Zadanie zaliczeniowe ze współbieżności w języku Java

Transakcja to sekwencja operacji, które stanowią pewną całość i jako takie muszą być wykonane niepodzielnie i w izolacji. Niepodzielność oznacza, że albo wykonają się wszystkie operacje z sekwencji stanowiącej transakcję, albo nie wykona się żadna z nich. Izolacja natomiast sprawia, że wyniki poszczególnych operacji z sekwencji danej transakcji nie są widoczne dla operacji z innych transakcji dopóki cała sekwencja danej transakcji się nie zakończy. Zapewnianie niepodzielności i izolacji jest celem zarządcy transakcji. Twoim zadaniem będzie napisanie takiego współbieżnego zarządcy w języku Java zgodnie z poniższymi wymaganiami. Do rozwiązania zadania należy wykorzystać załączony szablon.

Specyfikacja

Rozważmy system, w którym działa pewna (nieznana) liczba wątków. Wątki operują na ustalonej kolekcji zasobów (obiekty klas dziedziczących po klasie `cp1.base.Resource` z załączonego szablonu), z których każdy ma stały, unikalny identyfikator (`cp1.base.ResourceId`). W wyniku takich operacji (reprezentowanych przez dowolne obiekty implementujące interfejs `cp1.base.ResourceOperation`) kolekcja zasobów się nie zmienia, za to może się zmieniać stan poszczególnych zasobów.

Operacje na zasobach wykonywane są w ramach transakcji. W dowolnym momencie czasu, dany wątek może wykonywać co najwyżej jedną transakcję. Transakcje są koordynowane przez menedżera transakcji (implementującego poniższy interfejs `cp1.base.TransactionManager`):

```
public interface TransactionManager {

    public void startTransaction(
    ) throws
        AnotherTransactionActiveException;

    public void operateOnResourceInCurrentTransaction(
            ResourceId rid,
            ResourceOperation operation
    ) throws
        NoActiveTransactionException,
        UnknownResourceIdException,
        ActiveTransactionAborted,
        ResourceOperationException,
        InterruptedException;

    public void commitCurrentTransaction(
    ) throws
        NoActiveTransactionException,
        ActiveTransactionAborted;

    public void rollbackCurrentTransaction();

    public boolean isTransactionActive();

    public boolean isTransactionAborted();
}
```

Podczas tworzenia (metoda `newTM` klasy `cp1.solution.TransactionManagerFactory`) menedżer transakcji otrzymuje kolekcję zasobów, które przechodzą pod jego wyłączną kontrolę, oraz interfejs do pytania o aktualny czas, który jest potrzebny przy zarządzaniu transakcjami.

Rozpoczynanie transakcji

Zanim wątek wykona jakąkolwiek operację na zasobie, musi rozpocząć (aktywować) transakcję poprzez wywołanie metody `startTransaction` menedżera transakcji (MT). Rozpoczęcie transakcji nie udaje się jedynie wtedy, gdy wątek wywołujący ww. metodę nie zakończył jeszcze poprzedniej transakcji. W takim przypadku wywołanie metody `startTransaction` MT kończy się zgłoszeniem wyjątku `cp1.base.AnotherTransactionActiveException`. W efekcie wątek może zakończyć bieżącą transakcję, a następnie ponowić nieudaną próbę rozpoczęcia nowej transakcji. Jeśli natomiast wątkowi uda się rozpocząć transakcję, to transakcja ta pozostaje *aktywna* do momentu, gdy nie zostanie zakończona w sposób opisany dalej.

Wykonywanie operacji w ramach transakcji

Wykonanie przez wątek operacji na zasobie odbywa się poprzez wywołanie przez niego metody `operateOnResourceInCurrentTransaction` MT, której argumenty oznaczają identyfikator zasobu oraz operację do wykonania. Jeśli wątek nie ma aktywnej transakcji, wywołanie ww. metody kończy się zgłoszeniem wyjątku `cp1.base.NoActiveTransactionException`. Analogicznie, wywołanie dla identyfikatora zasobu, który nie jest pod kontrolą MT, kończy się wyjątkiem `cp1.base.UnknownResourceIdException` z argumentem odpowiadającym identyfikatorowi niekontrolowanego zasobu. Wreszcie, jeśli aktywna transakcja została wcześniej anulowana, jak wyjaśniamy dalej, to wywołanie kończy się wyjątkiem `cp1.base.ActiveTransactionAborted`.

