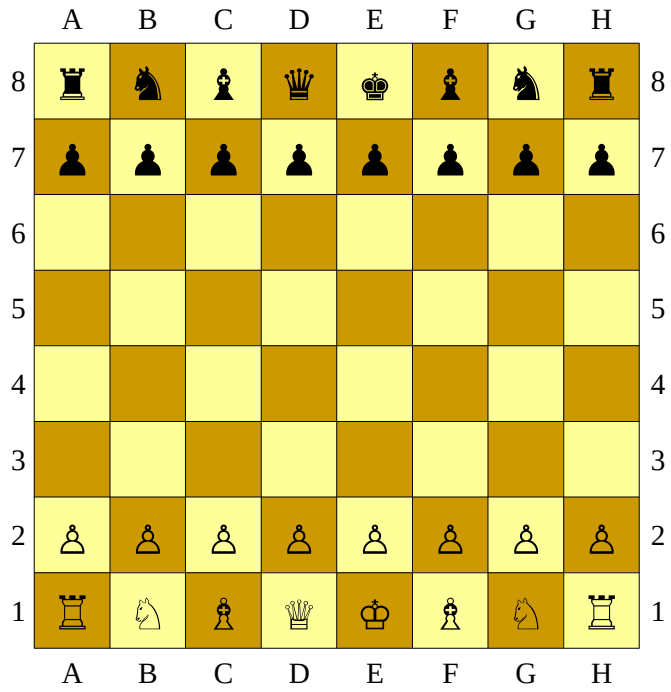
Semesterprojekt: Schachbrett

Gefragt ist ein Programm, das ein Schachbrett mit aufgestellten Figuren anzeigt, auf dem gespielt werden kann. Ein Schachbrett besteht aus 64 quadratischen, abwechselnd hell und dunkel eingefärbten Feldern, auf denen jeweils eine Figur stehen kann (die Abbildung zeigt die Grundaufstellung). Die Felder werden nach Spalte (Buchstabe) und Zeile (Ziffern) benannt, konventionsgemäß ist das links unterste Feld dunkel und trägt die Bezeichnung A1. Die schwarze Dame steht bspw. auf D8.



Dabei sollen folgende Regeln implementiert werden:

- Figuren können nur abwechselnd weiß und schwarz gezogen werden; den ersten Zug führt weiß aus. (Ausnahme: Die ›Rochade‹, bei der König *und* ein Turm gezogen werden, s.u.)
- Der König (♔) kann in beliebige Richtungen ziehen, aber nur 1 Feld weit. (Ausnahme: Rochade, s.u.)
- Die Dame (♕) kann beliebig weit senkrecht, waagrecht oder diagonal ziehen.
- Der Turm (♖) zieht beliebig weit senkrecht oder waagrecht.
- Der Läufer (♗) zieht beliebig weit diagonal.
- Der Springer (oder auch ›Rössel‹ oder ›Pferd‹ – ♘) zieht zwei Felder gerade und eines quer. Dabei kann er (und nur er) andere Figuren überspringen (eigene oder fremde – deswegen ›Springer‹).
- Der Bauer (♙) zieht ein Feld nach vor (in Richtung gegnerischer ›Grundlinie‹, siehe Grundaufstellung), schlägt aber ein Feld schräg (in ›Generalrichtung‹). Bei seinem ersten Zug kann er auch zwei Felder weit ziehen.

