# Relazione fine fase 2 ISS 25

Andrea Grano - andrea.grano@studio.unibo.it
REPO GITHUB: | https://github.com/AndreaGrano/issLab25.git

# Introduzione

In questa seconda fase del corso si è passati a un approccio **top-down** nello sviluppo di software, partendo da una prima fase di analisi dei requisiti, operando successivamente l'analisi del problema giungendo, infine, prima di procedere alla codifica, a costruire un **modello (eseguibile) del sistema** avente un alto livello di astrazione. Parallelamente sono stati introdotti i **sistemi software ad attori**, i quali agiscono come **automi a stati finiti di Moore**. Al fine di fornire un esempio di linguaggio volto alla modellizzazione tramite attori, è stato introdotto il *Domain Specific Language* Qak. In ultima istanza, sono stati introdotti gli **agenti situati**, realizzandone dei primi esempi con l'ausilio di RaspberryPi.

# Sviluppo top-down e modellizzazione

Nella fase precedente del corso si era operato seguendo un approccio bottom-up, sviluppando un primo applicativo per poi aggiornarlo incrementalmente aggiungendo componenti, integrando framework e riscrivendo parti di codice; in tal modo, però, vi è stato un proliferare di progetti e un costante rifacimento degli stessi componenti software (nonostante framework come Springboot limitino questa esigenza). L'applicativo sviluppato era esemplificativo, molto semplice, ma trasportando questa filosofia in un progetto reale, essa si dimostrerebbe profondamente *error-prone*, porterebbe a notevoli perdite di tempo e, collateralmente, alla scrittura di codice difficilmente manutenibile. Procedere in modo **top-down** permette, invece, di chiarire e comprendere da subito i requisiti e, conseguentemente, il problema da risolvere, ragionando anzitutto su cosa si debba realizzare, slegandosi in prima istanza dai problemi implementativi.

Generalmente è utile poter sviluppare un **modello**, ovvero una visione semplificata del sistema da realizzare, ancor meglio se eseguibile al fine di confrontarsi con il committente, analizzare approfonditamente alcuni aspetti, provare soluzioni, … Per velocizzarne la produzione è possibile avvalersi di Domain Specific Languages (DSL, si veda successivamente), sviluppati allo scopo; nel nostro caso è stato utilizzato Qak.

# Sistemi software ad attori come automi a stati finiti

I sistemi software distribuiti, specialmente nell'architettura a microservizi (o anche a nanoservizi), consistono di componenti software in esecuzione indipendente su vari nodi di elaborazione, aventi la necessità di scambiare informazioni tramite messaggi (anziché condividendo memoria, come tipicamente può avvenire nei sistemi concentrati) e di porre in esecuzione funzioni e/o oggetti al bisogno; tali componenti software vengono definiti **attori**. Questi è buona norma che agiscano quali **automi a stati finiti**, preferibilmente di Moore (ovvero con l'input che comporta solo un cambiamento di stato nell'automa, non direttamente dell'output); ogni attore a tempo di esecuzione si troverà sempre in un determinato stato e avvenimenti quali la ricezione di messaggi o la scadenza di *timer* avvieranno una determinata transizione di stato, qualora questa sia espressa nella definizione del comportamento dell'automa.

# Domain Specific Languages

Un Domain Specific Language (DSL) è un linguaggio elaborato per colmare l'***abstraction gap*** tra la programmazione, tramite linguaggio *general purpose* e con tutti i dettagli implementativi che comporta, e il dominio specifico di una tipologia di problemi, utilizzando costrutti e sintassi che rimandano specificatamente a esso. Possono essere utilizzati al fine di fare prototipazione rapida, in quanto un adeguato **motore di trasformazione** può convertire quanto espresso tramite il DSL in codifica in uno o più linguaggi di programmazione.

## Qak

A titolo esemplificativo, in questa fase è stato mostrato e utilizzato estensivamente il **linguaggio Qak**. Questo DSL permette di sviluppare velocemente modelli di sistemi ad attori distribuiti (riprendendo così, concettualmente, i microservizi visti nella fase 1), seguendo un metamodello volto a realizzare automi a stati finiti distribuiti che interagiscono tramite scambio di messaggi. In Qak vengono definiti **contesti** di esecuzione (che possono corrispondere ai vari nodi di elaborazione di una rete), nei quali operano uno o più attori, interni (se definiti nel medesimo file qak), *coded* (ovvero scritti "a mano" in un qualunque linguaggio) o esterni al modello corrente. Tali attori possono comunicare scambiando messaggi strutturati aventi diversa semantica, come *dispatch*, richieste ed eventi, localmente a un contesto o tramite l'utilizzo di protocolli quali TCP, UDP, WebSocket, MQTT (è presente il supporto per contattare un broker MQTT) e CoAP. Il motore di trasformazione, denominato Qak software factory, provvede a trasformare quanto espresso nei file qak in sorgenti Kotlin, principalmente, gestendo automaticamente e correttamente, ad esempio, tutta la parte implementativa riguardante la comunicazione. Un attore Qak è un automa a stati finiti di Moore e si trova sempre in un determinato stato durante l'esecuzione; la ricezione di messaggi, la scadenza di timer o istruzioni di goto causano transizioni dallo stato corrente a un altro (eventualmente coincidente con quello corrente), se opportunamente specificato (altrimenti, nel caso di un messaggio, questo viene ignorato). Ogni attore possiede una coda dei messaggi principale e una secondaria in cui memorizza i messaggi non elaborati.

