Praktikum Concurrency - Cooperation

Einleitung

Ziele dieses Praktikums sind:

- Sie können die verschiedenen Mechanismen zur Thread-Synchronisation sicher anwenden (Mutual-Exclusion, Condition-Synchronisation).
- Sie können das Monitor-Konzept (wait/notify, Lock & Conditions) in Java anwenden.
- Sie können Producer-Consumer Synchronisation praktisch anwenden.
- Sie wissen wie Deadlocks praktisch verhindert werden können.

Voraussetzungen

- Vorlesung Concurrency 1 bis 4

Tooling

- Installiertes JDK 17+
- Gradle 7.6+

Struktur

Ein Praktikum kann verschiedene Arten von Aufgaben enthalten, die wie folgt gekennzeichnet sind:

[TU] – Theoretische Übung

Dient der Repetition bzw. Vertiefung des Stoffes aus der Vorlesung und als Vorbereitung für die nachfolgenden Übungen.

[PU] – Praktische Übung

Übungsaufgaben zur praktischen Vertiefung von Teilaspekten des behandelten Themas.

[PA] – Pflichtaufgabe

Übergreifende Aufgabe zum Abschluss. Das Lösen dieser Aufgaben ist Pflicht. Sie muss bis zum definierten Zeitpunkt abgegeben werden, wird bewertet und ist Teil der Vornote.

Zeit und Bewertung

Für das Praktikum stehen die Wochen gemäss den Angaben in Moodle zur Verfügung.
Je nach Kenntnis- und Erfahrungsstufe benötigen Sie mehr oder weniger Zeit. Nutzen Sie die Gelegenheit den Stoff zu vertiefen, auszuprobieren, Fragen zu stellen und Lösungen zu diskutieren (Intensive-Track).
Falls Sie das Thema schon beherrschen, müssen Sie nur die Pflichtaufgaben lösen und bis zum angegebenen Zeitpunkt abgeben (Fast-Track).

Die Pflichtaufgaben werden mit 0 bis 2 Punkten bewertet (siehe *Leistungsnachweise* auf Moodle).

Referenzen

- Praktikumsverzeichnis – Quellcode, Projektstruktur

1. Concurrency 3 — Thread Synchronisation

1.1. Konto-Übertrag [PU]

Im Praktikumsverzeichnis finden sie das Modul `AccountTransfer`. Die Klasse `Account`, welche sie schon aus dem Unterricht kennen, repräsentiert ein Konto von welchem bzw. auf welches man Geld transferieren und den Kontostand abfragen kann.

Die Klasse `AccountTransferTask` ist ein `Runnable` welches einen Geldbetrag zwischen zwei Konten transferiert . Der entsprechende Ablauf ist in der Methode `accountTransfer` umgesetzt.

Um die Funktionalität zu verifizieren, wurde in der Klasse `AccountTransferSimulation` ein Testszenario umgesetzt, in welchem eine Grosszahl kleiner Beträge zwischen drei Konten transferiert wird. Dazu wird ein ExecutorService mit mehreren Threads verwendet, an welchen die TransferTasks übermittelt werden. Die Simulation gibt die Kontostände und das Vermögen (Summe der Kontostände) zu Beginn und am Ende des Transfers aus.

a. Was stellen Sie fest, wenn Sie die Simulation laufen lassen? Erklären Sie wie die Abweichungen zustande kommen.

b. Im Unterricht haben Sie gelernt, dass kritische Bereiche Ihres Codes durch Mutual-Exclusion geschützt werden sollen. Wie macht man das in Java?

   Versuchen Sie mittels von Mutual-Exclusion sicherzustellen, dass keine Abweichungen entstehen.

   - Reicht es die kritischen Bereiche in Account zu sichern?
   - Welches sind die kritischen Bereiche?

   Untersuchen Sie mehrere Varianten von Locks (Lock auf Methode oder Block, Lock auf Instanz oder Klasse).

   Ihre Implementierung muss noch nebenläufige Transaktionen erlauben, d.h. wenn Sie zu stark synchronisieren, werden alle Transaktionen in Serie ausgeführt und Threads ergeben keinen Sinn mehr.