W przeciwnym razie wątek rozpoczyna ubieganie się o dostęp do wskazanego zasobu. Jeśli w ramach aktywnej transakcji wątek już taki dostęp uzykał poprzednio, to jest udało mu się wcześniej co najmniej raz rozpocząć wykonanie jakiejś operacji na tym samym zasobie, to dostęp jest mu przyznawany natychmiastowo. W przeciwnym przypadku sprawdzane jest, czy inna aktywna transakcja miała kiedykolwiek przyznany dostęp do zasobu. Jeśli taka transakcja nie istnieje, to dostęp zostaje przyznany rozważanej transakcji. W przeciwnym razie z kolei rozważana transakcja musi potencjalnie zaczekać na zakończenie transakcji, która uzyskała wcześniej dostęp do wskazanego zasobu.

Jednakże gdy takie oczekiwanie doprowadziłoby do zakleszczenia pewnej grupy transakcji, MT *anuluje* tę z transakcji w grupie, która została rozpoczęta najpóźniej. Czas rozpoczęcia mierzony jest wartością metody `getTime` interfejsu `LocalTimeProvider` (przekazanemu MT przy jego tworzeniu), a w przypadku, gdy kilka transakcji ma ten sam czas rozpoczęcia, decydują identyfikatory wykonujących je wątków (metoda `getId` klasy `Thread`), to jest transakcja wątku z największym identyfikatorem uważana jest za najpóźniejszą spośród transakcji z tym samym czasem rozpoczęcia. Anulowanie transakcji wymaga także przerwania wątku, który ją rozpoczął (metoda `interrupt` klasy `Thread`).

Anulowana transakcja nie może być kontynuowana, to jest można jedynie ją zakończyć, jak wyjaśniamy dalej. W szczególności, jak wspomniano wcześniej, próba wykonania jakiejkolwiek operacji w ramach tej transakcji kończy się wyjątkiem `cp1.base.ActiveTransactionAborted`.

Gdy wątek uzyska już dostęp do zasobu, MT rozpoczyna wykonywanie na tym zasobie operacji przekazanej jako argument funkcji `operateOnResourceInCurrentTransaction` poprzez wywołanie na obiekcie tej operacji metody `execute` interfejsu `cp1.base.ResourceOperation` z argumentem odpowiadającym zasobowi. Wykonanie operacji musi się odbyć w kontekście wątku, który tę operację zlecił, to jest żaden wątek nie może wykonać ww. metody `execute` na obiekcie operacji przekazanej MT przez inny wątek.

Wykonanie operacji może zakończyć się wyjątkiem `cp1.base.ResourceOperationException`, który musi zostać przekazany przez MT do wołającego wątku. W takim przypadku mówimy, że operacja kończy się niepowodzeniem. Metoda `execute` operacji kończącej się niepowodzeniem gwarantuje, że zasób pozostaje w nienaruszonym stanie, to jest stanie sprzed wykonania tej operacji.

Operacja, której wykonanie nie spowoduje wyjątku `cp1.base.ResourceOperationException` kończy się natomiast sukcesem. Taka operacja może zmodyfikować stan zasobu.

Niezależnie od tego, jak się zakończy, dana operacja może być przez wątek wykonywana wiele razy, na tych samych lub różnych zasobach. Co więcej, różne wykonania tej samej operacji mogą się różnie kończyć.

Należy także zauważyć, iż w trakcie oczekiwania na zasób lub wykonywania na nim operacji, wątek może zostać przerwany (poprzez wywołanie metody `interrupt` z klasy `Thread`). W takim przypadku wywołanie metody `operateOnResourceInCurrentTransaction` MT kończy się wyjątkiem `InterruptedException`, pozostawiając bez zmian zarówno stan zasobu, jak i transakcji, przy czym w zależności od momentu, w którym pojawiło się przerwanie, transakcja może dostać lub nie dostęp do zasobu. Dokładniej, jeśli przerwanie pojawiło się przed lub w trakcie przyznawania dostępu, to transakcja tego dostępu nie uzyskuje. W przeciwnym przypadku dostęp zostaje przyznany.

