת את כלי משחק הפרש. |
| Knight | **int** piecePosition, Alliance pieceAlliance | פונקציית הבנאי המעדכנת את מיקום כלי המשחק ואת הצבע שלו ומאתחלת אותו כאשר הוא עדיין לא ביצע מהלך ראשון. |
| Knight | **int** piecePosition, Alliance pieceAlliance, **boolean** isFirstMove | פונקציית הבנאי המעדכנת את מיקום כלי המשחק ואת הצבע שלו ואם הוא ביצע מהלך ראשון. |
| calculateLegalMoves | Board board | פונקציה המחזירה רשימה של כל המהלכים החוקיים של כלי המשחק. לחישוב רשימת המהלכים של הכלי נעבור על כל ערכי האופסטים(מספרים קבועים שבהם אולי הכלי יכול לנוע) ונחשב את מיקום היעד ביחס למיקום הנוכחי ונבדוק אם מיקום היעד נמצא בתוך לוח המשחק ואם עבור אופסט מסויים מיקום היעד נמצא בעמודה הראשונה השנייה השביעית ושמינית(לחוקי ערכי האופסט יש מקרים היוצאים מן הכלל). כלומר, מיקום יעד אשר מפר את החוקים נעבור לאופסט אחר. לאחר מכן, אם במיקום היעד אין כלי משחק אז נוסיף לרשימת מהלכים את המהלך הזזה הדרוש ואם יש כלי משחק במיקום היעד נבודק שהוא לא של אותו צבע כמו של צבע הכלי הנוכחי ואם הוא לא נוסיף לרשימה את מהלך תקיפה הדרוש. |
| movePiece | Move move | פונקציה המזיזה כלי שחמט עלי ידי יצירת מופע חדש של כלי זה במיקום החדש. |
| toString | אין | פונקציה המחזירה את סימון סוג כלי המשחק כטקסט. |
| isFirstColumnExclusion | **int** currentPosition, **int** candidateOffset | פונקציה המחזירה עבור מיקום נוכחי ועבור אופסטים מסויימים אם הכלי נמצא בעמודה ראשונה. |
| isSecondColumnExclusion | **int** currentPosition, **int** candidateOffset | פונקציה המחזירה עבור מיקום נוכחי ועבור אופסטים מסויימים אם הכלי נמצא בעמודה שנייה. |
| isSeventhColumnExclusion | **int** currentPosition, **int** candidateOffset | פונקציה המחזירה עבור מיקום נוכחי ועבור אופסטים מסויימים אם הכלי נמצא בעמודה שביעית. |
| isEighthColumnExclusion | **int** currentPosition, **int** candidateOffset | פונקציה המחזירה עבור מיקום נוכחי ועבור אופסטים מסויימים אם הכלי נמצא בעמודה שמינית. |
| Rook |  | הורשה: Pieceמחלקה המייצגת את כלי משחק צריח. |
| Rook | **int** piecePosition, Alliance pieceAlliance | פונקציית הבנאי המעדכנת את מיקום כלי המשחק ואת הצבע שלו ומאתחלת אותו כאשר הוא עדיין לא ביצע מהלך ראשון. |
| Rook | **int** piecePosition, Alliance pieceAlliance, **boolean** isFirstMove | פונקציית הבנאי המעדכנת את מיקום כלי המשחק ואת הצבע שלו ואם הוא ביצע מהלך ראשון. |
| calculateLegalMoves | Board board | פונקציה המחזירה רשימה של כל המהלכים החוקיים של כלי המשחק. לחישוב רשימת המהלכים של הכלי נעבור על כל ערכי האופסטים(מספרים קבועים שבהם אולי הכלי יכול לנוע) ונחשב את מיקום היעד ביחס למיקום הנוכחי ולעומת הפרש נחשב את מיקום היעד שוב ושוב(הוספת האופסט כל פעם) עד שמיקום היעד כבר לא נמצא על לוח המשחק ועבור כל מיקום היעד חדש אשר נמצא בעמודה הראשונה ושמינית(לחוקי ערכי האופסט יש מקרים היוצאים מן הכלל). כלומר, מיקום יעד אשר מפר את החוקים נעבור לאופסט אחר. לאחר מכן, אם במיקום היעד אין כלי משחק אז נוסיף לרשימת מהלכים את המהלך הזזה הדרוש ואם יש כלי משחק במיקום היעד נבודק שהוא לא של אותו צבע כמו של צבע הכלי הנוכחי ואם הוא לא נוסיף לרשימה את מהלך תקיפה הדרוש. |
| movePiece | Move move | פונקציה המזיזה כלי שחמט עלי ידי יצירת מופע חדש של כלי זה במיקום החדש. |
| toString | אין | פונקציה המחזירה את סימון סוג כלי המשחק כטקסט. |
| isFirstColumnExclusion | **int** currentPosition, **int** candidateOffset | פונקציה המחזירה עבור מיקום נוכחי ועבור אופסטים מסויימים אם הכלי נמצא בעמודה ראשונה. |
| isEighthColumnExclusion | **int** currentPosition, **int** candidateOffset | פונקציה המחזירה עבור מיקום נוכחי ועבור אופסטים מסויימים אם הכלי נמצא בעמודה שמינית. |
| Queen |  | הורשה: Pieceמחלקה המייצגת את כלי משחק מלכה. |
| Queen | **int** piecePosition, Alliance pieceAlliance | פונקציית הבנאי המעדכנת את מיקום כלי המשחק ואת הצבע שלו ומאתחלת אותו כאשר הוא עדיין לא ביצע מהלך ראשון. |
| Queen | **int** piecePosition, Alliance pieceAlliance, **boolean** isFirstMove | פונקציית הבנאי המעדכנת את מיקום כלי המשחק ואת הצבע שלו ואם הוא ביצע מהלך ראשון. |
| calculateLegalMoves | Board board | פונקציה המחזירה רשימה של כל המהלכים החוקיים של כלי המשחק. לחישוב רשימת המהלכים של הכלי נעבור על כל ערכי האופסטים(מספרים קבועים שבהם אולי הכלי יכול לנוע) ונחשב את מיקום היעד ביחס למיקום הנוכחי ולעומת הפרש נחשב את מיקום היעד שוב ושוב(הוספת האופסט כל פעם) עד שמיקום היעד כבר לא נמצא על לוח המשחק ועבור כל מיקום היעד חדש אשר נמצא בעמודה הראשונה ושמינית(לחוקי ערכי האופסט יש מקרים היוצאים מן הכלל). כלומר, מיקום יעד אשר מפר את החוקים נעבור לאופסט אחר. לאחר מכן, אם במיקום היעד אין כלי משחק אז נוסיף לרשימת מהלכים את המהלך הזזה הדרוש ואם יש כלי משחק במיקום היעד נבודק שהוא לא של אותו צבע כמו של צבע הכלי הנוכחי ואם הוא לא נוסיף לרשימה את מהלך תקיפה הדרוש. |
| movePiece | Move move | פונקציה המזיזה כלי שחמט עלי ידי יצירת מופע חדש של כלי זה במיקום החדש. |
| toString | אין | פונקציה המחזירה את סימון סוג כלי המשחק כטקסט. |
| isFirstColumnExclusion | **int** currentPosition, **int** candidateOffset | פונקציה המחזירה עבור מיקום נוכחי ועבור אופסטים מסויימים אם הכלי נמצא בעמודה ראשונה. |
| isEighthColumnExclusion | **int** currentPosition, **int** candidateOffset | פונקציה המחזירה עבור מיקום נוכחי ועבור אופסטים מסויימים אם הכלי נמצא בעמודה שמינית. |
| King |  | הורשה: Pieceמחלקה המייצגת את כלי משחק מלך. |
| King | **int** piecePosition, Alliance pieceAlliance | פונקציית הבנאי המעדכנת את מיקום כלי המשחק ואת הצבע שלו ומאתחלת אותו כאשר הוא עדיין לא ביצע מהלך ראשון. |
| King | **int** piecePosition, Alliance pieceAlliance, **boolean** isFirstMove | פונקציית הבנאי המעדכנת את מיקום כלי המשחק ואת הצבע שלו ואם הוא ביצע מהלך ראשון. |
| calculateLegalMoves | Board board | פונקציה המחזירה רשימה של כל המהלכים החוקיים של כלי המשחק. לחישוב רשימת המהלכים של הכלי נעבור על כל ערכי האופסטים(מספרים קבועים שבהם אולי הכלי יכול לנוע) ונחשב את מיקום היעד ביחס למיקום הנוכחי ונבדוק אם מיקום היעד נמצא בתוך לוח המשחק ואם עבור אופסט מסויים מיקום היעד נמצא בעמודה הראשונה ושמינית(לחוקי ערכי האופסט יש מקרים היוצאים מן הכלל). כלומר, מיקום יעד אשר מפר את החוקים נעבור לאופסט אחר. לאחר מכן, אם במיקום היעד אין כלי משחק אז נוסיף לרשימת מהלכים את המהלך הזזה הדרוש ואם יש כלי משחק במיקום היעד נבודק שהוא לא של אותו צבע כמו של צבע הכלי הנוכחי ואם הוא לא נוסיף לרשימה את מהלך תקיפה הדרוש. |
| movePiece | Move move | פונקציה המזיזה כלי שחמט עלי ידי יצירת מופע חדש של כלי זה במיקום החדש. |
| toString | אין | פונקציה המחזירה את סימון סוג כלי המשחק כטקסט. |
| isFirstColumnExclusion | **int** currentPosition, **int** candidateOffset | פונקציה המחזירה עבור מיקום נוכחי ועבור אופסטים מסויימים אם הכלי נמצא בעמודה ראשונה. |
| isEighthColumnExclusion | **int** currentPosition, **int** candidateOffset | פונקציה המחזירה עבור מיקום נוכחי ועבור אופסטים מסויימים אם הכלי נמצא בעמודה שמינית. |
| Pawn |  | הורשה: Pieceמחלקה המייצגת את כלי משחק חייל. |
| Pawn | **int** piecePosition, Alliance pieceAlliance | פונקציית הבנאי המעדכנת את מיקום כלי המשחק ואת הצבע שלו ומאתחלת אותו כאשר הוא עדיין לא ביצע מהלך ראשון. |
| Pawn | **int** piecePosition, Alliance pieceAlliance, **boolean** isFirstMove | פונקציית הבנאי המעדכנת את מיקום כלי המשחק ואת הצבע שלו ואם הוא ביצע מהלך ראשון. |
| calculateLegalMoves | Board board | פונקציה המחזירה רשימה של כל המהלכים החוקיים של כלי המשחק. לחישוב רשימת המהלכים של הכלי נעבור על כל ערכי האופסט של הכלי ונחשב בהתאם לכיוון הכלי(שחור=1-, לבן=1) האופסט והמיקום הנוכחי את מיקום היעד ונבדוק אם הוא נמצא בלוח המשחק. לאחר מכן, נבדוק אם המהלך הוא מהלך חייל רגיל(חייל שזז קדימה משבצת אחת) ואם אין במיקום היעד כלי אחר נבדוק אם החייל הוא חייל שצריך לקבל קידום, אם כן נוסיף לרשימת המהלכים מהלך קידום חייל ואם לא נוסיף לרשימה מהלך חייל רגיל.אחרת אם המהלך הוא מהלך קפיצת חייל(החייל הקופץ שתי משבצות בהתחלה) ואם זהו המהלך הראשון של הכלי והוא נמצא בשורה שביעית או שנייה(שורה שביעית עבור חייל בצבע שחור ושנייה עבור חייל בצבע לבן) אז נבדוק שאין במשבצת שבדרכו ובמיקום היעד כלי. לכן, נוסיף לרשימת המהלכים מהלך קפיצת חייל.אחרת אם המהלך הוא מהלך תקיפה של חייל עבור אופסט מסויים(תקיפה של צד מסויים) נבדוק שעבור תקיפה זו מיקום היעד לא נמצא בעמודה הראשונה והשמינית(לחוקי ערכי האופסט יש מקרים היוצאים מן הכלל). לפיכך, אם במיקום היעד יש כלי בצבע השונה מהכלי אז נבדוק אם החייל הוא חייל שצריך לקבל קידום, אם כן נוסיף לרשימת המהלכים מהלך קידום חייל ואם לא נוסיף לרשימה מהלך חייל רגיל. אחרת אם עדיין עבור אותו מהלך תקיפה עם האופסט המסויים(תקיפה של צד מסויים) במיקום היעד אין כלי נבודק אם בלוח יש חייל בצבע שונה לכלי הנוכחי שהוא ביצע קפיצת חייל במהלך האחרון של המשחק. אם כן נבדוק אם עבור האופסט המסויים החייל היריב נמצא בהתאם לצבע שלו ליד הכיל הנוכחי בצד שמאל ביחס לפרספרקיבה של השחקן שרוצה לבצע את המהלך. ואם זה כלי של השחקן היריב נוסיף לרשימת המהלכים את מהלך תקיפה זה(EnPassantAttack).אחרת אם המהלך הוא מהלך תקיפה של חייל עבור אופסט מסויים(תקיפה של צד אחר) נבדוק בדיוק כמו שבדקנו תקיפה עבור אופסטט מסויים של צד מסויים) ונוסיף לרשימת המהלכים את המהלך שבחרנו לפי האופסט. |
| getPromotionPiece | אין | פונקציה המקדמת את החייל למלכה על ידי יצירת מופע של כלי של מלכה במיקום של החייל. |
| movePiece | Move move | פונקציה המזיזה כלי שחמט עלי ידי יצירת מופע חדש של כלי זה במיקום החדש. |
| toString | אין | פונקציה המחזירה את סימון סוג כלי המשחק כטקסט. |
| BoardUtils |  | הורשה: איןמחלקת עזר השומרת נתונים שעוזרים ללוח המשחק. |
| BoardUtils | אין | פונקציית הבנאי שמכיוון שאנחנו משתמשים רק בפונקציות הסטטיות שלה אין צורך ליצור מופע לכן אנחנו בודקים שאף אחד לא יקרא לפונקציה הזאת. |
| initColumn | **int** columnNumber | פונקציה המחזירה מערך של מספרי העמודה בלוח עבור מספר עמודה. |
| initRow | **int** rowNumber | פונקציה המחזירה מערך של מספרי השורה בלוח עבור מספר שורה. |
| initializePositionToCoordinateMap | אין | פונקציה המחזירה מפה של כל סימוני המשבצות בלוח עבור כל האינדקסים של המשבצות בלוח. |
| initializeAlgebraicNotation | אין | פונקציה המחזירה מערך של כל סימוני המשבצות בלוח כטקסט. |
| isValidTileCoordinate | **int** coordinate | פונקציה המחזירה אם אינדקס המשבצת נמצא בלוח המשחק. |
| getCoordinateAtPosition | String position | פונקציה המחזירה את אינדקס המשבצת עבור סימון המשבצת כטקסט. |
| getPositionAtCoordinate | **int** coordinate | פונקציה המחזירה סימון המשבצת כטקסט עבור אינדקס המשבצת. |
| isEndGame | Board board | פונקציה הבודקת אם הסתיים המשחק(אם השחקן הנוכחי בשחמט או בתיקו). |
| Builder |  | הורשה: איןמחלקת עזר העוזרת ביצירה ושימוש במופע לוח השחמט. |
| Builder | אין | פונקציית הבנאי המתחלת את מפת לוח המשחק(מפה שעבור אינדקס המשבצת נותנת כלי מסויים או כלי ריק). |
| setPiece | Piece piece | פונקציה המכניסה למפה בהתאם למיקום הכלי את הכלי. |
| setMoveMaker | Alliance nextMoveMaker | פונקציה הקובעת בהתאם לצבע שמקבלת תור מי לשחק. |
| build | אין | פונקציה היוצרת מופע של לוח המשחק(Board). |
| setEnPassantPawn | Pawn EnPassantPawn | פונקציה הקובעת בהתאם לכלי שמקבלת את כלי שאולי אפשר לבצע עליו מהלך(EnPassent). |
| Board |  | הורשה: איןמחלקה המייצגת את לוח המשחק. |
| Board | Builder builder | פונקציית הבנאי המאתחלת את רשימת המשבצות, את כלי המשחק של שחקן הלבן והשחור, את כלי הEnPassant, את רשימת המהלכים של השחקן הלבן והשחור ואת השחקן הלבן והשחור(לפי המחלקה שלו) ואת השחקן נוכחי. |
| getTile | **int** tileCoordinate | פונקציה המחזירה משבצת בהתאם לאינדקס המשבצת. |
| getWhitePlayer | אין | פונקציה המחזירה את השחקן הלבן. |
| getBlackPlayer | אין | פונקציה המחזירה את השחקן השחור. |
| getCurrentPlayer | אין | פונקציה המחזירה את השחקן הנוכחי. |
| getEnPassantPawn | אין | פונקציה המחזירה את כלי EnPassant של הלוח הנוכחי. |
| getBlackPieces | אין | פונקציה המחזירה רשימה של כלי משחק של השחקן השחור ושל הלוח הנוכחי. |
| getWhitePieces | אין | פונקציה המחזירה רשימה של כלי משחק של השחקן הלבן ושל הלוח הנוכחי. |
| calculateLegalMoves | Collection<Piece> pieces | פונקציה המחזירה רשימת מהלכים של שחקן מסויים לפי הוספת רשימת המהלכים של כל כלי משחק שנמצא ברשימת כלי המשחק. |
| getAllLegalMoves | אין | פונקציה המחזירה רשימת מהלכים של שני השחקנים ביחד. |
| toString | אין | פונקצייה המחזירה סימון של לוח שחמט וכל כלי המשחק שעליו כטקסט. |
| calculateActivePieces | List<Tile> gameBoard,Alliance alliance | פונקציה המחזירה רשימה של כל כלי המשחק נמצאים על לוח המשחק בהתאם ללוח המשחק הנוכחי וצבע הכלים. |
| createGameBoard | Builder builder | פונקציה היוצרת לוח שחמט המיוצג על ידי רשימת משבצות שעליהם יש כלי שחמט מסויים או ריק. |
| createStandardBoard | אין | פונקציה היוצרת לוח שחמט סטדנדרטי כאשר כלי המשחק המסויימים נמצאים במשבצות שלהם ותורו של השחקן הלבן להתחיל משחק וקוראת לפונקציית הבנאי(על ידי יצירת מופע חדש). |
| Move |  | הורשה: איןמחלקה המייצגת מהלך כללי של כלי. |
| Move | Board board, Piece movedPiece, **int** destinationCoordinate | פונקציית הבנאי המאתחלת את הלוח המסויים, את הכלי המוזז, את מיקום היעד ואם זהו המהלך הראשון של הכלי. |
| Move | Board board, **Int** destinationCoordinate | פונקציית הבנאי המאתחלת את הלוח המסויים ואת מיקום היעד(מיועד להזזה לא חוקית). |
| getCurrentCoordinate | אין | פונקציה המחזירה את המיקום הנוכחי של כלי המשחק. |
| getBoard | אין | פונקציה המחזירה את לוח המשחק הנוכחי. |
| getMovedPiece | אין | פונקציה המחזירה את כלי המשחק המוזז. |
| getDestinationCoordinate | אין | פונקציה המחזירה את מיקום היעד. |
| isAttack | אין | פונקציה המחזירה אם זהו מהלך תקיפה. |
| isCastlingMove | אין | פונקציה המחזירה אם זהו מהלך הצרחה. |
| getAttackedPiece | אין | פונקציה המחזירה את הכלי הנתקף. |
| equals | Object other | פונקציה הבודקת אם שני אובייקטים שווים לפי ערך קוד האובייקט ואם הוא ממחלקה Piece ואם יש להם אותם משתנים(תכונות). |
| hashCode | אין | פונקציית גיבוב(בעיברית) המחשבת את ערך הקוד של האובייקט. |
| execute | אין | פונקציה אשר מבצעת את המהלך על ידי יצירת מופע של לוח חדש(מדומה) עם כל כלי המשחק הנוכחים(בלי הכלי המוזז) ואז מוסיפים את הכלי המוזז במיקום היעד. בנוסף, משנה בלוח את תורו של היריב. אנחנו יוצרים מופע חדש של לוח עם התנאים האלה כדי לבדוק את חוקי הכלים בלוח לפני השוואותו ללוח הראשי. |
| AttackMove |  | הורשה: Moveמחלקה המייצגת מהלך תקיפה כללי של כלי. |
| AttackMove | Board board, Piece movedPiece, **int** destinationCoordinate, Piece attackedPiece | פונקציית הבנאי המאתחלת את הלוח המסויים, את הכלי המוזז, את מיקום היעד ואת הכלי המותקף. |
| equals | Object other | פונקציה הבודקת אם שני אובייקטים שווים לפי ערך קוד האובייקט ואם הוא ממחלקה Piece ואם יש להם אותם משתנים(תכונות). |
| hashCode | אין | פונקציית גיבוב(בעיברית) המחשבת את ערך הקוד של האובייקט. |
| isAttack | אין | פונקציה המחזירה אם זהו מהלך תקיפה. |
| getAttackedPiece | אין | פונקציה המחזירה את הכלי הנתקף. |
| execute | אין | פונקציה אשר מבצעת את המהלך על ידי יצירת מופע של לוח חדש(מדומה) עם כל כלי המשחק הנוכחים(בלי הכלי המוזז ובלי הכלי המותקף) ואז מוסיפים את הכלי המוזז במיקום היעד(בלי הכלי המותקף). בנוסף, משנה בלוח את תורו של היריב. אנחנו יוצרים מופע חדש של לוח עם התנאים האלה כדי לבדוק את חוקי הכלים בלוח לפני השוואותו ללוח הראשי. |
| MajorMove |  | הורשה: Moveמחלקה המייצגת מהלך כללי של כלי ראשי. |