   Stellen Sie für sich folgende Fragen:

   - Welches ist das Monitor-Objekt?
   - Braucht es eventuell das Lock von mehr als einem Monitor, damit eine Transaktion ungestört ablaufen kann?

c. Wenn Sie es geschafft haben die Transaktion thread-safe zu implementieren, ersetzen Sie in `AccountTransferSimulation` die folgende Zeile:
   `AccountTransferTask task1 = new AccountTransferTask(account3, account1, 2);`
   durch
   `AccountTransferTask task1 = new AccountTransferTask(account1, account3, 2);`
   und starten Sie das Programm noch einmal. Was stellen Sie fest? (evtl. müssen Sie mehrfach neu starten, damit der Effekt auftritt).
   Was könnte die Ursache sein und wie können Sie es beheben?

Falls Sie die Frage noch nicht beantworten können, kommen sie nach der Vorlesung "Concurrency 4" nochmals auf diese Aufgabe zurück und versuchen Sie sie dann zu lösen.

1.2. Traffic Light [PU]

In dieser Aufgabe sollen Sie die Funktionsweise einer Ampel und deren Nutzung nachahmen. Benutzen Sie hierzu die Vorgabe im Modul `TrafficLight`.

a. Ergänzen Sie zunächst in der Klasse `TrafficLight` drei Methoden:
   - Eine Methode zum Setzen der Ampel auf "rot".
   - Eine Methode zum Setzen der Ampel auf "grün".
   - Eine Methode mit dem Namen `passby()`. Diese Methode soll das Vorbeifahren eines Fahrzeugs an dieser Ampel nachbilden: Ist die Ampel rot, so wird der aufrufende Thread angehalten, und zwar so lange, bis die Ampel grün wird. Ist die Ampel dagegen grün, so kann der Thread sofort aus der Methode zurückkehren, ohne den Zustand der Ampel zu verändern. Verwenden Sie `wait`, `notify` und `notifyAll` nur an den unbedingt nötigen Stellen!

Die Zwischenphase „gelb“ spielt keine Rolle – Sie können diesem Zustand ignorieren!

b. Erweitern Sie nun die Klasse `Car` (abgeleitet von `Thread`).
Im Konstruktor wird eine Referenz auf ein Feld von Ampeln übergeben. Diese Referenz wird in einem entsprechenden Attribut der Klasse `Car` gespeichert. In der `run`-Methode werden alle Ampeln dieses Feldes passiert, und zwar in einer Endlosschleife (d.h. nach dem Passieren der letzten Ampel des Feldes wird wieder die erste Ampel im Feld passiert).
Natürlich darf das Auto erst dann eine Ampel passieren, wenn diese auf grün ist!
Für die Simulation der Zeitspanne für das Passieren des Signals sollten Sie folgende Anweisung verwenden: `sleep((int)(Math.random() * 500));`

Beantworten Sie entweder (c) oder (d) (nicht beide):

c. Falls Sie bei der Implementierung der Klasse TrafficLight die Methode `notifyAll()` benutzt haben: Hätten Sie statt `notifyAll` auch die Methode `notify` verwenden können, oder haben Sie `notifyAll()` unbedingt gebraucht? Begründen Sie Ihre Antwort!

d. Falls Sie bei der Implementierung der Klasse Ampel die Methode `notify()` benutzt haben: Begründen Sie, warum `notifyAll()` nicht unbedingt benötigt wird!

e. Testen Sie das Programm `TrafficLightOperation.java`. Die vorgegebene Klasse implementiert eine primitive Simulation von Autos, welche die Ampeln passieren. Studieren Sie den Code dieser Klasse und überprüfen Sie, ob die erzeugte Ausgabe sinnvoll ist.

1.3. Producer-Consumer Problem [PU]

In dieser Aufgabe wird ein Producer-Consumer Beispiel mittels einer Queue umgesetzt.

Dazu wird eine Implementation mittels eines Digitalen Ringspeichers umgesetzt.

