Hochschule für Angewandte Wissenschaften Hamburg
University of Applied Sciences Hamburg
Fakultät Technik und Informatik
Department Informatik

Aufgabe A2 Busy smurfs taking buses

Aufgaben-Idee:

Modellieren Sie Bus-fahrende Schlümpfe sowie die zugehörigen (Linien-)Busse als Threads.
Es sollen **n** Schlümpfe (Klassenname: **`Smurf`**), **2** Busse (Klassenname: **`Bus`**) und **5** Haltestellen unterstützt werden. (Bemerkung: **n** ∈ [1,..,1000])
Ganz wie Sie es kennen fahren die beiden Linienbusse die 5 Haltestellen ihrer Strecke/Buslinie ab und zwar wechselseitig von Endstation zu Endstation (einzige Besonderheit ist, dass die Busfahrer keine Pause benötigen). An einer Bushaltestelle können mehrere Busse halten (also "beliebig" viele und insbesondere zwei ;-). Der Zeitpunkt des Anhaltens des Busses wird durch den Aufruf der Methode **stopAt(...)** definiert. Der Bus verweilt dann eine gewisse (Halte-)Zeit an der Haltestelle modelliert durch **takeTimeForStopoverAt(...)**. Nach Ablauf dieser Haltezeit fährt der Bus ("sofort") zu seiner nächsten Haltestelle. Der Zeitpunkt des Abfahrens von der Haltestelle definiert der Aufruf der Methode **startFrom(...)**. Der Bus fährt dann zur nächsten Haltestelle. Hierfür wird eine gewisse Zeit benötigt, die durch **takeTimeForBusRideTo(...)** modelliert wird. Am "Anfang" fährt der Bus seine "Start-Haltestelle" an.

Die Haltestellen sind der Einfachheit halber von 0 aus beginnend aufsteigend durchnummeriert – also 0, 1, 2, 3 und 4. Während der eine Bus mit dem Anfahren von Haltestelle 0 beginnt, fährt der andere Bus als erstes die Haltestelle 4 an. Die Busse starten synchron! Danach kommen die Busse jeweils entsprechend Ihres Vorankommens voran (bestimmt durch die "Umstände" bzw. **takeTimeForBusRideTo(...)** und **takeTimeForStopoverAt(...)**).

Also:
1.Bus: →**0**→**1**→**2**→**3**→**4**→3→2→1→0→1→2→3→4→3→2→1→0→1→...
2.Bus (entgegengesetzte Richtung): →**4**→**3**→**2**→**1**→**0**→1→2→3→4→3→2→1→0→1→2→3→4→3→...

Die Schlümpfe (**`Smurf`**) haben einen **`Schedule`**, der ihnen vorgibt, welche Haltestellen sie wann aufsuchen müssen und wie lange sie dann dort (*bzw. an dem der jeweiligen Haltestelle zugeordneten Ort/Umgebung/Hinterland*) verweilen.
Der erste Schedule-Eintrag (**`SSI`**) im Schedule bestimmt u.a. die Start-Position/Haltestelle des Schlumpfes. An dieser Position verweilt der Schlumpf die im Schedule-Eintrag definierte Zeit. Das Verstreichen dieser Zeit wird durch **takeTimeForDoingStuffAtCurrentPosition(...)** modelliert. Danach sucht der Schlumpf die nächste im Schedule gegebene Station/Position/Haltestelle auf. Sobald die Verweilzeit an der letzten Station verstrichen ist, tut der Schlumpf seine letzte Tat **lastDeed(...)** und verschwindet danach (d.h.: Der zugehörige Thread soll terminieren).
Um von einer Position/Haltestelle zu einer anderen Position/Haltestelle zu kommen, muss der Schlumpf die Busse nutzen. Er kann/darf nur in solche Busse einsteigen, die gerade an "seiner" Haltestelle halten und in die gewünschte Richtung fahren. Der Zeitpunkt des Betretens eines Busses wird durch den Aufruf der Methode **enter(...)** definiert.
Ebenso kann und darf ein Schlumpf einen Bus (in dem er sich befindet) nur verlassen, während der jeweilige Bus an der (jeweiligen Ziel-)Haltestelle hält. Sobald dies der Fall ist, muss der Schlumpf aber auch den jeweiligen Bus verlassen. (Es sind also keine unnötigen Spazierfahrten erlaubt ;-). Der Zeitpunkt des Verlassens des jeweiligen Busses definiert der jeweilige Schlumpf durch den Aufruf der Methode **leave(...)**.
Noch einmal ganz deutlich: Verboten sind unnötiges Ein- und Aussteigen sowie unnötiges Warten oder ("Spazieren"-)Fahren mit den Bussen (*da die Schlümpfe unter hohen Zeitdruck stehen*). Warten als Konsequenz der Randbedingungen (z.B. "Bus voll" oder "Bus fährt in falsche" Richtung) ist natürlich erlaubt/erforderlich. Wenn Sie Unsicherheiten bezüglich der Aufgabenstellung haben, dann beobachten Sie sich selbst beim (sinnvollen und Ziel-gerichteten) Busfahren – die Schlümpfe sollen sich genau so verhalten ;-)

