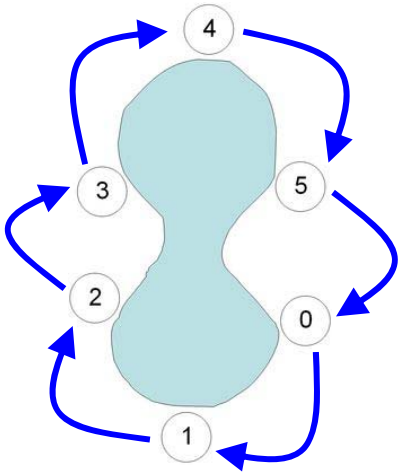
Aufgabe A3 Busy smurfs taking ships

Aufgaben-Idee:

Modellieren Sie Schiff-fahrende Schlümpfe sowie die zugehörigen Schiffe als Threads.
Es sollen **n** Schlümpfe (Klassenname: **`Smurf`**), **6** Schiffe (Klassenname: **`Ship`**) und **6** Anlegestellen (Klassenname: **`Landing`**) unterstützt werden. (Bemerkung: **n** ∈ [1,..,2500])
Die Anlegestellen sind der Einfachheit halber von 0 aus beginnend im Uhrzeigersinn aufsteigend durchnummeriert – also 0, 1, 2, 3, 4 und 5. Die Schiffe fahren zyklisch die Anlegestellen ab. Jedes der 6 Schiffe startet an einer der 6 Anlegestellen. Alle Schiffe, die an einer Anlegestelle mit gerader Nummer starten, fahren die Anlegestellen im Uhrzeigersinn ab und alle, die an einer Anlegestelle mit ungerader Nummer starten, fahren die Anlegestellen gegen den Uhrzeigersinn.
Das erste Schiff startet an der Anlegestelle 0 und fährt die Anlegestellen im Uhrzeigersinn ab (0→1→2→3→4→5→0→1→**...**).



Das zweite Schiff startet an der Anlegestelle 1 und fährt die Anlegestellen gegen den Uhrzeigersinn ab (1→0→5→4→3→2→1→0→**...**).
Das dritte Schiff startet an der Anlegestelle 2 und fährt die Anlegestellen im Uhrzeigersinn ab (2→3→4→5→0→1→2→3→**...**).
Das vierte Schiff startet an der Anlegestelle 3 und fährt die Anlegestellen gegen den Uhrzeigersinn ab (3→2→1→0→5→4→3→2→**...**).
Das fünfte Schiff startet an der Anlegestelle 4 und fährt die Anlegestellen im Uhrzeigersinn ab (4→5→0→1→2→3→4→5→**...**).
Das sechste Schiff startet an der Anlegestelle 5 und fährt die Anlegestellen gegen den Uhrzeigersinn ab (5→4→3→2→1→0→5→4→**...**).

An einer Anlegestelle können maximal zwei Schiffe gleichzeitig anlegen. Dabei ist egal um welche Schiffe es handelt. Es dürfen also auch zwei Schiffe anlegen, die in die gleiche Richtung fahren. Insbesondere darf keinem Schiff aufgrund seiner Richtung die Einfahrt verwehrt werden.
Der Zeitpunkt des Anlegens eines Schiffes wird durch den Aufruf der Methode **dockAt(...)** definiert. Nach dem Anlegen an der Anlegestelle beginnt das Schiff "sofort" für eine gewisse Halte-/Liege-Zeit an der Anlegestelle zu verweilen modelliert durch **takeTimeForBoardingAt(...)**. Nach Ablauf dieser Liegezeit legt das Schiff "sofort" ab und bricht zur nächsten Anlegestelle auf. Der Zeitpunkt des Ablegens definiert der Aufruf der Methode **castOff(...)**. Das Schiff fährt dann zur nächsten Anlegestelle. Hierfür wird eine gewisse Zeit benötigt, die durch **takeTimeForSailingTo(...)** modelliert wird.

Am "Anfang" legt das Schiff an seine "Start-Anlegestelle" an. Das obige "sofort" kennzeichnet, dass die entsprechenden Dinge sofort passieren sollen. Insbesondere soll nicht direkt auf Schlümpfe gewartet werden. Das gilt ganz besonders für das Ablegen nach Ablauf der Liege-/Haltezeit takeTimeForBoardingAt(...). Daher darf das Schiff dann auch **nicht** erst noch einmal zum Einsteigen auffordern. Gewisse Synchronisationsaktivitäten sind vermutlich unumgänglich, aber der Schiff darf nicht unmittelbar durch die Schlümpfe aufgehalten werden.

