Università di Padova - Laurea Triennale/Magistrale in Ingegneria Informatica
Corso di *Sistemi Operativi*
Libreria dei supporto al laboratorio v.2.3 25/05/2019 (M.Moro)

1.1 Aggiunta Regione Critica
1.2 Condition di Monitor resa public, aggiunta SyncMultiBuf2 con due semafori, altre variazioni minori
1.3 Aggiunto p(qty,timeout) in CountSem, aggiunto cWait(timeout) in Monitor.Condition, aggiunto il set di classi per la simulazione dello strato di I/O
1.4 Modificato package Os in os e altri aggiornamenti
1.5 Aggiunto enterWhen(condition, timeout) in os.Region e readLong in os.SysRS
2.0 Strutture di supporto che usano generics, Emulazione del rendez-vous di ADA
2.1 Demo RMI
2.2 Bug fix: Monitor, corretta gestione del timeout, corretta gestione del nesting mEnter; corretta gestione dei timer; RWMonSingleBuf, gestione buffer vuoto
2.3 Bug fix: Monitor, corretta una condizione di corsa che generava una eccezione del tipo "Not owner of MutexSem" in occasione di un mExit()

La libreria di classi Java, a supporto delle esercitazioni e del laboratorio del corso, è composta di 5 *package*:

- `os` la libreria vera e propria
- `os/ada` emulazione del rendez-vous di ADA
- `osTest` un insieme di classi di prova
- `osTest/ada` un insieme di classi di prova per la parte ADA
- `osExtra` classi accessorie (NB: in *appletDeprecated* classi che usano gli Applet ora deprecati)

La libreria di classi Java, a supporto delle esercitazioni e del laboratorio del corso, va utilizzata in due modi diversi:

- in una prima fase, lo studente può limitarsi a realizzare qualche esempio d'uso dei costrutti di sincronizzazione/comunicazione messi a disposizione dalla libreria, prendendo come spunto i test dimostrativi acclusi;
- in un secondo momento, una volta studiato il Monitor di Java, potrà analizzare l'implementazione di alcune delle classi, di cui si fornisce il relativo sorgente, e trarre spunto per la realizzazione di altri costrutti di uso generale.

Le classi del *package* `osExtra` non verranno illustrate: lo studente potrà analizzarle per comprendere meglio come utilizzare il *multithreading* con l'interfaccia grafica AWT di Java, come utilizzare le classi interne e realizzare *applet*.
Oltre a questo documento (readme.<version>.doc) il software è corredato da alcuni programmi *batch*:

- `gdoc.bat` genera i file con API javadoc
- `oscomp.bat` compilazione di tutti i sorgenti
- `delclass.bat` cancella tutti i .class
- `runj.bat` esegue il main della classe, passata come parametro, nel package corrente

Qui di seguito vengono presentati brevemente i costrutti disponibili, assieme ad alcune informazioni essenziali per il loro uso. Gli altri dettagli sono reperibili nella documentazione ipertestuale generata dai sorgenti (javadoc).
N.B. Alcuni sorgenti potrebbero produrre un *warning* su 'unchecked or unsafe operations' che può essere al momento ignorato.

PGraph

Si tratta di un'unica classe che serve come scheletro per la manipolazione di *grafi di precedenza*: il programma legge il grafo da *stdin* o da un file, il cui nome deve essere fornito come primo argomento della linea di comando, calcola la matrice di incidenza che rappresenta il grafo, produce la suddivisione dei nodi in livelli e calcola, per ciascun livello, il grado di parallelismo e quello massimo complessivo. Il calcolo del parallelismo è basato solo sulla struttura del grafo (cioè sulle sole dipendenze) e non sui tempi: questo significa che una interpretazione comprensiva dei tempi di esecuzione potrebbe portare ad un parallelismo effettivo inferiore al massimo qui indicato, nel senso che, pur avendo un numero di processori inferiore a quello di massimo parallelismo, si potrebbe ottenere in certi casi lo stesso tempo totale di esecuzione. L'output può essere salvato su file, il cui nome dev'essere fornito come secondo argomento sulla linea di comando, o può essere quello di default `$$PGraph.txt`: si vedano i file di esempio `PGraph1.txt`,

`PGraph2.txt, PGraph3.txt`. Quando un processo può essere 'spalmato' su più livelli, nell'output compare con il suo nome originario nel primo livello possibile, e con apici nei livelli successivi.

