Laboratorio di Sistemi Software

Maicol Forti

2 gennaio 2020

Non c'è codice senza progetto, non c'è progetto senza analisi del problema, non c'è problema senza requisiti.

—Antonio Natali

Indice

1 Laboratorio 1

1.1 Brainstorming

Sezione di brainstorming su come impostare il lavoro iniziale quando ti viene dato da svolgere un progetto.

1.2 Conclusione

Da questo laboratorio è nata la necessità di dividere il lavoro in 3 parti: Analisi dei Requisiti, Analisi del Problema, Progetto.

1.2.1 L'analisi dei Requisiti

In questa sezione bisogna porre domande al cliente per comprendere il sistema. Possono essere domande generali sul tipo di sistema, centralizzato/distribuito, eterogeneo/omogeneo, oppure relative alle componenti. Nel caso delle componenti è sempre importante fare riverimento a tre punti di vista: Struttura, Interazione e Comportamento. La struttura è ad esempio se il componente è atomico. L'interazione è come i componenti interagiscono. Il behaviour è come i componenti si devono comportare.

1.2.2 Formalizzare

Inoltre è necessario formalizzare ciò che viene scritto nell'analisi dei requisiti, in modo che una macchina possa capirlo e sia comprensibile allo stesso modo da umani diversi.

2 Laboratorio 2

2.1 Completare il laboratorio 1

Completamento del laboratorio 1.

2.2 Conclusione

Porre particolare attenzione ai termini usati.

2.2.1 Utilizzo dei termini corretti

Se viene usato un termine nei requisiti bisogna mantenerlo e capirlo. Non tradurre.
Se viene detto ad esempio: "Must Send", non devi tradurlo in "deve inviare". La prima cosa da fare è chidere al cliente cosa intende con "send". Alla fine è importante formalizzare.

3 Laboratorio 3

3.1 32, il disegno alla lavagna

Quando ci si trova ad un qualcosa di cui non conosciamo il significato, o pensiamo di saperlo, è fondamentale non avere conclusioni affrettate.

3.2 Conclusione

Bisogna sempre chiedersi che cosa aveva in mente colui che ha fatto il disegno, bisogna interpretare. Nel momento in cui capisci che fa riferimento ad esempio al mondo dell'informatica, stai assegnando una ontologia, poi lo puoi associare alla semantica. Questo discorso serve a far capire che vale la stessa cosa anche per Robot e Console. Il fatto che i nomi possano ricordarti qualcosa, non vuol dire che per il cliente sia la stessa, quindi è necessario domandargli cosa intende.

3.2.1 Piano di Testing

Il testing che ci interessa principalmente per il problema del robot che affrontavamo è il Functional Testing dei requisiti. Deve fare quello che deve fare. Può essere utile anche un Acceptance Test per verificare. Nel caso del robot per poter fare queste cose è necessario controllare il suo stato. Serve quindi una rappresentazione esplicita dello stato del robot e l'assunzione che il robot mi dica cosa sta facendo. Assumiamo che se il robot ci dice che si sta muovendo, effettivamente lo stia facendo.

4 Laboratorio 4

4.1 Un nuovo requrement: Step

Il nuovo requirement step viene presentato.

4.2 Conclusione

Quando viene posto un nuovo requirement è sempre importante creare un nuovo modello dei requisiti e definire l'Architettura Logica.

4.2.1 L'Architettura Logica

L'architetture logica è un modello eseguibile formale dell'architettura del sistema.

4.2.2 Modello dei Requisiti

Questo sistema ha un solo componente: il basicrobot. Questo basicrobot deve essere in grado di far muovere un virtualrobot in base ad un set di "basic movement-commands" scambiati in qualche modo per internet. La comunicazione è fire and forget e one dispatch at a time. Infine il basicrobot riceve le specifiche in forma testuale.

4.2.3 Analisi del Problema

Cosa intende il committente per virtual robot? Il virtual robot lo abbiamo già, c'è il link alla pagina sul sito per farlo andare. Il basicrobot fungerà da interprete di messaggi per poter far andare il virtual robot.

4.2.4 Le dipendenze tecnologiche

Bisogna sempre fare attenzione ad esse, ci si riserva di eliminarle in futuro. Un esempio classico è il contesto, che è una dipendenza dell'infrastruttura.

4.2.5 Il factory pattern Support

Il support serve a comunicare con altri nodi spostando la dipendenza tecnologica su di se, meglio che niente.

4.2.6 Il file di configurazione

Per non dover scrivere nel modello con chi stai lavorando, si crea un file di configurazione Prolog che viene letto all'inizio per definire il sistema. In questo modo il layer viene alzato di un livello.

5 Laboratorio 5

5.1 Non ho appunti, abbiamo probabilmente fatto il codice del basicrobot.

6 Laboratorio 6

6.1 Simulazione di un processo di produzione

In questo processo sono fondamentali il riutilizzo di software e la technology independence, che però è un pò technology aware. Ricorda di seguire il motto!
Si parte da una traduzione formale capibile da una macchina e dall'impostazione dei test plan che permettono la verifica.

6.1.1 Approccio bottom-up

L'approccio in questione parte dal problema e non dalla tecnologia, che sarà scelta solo in funzione del problema.

6.1.2 L'importanza della giusta astrazione

La macchina di Minsky funziona, ma ha un grosso problema, l'abstraction gap tra il suo livello e quello del problema. Serve quindi utilizzare un approccio a tenaglia che spinge la macchina verso l'alto e il linguaggio verso la macchina.

6.1.3 Modello eseguibile

Come sempre il modello del sistema deve essere eseguibile, il linguaggio naturale è fuffa. Importanza di usare il giusto spazio concettuale. (Figura 3D delle slide struttura, interazione, comportamento)

6.2 Conclusione

6.2.1 QActor

QActor come approccio top-down che usa entità autonome raggiungibili via rete con dei messaggi. Questo è il punto di partenza, entità con flusso di controllo che interagiscono emettendo e ricevendo inormazioni.

7 Laboratorio 7

7.1 basicrobotreq.qak

Se ci sono 2 attività che interagiscono, una si deve adattare, nel nostro caso il robot definisce il modo e il caller si adatta. Il modello qak è un buon meta modello perché adeguato a denotare i modelli del sistema nei vari stage del processo di sviluppo.

7.2 Conclusioni

7.3 L'analisi dei requisiti

Si deve catturare il cosa rimanendo technology independent. Ci sono 3 tipi di scambio di messaggio: dispatch, request, event. Dall'analisi ottieni l'architettura logica.

7.3.1 stop

Abbiamo discusso sul fatto che stop non si deve assumere cosa sia finché il cliente non ce lo spiega, ci manca la semantica. I Test plan servono poi a controllare la semantica.

7.3.2 Pro attività

La necessità di rispondere a stop preannuncia la necessità di un sistema pro attivo, il robot ha reasoning interno.

7.3.3 Problema vs Problematica

Una problematca è più generale, è qualcosa che si ripresenta. Il problema è specifico. Ad esempio la pro attività è una problematica.

7.3.4 L'hollywood principle

Per evitare di utilizzare il polling per controllare ad esempio un Timer, si può usare questo principio che cita: "Don't call us, we'll call you!".

Il timer che viene attivato si prende la responsabilità di avvertirci che è terminato.

7.3.5 Eventi, le supernove

Nasce quindi il concetto di evento, delle supernove che avvengono senza interessarsi di chi le riceveranno.

7.4 Il progetto - Pattern Adapter

Nel progetto è utile utilizzare dei Design Pattern ed eliminare le dipendenze tecnologiche. Un classico esempio è il pattern adapter. Si può creare il robotadapter che si occupa di rapportarsi con il robot particolare alla situazione in base alle configurazioni.

