

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:

- **PieceParty**: Enum für die beiden Kontrahenten WHITE und BLACK
- **PieceType**: Interface für Figurtypen
- **ChessPieceType**: Enum für Schach-Figurtypen, i.e. PAWN, KING, usw.
- **Piece**: Interface für positionelle Figuren beider Opponenten
- **ChessPiece**: Enum für konkrete Schach-Figuren
- **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 `ControlPanel` 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 abubrechen 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 abubrechen, 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: Multi-Threading mittels Java 8 Stream-API

Die Methode `ChessTableBoard#getCandidateMoves()` iteriert über alle Spielfiguren der aktiven Seite, und ruft für jede Spielfigur die Methode `#collectCandidateMoves()` auf. Diese Aufrufe sind voneinander zwar lose verknüpft (durch die Akkumulation des Ergebnisses), aber eine Vertauschung der Reihenfolge liefert kein falsches Ergebnis; daher kann die Operation im Prinzip auf multiple Threads verteilt implementiert werden.

Da in der Implementierung bereits das **Stream-API** zum Einsatz kommt, soll auch dieses zum Parallelisieren verwendet werden: Ein *Stream* ist dabei ein Konstrukt welches konzeptionell zwischen Mengen und Iteratoren angesiedelt ist, und es u.A. erlaubt ähnlich wie in einer Datenbank multiple Mengenoperationen zu deklarieren bevor diese in einer terminalen Operation zur Ausführung kommen. Diese Streams arbeiten im Normalfall sequentiell, können aber leicht auf Parallelität mittels Multi-Threading umgestellt werden!

Kopiert dazu die Klasse **ChessTableBoardTemplate** nach **ChessTableBoard2**, und kopiert zudem die oben genannten beiden Methoden in diese Klasse. Ändert nun `ChessTableBoard2#getCandidateMoves()` so ab dass dem Ergebnis von „`this.pieceStream()`“ als erstes die Nachricht `#parallel()` gesendet wird bevor die zwei anderen Operationen ausgeführt werden; dies liefert einen neuen Stream der seine Elemente automatisch in Segmente gruppiert, und die nachfolgenden Operationen auf einem internen Thread-Pool basierend parallelisiert ausführt.

Dies alleine sorgt nun zwar bereits für parallele Ausführung, aber dummerweise ergibt sich damit ein neues Problem: Die Menge `#candidateMoves` ist nun vielen Threads bekannt (den Pool-Threads die bei den Stream-Operationen implizit zum Einsatz kommen), und ihre Größe wird während der parallelen Prozessierung verändert. Damit sind alle ändernden Mengenoperationen, sowie alle Sequenzen beliebiger Mengenoperationen prinzipiell kritische Abschnitte, und verlangen entsprechend nach Synchronisation mittels **Mutex**!

Als Mutex verwenden wir einen **Monitor**, und zwar die Menge `#candidateMoves` selbst; in Java sind alle Objekte automatisch Monitore, und die Java-Klasse `Object` hätte man daher aus technischer Sicht eigentlich „Monitor“ nennen müssen. Synchronisiert nun

- alle Einzel-Methodenaufrufe gegen diese Menge durch Dekorierung bei Erzeugung (i.e. Verschachtlung) mittels `Collections#synchronizedList()`
- die Zweier-Sequenz von Methodenaufrufen gegen diese Menge in `#collectCandidateMoves()` mittels explizitem `synchronized`-Befehl

Schaut Euch nun die Spezifikationen der CPUs Eurer Test-Systeme auf **SMT** (Hyperthreading) bzw. **kein SMT** an, und schätzt darauf basierend welchen Skalierungserfolg ihr aufgrund der Bestückung des Systems beim Multi-Threading maximal erwarten dürft. Startet sodann das Schach-Programm unter Angabe der Klasse **ChessTableBoard2** in den Laufzeitargumenten, und überprüft ob diese Maßnahme eine befriedigend beschleunigte Berechnung der Computerzüge zur Folge hat. Falls nicht, überlegt woran dies liegen könnte.