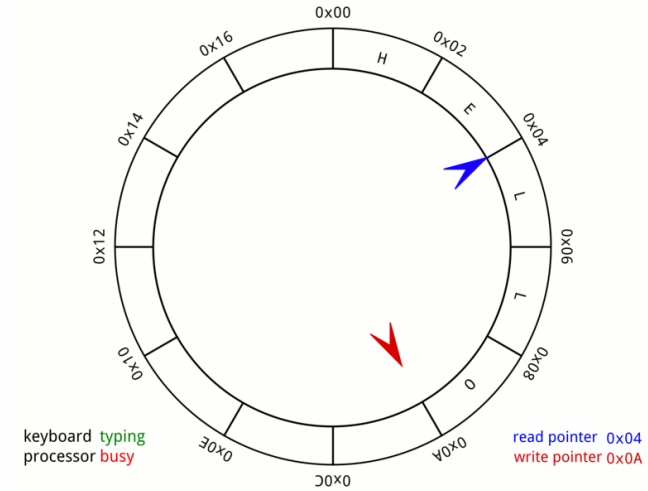

Figure 1. Circular Buffer [Wikipedia]

Hierzu sind zwei Klassen (`CircularBuffer.java`, `Buffer.java`) vorgegeben, mit folgendem Design:

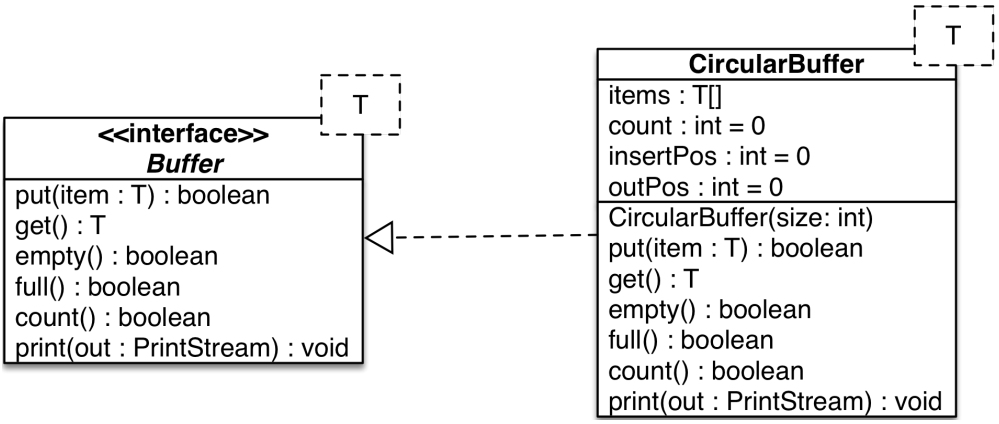

Figure 2. Circular Buffer Design

Analyse der abgegebenen CircularBuffer Umsetzung.

Mit dem Testprogramm `CircBufferSimulation` kann die Funktion der `CircularBuffer` Implementierung analysiert werden. Der mitgelieferte `Producer`-Thread generiert kontinuierlich Daten (in unserem Fall aufsteigende Nummern) und füllt diese mit `buffer.put(…)` in den Buffer. Der `Consumer` -Thread liest die Daten kontinuierlich mit `buffer.get()` aus dem Buffer aus. Beide Threads benötigen eine gewisse Zeit zum Produzieren bzw. Konsumieren der Daten. Diese kann über die Variablen `maxProduceTime` bzw. `maxConsumeTime` beeinflusst werden. Es können zudem mehrere Producer- bzw. Consumer-Threads erzeugt werden.

a. Starten Sie `CircularBufferSimulation` und analysieren Sie die Ausgabe. Der Status des Buffers (belegte Positionen und Füllstand) wird sekündlich ausgegeben. Alle fünf Sekunden wird zudem der aktuelle Inhalt des Buffers ausgegeben.
   - Wie ist das Verhalten des `CircularBuffer` bei den Standard-Testeinstellungen?

b. Analysieren Sie die Standard-Einstellungen in `CircularBufferSimulation`.
   - Welche Varianten gibt es, die Extrempositionen (Buffer leer, bzw. Buffer voll) zu erzeugen?
   - Was ist das erwartete Verhalten bei vollem bzw. leerem Buffer? (bei Producer bzw. Consumer)

c. Testen Sie was passiert, wenn der Buffer an die Kapazitätsgrenze kommt. Passen Sie dazu die Einstellungen in `CircularBufferSimulation` entsprechend an.

Belassen Sie die Anzahl Producer-Threads vorerst auf 1, damit der Inhalt des Buffers (Zahlenfolge) einfacher verifiziert werden kann.

   - Was Stellen Sie fest? Was passiert wenn der Buffer voll ist? Warum?

d. Wiederholen Sie das Gleiche für einen leeren Buffer. Passen Sie die Einstellungen so an, dass der Buffer sicher leer wird, d.h. der `Consumer` keine Daten vorfindet.