Aufgabe

Lösen Sie die Aufgabe in einem geeigneten Package z.B. solution (oder auch a2.solution). Greifen Sie nun von dort aus auf die Klassen/Interfaces im Sub-Package _untouchable_.busPart4 zu, aber lassen Sie die dort befindlichen Klassen/Interfaces in diesem Package. Sie dürfen die Klassen/Interfaces weder verändern noch in "Ihr" Package kopieren.

Sie haben die folgenden Auflagen für Ihre Lösung. Sie müssen die Klassen:

- `Smurf_A` mit einer von Ihnen zu implementierenden Klasse `Smurf` unterstützen, die das Verhalten eines Schlumpfes bestimmt,
- `Bus_A` mit einer von Ihnen zu implementierenden Klasse `Bus` unterstützen, die das Verhalten des Busses bestimmt und
- `TestAndEnvironment_A` mit einer von Ihnen zu implementierenden Klasse `TestFrame` unterstützen, die Ihren Test beinhaltet.

Es steht Ihnen frei, weitere Klasse zu implementieren.

Es folgt eine Kurzübersicht zu den Klassen/Interfaces – **die Details entnehmen Sie bitte der Dokumentation der jeweiligen Klassen und Attribute**. Diese sind u.a. auch als Tooltip unter Eclipse abrufbar.

Für **Busse** (`Bus`) gilt bzw. muss gelten:

Ihre Klasse `Bus` ist abgeleitet von der vorgegebenen abstrakten Klasse `Bus_A`.
Die `Bus`-Klasse muss eine `terminate()`-Methode unterstützen. Innerhalb der `doTest()`-Methode Ihrer Klasse `TestFrame` muss, sobald alle Schlümpfe ihren jeweiligen Schedule beendet haben und terminiert sind, `terminate()` für jeden Bus aufgerufen werden.
Die von Ihnen zu implementierende Methode `identify()` liefert eine eindeutige ID, die den Bus identifiziert.
Die Methode `takeTimeForBusRideTo(...)` modelliert die Zeit, die der Bus braucht um das jeweilige Ziel zu erreichen und `takeTimeForStopoverAt(...)` modelliert die Zeit, die der Bus an der jeweiligen Haltestelle verweilt.
Mit dem Aufruf der Methode `stopAt(...)` definieren Sie den genauen Zeitpunkt zu dem der Bus "anhält und die Türen öffnet" und mit dem Aufruf der Methode `startFrom(...)` definieren Sie den genauen Zeitpunkt zu dem der Bus "die Türen schließt und los fährt". (Bei Schlumpf-Bussen sind dies wirklich Zeitpunkte ;-)
Wenn die `terminate()`-Methode aufgerufen wird, dann muss der Bus (im sinnvollen Rahmen) so schnell wie möglich "seine letzte Tat tun" `lastDeed(...)` und daraufhin terminieren. Achtung! Der Aufruf der Methode `lastDeed(...)` muss wirklich die letzte Aktion des jeweiligen Busses/Threads sein.
Jeder einzelne Bus muss ein eigenständiger Thread sein. Insbesondere die Methoden