8 Laboratorio 8

8.1 Conclusione

8.1.1 Il pattern adapter

Il pattern adapter sta fuori dal modello, ha una facade/factory/adapter. Viene usato come meccanismo di delegazione. Nel codice lo richiamo come Coded Qactor. Il linguaggio con cui lo fai non è importante.

8.1.2 La percezione degli eventi

Gli eventi sono supernove, vengono quindi percepiti da chi si trova nel sistema, nell'universo.

9 Laboratorio 9

9.1 Python vs Meta modello

Serve piu tempo a fare tutto in python o meta-modello? Era solo un esempio per farci capire che senza l'architettura sotto saremmo noi a dover fare tutto. Ci metteremmo ad usare i thread, poi però nascono i classici problemi di sincronizzazione, mutua esclusione ecc, cose che ora fa l'infrastruttura per noi.

9.2 Eventi a me stesso?

Se emetto un evento lo ricevo? Nella versione 1.2 della libreria non viene fatto perché tanto lavoro come un automa a stati finiti e non lo riceverei.

9.3 Coroutine

Se usassimo Java potremmo avere 1000thread con 1000 attori, ma con Kotlin, utilizzando le coroutine, si può avere un solo thread rendendo il tutto piu efficiente.

9.4 I contesti del sistema

Quando emetto un evento tutti gli altri lo ricevono, anche in contesti diversi e su nodi diversi. Questo perché i contesti fanno parte del sistema. Ci si avvicina al pattern proxy. (Dovresti avere un disegno sul quaderno di questa cosa) Il sistema parte quindi quando tutti i contesti sono attivi. L'infrastruttura sceglie infatti di far partire l'applicazione solo quando i contesti sono partiti. Questo succede perché in distribuito non si parte tutti contemporaneamente.

9.5 External QActor

Si può interagire con attori esterni utilizzando la keyword external.

9.6 robotAdaperQa

Ha un actorybody che funziona quando riceve un messaggio, è quindi message driven. In questo livello i contesti non ci sono.

10 Laboratorio 10

10.1 Request

Quando si deve cambiare il modo di comunicazione bisogna sempre chiedere al cliente chi deve ricevere il messaggio. Se ad esempio solo il destinatario deve riceverlo non si può implementare una request con un evento, perché sennò lo ricevono tutti. Nel nostro caso è l'infrastruttura che tiene traccia delle connessioni con i proxy di chi contattare.

10.2 Layer

La struttura ha layer è utile per nascondere quello che sta sotto, anche se ciò che c'è sopra può ripetere alcune cose di quello che è sotto.

10.3 Differenza tra actor e adapter

Nel basicrobot si possono definire delle politiche, nell'adapter no. Possono parlare anche 2 linguaggi diversi.

10.4 Conclusione

10.4.1 Dipendenze tecnologiche

Dividendo il codice utilizzando adapter e support si evitano le dipendenze tecnologiche sul basicrobot, che rimane indifferente al cambiare del robot da utilizzare ecc cambiando i file di configurazione.

11 Laboratorio 11

11.1 I fogli da consegnare

11.2 Obiettivi da raggiungere nelle consegne settimanali

11.2.1 L'analisi dei requisiti

Tutti devono essere d'accordo su di essi. Il modello che si estrae deve essere il più simile possibile tra tutti. Definisci i componenti e la loro struttura, interazione e comportamento.

11.2.2 L'analisi del problema

Dopo aver analizzato il problema e compreso le problematiche si definisce l'architettura logica. In questa fase è fondamentale convincere chi ha i soldi sugli investimenti da fare. Bisogna quindi presentare le varie possibilità e i costi relativi in base alle tecnologie.

11.2.3 Progetto

In questo caso ci sono gradi di libertà dipendenti dal progettista. Si diversificano le architetture.

11.3 Libreria

11.3.1 Sistema (BLU)

Sono quelle librerie che connettono cose, configurano sistemi.

11.3.2 Algoritmo (ROSSO)

Sono quelle facili da usare dagli automi privi di cervello, noi informatici. Monkas.

11.3.3 Le librerie sono tutte rosse

Tutte le librerie sono rosse, perché danno interfacce user friendly.

11.4 Il Percettrone

Struttura: ha n input (non so quanti, 2-10) e un output.
Comportamento: flusso di controllo autonomo, read-eval-print-loop.
Interazione: sistema asincrono dove output di uno diventa input del successivo.
Ma è tutta fuffa, serve roba eseguibile.

11.4.1 Lo spazio concettuale

Si definisce un vocabolario per denotare lo spazio concettuale del percettrone. Visto che prospettiamo un linguaggio, serve un modo formale per descriverlo. (Sintassi e semantica, semantica è importante) Puoi avere tante sintassi ma la semantica deve essere chiara.

11.4.2 Serve un Mapping

Serve quindi fare un mapping tra i termini del dizionario a qualcosa capibile dalla macchina. Si utilizza quindi un altro linguaggio di cui so la semantica insieme alla macchina. Come traduco percettrone? Oggetto, Actor Akka, Agent? Ci mancano degli aspetti semantici per definire il percettrone, non trovo una corrispondenza 1-1.

11.5 Conclusione

11.5.1 Abstraction Gap

Tutto questo discorso è servito a capire che ogni volta che non troviamo una corrispondenza 1-1 c'è un abstraction gap tra linguaggio. Se non ce ne è uno che funziona 1-1, scelgo quello con gap minore. Il nostro obiettivo quindi è fare un DSL per reti neurali.

11.6 Qactor source/observer

Il QActor lavora sia come stream source che observer.

12 Laboratorio 12

12.1 Bottom-up buco nero

Partire dal bottom-up ci porta al buco nero della tecnologia. Se una agenzia viaggi conosce solo la bicicletta, organizzerà solo viaggi in bicicletta.

12.2 Arduino

Arduino è un sistema stand alone con: input, elaborazione e output.

12.3 Seriale, una rappresentazione

Quando si ha una connessione seriale ciò che trasferiamo sono rappresentazioni, ad esempio di numeri interi. L'importante è sapere una rappresentazione comune. Ad esempio delle stringhe codificate in un certo modo.

12.4 Conclusione

12.4.1 Technology independence quando definisci un modello

Per poter parlare in modo technology independent è necessario definire un livello che maschera l'interazione fisica. (Tutti rispondono che serve definire un protocollo di comunicazione ma non è quello che serve al prof) Si parla quindi di dispatch, request-response, event.

12.4.2 Standard di comunicazione

Ulteriore scelta da fare è di usare uno standard comune a tutti, nel nostro caso l'utilizzo di application message. Mandi rappresentazioni di messaggi sotto forma di stringa. Si ha quindi capacità espressiva a prescindere dal modello qak.

13 Laboratorio 13

13.1 Studio del Radar

Abbiamo visto il radarGui e il disegno con le nuvolette sulla comunicazione.

13.2 Conclusione

13.2.1 Il radar è Pojo

Il radar nasce come pojo, non si preoccupa di prendere informazioni, è il sonar che gliele manda.

13.3 Client-Server

Come core business definiamo una comunicazione diretta che può essere tcp, udp, http ecc. La cosa importante è definire che uno fa da server e uno da client. Il robot è quindi produttore di dati e il radar li consuma.

13.3.1 Il pattern mediator

Gestire il produttore/consumatore in ambiente distribuito eterogeneo senza vedere l'infrastruttura, porta a evocare il pattern mediator. Questo si occupa di far comunicare le parti. Un classico esempio è l'MQTT broker. L'importante è poi gestire le policy e le quality of service. Un ulteriore possibilità è usare CoAP con restful put e get.

13.3.2 Le risorse in CoAP richiamano MVC

Le risorse richiamano un pattern MVC dove la risorsa è il model. Quando leggo put aggiorno i dati del model.