Geben Sie gegebenenfalls die gelesenen Werte des Konsumenten-Threads aus.

   - Was stellen Sie fest, wenn der Buffer leer ist? Warum?

Thread-Safe Circular Buffer

In der vorangehenden Übung griffen mehrere Threads (`Producer` & `Consumer`) auf den gleichen Buffer zu. Die Klasse `CircularBuffer` ist aber nicht thread-safe. Deshalb soll jetzt eine Wrapper Klasse geschrieben werden, welche die CircularBuffer-Klasse "thread-safe" macht. Das führt zu folgendem Design:

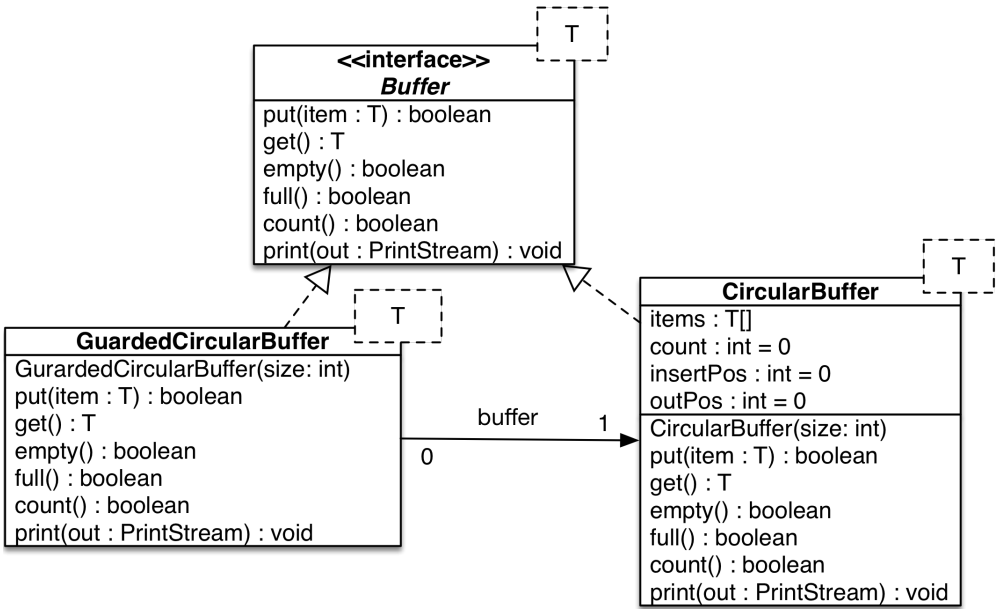

Figure 3. Guarded Circular Buffer Design

Beachten Sie, dass es sich hier um einen Wrapper (keine Vererbung) handelt. Der `GuardedCircularBuffer` hält eine Referenz auf ein `CircularBuffer`-Objekt welches er im Hintergrund für die Speicherung verwendet. Das heißt, viele Methodenaufrufe werden einfach an das gekapselte Objekt weitergeleitet. Einzelne Methoden werden jedoch in ihrer Funktion erweitert. Man spricht auch von "Dekorieren" des Original-Objektes (siehe Decorator-Pattern).

e. Ergänzen Sie die vorhandene Klasse `GuardedCircularBuffer` sodass:
   - Die Methoden `empty`, `full`, `count` das korrekte Resultat liefern.
   - Aufrufe von `put` blockieren, solange der Buffer voll ist, d.h. bis mindestens wieder ein leeres Buffer-Element vorhanden ist.
   - Analog dazu Aufrufe von `get` blockieren, solange der Buffer leer ist, d.h, bis mindestens ein Element im Buffer vorhanden ist.

Verwenden Sie den Java Monitor des `GuardedCircularBuffer`-Objektes! Wenn die Klasse fertig implementiert ist, soll sie in der `CircBufferSimulation` Klasse verwendet werden.

