ISS2025

Fase 2 - Relazione Finale

Bacchelli Valentina
Git repo: https://github.com/VBacchelli/ISS2025/

Introduzione

A differenza del processo di tipo *bottom-up* adottato nella fase 1 del corso, in questa seconda fase si è seguito un approccio *top-down*, partendo dall'analisi dei requisiti e del problema, per arrivare allo sviluppo di un sistema distribuito. Uno degli aspetti centrali di questa fase è stata la **realizzazione di un modello eseguibile del sistema utilizzando Qak**, un linguaggio DSL (Domain Specific Language) basato sul concetto di attori. Attraverso questi strumenti, si sono introdotti gli **agenti software situati**, ossia attori associati a un determinato ambiente (fisico o simulato), realizzati tramite Raspberry Pi o in ambienti virtuali come **WEnv**.

Finalità

Dal mio punto di vista, questa parte del corso ha sicuramente avuto come finalità principale quella di introdurre il modello ad attori, un modello di programmazione che non avevamo avuto modo di conoscere in altri corsi. Questo ha richiesto uno sforzo maggiore del previsto: se inizialmente gli attori potevano sembrare simili agli oggetti della programmazione orientata agli oggetti, ho poi compreso che confrontare un attore con un POJO (Plain Old Java Object) è come confrontare entità "vive" con entità "morte".

In secondo piano, approcciarci al problema in modo top-down ci ha fatto capire l'importanza dei **modelli** e, di conseguenza, dei DSL (Domain Specific Languages). Infatti, anche se in teoria qualsiasi linguaggio general-purpose può essere usato per risolvere un problema, non sempre la sua sintassi è sufficientemente espressiva per comunicare con precisione l'intenzione del progettista. In particolare, il vantaggio dato da Qak è quello di creare un modello **eseguibile** che mette già in evidenza il comportamento atteso del sistema pur tralasciando i dettagli implementativi, riducendo il rischio di ambiguità o fraintendimenti che spesso emergono quando si lavora con specifiche in linguaggio naturale.

Verso un approccio top-down

Nella fase precedente, l'approccio usato era stato di tipo bottom-up. Si era infatti partiti dai componenti software a nostra disposizione e si era creato il nuovo sistema per sintesi, un pezzo alla volta.

Stavolta invece abbiamo introdotto il processo top-down. Questo consiste nel fare un progetto prima di scrivere del codice, ma ancora prima nel fare un'analisi del problema che si ha davanti, in modo da assicurarsi di aver compreso tutti i requisiti e ridurre il rischio di malfunzionamenti. Proprio per questo però non ci basta un'analisi in linguaggio naturale, perché esso risulta spesso ambiguo: da qui siamo arrivati a parlare di **linguaggi di modellazione** e **metamodelli**. Conoscevamo già l'UML, questo però si è rivelato inadatto nel nostro caso perché in un sistema distribuito le entità sono vive, interagiscono via messaggi e non con le relazioni tra classi rappresentabili in UML. Inoltre, per i motivi che ho spiegato nella sezione sulle finalità, è preferibile avere un modello già eseguibile. Da questa premessa siamo arrivati alla costruzione di un modello in Qak.

DSL

L'importanza dei linguaggi, già discussa nella fase 1 del corso, si collega direttamente all'uso dei DSL (Domain Specific Languages) nella modellazione dei sistemi. Quando affrontiamo un problema secondo un approccio top-down, il punto di partenza non dovrebbe essere un linguaggio o una libreria, ma le **entità e le interazioni richieste dal problema stesso**. Solo in seguito possiamo chiederci se un certo linguaggio (general-purpose o DSL) offre i costrutti necessari per modellarle in modo espressivo.

