Verteilte Systeme

Übung B

Bearbeitungszeit 2 Wochen

Die Übungen adressieren Techniken zum Multi-Threading.

Aufgabe: Distributed Chess

Die Entwicklung eines Schach-Spiels auf Java-Basis ist an einigen kritischen Stellen ins Stocken geraten. Nach Entwicklung aller Komponenten und Medien, und des damit verbundenen Verbrauchs fast aller Entwicklungsgelder, hat sich herausgestellt, dass Bedienbarkeit und Performance des Spiele-Kerns bei der Berechnung längerer Zugsequenzen zu wünschen übrig lassen. Das erfahrene Design-Team hat mehrere mögliche Strategien zur Lösung der Probleme vorgeschlagen, welche Ihr nun implementieren und testen sollt.

Die wichtigsten Klassen der bestehenden Implementierung sind dabei:

- **`PieceType`**: Interface für Figurtypen
- **`ChessPieceType`**: Enum für Schach-Figurtypen, i.e. PAWN, KING, usw.
- **`Piece`**: Interface für konkrete Figuren welche durch Typ und Farbe definiert sind
- **`ChessPiece`**: Enum für positionelle Schach-Figuren beider Farben
- **`Board`**: Interface für Brettspiele
- **`ChessTableBoard`**: Konkrete Implementierung eines Schach-Spiels
- **`GamePanel`**: Swing-Panel für Brettspiele, bestehend aus **`ControlPanel`** und **`BoardPanel`**
- **`ChessClient`**: Swing basierter Schach GUI-Client

Das Paket `de.htw.ds.board` enthält zusätzlich ein UML Klassen-Diagramm (`board-games.png`) welches die verschiedenen Klassen, ihre Vererbungsbeziehungen und Relationen bietet. Dort werden auch die Klassen eines zweiten Brettspiels erwähnt welches nicht im Code-Umfang enthalten ist, deren Existenz jedoch einige gewählte Abstraktionen erklärt. Wer mit UML nicht vertraut ist, dem helfen eventuell folgende beiden URLs weiter:

- http://en.wikipedia.org/wiki/Class_diagram
- http://yuml.me/diagram/scruffy/class/samples

Benennt zuallererst die Klasse **`de.htw.ds.board.BoardPanelSkeleton`** nach **`BoardPanel`** um. Danach sollte das Projekt komplett ohne Compilefehler, und die Applikation **`de.htw.ds.board.chess.ChessClient`** mittels folgender Laufzeit-Argumente startbar sein:

- Modus: USER_INTERFACE
- Board-Klasse: z.B. `de.htw.ds.board.chess.ChessTableBoard`
- Suchtiefe: 5
- Anzahl Brett-Zeilen: 8
- Anzahl Brett-Spalten: 8

Statt der Anzahl der Brett-Zeilen und -Spalten kann auch alternativ eine Figurstellung als X-FEN Zustand angegeben werden (in Anführungszeichen wegen enthaltener Leerzeichen), z.B. `"6kr/2p5/5Q2/p5Np/5B2/bq5P/5PP1/4R1K1 w - - 0 31"`

Aufgabe 1: Flüssiges GUI ohne Exceptions

Ein Zug einer weißen Figur wird im GUI durch Klicken auf das Quell- und Zielfeld angegeben (bei Rochade nur Quell- und Zielfeld des Königs). Die Applikation beginnt daraufhin sofort mit der Berechnung des Gegenzugs für die schwarzen Figuren.

Die KI der Applikation (siehe `Board#analyze(int)`) sorgt jedoch dafür dass der GUI Event-Handler **`BoardPanel#handleFieldSelection(byte, byte)`** lange braucht um beendet zu werden; da der graphische Refresh von GUI-Elementen wiederum erst ausgeführt wird wenn alle GUI Event-Handler beendet sind, führt dies dazu dass der ausgeführte Zug erst angezeigt wird wenn der Computer seinen Gegenzug berechnet und ebenfalls ausgeführt hat. Des Weiteren lassen sich die Knöpfe für Zugrücknahme und Reset zwar drücken, aber der Benutzer erhält zum einen kein Feedback bis die begonnene und nun überflüssig gewordene Berechnung des Gegenzugs abgeschlossen ist.

Dies ist natürlich so nicht akzeptabel, daher sollt ihr für Abhilfe sorgen indem ihr Multi-Threading einsetzt um diese Probleme zu beheben. Verlagert dazu in der oben genannten Methode von BoardPanel den Aufruf von `BoardPanel#performComputerMove()` in einen separaten Thread, und startet diesen. Diese Maßnahme alleine „löst“ schon die oben genannten beiden Probleme.