Beantworten Sie entweder (i) oder (ii) (nicht beide):

i. Falls Sie bei der Implementierung der Klasse `GuardedCircularBuffer` die Methode `notifyAll()` benutzt haben: Hätten Sie statt `notifyAll()` auch die Methode `notify()` verwenden können oder haben Sie `notifyAll()` unbedingt benötigt? Begründen Sie Ihre Antwort!

ii. Falls Sie bei der Implementierung der Klasse `GuardedCircularBuffer` die Methode `notify()` benutzt haben: Begründen Sie, warum Sie `notifyAll()` nicht unbedingt benötigten!

2. Concurrency 4 — Lock & Conditions, Deadlocks

2.1. Single-Lane Bridge [PU]

Die Brücke über einen Fluss ist so schmal, dass Fahrzeuge nicht kreuzen können. Sie soll jedoch von beiden Seiten überquert werden können. Es braucht somit eine Synchronisation, damit die Fahrzeuge nicht kollidieren. Um das Problem zu illustrieren wird eine fehlerhaft funktionierende Anwendung, in welcher keine Synchronisierung vorgenommen wird, zur Verfügung gestellt. Ihre Aufgabe ist es, die Synchronisation der Fahrzeuge einzubauen.

Die Anwendung finden Sie im Praktikumsverzeichnis im Modul `Bridge`. Gestartet wird sie indem die Klasse `Main` ausgeführt wird (z.B. mit `gradle run`). Das GUI sollte selbsterklärend sein. Mit den zwei Buttons können sie Autos links bzw. rechts hinzufügen. Sie werden feststellen, dass die Autos auf der Brücke kollidieren, bzw. illegalerweise durcheinander hindurchfahren.

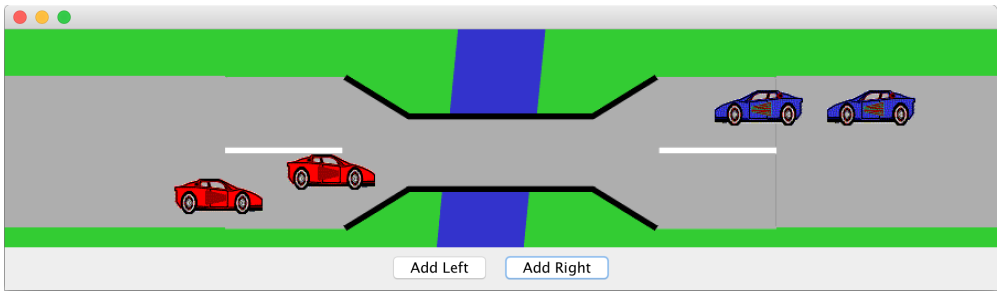

Figure 4. Single-Lane Bridge GUI

Um das Problem zu lösen, müssen Sie die den GUI Teil der Anwendung nicht verstehen. Sie müssen nur wissen, dass Fahrzeuge, die von links nach rechts fahren, die Methode `controller.enterLeft()` aufrufen bevor sie auf die Brücke fahren (um Erlaubnis fragen) und die Methode `controller.leaveRight()` aufrufen, sobald sie die Brücke verlassen. Fahrzeuge in die andere Richtung rufen entsprechend die Methoden `enterRight()` und `leaveLeft()` auf. Dabei ist `controller` eine Instanz der Klasse `TrafficController`, welche für die Synchronisation zuständig ist. In der mitgelieferten Klasse sind die vier Methoden nicht implementiert (Dummy-Methoden).

a. Erweitern sie `TrafficController` zu einer Monitorklasse, die sicherstellt, dass die Autos nicht mehr kollidieren. Verwenden Sie dazu den Lock und Conditions Mechanismus.

Verwenden Sie eine Statusvariable, um den Zustand der Brücke zu repräsentieren (e.g. `boolean bridgeOccupied`).

b. Passen Sie die Klasse `TrafficController` so an, dass jeweils abwechslungsweise ein Fahrzeug von links und rechts die Brücke passieren kann. Unter Umständen wird ein Auto blockiert, wenn auf der anderen Seite keines mehr wartet. Verwenden Sie für die Lösung mehrere Condition Objekte.

Um die Version aus a. zu behalten, können sie auch eine Kopie (z.B. `TrafficControllerB`) erzeugen und `Main` anpassen, damit eine Instanz dieser Klasse verwendet wird.

c. Bauen Sie die Klasse `TrafficController` so um, dass jeweils alle wartenden Fahrzeuge aus einer Richtung passieren können. Erst wenn keines mehr wartet, kann die Gegenrichtung fahren.