Semaforo

Vi sono diverse classi, di complessità via via crescente, che implementano le diverse varianti del costrutto.

SimpleSemaphore
Forma semplificata di semaforo binario/numerico. Non garantisce il FIFO di attesa dei *thread* sospesi.
Costruttori:

`SimpleSemaphore(int valoreIniziale, int massimoValore)`
: Semaforo numerico; si noti un v() sul semaforo, quando questo ha un valore pari al massimo fissato, corrisponde ad un no-operation.

`SimpleSemaphore(int valoreIniziale)`
: Come sopra con valore massimo pari a quello iniziale

`SimpleSemaphore(boolean valoreIniziale)`
: Semaforo binario.

`SimpleSemaphore()`
: Semaforo binario privato del processo che crea l'istanza, inizializzato a rosso (false).

Demo:

`TestMutex`
: Mutua esclusione con semafori binari

Classi di supporto: `Sys`, `Util`

SimpleSemaphore2
Altra forma semplificata, vedi precedente.

Demo:

`TestPipeFile`
: Pipe su file controllato da semafori numerici. Utilizza due finestre grafiche per l'interazione tra *thread* e utente.

Classi di supporto: `PipeFile`, `TASys`, `TextArea`, `TextAreaInputStream`, `TextAreaOutputStream`

Semaphore
La classe base per tutte le varianti di semaforo, alcune realizzate in questa stessa classe differenziando i costruttori e alcuni metodi, le altre come classi derivate da questa.
Costruttori:

`Semaphore(int valoreIniziale, int massimoValore, boolean codafifo)`
: Semaforo numerico; un v() sul semaforo con valore pari al massimo fissato corrisponde ad un no-operation. Se `codafifo` è `true`, la coda dei thread associata al semaforo è FIFO, altrimenti LIFO.

`Semaphore(int valoreIniziale, int massimoValore)`
: Come sopra con coda FIFO (semaforo numerico tradizionale).

`Semaphore(int valoreIniziale)`
: Come sopra con valore massimo pari a quello iniziale

`Semaphore(boolean valoreIniziale)`
: Semaforo binario.

Demo:

`TestPCSingleBuf`
: Produttori/Consumatori con singolo buffer controllato da semafori.

`TestPCMultiBuf`
: Produttori/Consumatori con buffer multiplo controllato da semafori.

`TestPCMultiBuf2`
: Produttori/Consumatori con buffer multiplo controllato da semafori (2 separati come mutex di produttori e consumatori) [v1.2]

`TestPhilDeadlock`
: N filosofi con semafori/forchetta, allocazione circolare che porta a *deadlock*. Animazione grafica. Il demo ha un *thread* che valuta periodicamente se si è in *deadlock* ed eventualmente lo segnala.

`TestPhilGerAll`
: N filosofi con semafori/forchetta, allocazione gerarchica.

`TestPhilGlobAll`
: N filosofi con semafori/forchetta, allocazione globale.

`TestTimedSem`
: Semaforo con timeout (metodo `p(long timeout)`).

`TestSendReceive`
: Comunicazione diretta tra *thread*, sincrona o asincrona, controllata da semafori. Utilizza una finestra di dialogo.

Classi di supporto: `WaitingThread`, `Timeout`, `Producer`, `Consumer`, `SyncSingleBuf`, `TestPC`, `SyncMultiBuf`, `SyncMultiBuf2`, `Phil`, `PhilAnim`, `PhilDeadlock`, `PhilGerAll`, `PhilGlobAll`, `SendRcv`, `CMsg`, `FifoSBuf`, `SBuffer`, `RcvThread`, `MsgDesc`

CountSem

Estende `Semaphore` per realizzare un semaforo numerico dotato di p e v con peso (multiple). Viene rimosso il controllo sul massimo valore. [1.3] Esiste anche la versione del metodo p() con timeout.
Costruttori:

`CountSem(int valoreIniziale, boolean codafifo)`
: Semaforo numerico senza massimo; con `p(int quantita)` e `v(int quantita)` con peso. Se `codafifo` è `true`, la coda dei thread associata al semaforo è FIFO, altrimenti LIFO.

`CountSem(int valoreIniziale)`
Come sopra con coda FIFO.
Demo:

`TestCountSem`
: Prova p() e v() con peso.

Classi di supporto: `Timeout`, `Util`

PrioSem

Estende Semaphore per realizzare un semaforo numerico con accodamento dei thread in base ad una chiave di priorità.
Costruttori:

`PrioSem(int valoreIniziale, int massimoValore)`
: Semaforo numerico con coda a priorità; `v()` ereditato da `Semaphore`.