Dummerweise wird dadurch alleine die asynchrone Berechnung nicht abgebrochen wenn ein Zug zurück genommen wird während die KI rechnet – CPU und Lüfter werden also weiter belastet bis die Operation ihr natürliches Ende nimmt. Schlimmer noch, wie so oft ergeben sich durch die Umstellung im Code nun kritische Abschnitte welche bei Multi-Threading Fehler verursachen!

Lösungsansatz 1: Mutual Exclusion (Mutex)

Versucht daher einen gegenseitigen Ausschluss der kritischen Code-Abschnitte mit synchronized-Blöcken gegen „`this.board`". Dies sollte das Problem mit den Exceptions in den Griff bekommen:

- `BoardPanel#performComputerMove()`: Der gesamte Try-Block
- `ControlPanel#handleResetButtonPressed()`: Die beiden Zeilen in denen X-FEN Zustände gelesen bzw. gesetzt werden
- `ControlPanel#handleBackButtonPressed()`: Alle Zeilen vor dem Refresh-Aufruf

Dummerweise bringt die Maßnahme das Problem des „eingefrorenen“ GUI-Verhaltens zurück, da durch den Mutex letztlich die Ausführung des GUI Event-Handlers im Control-Panel so lange angehalten (und damit dessen Ende verzögert) wird bis die KI fertig ist. Eine befriedigende Lösung des ursprünglichen Problems mittels Mutex lässt sich daher (wie so oft) nicht erreichen; nehmt daher die synchronized-Blöcke wieder aus dem Code heraus.

Lösungsansatz 2: Thread-Unterbrechung

Eine Alternative zum gegenseitigen Ausschluss mittels Mutex ist es die Bearbeitung des KI-Threads abzubrechen sobald einer der beiden Knöpfe gedrückt wird; durch das Abbrechen wird (außer unter seltenen Umständen) vermieden dass die kritischen Abschnitte parallel ausgeführt werden können, das Problem damit im Normalfall erfolgreich umgangen, und zudem verhindert dass die CPU nach Rücknahme eines Zuges weiter sinnlos belastet wird.

Dazu ist es allerdings notwendig die Ausführung eines Threads in Java abzubrechen, was nicht ganz einfach ist: Die ursprünglich dafür vorgesehene Methode `Thread#stop()` ist deprecated und ihrer ursprünglichen Funktion beraubt - siehe JavaDoc der Methode für die Hintergründe. Der Nachfolger **`Thread#interrupt()`** dagegen bricht einen Thread erst ab wenn er entweder blockiert um das Ende einer I/O-Operation abzuwarten, oder er so nett ist seinen eigenen Interrupted-Zustand mittels der statischen (sic!) Methode **`Thread#interrupted()`** zu überprüfen. Falls ein Thread also niemals I/O betreibt und zudem auch nicht so nett ist seinen eigenen Status ab und an abzufragen, dann kann er auch nicht mehr abgebrochen werden.

Fügt dazu der Klasse BoardPanel eine neue Instanzvariable „`asynchronousOperation`“ vom Typ `Thread` hinzu, und setzt diese jedes mal bevor ihr in **`BoardPanel#handleFieldSelection(byte, byte)`** den neuen Thread startet; falls ihr pedantisch veranlagt seid dürft ihr das `Runnable` des Threads zudem mittels try-finally Block so stricken dass dem Feld wieder der Wert null zugewiesen wird sobald alle anderen Operationen beendet sind. Außerdem muss noch die bislang leere Methode **`BoardPanel#interruptAsynchronousOperation()`** so angepasst werden dass sie dem in der Instanzvariable gespeicherten Thread die Nachricht `#interrupt()` zukommen lässt.

Dies alleine reicht aber leider immer noch nicht aus weil die KI-Zuganalyse leider weder I/O betreibt, noch so nett ist `Thread#interrupted()` immer wieder mal aufzurufen. Letzteres lässt sich jedoch leicht ändern weil die Zuganalyse rekursiv implementiert ist:

- `AbstractBoard#analyze(int)` bzw. ruft `ChessTableBoard#analyzeRecursively(int)` auf
- letztere Methode ruft `ChessTableBoard#analyzeRecursively(AbsoluteMotion[], int)` auf
- letztere Methode ruft wieder `ChessTableBoard#analyzeRecursively(int)` auf

Kopiert daher die Klasse ChessTableBoardTemplate nach ChessTableBoard1, und überschreibt dort die Methode **`#analyzeRecursively(AbsoluteMotion[], int)`** so dass zuerst `Thread#interrupted()`, und danach die Methoden-Implementierung der Superklasse (mittels *super* statt *this*) aufgerufen wird. Startet sodann das GUI einmal mit `ChessTableBoard1` und einmal mit `ChessTableBoard`; der Unterschied im Verhalten bei Rücknahme von Zügen sollte offensichtlich sein.

Aufgabe 2 & 3

TBD, wird noch per Mail kommuniziert.