14 Laboratorio 14

14.1 Sonar con dati al Radar

Definire il what: i dati prodotti arrivino al sonar, importante definire come è fatto il dato. Esempio: Rappresentazione di un numero in double. Definire l'how: Delineare il progetto, in base alle conoscenze si descriminano possibilità diverse. Ricordare comunque sempre il motto.

14.1.1 Problematica: Comunicazione distribuita eterogenea

Disegno con la nuvoletta su slide. Ci sono varie possibilità, ad esempio comunicazione diretta o mediata. Si potrebbe usare l'MQTT broker che è gratis. Oppure si può fare un primo prototipo in TCP con un client ed un server.

14.1.2 Il product backlog

Rappresenta i task da dividere al personale. Nel nostro caso ci sono 2 task principali: Creare il server tcp e il client.

14.1.3 Il pattern decorator

Che il server sia sul radar o sul sonar per ora non è importante, sono scelte che si possono rimandare, perché basta aggiungere un "esoscheletro" che aumenta le capacità del radar e del sonar. Questo esoscheletro è il decorator. A livello di analista preferirei comunque che il sonar fosse il trasmettitore perché si evita il polling, ma dipende sempre dal problema. Questa scelta evita anche che il radar debba sapere l'ID di ogni things.

14.1.4 Modello mvc

In questo caso si può vedere il tutto come un modello MVC dove la view è il radar e il model è il raspi.

14.1.5 Conversione di linguaggio

Quando qualcuno deve ricevere dei dati fatti in un certo modo, è sempre lui che li elabora/modifica per farli diventare come gli servono.

14.2 Conclusione

14.2.1 Le domande e la rivoluzione copernicana

Lavorando in questo modo, cioè partendo dalla tecnologia, abbiamo visto esserci varie problematiche che sono messe in evidenza da varie domande che possiamo farci. Se cambio il pattern di comunicazione quanto devo cambiare? Se cambio TCP con UDP o MQTT cosa rimane di cosa ho fatto? Se piu robot devono connettersi al radar funziona?

14.3 La rivoluzione copernicana

Si decide di ribaltare la visione cambiando le interazioni point to point con restfull. Questo ci porta a usare CoAP che è piu light e efficiente.

14.3.1 REST

Rest nasce per dare una semantica rigorosa alle GET (acquisizione info), PUT (Aggiorna stato), POST (Crea risorsa). Si parla di risorsa, quindi a sto punto l'esoscheletro di cui parlavamo prima non serve piu per comunicare. Servirà la risorsa che è denotata da URI. Tutto questo era solo un esempio per farci capire la difficoltà che c'è dal partire dal buco nero della tecnologia e poi cambiare tutto.

15 Laboratorio 15

15.1 Backpressure

Trasformare progetto TCP fire and forget ad uno request response con ACK. Inizialmente in modo sincrono, poi asincrono per far fare altro al server.

15.2 Conclusione

15.2.1 Modellazione concettuale

Questo problema presenta delle problematiche, come l'asincronismo, le request non bloccanti ecc. Dal problema precedente cambia il comportamento sia del client che del server. Sul quaderno abbiamo disegnato vari modelli concettuali per far vedere le possibilità che ci sono. Perché tutto sia fatto in modo semplice senza dover usare nozioni di thread che sono troppo a basso livello il nostro meta modello ci viene in aiuto.

15.2.2 La macchina a stati finiti

Per contestualizzare meglio questo laboratorio, come risultato abbiamo ottenuto la rappresentazione del problema usando la macchina a stati finiti che poi abbiamo implementato mantenendo come struttura quella della FSM.

16 Laboratorio 16

16.1 Da bottom up a top down

Come visto in precedenza partendo bottom up si cade nel buco nero della tecnologia. Per evitare questo si devono usare dei design pattern che fanno in modo che sia semplice passare da una tecnologia ad un altra.

16.1.1 uniboawtsupports.jar

Questa è l'interfaccia protocol independent che ci da il giusto supporto per essere technology independent. Da a disposizione le factory per usare il supporto corretto. (Motivazione del prof: Nessuno ci ha chiesto di farla, ma io la preparo e poi la vendo cara. Ad esempio se dobbiamo passare a bluetooth possiamo dire che ci vuole una settimana, in realtà in 5 minuti lo facciamo e veniamo pagati per la settimana, pog)

16.2 CoAP - Californium

Mondo REST ecc, introduzione sul mondo rest già vista. Leggi le slide. Nota: ogni risorsa non dovrebbe aver stato, ma se lo tieni lo fai lato client. HATEOAS è importante per relazionare le risorse usando i links. Come libreria si usa Californium. Abbiamo un CoAP server con un albero di risorse. Meccanismo observation: se cambi una risorsa, tutti i sottoscritti vengono notificati, è un sistema distribuito.

16.3 Uno standard per i messaggi

Questa cosa è molto importante: se inviamo un qualcosa ciò che si riceve è solo una rappresentazione di qualcosa. Perché ci si capisca, serve devinire uno standard di interazione tra componenti, piu uno standard per i payload.

16.3.1 Termini di alto livello

Per comunicare in fase di analisi è utile avere dei termini di alto livello. Questi termini sono dispatch, request e event.

Dispatch = Interazione in cui uno manda qualcosa, l'altro elabora.
Request = Chiedi l'esecuzione di un servizio.
Event = Emettere info senza una destinazione specifica.

16.4 Gedanken Experiment

Sono esperimenti mentali, non li realizzi sul serio.

17 Laboratorio 17

17.1 Il sistema Unico (Colpa di Bondi)

Se voglio rappresentare un sistema unico è fondamentale che gli automi che lo compongono si influenzino tra di loro con delle azioni.

17.1.1 Attenzione alla rappresentazione

Come sempre abbiamo discusso molto sul fatto che se una transizione di un automa scrivi "a", non si sa cosa sia "a". Magari chi lo manda pensa sia una request, chi lo riceve invece vede altro. Serve sempre uno standard di rappresentazione a livello di semantica.

17.1.2 Un sistema coerente

Un sistema si dice coerente se tutti i componenti trattano le cose allo stesso modo. Esempio: Se c'è "a" come request, per tutti i componenti deve essere una request.

17.2 Rispondere alla domanda: Cosa è?

Esempio: Se ti viene chiesto cosa è un agente, non puoi rispondere dicendo che è una entità autonoma con un goal dentro un ambiente. Questo perché il linguaggio naturale è fuffa. Devi dire di leggersi il manuale. Oppure gli dai una libreria, dei tutorial.
Attenzione anche al fatto che attore, non vuol dire attore akka. Non c'è un concetto globale di attore per il prof, come non c'è per oggetto. Noi intendiamo oggetto Java.

17.3 I Disegni del Prof - Actor Basic e Legenda

Quando si fanno dei disegni bisogna definire bene la legenda per poter capire tutti allo stesso modo cosa dicono. Dovresti avere i disegni sul quaderno spiegati, comunque la pallina con la freccia sono entitò con comportamento autonomo e le cellette rappresentano la coda di messaggi. Questa entità è l'actor basic, da cui specializzano tutte le classi.

17.4 Sistema confinato

Sistema in cui tutte le attività sono nello stesso thread.

18 Laboratorio 18

18.1 Sprint Review

Ogni team fa la review, insieme al PO, e poi si fa il software refactoring se serve. La cosa importante è fare una autovalutazione del prodotto fatto.

18.1.1 I modi per svilupparo il problema dell'automa a stati finiti

Sulle slide li trovate, tutti i modi del prof a partire dal modo piu naive a quello piu completo. Era solo per autovalutarci in rispetto a cosa avevamo consegnato.