Aufgabe 3: Multi-Threading in Rekursionen

Die Methode **AbstractBoard#analyze(int)** analysiert die aktuelle Stellung auf die Zugsequenz die bei bestmöglichem Spiel beider Kontrahenten entsteht (Mini-Max Prinzip). Solche Algorithmen sind stets rekursiv aufgebaut, und brechen bei Erreichen einer vorgegebenen Suchtiefe ab. Die Rekursion wird hier erreicht indem **#analyze(int)** die Methode **#analyzeRecursively(int)** aufruft; diese kloniert die aktuelle Stellung und sendet dem Klon die Nachricht **#analyzeRecursively(AbsoluteMotion[], int)**, welche wiederum rekursiv die Methode **#analyzeRecursively(int)** aufruft so lange die maximale Suchtiefe noch nicht erreicht ist.

Kopiert nun die Klasse **ChessTableBoardTemplate** nach **ChessTableBoard2**, und kopiert dorthin die Methode **ChessTableBoard#analyzeRecursively()** als **#analyzeRecursivelyMT(int)**; überschreibt zudem die Methode **#analyze(int)** selbst, so dass diese neue Methode immer dann aufgerufen wird wenn die Suchtiefe größer als vier ist; damit wird sichergestellt dass das Multi-Threading nur zum Einsatz kommt wenn auch genügend parallelisierbare Arbeit vorhanden ist um die mit Multi-Threading verbundenen Kosten mehr als wieder einzuspielen.

Ändert nun die Methode **#analyzeRecursivelyMT(int)** so ab dass in der Schleife der Aufruf der Nachricht **#analyzeRecursively(AbsoluteMotion[], int)** auf Basis von Multi-Threading mittels Futures erfolgt. Beachtet dabei folgendes:

- Definiert einen Thread-Pool mit so vielen Threads wie Prozessoren vorhanden sind als statische Variable, und erzeugt die Futures mittels indem ihr dem Pool die Nachricht **#submit()** sendet; sorgt dabei dafür dass *Daemon-Threads* verwendet werden indem ihr bei der Erzeugung des Thread-Pools eine geeignete Thread-Factory (siehe **Threads#newDaemonThreadFactory()**) übergebt. Dies erlaubt dass Prozess noch „normal“ beendet werden kann ohne den Thread-Pool wieder zu schließen.
- Die in der Schleife auf den Aufruf von **#analyzeRecursively(AbsoluteMotion[], int)** folgenden Befehle müssen dagegen wieder im Parent-Thread ausgeführt werden; dies ist mit Futures recht einfach, kopiert die Befehle einfach an die Stelle der zweiten Schleife wo ihr mittels **Future#get()** die Child-Threads resynchronisiert und das Operationsergebnis übernommen habt.
- Sorgt zudem dafür dass eventuell in **ExecutionException** verpackte Exceptions aus einem der Child-Threads im Eltern-Thread passend gecastet erneut geworfen werden (Tip: Hier sind 3 Arten zu unterscheiden).
- Das Auftreten von **InterruptedException** im Parent-Thread soll nicht speziell behandelt werden, da die komplette Zusanalyse durch Deklaration dieser Exception bereits auf unterbrechungsbedingten Abbruch der Operation hin ausgelegt ist.

Schaut Euch nun die Spezifikationen der CPUs Eurer Test-Systeme auf **SMT** (Hyperthreading) bzw. **kein SMT** an, und schätzt darauf basierend welchen Skalierungserfolg ihr aufgrund der Bestückung des Systems beim Multi-Threading maximal erwarten dürft. Startet sodann das Schach-Programm unter Angabe der Klasse **ChessTableBoard3** in den Laufzeitargumenten, und überprüft ob diese Maßnahme eine befriedigend beschleunigte Berechnung der Computerzüge zur Folge hat. Falls nicht, überlegt woran dies liegen könnte.