- `takeTimeForBusRideTo(...)`
- `takeTimeForStopoverAt(...)`
- `stopAt(...)`
- `startFrom(...)`
- `lastDeed(...)`

müssen jeweils vom jeweiligen Bus/Thread selbst ausgeführt werden.
In einem Bus dürfen sich zu keinem Zeitpunkt mehr als `maximumNumberOfSmurfsPerBus` (im weiteren kurz `mspb`) "Fahrgast"-Schlümpfe befinden. (Es gilt: 1≤mspb≤nos; Default ist **`mspb`**=17; der Busfahrer zählt nicht mit).
Über `getDebugState()` können Kontrollausgaben ein- und ausgeschaltet werden.

Die Linienbusse fahren die eingangs beschriebenen Linien ab.
1.Bus: →**0**→**1**→**2**→**3**→**4**→3→2→1→0→1→2→3→4→3→2→1→0→1→…
2.Bus (entgegengesetzte Richtung): →**4**→**3**→**2**→**1**→**0**→1→2→3→4→3→2→1→0→1→2→3→4→3→…

Für **Schlümpfe** (`Smurf`) gilt bzw. muss gelten:

Ihre Klasse `Smurf` ist abgeleitet von der vorgegebenen abstrakten Klasse `Smurf_A`.
Wie in der Vorlesung besprochen und im Labor geübt, ist vor dem jeweiligen Generieren eines Schlumpfs `waitUntilNextArrival()` aufzurufen.
Die von Ihnen zu implementierende Methode `identify()` liefert eine eindeutige ID, die den Schlumpf identifiziert.
Seinen jeweiligen Schedule bekommt der Schlumpf automatisch (konkret erbt er ihn von `Smurf_A`).
Der Schedule ist ein Iterator. Über den Schedule erfährt der Schlumpf, wie lange er wo zu bleiben hat – siehe auch das Code-Grundgerüst am Ende.
Die Methode `enter(...)` muss genau zu dem Zeitpunkt aufgerufen werden, zu dem der Schlumpf den jeweiligen Bus betritt und die Methode `leave(...)` genau zu dem Zeitpunkt, zu dem der Schlumpf den jeweiligen Bus verlässt. Nachdem der Schlumpf den jeweiligen Bus betreten hat (`enter(...)`) und bevor er den jeweiligen Bus wieder verlässt (`leave(...)`) zeigt der Schlumpf sein Bus-Ticket vor und ist einfach im Bus. Dies wird durch den Aufruf von `beThere(...)` modelliert.
Schlümpfe haben es eilig und steigen nur in Busse ein, die in die richtige Richtung fahren.
In einem Bus dürfen sich zu keinem Zeitpunkt mehr als `maximumNumberOfSmurfsPerBus` (im weiteren kurz `mspb`) "Fahrgast"-Schlümpfe befinden. (Es gilt: 1≤mspb≤nos; Default ist **`mspb`**=17; der Busfahrer zählt nicht mit).
Jeder einzelne Schlumpf muss ein eigenständiger Thread sein. Insbesondere die Methoden

- `beThere(...)`
- `takeTimeForDoingStuffAtCurrentPosition(...)`
- `lastDeed(...)`

müssen jeweils vom jeweiligen Schlumpf/Thread selbst ausgeführt werden.
Über `getDebugState()` können Kontrollausgaben ein- und ausgeschaltet werden.