18.1.2 Le transizioni

Cosa importante dei passaggi del prof, è che da una FSM a una HL-Comm cambia una cosa fondamentale, le transizioni devono stare fuori dagli stati. Infatti nel QAK sono fuori. Le transizioni vengono infatti eseguite alla terminazione dello stato.

18.1.3 Lo standard MsgUtil

L'altra cosa fondamentale è lo standard dei messaggi a livello applicativo.

18.2 La polvere sotto al tappeto

Nelle architetture a livelli si ha una gerarchia di tappeti con un sacco di polvere. Il vantaggio è che la polvere non la vedo, ma allo stesso tempo non so cosa ci sia sotto. Questo pone la problematica di scegliere se farsi un sistema da solo o farlo fare. Se lo fai fare devi avere fiducia.

18.3 Il neo assunto

Alla luce di tutto ci chiediamo quale approccio di quelli affrontati sarebbe migliore spiegare ad un neo assunto per farlo imparare velocemente. L'approccio è quello con piu benefici e minimo costo.

19 Laboratorio 19

19.1 Transizioni non bloccanti - Disaccoppiamento input

La discussione del laboratorio precedente fa nascere una problematica: la transition fa una receive bloccante, ma se uno stato ha piu transizioni? Bisogna disaccoppiare la tematica dell'input in modo che la transizione non sia una attuazione dell'input, ma l'interesse nel ricevere una informazione.

19.1.1 La select - message based

La select permette di esprimere una intenzione, in questo modo hai messaggi che arrivano indipendenti dal canale. Questo è possibile perché ho un server per canale che si occupa di raccogliere i messaggi e metterli nella coda dell'attore. Si passa quindi da message driven a message based. (Guardie su transizioni)

19.2 Custom Language

Interni: Linguaggio costruito con la sintassi interna, ma che sembrano costrutti built in che in linguaggio non ha.
Esterni: Sintassi reinventata. (Utile per non far percepire la base all'utente finale)

19.3 Il contesto

L'application designer non deve scrivere dettagli legati alle connessioni client/server, serve qualcuno che sappia i componenti per nome, per questo nasce il contesto. Ll contesto serve per mappare il nome a ciò che server per poter comunicare. (Ha fatto l'esempio del DNS)

19.4 Conclusione

19.4.1 Il DSL

Questi ultimi laboratori sono serviti a fare un processo che ci ha portato a varie problematiche ed esigenze. Queste necessità ci hanno portato a costruire il DSL che usiamo. Il DSL ci permette di concentrarci sull'obiettivo, il cosa, e non sul come. Il DSL ci permette anche di cambiare infrastruttura, ad esempio con mqtt, utilizzando un semplice flag +mqtt vicino al contesto.

20 Laboratorio 20

20.1 Il progetto finale

Per il progetto finale bisogna consegnare 3 cose: Analisi dei Requisiti, Analisi del Problema, Progetto. (Mi riservo in futuro di confermare)

20.2 Discorso con il nostro gruppo - Eventi

Visto che noi avevamo fatto in fire forget, il prof ci ha posto una problematica: ha senso che tutti debbano sapere gli indirizzi degli altri ecc? Se la collina esplode noi la vediamo esplodere, ma non ci siamo mica iscritti a lei. Per questo ci sono gli eventi. Gli eventi vengono propagati automaticamente a tutti i componenti del sistema.

20.3 Le Pipe

L'actor basic può fungere da streamer, se qualcuno si è sottoscritto lui gli spara i dati. In questo modo si fanno le pipe, dove chi si sottoscrive può essere a sua volte streamer. Un esempio è il filter, utile per evitare i dati spuri del sonar ecc, oppure il logger, usato prima di filtrare.

20.3.1 ForRadar

Forradar fa da cima della catena, riceve i messaggi filtrati. Però non sa fare emit nonostante sia un ActorBasic. Questo perché non ha un contesto, quindi lo fa fare al suo owner.

20.3.2 Due tipi di eventi

Le pipe portano ad avere 2 tipi di eventi: quelli propagati sulla catena locale, quelli propagati a tutti nel sistema.

20.4 A quale livello si parla di queste cose? L'analisi del problema

È importante notare che la necessità dei filtri, di usare le pipe ecc, nascono in fase di analisi del problema. Non sono cose che sono presenti nei requisiti, ma l'analista sa che sarebbe bello averle, senza porsi il problema di come poi si possano implementare, sarà il progettista a farlo. L'analisi del problema è l'analisi del problema, non come voglio risolverlo. Poi è sempre fondamentale fare il modello eseguibile.

20.4.1 Pro e Contro in analisi del problema

Il compito dell'analista è studiare i vari casi possibili e stilare i pro e contro così che poi si possano fare le scelte corrette. (Anche dal punto di vista economico)
Ad esempio nel nostro progetto ci sono 3 scenari di comunicazione: Request, Dispatch, Event. Studiando questi pro e contro nascono poi le necessità viste in precedenza.

21 Laboratorio 21

21.1 Il Pojo

Abbiamo un radar se ci viene dato il pojo? No, perché il pojo è un componente software usabile solo se si ha un riferimento e attraverso procedure call. Posso usarlo in un sistema, in un environment/virtual machine e in un linguaggio, difficile quindi in remoto. Di conseguenza non nel nostro sistema che è distribuito ed eterogeneo.

21.1.1 Il radar

Il radar quindi è un qualcosa a cui accedo via rete, sia che sia rest, http, udp, tcp ecc.

21.2 Sintassi e semantica messaggi: interpretazione

Indirizzo IP, Porta, Protocollo, LA SINTASSI E SEMANTICA DEI MESSAGGI. L'ultima cosa è fondamentale così che chi riceve il messaggio possa INTERPRETARLO. Il radar ha un interprete

21.3 Interprete

Ricevo una stringa, sequenza di caratteri, la prendo e la passo all'analizzatore sintattico (Parser). Il parser guarda la sequenza e vede se è valida, deve essere in accordo alle regole sintattiche definite. Il parser passa poi il tutto all'EXECUTOR.

21.3.1 L'executor

L'executor sa eseguire le frasi scritte bene. Lui però non deve ricevere la stessa stringa del parser, perché sennò deve riguardarla tutta per ricavare le informazioni.

21.3.2 Gli application message

Ciò che viene passato all'executor è un application message, una classe con vari gettere per ottenere le informazioni dalla stringa. Se abbiamo bisogno di confezionare una stringa basta fare il procedimento contrario: crei un application message passsando i singoli campi e poi invii come messaggio la toString.
Nel nostro caso il campo msgType decide il tipo di comunicazione: request, dispatch, event.

21.4 Approccio top down o bottom up

Con tutte queste informazioni ora si può decidere, o si parte bottom up facendo socket ecc., oppure vai top down con i QActor.

21.4.1 Un prototipo immediato

I QActor permettono di avere un prototipo immediato che nasconde i dettagli tecnologici.

21.4.2 Componenti amici vs alieni

I componenti amici sono quelli fatti in QActor con cui possiamo comunicare sapendo il nome.

21.4.3 Il coded QActor

Nel caso che un componente sia alieno serve un CodedQActor che dentro gestisce le cose a basso livello e me le fa vedere a livello di QActor.

21.5 Conclusione

21.5.1 Kono Giorno Giovanna niwa yume ga aru - Technolgy independence

Noi abbiamo un sogno, scrivere codice technology independent. Un esempio è usare gli eventi per evitare di usare External. Oppure usare MQTT per ricevere piu eventi.

21.6 Architetture Layered

Per ora stiamo usando una architettura layered, vari layer nel sistema tra cui la infrastruttura. Si passerà poi a quelle esagonali.

22 Laboratorio 22

22.1 Ritorno allo smart robot

Siamo ripartiti dallo smart robot del lab 3 per fare vari discorsi.