Kończenie transakcji

Wątek może *zakończyć* aktywną transakcję na jeden z dwóch sposobów:

- poprzez zatwierdzenie (wywołanie metody `commitCurrentTransaction` MT) albo
- poprzez cofnięcie (wywołanie metody `rollbackCurrentTransaction` MT).

Zatwierdzenie transakcji powoduje, że wszelkie zmiany zasobów MT wykonane przez operacje w ramach tej transakcji stają się widoczne dla innych transakcji. Zatwierdzenie transakcji przez wątek nie udaje się tylko w dwóch przypadkach. Po pierwsze, gdy wątek nie ma aktywnej transakcji, zgłaszany jest wyjątek `cp1.base.NoActiveTransactionException`. Po drugie, gdy aktywna transakcja wątku została wcześniej anulowana, zgłaszany jest wyjątek `cp1.base.ActiveTransactionAborted` a transakcja nie jest kończona.

Cofnięcie transakcji powoduje z kolei, że wszelkie zmiany zasobów MT wykonane przez operacje w ramach tej transakcji zostają wycofane, to jest stan tych zasobów jest taki sam, jak w momencie rozpoczynania transakcji. Cofnięcie transakcji przez wątek zawsze się udaje. W szczególności, gdy wątek nie ma aktywnej transakcji, jest to po prostu operacja pusta.

Wycofanie zmian wykonanych przez daną operację na danym zasobie odbywa się poprzez wywołanie na obiekcie tej operacji metody `undo` interfejsu `cp1.base.ResourceOperation` z argumentem odpowiadającym zasobowi. Jak w przypadku metody `execute`, wywołanie metody `undo` dla operacji musi odbyć się w kontekście wątku, który tę operację zlecił. W przeciwieństwie do metody `execute`, metoda `undo` nie zgłasza żadnych wyjątków. Wycofywane są jedynie operacje zakończone sukcesem i robione jest to w odwrotnej kolejności niż kolejność ich wykonania w ramach transakcji.

Zakończenie transakcji na którykolwiek z dwóch możliwych sposobów powoduje, że MT przestaje przechowywać jakiekolwiek informacje związane z tą transakcją.

Sprawdzanie stanu transakcji

MT udostępnia jeszcze dwie metody pozwalające wątkowi sprawdzić stan swojej transakcji. Metoda `isTransactionActive`, wywołana w kontekście wątku, zwraca `true` wtedy i tylko wtedy, gdy wątek ten ma aktywną transakcję. Metoda `isTransactionAborted` z kolei zwraca `true` wtedy i tylko wtedy, gdy wątek ma aktywną transakcję, lecz została ona anulowana.

Wymagania

Twoim zadaniem jest zaimplementowanie MT według powyższej specyfikacji i dostarczonego szablonu przy wykorzystaniu mechanizmów współbieżności języka Java 11. Musisz zadbać o zapewnienie żywotności i bezpieczeństwa rozwiązania. Staraj się także zmaksymalizować równoległość. W szczególności operacje na różnych zasobach zlecane przez różne wątki powinny wykonywać się równolegle (oczwiście respektując ograniczenia wynikające z powyższej specyfikacji). Twój kod źródłowy powinien być napisany w zgodzie z dobrymi praktykami programistycznymi

Szczegółowe dodatkowe wymagania formalne są następujące.