Im **Schedule** (`Schedule`) sind die Schedule-Einträgen (`SSI`) organisiert. Der Schedule ist ein `Iterator<SSI>`. Es werden `hasNext()` und `next()` unterstützt, aber nicht remove().
Jeder Schedule-Eintrag (`SSI`) enthält die zugehörige Position. Mit `getPlanedPostion()` kann die jeweilige Position/Haltestelle erfragt werden.
Die Methode `takeTimeForDoingStuffAtCurrentPosition(...)` greift "selbstständig" auf die (im Schedule-Eintrag gespeicherte) Verweilzeit an der Position zu und modelliert die Zeit, die der Schlumpf am jeweiligen Ort verbringt. Es reicht der Methode den aktuellen Schedule-Eintrag zu übergeben.

Die Klasse **`TestFrame`** muss der „Rahmen für alles“ sein und ist abzuleiten von der vorgegebenen abstrakten Klasse **`TestAndEnvironment_A`**.
Die von Ihnen zu implementierenden Methode **getAuthor()** sollen Ihre Namen bzw. Team-Nr abliefern.
Die von Ihnen zu implementierenden Methode **getWantedNumberOfSmurfs()** soll Ihren Wunsch für die Anzahl der Schlümpfe in jeweiligen Testlauf abliefern.
Die von Ihnen zu implementierenden Methode **getWantedNumberOfBuses()** soll Ihren Wunsch für die Anzahl der Busse in jeweiligen Testlauf abliefern. Der Rückgabewert muss 2 sein.
Die von Ihnen zu implementierenden Methode **getWantedNumberOfBusStops()** soll Ihren Wunsch für die Anzahl der Bushaltestellen in jeweiligen Testlauf abliefern. Der Rückgabewert muss 5 sein.
Die von Ihnen zu implementierenden Methode **getWantedMaximumNumberOfSmurfsPerBus()** soll Ihren Wunsch für die Obergrenze der Anzahl Schlümpfe in einem Bus abliefern.
Die von Ihnen zu implementierenden Methode **getWantedMaximumNumberOfBusesPerBusStop()** soll Ihren Wunsch für die Obergrenze der Anzahl Busse an einer Bushaltestelle.
Die von Ihnen zu implementierenden Methode **doTest(...)** soll Ihren Test durchführen, der die korrekte Funktion Ihrer Implementierung belegt. **"Alles dafür Nötige" wird innerhalb dieser Methode gemacht.** Als Parameter hat diese Methode die Anzahl der Schlümpfe, die Anzahl der Busse, die Anzahl der Bushaltestellen, die Obergrenze für Schlümpfe pro Bus und die Obergrenze für Busse pro Haltestelle die **tatsächlich** im Rahmen des Tests auftreten.

Die vorgegebene Methode **letThereBeLife()** ist von Ihnen zum Anstarten Ihres Tests zu verwenden. Diese Methode **letThereBeLife()** nutzt das von Ihnen zu implementierende **doTest(...)**. Der interne Aufbau von **letThereBeLife()** kann sich wie folgt vorgestellt werden:

```
public final void letThereBeLife() {
    mach interne Dinge
    mach Ausgaben
    doTest(
        getNumberOfWantedSmurfs(),
        getWantedNumberOfBuses(),
        getWantedNumberOfBusStops(),
        getWantedMaximumNumberOfSmurfsPerBus(),
        getWantedMaximumNumberOfBusesPerBusStop()
     );
    mach Ausgaben
}
```

Ferner darf kein Thread zu irgendeinem Zeitpunkt aktiv warten. Die Problemstellung ist "sauber" mit Synchronisation zu lösen. Ebenso ist eine (vorgetäuschte) "Synchronisation" über Race-Conditions unzulässig. Z.B. der Einsatz von `sleep`-Varianten oder `Thread.yield()` ist **nicht** erlaubt. Lediglich die indirekte Nutzung über die im _untouchable_-Package vorgegeben Methoden für die geforderten Aufgaben ist gestattet.