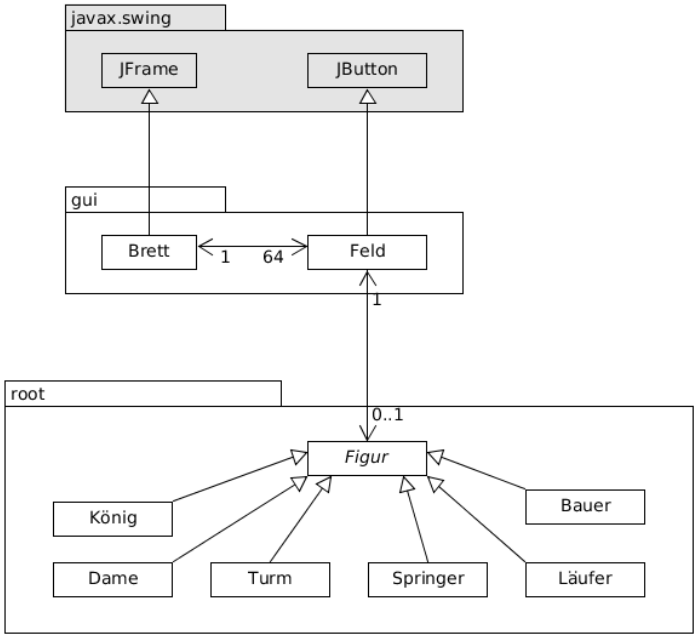
Eine Figur kann auf jedem Feld abgestellt werden, auf dem keine eine eigene steht. Eine gegnerische, die ggf. dort ist, wird vom Brett entfernt (sie wird ›geschlagen‹).

Es gibt noch eine Reihe von Spezialregeln, die Sie aber nur implementieren, wenn Sie Spaß daran haben:

- ›Rochade‹: Man kann als *einen* Zug einen Turm neben den König ziehen und den König auf die andere Seite neben den Turm stellen, sofern weder betroffene Turm, noch der König schon gezogen wurden und keine überzogenen Felder von gegnerischen Figuren bedroht sind.
- Bauernumwandlung: erreicht ein Bauer die gegnerische Grundlinie, so wird er durch eine Figur ersetzt, die sich der Spieler aussuchen kann (praktisch ist das fast immer eine Dame und nur ganz selten ein Springer).
- ›Schlagen *en passant*‹: wenn ein Bauer bei seinem ersten Zug zwei Felder weit zieht und dabei neben einem gegnerischen zu stehen kommt, so kann er von Letzterem so geschlagen werden, wie wenn er nur ein Feld weit gezogen worden wäre – aber nur unmittelbar nach seinem 2-Felder-Zug.

Tipps zur Lösung der Aufgabe »Schachbrett«:

Da wir eine solide Architektur haben wollen, schlage ich eine Klassenstruktur wie die folgende vor:



gui:

Auf dem Bildschirm sichtbar sind das Brett und seine Felder. Daher kommen diese Beiden Klassen ins Packet ›`gui`‹. `Brett` kann ohne Weiteres mit einem Grid-Layout (8x8) versehen werden, `Feld` könnte eine Ableitung von `JButton` sein (das macht es einfach, Mausklicks abzufangen).

Eine abstrakte Klasse ›`Figur`‹ kann praktisch werden, weil so ein einheitlicher Datentyp vorhanden ist, der für alle Figuren benützt werden kann. Da sie oder die daraus abgeleiteten Klassen nicht direkt sichtbar sein müssen, kommen diese ins Packet ›`root`‹. (N.b.: die *Anzeige* des Symbols am Schirm muss in dieser Architektur die Klasse ›`Feld`‹ übernehmen.)

Das hier soll Ihnen wie immer nur ein Anhalt und Leitfaden sein, auch ganz andere Lösungen sind denkbar und sinnvoll!

Vielleicht fragen Sie sich, ob die vielen Klassen und Objekte nicht ein gewisser »Overkill« für ein Projekt dieser Größe darstellen. Antwort klipp und klar: ja, sind sie! Man könnte simplere Lösungen konstruieren (z.B. ein char-Array stellt das Brett dar, mit Buchstaben als Figuren), und Methoden die Züge auf dem Feld ausführen. Aber darauf zielen wir nicht ab! Uns geht es darum, uns den Umgang mit Klassen und Objekten und den entsprechenden Zusammenhängen aufzuschließen!

Wir machen uns erst einmal Gedanken, wer was »erfahren« muss, damit erfolgreich gespielt werden kann. Im Klartext: welche Variablen müssen wo entstehen? (Anm.: jedesmal, wenn wir davon sprechen, dass »jemand etwas wissen muss«, »wir uns etwas merken wollen« o.Ä. heißt das nichts anderes als dass wir irgendwo eine oder mehrere Variablen anlegen müssen. Wo das sein soll oder kann, darüber muss man natürlich ebenfalls nachdenken.)