Di base il linguaggio Qak non è Turing completo (similmente alla maggioranza dei DSL) e per renderlo tale è consentito includere nella specifica del comportamento degli attori frammenti di codice Kotlin.

# Agenti situati

Quanto fin qui visto nel corso, in termini di microservizi e attori, si presta bene alla realizzazione di **agenti situati**, ovvero software operanti in ambienti, digitali ma anche fisici, specifici. Tali agenti sono in grado di reagire a stimoli e cambiamenti del contesto in cui si trovano, in modo autonomo e proattivo, e sono in grado di coordinarsi in gruppo interagendo tra loro. La realizzazione a microservizi li rende particolarmente modulari, dunque più facilmente manutenibili e resilienti ai guasti; inoltre, la natura indipendente dei microservizi consente la realizzazione di ciascuno attraverso tecnologie eterogenee.

## Robot reali e primi esempi su RaspberryPi

Come detto, software per agenti situati possono gestire anche **robot** reali, implementati materialmente tramite sistemi a microcontrollore (e.g. Arduino) o processore (e.g. RaspberryPi) appositamente realizzati per ricevere informazioni da una grande varietà di **sensori** (i.e. dispositivi fisici che rilevano diverse tipologie di grandezze fisiche) e di **trasduttori** (i.e. dispositivi fisici che convertono grandezze fisiche in segnali) e governare una serie di **attuatori**. In questa seconda fase sono stati mossi i primissimi passi in questo senso .In prima istanza sono stati provatati semplici script Python eseguiti su RaspberryPi che raccolgono informazioni tramite un sensore, un sonar, e governano un attuatore, un led; successivamente, si è passati allo sviluppo di un prima sistema software, Sonarqak24, che gestisca il secondo sulla base dei dati misurati dal primo.

### Sonarqak24

Nel repository GitHub è possibile trovare il progetto Sonarqak24, ovvero un sistema software che una volta eseguito su una RaspberryPi permette di accendere e spegnere un led sulla base dei dati misurati da un sonar, nello specifico il modello HC-SR04, per il quale era stata fornita l'analisi del problema. La struttura del sistema è articolata su tre attori, operanti nel medesimo contesto (la scheda), ciascuno avente una specifica responsabilità (secondo le buone pratiche dell'ingegneria del software), denominati sonar24, sonardevice e datacleaner.

### *sonar24*

Questo attore si colloca al più alto livello di astrazione e quale porta di interazione con il sistema dall'esterno, sia in input che in output. Esso riceve e gestisce, infatti, due dispatch che comportano l'accensione o lo spegnimento del sonar; di fatto, ne delega, però, la gestione all'attore sondardevice, di più "basso livello". Quando il sonar risulta avviato, riceve i dati misurati e filtrati, tramite un evento locale al contesto al quale ha effettuato la sottoscrizione, e li pubblica presso un broker MQTT, perché un sistema esterno che abbia fatto la *subscribe* alla relativa topic possa ricevere i dati.

### *sonardevice*

Il componente sonardevice incapsula la gestione del sonar, accendendo o spegnendo il sonar, dipendentemente dal dispatch ricevuto, la cui gestione è stata delegata da sonar24. Agisce eseguendo opportuni script Python e, quando il sonar è in funzione, leggendo i dati prodotti, inviandoli, tramite un evento locale al contesto, a datacleaner perché li filtri.

### *datacleaner*

Datacleaner è sostanzialmente un componente filtro; riceve i dati da sonardevice, tramite un evento locale al contesto al quale si è sottoscritto, e li filtra, inviando solo i valori entro l'intervallo 0-150, per poi emettere un altro evento locale al contesto (che sarà ricevuto da sonar24).

## Prospettive future

Si ipotizza che verrà portato a definitivo compimento il percorso cominciato, concentrandosi sullo sviluppo di microservizi ad attori attivi su agenti situati (anche fisicamente realizzati). Grazie all'ausilio del DSL Qak sarà possibile procedere a rapide fasi di modellizzazione e prototipazione, nell'ottica di uno sviluppo Agile/Scrum.