Technische Hochschule Rosenheim, Fakultät für Informatik

Vorlesung: Verteilte Verarbeitung SS 2020 Prof. Dr. Gerd Beneken

Praktikum 1 von 3 – Reaktive Systeme

Ziel dieses Praktikums

Dieses Praktikum hat zwei Ziele, erstens sollen Sie nach erfolgreichem Abschluss aller Praktikumsaufgaben verteilte Systeme besser verstehen und diese praktisch umsetzen können. Zweitens sollen Sie möglichst viele praktische Erfahrungen beim Programmieren sammeln, damit sie das Praxissemester möglichst erfolgreich überstehen.

Ziel ist es, dass Sie während des Programmierens möglichst umfangreich Feedback zu Ihren Ergebnissen erhalten und zwar von dem betreuenden Professor und auch von Ihren KommilitonInnen. Je mehr Feedback Sie erhalten, desto mehr und besser lernen Sie. Ich hoffe, dass dadurch auch Ihre Motivation steigt, sich in das Thema Programmierung zu vertiefen.

Alle Aufgaben sind jeweils über einen Zeitraum von etwa einem Monat zu bearbeiten. Das Semester wird in vier große Praktika unterteilt. Diese sind jeweils im Laufe des Semesters (weit vor dem Termin des Kolloquiums abzugeben).

Ziel der Aufgabe: Reaktive Systeme

Wenn Sie diese Aufgabe erfolgreich abgeschlossen haben, können Sie

- Dateizugriff in Java und anderen Sprachen über Streams und Sie haben das Konzept der Streams verstanden
- Umwandeln von Java Objekten in JSON-Strings und Sie haben grob verstanden wie JSON funktioniert.
- Programmierung eines endlichen Automaten in einer beliebigen Programmiersprache.
- Bau eines einfachen reaktiven Systems und sie haben verstanden, was ein reaktives System ist.
- Konzepte der synchronen und asynchronen Verarbeitung: Polling, Producer / Consumer, Executor-Framework und Futures.

Der Aufgabentext wird Ihnen völlig unlösbar erscheinen (das ist Absicht). Die Kunst besteht jetzt darin, dieses **viel zu komplizierte Problem** in für Sie machbare Teilschritte zu zerlegen und diese dann, soweit Sie kommen, abzuarbeiten. Also von einem kleinen Teilprogramm zum nächsten und diese dann wieder zu einer Gesamtlösung zu integrieren.

Aufgabe:

Wir versuchen in dieser Aufgabe ein **Werkzeug zur Synchronisation von Verzeichnissen** zu erstellen, irgendwas zwischen Dropbox und git. Dieses Werkzeug soll über mehrere Rechner hinweg funktionieren. Es überwacht ein Quellverzeichnis und synchronisiert die darin befindlichen Dateien mit einem Zielverzeichnis. Das eigentliche Kopieren der Dateien programmieren wir nicht, wir finden nur heraus, welche Dateien zu synchronisieren wären. Wir bauen dieses Werkzeug nun Schritt für Schritt auf.

Schritt 0

Erstellen sie für diese Aufgabe ein **Ticket in Gitlab** mit einer Beschreibung der hier dargestellten wichtigsten Schritte. Erzeugen Sie aus dem Ticket heraus einen **Feature-Branch** und checken diesen auf ihrem Rechner aus. Erstellen sie auf diesem Feature-Branch dann ihr Projekt unter der Verwendung von Gradle. Erstellen sie zusätzlich eine **Build-Pipeline in Gitlab**, welche ihren Code übersetzt, die Testtreiber ausführt und die

Testcoverage berechnet. ***Ich erstelle noch für alles ein Trainingsvideo***. Sie können schon mit dem Programmieren beginnen, den Code können Sie später kopieren.