BMT – EPro II – SS22 – Semesteraufgabe

Beginnen wir mit der

Klasse `Figur`:

Eine Schachfigur muss wissen, welche Farbe sie hat, und auf welchem Feld sie steht.

Wie die Farbe gespeichert wird, ob als `String farbe` (mit den Werten „weiß“ & „schwarz“), als `char farbe` ('w'/'s') oder als `boolean weißeFigur` (mit naheliegender Interpretation von `true` und `false`) ist derzeit völlig nebensächlich. Wenn es soweit ist, dass irgendjemand fragen wird (vermutlich ein Objekt vom Typ `Brett`), werden wir die Farbe mit einer get-Methode auf die geeignete Weise herausgeben.

Das Feld, auf dem sie steht, können wir einfach mit 2 char-Variablen darstellen: `spalte` (enthält 'A' – 'H') und `zeile` ('1'-'8').

Klasse `Brett`:

Das Schachbrett sollte seine Figuren kennen: 32 Figuren sind es, also bietet sich 1 array
`Figur[] figur = new Figur[32];`

an, oder 2 arrays `Figur weißeFiguren = new Figur[16];` (schwarz analog) oder ähnlich oder auch hier ein 2-dimensionales Array `Figur[][] figur = new Schachfigur[2][16];`

Diese Entscheidung ist letztendlich Geschmackssache! Alle Varianten haben ihre Vor- und Nachteile. Aus Gründen der Einfachheit hat für mich das 1-dimensionale Array mit 32 Elementen die Nase vorn.

Das Schachbrett sollte auch »wissen« wer gerade den (nächsten) Zug ausführt, z.B. mit

```
boolean weißAmZug = true;
```

Die Figur, die gerade gezogen wird, sollten wir uns auch merken können:

```
Figur ziehendeFigur;
```

(weiter unten mehr dazu)

Aufstellen der Figuren:

Im Konstruktor sollten auch die Figuren aufgestellt werden. Es empfiehlt sich eine Methode

```
void aufstellen();
```

die das übernimmt und aus dem Konstruktor heraus aufgerufen wird! Die Methode sollte aufgerufen werden, nachdem die JButtons abgearbeitet sind (also nach den beiden Schleifen). Sie soll: die 32 Figuren-Objekte erzeugen UND die Buttons des Schachbrett mit den richtigen Beschriftungen bzw. Bildern belegen (deswegen muss es auch *nach* den Buttons passieren).

> (**Tipp**: Der Unicode-Zeichensatz enthält u.a. auch die oben dargestellten Symbole für Schachfiguren, und zwar die Unicodes \u2654 ... \u265f.
>
> Falls Sie aber ein *Bild* auf die Feld-Buttons bringen wollen, geht das geht folgendermaßen:
>
> Ihr Projektordner enthält einen Ordner ›src‹. Erzeugen Sie dort einen Ordner für Graphiken (z.B. ›pngs‹ oder ›gifs‹). Speichern Sie Ihre Bilddateien dort.
>
> Nachdem Sie einen Button erzeugt haben (z.B. `btnFeld[3][6]`), bekommt er mit folgender Zeile ein Bild (das Folgende ist eine Zeile):

```
btnFeld[3][6].setIcon (
    new ImageIcon (getClass().getRessource("/pngs/turm_weiß.png")));
```

Achten Sie darauf, dass Sie oben `import javax.swing.*;` kodiert haben, sonst müssen Sie ImageIcon mit dem Packetpfad `javax.swing.` qualifizieren!)

Züge ausführen:

Nachdem die Figuren nun aufgestellt, und die Buttons entsprechend beschriftet/bebildert sind, gilt es, Züge auszuführen.