`PrioSem(int valoreIniziale)`
: Come sopra con valore massimo pari a quello iniziale

`PrioSem(boolean valoreIniziale)`
: Semaforo binario con coda a priorità.

Demo:

`TestPrioSem`
: Inserimento nel semaforo in ordine casuale ed estrazione in ordine di priorità.

Classi di supporto: `Util`

MutexSem

Estende `Semaphore` per realizzare un semaforo binario di mutua esclusione (valore iniziale verde) ma che consente al medesimo *thread* un *nesting* delle chiamate, entrando successivamente più volte nella regione critica e uscendone altrettante. Viene tenuto un conteggio del livello di *nesting* proprio del medesimo *thread*.

Costruttori:

`MutexSem()`

Semaforo binario inizializzato a verde con funzione di mutex innestabile.

Demo:

`TestMutexSem`

Analogo a `TestMutex` ma con *nesting* variabile.

Classi di supporto: `Sys`, `Util`

Regione critica condizionale

L'implementazione è contenuta nella classe `Region` che di norma viene estesa dalla regione critica definita dalla applicazione al fine di dichiarare, come attributi d'istanza privati, le variabili condivise il cui accesso va protetto in mutua esclusione. Quest'ultima viene imposta facendo uso delle primitive `enterWhen()` e `leave()`. Le forme previste sono:

- `enterWhen()` ingresso in mutua esclusione non condizionale
- `enterWhen(long t)` ingresso in mutua esclusione non condizionale con timeout
- `enterWhen(RegionCondition condition)` ingresso in mutua esclusione con condizione
- `enterWhen(RegionCondition condition, long timeout)` ingresso in mutua esclusione con condizione e timeout [1.5]
- `leave()` uscita

Al fine di consentire, all'interno del metodo `enterWhen()`, chiamato nella variante condizionata, di poter calcolare più volte la condizione di ingresso, secondo l'usuale semantica della regione critica condizionale, tale calcolo viene espresso sottoforma di un metodo di nome stabilito (`evaluate()`) che deve essere in qualche modo passato all'atto della chiamata `enterWhen()`. La tecnica normale per passare come parametro una tale informazione (nel linguaggio C esiste il concetto di 'puntatore a funzione' che manca in Java) è quella di passare un oggetto di una classe che implementa un'interfaccia che include quel metodo nella sua definizione. Nel nostro caso, l'interfaccia è la `RegionCondition`, e la classe che la implementa dovrà definire il metodo `evaluate()` in modo che esegua il calcolo dell'espressione che costituisce la condizione che deve essere vera per autorizzare l'ingresso del *thread* nella regione. La classe che implementa `RegionCondition` può essere una classe interna a quella che estende `Region` (al limite anche una classe anonima). Come esempio di definizione di una classe di questo tipo si veda l'esempio di un allocatore globale in `TestRegion`: l'espressione da calcolare come condizione per l'allocazione (`free>= qty`) è resa dal metodo `evaluate()` contenuto nella classe interna `CondEval`: un nuovo oggetto di questa classe viene passato in occasione della chiamata d'ingresso `enterWhen()`. Invece nell'esempio di allocatore 1-2 in `All12Reg` le 4 condizioni diverse sono espresse in altrettante classi anonime, rispettivamente definite all'atto delle chiamate a `enterWhen()`.

L'implementazione prevede anche un controllo sul corretto *nesting* delle regioni: a tale scopo ad ogni istanza di regione occorre associare, alla sua creazione, un livello > 0. Se vengono eseguite dallo stesso *thread* due successive chiamate a `enterWhen()` su regioni diverse, queste devono essere in ordine crescente di livello, pena il sollevarsi di un'eccezione `RegionNestingException`. Un'eccezione dello stesso tipo viene egualmente lanciata se si esce da un *nesting* di regioni non in ordine perfettamente inverso da quello d'ingresso e se si tenta di rientrare in una regione senza esserne usciti.

Costruttori:

`Region(int newLevel)`

Regione di livello di *nesting* `newLevel`.

Demo:

`TestRegion`

Allocatore globale di un pool di N risorse con regione critica condizionale e con timeout [1.5].

`TestRegionErr`

Genera volutamente errori di *nesting* per far vedere i controlli realizzati.

`TestAll12Reg`

Allocatore 1-2 con regione critica condizionale.