Schritt 1

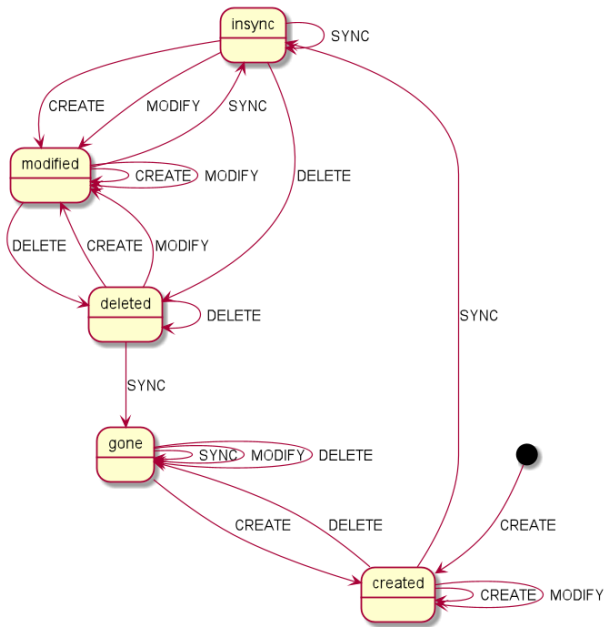
Schreiben sie eine Klasse **WatchedFile.** Objekte dieser Klasse stellen jeweils eine Datei des überwachten Verzeichnisses dar und deren Zustand. WatchedFile hat den Dateinamen und das Verzeichnis, sowie den letzen bekannten Zeitstempel als Attribute. WatchedFile hat fünf Zustände: `INSYNC`, `CREATED`, `MODIFIED`, `DELETED` und `GONE`. Die Zustände bedeuten folgendes:

- INSYNC: das Verzeichnis und die synchronisierten Verzeichnisse sind synchron.
- CREATED: Lokal wurde eine neue Datei erzeugt.
- MODIFIED: Eine existierende (schon mal synchronisierte) Datei wurde lokal verändert. Die Datei wurde schon mal synchronisiert.
- DELETED: Eine existierende (schon mal synchronisierte) Datei wurde lokal gelöscht.
- GONE: Die Datei ist endgültig gelöscht, z.B. nachdem sie lokal gelöscht und dann synchronisiert wurde oder sie wurde lokal erzeugt und dann gleich wieder gelöscht.

Das Eingabealphabet für den entsprechenden Automaten hat vier Symbole, diese werden vom **DirectoryWatcher** (siehe unten) geliefert:

- CREATE: Datei wurde erzeugt.
- DELETE: Datei wurde gelöscht.
- SYNC: Datei wurde synchronisiert.
- MODIFY: Datei wurde geändert.

Daraus ergibt sich dann ungefähr folgender etwas komplizierter Zustandsautomat für die Klasse WatchedFile:



Versuchen Sie diese Klasse testgetrieben zu entwickeln.

Schritt 2

Erstellen Sie eine Klasse „**DirectoryWatcher**“ mit einer eigenen main()-Methode. Achten Sie darauf, dass sie Unit-Tests für diese Klasse schreiben müssen, daher sollte die main() Methode eher kurz sein. Die Methoden von DirectoryWatcher machen Sie bitte NICHT static (bis auf main).

- DirectoryWatcher erhlät als **Kommandozeilen-Parameter** das Verzeichnis mitgeteilt, das er überwachen soll. Wenn kein Verzeichnis spezifiziert ist, verwendet ihr DirectoryWatcher das Verzeichnis in dem er gestartet wurde.
- DirectoryWatcher überwacht das angegebene Verzeichnis auf ihrem Laptop. Dazu verwenden Sie bitte das Interface java.nio.file.WatchService. Implementierungen dieses Interfaces (bitte recherchieren) lieferen ihnen Events jeweils zu neuen, geänderten und gelöschten Dateien in dem Verzeichnis: `CREATE`, `DELETE` und `MODIFY`
- Schreiben Sie die Überwachungsfunktion des DirectoryWatchers. Diese erzeugt **FileEvent** Objekte für jede gesehene Änderung. Ein FileEvent-Objekt hat den Dateinamen und das Symbol (CREATE, MODIFY …) als Attribute.
- Schreiben Sie den DirectoryWatcher so, dass dieser sinnvoll auf Fehler reagiert.
- Zum Testen geben Sie die Informationen über Änderungen in ihrer Log-Datei aus, bzw. loggen auf die Konsole (ohne System.out, sondern mit logger).