Erstens müssen wir entscheiden, WO ein Zug programmiert werden könnte: da mehrere Felder betroffen sind (Ausgangsfeld/Zielfeld sowie alle Felder dazwischen, die wir überprüfen müssen, ob sie leer sind) und mehrere Figuren betroffen sein können (gezogene Figur und ev. geschlagene Figur), empfiehlt sich die Klasse `Schachbrett`, um alles Nötige zu koordinieren. Ein Zug besteht aus 2 Klicks: dem Ausgangs- und dem Zielfeld, (die immer abwechselnd geklickt werden, beginnend mit dem Ausgangsfeld)

Der Zug gliedert sich in folgende Schritte:

1. Feststellen, ob eine Figur der richtigen Farbe angeklickt ist

Wenn die Figur auf dem Button die falsche Farbe hat oder das Feld leer ist, machen wir nichts und sind fertig. (Der Nächste Klick ist dann wieder ein Klick auf ein Ausgangsfeld)

2. Wurde eine Figur der richtigen Farbe ausgewählt, merken wir uns diese Figur (`Figur ziehendeFigur`)

Das schließt die erste Phase des Zugs ab. Wir warten auf den 2. Klick des Users, der per Definition auf das Zielfeld geht.

3. Feststellen, ob es sich um einen erlaubten Zug handelt:

Dazu ist es das Einfachste, wir programmieren in der Klasse der Figur eine boolesche Methode `kannZug()`, die für den Anfang immer `true` zurück liefert! Das können wir später verfeinern, indem `true` nur dann zurück gegeben wird, wenn es sich um einen für die Figur erlaubten Zug handelt. Also bspw. für einen Läufer, wenn es sich wirklich um einen diagonalen Zug handelt. Ist der Zug nicht o.k., geben wir `false` zurück und der Aufrufer macht weiter gar nichts und wartet weiter auf einen zweiten Klick. Oder wir verwerfen die Figur, die am Zug ist (`ziehendeFigur = null;`), das würde heißen, man muss (bzw. kann) nach einem ungültigen Zug neu anfangen.

4. Jetzt gilt es, den eigentlichen Zug auszuführen. Der gesamte Schritt 4 lässt sich am besten in der Klasse `Schachfigur` programmieren, da er für alle Figuren prinzipiell denselben Ablauf vorsieht:

Dass die Figur den Zug machen kann, wissen wir bereits – aber lässt er sich auch ausführen? Hilfreich könnte eine boolesche Methode `okZug()` sein, die `true` liefert, wenn alle Felder zwischen Ausgangs- und Zielfeld leer sind und das Zielfeld *keine* Figur der enthält Farbe enthält.

Das Symbol der Figur muss vom Ausgangsfeld genommen werden (=Symbol und Objekt werden vom `Feld`-Objekt entfernt).

Sollte eine Figur das Zielfeld besetzen, muss diese geschlagen werden. D.h., das entsprechende Element des Figuren-Array null gesetzt. Beispiel:

```
figur[29] = null;
```

Damit würde Figur#29 aus dem Spiel genommen.

Die gezogene Figur muss auf das neue Feld gesetzt werden, also

▪ die Variablen ihrer Position verändert

▪ ihr Symbol muss auf dem Zielfeld angezeigt werden

5. Der Zug ist abgeschlossen, und im Schachbrett-Objekt muss noch die `ziehendeFigur` zurück auf `null` gesetzt werden und die Farbe am Zug muss gewechselt werden, um den nächsten Zug einzuleiten.

Dazu noch ein Tipp: Wenn der Benutzer also auf ein Feld klickt, so muss das Schachbrett entscheiden können, ob das ein erster oder ein zweiter Teil des Zuges war (und vielleicht eine entsprechende Methode aufrufen)! Wenn `ziehendeFigur` null ist, muss es ein Zugbeginn sein, sonst ein Zugende!

Schlussbemerkung

Besonders beeindruckend fand ich in vergangenen Kursen zum Einen Lösungen, die ich bereits sehen konnte und ebenso, dass sehr viele von Ihnen sich so in die Aufgabenstellung hineingetigert haben! Besonders, dass so viele nachgefragt haben, wie Graphiken auf den Buttons hinzukriegen seien. Aber seien Sie nicht entmutigt, wenn nicht alles gleich auf Anhieb so funktioniert, wie Sie sich das vorstellen. Leider ist das ganz normal, es sind noch keine Meister vom Himmel gefallen. Das Wichtigste ist: haben Sie weiter Freude an der Lösungssuche, lassen Sie sich nicht verdrießen, und – Sie wissen – fragen Sie!