

Studente: Koltraka Kevin

Matricola: 0001189565

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



Relazione fine fase 1

1. Quello che lo scrivente ha percepito sia la finalità dei contenuti discussi in questa prima fase, e quello che lo scrivente si aspetta sia lo scopo delle prossime fasi del corso.

Le finalità dei contenuti di questa prima fase del corso secondo quanto ho percepito sono state:

- 1) Richiamare/evidenziare l'importanza di una attenta analisi dei requisiti e del problema per capire COSA bisogna fare. Inoltre, richiamare/evidenziare l'importanza dell'applicazione di principi dell'ingegneria del software come: il Single Responsibility Principle e il Dependency Inversion Principle, durante la fase di progettazione (come), per poi poter andare ad implementare del codice manutenibile, riutilizzabile in tanti contesti diversi ("principio di iron-man")
- 2) Mostrare come nei sistemi software moderni il componente non è più un singolo oggetto ma un (micro)servizio all'interno di un sistema distribuito ed alcune conseguenze. In questo contesto, si possono generalizzare i principi citati nel punto (1) al fine di ottenere gli stessi benefici: vedi separazione del servizio di GUI dal servizio di Life; Siccome il microservizio life diventa un'entità a se stante posso pensare a tanti clienti che gli mandano/da cui riceve dati oltre alla GUI (logger, chatgpt, ecc.). Passaggio da interazioni H2M a interazioni M2M.
- 3) Evidenziare l'importanza dell'utilizzo di strumenti con un livello di espressività appropriato per lo sviluppo del sistema software desiderato. In caso quest'ultimi mancassero, l'importanza di saperseli costruire da se con librerie/linguaggi(DSL?).

2. I sistemi che lo scrivente è riuscito a realizzare e sperimentare, nell'ambito di quelli discussi nella prima fase del corso.

Tutti quelli discussi e presentati a lezione.

3. Le nuove abilità / competenze che lo scrivente pensa di avere appreso (o che sia possibile apprendere) dopo questa prima fase, sia in relazione alla parte 'pratica' del saper-fare (strumenti, tecnologie, etc.) sia con riferimento ad aspetti più 'concettuali' (metodologie, modelli, etc.)

Per quanto riguarda le competenze tecniche, ho imparato ad utilizzare le seguenti tecnologie: Gradle, SpringBoot e MQTT. Avevo già familiarità con Docker, i container e il concetto di Microservizio ma anche queste sono tecnologie che è possibile imparare in questa prima fase del corso.

Per quanto riguarda le competenze concettuali, non mi sento di aver "imparato" propriamente qualcosa di nuovo. Piuttosto, sento di star per imparare qualcosa di nuovo. Ho ancora dubbi in sospeso di questo tipo: come mai ci siamo concentrati tanto sulla comunicazione tra componenti distribuiti? I sistemi software concentrati sono obsoleti? Come mai il prof ha evidenziato l'importanza del linguaggio? Come mai il prof ha parlato tanto di modelli?

4. Quello che lo scrivente pensa sia il motivo per cui, come primo esempio di sistema software da sviluppare, è stato proposto, come caso di studio, il gioco della vita di Conway.

Penso che il gioco della vita di Conway sia stato scelto come primo caso di studio in quanto (come scritto nel materiale del corso) è un sistema complicato ma non complesso; ovvero, il sistema è formato da tante entità interagenti (il client, la GUI, le celle), ma le interazioni sono prevedibili e seguono regole semplici. Un sistema di questo tipo penso si presti bene per introdurre il modello dei microservizi, e per discutere di come realizzare le interazioni con vari tipi di protocollo (MQTT, WebSocket, ecc.)

5. Le motivazioni che lo studente ritiene siano alla base della scelta di utilizzare Java e framework (come SpringBoot) legati a Java per lo sviluppo del caso di studio e se ritiene queste scelte limitative oppure obsolete, indicando alternative che ritiene più moderne o più adatte.

Penso che la scelta di utilizzare Java e SpringBoot sia stata dettata da una questione di semplicità. Java è un linguaggio che tutti noi studenti conosciamo già, Spring boot un po' meno ma tutto sommato è abbastanza facile da usare. Non ritengo che queste scelte siano state nè limitative nè obsolete, in quanto penso che con qualsiasi tecnologia le cose da fare sarebbero state più o meno le stesse. (Se avessimo utilizzato le Java servlet, come a Tecnologie Web-T, la mia opinione sarebbe stata diversa).

Un'alternativa da considerare, penso possa essere il linguaggio Go, in quanto implementa degli strumenti interessanti per la comunicazione a scambio di messaggi (canali, comandi con guardia, ...) e la sua libreria standard contiene già strumenti come un: web server, pacchetti per definire

REST API e cose di questo tipo. Infine è un po' meno verboso di Java e forse un po' più espressivo dato che non utilizza le annotazioni che, personalmente, non apprezzo tanto in quanto rendono il codice più difficile da capire per me.

6. Gli aspetti di Ingegneria del software (key points) che il caso di studio ha permesso di richiamare o di introdurre nelle varie fasi 'evolutive' dei prototipi sviluppati durante la fase 1.