Idee eines "(Code-)Grundgerüsts":

```
package solution;
import _untouchable_.busPart4.*;
…
public class TestFrame … TestAndEnvironment_A {
    …
    @Override public String getAuthor() { … }
    @Override public int getWantedNumberOfSmurfs() { … }                                    // <- 1000
    @Override public int getWantedNumberOfBuses() { return 2; }
    @Override public int getWantedNumberOfBusStops() { return 5; }
    @Override public int getWantedMaximumNumberOfSmurfsPerBus() { … }                       // <- 17
    @Override public int getWantedMaximumNumberOfBusesPerBusStop() { return Integer.MAX_VALUE; }

    @Override public void doTest(
        Integer requestedNumberOfSmurfs,
        Integer requestedNumberOfBuses,
        Integer requestedNumberOfBusStops,
        Integer requestedMaximumNumberOfSmurfsPerBus,
        Integer requestedMaximumNumberOfBusesPerBusStop
    ){
        …
        Loop-solange noch Schlümpfe erzeugt werden müssen {
            …
            erzeuge und starte Schlumpf
            …
            Smurf.waitUntilNextArrival();
            …
        }
        …
    }

    …

    public static void main( String[] unused ){
        …
        TestFrame tfo = new TestFrame();
        …
        tfo.letThereBeLife();
        …
    }
}
```

Im Rahmen seines "Tuns" muss der `TestFrame` innerhalb von `doTest(…)` zu den richtigen Momenten `waitUntilNextArrival(…)` "machen"/aufrufen oder für die entsprechenden Bus-Threads `terminate()` aufrufen.
Es gibt zulässig Alternativen: Z.B. eine eigene Test-Klasse, die sich von `TestAndEnvironment_A` ableitet und im `TestFrame` instanziiert wird.

```
package solution;
import _untouchable_.busPart4.*;
…
public class Bus … Bus_A … {
    …
    @Override public int identify() { … }
    @Override public void terminate() { … }
    @Override public boolean getDebugState(){ … }
    …
}
```

Im Rahmen seines "Tuns" muss der `Bus` zu den richtigen Momenten `lastDeed()`, `startFrom(…)`, `stopAt(…)`, `takeTimeForBusRideTo(…)` oder `takeTimeForStopoverAt(…)` "machen"/aufrufen.
`lastDeed()` ist die letzte "Aktion" eines `Bus`es.

```
package solution;
import _untouchable_.busPart4.*;
…
public class Smurf … Smurf_A … {
    …
    @Override public int identify() { … }
    @Override public boolean getDebugState(){ … }

    @Override
    public void run(){
        …
        while ( schedule.hasNext() ){                          // Attribut schedule wurde von Smurf_A geerbt
            …
            ssi = schedule.next();                             // ssi ist vom importierten Typ SSI
            int woGibtEsWasFürMichZuTun = ssi.getPlanedPosition();
            Falls nicht bereits "dort", begib Dich (per Bus) so schnell wie möglich "dort" hin.
            Achtung: Ganz am Anfang befindet sich der Schlumpf bereits an der jeweils geforderten Position
            und am Ende bleibt der Schlumpf am Ort des letzten Jobs. (Niemand wartet "zu Hause" auf den Schlumpf :'-(.
            …
            takeTimeForDoingStuffAtCurrentPosition( aktuelle-Position, ssi );      // Metapher für: Mach "dort" Deinen Job
            …
        }
        …
        lastDeed();
    }
    …
}
```

Im Rahmen seines "Tuns" muss der Schlumpf zu den richtigen Momenten `beThere(…)`, `enter(…)`, `lastDeed()`, `leave(…)` oder `takeTimeForDoingStuffAtCurrentPosition(…)` "machen"/aufrufen.
`lastDeed()` ist die letzte "Aktion" eines Schlumpfes.