22.1.1 Testing automatico senza informazioni

Come è possibile fare un testing automatizzato se non si hanno informazioni? Serve della conoscenza.
Il modo piu semplice per farlo è usare delle risorse accessibili in modo restfull. Queste risorse vengono aggiornate dal robot e poi possiamo vedere lo stato. Da notare che comunque dobbiamo assumere che il robot faccia effettivamente quello che poi dice di fare aggiornando la risorsa.

22.1.2 MQTT aggiuntivo

Dall'aggiornamento della libreria mqtt può essere usato come modo aggiuntivo per comunicare, non precludendo l'interazione con la porta di ingresso.

22.1.3 Come comunicano i componenti?

Guarda i modelli di progettazione. (QAK)

22.2 Riproduzione dell'esercizio smartrobot

Visto che sono da usare 3 progetti diversi per aiutare chi dovrà provare sta roba scrivo dove sono le cose che servono.
Reinstallate plugin per generare i giusti .pl con dentro MQTT diverso da "0" e mettete le librerie nuove.
Avvia basicrobot che sta in basicrobot2020.
Avvia smartrobot che sta in basicrobotusage.
Avvia la console python che sta in smartrobot2020.

22.2.1 Lo smartrobot e la guardia

Lo smartrobot ha una transizione con guardia su cui decide cosa fare in base alla guardia. Serve per comportarsi in maniera diversa se vogliamo lavorare con le risorse o meno. C'è un observer della risorsa che si updata ogni volta che la risorsa cambia.

22.2.2 La risorsa: due modi per vederla

Una risorsa del nostro progetto può essere vista in due modi: qualcosa che osservi e aggiorni a livello di stato (comandi con la "u" davanti), qualcosa che aggiorni a livello fisico.

22.2.3 Da IoT a WoT

Si ha una interfaccia restfull standard con get, put ecc. La nostra risorsa è una mind con rappresentazioni, ma noi dobbiamo fare in modo che ci sia corrispondenza con il mondo fisico. Gli attuatori devono essere attivati in corrispondenza ai comandi relativi.

22.2.4 Possibilità di usare vari canali

Lo smart robot ha vari ingressi e puoi decidere quali usare per comunicare.

22.3 Conclusione

22.3.1 Riassunto finale

Ho un servizio (qualcosa di accedibile dall'esterno in modi diversi). Non voglio cambiare il componente sia che ricevo da tcp, risorsa ecc. Si può fare testing grazie allo stato.

22.3.2 Attivare la risorsa

Si può fare o dallo smartrobot, oppure potrei attivare solo coapsupport e crearla dall'esterno. Meglio da dentro o mi scordo. L'attore alza le

interfacce verso l'esterno. Potrei anche togliere MQTT e dare una risorsa ad ogni attore, ma sarei costretto a limitarmi al protocollo restfull.

22.3.3 CoAP

CoAP è nato perché rest ha una infrastruttura troppo onerosa per Arduino.

23 Laboratorio 23

23.1 Produttore Consumatore

Come si fa un sistema prod/cons in kotlin?

23.1.1 Un Sistema

Quando si parla di sistema bisogna sempre parlare di Architettura, di conseguenza di struttura interazione e comportamento.

23.1.2 Tematica sincrona/asincrona

Questa tematica serve solo con il dispatch, con la request non serve. L'asincronismo fa ribrezzo perché non c'è quello assoluto, per esserlo dovremmo assumere capacità infinità del fiume con le barchette.

23.1.3 La convenzione - Il buffer

Per convenzione il caso sincrono è sinonimo di unbuffered, quello asincrono di buffered. Questa è l'analisi del problema del prod/cons.

23.1.4 Interazione in prod/cons

L'interazione può essere diretta o mediata. Nel caso mediato il produttore non vede il consumatore, ma vede il buffer che si comporta da middleware.

23.2 Prod/Cons in Java

Usi 2 thread, nel caso asincrono usi MQTT e gestisci la backpressure.

23.3 Kotlin

Kotlin mi offre un modo per organizzare il produttore consumatore, però non ne abbiamo parlato. Kotlin si usano top-function che non sono metodi di oggetti. Sta abbracciando il paradigma funzionale in maniera

tipizzata.
Puoi mettere le lambda fuori dalle parentesi: "Sorbole che volpata!". "it" denota un argomento default, funziona però solo con un argomento.

23.3.1 Continuation passing style

Consiste nel fare quello che devi fare quando hai finito di fare quello che stai facendo.

23.3.2 Le chiusure lessicali

Kotlin usa le chiusure lessicali che risolvono i simboli in base allo scope. (a differenza di quelle dinamiche)

24 Laboratorio 24

24.1 Attenzione: Non ero presente

A questo laboratorio non ero presente, i seguenti appunti sono colpa di Christian Serra, prendetevela con lui se manca roba LUL.

24.2 Classico riassuntone

Quando viene richiesto un progetto si consegna: Progetto, Analisi del Problema e Analisi dei Requisiti. Fatto questo servono dei test plan, una pianificazione almeno.

24.2.1 I test plan

I test plan sono quasi una conseguenza dell'analisi dei requisiti.

24.3 Formalizzare

Come sempre bisogna formalizzare tutto in modo che la macchina possa capirlo. Bisogna quindi consegnare un testo formale che contiene un modello espresso come rappresentazione testuale del progetto, deve essere eseguibile.

24.3.1 meta modello

Si scrive in linguaggio custom riusando varie conoscenze proprie. Il modello sarà una istanza del meta modello. Questo modello si può costruire o bottom-up unendo piccole parti utili o top-down in modo da arrivare al codice il piu tardi possibile. L'approccio piu usato da noi è quello a tenaglia, un po e un po.

24.4 Cosa è un QActor

Il QActor è il concetto fondamentale del sistema qak, questi qactor interagiscono senza memo condivisa. Questo concetto è definito in modo formale nelle software factory che lo implementano sfruttando la libreria.

24.4.1 Perché i QActor

Durante l'analisi è nata una pressione che mi ha spinto a non usare gli oggetti in un sistema distribuito eterogeneo. Serviva qualcosa per cattuare meglio gli aspetti e le entità essenziali.

24.5 Il disegno

Ha fatto vedere un disegno generale in cui ci sono QActor, mqtt, tcp ecc.. I tondi rosa sono i QActor, i tondi azzurri le porte TCP, i rettangoli arancioni sono MQTT. Infatti i QActor possono comunicare o con TCP o con MQTT.

24.5.1 La legenda

Fondamentale oppure siamo considerati cioccolatai e non ingegneri.

24.6 2 Parti

24.6.1 La business Logic

Questa sta tutta nello smart robot, il basicrobot è uno slave.

24.6.2 Le parti per fare HMI e MMI

Senza questa parte il sistema è autistico (non comunica).

24.7 CoAP

Un altro modo per accedere allo smartrobot è usando CoAP, rappresentato dal rettangolo verde. Lo usiamo per mostrare all'esterno una risorsa RESTful e perché è leggero rispetto a HTTP usato in modo RESTful.

24.7.1 Le pipe

I dati prodotti dal sonar vengono filtrati attraverso delle pipe in modo reactive, la cima della pipe sparerà poi gli eventi: obsatcle, polar. Questi

sono percepiti sia da smart che da basicrobot e anche dai componenti alienti sottoscritti alla topic MQTT corretta.

24.8 Node

Il server node non vogliamo riceva questi eventi MQTT (croce rossa), perché è un dettaglio implementativo. Per questo motivo faccio un cambio di paradigma a REST per evitare di mostrare i QActor e MQTT a Node.

24.8.1 socket.io

Lo smartrobot ha un observer per controllare la risorsa, aggiornata automaticamente da un CoAP observer, un protocollo. Il server invece capisce solo request-response, non si aggiorna automaticamente, per questo aggiungiamo la tecnologia socket.io.