Mit `ReentrentLock.hasWaiters(Condition c)` können Sie abfragen ob Threads auf eine bestimmte Condition warten.

2.2. The Dining Philosophers [PA]

Beschreibung des Philosophen-Problems:

Fünf Philosophen sitzen an einem Tisch mit einer Schüssel, die immer genügend Spaghetti enthält. Ein Philosoph ist entweder am Denken oder am Essen. Um zu essen braucht er zwei Gabeln. Es hat aber nur fünf Gabeln. Ein Philosoph kann zum Essen nur die neben ihm liegenden Gabeln gebrauchen. Aus diesen Gründen muss ein Philosoph warten und hungern, solange einer seiner Nachbarn am Essen ist.

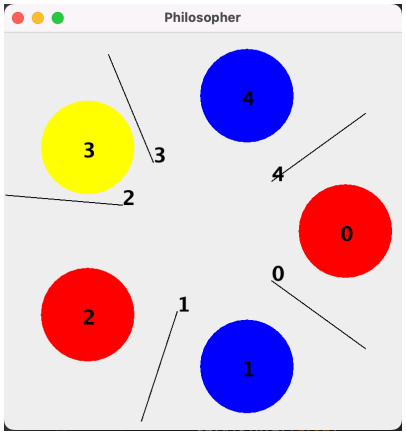

Figure 5. Philosopher UI

Das Bild zeigt die Ausgabe des Systems, das wir in dieser Aufgabe verwenden. Die blauen Kreise stellen denkende Philosophen dar, die gelben essende und die roten hungernde. Bitte beachten Sie, dass eine Gabel, die im Besitz eines Philosophen ist, zu dessen Teller hin verschoben dargestellt ist.

Die Anwendung besteht aus drei Dateien / Hauptklassen (jeweils mit zusätzlichen inneren Klassen):

`PhilosopherGui`

Ist das Hauptprogramm und repräsentiert das GUI (Java-Swing basiert). Die Klasse initialisiert die Umgebung `PhilosopherTable`, startet die Simulation und erzeugt die obige Ausgabe. Zudem werden Statusmeldungen der Philosophen auf der Konsole ausgegeben.

`PhilosopherTable`

Repräsentiert das Datenmodell. Sie initialisiert, startet und stoppt die Threads der Klasse `Philosopher`, welche das Verhalten und Zustände (THINKING, HUNGRY, EATING) der Philosophen abbildet und als innere Klasse umgesetzt ist.

`ForkManager`

Diese Klasse enthält die Strategie, wie die Philosophen die zwei Gabeln (Klasse `Fork`) aufnehmen (`aquireForks(int philosopherId)`) und zurücklegen (`releaseForks(int philosopherId)`).

a. Analysieren Sie die bestehende Lösung (vor allem `ForkManager`), die bekanntlich nicht Deadlock-frei ist. Kompilieren und starten Sie die Anwendung. Nach einiger Zeit geraten die Philosophen in eine Deadlock-Situation und verhungern. Überlegen Sie sich, wo im Code der Deadlock entsteht.

b. Passen Sie die bestehende Lösung so an, dass keine Deadlocks mehr möglich sind. Im Unterricht haben Sie mehrere Lösungsansätze kennengelernt.
In der umzusetzenden Lösung soll der `ForkManager` immer das Gabelpaar eines Philosophen in einer *atomaren* Operation belegen bzw. freigeben und nicht die einzelnen Gabeln sequentiell nacheinander. Dazu müssen beide Gabeln (links & rechts) auch verfügbar (`state == FREE`) sein, ansonsten muss man warten, bis beide verfügbar sind.
   - Es sind nur Anpassungen in der Datei `ForkManager.java` notwendig. Die `PhilosopherGui` und `PhilosopherTable`-Klassen müssen nicht angepasst werden.
   - Verändern Sie nicht das `public` interface des `ForkManager` – `acquireForks(int philosopherId)` und `releaseForks(int philosopherId)` müssen bestehen bleiben und verwendet werden.
   - Verwenden Sie für die Synchronisation Locks und Conditions!

Abschluss

Stellen Sie sicher, dass die Pflichtaufgaben mittels `gradle run` gestartet werden können, die Tests mit `gradle test` erfolgreich laufen und pushen Sie die Lösung vor der Deadline in Ihr Abgaberepository.