ANNO ACCADEMICO 2025/2026

Tecnologie e Architetture Avanzate di Sviluppo Software

Teoria

Altair's Notes

DIPARTIMENTO DI INFORMATICA

Premessa

Licenza

Formato utilizzato

Box di "Concetto sbagliato":

Concetto sbagliato 0.1: Testo del concetto sbagliato

Testo contente il concetto giusto.

Box di "Corollario":

Corollario 0.0.1 Nome del corollario

Testo del corollario. Per corollario si intende una definizione minore, legata a un'altra definizione.

Box di "Definizione":

Definizione 0.0.1: Nome delle definizione

Testo della definizione.

Box di "Domanda":

Domanda 0.1

Testo della domanda. Le domande sono spesso utilizzate per far riflettere sulle definizioni o sui concetti.

Box di "Esempio":

Esempio 0.0.1 (Nome dell'esempio)

Testo dell'esempio. Gli esempi sono tratti dalle slides del corso.

Box di "Note":

Note:-

Testo della nota. Le note sono spesso utilizzate per chiarire concetti o per dare informazioni aggiuntive.

Box di "Osservazioni":

Osservazioni 0.0.1

Testo delle osservazioni. Le osservazioni sono spesso utilizzate per chiarire concetti o per dare informazioni aggiuntive. A differenza delle note le osservazioni sono più specifiche.

1 Introduzione

1.1 Intro al Corso

Parole chiave:

- Web Apps.
- Mission Critical.
- DevOps.
- Cloud Native.

Definizione 1.1.1: Mission Critical Applications

Un'applicazione o sistema le cui operazioni sono fondamentali per una compagnia o un'istituzione.

Osservazioni 1.1.1

- Enfasi sui requisiti non funzionali: i requisiti funzionali sono la baseline, ma ci si aspetta di più per rimanere competitivi.
- Da non confondere con life critical: non muore nessuno.

Definizione 1.1.2: Enterprise Application Integration (EAI)

Tutto l'insieme di pratiche architetturali, tecnologie, patterns, frameworks e strumenti che consentono la comunicazione e la condivisione tra diverse applicazioni nella stessa organizzazione.

Si ha enfasi sull'infrastruttura:

- *Data Integration:* combinare dati da più moduli diversi (coinvolge database).
- *Process Integration:* le interazioni tra più moduli.
- *Functional Integration:* si vuole fornire una nuova funzionalità sfruttando funzionalità già esistenti.

1.1.1 Esempio e Requisiti Non Funzionali

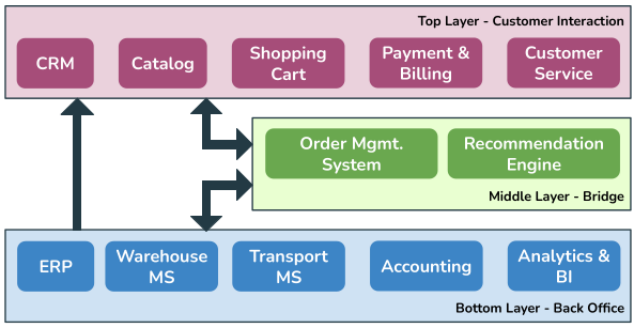

Figure 1.1: Esempio di e-commerce.

Commento dell'esempio:

- Ci sono tre livelli:
  - Top Layer: moduli che si rivolgono al cliente.
  - Middle Layer: gestione della comunicazione tra cliente e azienda.
  - Bottom Layer: moduli interni aziendali.

Requisiti non funzionali:

- High availability/zero downtime: l'applicativo deve essere sempre o quasi sempre disponibile.
- Affidabilità: in caso di interruzione di workflow si deve far sì che non ci siano stati danni (e.g. un'interruzione durante una transazione).
- Consistenza dei dati.
- Integrità dei dati.
- Low latency: per avere una buona performance, tutto deve essere fluido.
- Scalabilità.
- Sicurezza.
- Resilienza: capacità di reagire agli errori.
- Mantenibilità: quanto un pezzo di software sia mantenibile o riutilizzabile.
- Osservabilità: per comprendere eventuali problemi in un sistema distribuito.
- Auditability: le verifiche di qualità fatte su software[1].