| MajorMove | Board board, Piece movedPiece, **int** destinationCoordinate | פונקציית הבנאי המאתחלת את הלוח המסויים, את הכלי המוזז, את מיקום היעד. |
| equals | Object other | פונקציה הבודקת אם שני אובייקטים שווים לפי ערך קוד האובייקט ואם הוא ממחלקה Piece ואם יש להם אותם משתנים(תכונות). |
| toString | אין | פונקציה המחזירה סימון של מהלך הזזה של כלי ראשי(כל כלי המשחק חוץ מחייל). |
| MajorAttackMove |  | הורשה: AttackMoveמחלקה המייצגת מהלך תקיפה כללי של כלי ראשי. |
| MajorAttackMove | Board board, Piece movedPiece, **int** destinationCoordinate, Piece attackedPiece | פונקציית הבנאי המאתחלת את הלוח המסויים, את הכלי המוזז, את מיקום היעד ואת הכלי הנתקף. |
| equals | Object other | פונקציה הבודקת אם שני אובייקטים שווים לפי ערך קוד האובייקט ואם הוא ממחלקה Piece ואם יש להם אותם משתנים(תכונות). |
| toString | אין | פונקציה המחזירה סימון של מהלך תקיפה של כלי ראשי(כל כלי המשחק חוץ מחייל). |
| PawnMove |  | הורשה: Moveמחלקה המייצגת מהלך של כלי החייל. |
| PawnMove | Board board, Piece movedPiece, **int** destinationCoordinate | פונקציית הבנאי המאתחלת את הלוח המסויים, את הכלי המוזז, את מיקום היעד. |
| equals | Object other | פונקציה הבודקת אם שני אובייקטים שווים לפי ערך קוד האובייקט ואם הוא ממחלקה Piece ואם יש להם אותם משתנים(תכונות). |
| toString | אין | פונקציה המחזירה סימון של מהלך הזזה של משבצת אחת של כלי משחק החייל(רגלי). |
| PawnAttackMove |  | הורשה: AttackMoveמחלקה המייצגת מהלך תקיפה כללי של כלי החייל. |
| PawnAttackMove | Board board, Piece movedPiece, **int** destinationCoordinate, Piece attackedPiece | פונקציית הבנאי המאתחלת את הלוח המסויים, את הכלי המוזז, את מיקום היעד ואת הכלי הנתקף. |
| equals | Object other | פונקציה הבודקת אם שני אובייקטים שווים לפי ערך קוד האובייקט ואם הוא ממחלקה Piece ואם יש להם אותם משתנים(תכונות). |
| toString | אין | פונקציה המחזירה סימון של מהלך תקיפה של משבצת אחת(באלכסון) של כלי משחק החייל(רגלי). |
| PawnEnPassantAttack |  | הורשה: PawnAttackMoveמחלקה המייצגת מהלך תקיפה EnPassant של כלי החייל. |
| PawnEnPassantAttack | Board board, Piece movedPiece, **int** destinationCoordinate, Piece attackedPiece | פונקציית הבנאי המאתחלת את הלוח המסויים, את הכלי המוזז, את מיקום היעד ואת הכלי הנתקף. |
| equals | Object other | פונקציה הבודקת אם שני אובייקטים שווים לפי ערך קוד האובייקט ואם הוא ממחלקה Piece ואם יש להם אותם משתנים(תכונות). |
| toString | אין | פונקציה המחזירה סימון של מהלך תקיפה(EnPassant) של משבצת אחת(באלכסון) של כלי משחק החייל(רגלי). |
| PawnJump |  | הורשה: PawnMoveמחלקה המייצגת מהלך קפיצה של כלי החייל. |
| PawnJump | Board board, Piece movedPiece, **int** destinationCoordinate | פונקציית הבנאי המאתחלת את הלוח המסויים, את הכלי המוזז, את מיקום היעד. |
| equals | Object other | פונקציה הבודקת אם שני אובייקטים שווים לפי ערך קוד האובייקט ואם הוא ממחלקה Piece ואם יש להם אותם משתנים(תכונות). |
| execute | אין | פונקציה אשר מבצעת את המהלך על ידי יצירת מופע של לוח חדש(מדומה) עם כל כלי המשחק הנוכחים(בלי הכלי המוזז) ואז מוסיפים את הכלי המוזז במיקום היעד. בנוסף, הכלי המוזז הוא כלי שאולי יכולים לעשות עליו מהלך תקיפה EnPassant ומשנה בלוח את תורו של היריב. אנחנו יוצרים מופע חדש של לוח עם התנאים האלה כדי לבדוק את חוקי הכלים בלוח לפני השוואותו ללוח הראשי. |
| PawnPromotion |  | הורשה: PawnMoveמחלקה המייצגת מהלך קפיצה של כלי החייל. |
| PawnPromotion | Move decoratedMove | פונקציית הבנאי המאתחלת את הלוח המסויים, את הכלי המוזז, את מיקום היעד. |
| isAttack | אין | פונקציה המחזירה אם זהו מהלך תקיפה. |
| getAttackedPiece | אין | פונקציה המחזירה את הכלי הנתקף. |
| equals | Object other | פונקציה הבודקת אם שני אובייקטים שווים לפי ערך קוד האובייקט ואם הוא ממחלקה Piece ואם יש להם אותם משתנים(תכונות). |
| hashCode | אין | פונקציית גיבוב(בעיברית) המחשבת את ערך הקוד של האובייקט. |
| execute | אין | פונקציה אשר מבצעת את המהלך על ידי יצירת מופע של לוח חדש(מדומה) עם כל כלי המשחק הנוכחים(בלי הכלי המוזז) ואז מוסיפים את הכלי המוזז עם קידום(מלכה) במיקום היעד. בנוסף, משנה בלוח את תורו של היריב. אנחנו יוצרים מופע חדש של לוח עם התנאים האלה כדי לבדוק את חוקי הכלים בלוח לפני השוואותו ללוח הראשי. |
| toString | אין | פונקציה המחזירה סימון של מהלך קידום חייל(רגלי). |
| CastleMove |  | הורשה: Moveמחלקה המייצגת מהלך הצרחה בין המלך והצריח. |
| CastleMove | Board board, Piece movedPiece, **int** destinationCoordinate, Rook castleRook, **int** castleRookDestination | פונקציית הבנאי המאתחלת את הלוח המסויים, את הכלי המוזז, את מיקום היעד של המלך, את הצריח המוזז ואת מיקום היעד של הצריח. |
| getCastleRook | אין | פונקציה המחזירה את הצריח המוזז. |
| isCastlingMove | אין | פונקציה המחזירה אם זהו מהלך הצרחה. |
| equals | Object other | פונקציה הבודקת אם שני אובייקטים שווים לפי ערך קוד האובייקט ואם הוא ממחלקה Piece ואם יש להם אותם משתנים(תכונות). |
| hashCode | אין | פונקציית גיבוב(בעיברית) המחשבת את ערך הקוד של האובייקט. |
| Execute | אין | פונקציה אשר מבצעת את המהלך על ידי יצירת מופע של לוח חדש(מדומה) עם כל כלי המשחק הנוכחים(בלי הכלי המוזז והצריח המוזז) ואז מוסיפים את הכלי המוזז במיקום היעד ואת הצריח המוזז במיקום היעד שלו. בנוסף, משנה בלוח את תורו של היריב. אנחנו יוצרים מופע חדש של לוח עם התנאים האלה כדי לבדוק את חוקי הכלים בלוח לפני השוואותו ללוח הראשי. |
| KingSideCastleMove |  | הורשה: CastleMoveמחלקה המייצגת מהלך הצרחה בין המלך והצריח בצד המלך. |
| KingSideCastleMove | Board board, Piece movedPiece, **int** destinationCoordinate, Rook castleRook, **int** castleRookDestination | פונקציית הבנאי המאתחלת את הלוח המסויים, את הכלי המוזז, את מיקום היעד של המלך, את הצריח המוזז ואת מיקום היעד של הצריח. |
| equals | Object other | פונקציה הבודקת אם שני אובייקטים שווים לפי ערך קוד האובייקט ואם הוא ממחלקה Piece ואם יש להם אותם משתנים(תכונות). |
| toString | אין | פונקציה המחזירה סימון של מהלך הצרחה בצד המלך. |
| QueenSideCastleMove |  | הורשה: CastleMoveמחלקה המייצגת מהלך הצרחה בין המלך והצריח בצד המלכה. |
| QueenSideCastleMove | Board board, Piece movedPiece, **int** destinationCoordinate, Rook castleRook, **int** castleRookDestination | פונקציית הבנאי המאתחלת את הלוח המסויים, את הכלי המוזז, את מיקום היעד של המלך, את הצריח המוזז ואת מיקום היעד של הצריח. |
| equals | Object other | פונקציה הבודקת אם שני אובייקטים שווים לפי ערך קוד האובייקט ואם הוא ממחלקה Piece ואם יש להם אותם משתנים(תכונות). |
| toString | אין | פונקציה המחזירה סימון של מהלך הצרחה בצד המלכה. |
| NullMove |  | הורשה: Moveמחלקה המייצגת מהלך לא חוקי של כלי. |
| NullMove | אין | פונקציית הבנאי המייצגת מהלך לא חוקי של כלי. מאתחלת את הלוח בלוח ריק ובמיקום יעד 1-. |
| getCurrentCoordinate | אין | פונקציה המחזירה את מיקום הנוכחי של הכלי(בגלל שאין כלי הוא 1-). |
| Execute | אין | פונקציה אשר מבצעת מהלך(מכיוון שזהו מהלך לא חוקי נוודא שפונקציה הזאת לא תקרא). |
| MoveFactory |  | הורשה: איןמחלקה סטטית הנועדה ליצירת מהלכים שהם חוקיים. |
| MoveFactory | אין | פונקציית הבנאי(המחלקה היא לא מהלך מסויים לכן לא נרצה ליצור מופע חדש(לקרוא לפונקציית הבנאי) לכן נוודא שלא נקרא לפונקצייה זו. |
| createMove | Board board, **int** currentCoordinate, **int** destinationCoordinate | פונקציה המחזירה מהלך אשר קיים ברשימת המהלכים הקיימים של הלוח. אם לא מחזירה מהלך ריק(Null Move). |
| MoveTransition |  | הורשה: איןמחלקה המייצגת לוח(מדומה) שבו מהלכי המשחק מתרחשים ונבדקים. |
| MoveTransition | Board transitionBoard, Move move, MoveStatus moveStatus | פונקציית הבנאי המאתחלת את הלוח המדומה המסויים, את מהלך הנוכחי ומשתנה אם המהלך חוקי בלוח המדומה. |
| getTransitionBoard | אין | פונקציה המחזירה את הלוח המדומה. |
| getMoveStatus | אין | פונקציה המחזירה אם המהלך חוקי בלוח המדומה. |
| MoveStatus |  | הורשה: איןEnum(קבוצה של משתנים קבועים עם מאפיינים והתנהגות) המייצגת את מצב הלוח לאחר ביצוע המהלך. |
| isDone | אין | פונקציה הבודקת לפי שם המשתנה אם המהלך לפי מצב הלוח חוקי. |
| Player |  | הורשה: איןמחלקה המייצגת שחקן כללי. |
| Player | Board board, Collection<Move> legalMoves, Collection<Move> opponentMoves | פונקציית הבנאי המאתחלת את לוח המשחק, את רשימת המהלכים החוקיים של השחקן הנוכחי ושל יריבו, ובודקת אם הוא בשח. |
| calculateAttackOnTile | int piecePosition, Collection<Move> moves | פונקציה המחזירה רשימה של מהלכי תקיפה על מיקום יעד מסויים. |
| getPlayerKing | אין | פונקציה המחזירה את המלך של השחקן. |
| getLegalMoves | אין | פונקציה המחזירה את המהלכים החוקיים של השחקן. |
| establishKing | אין | פונקציה המחזירה מלך בלוח ובודקת שהוא אכן קיים. |
| isMoveLegal | Move candidateMove | פונקציה המחזירה אם מהלך חוקי. |
| isInCheck | אין | פונקציה המחזירה אם יש שח. |
| isInCheckMate | אין | פונקציה המחזירה אם יש שחמט. |
| isInStaleMate | אין | פונקציה המחזירה אם יש תיקו. |
| hasNoEscapeMoves | אין | פונקציה הבודקת אם יש למלך יש מהלכים חוקיים בלוח. |
| makeMove | Move candidateMove | פונקציה המחזירה לוח(מדומה) בהתאם למהלך ומבצעת את המהלך בלוח המדומה ובודקת שהוא אכן חוקי.כלומר, מהלך שלא יגרום לשח לאותו שחקן. אם התנאים מתקיימים מחזירה את הלוח המדומה עם המהלך ואם לא מחזירה את הלוח ללא ביצוע המהלך. |
| getActivePieces | אין | פונקציה המחזירה רשימה של כלי משחק הקיימים בלוח עבור שחקן מסויים. |
| getAlliance | אין | פונקציה המחזירה את צבע השחקן. |
| getOpponent | אין | פונקציה המחזירה את השחקן היריב. |
| calculateKingCastles | Collection<Move> playerLegals, Collection<Move> opponentLegals | פונקציה הבודקת אם אפשר לבצע מהלך הצרחה שהוא חוקי. |
| WhitePlayer |  | הורשה: Playerמחלקה המייצגת שחקן הלבן. |
| WhitePlayer | Board board, Collection<Move> whiteStandardLegalMoves, Collection<Move> blackStandardLegalMoves | פונקציית הבנאי המאתחלת את לוח המשחק, את רשימת המהלכים החוקיים של השחקן הנוכחי ושל יריבו, ובודקת אם הוא בשח. |
| getActivePieces | אין | פונקציה המחזירה רשימה של כלי משחק הקיימים בלוח עבור שחקן מסויים. |
| getAlliance | אין | פונקציה המחזירה את צבע השחקן. |
| getOpponent | אין | פונקציה המחזירה את השחקן היריב. |
| calculateKingCastles | Collection<Move> playerLegals, Collection<Move> opponentLegals | פונקציה הבודקת אם אפשר לבצע מהלך הצרחה שהוא חוקי. |
| BlackPlayer |  | הורשה: Playerמחלקה המייצגת שחקן השחור. |
| BlackPlayer | Board board, Collection<Move> whiteStandardLegalMoves, Collection<Move> blackStandardLegalMoves | פונקציית הבנאי המאתחלת את לוח המשחק, את רשימת המהלכים החוקיים של השחקן הנוכחי ושל יריבו, ובודקת אם הוא בשח. |
| getActivePieces | אין | פונקציה המחזירה רשימה של כלי משחק הקיימים בלוח עבור שחקן מסויים. |
| getAlliance | אין | פונקציה המחזירה את צבע השחקן. |
| getOpponent | אין | פונקציה המחזירה את השחקן היריב. |
| calculateKingCastles | Collection<Move> playerLegals, Collection<Move> opponentLegals | פונקציה הבודקת אם אפשר לבצע מהלך הצרחה שהוא חוקי. |
| Controller |  | הורשה: איןמחלקה המייצגת |
| initialize | URL url, ResourceBundle resourceBundle | פונקציה המאתחלת את הלוחות בממשק הגרפי ואת לוח המשחק. |
| createTableMenuBar | אין | פונקציה היוצרת לוח כלים בממשק הגרפי. |
| createFileMenu | אין | פונקציה היוצרת תפריט בשם "File" בלוח כלים ואת כל הפריטים שלו. |
| createPreferencesMenu | אין | פונקציה היוצרת תפריט בשם "Preferences" בלוח כלים ואת כל הפריטים שלו. |
| createMenuItem | אין | פונקציה היוצרת פריט לתפריט. |
| BoardDirection |  | הורשה: איןEnum(קבוצה של משתנים קבועים עם מאפיינים והתנהגות) המייצגת את כיוון הלוח(כדי שיהיה אפשר להפוך את הלוח). |
| traverse | List<TilePanel> boardTiles | פונקציה המהפכת את סדר הרשימה של המשבצות בלוח לוח המשחק(לוח שמיוצג בממשק הגרפי). |
| opposite | אין | פונקציה המהפכת את כיוון שם המשתנה. |
| MoveLog |  | הורשה: איןמחלקה המייצגת תיעוד בקולקציה את המהלכים שבוצעו במשחק. |
| MoveLog | אין | פונקציית הבנאי המאתחלת את הקולקציה. |
| getMoves | אין | פונקציה המחזירה את תיעוד המהלכים. |
| addMove | Move move | פונקציה המוסיפה מהלך הקולקציה. |
| size | אין | פונקציה המחזירה את גודל הקולקציה. |
| clear | אין | פונקציה המוחקת את כל המהלכים בקולקציה. |
| removeMove | **int** index | פונקציה המוחקת מהלך לפי אינדקס. |
| removeMove | Move move | פונקציה המוחקת מהלך לפי מהלך מסויים. |
| BoardPanel |  | הורשה: GridPaneמחלקה המייצגת צורת לוח המשחק בממשק הגרפי ויורשת מלוח גרפי שמחולק ל64 תאים. |
| BoardPanel | אין | פונקציית הבנאי המאתחלת את גודל הלוח ובכל תא מתוך 64 התאים יוצרת משבצת המייצגת לוח גרפי(TilePanel). |
| drawBoard | Board board | פונקצייה המציירת שוב את הלוח כאשר מבוצע מהלך. |
| TilePanel |  | הורשה: Paneמחלקה המייצגת צורת תא בלוח המשחק בממשק הגרפי ויורשת מלוח גרפי. |
| TilePanel | BoardPanel boardPanel, **int** tileId | פונקציית הבנאי המאתחלת את גודל התא וקוראת לפונקציות פנימיות. בנוסף, היא מנהלת את אופן לחיצת העכבר עם מקש ימיני או ראשי, כאשר יש שני לחיצות ניצור מהלך חדש ונבדוק בלוח המדומה אם הוא מקיים את החוקים ונשווה אותו ללוח החדש. לאחר מכאן נעדכן את לוח המשחק ואת שאר הלוחות(לוח סימוני המשחק, ולוח אכילת כלי). |
| drawTile | Board board | פונקציה המציירת מחדש את התא וקוראת לפונקציות פנימיות. |
| assignTilePieceIcon | Board gameBoard | פונקציה המתאמת לכל משבצת תמונה של כלי המשחק בהתאם ללוח המשחק. |
| highlightLegals | Board board | פונקציה המתאמת בהתאם לבחירה בפריט התפריט ובהתאם לרשימת מהלכי החוקיים בלוח לכל משבצת נקודה ירוקה בהתאם לרשימת מהלכים של הכלי משחק שנבחר. |
| pieceLegalMoves | Board board | פונקציה המחזירה את רשימת המהלכים של הכלי שנבחר להזזה. |
| assignTileColor | אין | פונקציה המתאמת צבע לכל משבצת. |
| LogHistoryPanel |  | הורשה: Paneמחלקה המייצגת לוח סימוני מהלכי משחק בממשק הגרפי ויורשת מלוח גרפי. |
| LogHistoryPanel | אין | פונקציית הבנאי המאתחלת גודל הלוח ומייצרת טבלה גרפית עם שני עמודות אחת מייצגת מהלכים שנעשו על ידי השחקן השחור והשני על ידי השחקן הלבן. |
| add | Board board,Move move | פונקצייה המוסיפה מהלך ומציגה אותו על הטבלה הגרפית. |
| calculateCheckAndCheckMateHash | Board board | פונקציה המחזירה סימון של שח ושחמט כאשר הם מתקיימים בלוח. |
| Row |  | הורשה: איןמחלקה המייצגת שורה בטבלה(מהלך של השחקן השחור והלבן). |
| Row | String whiteMove,String blackMove | פונקציית הבנאי המאתחלת את סימון המהלך של השחקן השחור והלבן. |
| getWhiteMove | אין | פונקציה המחזירה את סימון המהלך של השחקן הלבן. |
| setWhiteMove | String whiteMove | פונקציה הקובעת את סימון המהלך של השחקן הלבן. |
| getBlackMove | אין | פונקציה המחזירה את סימון המהלך של השחקן השחור. |
| setBlackMove | String blackMove | פונקציה הקובעת את סימון המהלך של השחקן השחור. |
| TakenPiecesPanel |  | הורשה: VBoxמחלקה המייצגת לוח תמונות של כלי משחק שנאכלו. |
| TakenPiecesPanel | אין | פונקציית הבנאי המעדכנת את גודל הלוח ויוצרת שני תתי לוחות אחד מעל ואחד מתחת. |
| addTakenPiece | MoveLog moveLog | פונקציה המוסיפה את לכל תת לוח את תמונת הכלי שנאכל. |
| Main |  | הורשה: Applicationמחלקה המאתחלת את הממשק הגרפי . |
| start | Stage primaryStage | פונקציית המגדירה את הבמה ואת המסך שקשור אליו(מופעים הקשורים לספריית הממשק הגרפי). |
| main | String[] args | פונקציית הבנאי הקוראת לפונקציית Start. |