Tuttavia, spesso il linguaggio scelto non mette direttamente a disposizione i concetti di cui abbiamo bisogno. Si tratta del concetto di abstraction gap già introdotto nella fase 1. Per colmarlo, abbiamo varie possibilità:

- cercare una libreria nel linguaggio che già conosciamo;
- cercarne una in un altro linguaggio che implementi le entità richieste;
- scegliere un altro linguaggio che supporti in modo nativo i concetti di interesse;
- in mancanza di alternative, realizzare una libreria o addirittura un nuovo linguaggio.

I DSL, a differenza dei linguaggi general-purpose, non sono solitamente Turing-completi, ma sono specifici per un dominio e, in quanto tali, offrono costrutti ad alto livello che consentono di esprimere concetti rilevanti in modo più diretto ed efficace. Questo rende i DSL particolarmente utili quando si desidera modellare sistemi complessi partendo dai concetti del dominio piuttosto che dai dettagli implementativi.

POJO vs Attori

Inizialmente, la distinzione tra il modello ad oggetti e quello ad attori può sembrare sottile: entrambi definiscono entità con stato e comportamento. Tuttavia, approfondendo il lavoro con Qak, è emersa una differenza sostanziale legata alla **natura attiva** degli attori rispetto alla **passività** degli oggetti.

Un POJO rappresenta una struttura passiva che espone metodi i quali vengono invocati da entità esterne. È il flusso di controllo del programma a determinare quando e come un oggetto

viene attivato. Gli oggetti non possiedono un ciclo di vita autonomo: non "vivono" se non invocati.

Al contrario, un attore è una **unità computazionale autonoma**, dotata di un mailbox per ricevere messaggi e che decide in modo reattivo come comportarsi a seconda del messaggio ricevuto. Il flusso di controllo non è imposto dall'esterno, ma è **gestito internamente all'attore** attraverso i suoi possibili stati. Gli attori infatti si comportano come automi a stati finiti, per la precisione automi di Moore (cioè in cui l'output dipende solo dallo stato attuale, e l'input innesca solo un cambiamento di stato).

Questo porta a usare approcci differenti nei due casi: quando usiamo i POJO, il focus è sull'invocazione di metodi e la condivisione dello stato tramite la memoria; nel modello ad attori, lo stato è locale e la comunicazione avviene solo tramite **messaggi asincroni**, riducendo drasticamente il problema della concorrenza e della sincronizzazione (da qui il motto "Do not communicate by sharing memory… instead, share memory by communicating").

Rappresentare un sistema sotto forma di attori aiuta a comprendere meglio le interazioni che vi avvengono, e quindi riuscirlo a fare già dalla fase di modello grazie all'uso del Qak semplifica la successiva fase di progettazione.

Agenti situati

Gli **agenti software situati** sono entità computazionali strettamente dipendenti dall'ambiente specifico in cui operano, infatti:

- **Percepiscono l'ambiente** circostante tramite sensori logici o fisici;
- **Agiscono sull'ambiente**, modificandolo (ad esempio, invio di messaggi, aggiornamento di dati, attivazione di processi);
- Sono **integrati in ambienti dinamici**, che evolvono nel tempo indipendentemente dalle azioni dell'agente stesso;
- Esibiscono un comportamento **influenzato dal contesto**, adattandosi in base alla propria percezione e alla storia delle interazioni.

Noi abbiamo avuto modo di sperimentare direttamente questi concetti tramite la realizzazione di progetti via via più complessi.

Ad esempio, nel primo progetto, *rasp2025ledalone*, l'attore incaricato del controllo del LED riceve comandi (*turnOn*, *turnOff*) sotto forma di messaggi, percepisce eventi dal mondo esterno e agisce sul dispositivo hardware, modificandone lo stato fisico.

In seguito si è realizzato *sonarled2025*, in cui abbiamo un led e un sonar collegati a un Raspberry e controllati da attori. Se la distanza rilevata dal sonar è inferione a un valore prefissato LIMIT si deve accendere il Led. L'evoluzione rispetto al primo è evidente poiché è stato aggiunto il sonar come nuovo componente.