Schritt 3

Schreiben Sie eine Klasse **WatchedDirectory.** Diese verwaltet eine Liste mit Objekten der Klasse **WatchedFile.** Verwenden Sie für die Liste eine Map mit dem Dateinamen als Key und dem WatchedFile-Objekt als Value. Die Klasse hat eine Methode **update(FileEvent ev),** diese Methode bekommt ein FileEvent-Objekte übergeben. Die FileEvent-Klasse ist von ihnen zu programmieren, sie enthält nur zwei Attribute: Einen Dateinamen und ein Symbol des Eingabealphabets von oben. Update sorgt dafür, dass das WachtedFile Objekt für die Datei mit dem Dateinahmen den entsprechenden Zustandübergang im oben dargestellten Automaten durchführt. Damit kennt WatchedDirectory immer den aktuellen Zustand aller Dateien des überwachten Verzeichnisses.

- Entwickeln Sie WatchedDirectory testgetrieben.
- Implementieren Sie eine sync() Methode, welche die gespeicherten WatchedFile-Objekte auf einen OutputStream ausgibt. Die Ausgabe erfolgt im JSON-Format. sync(OutputStream out) sendet an alle gespeicherten WatchedFile-Objekte das Zeichen SYNC.
- Optional: Testen Sie sync() mithilfe eines ge-mockten OutputStream-Objektes (Framework z.B.Mockito)

Schritt 4

Ändern Sie die implementierten Klassen wie folgt:

1. WatchedDirectory wird in einem **eigenen Thread** ausgeführt. In einer Run-Methode liest WatchedDirectory eine **BlockingQueue fileEvents** aus. In der Queue befinden sich FileEvent Objekte. Mit diesen Objekten wird die update Methode ausgeführt.
2. DirectoryWatcher: Auch hier wird das Überwachen des Verzeichnisses auf ihrer Festplatte in einem eigenen Thread ausgeführt. Die run() Methode stellt fest, welche Veränderungen von Dateien im überwachten Verzeichnis stattgefunden haben. Sie erstellt daraus jeweils FileEvent Objekte. Diese schreibt sie in die BlockingQueue fileEvents.
3. Implementieren Sie einen weiteren Thread, der im Abstand von 10 Sekunden auf dem WatchedDirectory-Objekt sync(OutputStream out) aufruft und als OutputStream System.out übergibt. Damit müssten sie eine Konsolen-Ausgabe der Überwachung kriegen.
4. Wichtig: Die Threads, das WatchedDirectory-Objekt und das DirectoryWatcher-Objekt sowie die BlockingQueue werden von der main-Methode des DirectoryWatchers erzeugt. Zum „Zusammensetzen“ verwenden Sie das **Dependency-Injection Patten**, soweit das geht.

Schritt 5

Schreiben Sie eine Klasse SyncServer, ein Objekt von SyncServer wird in einem separaten Thread ausgeführt und wie in Schritt 4 erstellt. SyncServer ruft regelmäßig sync(OutputStream out) auf. Und schreibt die Ausgabe auf den jeweiligen TCP-Socket aller verbundenen Clients. Schreiben sie eine Klasse SyncClient, diese hat eine

main() Methode. SyncClient verbindet sich per Socket mit dem SyncServer Objekt, erhält die Daten vom SyncServer über den Socket und gibt diese auf der Konsole aus.

Schritt 6

Gehen Sie ihren Code noch mal durch. Und achten Sie auf folgendes:

- Die Build Pipeline läuft durch.
- Ihre Testabdeckung ist über 80%.
- Sie haben sich an alle Coding-Konventionen aus Java gehalten.
- Sie gehen sauber mit Fehlern / Exceptions um.
- Sie können alle wichtigen Parameter von außen als Kommandozeilenparameter oder als Property setzen.
- Sie verwenden durchgehend Logging und nicht System.out
- Ihr Code ist auf einem Feature-Branch.

Wenn das alles erfolgt ist oder wenn sie sinnvolle Zwischenstände haben, dann stellen sie einen Merge-Request an ihren Betreuer.