**4.4 קובץ** LIST **של התוכנה במחשב**

**Tile:**

**package** com.engine.board;  
  
**import** com.engine.pieces.Piece;  
  
**import** java.util.Map;  
**import** java.util.HashMap;  
**import** com.google.common.collect.ImmutableMap;  
  
*/\*\*  
 \* Tile class that is every Tile in the chess board that contains a piece  
 \*/***public abstract class** Tile {  
 **private static final** Map<Integer, EmptyTile> ***EMPTY\_TILES\_CACHE*** = *createAllPossibleEmptyTiles*();  
 **private final int tileCoordinate**;  
  
 **private** Tile(**final int** tileCoordinate) {  
 **this**.**tileCoordinate** = tileCoordinate;  
 }  
  
 **private static** Map<Integer, EmptyTile> createAllPossibleEmptyTiles() {  
 **final** Map<Integer, EmptyTile> emptyTileMap = **new** HashMap<>();  
 **for** (**int** i = 0; i < BoardUtils.***NUM\_TILES***; i++) {  
 emptyTileMap.put(i, **new** EmptyTile(i));  
 }  
 **return** ImmutableMap.*copyOf*(emptyTileMap);  
 }  
  
 **static** Tile createTile(**final int** tileCoordinate, **final** Piece piece) {  
 **return** (piece != **null**) ? **new** OccupiedTile(tileCoordinate, piece) : ***EMPTY\_TILES\_CACHE***.get(tileCoordinate);  
 }  
  
 **public abstract boolean** isTileOccupied();  
  
 **public abstract** Piece getPiece();  
  
 **public int** getTileCoordinate() {  
 **return this**.**tileCoordinate**;  
 }  
  
 **public static final class** EmptyTile **extends** Tile {  
 **private** EmptyTile(**final int** tileCoordinate) {  
 **super**(tileCoordinate);  
 }  
  
 **@Override  
 public boolean** isTileOccupied() {  
 **return false**;  
 }  
  
 **@Override  
 public** Piece getPiece() {  
 **return null**;  
 }  
  
 **@Override  
 public** String toString() {  
 **return "-"**;  
 }  
 }  
  
 **public static final class** OccupiedTile **extends** Tile {  
 **private final** Piece **pieceOnTile**;  
  
 **private** OccupiedTile(**final int** tileCoordinate, **final** Piece piece) {  
 **super**(tileCoordinate);  
 **this**.**pieceOnTile** = piece;  
 }  
  
 **@Override  
 public boolean** isTileOccupied() {  
 **return true**;  
 }  
  
 **@Override  
 public** Piece getPiece() {  
 **return this**.**pieceOnTile**;  
 }  
  
 **@Override  
 public** String toString() {  
 **return this**.getPiece().getPieceAlliance().isBlack() ?  
 **this**.getPiece().toString().toLowerCase() : **this**.getPiece().toString();  
 }  
 }  
}

**Board:**

**package** com.engine.board;  
  
**import** com.engine.Alliance;  
**import** com.engine.pieces.\*;  
**import** com.engine.player.BlackPlayer;  
**import** com.engine.player.Player;  
**import** com.engine.player.WhitePlayer;  
**import** com.google.common.collect.ImmutableList;  
**import** com.google.common.collect.Iterables;  
  
**import** java.util.\*;  
  
*/\*\*  
 \* Board class that represents the gameBoard using a static Builder class  
 \*/***public class** Board {  
 **private final** List<Tile> **gameBoard**;  
 **private final** Collection<Piece> **whitePieces**;  
 **private final** Collection<Piece> **blackPieces**;  
  
 **private final** WhitePlayer **whitePlayer**;  
 **private final** BlackPlayer **blackPlayer**;  
 **private final** Player **currentPlayer**;  
 **private final** Pawn **enPassantPawn**;  
  
 **private** Board(**final** Builder builder) {  
 **this**.**gameBoard** = *createGameBoard*(builder);  
 **this**.**whitePieces** = *calculateActivePieces*(**this**.**gameBoard**, Alliance.***WHITE***);  
 **this**.**blackPieces** = *calculateActivePieces*(**this**.**gameBoard**, Alliance.***BLACK***);  
 **this**.**enPassantPawn** = builder.**enPassantPawn**;  
 **final** Collection<Move> whiteStandardLegalMoves = calculateLegalMoves(**this**.**whitePieces**);  
 **final** Collection<Move> blackStandardLegalMoves = calculateLegalMoves(**this**.**blackPieces**);  
 **this**.**whitePlayer** = **new** WhitePlayer(**this**, whiteStandardLegalMoves, blackStandardLegalMoves);  
 **this**.**blackPlayer** = **new** BlackPlayer(**this**, whiteStandardLegalMoves, blackStandardLegalMoves);  
 **this**.**currentPlayer** = builder.**nextMoveMaker**.choosePlayer(**this**.**whitePlayer**, **this**.**blackPlayer**);  
 }  
  
 */\*\*  
 \* Creates a list of all possible active pieces(alive pieces on the board) for a given alliance(Black/White)  
 \** ***@param gameBoard*** *is needed to access its contents(tiles, pieces)  
 \** ***@param alliance*** *is determining which side is activePieces for  
 \** ***@return*** *a none changeable list of active pieces on the board  
 \*/* **private static** Collection<Piece> calculateActivePieces(**final** List<Tile> gameBoard, **final** Alliance alliance) {  
 **final** List<Piece> activePieces = **new** ArrayList<>();  
 **for**(**final** Tile tile : gameBoard) {  
 **if**(tile.isTileOccupied()) {  
 **final** Piece piece = tile.getPiece();  
 **if**(piece.getPieceAlliance() == alliance) {  
 activePieces.add(piece);  
 }  
 }  
 }  
 **return** ImmutableList.*copyOf*(activePieces);  
 }  
  
 */\*\*  
 \* creates all the tiles for the board  
 \** ***@param builder*** *is for accessing boardConfig  
 \** ***@return*** *a none changeable list of tiles  
 \*/* **private static** List<Tile> createGameBoard(**final** Builder builder) {  
 **final** Tile[] tiles = **new** Tile[BoardUtils.***NUM\_TILES***];  
 **for**(**int** i = 0; i < BoardUtils.***NUM\_TILES***; i++) {  
 tiles[i] = Tile.*createTile*(i, builder.**boardConfig**.get(i));  
 }  
 **return** ImmutableList.*copyOf*(tiles);  
 }  
  
 */\*\*  
 \* Creates all pieces and initialize the Board for the start of the game  
 \* and makes White start first  
 \*  
 \** ***@return*** *the board that  
 \*/* **public static** Board createStandardBoard() {  
 **final** Builder builder = **new** Builder();  
 *//Black layout* builder.setPiece(**new** Rook(0, Alliance.***BLACK***));  
 builder.setPiece(**new** Knight(1, Alliance.***BLACK***));  
 builder.setPiece(**new** Bishop(2, Alliance.***BLACK***));  
 builder.setPiece(**new** Queen(3, Alliance.***BLACK***));  
 builder.setPiece(**new** King(4, Alliance.***BLACK***));  
 builder.setPiece(**new** Bishop(5, Alliance.***BLACK***));  
 builder.setPiece(**new** Knight(6, Alliance.***BLACK***));  
 builder.setPiece(**new** Rook(7, Alliance.***BLACK***));  
 builder.setPiece(**new** Pawn(8, Alliance.***BLACK***));  
 builder.setPiece(**new** Pawn(9, Alliance.***BLACK***));  
 builder.setPiece(**new** Pawn(10, Alliance.***BLACK***));  
 builder.setPiece(**new** Pawn(11, Alliance.***BLACK***));  
 builder.setPiece(**new** Pawn(12, Alliance.***BLACK***));  
 builder.setPiece(**new** Pawn(13, Alliance.***BLACK***));  
 builder.setPiece(**new** Pawn(14, Alliance.***BLACK***));  
 builder.setPiece(**new** Pawn(15, Alliance.***BLACK***));  
 *//White layout* builder.setPiece(**new** Pawn(48, Alliance.***WHITE***));  
 builder.setPiece(**new** Pawn(49, Alliance.***WHITE***));  
 builder.setPiece(**new** Pawn(50, Alliance.***WHITE***));  
 builder.setPiece(**new** Pawn(51, Alliance.***WHITE***));  
 builder.setPiece(**new** Pawn(52, Alliance.***WHITE***));  
 builder.setPiece(**new** Pawn(53, Alliance.***WHITE***));  
 builder.setPiece(**new** Pawn(54, Alliance.***WHITE***));  
 builder.setPiece(**new** Pawn(55, Alliance.***WHITE***));  
 builder.setPiece(**new** Rook(56, Alliance.***WHITE***));  
 builder.setPiece(**new** Knight(57, Alliance.***WHITE***));  
 builder.setPiece(**new** Bishop(58, Alliance.***WHITE***));  
 builder.setPiece(**new** Queen(59, Alliance.***WHITE***));  
 builder.setPiece(**new** King(60, Alliance.***WHITE***));  
 builder.setPiece(**new** Bishop(61, Alliance.***WHITE***));  
 builder.setPiece(**new** Knight(62, Alliance.***WHITE***));  
 builder.setPiece(**new** Rook(63, Alliance.***WHITE***));  
 *//White moves first* builder.setMoveMaker(Alliance.***WHITE***);  
 **return** builder.build();  
 }  
  
 **public** Tile getTile (**final int** tileCoordinate) {  
 **return gameBoard**.get(tileCoordinate);  
 }  
  
 **public** Player getWhitePlayer() {  
 **return this**.**whitePlayer**;  
 }  
  
 **public** Player getBlackPlayer() {  
 **return this**.**blackPlayer**;  
 }  
  
 **public** Player getCurrentPlayer() {  
 **return this**.**currentPlayer**;  
 }  
  
 **public** Pawn getEnPassantPawn() {  
 **return this**.**enPassantPawn**;  
 }  
  
 **public** Collection<Piece> getBlackPieces() {  
 **return this**.**blackPieces**;  
 }  
  
 **public** Collection<Piece> getWhitePieces() {  
 **return this**.**whitePieces**;  
 }  
  
 */\*\*  
 \* Creates a list of all possible legal moves for a given collection of pieces we use this func  
 \* to calculate all legal moves of the whites pieces and all legal moves of the black pieces  
 \** ***@param pieces*** *for each piece we calculate its legal moves through its function  
 \* we implemented in each piece type  
 \** ***@return*** *a none changeable list of legal moves  
 \*/* **private** Collection<Move> calculateLegalMoves(**final** Collection<Piece> pieces) {  
 **final** List<Move> legalMoves = **new** ArrayList<>();  
 **for**(**final** Piece piece : pieces) {  
 legalMoves.addAll(piece.calculateLegalMoves(**this**));  
 }  
 **return** ImmutableList.*copyOf*(legalMoves);  
 }  
  
 Iterable<Move> getAllLegalMoves() {  
 **return** Iterables.*unmodifiableIterable*(Iterables.*concat*(  
 **this**.**whitePlayer**.getLegalMoves(), **this**.**blackPlayer**.getLegalMoves()));  
 }  
  
 **@Override  
 public** String toString() {  
 **final** StringBuilder sBuilder = **new** StringBuilder();  
 **for**(**int** i = 0; i < BoardUtils.***NUM\_TILES***; i++) {  
 **final** String tileText = **this**.**gameBoard**.get(i).toString();  
 sBuilder.append(String.*format*(**"%3s"**, tileText));  
 **if**((i + 1) % BoardUtils.***NUM\_TILES\_PER\_ROW*** == 0) {  
 sBuilder.append(**"\n"**);  
 }  
 }  
 **return** sBuilder.toString();  
 }  
  
 */\*\*  
 \* Using a static Builder class for the complex class Board so it will be easier to manage  
 \*/* **public static class** Builder {  
 **final** Map<Integer, Piece> **boardConfig**;  
 Alliance **nextMoveMaker**;  
 Pawn **enPassantPawn**;  
  
 Builder() {  
 **this**.**boardConfig** = **new** HashMap<>();  
 }  
  
 **public** Builder setPiece(**final** Piece piece) {  
 **this**.**boardConfig**.put(piece.getPiecePosition(), piece);  
 **return this**;  
 }  
 *//using the builder pattern* Builder setMoveMaker(**final** Alliance nextMoveMaker) {  
 **this**.**nextMoveMaker** = nextMoveMaker;  
 **return this**;  
 }  
  
 Board build() {  
 **return new** Board(**this**);  
 }  
  
 **void** setEnPassantPawn(Pawn EnPassantPawn) {  
 **this**.**enPassantPawn** = EnPassantPawn;  
 }  
 }  
}

**BoardUtils:**

**package** com.engine.board;  
  
**import** com.google.common.collect.ImmutableMap;  
  
**import** java.util.HashMap;  
**import** java.util.Map;  
  
*/\*\*  
 \* BoardUtils class that represents utilities for the board  
 \*/***public class** BoardUtils {  
 **public static final int *NUM\_TILES*** = 64;  
 **public static final int *NUM\_TILES\_PER\_ROW*** = 8;  
 **public static final boolean**[] ***FIRST\_FILE*** = *initColumn*(0);  
 **public static final boolean**[] ***SECOND\_FILE*** = *initColumn*(1);  
 **public static final boolean**[] ***SEVENTH\_FILE*** = *initColumn*(6);  
 **public static final boolean**[] ***EIGHTH\_FILE*** = *initColumn*(7);  
 **public static final boolean**[] ***EIGHTH\_RANK*** = *initRow*(0);  
 **public static final boolean**[] ***SEVENTH\_RANK*** = *initRow*(8);  
 **public static final boolean**[] ***SECOND\_RANK*** = *initRow*(48);  
 **public static final boolean**[] ***FIRST\_RANK*** = *initRow*(56);  
 **private static final** String[] ***ALGEBRAIC\_NOTATION*** = *initializeAlgebraicNotation*();  
 **private static final int *START\_TILE\_INDEX*** = 0;  
 **private static final** Map<String, Integer> ***POSITION\_TO\_COORDINATE*** = *initializePositionToCoordinateMap*();  
  
 **private** BoardUtils() {  
 **throw new** RuntimeException(**"You cannot instantiate me"**);  
 }  
  
 **private static boolean**[] initColumn(**int** columnNumber) {  
 **final boolean**[] column = **new boolean**[***NUM\_TILES***];  
 **do** {  
 column[columnNumber] = **true**;  
 columnNumber += ***NUM\_TILES\_PER\_ROW***;  
 } **while**(columnNumber < ***NUM\_TILES***);  
 **return** column;  
 }  
  
 **private static boolean**[] initRow(**int** rowNumber) {  
 **final boolean**[] row = **new boolean**[***NUM\_TILES***];  
 **do** {  
 row[rowNumber] = **true**;  
 rowNumber++;  
 } **while**(rowNumber % ***NUM\_TILES\_PER\_ROW*** != 0);  
 **return** row;  
 }  
  
 **private static** Map<String, Integer> initializePositionToCoordinateMap() {  
 **final** Map<String, Integer> positionToCoordinate = **new** HashMap<>();  
 **for** (**int** i = ***START\_TILE\_INDEX***; i < ***NUM\_TILES***; i++) {  
 positionToCoordinate.put(***ALGEBRAIC\_NOTATION***[i], i);  
 }  
 **return** ImmutableMap.*copyOf*(positionToCoordinate);  
 }  
  
 **private static** String[] initializeAlgebraicNotation() {  
 **return new** String[] {  
 **"a8"**, **"b8"**, **"c8"**, **"d8"**, **"e8"**, **"f8"**, **"g8"**, **"h8"**,  
 **"a7"**, **"b7"**, **"c7"**, **"d7"**, **"e7"**, **"f7"**, **"g7"**, **"h7"**,  
 **"a6"**, **"b6"**, **"c6"**, **"d6"**, **"e6"**, **"f6"**, **"g6"**, **"h6"**,  
 **"a5"**, **"b5"**, **"c5"**, **"d5"**, **"e5"**, **"f5"**, **"g5"**, **"h5"**,  
 **"a4"**, **"b4"**, **"c4"**, **"d4"**, **"e4"**, **"f4"**, **"g4"**, **"h4"**,  
 **"a3"**, **"b3"**, **"c3"**, **"d3"**, **"e3"**, **"f3"**, **"g3"**, **"h3"**,  
 **"a2"**, **"b2"**, **"c2"**, **"d2"**, **"e2"**, **"f2"**, **"g2"**, **"h2"**,  
 **"a1"**, **"b1"**, **"c1"**, **"d1"**, **"e1"**, **"f1"**, **"g1"**, **"h1"** };  
 }  
  
 **public static boolean** isValidTileCoordinate(**final int** coordinate) {  
 **return** coordinate >= 0 && coordinate < ***NUM\_TILES***;  
 }  
  
 **public static int** getCoordinateAtPosition(**final** String position) {  
 **return *POSITION\_TO\_COORDINATE***.get(position);  
 }  
  
 **static** String getPositionAtCoordinate(**final int** coordinate) {  
 **return *ALGEBRAIC\_NOTATION***[coordinate];  
 }  
  
 **public static boolean** isEndGame(**final** Board board) {  
 **return** board.getCurrentPlayer().isInCheckMate() ||  
 board.getCurrentPlayer().isInStaleMate();  
 }  
}

**Move:**

**package** com.engine.board;  
  
**import** com.engine.pieces.Pawn;  
**import** com.engine.pieces.Piece;  
**import** com.engine.pieces.Rook;  
  
**import static** com.engine.board.Board.\*;  
  
**public abstract class** Move {  
 **private static final** Move ***NULL\_MOVE*** = **new** NullMove();  
 **protected final** Board **board**;  
 **protected final boolean isFirstMove**;  
 **final** Piece **movedPiece**;  
 **final int destinationCoordinate**;  
  
 **private** Move(**final** Board board, **final** Piece movedPiece, **final int** destinationCoordinate) {  
 **this**.**board** = board;  
 **this**.**movedPiece** = movedPiece;  
 **this**.**destinationCoordinate** = destinationCoordinate;  
 **this**.**isFirstMove** = movedPiece.isFirstMove();  
 }  
  
 **private** Move(**final** Board board, **final int** destinationCoordinate) {  
 **this**.**board** = board;  
 **this**.**destinationCoordinate** = destinationCoordinate;  
 **this**.**movedPiece** = **null**;  
 **this**.**isFirstMove** = **false**;  
 }  
  
 **public int** getCurrentCoordinate() {  
 **return this**.getMovedPiece().getPiecePosition();  
 }  
  
 **public** Board getBoard() {  
 **return this**.**board**;  
 }  
  
 **public** Piece getMovedPiece() {  
 **return this**.**movedPiece**;  
 }  
  
 **public int** getDestinationCoordinate() {  
 **return this**.**destinationCoordinate**;  
 }  
  
 **public boolean** isAttack() {  
 **return false**;  
 }  
  
 **public boolean** isCastlingMove() {  
 **return false**;  
 }  
  
 **public** Piece getAttackedPiece() {  
 **return null**;  
 }  
  
 **@Override  
 public boolean** equals(**final** Object other) {  
 **if**(**this** == other) {  
 **return true**;  
 }  
 **if**(!(other **instanceof** Move)) {  
 **return false**;  
 }  
 **final** Move otherMove = (Move) other;  
 **return** getCurrentCoordinate() == otherMove.getCurrentCoordinate() &&  
 **this**.**destinationCoordinate** == otherMove.getDestinationCoordinate() &&  
 **this**.**movedPiece**.equals(otherMove.getMovedPiece());  
 }  
  