1.2 Panoramica Storica

1.2.1 Dagli Anni '70 al 2000

Definizione 1.2.1: Waterfall

Le metodologie a cascata[a] sono metodologie in cui ci sono fasi ben distinte e separate tra loro.

[a] Viste a "Sviluppo delle Applicazioni Software".

[1] Meglio visto in "Etica, Società e Privacy".

Note:-

È un modello prevedibile, ma lento a gestire i cambiamenti.

Osservazioni 1.2.1

- Software on the shelf: una volta acquistato è proprio.
- Software custom: prodotto su richiesta, ha bisogno di tutto un servizio di manutenzione.

Definizione 1.2.2: Lean

Metodologie nate negli anni '50 alla Toyota, verranno applicate al software dagli anni '90. Si basa su tre principi:

- Muda[a] (waste): si deve stare sui requisiti, non mettere troppe funzioni non necessarie.
- Mura (unevenness): è necessaria consinstenza per aumentare la prevedibilità.
- Muri (overburden): non sovraccaricare le persone o le macchine. Non progettare software utilizzando strumenti greedy di risorse.

[a] JOJO'S Reference

Note:-

Lo strumento fondamentale è il *kanban*: la lavagna, per organizzare il lavoro.

Definizione 1.2.3: Siloed

Organizzazione aziendale a silos: si comunica poco e male. Ci sono 4 gruppi:

- BA Team: relazioni con gli stakeholders, requisiti, specifiche, documentazione.
- Dev Team: programma e fa un minimo di unit testing.
- Test Team: testa e decide se il sistema è pronto.
- Ops Team: si occupa del deployement.

Note:-

I vari team si parlano in maniera molto limitata.

Definizione 1.2.4: Transaction Processing Monitor

I TP monitor erano il primo esempio di soluzione middleware. Usata nei sistemi di mainframe erano: centralizzati, monolitici, mission critical, con accesso da vari terminali.

Corollario 1.2.1 Middleware

Software nel mezzo tra applicazioni e infrastrutture. Permette alle applicazioni di utilizzare le infrastrutture per farle comunicare tra di loro.

Obiettivi:

- Performance: si occupa di transazioni rispettando le proprietà ACID.
- Scalabilità: se un programma crasha ne avvia un'altra istanza.
- Affidabilità.
- Consistenza dei dati.

Limiti:

- Proprietario.
- Tight coupling.
- Costosi.
- Complessi.

Domanda 1.1

Cosa rimane dei TP monitors?

- *Gestione delle transazioni e coordinazione:*
  - Soluzioni basate su 2PC (2 Phase Commit).
  - Le proprietà ACID, attualmente supportate internamente da molti database.
  - Proprietà BASE:
    - Basically: risposte basiche.
    - Available: si accetta che si possa non avere il dato più aggiornato.
    - State: la consistenza potrebbe non essere rispettata.
    - Eventually: prima o poi si riceverà il dato corretto.
- *Pool di connessioni*.
- *Distribuzione del carico:*
  - Le richieste vengono distribuite su varie istanze.
  - In caso di fallimento l'applicazione riparte.

Definizione 1.2.5: Remote Procedure Call

Si chiama una funzione da una macchina remota come se fosse locale. È indipendente dal linguaggio e a una struttura silos. RIchiede aggiunte sia nello sviluppo che a runtime.