Die Schlümpfe (`Smurf`) haben einen `Schedule`, der ihnen vorgibt, welche Anlegestelle sie wann aufsuchen müssen und wie lange sie dann dort (*bzw. an dem der jeweiligen Anlegestelle zugeordneten Ort/Umgebung/Hinterland*) verweilen.
Der erste Schedule-Eintrag (`SSI`) im Schedule bestimmt u.a. die Start-Position/Anlegestelle 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/Anlegestelle 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/Anlegestelle zu einer anderen Position/Anlegestelle zu kommen, muss der Schlumpf die Schiffe nutzen. Er kann/darf nur in solche Schiffe einsteigen, die keine unnötigen Strecken fahren. Beispielsweise nimmt ein Schlumpf, der von Anlegestelle 1 nach Anlegestelle 5 will, nur ein Schiff, das gegen den Uhrzeigersinn verkehrt. Hingegen nimmt ein Schlumpf, der von der Anlegestelle 1 nach Anlegestelle 4 will, jedes Schiff bzw. das erste, das er bekommen kann.
Der Zeitpunkt des Betretens eines Schiffes wird durch den Aufruf der Methode enter(...) definiert.
Ebenso kann und darf ein Schlumpf ein Schiff (in dem er sich befindet) nur verlassen, während das jeweilige Schiff an der (jeweiligen Ziel-)Anlege-Stelle liegt. Sobald dies der Fall ist, muss der Schlumpf aber auch das jeweilige Schiff 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 Schiffen (*da die Schlümpfe unter hohen Zeitdruck stehen*). Warten als Konsequenz der Randbedingungen (z.B. "Schiff voll" oder "Schiff fährt in falsche" Richtung) ist natürlich erlaubt/erforderlich.

Aufgabe

Lösen Sie die Aufgabe in einem geeigneten Package z.B. solution (oder auch a3.solution). Greifen Sie nun von dort aus auf die Klassen/Interfaces im Sub-Package _untouchable_.shipPart5 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,
- `Ship_A` mit einer von Ihnen zu implementierenden Klasse `Ship` unterstützen, die das Verhalten des Schiffes 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 **Schiffe** (`Ship`) gilt bzw. muss gelten:

Ihre Klasse `Ship` ist abgeleitet von der vorgegebenen abstrakten Klasse `Ship_A`.
Die `Ship`-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 jedes Schiff aufgerufen werden. (Alle Threads, sie Sie gestartet haben, sind zu terminieren).
Die von Ihnen zu implementierende Methode `identify()` liefert eine eindeutige ID, die das Schiff identifiziert.
Die Methode `takeTimeForSailingTo(...)` modelliert die Zeit, die das Schiff braucht um das jeweilige Ziel zu erreichen und `takeTimeForBoardingAt(...)` modelliert die Zeit, die das Schiff an der jeweiligen Anlegestelle verweilt.
Mit dem Aufruf der Methode `dockAt(...)` definieren Sie den genauen Zeitpunkt zu dem das Schiff anlegt und mit dem Aufruf der Methode `castOff(...)` definieren Sie den genauen Zeitpunkt zu dem das Schiff ablegt. (Bei Schlumpf-Schiffen sind dies wirklich Zeitpunkte ;-)
Wenn die `terminate()`-Methode aufgerufen wird, dann muss das Schiff (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.
Jedes einzelne Schiff muss ein eigenständiger Thread sein. Insbesondere die Methoden

- `takeTimeForSailingTo(...)`
- `takeTimeForBoardingAt(...)`
- `dockAt(...)`
- `castOff(...)`
- `lastDeed(...)`

müssen jeweils vom jeweiligen Bus/Thread selbst ausgeführt werden.
In einem Schiff dürfen sich zu keinem Zeitpunkt mehr als `maximumNumberOfSmurfsPerShip` (im weiteren kurz `msps`) "Fahrgast"-Schlümpfe befinden. (Es gilt: 1≤mspb≤nos; Default ist `msps`=97; der Kapitän zählt nicht mit).
Über `getDebugState()` können Kontrollausgaben ein- und ausgeschaltet werden.

Die Schiffe fahren die eingangs beschriebenen Linien ab.
Das erste Schiff startet an der Anlegestelle 0 und fährt die Anlegestellen im Uhrzeigersinn ab (0→1→2→3→4→5→0→1→...).
Das zweite Schiff startet an der Anlegestelle 1 und fährt die Anlegestellen gegen den Uhrzeigersinn ab (1→0→5→4→3→2→1→0→...).
Das dritte Schiff startet an der Anlegestelle 2 und fährt die Anlegestellen im Uhrzeigersinn ab (2→3→4→5→0→1→2→3→...).
Das vierte Schiff startet an der Anlegestelle 3 und fährt die Anlegestellen gegen den Uhrzeigersinn ab (3→2→1→0→5→4→3→2→...).
Das fünfte Schiff startet an der Anlegestelle 4 und fährt die Anlegestellen im Uhrzeigersinn ab (4→5→0→1→2→3→4→5→...).
Das sechste Schiff startet an der Anlegestelle 5 und fährt die Anlegestellen gegen den Uhrzeigersinn ab (5→4→3→2→1→0→5→4→...).