1. Rozwiązanie musi być napisane samodzielnie.

2. Nie możesz w żaden sposób zmieniać zawartości pakietów `cp1.base` oraz `cp1.demo`.
3. Klasy implementujące rozwiązanie możesz dodawać jedynie w pakiecie `cp1.solution`, ale nie możesz tworzyć w tym pakiecie żadnych podpakietów.
4. W klasie `cp1.solution.TransactionManagerFactory` musisz dodać treść metody `newTM`, która będzie wykorzystywana do instancjonowania zaimplementowanego przez Ciebie MT. Każde wywołanie tej metody powinno tworzyć nowy obiekt MT. Wiele obiektów MT powinno być w stanie działać w tym samym czasie. Nie wolno natomiast w żaden sposób zmieniać sygnatury tej metody ani nazwy klasy czy jej lokalizacji.
5. Możesz stworzyć sobie własne pakiety do testów, np. `cp1.tests`, ale te pakiety będą ignorowane przy testowaniu przez nas, więc w szczególności kod Twojego MT nie może od nich zależeć.
6. W plikach źródłowych Javy nie możesz używać nieanglojęzycznych znaków (w szczególności polskich znaków).
7. Twoje rozwiązanie powinno składać się z jednego pliku `ab123456.zip`, gdzie `ab123456` należy zastąpić swoim loginem z maszyny students (będącym zwykle konkatenacją inicjałów i numeru indeksu). Plik ten musi mieć taką samą strukturę, jak szablon, to jest musi zawierać jedynie katalog `cp1` reprezentujący pakiet o tej samej nazwie, który zawiera katalogi odpowiednich podpakietów, co najmniej `base`, `demo` i `solution`, które z kolei zawierają odpowiednie pliki źródłowe (`*.java`).
8. Twoje rozwiązanie musi kompilować się na maszynie `students.mimuw.edu.pl` poleceniem `javac cp1/base/*.java cp1/solution/*.java cp1/demo/*.java`.
9. W Twoim rozwiązaniu musi działać program demonstracyjny, wywoływany poleceniem `java cp1.demo.Transactions`, to jest nie może on zgłaszać żadnych wyjątków.

Rozwiązania niespełniające któregokolwiek z powyższych wymagań nie będą sprawdzane i automatycznie dostaną 0 punktów.

Prosimy o zrozumienie! Będziemy mieli do sprawdzenia nawet blisko 170 rozwiązań. Rozwiązania te będą w pierwszej fazie testowane automatycznie. Gdybyśmy musieli każde rozwiązanie w jakikolwiek sposób poprawiać, aby uruchomienie testów było możliwe, stracilibyśmy niepotrzebnie mnóstwo czasu. Dlatego też zapewnienie zgodności z powyższymi wymaganiami jest po Państwa stronie.

Aby w tym celu dać Państwu więcej informacji co do samej procedury testowania, to dla każdego rozwiązania przebiegać będzie ona z grubsza następująco.

1. Archiwum `ZIP` z rozwiązaniem zostanie rozpakowane do dedykowanego katalogu głównego na maszynie `students` (lub kompatybilnej jeśli chodzi o wersję Javy).
2. Z katalogu tego zostaną usunięte wszystkie pliki i podkatalogi za wyjątkiem podkatalogu `cp1/solution` i jego zawartości.
3. Z podkatalogu `cp1/solution` zostaną usunięte wszystkie pliki (i podkatalogi) za wyjątkiem plików `*.java`.
4. Do katalogu głównego zostaną skopiowane katalogi `cp1/base` oraz `cp1/demo` z dostarczanego szablonu, aby pliki z interfejsami oraz aplikacja demonstracyjna były w wersji oryginalnej, oraz katalogi z kodem naszych testów.
5. Wszystko zostanie skompilowane.
6. Uruchomiona zostanie aplikacja demonstracyjna, aby sprawdzić, czy działa.
7. Jeśli rozpakowanie archiwum, kompilacja lub uruchomienie aplikacji demonstracyjnej się nie powiedzie, to rozwiązanie otrzymuje automatycznie 0 punktów. W przeciwnym przypadku, w ramach testowania, uruchamiane będą kolejne aplikacje testowe oraz ewentualne dodatkowe programy (np. weryfikacja anty-plagiatowa).

Wszelkie pytania i uwagi powinny być kierowane do Konrada Iwanickiego poprzez forum Moodle dedykowane zadaniu. Przed wysłaniem pytania, proszę sprawdzić na forum, czy ktoś wcześniej nie zadał podobnego.

Powodzenia!