 **@Override  
 public int** hashCode() {  
 **final int** prime = 31;  
 **int** result = 1;  
 result = prime \* result + **this**.**destinationCoordinate**;  
 result = prime \* result + **this**.getMovedPiece().hashCode();  
 result = prime \* result + **this**.**movedPiece**.getPiecePosition();  
 **return** result;  
 }  
  
 */\*\*  
 \* Creates a new imgBoard(opposed to the default board) and will act as checking board for the default board  
 \* the players will makeMoves using makeMove func in Player class, on the imgBoard to check if it is valid  
 \* and if it is valid it will make them on the default board  
 \*  
 \* the function creates a builder(a helper class for board) and iterates on all currentPlayer active pieces(not dead pieces)  
 \* and on all the opponent active pieces and add them to the imgBoard. then it makes the testing move  
 \* and sets the move maker to the opposing team and returns the builder.build()(the board)  
 \** ***@return*** *the imgBoard that is created for testing moves  
 \*/* **public** Board execute() {  
 **final** Builder Builder = **new** Builder();  
 **for**(**final** Piece piece : **this**.**board**.getCurrentPlayer().getActivePieces()) {  
 **if**(!**this**.**movedPiece**.equals(piece)) {  
 Builder.setPiece(piece);  
 }  
 }  
 **for**(**final** Piece piece : **this**.**board**.getCurrentPlayer().getOpponent().getActivePieces()) {  
 Builder.setPiece(piece);  
 }  
 *//move the movedPiece the imgBoard* Builder.setPiece(**this**.**movedPiece**.movePiece(**this**));  
 Builder.setMoveMaker(**this**.**board**.getCurrentPlayer().getOpponent().getAlliance());  
 **return** Builder.build();  
 }  
  
 **public static class** AttackMove **extends** Move {  
 **final** Piece **attackedPiece**;  
  
 AttackMove(**final** Board board, **final** Piece movedPiece, **final int** destinationCoordinate, **final** Piece attackedPiece) {  
 **super**(board, movedPiece, destinationCoordinate);  
 **this**.**attackedPiece** = attackedPiece;  
 }  
  
 **@Override  
 public boolean** equals(**final** Object other) {  
 **if**(**this** == other) {  
 **return true**;  
 }  
 **if**(!(other **instanceof** AttackMove)) {  
 **return false**;  
 }  
 **final** AttackMove otherAttackMove = (AttackMove) other;  
 **return super**.equals(otherAttackMove) &&  
 **this**.getAttackedPiece().equals(otherAttackMove.getAttackedPiece());  
 }  
  
 **@Override  
 public int** hashCode() {  
 **return this**.**attackedPiece**.hashCode() + **super**.hashCode();  
 }  
  
 **@Override  
 public boolean** isAttack() {  
 **return true**;  
 }  
  
 **@Override  
 public** Piece getAttackedPiece() {  
 **return this**.**attackedPiece**;  
 }  
  
 **@Override  
 public** Board execute() {  
 **final** Builder builder = **new** Builder();  
 **for**(**final** Piece piece : **this**.**board**.getCurrentPlayer().getActivePieces()) {  
 **if**(!**this**.**movedPiece**.equals(piece)) {  
 builder.setPiece(piece);  
 }  
 }  
 **for**(**final** Piece piece : **this**.**board**.getCurrentPlayer().getOpponent().getActivePieces()) {  
 **if**(!piece.equals(**this**.getAttackedPiece())) {  
 builder.setPiece(piece);  
 }  
 }  
 builder.setPiece(**this**.**movedPiece**.movePiece(**this**));  
 builder.setMoveMaker(**this**.**board**.getCurrentPlayer().getOpponent().getAlliance());  
 **return** builder.build();  
 }  
 }  
  
 **public static final class** MajorMove **extends** Move {  
 **public** MajorMove(**final** Board board, **final** Piece movedPiece, **final int** destinationCoordinate) {  
 **super**(board, movedPiece, destinationCoordinate);  
 }  
  
 **@Override  
 public boolean** equals(**final** Object other) {  
 **return this** == other || other **instanceof** MajorMove && **super**.equals(other);  
 }  
  
 **@Override  
 public** String toString() {  
 **return movedPiece**.getPieceType().toString().toUpperCase() + BoardUtils.*getPositionAtCoordinate*(**this**.**destinationCoordinate**);  
 }  
 }  
  
 **public static class** MajorAttackMove **extends** AttackMove {  
 **public** MajorAttackMove(**final** Board board, **final** Piece movedPiece, **final int** destinationCoordinate, **final** Piece attackedPiece) {  
 **super**(board, movedPiece, destinationCoordinate, attackedPiece);  
 }  
  
 **@Override  
 public boolean** equals(**final** Object other) {  
 **return this** == other || other **instanceof** MajorAttackMove && **super**.equals(other);  
 }  
  
 **@Override  
 public** String toString() {  
 **return movedPiece**.getPieceType().toString().toUpperCase() + BoardUtils.*getPositionAtCoordinate*(**this**.**destinationCoordinate**);  
 }  
 }  
  
 **public static class** PawnMove **extends** Move {  
 **public** PawnMove(**final** Board board, **final** Piece movedPiece, **final int** destinationCoordinate) {  
 **super**(board, movedPiece, destinationCoordinate);  
 }  
  
 **@Override  
 public boolean** equals(**final** Object other) {  
 **return this** == other || other **instanceof** PawnMove && **super**.equals(other);  
 }  
  
 **@Override  
 public** String toString() {  
 **return** BoardUtils.*getPositionAtCoordinate*(**this**.**destinationCoordinate**);  
 }  
 }  
  
 **public static class** PawnAttackMove **extends** AttackMove {  
 **public** PawnAttackMove(**final** Board board, **final** Piece movedPiece, **final int** destinationCoordinate, **final** Piece attackedPiece) {  
 **super**(board, movedPiece, destinationCoordinate, attackedPiece);  
 }  
  
 **@Override  
 public boolean** equals(**final** Object other) {  
 **return this** == other || other **instanceof** PawnAttackMove && **super**.equals(other);  
 }  
  
 **@Override  
 public** String toString() {  
 **return** BoardUtils.*getPositionAtCoordinate*(**this**.**movedPiece**.getPiecePosition()).substring(0, 1) + **"x"** +  
 BoardUtils.*getPositionAtCoordinate*(**this**.**destinationCoordinate**);  
 }  
 }  
  
 **public static final class** PawnEnPassantAttack **extends** PawnAttackMove {  
 **public** PawnEnPassantAttack(**final** Board board, **final** Piece movedPiece, **final int** destinationCoordinate, **final** Piece attackedPiece) {  
 **super**(board, movedPiece, destinationCoordinate, attackedPiece);  
 }  
  
 **@Override  
 public boolean** equals(**final** Object other) {  
 **return this** == other || other **instanceof** PawnEnPassantAttack && **super**.equals(other);  
 }  
  
 **@Override  
 public** String toString() {  
 **return super**.toString() + **"e.p"**;  
 }  
 }  
  
 **public static final class** PawnJump **extends** PawnMove {  
 **public** PawnJump(**final** Board board, **final** Piece movedPiece, **final int** destinationCoordinate) {  
 **super**(board, movedPiece, destinationCoordinate);  
 }  
  
 **@Override  
 public boolean** equals(**final** Object other) {  
 **return this** == other || other **instanceof** PawnJump && **super**.equals(other);  
 }  
 */\*\*  
 \* Same purpose as before, but now we might detect that when making a pawn jump the opponent can maybe make an enPassant Attack  
 \** ***@return*** *the imgBoard  
 \*/* **@Override  
 public** Board execute() {  
 **final** Builder builder = **new** Builder();  
 **for**(**final** Piece piece : **this**.**board**.getCurrentPlayer().getActivePieces()) {  
 **if**(!**this**.**movedPiece**.equals(piece)) {  
 builder.setPiece(piece);  
 }  
 }  
 **for**(**final** Piece piece : **this**.**board**.getCurrentPlayer().getOpponent().getActivePieces()) {  
 builder.setPiece(piece);  
 }  
 **final** Pawn movedPawn = (Pawn)**this**.movedPiece.movePiece(**this**);  
 builder.setPiece(movedPawn);  
 builder.setEnPassantPawn(movedPawn);  
 builder.setMoveMaker(**this**.board.getCurrentPlayer().getOpponent().getAlliance());  
 **return** builder.build();  
 }  
 }  
  
 **public static class** PawnPromotion **extends** PawnMove {  
 *//using a decorator pattern the Move needs to be protected* **protected final** Move decoratedMove;  
 **final** Pawn promotedPawn;  
  
 **public** PawnPromotion(**final** Move decoratedMove) {  
 **super**(decoratedMove.getBoard(), decoratedMove.getMovedPiece(), decoratedMove.getDestinationCoordinate());  
 **this**.decoratedMove = decoratedMove;  
 **this**.promotedPawn = (Pawn) decoratedMove.getMovedPiece();  
 }  
  
 @Override  
 **public boolean** isAttack() {  
 **return this**.decoratedMove.isAttack();  
 }  
  
 @Override  
 **public** Piece getAttackedPiece() {  
 **return this**.decoratedMove.getAttackedPiece();  
 }  
  
 @Override  
 **public boolean** equals(**final** Object other) {  
 **return this** == other || other **instanceof** PawnPromotion && **super**.equals(other);  
 }  
  
 @Override  
 **public int** hashCode() {  
 **return** decoratedMove.hashCode() + (31 \* promotedPawn.hashCode());  
 }  
  
 @Override  
 **public** Board execute() {  
 **final** Board pawnMovedBoard = **this**.decoratedMove.execute();  
 **final** Builder builder = **new** Builder();  
 **for**(**final** Piece piece : pawnMovedBoard.getCurrentPlayer().getActivePieces()) {  
 **if**(!**this**.promotedPawn.equals(piece)) {  
 builder.setPiece(piece);  
 }  
 }  
 **for**(**final** Piece piece : pawnMovedBoard.getCurrentPlayer().getOpponent().getActivePieces()) {  
 builder.setPiece(piece);  
 }  
 builder.setPiece(**this**.promotedPawn.getPromotionPiece().movePiece(**this**));  
 builder.setMoveMaker(pawnMovedBoard.getCurrentPlayer().getAlliance());  
 **return** builder.build();  
 }  
  
 @Override  
 **public** String toString() {  
 **return super**.toString() + **"=Q"**;  
 }  
 }  
  
 **static abstract class** CastleMove **extends** Move {  
 **final** Rook castleRook;  
 **final int** CastleRookDestination;  
  
 CastleMove(**final** Board board, **final** Piece movedPiece, **final int** destinationCoordinate,  
 Rook castleRook, **int** castleRookDestination) {  
 **super**(board, movedPiece, destinationCoordinate);  
 **this**.castleRook = castleRook;  
 CastleRookDestination = castleRookDestination;  
 }  
  
  
 Rook getCastleRook() {  
 **return this**.castleRook;  
 }  
  
 @Override  
 **public boolean** isCastlingMove() {  
 **return true**;  
 }  
  
 @Override  
 **public boolean** equals(**final** Object other) {  
 **if**(**this** == other) {  
 **return true**;  
 }  
 **if**(!(other **instanceof** CastleMove)) {  
 **return false**;  
 }  
 **final** CastleMove otherCastleMove = (CastleMove)other;  
 **return super**.equals(otherCastleMove) && **this**.castleRook.equals(otherCastleMove.getCastleRook());  
 }  
  
 @Override  
 **public int** hashCode() {  
 **final int** prime = 31;  
 **int** result = **super**.hashCode();  
 result = prime \* result + **this**.castleRook.hashCode();  
 result = prime \* result + **this**.CastleRookDestination;  
 **return** result;  
 }  
  
 @Override  
 **public** Board execute() {  
 **final** Builder builder = **new** Builder();  
 **for**(**final** Piece piece : **this**.board.getCurrentPlayer().getActivePieces()) {  
 **if**(!**this**.movedPiece.equals(piece) && !**this**.castleRook.equals(piece)) {  
 builder.setPiece(piece);  
 }  
 }  
 **for**(**final** Piece piece : **this**.board.getCurrentPlayer().getOpponent().getActivePieces()) {  
 builder.setPiece(piece);  
 }  
 builder.setPiece(**this**.movedPiece.movePiece(**this**));  
 builder.setPiece(**new** Rook(**this**.CastleRookDestination, **this**.board.getCurrentPlayer().getAlliance()));  
 builder.setMoveMaker(**this**.board.getCurrentPlayer().getOpponent().getAlliance());  
 **return** builder.build();  
 }  
 }  
  
 **public static final class** KingSideCastleMove **extends** CastleMove {  
 **public** KingSideCastleMove(**final** Board board, **final** Piece movedPiece, **final int** destinationCoordinate,  
 Rook castleRook, **int** castleRookDestination) {  
 **super**(board, movedPiece, destinationCoordinate, castleRook, castleRookDestination);  
 }  
  
 @Override  
 **public boolean** equals(**final** Object other) {  
 **return this** == other || other **instanceof** KingSideCastleMove && **super**.equals(other);  
 }  
  
 @Override  
 **public** String toString() {  
 **return "0-0"**;  
 }  
 }  
  
 **public static final class** QueenSideCastleMove **extends** CastleMove {  
 **public** QueenSideCastleMove(**final** Board board, **final** Piece movedPiece, **final int** destinationCoordinate,  
 Rook castleRook, **int** castleRookDestination) {  
 **super**(board, movedPiece, destinationCoordinate, castleRook, castleRookDestination);  
 }  
  
 **@Override  
 public boolean** equals(**final** Object other) {  
 **return this** == other || other **instanceof** QueenSideCastleMove && **super**.equals(other);  
 }  
  
 **@Override  
 public** String toString() {  
 **return "0-0-0"**;  
 }  
 }  
  
 **public static final class** NullMove **extends** Move {  
 NullMove() {  
 **super**(**null**, -1);  
 }  
  
 **@Override  
 public int** getCurrentCoordinate() {  
 **return** -1;  
 }  
  
 **@Override  
 public** Board execute() {  
 **throw new** RuntimeException(**"cannot execute the null move"**);  
 }  
 }  
  
 **public static class** MoveFactory {  
 **private** MoveFactory() {  
 **throw new** RuntimeException(**"Not instantiable"**);  
 }  
  
 **public static** Move createMove(**final** Board board, **final int** currentCoordinate, **final int** destinationCoordinate) {  
 **for**(**final** Move move : board.getAllLegalMoves()) {  
 **if**(move.getCurrentCoordinate() == currentCoordinate && move.getDestinationCoordinate() == destinationCoordinate) {  
 **return** move;  
 }  
 }  
 **return** NULL\_MOVE;  
 }  
 }  
}

**Pawn:**

**package** com.engine.pieces;  
  
**import** com.engine.Alliance;  
**import** com.engine.board.Board;  
**import** com.engine.board.BoardUtils;  
**import** com.engine.board.Move;  
**import** com.google.common.collect.ImmutableList;  
  
**import** java.util.ArrayList;  
**import** java.util.Collection;  
**import** java.util.List;  
  
**import static** com.engine.board.Move.\*;  
  
**public class** Pawn **extends** Piece{  
 **private final static int**[] ***CANDIDATE\_MOVE\_COORDINATE*** = {7, 8, 9, 16};  
  
 **public** Pawn(**final int** piecePosition, **final** Alliance pieceAlliance) {  
 **super**(piecePosition, pieceAlliance, PieceType.***PAWN***, **true**);  
 }  
  
 **public** Pawn(**final int** piecePosition, **final** Alliance pieceAlliance, **final boolean** isFirstMove) {  
 **super**(piecePosition, pieceAlliance, PieceType.***PAWN***, isFirstMove);  
 }  
  
 */\*\*  
 \* Calculates all the legal(available) moves of the Pawn and return it as a list  
 \*  
 \* Iterates over all the offsets compared to the current position and checks if the candidateDestinationCoordinate  
 \* is on the board and check if the Pawn can move 1 step else if the Pawn can make 2 steps when it is the first move.  
 \* else we check if there's a specific offset(7,9) of an attack move  
 \*  
 \** ***@param board*** *is needed for access to the Tiles and Pieces on the board  
 \** ***@return*** *the list of legalMoves that cannot be change hench it is "final" and return as "Immutable.copyOf(legalMoves)"  
 \*/* **@Override  
 public** Collection<Move> calculateLegalMoves(**final** Board board) {  
 **final** List<Move> legalMoves = **new** ArrayList<>();  
 **for**(**final int** currentCandidateOffset : ***CANDIDATE\_MOVE\_COORDINATE***) {  
 **final int** candidateDestinationCoordinate = **this**.getPiecePosition() + (**this**.getPieceAlliance().getDirection() \* currentCandidateOffset);  
 **if**(!BoardUtils.*isValidTileCoordinate*(candidateDestinationCoordinate)) {  
 **continue**;  
 }  
 **if**(currentCandidateOffset == 8 && !board.getTile(candidateDestinationCoordinate).isTileOccupied()) {  
 **if**(**this**.getPieceAlliance().isPawnPromotionSquare(candidateDestinationCoordinate)) {  
 legalMoves.add(**new** PawnPromotion(**new** PawnMove(board, **this**, candidateDestinationCoordinate)));  
 } **else** {  
 legalMoves.add(**new** PawnMove(board, **this**, candidateDestinationCoordinate));  
 }  
 } **else if**(currentCandidateOffset == 16 && **this**.isFirstMove() &&  
 (BoardUtils.***SEVENTH\_RANK***[**this**.getPiecePosition()] && **this**.getPieceAlliance().isBlack() ||  
 (BoardUtils.***SECOND\_RANK***[**this**.getPiecePosition()] && **this**.getPieceAlliance().isWhite()))) {  
 **final int** behindCandidateDestinationCoordinate = **this**.getPiecePosition() + (**this**.getPieceAlliance().getDirection() \* 8);  
 **if**(!board.getTile(behindCandidateDestinationCoordinate).isTileOccupied() &&  
 !board.getTile(candidateDestinationCoordinate).isTileOccupied()) {  
 legalMoves.add(**new** PawnJump(board, **this**, candidateDestinationCoordinate));  
 }  
 } **else if**(currentCandidateOffset == 7 &&  
 !((BoardUtils.***EIGHTH\_FILE***[**this**.getPiecePosition()] && **this**.getPieceAlliance().isWhite() ||  
 (BoardUtils.***FIRST\_FILE***[**this**.getPiecePosition()] && **this**.getPieceAlliance().isBlack())))) {  
 **if**(board.getTile(candidateDestinationCoordinate).isTileOccupied()) {  
 **final** Piece pieceAtDestination = board.getTile(candidateDestinationCoordinate).getPiece();  
 **if**(**this**.getPieceAlliance() != pieceAtDestination.getPieceAlliance()) {  
 **if**(**this**.getPieceAlliance().isPawnPromotionSquare(candidateDestinationCoordinate)) {  
 legalMoves.add(**new** PawnPromotion(**new** PawnAttackMove(board, **this**, candidateDestinationCoordinate, pieceAtDestination)));  
 } **else** {  
 legalMoves.add(**new** PawnAttackMove(board, **this**, candidateDestinationCoordinate, pieceAtDestination));  
 }  
 }  
 } **else if**(board.getEnPassantPawn() != **null**) {  
 **if**(board.getEnPassantPawn().getPiecePosition() == (**this**.getPiecePosition() + (**this**.getPieceAlliance().getOppositeDirection()))) {  
 **final** Piece pieceOnCandidate = board.getEnPassantPawn();  
 **if**(**this**.getPieceAlliance() != pieceOnCandidate.getPieceAlliance()) {  
 legalMoves.add(**new** PawnEnPassantAttack(board, **this**, candidateDestinationCoordinate, pieceOnCandidate));  
 }  
 }  
 }  
 } **else if**(currentCandidateOffset == 9 &&  
 !((BoardUtils.***FIRST\_FILE***[**this**.getPiecePosition()] && **this**.getPieceAlliance().isWhite() ||  
 (BoardUtils.***EIGHTH\_FILE***[**this**.getPiecePosition()] && **this**.getPieceAlliance().isBlack())))) {  
 **if**(board.getTile(candidateDestinationCoordinate).isTileOccupied()) {  
 **final** Piece pieceAtDestination = board.getTile(candidateDestinationCoordinate).getPiece();  
 **if**(**this**.getPieceAlliance() != pieceAtDestination.getPieceAlliance()) {  
 **if**(**this**.getPieceAlliance().isPawnPromotionSquare(candidateDestinationCoordinate)) {  
 legalMoves.add(**new** PawnPromotion(**new** PawnAttackMove(board, **this**, candidateDestinationCoordinate, pieceAtDestination)));  
 } **else** {  
 legalMoves.add(**new** PawnAttackMove(board, **this**, candidateDestinationCoordinate, pieceAtDestination));  
 }  
 }  
 } **else if**(board.getEnPassantPawn() != **null**) {  
 **if**(board.getEnPassantPawn().getPiecePosition() == (**this**.getPiecePosition() - (**this**.getPieceAlliance().getOppositeDirection()))) {  
 **final** Piece pieceOnCandidate = board.getEnPassantPawn();  
 **if**(**this**.getPieceAlliance() != pieceOnCandidate.getPieceAlliance()) {  
 legalMoves.add(**new** PawnEnPassantAttack(board, **this**, candidateDestinationCoordinate, pieceOnCandidate));  
 }  
 }  
 }  
 }  
 }  
 **return** ImmutableList.*copyOf*(legalMoves);  
 }  
  