Classi di supporto: `Sys`, `Util`, `Timeout`, `All12Region`

Timer software

Si tratta dell'equivalente di un *timer* hardware, un dispositivo che, avviato, genera un'interruzione dopo un prestabilito lasso di tempo, impostato in fase di avvio. La routine di servizio ha di solito la responsabilità di segnalare a qualche processo lo spirare del *timer*. La simulazione software prevede che il *timer*, allo scadere del tempo, attivi un metodo, denominato *callback*, associato al *timer* alla sua creazione, che funge da routine di servizio. Il *timer*, durante il periodo di attesa, può essere momentaneamente sospeso e riattivato (`freeze()` `start()`) o fatto ripartire (`retrigger()`). Il *retrigger* può essere effettuato anche dal *callback* per produrre un funzionamento continuo del *timer*. Il periodo può essere cambiato (`setPeriod()`).

Timer

La classe base ha un periodo fisso (salvo reimpostazione). Si può utilizzare per simulare un dispositivo fisico che genera interruzioni periodiche.

Costruttori:

`Timer(TimerCallback callback, long periodo)`

Crea un *timer* associandovi l'istanza della classe che implementa `TimerCallback`, di cui verrà chiamato, allo spirare del timer, il metodo `call()`, e fissa il suo periodo caratteristico.

Demo:

`TestTimer`

Animazione grafica con cui è possibile regolare il periodo del *timer* circolarmente in un insieme di valori, sospendere e attivare il *timer*.

RandTimer

Variante del Timer con periodo casuale che viene automaticamente cambiato ad ogni *retrigger* con un valore in un *range* fissato alla creazione. Si può utilizzare per simulare un dispositivo fisico che genera interruzioni non periodiche.

Costruttori:

`RandTimer(TimerCallback callback, long periodoMinimo, long periodoMassimo)`

Determina il *range* (`periodoMinimo..periodoMassimo`) entro il quale verrà impostato il periodo casuale.

Demo:

`TestRandTimer`

Simile a `TestTimer` ma con periodo casuale.

Classi di supporto: `Util`

Eventi

Un evento è una forma elementare di sincronizzazione diretta asincrona asimmetrica. Un *thread* invia eventi ad un altro indicando una maschera di eventi e un *thread* destinatario. L'inviante non attende. Il ricevente può attendere se, in base al tipo di chiamata, gli eventi richiesti non sono presenti. Quando un ricevente cattura alcuni eventi, li cancella. Un evento può essere sovrascritto, perdendo memoria del precedente, se non viene catturato in tempo prima che arrivi il secondo (gli eventi non si accumulano). Le maschere di invio e ricezione sono sequenze di bit contenute in una parola (`int`). Con le funzioni OR e AND delle maschere, gli eventi possono servire per realizzare sincronizzazioni complesse (ad esempio far sì che un *thread* venga risvegliato se arriva un dato da uno qualsiasi di due dispositivi d'ingresso).

Events

Classe i cui metodi statici servono per l'invio (`Events.signal()`) e la ricezione (`Events.wait()`) di eventi. Le maschere degli eventi inviati sono associate al *thread* destinatario in un'istanza di `EventThread`.

Demo:

`TestEvent`

Prova di invio e ricezione eventi.

`TestEvent1`

Attesa di eventi.con timeout. Gli eventi sono generati con click sui bottoni dell'interfaccia grafica.

Classi di supporto: `Util`, `EventThread`

Mailbox

Costrutto di comunicazione a messaggi con le usuali primitive sincronizzate put (ovvero send) e get (ovvero receive). I messaggi sono oggetti generici, a parte le versioni con copia per le quali i messaggi devono estendere la classe astratta `CMsg` che consente di fare la copia dell'oggetto con il metodo `clone()`.

Demo:

`TestMailbox`

Prova di invio e ricezione via mailbox con stringhe impostate sull'interfaccia grafica..

Classi di supporto: `SyncMultiBuf`, `TASys`, `Timeout`, `CMsg`

Pipe

Un mailbox che è visto, ed usato, come una *stream* di byte. Le tipiche operazioni prevedono lettura e scrittura di un certo numero di byte indicato nella chiamata. Lo scrittore (il lettore) scrive (legge) quanti byte può fino a riempimento (svuotamento) del pipe, e poi si sospende se ne deve scrivere (leggere) ancora, eventualmente più volte finché l'operazione non è completata. La realizzazione usa semafori con peso.

