UNIVERSITÀ DEGLI STUDI DI BERGAMO

Dipartimento di Ingegneria Gestionale, dell'Informazione e della Produzione

itemisCREATE

Contatti:
Prof. Angelo Gargantini – angelo.gargantini@unibg.it
Dott.ssa Simona Piazzi – simona.piazzi@unibg.it

YAKINDU Statechart Tools (SCT)

È un plugin di Eclipse che permette di eseguire le seguenti operazioni su statechart:

- Modellazione
- Validazione e simulazione
- Code generation
- Unit testing

Esercizio Guidato

Si supponga di voler modellare attraverso uno statechart il comportamento di una lampadina.

- Inizialmente la lampadina è spenta.
- Se la lampadina è spenta e viene premuto l'interruttore, la lampadina si accende.
- Se la lampadina è accesa e viene premuto l'interruttore, la lampadina si spegne.

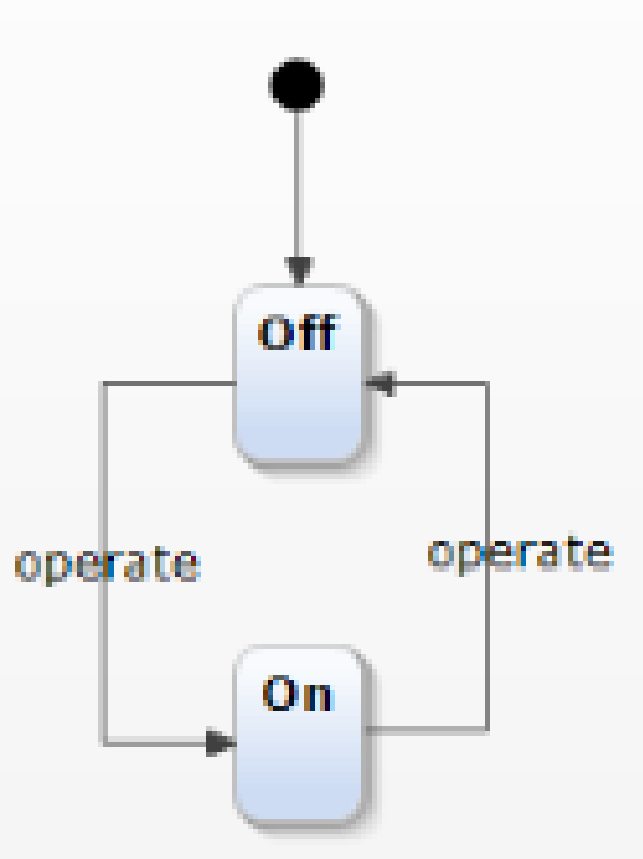
Step 1 – Eclipse set up

Creare un nuovo progetto: File → New → Project → General → Project

Creare una cartella per il modello: tasto destro sul progetto → New → Folder

Creare uno statechart vuoto: tasto destro sul progetto → New → Other → Statechart Model

Step 2 – Disegnare il modello



Ciascuno statechart è costituito da 3 elementi principali:

- Uno stato iniziale (indicato con il pallino nero)
- Un insieme di stati ciascuno caratterizzato da un nome significativo
- Un insieme di transizioni che permettono di passare fra i diversi stati

Ognuno di questi elementi può essere inserito tramite la Palette e spostato a piacere nell'aerea di lavoro.

Per simulare il modello ottenuto, tasto destro sul file → Run As → Statechart Simulation. Lo stato corrente sarà colorato di giallo e gli eventi potranno essere lanciati cliccando sul loro nome all'interno della Simulation View (lato destro).

Step 3 – Code Generation

```
GeneratorModel for yakindu::java {

    const PROJECT : string = "Lampadina"

    const FOLDER : string = "src-gen"

    statechart Lamp {

        feature Outlet {
            targetProject = PROJECT
            targetFolder = FOLDER
            libraryTargetFolder = "src"
        }

        /* Specify package names */
        feature Naming {
            basePackage = "example"
        }
    }
}
```

Uno dei vantaggi di itemis CREATE è il fatto che permette di generare codice in maniera automatica a partire dal modello disegnato.

Tasto destro sul file del modello (.ysc) → New → Other → Code generator model → Next → Selezionare la cartella di model e dare un nome al file .sgen → Next → Dal menù a tendina selezionare Java e poi selezionare il file di modello (.ysc) → Finish

Aprendo il file che viene generato, vengono mostrate tutte le indicazioni relative all'organizzazione del progetto e il linguaggio utilizzato per la generazione del codice.