 **public** Piece getPromotionPiece() {  
 **return new** Queen(**this**.getPiecePosition(), **this**.getPieceAlliance(), **false**);  
 }  
  
 **@Override  
 public** Pawn movePiece(**final** Move move) {  
 **return new** Pawn(move.getDestinationCoordinate(), move.getMovedPiece().getPieceAlliance());  
 }  
  
 **@Override  
 public** String toString() {  
 **return** PieceType.***PAWN***.toString();  
 }  
}

**PieceType:**

**package** com.engine.pieces;  
  
**public enum** PieceType {  
 ***PAWN***(**"p"**, 1) {  
 **@Override  
 public boolean** isKing() {  
 **return false**;  
 }  
  
 **@Override  
 public boolean** isRook() {  
 **return false**;  
 }  
 },  
 ***KNIGHT***(**"n"**, 3) {  
 **@Override  
 public boolean** isKing() {  
 **return false**;  
 }  
  
 **@Override  
 public boolean** isRook() {  
 **return false**;  
 }  
 },  
 ***BISHOP***(**"b"**, 3) {  
 **@Override  
 public boolean** isKing() {  
 **return false**;  
 }  
  
 **@Override  
 public boolean** isRook() {  
 **return false**;  
 }  
 },  
 ***ROOK***(**"r"**, 5) {  
 **@Override  
 public boolean** isKing() {  
 **return false**;  
 }  
  
 **@Override  
 public boolean** isRook() {  
 **return true**;  
 }  
 },  
 ***QUEEN***(**"q"**, 9) {  
 **@Override  
 public boolean** isKing() {  
 **return false**;  
 }  
  
 **@Override  
 public boolean** isRook() {  
 **return false**;  
 }  
 },  
 ***KING***(**"k"**, 20) {  
 **@Override  
 public boolean** isKing() {  
 **return true**;  
 }  
  
 **@Override  
 public boolean** isRook() {  
 **return false**;  
 }  
 };  
  
 **private final** String **pieceName**;  
 **private final int pieceValue**;  
  
 PieceType(**final** String pieceName, **final int** pieceValue) {  
 **this**.**pieceName** = pieceName;  
 **this**.**pieceValue** = pieceValue;  
 }  
  
 **public int** getPieceValue() {  
 **return this**.**pieceValue**;  
 }  
  
 **@Override  
 public** String toString() {  
 **return this**.**pieceName**;  
 }  
  
 **public abstract boolean** isKing();  
 **public abstract boolean** isRook();  
}

**Bishop:**

**package** com.engine.pieces;  
  
**import** com.engine.Alliance;  
**import** com.engine.board.Board;  
**import** com.engine.board.BoardUtils;  
**import** com.engine.board.Move;  
**import** com.engine.board.Tile;  
**import** com.google.common.collect.ImmutableList;  
  
**import** java.util.ArrayList;  
**import** java.util.Collection;  
**import** java.util.List;  
  
**import static** com.engine.board.Move.\*;  
  
**public class** Bishop **extends** Piece {  
 **private final static int**[] ***CANDIDATE\_MOVE\_VECTOR\_COORDINATE*** = {-9, -7, 7, 9};  
  
 **public** Bishop(**final int** piecePosition, **final** Alliance pieceAlliance) {  
 **super**(piecePosition, pieceAlliance, PieceType.***BISHOP***, **true**);  
 }  
  
 **public** Bishop(**final int** piecePosition, **final** Alliance pieceAlliance, **final boolean** isFirstMove) {  
 **super**(piecePosition, pieceAlliance, PieceType.***BISHOP***, isFirstMove);  
 }  
  
 **private static boolean** isFirstColumnExclusion(**final int** currentPosition, **final int** candidateOffset) {  
 **return** BoardUtils.***FIRST\_FILE***[currentPosition] && (candidateOffset == -9 || candidateOffset == 7);  
 }  
  
 **private static boolean** isEighthColumnExclusion(**final int** currentPosition, **final int** candidateOffset) {  
 **return** BoardUtils.***EIGHTH\_FILE***[currentPosition] && (candidateOffset == -7 || candidateOffset == 9);  
 }  
  
 */\*\*  
 \* Calculates all the legal(available) moves of the Bishop and return it as a list  
 \* <p>  
 \* Iterates over all the offsets compared to the current position and compared to the knight it add the offsets again and  
 \* again until the candidateDestinationCoordinate is not on the Board and checks if the candidateDestinationCoordinate  
 \* is on the board and the exceptions are checks against the offset rules(because they do not always apply)  
 \* and breaks the loop if they don't apply.  
 \* after that we check if the candidateDestinationCoordinate has a Piece, if not we add a legal MajorMove and else it  
 \* checks whether the Piece is an enemy, if it is we add a AttackMove and because it has a Piece we don't want to  
 \* keep adding offsets because the pieceAtDestination is blocking the rest of the available Moves of the Bishop.  
 \*  
 \** ***@param board*** *is needed for access to the Tiles and Pieces on the board  
 \** ***@return*** *the list of legalMoves that cannot be change hench it is "final" and return as "Immutable.copyOf(legalMoves)"  
 \*/* **@Override  
 public** Collection<Move> calculateLegalMoves(**final** Board board) {  
 **final** List<Move> legalMoves = **new** ArrayList<>();  
 **for** (**final int** currentCandidateOffset : ***CANDIDATE\_MOVE\_VECTOR\_COORDINATE***) {  
 **int** candidateDestinationCoordinate = **this**.getPiecePosition();  
 **while** (BoardUtils.*isValidTileCoordinate*(candidateDestinationCoordinate)) {  
 **if** (*isFirstColumnExclusion*(candidateDestinationCoordinate, currentCandidateOffset) ||  
 *isEighthColumnExclusion*(candidateDestinationCoordinate, currentCandidateOffset)) {  
 **break**;  
 }  
 candidateDestinationCoordinate += currentCandidateOffset;  
 **if** (BoardUtils.*isValidTileCoordinate*(candidateDestinationCoordinate)) {  
 **final** Tile candidateDestinationTile = board.getTile(candidateDestinationCoordinate);  
 **if** (!candidateDestinationTile.isTileOccupied()) { *// new Move if Tile is empty* legalMoves.add(**new** MajorMove(board, **this**,  
 candidateDestinationCoordinate));  
 } **else** {  
 **final** Piece pieceAtDestination = candidateDestinationTile.getPiece();  
 **final** Alliance pieceAlliance = pieceAtDestination.getPieceAlliance();  
 **if** (**this**.getPieceAlliance() != pieceAlliance) { *// new Move if on the destination Tile there is an enemy* legalMoves.add(**new** MajorAttackMove(board, **this**,  
 candidateDestinationCoordinate, pieceAtDestination));  
 }  
 **break**;  
 }  
 }  
 }  
 }  
 **return** ImmutableList.*copyOf*(legalMoves);  
 }  
  
 **@Override  
 public** Bishop movePiece(**final** Move move) {  
 **return new** Bishop(move.getDestinationCoordinate(), move.getMovedPiece().getPieceAlliance());  
 }  
  
 **@Override  
 public** String toString() {  
 **return** PieceType.***BISHOP***.toString();  
 }  
}

**King:**

**package** com.engine.pieces;  
  
**import** com.engine.Alliance;  
**import** com.engine.board.Board;  
**import** com.engine.board.BoardUtils;  
**import** com.engine.board.Move;  
**import** com.engine.board.Tile;  
**import** com.google.common.collect.ImmutableList;  
  
**import** java.util.ArrayList;  
**import** java.util.Collection;  
**import** java.util.List;  
  
**import static** com.engine.board.Move.\*;  
  
**public class** King **extends** Piece {  
 **private final static int**[] ***CANDIDATE\_MOVE\_COORDINATE*** = {-9, -8, -7, -1, 1, 7, 8, 9};  
  
 **public** King(**final int** piecePosition, **final** Alliance pieceAlliance) {  
 **super**(piecePosition, pieceAlliance, PieceType.***KING***, **true**);  
 }  
  
 **public** King(**final int** piecePosition, **final** Alliance pieceAlliance, **final boolean** isFirstMove) {  
 **super**(piecePosition, pieceAlliance, PieceType.***KING***, isFirstMove);  
 }  
  
 **private static boolean** isFirstColumnExclusion(**final int** currentPosition, **final int** candidateOffset) {  
 **return** BoardUtils.***FIRST\_FILE***[currentPosition] && (candidateOffset == -9 || candidateOffset == -1 ||  
 candidateOffset == 7);  
 }  
  
 **private static boolean** isEighthColumnExclusion(**final int** currentPosition, **final int** candidateOffset) {  
 **return** BoardUtils.***EIGHTH\_FILE***[currentPosition] && (candidateOffset == -7 || candidateOffset == 1 ||  
 candidateOffset == 9);  
 }  
  
 */\*\*  
 \* Calculates all the legal(available) moves of the King and return it as a list  
 \* <p>  
 \* Iterates over all the offsets compared to the current position and compared to the knight it add the offsets again and  
 \* again until the candidateDestinationCoordinate is not on the Board and checks if the candidateDestinationCoordinate  
 \* is on the board and the exceptions are checks against the offset rules(because they do not always apply)  
 \* and breaks the loop if they don't apply.  
 \* after that we check if the candidateDestinationCoordinate has a Piece, if not we add a legal MajorMove and else it  
 \* checks whether the Piece is an enemy, if it is we add a AttackMove and because it has a Piece we don't want to  
 \* keep adding offsets because the pieceAtDestination is blocking the rest of the available Moves of the Bishop.  
 \*  
 \** ***@param board*** *is needed for access to the Tiles and Pieces on the board  
 \** ***@return*** *the list of legalMoves that cannot be change hench it is "final" and return as "Immutable.copyOf(legalMoves)"  
 \*/* **@Override  
 public** Collection<Move> calculateLegalMoves(**final** Board board) {  
 **final** List<Move> legalMoves = **new** ArrayList<>();  
 **for**(**final int** currentCandidateOffset : ***CANDIDATE\_MOVE\_COORDINATE***) {  
 **final int** candidateDestinationCoordinate = **this**.getPiecePosition() + currentCandidateOffset;  
 **if**(BoardUtils.*isValidTileCoordinate*(candidateDestinationCoordinate)) {  
 **if**(*isFirstColumnExclusion*(**this**.getPiecePosition(), currentCandidateOffset) ||  
 *isEighthColumnExclusion*(**this**.getPiecePosition(), currentCandidateOffset)) {  
 **continue**;  
 }  
 **final** Tile candidateDestinationTile = board.getTile(candidateDestinationCoordinate);  
 **if**(!candidateDestinationTile.isTileOccupied()) { *// new Move if Tile is empty* legalMoves.add(**new** MajorMove(board, **this**, candidateDestinationCoordinate));  
 } **else** {  
 **final** Piece pieceAtDestination = candidateDestinationTile.getPiece();  
 **if**(**this**.getPieceAlliance() != pieceAtDestination.getPieceAlliance()) { *// new Move if on the destination Tile there is an enemy* legalMoves.add(**new** MajorAttackMove(board, **this**, candidateDestinationCoordinate, pieceAtDestination));  
 }  
 }  
 }  
 }  
 **return** ImmutableList.*copyOf*(legalMoves);  
 }  
  
 **@Override  
 public** King movePiece(**final** Move move) {  
 **return new** King(move.getDestinationCoordinate(), move.getMovedPiece().getPieceAlliance());  
 }  
  
 **@Override  
 public** String toString() {  
 **return** PieceType.***KING***.toString();  
 }  
}

**Knight:**

**package** com.engine.pieces;  
  
**import** com.engine.Alliance;  
**import** com.engine.board.Board;  
**import** com.engine.board.BoardUtils;  
**import** com.engine.board.Move;  
**import** com.engine.board.Tile;  
**import** com.google.common.collect.ImmutableList;  
  
**import** java.util.ArrayList;  
**import** java.util.Collection;  
**import** java.util.List;  
  
**import static** com.engine.board.Move.\*;  
  
**public class** Knight **extends** Piece {  
 *// offsets for knight with the perspective of its position* **private static final int**[] ***CANDIDATE\_MOVE\_COORDINATE*** = {-17, -15, -10, -6, 6, 10, 15, 17};  
  
 **public** Knight(**final int** piecePosition, **final** Alliance pieceAlliance) {  
 **super**(piecePosition, pieceAlliance, PieceType.***KNIGHT***, **true**);  
 }  
  
 **public** Knight(**final int** piecePosition, **final** Alliance pieceAlliance, **final boolean** isFirstMove) {  
 **super**(piecePosition, pieceAlliance, PieceType.***KNIGHT***, isFirstMove);  
 }  
  
 **private static boolean** isFirstColumnExclusion(**final int** currentPosition, **final int** candidateOffset) {  
 **return** BoardUtils.***FIRST\_FILE***[currentPosition] && (candidateOffset == -17 || candidateOffset == -10 ||  
 candidateOffset == 6 || candidateOffset == 15);  
 }  
  
 **private static boolean** isSecondColumnExclusion(**final int** currentPosition, **final int** candidateOffset) {  
 **return** BoardUtils.***SECOND\_FILE***[currentPosition] && (candidateOffset == -10 || candidateOffset == 6);  
 }  
  
 **private static boolean** isSeventhColumnExclusion(**final int** currentPortion, **final int** candidateOffset) {  
 **return** BoardUtils.***SEVENTH\_FILE***[currentPortion] && (candidateOffset == -6 || candidateOffset == 10);  
 }  
  
 **private static boolean** isEighthColumnExclusion(**final int** currentPosition, **final int** candidateOffset) {  
 **return** BoardUtils.***EIGHTH\_FILE***[currentPosition] && (candidateOffset == -15 || candidateOffset == -6 ||  
 candidateOffset == 10 || candidateOffset == 17);  
 }  
  
 */\*\*  
 \* Calculates all the legal(available) moves of the knight and return it as a list  
 \*  
 \* Iterates over all the offsets compared to the current position and checks if the candidateDestinationCoordinate  
 \* is on the board and the exceptions are checks against the offset rules(because they do not always apply).  
 \* after that we check if the candidateDestinationCoordinate has a Piece, if not we add a legal MajorMove and else it  
 \* checks whether the Piece is an enemy, if it is we add a AttackMove.  
 \*  
 \** ***@param board*** *is needed for access to the Tiles and Pieces on the board  
 \** ***@return*** *the list of legalMoves that cannot be change hench it is "final" and return as "Immutable.copyOf(legalMoves)"  
 \*/* **@Override  
 public** Collection<Move> calculateLegalMoves(**final** Board board) {  
 **final** List<Move> legalMoves = **new** ArrayList<>();  
 **for**(**final int** currentCandidateOffset : ***CANDIDATE\_MOVE\_COORDINATE***) {  
 **final int** candidateDestinationCoordinate = **this**.getPiecePosition() + currentCandidateOffset;  
 **if**(BoardUtils.*isValidTileCoordinate*(candidateDestinationCoordinate)) {  
 **if**(*isFirstColumnExclusion*(**this**.getPiecePosition(), currentCandidateOffset) ||  
 *isSecondColumnExclusion*(**this**.getPiecePosition(), currentCandidateOffset) ||  
 *isSeventhColumnExclusion*(**this**.getPiecePosition(), currentCandidateOffset) ||  
 *isEighthColumnExclusion*(**this**.getPiecePosition(), currentCandidateOffset)) {  
 **continue**;  
 }  
 **final** Tile candidateDestinationTile = board.getTile(candidateDestinationCoordinate);  
 **if**(!candidateDestinationTile.isTileOccupied()) { *// new Move if Tile is empty* legalMoves.add(**new** MajorMove(board, **this**, candidateDestinationCoordinate));  
 } **else** {  
 **final** Piece pieceAtDestination = candidateDestinationTile.getPiece();  
 **if**(**this**.getPieceAlliance() != pieceAtDestination.getPieceAlliance()) { *// new Move if on the destination Tile there is an enemy* legalMoves.add(**new** MajorAttackMove(board, **this**, candidateDestinationCoordinate, pieceAtDestination));  
 }  
 }  
 }  
 }  
 **return** ImmutableList.*copyOf*(legalMoves);  
 }  
  
 **@Override  
 public** Knight movePiece(**final** Move move) {  
 **return new** Knight(move.getDestinationCoordinate(), move.getMovedPiece().getPieceAlliance());  
 }  
  
 **@Override  
 public** String toString() {  
 **return** PieceType.***KNIGHT***.toString();  
 }  
}

**Pawn:**

**package** com.engine.pieces;  
  
**import** com.engine.Alliance;  
**import** com.engine.board.Board;  
**import** com.engine.board.BoardUtils;  
**import** com.engine.board.Move;  
**import** com.google.common.collect.ImmutableList;  
  
**import** java.util.ArrayList;  
**import** java.util.Collection;  
**import** java.util.List;  
  
**import static** com.engine.board.Move.\*;  
  
**public class** Pawn **extends** Piece{  
 **private final static int**[] ***CANDIDATE\_MOVE\_COORDINATE*** = {7, 8, 9, 16};  
  
 **public** Pawn(**final int** piecePosition, **final** Alliance pieceAlliance) {  
 **super**(piecePosition, pieceAlliance, PieceType.***PAWN***, **true**);  
 }  
  
 **public** Pawn(**final int** piecePosition, **final** Alliance pieceAlliance, **final boolean** isFirstMove) {  
 **super**(piecePosition, pieceAlliance, PieceType.***PAWN***, isFirstMove);  
 }  
  