Demo:

`TestPipe`

Prova di invio e ricezione via pipe con scritture di array di byte di lunghezza casuale e letture di lunghezza fissa.

Classi di supporto: `MutexSem`, `CountSem`, `Util`

Monitor di Hoare

Il Monitor di Hoare differisce da quello di Brinch Hansen per il fatto di consentire che un *condition signal* sia eseguito non come ultima istruzione di una procedura *entry*: pertanto può essere anche eseguito più volte nella stessa istanza di chiamata della procedura. Qui vengono proposte due diverse implementazioni.

SimpleMonitor

In questa implementazione il monitor applicativo ha l'obbligo di estendere la classe `SimpleMonitor`, di realizzare i propri metodi *extry* come `synchronized`, e di utilizzare istanze della classe `Condition`, interna a `SimpleMonitor`, per le code di attesa. Quando viene eseguito un *condition signal*, vengono risvegliati tutti i *thread* eventualmente in attesa sul *condition*: il primo che riesce a rientrare nella mutua esclusione (riacquisisce il *lock*) può effettivamente proseguire, gli altri tornano ad attendere sul *condition.* Non c'è pertanto alcuna garanzia sulla scelta del *thread* che viene risvegliato, e comunque il *thread* risvegliante mantiene il *lock* fino all'uscita dal metodo sincronizzato.

`TestCassettaPostale`

Si tratta dell'esempio del libro di testo.

`TestAllSimpleMon`
Allocatore 1/2 di risorse.

Classi di supporto: `CassettaPostale`, `Util`, `Sys`

Monitor

Questa seconda implementazione utilizza lo schema di realizzazione 'canonico' mediante semafori, e richiede che i metodi *entry* del monitor applicativo, che deve estendere `Monitor`, siano aperti e chiusi dalle chiamate ai metodi ereditati `mEnter` e `mExit`. Il limite è proprio dato da questo vincolo che costituisce per il programmatore dell'applicazione una disciplina non controllabile in fase di compilazione e potenzialmente fonte di errore rilevabile in esecuzione. Il vantaggio è che viene utilizzato un semaforo *urgent*, con coda gestita come LIFO (stack), su cui vengono accodati i *thread* che effettuano un *condition signal*, in modo da dare priorità al *thread* risvegliato. Quest'ultimo riacquisisce immediatamente la mutua esclusione ma allo stesso tempo blocca nuovi *thread* che vogliano entrare fino al completamento dei metodi *entry* chiamati precedentemente.

`TestPCMonSingleBuf`
Produttori-Consumatori con buffer singolo controllato da Monitor.

`TestPCMonMultiBuf`
Produttori-Consumatori con buffer multiplo controllato da Monitor.

`TestRWMonSingleBuf`
Lettori-scrittori su buffer singolo controllato da Monitor.

Classi di supporto: `TestPC`, `MonSingleBuf`, `MonMultiBuf`, RW`MonSingleBuf`, `Reader`, `Writer`, `MutexSem`, `CountSem`, `Sys`

Simulazione strato di I/O

Al solo fine di rendere più comprensibile il modello dello strato di gestione di I/O presentato dal libro di testo, è stata realizzata una simulazione di un sistema dotato di dispositivi installabili. Ogni dispositivo fisico viene simulato con l'ausilio di un *thread* ed ha associati un *device handler* e una routine di servizio delle interruzioni (*isr*). All'interno del sistema viene mantenuta (in una *hashtable*) l'associazione tra i *thread* e i canali che essi singolarmente aprono. Ogni canale esprime a sua volta l'associazione con il dispositivo rappresentato da un singolo descrittore; il dispositivo può essere aperto in forma esclusiva da un singolo *thread* alla volta oppure in forma non esclusiva da più *thread* concorrentemente. Sul canale è possibile effettuare chiamate ai metodi di interfaccia `read` e `write`, ciascuna delle quali si traduce in una chiamata al metodo fondamentale `doIO` che prevede i parametri descritti nel libro. Il metodo interagisce con il *device handler* del dispositivo indirizzato costruendo un IORB e accodandolo in un FIFO (un'istanza della classe `java.util.LinkedList`) associato al dispositivo. Le sincronizzazioni tra `doIO`, DH, dispositivo fisico e ISR sono quelle descritte nel libro.

Lo studente è invitato a confrontare la simulazione con la descrizione del libro e ad apportare estensioni quali l'aggiunta di altri dispositivi, la realizzazione di un metodo `doBuffIO` per l'I/O bufferizzato e di un dimostrativo che includa diversi *thread* applicativi.