24.9 FUFFA, facciamo una demo

Per ora è tutta fuffa, per questo è importante fare una demo, in cui vengono anche date tutte le informazioni sul depoly del sistema. Per ora la demo si fa a mano ma potrebbe essere automatizzata.

24.9.1 Come eseguire

Si avviano le solite cose, mosquitto, virtual robot, il jar del basicrobot, poi potremmo far partire smartrobot e COAP, anche separatamente. Infine si avvia il frontend server node, si fa npm install e si fa partire il frontendServer.js e si va a coap://localhost:5689 indicando la risorsa COAP. (Mi dicono non funzionasse molto bene)

24.10 Conclusione

Si poteva fare questa architettura solo bottom up? Siamo partiti da Kotlin perché ha le coroutine e non ci preoccupiamo di gestire i thread. Si parte dagli ActorBasic che sono piu in dettaglio delle coroutine, channel e actors.

24.10.1 Map

Nuova idea malsana, fare la mappa della stanza attraverso il robot e poi farlo muovere a varie coordinate.

25 Laboratorio 25

25.1 Attenzione: Non ero presente ma ho seguito live da casa su Discord

(Grazie a Casta per il microfono LUL)

25.1.1 Power Point

Il prof ha quasi voglia di farci fare una presentazione che ha 3-4 fogli, indovinate quali? Bravi! Progetto, analisi dei requisiti e analisi del problema. L'ordine dipende dall'esposizione.

25.2 Robot, Again

Questa lezione è stata un altro grande riassunto. Tutto nasce dalla domanda di Aguzzi che non può fare a meno di chiedersi se l'architettura del prof sia nata top-down o bottom-up. Per rispondere alla domanda, ci vorranno 3 ore...

25.2.1 Criteri di progettazione

Il prof ha voluto separare la business logic dalla presentation logic. Il core business è la parte fondamentale e sta nello smartrobot. L'idea del prof è usare una manovra a tenaglia partendo prima top-down, definendo le cose fondamentali, concretizzando poi con un prototipo. Questo permette di avere una business logic confrontabile con il committente in sprint review. Non farti problemi di interfaccia, anche se il cliente non sarà contento. (Viroli non è così d'accordo).

25.3 Il prototipo

Per poter arrivare ad un prototipo bisogna farsi delle domande: cosa è sto smartrobot: funzione, oggetto, attore, agente di Omicini, altro? Ci rifiutiamo di vederlo come oggetto perché da requisito comunica via rete

e deve generare info. A questo livello TCP ecc non è roba importante, non ci si sbilancia sul come, ma sul cosa. Certo potresti partire da un oggetto o da una funziona, ma potresti anche partire da Minsky, capisci bene che non è molto sensato, c'è troppo abstraction gap. Una cosa utile è avere una tecnologia che abbatte i costi di prototipazione.

25.3.1 Il disegno informale

Rappresentiamo quello che pensiamo che questo componente debba essere. Fai vedere che manda e riceve info. Questo non è comunque abbastanza, serve farsi altre domande. Cosa è un comando? Non poniamoci il problema di come mandarlo, ci pensa il progettista al come.

25.3.2 Un comando

Un comando è una entità che ha come rappresentazione una stringa, di cui da requisito sappiamo la sintassi: w,a,s,d,step,stop. Quale è la forma di interazione che ci permette di mandare i comandi? (request-response, dispatch...) Ma parlando di questo sai già che c'è un canale di comunicazione, ma come faccio a mandare questi bit all'interessato?

25.3.3 Canale

In questa fase si è discusso se l'approccio request-response fosse l'unico o se ci fossero alternative, ovviamente ce ne sono, tipo l'hollywood principle. Queste scelte sono da discutere in analisi, prima della progettazione, si scrive cosa si preferisce e il perché. La cosa fondamentale è spiegare le motivazioni.

25.3.4 request-response

Quando si pensa di scegliere questo modello bisogna porsi sempre il problema di cosa rispondere. Se la risposta è un semplice ACK, allora forse non è una buona idea, perché non ti da comunque certezza del fatto che il ricevente abbia fatto ciò che gli hai richiesto, ma solo che ha ricevuto la richiesta. Per questo è meglio controllare gli effetti.

25.3.5 Controllare l'effetto, Risorse

Per controllare gli effetti, da analista penso che ci debba essere qualcosa nel sistema che posso interrogare per capire le informazioni sullo stato. In questo modello vengono mandati dei dispatch che cambiano lo stato del mondo. Ora ho qualcosa da qualche parte a cui chiedere lo stato del robot. Gli effetti dei comandi e i dati sensoriali sono tutti riflessi in un nuovo componente, distinto dal robot, che rappresenta lo stato del mondo, questa base di conoscenza è una risorsa, che può essere addirittura osservata. Attenzione: tutto questo discorso è campato in aria, nel senso che è in analisi, non so se sta roba esista o meno.

25.3.6 Riflettere il mondo reale

La risorsa non è solo un punto che ci da informazione, ma a cui anche mandare i comandi. Se mando w ad un front-end che rappresenta un rapporto tra me e il sistema, voglio che poi w sia riflesso anche nel mondo fisico. Per questo non solo dovrà aggiornare lo stato, ma attivare anche gli attuatori fisici.

25.3.7 Le problematiche

Tutti questi scenari non sono cose speciali del nostro case study, sono cose generali che possono essere utili in piu situazioni.

25.3.8 Diritto di comunicare

Del diritto di comunicare con la risorsa non se ne è ancora parlato, tanto sei un analista, tu lo segnali, poi ci penserò qualcun altro.

25.4 Via la Fuffa, formalizzare

Qualcosa che sia comprensibile da una macchina, serve un linguaggio di modellazione, non di programmazione.

25.4.1 Non un linguaggio di programmazione

Questo tipo di linguaggio non va bene, perché ci trascina ad un livello troppo basso per questo livello di analisi. Ad esempio UML si, si potrebbe usare, ma alla fine ha lo stesso handicap di Java, si pensa in oggetti.

25.5 Manovra a tenaglia

Si parte da un punto di partenza, tipo kotlin, e si cerca di costruire qualcosa che mi porta verso la mia idea in modo incrementale.

25.6 ActorBasic

Nel nostro caso si usa una classe ActorBasic su cui viene wrappato un attore. Il nostro sistema software è fatto da questi elementi. Per sapere cosa sono devi leggerti il codice e il manuale d'uso. Un ActorBasic lavora in modo message driven e lo puoi contattare attraverso il suo riferimento, senza però iniettare flusso di controllo, è una entità autonoma. Il messaggio va semplicemente in una queue, sarà l'attore a decidere quando leggerlo.

25.6.1 Da message driven a message based

Il prof fa fatica a lavorare message driven, ogni volta che ti arriva un messaggio devi chiederti cosa stavi facendo prima e se tu possa eseguire attualmente il messaggio arrivato.

25.6.2 ActorBasicFSM

Per questo motivo il prof ha creato ActorBasicFSM, un actor che lavora in modo message based e a stati finiti.

25.6.3 I messaggi

Ogni attore usa dispatch, request, event, e per ognuno c'è una semantica e una sintassi che lo caratterizza. In particolare si creano gli application message, in questo modo l'interazione è standardizzata.

25.6.4 Il contesto

Un attore può far parte di una entità piu ampia chiamata contesto, è utile per poter parlare con qactor di altri mondi, altri contesti.