 */\*\*  
 \* Calculates all the legal(available) moves of the Pawn and return it as a list  
 \*  
 \* Iterates over all the offsets compared to the current position and checks if the candidateDestinationCoordinate  
 \* is on the board and check if the Pawn can move 1 step else if the Pawn can make 2 steps when it is the first move.  
 \* else we check if there's a specific offset(7,9) of an attack move  
 \*  
 \** ***@param board*** *is needed for access to the Tiles and Pieces on the board  
 \** ***@return*** *the list of legalMoves that cannot be change hench it is "final" and return as "Immutable.copyOf(legalMoves)"  
 \*/* **@Override  
 public** Collection<Move> calculateLegalMoves(**final** Board board) {  
 **final** List<Move> legalMoves = **new** ArrayList<>();  
 **for**(**final int** currentCandidateOffset : ***CANDIDATE\_MOVE\_COORDINATE***) {  
 **final int** candidateDestinationCoordinate = **this**.getPiecePosition() + (**this**.getPieceAlliance().getDirection() \* currentCandidateOffset);  
 **if**(!BoardUtils.*isValidTileCoordinate*(candidateDestinationCoordinate)) {  
 **continue**;  
 }  
 **if**(currentCandidateOffset == 8 && !board.getTile(candidateDestinationCoordinate).isTileOccupied()) {  
 **if**(**this**.getPieceAlliance().isPawnPromotionSquare(candidateDestinationCoordinate)) {  
 legalMoves.add(**new** PawnPromotion(**new** PawnMove(board, **this**, candidateDestinationCoordinate)));  
 } **else** {  
 legalMoves.add(**new** PawnMove(board, **this**, candidateDestinationCoordinate));  
 }  
 } **else if**(currentCandidateOffset == 16 && **this**.isFirstMove() &&  
 (BoardUtils.***SEVENTH\_RANK***[**this**.getPiecePosition()] && **this**.getPieceAlliance().isBlack() ||  
 (BoardUtils.***SECOND\_RANK***[**this**.getPiecePosition()] && **this**.getPieceAlliance().isWhite()))) {  
 **final int** behindCandidateDestinationCoordinate = **this**.getPiecePosition() + (**this**.getPieceAlliance().getDirection() \* 8);  
 **if**(!board.getTile(behindCandidateDestinationCoordinate).isTileOccupied() &&  
 !board.getTile(candidateDestinationCoordinate).isTileOccupied()) {  
 legalMoves.add(**new** PawnJump(board, **this**, candidateDestinationCoordinate));  
 }  
 } **else if**(currentCandidateOffset == 7 &&  
 !((BoardUtils.***EIGHTH\_FILE***[**this**.getPiecePosition()] && **this**.getPieceAlliance().isWhite() ||  
 (BoardUtils.***FIRST\_FILE***[**this**.getPiecePosition()] && **this**.getPieceAlliance().isBlack())))) {  
 **if**(board.getTile(candidateDestinationCoordinate).isTileOccupied()) {  
 **final** Piece pieceAtDestination = board.getTile(candidateDestinationCoordinate).getPiece();  
 **if**(**this**.getPieceAlliance() != pieceAtDestination.getPieceAlliance()) {  
 **if**(**this**.getPieceAlliance().isPawnPromotionSquare(candidateDestinationCoordinate)) {  
 legalMoves.add(**new** PawnPromotion(**new** PawnAttackMove(board, **this**, candidateDestinationCoordinate, pieceAtDestination)));  
 } **else** {  
 legalMoves.add(**new** PawnAttackMove(board, **this**, candidateDestinationCoordinate, pieceAtDestination));  
 }  
 }  
 } **else if**(board.getEnPassantPawn() != **null**) {  
 **if**(board.getEnPassantPawn().getPiecePosition() == (**this**.getPiecePosition() + (**this**.getPieceAlliance().getOppositeDirection()))) {  
 **final** Piece pieceOnCandidate = board.getEnPassantPawn();  
 **if**(**this**.getPieceAlliance() != pieceOnCandidate.getPieceAlliance()) {  
 legalMoves.add(**new** PawnEnPassantAttack(board, **this**, candidateDestinationCoordinate, pieceOnCandidate));  
 }  
 }  
 }  
 } **else if**(currentCandidateOffset == 9 &&  
 !((BoardUtils.***FIRST\_FILE***[**this**.getPiecePosition()] && **this**.getPieceAlliance().isWhite() ||  
 (BoardUtils.***EIGHTH\_FILE***[**this**.getPiecePosition()] && **this**.getPieceAlliance().isBlack())))) {  
 **if**(board.getTile(candidateDestinationCoordinate).isTileOccupied()) {  
 **final** Piece pieceAtDestination = board.getTile(candidateDestinationCoordinate).getPiece();  
 **if**(**this**.getPieceAlliance() != pieceAtDestination.getPieceAlliance()) {  
 **if**(**this**.getPieceAlliance().isPawnPromotionSquare(candidateDestinationCoordinate)) {  
 legalMoves.add(**new** PawnPromotion(**new** PawnAttackMove(board, **this**, candidateDestinationCoordinate, pieceAtDestination)));  
 } **else** {  
 legalMoves.add(**new** PawnAttackMove(board, **this**, candidateDestinationCoordinate, pieceAtDestination));  
 }  
 }  
 } **else if**(board.getEnPassantPawn() != **null**) {  
 **if**(board.getEnPassantPawn().getPiecePosition() == (**this**.getPiecePosition() - (**this**.getPieceAlliance().getOppositeDirection()))) {  
 **final** Piece pieceOnCandidate = board.getEnPassantPawn();  
 **if**(**this**.getPieceAlliance() != pieceOnCandidate.getPieceAlliance()) {  
 legalMoves.add(**new** PawnEnPassantAttack(board, **this**, candidateDestinationCoordinate, pieceOnCandidate));  
 }  
 }  
 }  
 }  
 }  
 **return** ImmutableList.*copyOf*(legalMoves);  
 }  
  
 **public** Piece getPromotionPiece() {  
 **return new** Queen(**this**.getPiecePosition(), **this**.getPieceAlliance(), **false**);  
 }  
  
 **@Override  
 public** Pawn movePiece(**final** Move move) {  
 **return new** Pawn(move.getDestinationCoordinate(), move.getMovedPiece().getPieceAlliance());  
 }  
  
 **@Override  
 public** String toString() {  
 **return** PieceType.***PAWN***.toString();  
 }  
}

**Queen:**

**package** com.engine.pieces;  
  
**import** com.engine.Alliance;  
**import** com.engine.board.Board;  
**import** com.engine.board.BoardUtils;  
**import** com.engine.board.Move;  
**import** com.engine.board.Tile;  
**import** com.google.common.collect.ImmutableList;  
  
**import** java.util.ArrayList;  
**import** java.util.Collection;  
**import** java.util.List;  
  
**import static** com.engine.board.Move.\*;  
  
**public class** Queen **extends** Piece{  
  
 **private final static int**[] ***CANDIDATE\_MOVE\_VECTOR\_COORDINATE*** = {-9, -8, -7, -1, 1, 7, 8, 9};  
  
 **public** Queen(**final int** piecePosition, **final** Alliance pieceAlliance) {  
 **super**(piecePosition, pieceAlliance, PieceType.***QUEEN***, **true**);  
 }  
  
 Queen(**final int** piecePosition, **final** Alliance pieceAlliance, **final boolean** isFirstMove) {  
 **super**(piecePosition, pieceAlliance, PieceType.***QUEEN***, isFirstMove);  
 }  
  
 **private static boolean** isFirstColumnExclusion(**final int** currentPosition, **final int** candidateOffset) {  
 **return** BoardUtils.***FIRST\_FILE***[currentPosition] && (candidateOffset == -9 || candidateOffset == -1 || candidateOffset == 7);  
 }  
  
 **private static boolean** isEighthColumnExclusion(**final int** currentPosition, **final int** candidateOffset) {  
 **return** BoardUtils.***EIGHTH\_FILE***[currentPosition] && (candidateOffset == -7 || candidateOffset == 1 || candidateOffset == 9);  
 }  
  
 */\*\*  
 \* Calculates all the legal(available) moves of the Queen and return it as a list  
 \*  
 \* Iterates over all the offsets compared to the current position and compared to the knight it add the offsets again and  
 \* again until the candidateDestinationCoordinate is not on the Board and checks if the candidateDestinationCoordinate  
 \* is on the board and the exceptions are checks against the offset rules(because they do not always apply)  
 \* and breaks the loop if they don't apply.  
 \* after that we check if the candidateDestinationCoordinate has a Piece, if not we add a legal MajorMove and else it  
 \* checks whether the Piece is an enemy, if it is we add a AttackMove and because it has a Piece we don't want to  
 \* keep adding offsets because the pieceAtDestination is blocking the rest of the available Moves of the Queen.  
 \*  
 \** ***@param board*** *is needed for access to the Tiles and Pieces on the board  
 \** ***@return*** *the list of legalMoves that cannot be change hench it is "final" and return as "Immutable.copyOf(legalMoves)"  
 \*/* **@Override  
 public** Collection<Move> calculateLegalMoves(**final** Board board) {  
 **final** List<Move> legalMoves = **new** ArrayList<>();  
 **for** (**final int** currentCandidateOffset : ***CANDIDATE\_MOVE\_VECTOR\_COORDINATE***) {  
 **int** candidateDestinationCoordinate = **this**.getPiecePosition();  
 **while** (BoardUtils.*isValidTileCoordinate*(candidateDestinationCoordinate)) {  
 **if** (*isFirstColumnExclusion*(candidateDestinationCoordinate, currentCandidateOffset) ||  
 *isEighthColumnExclusion*(candidateDestinationCoordinate, currentCandidateOffset)) {  
 **break**;  
 }  
 candidateDestinationCoordinate += currentCandidateOffset;  
 **if** (BoardUtils.*isValidTileCoordinate*(candidateDestinationCoordinate)) {  
 **final** Tile candidateDestinationTile = board.getTile(candidateDestinationCoordinate);  
 **if** (!candidateDestinationTile.isTileOccupied()) { *// new Move if Tile is empty* legalMoves.add(**new** MajorMove(board, **this**,  
 candidateDestinationCoordinate));  
 } **else** {  
 **final** Piece pieceAtDestination = candidateDestinationTile.getPiece();  
 **final** Alliance pieceAlliance = pieceAtDestination.getPieceAlliance();  
 **if** (**this**.getPieceAlliance() != pieceAlliance) { *// new Move if on the destination Tile there is an enemy* legalMoves.add(**new** MajorAttackMove(board, **this**,  
 candidateDestinationCoordinate, pieceAtDestination));  
 }  
 **break**;  
 }  
 }  
 }  
 }  
 **return** ImmutableList.*copyOf*(legalMoves);  
 }  
  
 **@Override  
 public** Queen movePiece(**final** Move move) {  
 **return new** Queen(move.getDestinationCoordinate(), move.getMovedPiece().getPieceAlliance());  
 }  
  
 **@Override  
 public** String toString() {  
 **return** PieceType.***QUEEN***.toString();  
 }  
}

**Rook:**

**package** com.engine.pieces;  
  
**import** com.engine.Alliance;  
**import** com.engine.board.Board;  
**import** com.engine.board.BoardUtils;  
**import** com.engine.board.Move;  
**import** com.engine.board.Tile;  
**import** com.google.common.collect.ImmutableList;  
  
**import** java.util.ArrayList;  
**import** java.util.Collection;  
**import** java.util.List;  
  
**import static** com.engine.board.Move.\*;  
  
**public class** Rook **extends** Piece {  
 **private final static int**[] ***CANDIDATE\_MOVE\_VECTOR\_COORDINATE*** = {-8, -1, 1, 8};  
  
 **public** Rook(**final int** piecePosition, **final** Alliance pieceAlliance) {  
 **super**(piecePosition, pieceAlliance, PieceType.***ROOK***, **true**);  
 }  
  
 **public** Rook(**final int** piecePosition, **final** Alliance pieceAlliance, **final boolean** isFirstMove) {  
 **super**(piecePosition, pieceAlliance, PieceType.***ROOK***,isFirstMove);  
 }  
  
 **private static boolean** isFirstColumnExclusion(**final int** currentPosition, **final int** candidateOffset) {  
 **return** BoardUtils.***FIRST\_FILE***[currentPosition] && (candidateOffset == -1);  
 }  
  
 **private static boolean** isEighthColumnExclusion(**final int** currentPosition, **final int** candidateOffset) {  
 **return** BoardUtils.***EIGHTH\_FILE***[currentPosition] && (candidateOffset == 1);  
 }  
  
 */\*\*  
 \* Calculates all the legal(available) moves of the Rook and return it as a list  
 \*  
 \* Iterates over all the offsets compared to the current position and compared to the knight it add the offsets again and  
 \* again until the candidateDestinationCoordinate is not on the Board and checks if the candidateDestinationCoordinate  
 \* is on the board and the exceptions are checks against the offset rules(because they do not always apply)  
 \* and breaks the loop if they don't apply.  
 \* after that we check if the candidateDestinationCoordinate has a Piece, if not we add a legal MajorMove and else it  
 \* checks whether the Piece is an enemy, if it is we add a AttackMove and because it has a Piece we don't want to  
 \* keep adding offsets because the pieceAtDestination is blocking the rest of the available Moves of the Rook.  
 \*  
 \** ***@param board*** *is needed for access to the Tiles and Pieces on the board  
 \** ***@return*** *the list of legalMoves that cannot be change hench it is "final" and return as "Immutable.copyOf(legalMoves)"  
 \*/* **@Override  
 public** Collection<Move> calculateLegalMoves(**final** Board board) {  
 **final** List<Move> legalMoves = **new** ArrayList<>();  
 **for** (**final int** currentCandidateOffset : CANDIDATE\_MOVE\_VECTOR\_COORDINATE) {  
 **int** candidateDestinationCoordinate = **this**.getPiecePosition();  
 **while** (BoardUtils.isValidTileCoordinate(candidateDestinationCoordinate)) {  
 **if** (isFirstColumnExclusion(candidateDestinationCoordinate, currentCandidateOffset) ||  
 *isEighthColumnExclusion*(candidateDestinationCoordinate, currentCandidateOffset)) {  
 **break**;  
 }  
 candidateDestinationCoordinate += currentCandidateOffset;  
 **if** (BoardUtils.*isValidTileCoordinate*(candidateDestinationCoordinate)) {  
 **final** Tile candidateDestinationTile = board.getTile(candidateDestinationCoordinate);  
 **if** (!candidateDestinationTile.isTileOccupied()) { *// new Move if Tile is empty* legalMoves.add(**new** MajorMove(board, **this**,  
 candidateDestinationCoordinate));  
 } **else** {  
 **final** Piece pieceAtDestination = candidateDestinationTile.getPiece();  
 **final** Alliance pieceAlliance = pieceAtDestination.getPieceAlliance();  
 **if** (**this**.getPieceAlliance() != pieceAlliance) { *// new Move if on the destination Tile there is an enemy* legalMoves.add(**new** MajorAttackMove(board, **this**,  
 candidateDestinationCoordinate, pieceAtDestination));  
 }  
 **break**;  
 }  
 }  
 }  
 }  
 **return** ImmutableList.*copyOf*(legalMoves);  
 }  
  
 **@Override  
 public** Rook movePiece(Move move) {  
 **return new** Rook(move.getDestinationCoordinate(), move.getMovedPiece().getPieceAlliance());  
 }  
  
 **@Override  
 public** String toString() {  
 **return** PieceType.***ROOK***.toString();  
 }  
}

**Player:**

**package** com.engine.player;  
  
**import** com.engine.Alliance;  
**import** com.engine.board.Board;  
**import** com.engine.board.Move;  
**import** com.engine.pieces.King;  
**import** com.engine.pieces.Piece;  
**import** com.google.common.collect.ImmutableList;  
**import** com.google.common.collect.Iterables;  
  
**import** java.util.ArrayList;  
**import** java.util.Collection;  
**import** java.util.List;  
  
**public abstract class** Player {  
 **protected final** Board **board**;  
 **final** King **playerKing**;  
 **private final** Collection<Move> **legalMoves**;  
 **private final boolean isInCheck**;  
  
 Player(**final** Board board,  
 **final** Collection<Move> legalMoves,  
 **final** Collection<Move> opponentMoves) {  
 **this**.**board** = board;  
 **this**.**playerKing** = establishKing();  
 **this**.**legalMoves** = ImmutableList.*copyOf*(Iterables.*concat*(legalMoves, calculateKingCastles(legalMoves, opponentMoves)));  
 **this**.**isInCheck** = !*calculateAttackOnTile*(**this**.**playerKing**.getPiecePosition(), opponentMoves).isEmpty();  
 }  
  
 **static** Collection<Move> calculateAttackOnTile(**final int** piecePosition, **final** Collection<Move> moves) {  
 **final** List<Move> attackTileMoves = **new** ArrayList<>();  
 **for**(**final** Move candidateTileMove : moves) {  
 **if**(piecePosition == candidateTileMove.getDestinationCoordinate()) {  
 attackTileMoves.add(candidateTileMove);  
 }  
 }  
 **return** ImmutableList.*copyOf*(attackTileMoves);  
 }  
  
 **private** King getPlayerKing() {  
 **return playerKing**;  
 }  
  
 **public** Collection<Move> getLegalMoves() {  
 **return this**.**legalMoves**;  
 }  
  
 **private** King establishKing() {  
 **for**(**final** Piece piece : getActivePieces()) {  
 **if**(piece.getPieceType().isKing()) {  
 **return** (King) piece;  
 }  
 }  
 **throw new** RuntimeException(**"Should not reach here! Not a valid board"**);  
 }  
  
 **private boolean** isMoveLegal(**final** Move candidateMove) {  
 **return this**.**legalMoves**.contains(candidateMove);  
 }  
  
 **public boolean** isInCheck() {  
 **return this**.**isInCheck**;  
 }  
  
 **public boolean** isInCheckMate() {  
 **return this**.**isInCheck** && hasNoEscapeMoves();  
 }  
  
 **public boolean** isInStaleMate() {  
 **return** !**this**.**isInCheck** && hasNoEscapeMoves();  
 }  
  
 **private boolean** hasNoEscapeMoves() {  
 **for**(**final** Move move : **this**.**legalMoves**) {  
 **final** MoveTransition transition = makeMove(move);  
 **if**(transition.getMoveStatus().isDone()) {  
 **return false**;  
 }  
 }  
 **return true**;  
 }  
  
 */\*\*  
 \* the function checks if currentPlayer can make the candidateMove so it will not lead to an illegal move or to checking himself  
 \*  
 \* the function checks if the move is not legal so it creates a moveTransition with status that is ILLEGAL\_MOVE else  
 \* if the move is legal it checks if the move will lead to an attack on currentPlayer king(checking himself)  
 \*  
 \** ***@param candidateMove*** *is for checking if currentPlayer can make candidateMove  
 \** ***@return*** *a MoveTransition with the candidateMove and the status of it  
 \*/* **public** MoveTransition makeMove(**final** Move candidateMove) {  
 **if**(!isMoveLegal(candidateMove)) {  
 **return new** MoveTransition(**this**.**board**, candidateMove, MoveStatus.***ILLEGAL\_MOVE***);  
 }  
 **final** Board transitionBoard = candidateMove.execute();  
 **final** Collection<Move> kingAttacks = *calculateAttackOnTile*(transitionBoard.getCurrentPlayer().getOpponent().getPlayerKing().getPiecePosition(),  
 transitionBoard.getCurrentPlayer().getLegalMoves());  
 **if**(!kingAttacks.isEmpty()) {  
 **return new** MoveTransition(**this**.**board**, candidateMove, MoveStatus.***LEAVES\_PLAYER\_IN\_CHECK***);  
 }  
 **return new** MoveTransition(transitionBoard, candidateMove, MoveStatus.***DONE***);  
 }  
  
 **public abstract** Collection<Piece> getActivePieces();  
  
 **public abstract** Alliance getAlliance();  
  
 **public abstract** Player getOpponent();  
  
 **protected abstract** Collection<Move> calculateKingCastles(**final** Collection<Move> playerLegals,  
 **final** Collection<Move> opponentLegals);  
}

**WhitePlayer:**

**package** com.engine.player;  
  
**import** com.engine.Alliance;  
**import** com.engine.board.Board;  
**import** com.engine.board.Move;  
**import** com.engine.board.Tile;  
**import** com.engine.pieces.Piece;  
**import** com.engine.pieces.Rook;  
**import** com.google.common.collect.ImmutableList;  
  
**import** java.util.ArrayList;  
**import** java.util.Collection;  
**import** java.util.List;  
  
**import static** com.engine.board.Move.\*;  
  
**public class** WhitePlayer **extends** Player {  
  
 **public** WhitePlayer(**final** Board board, **final** Collection<Move> whiteStandardLegalMoves, **final** Collection<Move> blackStandardLegalMoves) {  
 **super**(board, whiteStandardLegalMoves, blackStandardLegalMoves);  
 }  
  
 **@Override  
 public** Collection<Piece> getActivePieces() {  
 **return this**.**board**.getWhitePieces();  
 }  
  
 **@Override  
 public** Alliance getAlliance() {  
 **return** Alliance.***WHITE***;  
 }  
  
 **@Override  
 public** Player getOpponent() {  
 **return this**.**board**.getBlackPlayer();  
 }  
  
 **@Override  
 protected** Collection<Move> calculateKingCastles(**final** Collection<Move> playerLegals, **final** Collection<Move> opponentLegals) {  
 **final** List<Move> kingCastles = **new** ArrayList<>();  
 **if**(**this**.**playerKing**.isFirstMove() && !**this**.isInCheck()) {  
 *//white king side castle* **if**(!**this**.**board**.getTile(61).isTileOccupied() && !**this**.**board**.getTile(62).isTileOccupied()) {  
 **final** Tile rookTile = **this**.**board**.getTile(63);  
 **if**(rookTile.isTileOccupied() && rookTile.getPiece().getPieceType().isRook() && rookTile.getPiece().isFirstMove()) {  
 **if**(Player.*calculateAttackOnTile*(61, opponentLegals).isEmpty() &&  
 Player.*calculateAttackOnTile*(62, opponentLegals).isEmpty()) {  
 kingCastles.add(**new** KingSideCastleMove(**this**.**board**, **this**.**playerKing**,  
 62, (Rook) rookTile.getPiece(),61));  
 }  
 }  
 }  
 *//white queen side castle* **if**(!**this**.**board**.getTile(59).isTileOccupied() && !**this**.**board**.getTile(58).isTileOccupied() &&  
 !**this**.**board**.getTile(57).isTileOccupied()) {  
 **final** Tile rookTile = **this**.**board**.getTile(56);  
 **if**(rookTile.isTileOccupied() && rookTile.getPiece().getPieceType().isRook() && rookTile.getPiece().isFirstMove()) {  
 **if**(Player.*calculateAttackOnTile*(59, opponentLegals).isEmpty() &&  
 Player.*calculateAttackOnTile*(58, opponentLegals).isEmpty())  
 kingCastles.add(**new** QueenSideCastleMove(**this**.**board**, **this**.**playerKing**,  
 58, (Rook) rookTile.getPiece(), 59));  
 }  
 }  
 }  
 **return** ImmutableList.*copyOf*(kingCastles);  
 }  
}