Le classi che compongono questa realizzazione sono qui sotto brevemente illustrate.

Device fisico

`Device`
Classe base astratta per la simulazione di un *device* fisico

`InputDevice`
: Estende `Device` per un dispositivo di ingresso

`OutputDevice`
: Estende `Device` per un dispositivo di uscita

`SerialLineInDev`
: Estende `InputDevice`: simulazione di una linea seriale di input. I byte vengono letti da `Sys.in` (che essendo bufferizzato, attende si batta Return a fine riga; i byte acquisiti sono i codici ASCII dei caratteri letti).

`SerialLineOutDev`
: Estende `OutputDevice`: simulazione di una linea seriale di output. I byte vengono emessi su `Sys.out`.

`ADCDev`
: Estende `InputDevice`: simulazione di un ADC. I campioni da 16 bit vengono acquisiti come valori decimali dal file di testo `ADCSamples.dat`.

Device handler

`DeviceHandler`
: Classe base astratta che rappresenta il *thread* che funge da DH assieme alla associata ISR del dispositivo fisico

`SerialLineInDH`
: Estende `DeviceHandler`: DH per la linea seriale di input.

`SerialLineOutDH`
: Estende `DeviceHandler`: DH per la linea seriale di output.

`ADCDH`
: Estende `DeviceHandler`: DH per il dispositivo ADC.

Canale

`DeviceDesc`
: Descrittore di dispositivo.

`IOChannel`
: Canale di I/O apribile dai singoli *thread*. Include i metodi `open`, `read`, `write`, `close` e `doIO`.

`IORB`
: I/O Request Block costruito dal metodo `doIO` e passato al DH.

`ChannelThread`
: Associazione Thread-coda dei canali aperti dal *thread*.

`MachineConfig`
: Configurazione dei dispositivi di un sistema demo (1 seriale di input, 1 seriale di output, 1 ADC) con un main di collaudo (un solo *thread*).

Altro

`DeviceType`
`SerialLineType`
`ADCType`
: Parametri descrittivi della classe del dispositivo (non usati ma citati sul libro).

`IOErr`
: Classe attraverso la quale il metodo `doIO` può ritornare una descrizione di errore valutabile dall'applicazione..

`TestIO`
Demo con più thread che aprono i canali di input e output in modo non esclusivo.

Classi di supporto: `MutexSem`, `Semaphore`, `Util`

Emulazione di ADA

In larga misura lo API è descritto nel capitolo 13 del libro di testo (3^ edizione). Le classi di supporto all'emulazione sono inserite nel *package* `os.ada`. Gli esempi mettono di evidenza come usare lo API messo a disposizione e la semantica equivalente del rendez-vous.
Demo:

`TestADAMutexTA`
: Test della classe `ADATASys` che rappresenta un server su `TextArea`; in questo caso l'oggetto `ADAThread` non coincide con l'oggetto che rappresenta il thread di Java.

`TestADANSemSBuf`
: Buffer singolo con 2 semafori binari e RV di Ada.

`TestADARW`
: Test di lettori-scrittori in ADA.

`TestADAAll123`
: Test di allocatore 1-2-3 in ADA.

`ADA3Task`
: Esempio della sincronizzazione di 3 thread in ADA.

`ADABox`
: Esempio del libro.

`ADADelayedRV`
: Esempio del RV ritardato: 4 operazioni.

`ADAIncDec`
: Esempio di RV di Ada con classe che estende altra classe.

`ADAMBufPC`
: Multi buffer in ADA.

`ADAOper`
: Test 4 operazioni con guardie casuali.

`ADASem`
: Esempio semaforo in ADA.

Classi di supporto: `Sys`, `Util`, `Timeout`, `ADA3TaskStr`, `ADAAll123Str`, `ADAMBuf`, `ADANSem`, `ADANSemStr`, `ADARW`, `ADARWDb`, `ADARWStr`, `ADATASys`

Altri demo

`TestTASys`
Demo di prova della classe di supporto `TASys` che fornisce finestre grafiche di interfccia.

`TestTimedMon`
Demo di prova della funzione di timeout nel Monitor.

`TestRemDate`
Demo illustrativa del meccanismo RMI di Java: per una prova completamente in locale eseguire il comando batch `testrmi`; se si vuole eseguire solo il client con server su altra macchina, eseguire il comando batch `testrmiclient <indirizzo del server>`.