25.7 Architettura logica

Avendo tutto questo di cui abbiamo parlato, possiamo fare un prototipo? Certo, possiamo fare la nostra architettura logica.
Importante decidere se lo smartrobot sarà un ActorBasic o un ActorBasicFSM, FSM sembra la scelta giusta perché per requisito certi comandi devono essere ricevuti in certi stati. (stop durante step)

25.7.1 Le entità aliene

Come fanno le entità aliene a ricevere informazioni? O si fa request-response o hollywood principle come sempre. Per i motivi visti in precedenza, va bene l'hollywood principle per cui prima con un dispatch mi registro, poi ricevo sotto forma di dispatch i dati richiesti.

25.8 Descrivere un behaviour

Dopo aver sceto ActorBasic o ActorBasicFSM bisogna modellare il behaviour. Questo risulta facile in FSM. Nel caso della richiesta di registrazione dovrebbe essere ricevibile in ogni stato, ma poi quando rispondo mandando i dati? Può essere utile una altra entità, questa in s0 chiede i dati al sonar, se ci sono li manda, poi devo rilasciare il controllo. (No while true o non ricevi piu nulla) Nel caso di un FSM puoi usare un auto messaggio per cambiare stato e poi tornare all'inizio.

25.8.1 Dividi et Impera

Sempre corretto dividere le cose in piu attori. Le tematiche degli attuatori/sensori vengono gestite dall'adpater, modellato come attore, così sono

technology independent. Visto che gli ActorBasic possono essere reactive, creo un adapterStream che crea un ActorBasic che prende i dati e li manda.

25.9 Obstacle e Eventi

Gli obstacle devono essere mandati a tutti quelli che sono registrati per riceverli. Partendo bottom-up inizia comunque a nascere una esigenza di evento. Il prof ha usato uno stile funzionale con le pipe che funzionano in base ai dati emessi e li filtrano. Anche se obstacle non è magari nei requisiti, è comunque fondamentale da analisi, è un dato semanticamente proiettato a dirmi che è una distanza talmente piccola da dovermi fermare. (vedi sonardsh.kt per filter)

25.10 Il Basicrobot

Questo livello in piu serve per distinguere i comandi base da quelli piu complessi, di piu alto livello. Per questo questi comandi sono dati in responsabilità ad altri attori. Gli attori tanto non costano niente, sono coroutine, non instanzi dei thread.

25.11 Il metamodello è nato

Dal momento in cui uno utilizza ActorBasic, sta usando qualcosa che non è una funzione od un oggetto, è qualcosa nella testa di qualcuno usabile con una libreria. Per questo è nato il metamodello.

25.12 Subsumption | Sussunzione

Questa architettura dove si usano piu attori per non sporcare quelli che già funzionano è detta a sussunzione. Ci sono vari layer che rimandano a quelli sotto. Nasce qui l'idea di un nuovo layer, quello CoAP per le risorse.

25.12.1 CoAP per il test-plan

Fin dalla analisi CoAP nasce come necessità perché serve un modo per effettuare il test-plan, così posso esprimere ciò che accade nel sistema attraverso la risorsa. Da ricordare è che è importante che ciò che c'è nella risorsa sia riflesso nel mondo reale, sarà per questo la risorsa che manda i comandi allo smartrobot, il quale li fa arrivare al basicrobot, il quale passando per l'adapter li fa arrivare agli attuatori fisici.

25.12.2 La pagina web

Come ultimissima cosa creo una pagina web per accedere a ste cose. Questa necessità nasce comunque in analisi, il modo tecnologico poi sarà però nel progetto.

25.13 Conclusione

Siete ready boiz?

25.13.1 La risposta ad Aguzzi

Tutti i componenti del sistema sono nati in fase di analisi dal punto di vista logico. Questo perché dai requisiti capisco che devo dividere le responsabilità. La business logic sta nel basicrobot, e questo nasce sempre dall'analisi. L'opportunità di avere una risorsa nasce sempre dall'analisi perché so che esistono le strutture esagonali. Il front-end anche è da analisi perché tanto so che tutti hanno la possibilità di accedere ad un browser, quindi è utile. Per di più in fase di analisi posso già prefigurare il deployment del sistema.

25.13.2 Deploy

Solita roba: mosquitto, i jar ecc ecc ecc...

25.13.3 Progetti futuri - Planner

Da analisi del problema evoco la possibilità/necessità di avvalermi di un tool di conoscenza artificiale, un pianificatore. Questo può dire ad un

robot che si trova in una certa posizione, di muoversi ad una coordinata diversa. In questo modo data una mappa puoi muoverti da punto A a punto B.

25.14 Case Study

Fondamentale: La metodologia con cui ci si aggancia ad un problema è quello che dobbiamo imparare. Dato un problema random, il modo in cui lo approcciamo fa la differenza.

26 Laboratorio 26

26.1 La tecnologia XText

Capire la volpata dei progettisti di XText, un modeling framework di eclipse che origina da UML 2. Si tratta di un fratello di UML. Creato il progetto, ciò che ci interessa è la cartella generata, il resto è autogenerato, ma comunque modificabile quando servirà. Questi file .xtend sono del codice scritto in modo piu veloce di Java, esprimendo le cose in modo piu dichiarativo.

26.1.1 Gli elementi del codice

Nel codice di base sono definiti Model e Greeting. Queste sono definizioni di elementi non terminali, le variabili sintattiche. Sono in Backus-Naur Form Estesa (EBNF), e fanno riferimendo alle gramatico context free (CFG), riconoscibili dalle pile. Quello scritto tra gli apici è una keyword.

26.1.2 Lo scopo

Ogni grammatica ha uno scopo. Lo scopo è il primo non terminale che si trova nel codice, nel nostro caso è Model. Tutte le frasi deriveranno da questo seme iniziale in base a certe regole di riscrittura.

26.1.3 La semantica

Abbiamo un linguaggio, ma è importante dargli una semantica. La semantica è il mapping tra una frase del nostro linguaggio ad Tecnicamente questo si fa generando del codice. Per poter generare del codice mi serve un compilatore.

26.1.4 L' AST

L'AST è l'albero sintattico che rappresenta gli elementi strutturali di una frase ad oggetti. In pratica viene fatto questo: si esplora l'AST in un certo ordine, e man mano che si trovano le frasi si esegue il mapping. La radice dell'AST si potrebbe dire che è Model, ma piu in particolare è una

opportuna rappresentazione scritta in Java di quello che la macchina può rappresentare a partire dalla definizione che abbiamo dato. Il Model è la rappresentazione in un linguaggio di programmazione del concetto che esprime. L'AST può non avere figli nel nostro caso, ma se li ha li referenzi con l'attributo greetings += che mappa ad una lista.

26.1.5 La generazione di codice

Se la lista ha degli elementi, prendi il primo, vedi di che tipo è il figlio e quanti attributi ha. Si prende quindi il figlio con il suo attributo e lo si manda alla macchina. Il front-end sarà quindi il parser che produce l'AST, il backend lo esplora e genera codice.

26.1.6 Grammatica del prof (iot)

Vedere il progetto xtext iot del prof. L'unica cosa da ricordare è che ci sono 2 spazi differenti, quello in cui scrivi il codice xtext e quello dell'application designer, colui che userà il linguaggio.

26.2 Model Checking

Fatto un modello, l'ho espresso in maniera formale. Ma avere un modello sintatticamente corretto non vuol dire che sia semanticamente corretto. Il model checking serve per validare la semantica definendo delle regole di validazione nel package validation. Queste regole sono dichiarative, non devi specificare dove vengono usate. Un esempio è mettere una regola per cui le porte devono stare tra 1000 e 3000, altrimenti mandi un warning/error.

26.2.1 Generatori

Il prof consiglia di non aggiungere roba nel generatore per non contaminarlo, ma di fare altre classi che delegano. Dentro IotGenerator ci sono alcune cose interessanti: la Resource, che è l'entry point dell'AST, IFileSystemAccess2 che è l'entry point del file system dell'application designer. GenIotSystem è la mia classe di generazione, chiami la doGenerate e passi il system, che è il puntatore alla radice dell'AST.

