Studente: Koltraka Kevin

Matricola: 0001189565

Github: https://github.com/ingegneria-sistemi-software-m/ISS_koltraka

Strutturerò questa relazione rispondendo a domande simili a quella di fine fase 1.

Relazione fine fase 2

L'obiettivo principale che ho percepito di questa secoda parte del corso è stato quello di introdurre un nuovo paradigma di programmazione: **il modello ad attori**.

Questo modello può essere visto come un'evoluzione del più classico modello ad oggetti. Qui il sistema diventa "un mondo di vivi", ovvero un sistema formato da entità autonome che hanno una capacità intrinseca di comunicare con l'esterno mediante scambio di messaggi.

Gli attori hanno "libero arbitrio" sui tipi di messaggi che vogliono gestire. In questo modello non si è più quindi limitati dalle semantiche di chiamata di procedura (caso concentrato) o da una semantica event-driven (caso distribuito); piuttosto, ogni attore è organizzato come un ASF in cui specifica a quali messaggi è interessato in ognuno dei suoi stati.

Per me è stato interessante realizzare che **questi stati e queste transizioni esistono anche in un sistema che non è organizzato nativamente con degli attori**. Esprimendo esplicitamente però questi concetti a livello di modello, il sistema risulta più chiaro e comprensibile (per un essere umano). Questo ritengo essere il vantaggio principale di un modello ad attori, oltre ad altri vantaggi come thread-safety o altre caratteristiche "più tecniche".

Un secondo obiettivo è stato quello di cementificare l'idea introdotta nella prima fase del corso riguardante il ruolo che hanno linguaggi nell'esprimere le idea che il programmatore ha in testa mentre tenta di risolvere un problema.

Abbiamo visto che le librerie, sono limitate al piano semantico: con una libreria posso sicuramente risolvere qualsiasi problema computabile (i linguaggi sono turing completi), ma non potrò mai costruirmi una sintassi che comunica le mie idee in maniera espressiva tanto quanto potrebbe fare un linguaggio.

In questo contesto abbiamo anche esplorato in che cosa consiste l'essenza del top-down piuttosto che del bottom-up (differenza tra informatico e ingegnere).

Quando si affronta un nuovo problema, il punto di partenza non dovrebbero essere le mosse di un linguaggio o di una libreria (bottom-up), piuttosto di dovrebbero studiare **le mosse richieste dal problema** (top-down). A questo se il linguaggio scelto supporta queste mosse, si può già procedere ad una progettazione. In caso contrario, si ha un abstraction gap da colmare.

Come si può colmare questo abstraction gap? Abbiamo varie possibilità:

- cerco qualche libreria nel linguaggio che ho scelto che implementa le mosse che mi servono
- se non esiste nessuna libreria nel linguaggio che ho scelto, la posso cercare in un altro linguaggio
- posso cercare direttamente un linguaggio, DSL o GP, che implementa queste mosse
- se non trovo niente, non mi rimane che scrivermi da solo o una libreria o un nuovo liguaggio considerando vantaggi e svantaggi delle due alternative

Nel nostro caso, siamo stati interessati a modellari sistemi distribuiti a microservizi, in cui ogni microservizio era un attore. Per affrontare questo problema abbiamo sfruttato il DSL QAK che, con i suoi costrutti linguistici (sistema, contesto, attore, messaggi, ...), ci ha offerto dei concetti di più alto livello con cui modellare il sistema e le interazioni al suo interno. In poche parole, ci ha aiutato a **colmare l'abstraction gap**.

Terzo obiettivo che ho percepito è stato quello di convincerci che il risultato dell'analisi del problema non è un qualcosa scritto in linguaggio naturale (ambiguo) ma è un **modello** (ovvero, una descrizione del problema che mira a catturarne gli aspetti salienti) **eseguibile** (ovvero, **comprensibile da una macchina (no "fuffa"))**

Il fatto che i modelli siano eseguibili è importante in quanto gli rende per forza di cose non ambigui. Inoltre, il fatto che il prodotto dell'analisi sia un modello, piuttosto che un programma vero e proprio, significa che si possono tralasciare i dettagli implementativi (che vengono nascosti nella parte sommersa) e ci si può concentrare sulle caratteristiche centrali del problema.

La caratteristica fondamentale di un modello è quindi l'essere comprensibile da una macchina, un modello può avere quindi tante forme (Classe, attore QAK, programma Prolog, ...). La scelta di quale modello utilizzare si basa sulla comprensione di quale **metamodello** si adatta meglio al problema. Nel nostro contesto di sistemi distribuiti, abbiamo osservato che il metamodello UML non risultava adatto ("mondo di morti"), per questo motivo abbiamo adottato il metamodello QAK.

Infine, l'ultimo obiettivo di questa fase penso sia stato quello di dare un esempio pratico che combinasse i vari concetti esplorati fino ad ora nel corso (microservizi, attori, scambio di messaggi, interazione, ...) esplorando il mondo degli agenti software situati; sia reali (raspberry pi) che virtuali (wenv).