**BlackPlayer:**

**package** com.engine.player;  
  
**import** com.engine.Alliance;  
**import** com.engine.board.Board;  
**import** com.engine.board.Move;  
**import** com.engine.board.Tile;  
**import** com.engine.pieces.Piece;  
**import** com.engine.pieces.Rook;  
**import** com.google.common.collect.ImmutableList;  
  
**import** java.util.ArrayList;  
**import** java.util.Collection;  
**import** java.util.List;  
  
**import static** com.engine.board.Move.\*;  
  
**public class** BlackPlayer **extends** Player {  
 **public** BlackPlayer(**final** Board board, **final** Collection<Move> whiteStandardLegalMoves, **final** Collection<Move> blackStandardLegalMoves) {  
 **super**(board, blackStandardLegalMoves, whiteStandardLegalMoves);  
 }  
  
 **@Override  
 public** Collection<Piece> getActivePieces() {  
 **return this**.**board**.getBlackPieces();  
 }  
  
 **@Override  
 public** Alliance getAlliance() {  
 **return** Alliance.***BLACK***;  
 }  
  
 **@Override  
 public** Player getOpponent() {  
 **return this**.**board**.getWhitePlayer();  
 }  
  
 **@Override  
 protected** Collection<Move> calculateKingCastles(**final** Collection<Move> playerLegals, **final** Collection<Move> opponentLegals) {  
 **final** List<Move> kingCastles = **new** ArrayList<>();  
 **if**(**this**.**playerKing**.isFirstMove() && !**this**.isInCheck()) {  
 *//black king side castle* **if**(!**this**.**board**.getTile(5).isTileOccupied() && !**this**.**board**.getTile(6).isTileOccupied()) {  
 **final** Tile rookTile = **this**.**board**.getTile(7);  
 **if**(rookTile.isTileOccupied() && rookTile.getPiece().getPieceType().isRook() && rookTile.getPiece().isFirstMove()) {  
 **if**(Player.*calculateAttackOnTile*(5, opponentLegals).isEmpty() &&  
 Player.*calculateAttackOnTile*(6, opponentLegals).isEmpty()) {  
 kingCastles.add(**new** KingSideCastleMove(**this**.**board**, **this**.**playerKing**,  
 6, (Rook) rookTile.getPiece(),5));  
 }  
 }  
 }  
 *//black queen side castle* **if**(!**this**.**board**.getTile(3).isTileOccupied() && !**this**.**board**.getTile(2).isTileOccupied() &&  
 !**this**.**board**.getTile(1).isTileOccupied()) {  
 **final** Tile rookTile = **this**.**board**.getTile(0);  
 **if**(rookTile.isTileOccupied() && rookTile.getPiece().getPieceType().isRook() && rookTile.getPiece().isFirstMove()) {  
 **if**(Player.*calculateAttackOnTile*(3, opponentLegals).isEmpty() &&  
 Player.*calculateAttackOnTile*(2, opponentLegals).isEmpty())  
 kingCastles.add(**new** QueenSideCastleMove(**this**.**board**, **this**.**playerKing**,  
 2, (Rook) rookTile.getPiece(), 3));  
 }  
 }  
 }  
 **return** ImmutableList.*copyOf*(kingCastles);  
 }  
}

**MoveStatus:**

**package** com.engine.player;  
  
**public enum** MoveStatus {  
 ***DONE***{  
 **@Override  
 public boolean** isDone() {  
 **return true**;  
 }  
 },  
 ***ILLEGAL\_MOVE*** {  
 **@Override  
 public boolean** isDone() {  
 **return false**;  
 }  
 },  
 ***LEAVES\_PLAYER\_IN\_CHECK***{  
 **@Override  
 public boolean** isDone() {  
 **return false**;  
 }  
 };  
  
 **public abstract boolean** isDone();  
}

**MoveTransition:**

**package** com.engine.player;  
  
**import** com.engine.board.Board;  
**import** com.engine.board.Move;  
  
**public class** MoveTransition {  
 **private final** Board **transitionBoard**;  
 **private final** Move **move**;  
 **private final** MoveStatus **moveStatus**;  
  
 **public** MoveTransition(**final** Board transitionBoard, **final** Move move, **final** MoveStatus moveStatus) {  
 **this**.**transitionBoard** = transitionBoard;  
 **this**.**move** = move;  
 **this**.**moveStatus** = moveStatus;  
 }  
  
 **public** Board getTransitionBoard() {  
 **return this**.**transitionBoard**;  
 }  
  
 **public** MoveStatus getMoveStatus() {  
 **return this**.**moveStatus**;  
 }  
}

**Alliance:**

**package** com.engine;  
  
**import** com.engine.board.BoardUtils;  
**import** com.engine.player.BlackPlayer;  
**import** com.engine.player.Player;  
**import** com.engine.player.WhitePlayer;  
  
**public enum** Alliance {  
 ***WHITE***(**"w"**) {  
 **@Override  
 public int** getDirection() {  
 **return** -1;  
 }  
  
 **@Override  
 public int** getOppositeDirection() {  
 **return** 1;  
 }  
  
 **@Override  
 public boolean** isBlack() {  
 **return false**;  
 }  
  
 **@Override  
 public boolean** isWhite() {  
 **return true**;  
 }  
  
 **@Override  
 public boolean** isPawnPromotionSquare(**int** position) {  
 **return** BoardUtils.***EIGHTH\_RANK***[position];  
 }  
  
 **@Override  
 public** Player choosePlayer(**final** WhitePlayer whitePlayer, **final** BlackPlayer blackPlayer) {  
 **return** whitePlayer;  
 }  
 },  
 ***BLACK***(**"b"**) {  
 **@Override  
 public int** getDirection() {  
 **return** 1;  
 }  
  
 **@Override  
 public int** getOppositeDirection() {  
 **return** -1;  
 }  
  
 **@Override  
 public boolean** isBlack() {  
 **return true**;  
 }  
  
 **@Override  
 public boolean** isWhite() {  
 **return false**;  
 }  
  
 **@Override  
 public boolean** isPawnPromotionSquare(**int** position) {  
 **return** BoardUtils.***FIRST\_RANK***[position];  
 }  
  
 **@Override  
 public** Player choosePlayer(**final** WhitePlayer whitePlayer, **final** BlackPlayer blackPlayer) {  
 **return** blackPlayer;  
 }  
 };  
  
 **private final** String **color**;  
  
 Alliance(**final** String color) {  
 **this**.**color** = color;  
 }  
  
 **@Override  
 public** String toString(){  
 **return this**.**color**;  
 }  
  
 **public abstract int** getDirection();  
 **public abstract int** getOppositeDirection();  
 **public abstract boolean** isBlack();  
 **public abstract boolean** isWhite();  
 **public abstract boolean** isPawnPromotionSquare(**final int** position);  
 **public abstract** Player choosePlayer(**final** WhitePlayer whitePlayer, **final** BlackPlayer blackPlayer);  
}

**LogHistoryPanel:**

**package** com.gui;  
  
**import** com.engine.board.Board;  
**import** com.engine.board.Move;  
**import** javafx.scene.control.TableColumn;  
**import** javafx.scene.control.TableView;  
**import** javafx.scene.control.cell.PropertyValueFactory;  
**import** javafx.scene.layout.Pane;  
  
**public class** LogHistoryPanel **extends** Pane {  
 **private final** TableView **table**;  
 **private** String **whiteMove** = **null**;  
 **private int currentRow** = 0;  
 **public** LogHistoryPanel() {  
  
 **table** = **new** TableView();  
 **table**.setPrefSize(175, 550);  
 **table**.setEditable(**false**);  
  
 TableColumn<String, Row> whiteCol = **new** TableColumn<>(**"White"**);  
 whiteCol.setCellValueFactory(**new** PropertyValueFactory<>(**"whiteMove"**));  
 whiteCol.setResizable(**false**);  
 TableColumn<String, Row> blackCol = **new** TableColumn<>(**"Black"**);  
 blackCol.setCellValueFactory(**new** PropertyValueFactory<>(**"blackMove"**));  
 blackCol.setResizable(**false**);  
 **table**.getColumns().addAll(whiteCol, blackCol);  
 **this**.getChildren().add(**table**);  
  
 }  
  
 **void** add(**final** Board board, **final** Move move) {  
 **final** String moveText = move.toString() + calculateCheckAndCheckMateHash(board);  
 **if**(move.getMovedPiece().getPieceAlliance().isWhite()) {  
 **whiteMove** = moveText;  
 **table**.getItems().add(**new** Row(moveText, **""**));  
 } **else if**(move.getMovedPiece().getPieceAlliance().isBlack())  
 {  
 **table**.getItems().set(**currentRow**, **new** Row(**whiteMove**, moveText));  
 **currentRow**++;  
 }  
 }  
  
 **private** String calculateCheckAndCheckMateHash(**final** Board board) {  
 **if**(board.getCurrentPlayer().isInCheckMate()) {  
 **return "#"**;  
 } **else if**(board.getCurrentPlayer().isInCheck()) {  
 **return "+"**;  
 }  
 **return ""**;  
 }  
  
 **public static class** Row {  
 **private** String **whiteMove**;  
 **private** String **blackMove**;  
  
 Row(**final** String whiteMove, **final** String blackMove) {  
 **this**.**whiteMove** = whiteMove;  
 **this**.**blackMove** = blackMove;  
 }  
  
 **public** String getWhiteMove() {  
 **return this**.**whiteMove**;  
 }  
  
 **public void** setWhiteMove(**final** String whiteMove) {  
 **this**.**whiteMove** = whiteMove;  
 }  
  
 **public** String getBlackMove() {  
 **return this**.**blackMove**;  
 }  
  
 **public void** setBlackMove(**final** String blackMove) {  
 **this**.**blackMove** = blackMove;  
 }  
 }  
}

**TakenPiecesPanel:**

**package** com.gui;  
  
**import** com.engine.board.Move;  
**import** com.engine.pieces.Piece;  
**import** com.google.common.primitives.Ints;  
**import** javafx.scene.image.Image;  
**import** javafx.scene.image.ImageView;  
**import** javafx.scene.layout.TilePane;  
**import** javafx.scene.layout.VBox;  
  
**import** java.io.FileInputStream;  
**import** java.io.FileNotFoundException;  
**import** java.util.ArrayList;  
**import** java.util.List;  
  
**import static** com.gui.Controller.MoveLog;  
**import static** com.gui.Controller.***RESOURCES\_PATH***;  
  
**class** TakenPiecesPanel **extends** VBox {  
 **private static final** String ***BACKGROUND\_COLOR\_PANEL*** = **"-fx-background-color : rgb(255, 222, 173)"**;  
 **private final** TilePane **upperPanel**;  
 **private final** TilePane **lowerPanel**;  
  
 TakenPiecesPanel() {  
 **this**.setPrefSize(80, 550);  
 **this**.setStyle(***BACKGROUND\_COLOR\_PANEL***);  
 **upperPanel** = **new** TilePane();  
 **lowerPanel** = **new** TilePane();  
 **upperPanel**.setPrefSize(80, 275);  
 **lowerPanel**.setPrefSize(80, 275);  
 *//****TODO check about the properties Hgap and Vgap and prefColumns of TilePane* upperPanel**.setHgap(2.5);  
 **upperPanel**.setVgap(2.5);  
 **lowerPanel**.setHgap(2.5);  
 **lowerPanel**.setVgap(2.5);  
 **lowerPanel**.setStyle(***BACKGROUND\_COLOR\_PANEL***);  
 **lowerPanel**.setStyle(***BACKGROUND\_COLOR\_PANEL***);  
 **this**.getChildren().addAll(**upperPanel**, **lowerPanel**);  
 }  
  
 **void** addTakenPiece(**final** MoveLog moveLog) {  
 **this**.**upperPanel**.getChildren().clear();  
 **this**.**lowerPanel**.getChildren().clear();  
 **final** List<Piece> whiteTakenPieces = **new** ArrayList<>();  
 **final** List<Piece> blackTakenPieces = **new** ArrayList<>();  
 **for**(**final** Move move : moveLog.getMoves()) {  
 **if**(move.isAttack()) {  
 **final** Piece takenPiece = move.getAttackedPiece();  
 **if**(takenPiece.getPieceAlliance().isWhite()) {  
 whiteTakenPieces.add(takenPiece);  
 } **else if**(takenPiece.getPieceAlliance().isBlack()) {  
 blackTakenPieces.add(takenPiece);  
 } **else** {  
 **throw new** RuntimeException(**"Should not reach here"**);  
 }  
 }  
 }  
 *//****TODO fix the add functions*** whiteTakenPieces.sort((p1, p2) -> Ints.*compare*(p1.getPieceValue(), p2.getPieceValue()));  
 blackTakenPieces.sort((p1, p2) -> Ints.*compare*(p1.getPieceValue(), p2.getPieceValue()));  
 **for**(**final** Piece takenPiece : whiteTakenPieces) {  
 **try** {  
 ImageView logPieceIcon = **new** ImageView(**new** Image(**new** FileInputStream(  
 ***RESOURCES\_PATH*** +  
 takenPiece.getPieceAlliance().toString() +  
 takenPiece.toString() + **".png"**)));  
 logPieceIcon.setFitWidth(35);  
 logPieceIcon.setFitHeight(29.375);  
 **this**.**upperPanel**.getChildren().add(logPieceIcon);  
 } **catch** (FileNotFoundException e) {  
 e.printStackTrace();  
 }  
 }  
 **for**(**final** Piece takenPiece : blackTakenPieces) {  
 **try** {  
 ImageView logPieceIcon = **new** ImageView(**new** Image(**new** FileInputStream(  
 ***RESOURCES\_PATH*** +  
 takenPiece.getPieceAlliance().toString() +  
 takenPiece.toString() + **".png"**)));  
 logPieceIcon.setFitWidth(35);  
 logPieceIcon.setFitHeight(29.375);  
 **this**.**lowerPanel**.getChildren().add(logPieceIcon);  
 } **catch** (FileNotFoundException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
}

**Controller:**

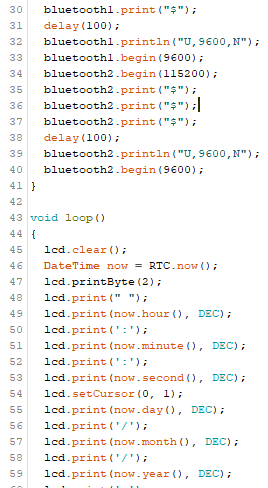
**package** com.gui;  
  
**import** com.engine.board.\*;  
  
**import** com.engine.pieces.Piece;  
**import** com.engine.player.MoveTransition;  
**import** com.google.common.collect.Lists;  
**import** javafx.application.Platform;  
**import** javafx.event.ActionEvent;  
**import** javafx.event.EventHandler;  
**import** javafx.fxml.**FXML**;  
**import** javafx.fxml.Initializable;  
**import** javafx.scene.control.\*;  
**import** javafx.scene.image.Image;  
**import** javafx.scene.image.ImageView;  
**import** javafx.scene.input.MouseButton;  
**import** javafx.scene.input.MouseEvent;  
**import** javafx.scene.layout.BorderPane;  
**import** javafx.scene.layout.GridPane;  
**import** javafx.scene.layout.Pane;  
  
**import** java.io.FileInputStream;  
**import** java.io.FileNotFoundException;  
**import** java.net.URL;  
**import** java.util.\*;  
  
*/\*\*  
 \* Controller class that Controls the Board of the game (GUI Implantation)  
 \*/***public class** Controller **implements** Initializable {  
 **static final** String ***RESOURCES\_PATH*** = **"Resources\\"**;  
 **private static final int *BOARD\_PANEL\_WIDTH*** = 650;  
 **private static final int *BOARD\_PANEL\_HEIGHT*** = 550;  
 **private static final double *TILE\_PANEL\_WIDTH*** = 81.25;  
 **private static final double *TILE\_PANEL\_HEIGHT*** = 68.75;  
 **@FXML  
 private** BorderPane **borderPane**;  
  
 **private** LogHistoryPanel **logHistoryPanel**;  
 **private** TakenPiecesPanel **takenPiecesPanel**;  
 **private** BoardPanel **chessBoard**;  
 **private** MoveLog **moveLog**;  
  
 **private** Board **gameBoard**;  
 **private** Tile **sourceTile**;  
 **private** Tile **targetTile**;  
 **private** Piece **movedPiece**;  
 **private** BoardDirection **boardDirection**;  
 **private boolean highlightLegalMoves**;  
  
 */\*\*  
 \* Initialize method that will create all the objects on the board before starting the game  
 \*  
 \* The method iterates on all the GridPane nodes and creating new Panes  
 \* with different colors(BLACK, WHITE - DARK BROWN, WHITE BROWN)  
 \* On each Pane we create a ImageView control that has an Image of EmptyPiece(no Image, transparent) or a piece image  
 \** ***@param url*** *add text here  
 \** ***@param resourceBundle*** *add text here  
 \*/* **@Override  
 public void** initialize(URL url, ResourceBundle resourceBundle) {  
 **gameBoard** = Board.*createStandardBoard*();  
  
 **this**.**moveLog** = **new** MoveLog();  
  
 **boardDirection** = BoardDirection.***NORMAL***;  
 **highlightLegalMoves** = **true**;  
  
 MenuBar menuBar = createTableMenuBar();  
 **borderPane**.setTop(menuBar);  
  
 **logHistoryPanel** = **new** LogHistoryPanel();  
 **borderPane**.setRight(**logHistoryPanel**);  
  
 **takenPiecesPanel** = **new** TakenPiecesPanel();  
 **borderPane**.setLeft(**takenPiecesPanel**);  
  
 **chessBoard** = **new** BoardPanel();  
 **borderPane**.setCenter(**chessBoard**);  
  
 }  
  
 **private** MenuBar createTableMenuBar() {  
 **final** MenuBar tableMenuBar = **new** MenuBar();  
 tableMenuBar.getMenus().add(createFileMenu());  
 tableMenuBar.getMenus().add(createPreferencesMenu());  
 **return** tableMenuBar;  
 }  
  
 **private** Menu createFileMenu() {  
 **final** Menu fileMenu = **new** Menu(**"File"**);  
 MenuItem openPGNMenuItem = createMenuItem(**"Load PGN file"**,  
 e -> System.***out***.println(**"open up that PGN file!"**));  
 fileMenu.getItems().add(openPGNMenuItem);  
 **final** MenuItem exitMenuItem = createMenuItem(**"Exit"**,  
 e -> System.*exit*(0));  
 fileMenu.getItems().add(exitMenuItem);  
 **return** fileMenu;  
 }  
  
 **private** Menu createPreferencesMenu() {  
 **final** Menu preferencesMenu = **new** Menu(**"Preferences"**);  
 MenuItem flipBoardMenuItem = createMenuItem(**"Flip Board"**, e -> {  
 **boardDirection** = **boardDirection**.opposite();  
 **chessBoard**.drawBoard(**gameBoard**);  
 });  
 preferencesMenu.getItems().add(flipBoardMenuItem);  
 preferencesMenu.getItems().add(**new** SeparatorMenuItem());  
 CheckMenuItem legalMoveHighlighterCheckbox = **new** CheckMenuItem(**"Highlight legal Moves"**);  
 legalMoveHighlighterCheckbox.setSelected(**false**);  
 legalMoveHighlighterCheckbox.setOnAction(e -> **highlightLegalMoves** = legalMoveHighlighterCheckbox.isSelected());  
 preferencesMenu.getItems().add(legalMoveHighlighterCheckbox);  
 **return** preferencesMenu;  
 }  
  
 **private** MenuItem createMenuItem(**final** String ItemTitle, **final** EventHandler<ActionEvent> eventHandler) {  
 **final** MenuItem openPGNMenuItem = **new** MenuItem(ItemTitle);  
 openPGNMenuItem.setOnAction(eventHandler);  
 **return** openPGNMenuItem;  
 }  
  
 **public enum** BoardDirection {  
 ***NORMAL*** {  
 **@Override** List<TilePanel> traverse(**final** List<TilePanel> boardTiles) {  
 **return** boardTiles;  
 }  
  
 **@Override** BoardDirection opposite() {  
 **return *FLIPPED***;  
 }  
 },  
 ***FLIPPED*** {  
 **@Override** List<TilePanel> traverse(List<TilePanel> boardTiles) {  
 **return** Lists.*reverse*(boardTiles);  
 }  
  