26.3 Cosa bisogna definire

Quando si hanno dei vincoli di specifica bisogna ricordarsi che l'ordine è importante. Ciò che noi dobbiamo definire sono la struttura, il comportamento e l'interazione. Un esempio di comportamento è una classe kotlin con un certo body.

26.3.1 Microservizio è un piatto

Un microservizio è un piatto che casca e si spezza. I pezzi sono modulari e intercambiabili ma se rivoglio il piatto è difficile. Per questo l'architettura è da vedere prima del codice. Per poter ricongiungere il piatto rotto si usa l'interazione.

26.4 Sfruttare i QAK

Se ad esempio volessimo avere dei microservizi autonomi che interagiscono tra loro, potrei vederli come degli attori. Un attore che fa riferimento ad un microservizio avrà il nome del microservizion e nello stato iniziale eseguirà il body del microservizio. Facendo questo, posso semplicemente generare un QAK, così in automatico mi genera tutto quello che serve per farlo girare. Si crea quindi del codice che funziona, poi lo facciamo generare al generatore lasciando all'application designer solo quello che gli serve.

26.4.1 Distribuzione

Export, plug-in development, deployable fragment, selezioni quello che vuoi, crea plug-in. Il vantaggio è che se qualcosa non va devo solo modificare la software factories e fare una nuova release.

26.5 Conclusione

26.5.1 Towards Software Factories

Tutta questa lezione è per farci poi capire la slide in cui si trova l'XText dei QAK. Troviamo qui l'organizzazione dei QActor, hanno il context tra

le quadre perché fanno riferimento a un context che deve esistere. Un tasto utile è ctl-spacebar che ti autocompleta cose. La cosa da notare è che bisogna usare una sequenza sensata, perché le cose definite prima possono essere obbligatore per le definizioni successive. Un esempio di validazione è quello dello stato initial, che senza validazione potrebbe essere usato piu volte in un attore.

26.5.2 Cambio di stato

Ogniqualvolta si cambia uno stato, il messaggio corrente viene buttato. Questo messaggio è quello che viene stampato con printCurrentMessage.

26.5.3 Zooming

Guardare il video dei tizi che fanno picnic. Con noi si correla perché il picninc è l'applicazione, la tecnologia sta bottom, nelle particelle. Ad un certo fattore di scala corrisponderanno certi concetti. L'informatica lavora in mondi virtuale per cui, se parti dalla fisica, dalle particelle, è difficile costruire gli esseri umani. Detto in modo piu chiaro, se parti bottom up ti perdi nella tecnologia. Per questo si parte top-down per poi portarsi ai livelli sottostanti che decideremo noi. Negli ultimi tempi anche i livelli bassi sono relativamente alti, basti vedere IaaS e PaaS come bottom.

27 Laboratorio 27

27.1 Utilities per la consegna finale

27.1.1 IL frontend

Si può trovare un esempio di front-end fatto in node per interagire con il robot all'interno del progetto frontend20. La cosa assolutamente importante è che nel frontend non ci sia nulla relativo alla logica applicativa. Poi il prof ha detto che un pochino ce ne è perché se faccio w effettivamente lo stato del robot deve cambiare, però lo si fa per delegazione, passando il compito a qualcun altro.

27.1.2 L'architetture generale

Vedi l'immagine sulle slide. La logica starà sul basicrobot, la CoAP resource fa da mediatore e il resto è front-end.

27.1.3 Il planner

Il robot è un animale prudente, fa un passo alla volta e poi ricalcola il piano da eseguire. L'idea è che se il piano fallisce, segni la casella come occupata e ricalcoli.

27.2 La consegna finale

Per vedere i dettagli dei requisiti vedere le slide del prof. L'idea è avere un robot che sniffa la plastica e la raccoglie. Sarebbe carino anche fare interazione con un altro robot che raccoglie il vetro. Riempito il proprio bidone, lo svuota ad un bidone della plastica e poi torna a casa. In piu c'è un sensore per la qualità dell'aria per determinare una situazione di allarme. (In realtà questo sensore è simulato, potrebbe essere qualsiasi cosa: temperatura, umidità ecc)

27.2.1 Il software

Dovremo avere del software che gira sul robot e sul bidone. Il contenitore sarà smart, accessibile via CoAP, così da poter sapere se è vuoto o meno.

I calcoli si fanno in quantità di bottiglie. Serve anche un front-end per comunicare con il robot, uno smart device come uno smartphone android. (In realtà basta fare sito accessibile da cellulare, non serve apk apposta, anche se a bologna gliela hanno fatta)

27.2.2 Condizioni di esercizio

Si devono simulare le condizioni di esercizio, se richiedi un comando che non può essere eseguito, il robot deve dirtelo.

27.2.3 Attenzione ai requisiti

I Requisiti possono essere ambigui, inconsistenzi, incoerenti, per questo il primo passo è l'analisi dei requisiti.

27.3 Gli Sprint

Si lavora idealmente in team di 3 persone, si presenta un workplan come risultato dell'analisi dei requisiti e del problema, includento dei test-plan significativi. Quando ci troveremo con il prof per gli sprint sarà necessario avere già un prototipo eseguibile. Serve quindi esprimere una architettura logica espressa come modello eseguibile su cui poter fare dei test-plan. Si presenta la sequenza di Sprint facendo un Github con i progetti separati per sprint. Ogni componente deve presentare l'attività svolta e bisogna parallelizzare i lavori, a parte nelle fasi preliminari che sono da fare insieme.

27.4 Conclusione

27.4.1 Il riscaldamento del prof

Tendenzialmente il prof inizia dall'analisi del problema, poi in base ad essa trova i requisiti.

27.4.2 La stanza

Bisogna farsi varie domande:

La stanza è vuota o ci sono degli oggetti? Essendo una sala d'attesa di una stazione sarà pavimentata, la mattina dovrebbe essere pulita e la sera sporca. In generale bisogna ragionare come il rasoio di Occam, si taglia l'inutile e si parte dal facile. Si ipotizza quindi che la stanza sia pulita.
Ci serve sapere come è fatta la stanza o no? Suppongo sia uno spazio euclideo piano e pavimentato, l'unica cosa da sapere è se rimane sempre uguale è meno. La situazione piu semplice è che se ci sono degli ostacoli essi siano fermi, in tal caso mi basta analizzare la stanza una sola volta e creare una mappa. Essendo che il robot non è puntiforme, bisogna definire la mappa in misura del robot, piu in particolare, in misura di una box che racchiude il robot. Gli oggetti quindi occuperanno delle celle robot. Più il robot è piccolo, più si evitano errori di scala. Ho un tool nella softwarehouse che mi permette di dire come è fatta la mappa e di usare il panificatore su di essa. Si rappresenta la mappa con gli 1 e gli 0. Ogniqualvolta dobbiamo muoverci, faccio un movimento alla volta poi ricalcolo con il planner. In questo caso la mappatura può essere automatizzata se i mobili sono fissi.
Se i mobili si muovono allora si manda il robot all'avventura.

Il robot si muoverà quindi in maniera sistematica, sapendo che mossa ha fatto, se ha incontrato un ostacolo e reagendo di conseguenza.
Un'altra problematica potrebbero essere se devo svuotare ma ancora non so dove è il bidone. Per questo ipotizzo che il bidone sia cablata al muro da qualche parte.

27.4.3 I Test plan

Finita l'analisi serve un test plan significativo. Problematica: devo mettere le conoscenze relative a tutto da qualche parte, le metto sul robot o magari qualcos'altro? Nasce quindi la necessità di una risorsa, individuando una architettura logica per cui serve un metamodello ad architettura esagonale.