Keypoints:

- Domain Driven software development: bisogna capire quali entità appartengono al dominio del sistema da sviluppare per poter realizzare un modello che cattura tutti gli aspetti essenziali del sistema. Il rischio di un modello incompleto è quello di un sistema che non rispetta i requisiti o (forse peggio) di un sistema che fa delle assunzioni non presenti nei requisiti.
- Applicazione dell'inversione delle dipendenze con l'interfaccia IO/Dev: applicando questo principio siamo riusciti ad essere TECHNOLOGY INDIPENDENT, infatti mentre inizialmente stampavamo solo sulla console, è bastato definire una nuova classe "WsIODev", senza modificare nient'altro, per inviare messaggi su una websocket.
- Principi come DIP, SRP e l'utilizzo del pattern MVC combinati hanno formato il "principio di iron-man". Questo principio è ciò che ha permesso di prendere il gioco così com'era, e di incastonarlo nel framework Spring Boot per poter servire lo stato del gioco con una pagina HTML. Avremmo potuto fare la stessa cosa ma utilizzare qualsiasi altro framework (e.g. JavaFX) e il gioco non sarebbe cambiato in quanto qualsivoglia framework interagisce con il gioco solamente attraverso il suo controller.
- Il Single Responsibility Principle applicato ai servizi e non solo alle classi: separando le responsabilità in microservizi distinti (GUI e life) abbiamo reso possibile l'interazione col gioco bypassando la GUI. Questo apre la strada ad altri microservizi come un logger, uno stress-tester, ecc. Inoltre, si ottengono i vantaggi di un sistema a microservizi: scalabilità indipendente, guasti indipendenti, scrittura dei microservizi in linguaggi indipendenti, ecc.

7. Se lo scrivente ritiene oppure no, che il sistema software sviluppato al termine della prima fase sia un sistema distribuito basato su microservizi.

Io non sono troppo sicuro di quando servizio si possa considerare 'micro'. Tuttavia, posso dire che il sistema formato da 'conwayguialone' e 'conway25JavaMqtt' è "più a microservizi" rispetto al sistema 'conwaygui' che avendo due responsabilità (servire la GUI e implementare il gioco) può essere considerato più monolitico.

8. Il ruolo che lo scrivente ritiene abbia lo sviluppo di librerie 'custom' durante la realizzazione di un sistema software e perchè, per il caso di studio, sono state realizzate classi (relative al protocollo MQTT) inserite nella libreria unibo.basicomm23-1.0.jar

Lo sviluppo di librerie 'custom' durante la realizzazione di un sistema software serve a colmare l'abstraction gap. Se durante lo sviluppo del mio sistema mi rendo conto che gli strumenti a mia disposizione sono troppo lontani dai concetti del mio dominio, costruendo mediante librerie degli strumenti più vicini al livello di astrazione che desidero, posso sviluppare un sistema più robusto, leggibile, manutenibile, ecc. Inoltre, risparmio la fatica mentale associata all'utilizzo di strumenti di più basso livello.

La libreria `unibo.basicomm23-1.0` è stata utilizzata per colmare l'abstraction-gap formatosi con la decisione di voler utilizzare MQTT ma con la semantica di una comunicazione request-response. Siccome MQTT mette a disposizione solo `publish` e `subscribe` come strumenti, c'è tutta una fatica legata a:

- fare una request mediante una `publish`
- creare una topic per la risposta al mittente
- far aspettare il mittente su questa topic rendendo la comunicazione sincrona
- costruire un messaggio strutturato in maniera tale che il ricevente possa dedurre su quale topic fare la `publish` della risposta
- ...

La libreria è servita a nascondere tutta questo lavoro. In questo caso però, mi viene da pensare che forse una scelta più intelligente sarebbe stata utilizzare semplicemente utilizzare una REST API o una qualsiasi tecnologia che implementasse già il tipo di comunicazione desiderata. Mi rendo conto però che non sempre esiste già una soluzione pronta ai proprio scopi.

9. Opzionalmente: Il ruolo che lo scrivente ritiene abbia il concetto di linguaggio (di programmazione, ma non solo) più volte evocato dal docente come esempio di un salto evolutivo fondamentale dell'informatica rispetto al solo sviluppo di librerie.

I linguaggi di programmazione, un po' come le librerie, mettono a disposizione degli strumenti, che in base al sistema che si sta sviluppando, vanno a definire un abstraction gap più o meno grande. In questo senso, più un linguaggio permette di fare operazioni "vicine" al livello di astrazione dell'applicazione, più quel linguaggio risulta espressivo in quel contesto e quindi più l'abstraction gap è piccolo.

La differenza tra sviluppo di librerie e sviluppo di linguaggi penso stia nel fatto che le librerie siano più limitate. Le librerie infatti, possono solo mettere assieme gli strumenti che un determinato linguaggio fornisce, non possono creare qualcosa di completamente originale. Ad esempio: in Java non esistono dei veri tipi funzione ma solo interfacce funzionali, questo significa che quando passo una lambda expression ad una funzione, quest'ultima per invocare la funzione passata, dovrà chiamare un suo metodo `".apply()"`; NON potrà semplicemente utilizzare la notazione con le parentesi tonde come le normali funzioni. In questo contesto, non sarà mai

possibile costruire una libreria che permette di costruire oggetti funzione veri.

Autovalutazione

Il voto che mi do è un B-. I motivi sono i seguenti:

- non ho parlato del testing che, sempre se si seguono i principi dell'ingegneria del software, sono facilmente implementabili (al posto di ricevere input da un utente lo ricevo da un test, al posto di emettere output su una GUI lo invio ad un test, ecc...)
- non avevo troppo chiara la differenza sostanziale tra libreria e linguaggio (espressività delle idee del progettista)
- non ho citato molti design-patterns come Observer e Singleton
- manca la tematica sincrónico-asincrono e le conseguenze che le comunicazioni asincrone hanno sul codice (funzioni di callback intrusive) e come mai queste possono fare fastidio