La creazione del codice è automatica e viene modificata ogni volta che si apportano dei cambiamenti al modello.

Event-Driven vs Cycle-Based Execution

Il codice generato può seguire uno dei seguenti schemi durante l'esecuzione:

- Cycle-Based: la macchina a stati esegue gli eventi ad intervalli di tempo regolari. Questo significa che più eventi possono essere processati nello stesso momento e di conseguenza è possibile la combinazione di più eventi per il passaggio da uno stato all'altro (esempio eventA && eventB)
- Event-Driven: la macchina a stati esegue un evento non appena questo viene scatenato. Questo comporta che la macchina a stati non sarà mai in grado di processare due eventi contemporaneamente rendendo una condizione come eventA && eventB sempre falsa.

Per specificare il tipo di esecuzione si deve utilizzare la notazione @CycleBased o @EventDriven. L'impostazione di default prevede una esecuzione cycle-based con un intervallo di tempo di 200ms

Esercizio 1 – Gestore di chiamate

Si supponga di voler modellare il seguente gestore di chiamate:

- Inizialmente il sistema si trova in uno stato di attesa.
- Quando arriva una chiamata, l'utente può scegliere se accettarla o rifiutarla.
- Se l'utente accetta la chiamata, il sistema comincia a tenere traccia della durata e attende che la chiamata termini.
- Dopo aver riattaccato, il sistema mostra la durata totale della chiamata per poi tornare nello stato di attesa.

Creare il modello e generare il codice.

Tip – Interfaces

```
interface User:

    in event accept_call

    in event dismiss_call


interface Phone:

    var duration : integer

    in event incoming_call
```

Per una migliore gestione di numerosi eventi scatenati da attori diversi, è buona pratica suddividerli in interfaces.

In questo modo ad ogni attore viene assegnato un insieme di eventi che può lanciare.

È possibile anche definire delle variabili secondo la notazione:

var nomeVariabile: tipoVariabile

Tip – Time events

Se durante la scrittura del modello vengono utilizzare delle espressioni temporali, quando si va a generare il codice è necessario aggiungere un timer service come quello mostrato nell'immagine.

Esempi di espressioni temporali:

- every 1 s → esegue un azione ogni secondo
- after 2 s → la transizione viene eseguita dopo 2 s

```
feature GeneralFeatures {

    TimerService = true

}
```

Esercizio 2 - Ascensore

Si supponga di voler modellare il comportamento di un ascensore.

- Inizialmente l'ascensore si trova al piano terra ed è fermo.
- Quando viene selezionato un piano, l'ascensore chiude le porte dopo 2s
- Dopo che le porte dell'ascensore si sono chiuse, l'ascensore comincia a muoversi verso l'alto o verso il basso.
- Quando il piano desiderato è stato raggiunto, l'ascensore apre le porte dopo 1s

Scrivere il modello, generare il codice e scrivere qualche unit test per verificare il comportamento.

Tip – Unit test example

Gli unit test devono essere posizionati all'interno di una cartella di test in modo da tenere in ordine il progetto. Tasto destro sul progetto → New → Folder → test

Per creare una classe di test: tasto destro sul folder test → New → Other → SCTUnit Test Class

La scrittura dei test è molto simile agli unit test di Java con una leggera differenza di sintassi:

- raise event: variableValue → permette di lanciare un evento e di settare il valore di una variabile
- proceed 1 cycle / proceed 5 s → permette di far avanzare la simulazione di uno step o di un determinato intervallo temporale
- active(stateName): indica se lo stato è quello corrente oppure no

```
@Test
operation startMovingWhenFloorRequested() {
    enter
    raise floor: currentFloor+5
    proceed 1 cycle
    assert(active(Elevator.main.Moving))
    exit
}
```

Esercizio 3 – Vending Machine

Si supponga di voler modellare il comportamento di una macchinetta automatica.

- Inizialmente la macchinetta ha 5 snack e 3 bibite ed è in attesa.
- Se l'utente ha inserito un numero sufficiente di monete, la macchinetta eroga il prodotto scelto
- Se l'utente ha inserito più denaro del dovuto, la macchinetta eroga il resto dopo 1 s
- Se l'utente ha inserito delle monete per errore e preme il pulsante di reset, la macchinetta restituisce il denaro e torna nello stato di attesa.
- Se la macchinetta termina i prodotti, passa in uno stato di blocco fino a quando non viene riempita.
- Ogni 100 ordini la macchinetta richiede manutenzione.

Scrivere il modello, generare il codice e scrivere gli unit test