 **@Override** BoardDirection opposite() {  
 **return *NORMAL***;  
 }  
 };  
  
 **abstract** List<TilePanel> traverse(**final** List<TilePanel> boardTiles);  
  
 **abstract** BoardDirection opposite();  
 }  
  
 **public static class** MoveLog {  
 **private final** List<Move> **moves**;  
  
 MoveLog() {  
 **this**.**moves** = **new** ArrayList<>();  
 }  
  
 **public** List<Move> getMoves() {  
 **return this**.**moves**;  
 }  
  
 **void** addMove(**final** Move move) {  
 **this**.**moves**.add(move);  
 }  
  
 **int** size() {  
 **return this**.**moves**.size();  
 }  
  
 **void** clear() {  
 **this**.**moves**.clear();  
 }  
  
 **public** Move removeMove(**final int** index) {  
 **return this**.**moves**.remove(index);  
 }  
  
 **public boolean** removeMove(**final** Move move) {  
 **return this**.**moves**.remove(move);  
 }  
 }  
  
 **private class** BoardPanel **extends** GridPane {  
 **final** List<TilePanel> **boardTiles**;  
  
 BoardPanel() {  
 **this**.setPrefSize(***BOARD\_PANEL\_WIDTH***, ***BOARD\_PANEL\_HEIGHT***);  
 **this**.**boardTiles** = **new** ArrayList<>(BoardUtils.***NUM\_TILES***);  
 **for**(**int** i = 0; i < BoardUtils.***NUM\_TILES\_PER\_ROW***; i++) {  
 **for**(**int** j = 0, tileId; j < BoardUtils.***NUM\_TILES\_PER\_ROW***; j++) {  
 tileId = i \* BoardUtils.***NUM\_TILES\_PER\_ROW*** + j;  
 **final** TilePanel tilePanel = **new** TilePanel(**this**, tileId);  
 **this**.**boardTiles**.add(tilePanel);  
 **this**.add(tilePanel, j, i);  
 }  
 }  
 }  
  
 **void** drawBoard(**final** Board board) {  
 **this**.getChildren().clear();  
 **for**(**int** i = 0; i < BoardUtils.***NUM\_TILES\_PER\_ROW***; i++) {  
 **for**(**int** j = 0, tileId; j < BoardUtils.***NUM\_TILES\_PER\_ROW***; j++) {  
 tileId = i \* BoardUtils.***NUM\_TILES\_PER\_ROW*** + j;  
 **boardDirection**.traverse(**boardTiles**).get(tileId).drawTile(board);  
 **this**.add(**boardDirection**.traverse(**boardTiles**).get(tileId), j, i);  
 }  
 }  
 }  
 }  
  
 **private class** TilePanel **extends** Pane {  
 **private final int tileId**;  
  
 **private** TilePanel(**final** BoardPanel boardPanel, **final int** tileId) {  
 **this**.setPrefSize(***TILE\_PANEL\_WIDTH***, ***TILE\_PANEL\_HEIGHT***);  
 **this**.**tileId** = tileId;  
 assignTileColor();  
 assignTilePieceIcon(**gameBoard**);  
 EventHandler<MouseEvent> eventHandler = mouseEvent -> {  
 **if**(mouseEvent.getButton() == MouseButton.***SECONDARY***) {  
 **sourceTile** = **null**;  
 **targetTile** = **null**;  
 **movedPiece** = **null**;  
 } **else if**(mouseEvent.getButton() == MouseButton.***PRIMARY***) {  
 **if**(**sourceTile** == **null**) {  
 *//first click* **sourceTile** = **gameBoard**.getTile(tileId);  
 **movedPiece** = **sourceTile**.getPiece();  
 **if**(**movedPiece** == **null**) {  
 **sourceTile** = **null**;  
 }  
 } **else** {  
 *// second click* **targetTile** = **gameBoard**.getTile(tileId);  
 **final** Move move = Move.MoveFactory.*createMove*(**gameBoard**, **sourceTile**.getTileCoordinate(),  
 **targetTile**.getTileCoordinate());  
 **final** MoveTransition transition = **gameBoard**.getCurrentPlayer().makeMove(move);  
 **if**(transition.getMoveStatus().isDone()) {  
 **gameBoard** = transition.getTransitionBoard();  
 **moveLog**.addMove(move);  
 **logHistoryPanel**.add(**gameBoard**, move);  
 }  
 **sourceTile** = **null**;  
 **targetTile** = **null**;  
 **movedPiece** = **null**;  
 }  
 Platform.*runLater*(() -> {  
 **takenPiecesPanel**.addTakenPiece(**moveLog**);  
 boardPanel.drawBoard(**gameBoard**);  
 });  
 }  
 };  
 **this**.addEventFilter(MouseEvent.***MOUSE\_CLICKED***, eventHandler);  
 }  
  
 **void** drawTile(**final** Board board) {  
 assignTileColor();  
 assignTilePieceIcon(board);  
 highlightLegals(board);  
 }  
  
 **private void** assignTilePieceIcon(**final** Board gameBoard) {  
 **this**.getChildren().clear();  
 ImageView tilePieceIcon = **null**;  
 **if**(gameBoard.getTile(**this**.**tileId**).isTileOccupied()) {  
 **final** Piece tilePiece = gameBoard.getTile(**this**.**tileId**).getPiece();  
 **try** {  
 tilePieceIcon = **new** ImageView(**new** Image(**new** FileInputStream(  
 ***RESOURCES\_PATH*** +  
 tilePiece.getPieceAlliance().toString() +  
 tilePiece.toString() + **".png"**)));  
 } **catch** (FileNotFoundException e) {  
 e.printStackTrace();  
 }  
 } **else** {  
 tilePieceIcon = **new** ImageView();  
 }  
 **if** (tilePieceIcon != **null**) {  
 tilePieceIcon.setFitWidth(***TILE\_PANEL\_WIDTH***);  
 tilePieceIcon.setFitHeight(***TILE\_PANEL\_HEIGHT***);  
 **this**.getChildren().add(tilePieceIcon);  
 }  
 }  
  
 **private void** highlightLegals(**final** Board board) {  
 **if**(**highlightLegalMoves**) {  
 **for**(**final** Move move : pieceLegalMoves(board)) {  
 **if**(move.getDestinationCoordinate() == **this**.**tileId**) {  
 ImageView tilePieceIcon = **null**;  
 **try** {  
 tilePieceIcon = **new** ImageView(**new** Image(**new** FileInputStream(  
 ***RESOURCES\_PATH*** + **"HighlightedTile.png"**)));  
 } **catch** (FileNotFoundException e) {  
 e.printStackTrace();  
 }  
 **if** (tilePieceIcon != **null**) {  
 tilePieceIcon.setFitWidth(***TILE\_PANEL\_WIDTH***);  
 tilePieceIcon.setFitHeight(***TILE\_PANEL\_HEIGHT***);  
  
 **this**.getChildren().add(tilePieceIcon);  
 }  
 }  
 }  
 }  
 }  
  
 **private** Collection<Move> pieceLegalMoves(**final** Board board) {  
 **if**(**movedPiece** != **null** && **movedPiece**.getPieceAlliance() == board.getCurrentPlayer().getAlliance()) {  
 **return movedPiece**.calculateLegalMoves(board);  
 }  
 **return** Collections.*emptyList*();  
 }  
  
 **private void** assignTileColor() {  
 **if**((**this**.**tileId** + **this**.**tileId** / 8) % 2 == 0) {  
 **this**.setStyle(**"-fx-background-color : rgb(255, 222, 173)"**);  
 } **else** {  
 **this**.setStyle(**"-fx-background-color : rgb(244, 164, 96)"**);  
 }  
 }  
 }  
}

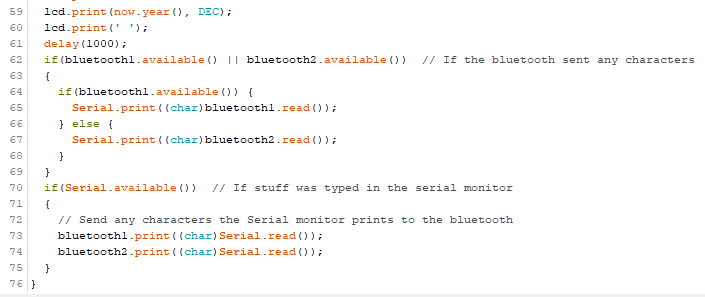
**Main:**

**package** com;  
  
**import** com.engine.board.Board;  
**import** javafx.application.Application;  
**import** javafx.fxml.FXMLLoader;  
**import** javafx.scene.Parent;  
**import** javafx.scene.Scene;  
**import** javafx.scene.image.Image;  
**import** javafx.stage.Stage;  
  
**public class** Main **extends** Application {  
  
 **@Override  
 public void** start(Stage primaryStage) **throws** Exception{  
 Parent root = FXMLLoader.*load*(getClass().getResource(**"gui/Chess.fxml"**));  
 primaryStage.getIcons().add(**new** Image(**"/wq.png"**));  
 primaryStage.setTitle(**"Chess"**);  
 primaryStage.setScene(**new** Scene(root));  
 primaryStage.setResizable(**false**);  
 primaryStage.show();  
 }  
  
 **public static void** main(String[] args) {  
 *launch*(args);  
 }  
}

**4.4 קובץ** List **של תוכנה בארדואינו:**







פרק 5 – סיכום ומסקנות

**5.1 מסקנות**

**לאחר כמה פרויקטים שביצעתי בתוכנה, ביצוע פרויקט זה היה מאתגר הרבה יותר מהפרויקטים הקודמים היחסית קטנים אליו. בנוסף, הפרויקט שילב חומרה שהייתי צריך ליישם מהידע שלמדתי באלקטרוניקה. הפרויקט היה מאתגר מכיוון שהייתי צריך לעבוד עליו בזמן הפנוי וגם ללמוד ולעשות עבודות בקורסים מסויימים באותו הזמן. יתר על כן, הפרויקט חשף אותי לנושאים רבים שלא ידעתי לפני(כמו למשל עבודה עם רכיבים שלא עבדתי איתם קודם שימוש בממשק גרפי בתוכנה ועוד).**

**בתחילת הפרויקט היה לי קושי לתכנן את הפרויקט כולו. אך, היה לי רעיון על איך להתחיל אותו ולעשות קטעים מסויימים ממנו. לכן, מהלך כל הפרויקט היה לעבוד על חלק מסויים להמשיך להבין איזה חלקים ותכונות נוספים הפרויקט צריך. בנוסף לכך, צריך לממש חלקים ותכונות אלו. לפי זאת, הייתי צריך להיאבק ולנסות למצוא פתרון באינטרנט או דרך המנחה. חלק מהבעיות לקחו לי שעות רבות ואפילו ימים שלמים לפתור.**

**אם היה לי את האפשרות להתחיל את הפרויקט מהתחלה הייתי מתכנן פחות את הפרויקט כדי לא לבזבז זמן ובניית הפרויקט הייתה הרבה יותר קלה. כלומר, יהיה לי יותר זמן להוסיף יותר תכונות והתנהגות למשחק.**

**יש אפשרות להרחיב ולהוסיף עוד לפרויקט על ידי שינוי התוכנה כך שהיא תתמוך בויריאציות שונות של משחקי שחמט. בנוסף, יש אפשרות להוסיף שני מחשבי PC כך שהמחשק יתמוך ויריאצית משחק בה 4 שחקנים משחקים שחמט ואף הוספת שיטת דירוג לשחקן כך שחקן מתחיל לא יצרך לשחק עם שחקן שהוא טוב מאוד.**

**אם היה לי עוד זמן לעבוד על הפרויקט הייתי מוסיף אפשרות לשחק גם נגד שחקן מחשב באמצעות בינה מלאכותית(שרציתי ללמוד לעשות) דרך שיטה שנקראת min max.**

**5.1 יומן עבודה**

**בתחילת הפרויקט, ניסיתי לתכנן את התוכנה של השחמט על ידי כך שרשמתי את רוב המחלקות הגדולות שאני צריך ואת הקשר ביניהם(בעניין של תכונות והתנהגות של כל מחלקה). בתכנון היו לי כמה מחלקות גדולות:**

1. **Tile המייצג את משבצת לוח השחמט שעל כל משבצת היה כלי שחמט מסויים או ריק.**
2. **Board המייצג את לוח השחמט עם 8x8 משבצות המסדורת ברשימה משבצות.**
3. **Move המייצג הזזה של כלי משחק על ידי יצרית לוח שחמט חדש עם מופע Board חדש.**
4. **Piece המייצג את כלי משחק הכללי.**
5. **Knight המייצג את כלי משחק הראשון שהיה לי שהוא הפרש.**
6. **Main המייצג המחלקה הראשית של התוכנה שכולל את פונקציית ההתחלה.**

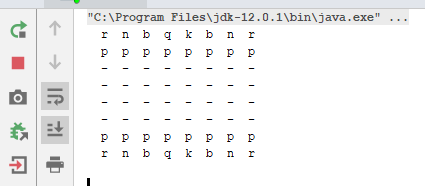
**לפי התכנון בהתחלה ייצגתי את הצגת הלוח וביצוע מהלכים לפי טקסט.**

**כעת כדי לממש את התכנון קודם כל התחלתי עם המחלקה Tile שיש לה את התכונות(משתנים) הבאות: מיקום נוכחי, צבע(שחור/לבן) וכלי משחק מסויים או ריק. ואת ההתנהגות(פונקציות) של קבלת מיקום הנוכחי, קבלת כלי שחמט שנמצא על המשבצת, ואת תצורת המשבצת ואת כלי המשחק שנמצא עליה כטקסט.**

**לאחר מכן, ממשתי את המחלקה Piece שיש לה את התכונות הבאות: מיקום נוכחי של כלי המשחק ואת צבע הכלי. ואת ההתנהגות של קבלת מיקום הכלי, קבלת צבע הכלי, הזזת כלי ותצורת כלי המשחק כטקסט. בנוסף, ממשתי את מחלקה Kinght שההתנהגות שלה היא תזוזה חוקית בלוח שחמט של פרש.**

**אחר כך, ממשתי את המחלקה Board יש לה את התכונות הבאות: רשימת משבצות ומפה עבור כל אינדקס(מיקום נוכחי) יתן את כלי המשחק. ואת ההתנהגות של מיקום כלי המשחק(בתחלית המשחק הם נמצאים במקומות המסויימים שלהם ותצורת לוח המשחק וכל המשבצות שעליו כולל כלי המשחק כטקסט.**

**ולבסוף ממשתי את Main שקוראת באופן מסודר לכל ההפונקציות כדי להציג משחק בטקסט ואת התזוזה שלו כטקסט.**



**במשך כמה שבועות הרחבתי את הפרויקט שגם יעבוד עבור שני שחקנים(במחשב אחד לצורך בדיקה) ובממשק גרפי. על ידי הרחבת התכנון ושימוש בתכנות נכון(שימוש בפונקציות כמו Equals ו HashCode כדי לבדוק בין שני אובייקטים וביצירת עד כמות מסויימת של אוביקטים.**

**וכמובן במקביל לעבודה על התוכנה בדקתי איך הרכיבים עובדים בפרויקט ותכנון על איך לשלב אותם עם התוכנה. למשל, רכיב הBluetooth שממומש גם דרך הארדואינו וגם דרך התוכנה ישלח נתונים בין מחשב מסויים לבקר והפוך.**

**הבעיות שנתקלתי בהם במהלך בניית הפרויקט הם:**

1. **איך לגרום לכלי משחק לבצע מהלך בדרך מסויימת. הפתרון שלי היה בעת ביצוע המהלך ניצור מופע ללוח משחק חדש ונבצע שם את המהלך ונבדוק שהוא מהלך חוקי.**
2. איך לתכנן את הממשק הגרפי שיראה מהלכי משחק בצורה יפה. יש שני פתרונון לבעיה הזאת. הראשונה היא להשתמש בקנבס ולצייר את הלוח ולצייר אנימציה של תזוזה לכל חלק(זה הפתרון הקשה לכן לא השתמשתי בו) והפתרון השני הוא להשתמש בתמונות לייצג לוח משחק ותזוזה על ידי יצירת התמונה במקום אחר.
3. איך לשלב את שליחת המהלכים בין מחשב למחשב בשימוש בארדואינו. הפתרון היה יחסית פשוט. יהיה לי שתי תוכנות בכל מחשב ואצטרך לבצע מהלך מסויים עבור כל שחקן בשתי המחשבים ולשלוח את המידע על ידי ה **Bluetooth**.

פרק 6 - נספחים

<https://www.chessprogramming.org/Main_Page>

<https://en.wikipedia.org/wiki/Chess_strategy>

<https://learn.sparkfun.com/tutorials/using-the-bluesmirf/all>

<https://www.chess.com/forum/view/general/sound-fx-for-moves>

<http://tutorials.jenkov.com/javafx/menubar.html>

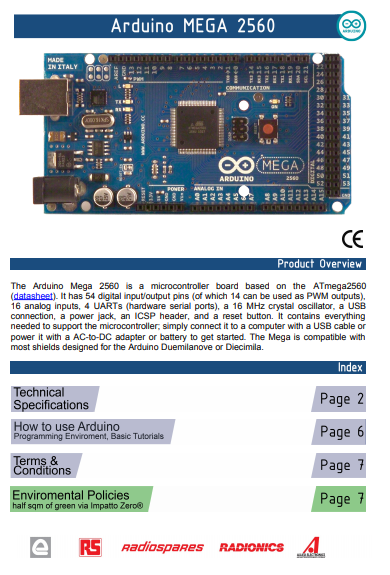
<https://cyaninfinite.com/rtc-module-with-serial-lcd-display/>

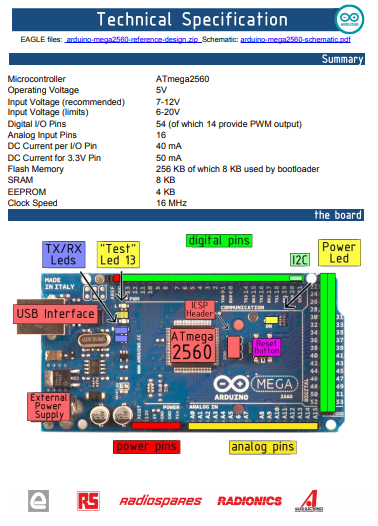
<https://stackoverflow.com/questions/53453212/how-to-deploy-a-javafx-11-desktop-application-with-a-jre>

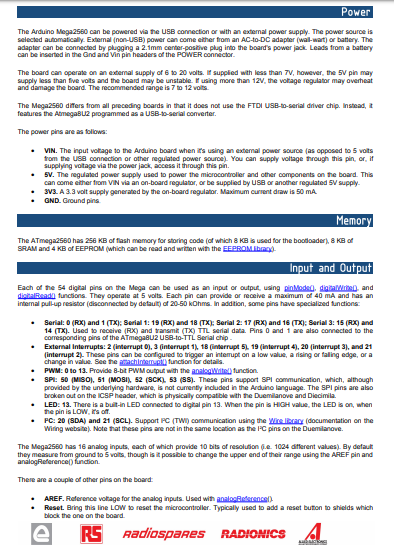
<https://openjdk.java.net/jeps/343>

6.1 דפי יצרן

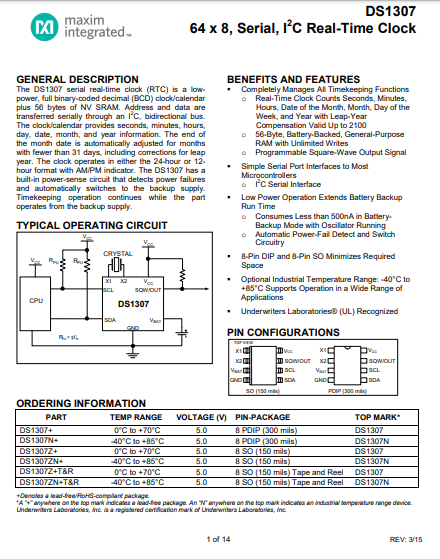
**1) ארדואינו מגה 2560**

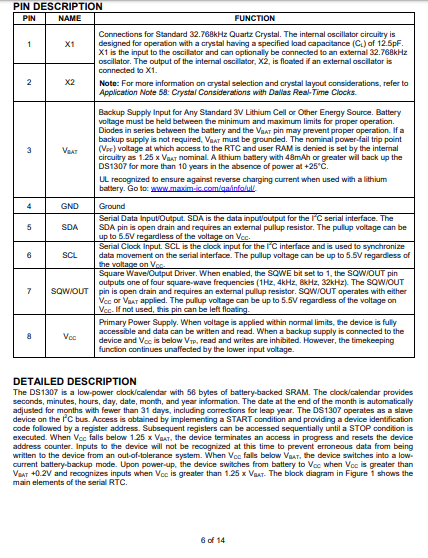






**2) Tiny RTC**





**3) Bluetooth**