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 das jeweilige Schiff betritt und die Methode `leave(...)` genau zu dem Zeitpunkt, zu dem der Schlumpf das jeweilige Schiff verlässt. Nachdem der Schlumpf das jeweilige Schiff betreten hat (`enter(...)`) und bevor er das jeweilige Schiff wieder verlässt (`leave(...)`) zeigt der Schlumpf sein Schiff-Ticket vor und ist einfach auf dem Schiff. Dies wird durch den Aufruf von `beThere(...)` modelliert.
Schlümpfe haben es eilig und steigen nur in Schiffe ein, die sie ohne Umwege an das Ziel bringen.
Auf einem Schiff dürfen sich zu keinem Zeitpunkt mehr als `maximumNumberOfSmurfsPerShip` (im weiteren kurz `msps`) "Fahrgast"-Schlümpfe befinden. (Es gilt: 1≤msps≤nos; Default ist **`mspb`**=97; das Schiffspersonal 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 **getWantedNumberOfShips()** soll Ihren Wunsch für die Anzahl der Schiffe in jeweiligen Testlauf abliefern. Der Rückgabewert muss 6 sein.
Die von Ihnen zu implementierenden Methode **getWantedNumberOfLandings()** soll Ihren Wunsch für die Anzahl der Anlegestellen in jeweiligen Testlauf abliefern. Der Rückgabewert muss 6 sein.
Die von Ihnen zu implementierenden Methode **getWantedMaximumNumberOfSmurfsPerShip()** soll Ihren Wunsch für die Obergrenze der Anzahl Schlümpfe auf einem Schiff abliefern.
Die von Ihnen zu implementierenden Methode **getWantedMaximumNumberOfShipsPerLanding()** soll Ihren Wunsch für die Obergrenze der Anzahl Schiffe an einer Anlegestelle.
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 Schiffe, die Anzahl der Anlegestellen, die Obergrenze für Schlümpfe pro Schiff und die Obergrenze für Schiffe pro Anlegestelle 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(),
        getWantedNumberOfShips(),
        getWantedNumberOfLandings(),
        getWantedMaximumNumberOfSmurfsPerShip(),
        getWantedMaximumNumberOfBussesPerLanding()
     );
    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_.shipPart5.*;
…
public class TestFrame … TestAndEnvironment_A {
    …
    @Override public String getAuthor() { … }
    @Override public int getWantedNumberOfSmurfs() { … }                                    // <- 2500
    @Override public int getWantedNumberOfShips() { return 6; }
    @Override public int getWantedNumberOfLandings() { return 6; }
    @Override public int getWantedMaximumNumberOfSmurfsPerShip() { … }                       // <- 97
    @Override public int getWantedMaximumNumberOfShipsPerLanding() { return 2; }

    @Override public void doTest(
        Integer requestedNumberOfSmurfs,
        Integer requestedNumberOfShips,
        Integer requestedNumberOfLandings,
        Integer requestedMaximumNumberOfSmurfsPerShip,
        Integer requestedMaximumNumberOfShipsPerLanding
    ){
        …
        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 Ship … Ship_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()**, **castOff(…)**, **dockAt(…)**, **takeTimeForSailingTo(…)** oder **takeTimeForBoardingAt(…)** "machen"/aufrufen.
Der "Aktivitäts-Zyklus" - beginnend mit der ersten "Aktion" - eines **Ship**s ist:
dockAt(…), **takeTimeForBoardingAt(…)**, **castOff(…)**, **takeTimeForSailingTo(…)**
lastDeed() ist die letzte "Aktion" eines **Ship**s.

```
package solution;
import _untouchable_.shipPart5.*;
…
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.
takeTimeForDoingStuffAtCurrentPosition(…)ist die erste "Aktion" eines **Smurf**s.
Der "Aktivitäts-Zyklus" - beginnend mit der zweiten "Aktion" - eines **Smurf**s ist:
enter(…), **beThere(…)**, **leave(…)**, **takeTimeForDoingStuffAtCurrentPosition(…)**
lastDeed() ist die letzte "Aktion" eines Schlumpfes.