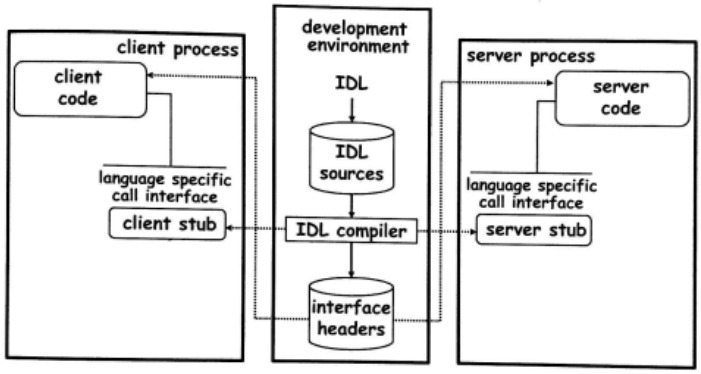

Figure 1.2: Remote Procedure Call - Development.

- Serializzazione: trasformare i dati in qualcosa che può essere comunicato.
- Marshalling: usa la serializzazione e inserisce meta-dati per permettere la ricostruzione della struttura dati.

Definizione 1.2.6: Common Object Request Broker Architecture (CORBA)

Evoluzione di rpc pensata per gli oggetti. Si possono creare oggetti in un server che possono rispondere a chiamate remote.

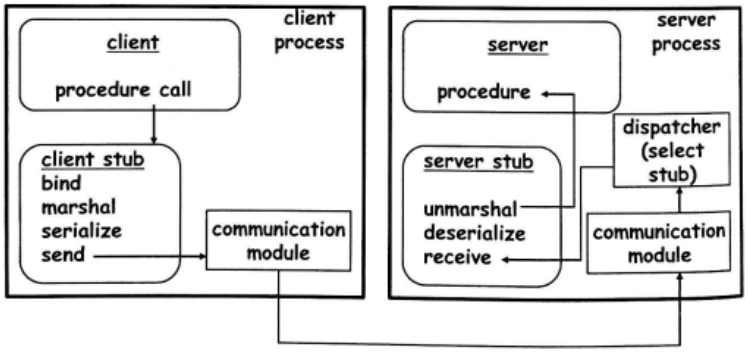

Figure 1.3: Remote Procedure Call - Runtime.

Note:-

Più successo lo ha avuto RMI (Remote Method Invocation) che è CORBA, ma solo con Java.

Limiti:

- Nascondere le cose al programmatore: si ha un falso senso di disaccoppiamento e i programmatori tendono a non vedere la rete.
- La programmazione sembra semplice perché i problemi vengono sottovalutati.

Definizione 1.2.7: Message Oriented Middleware

Invece di chiamarsi a vicenda le applicazioni si inviano messaggi a vicenda:

- Sincronizzazione tra operazioni in applicazioni diverse.
- Notifiche di eventi.
- Non c'è necessità di conoscere il ricevente.

Due modelli di comunicazione:

- Point-to-Point: il mittente manda un messaggio nella coda del middleware, il ricevente lo consuma.
- Publish and Subscribe: c'è una bacheca su cui chiunque può pubblicare un evento.

Definizione 1.2.8: Enterprise Service Bus (ESB)

Un middleware coscente della logica di business. Si occupa di tradurre protocolli e dati.

Note:-

Caduto totalmente in disuso.

1.2.2 Dal 2000 ai Giorni Nostri

Definizione 1.2.9: AGILE

Metodologie fondate su itertività e incrementalità.

Corollario 1.2.2 XP - Xtreme Programming

Si concentra sul codice, lo sviluppo di software si fa in team. Si dà importanza ai feedback sia dai clienti che dagli sviluppatori (small release, test-driven development, on-site customer).

Principi di XP:

- Comunicazione.
- Semplicità.
- Feedback.
- Coraggio.
- Rispetto.

Definizione 1.2.10: Scrum

Prassi di organizzazione dell'attività lavorativa degli sviluppatori, si concentra sulla comunicazione:

- Organizzazione: esiste una lista del lavoro che deve essere svolto (Product Backlog e PDI), un'iterazione di lavoro di massimo 4 settimane (sprint) e deve esserci un incremento (valore percepibile dal cliente).
- Ruoli: ci si organizza in piccoli teams per ogni modulo.

Ruoli in Scrum:

- Product owner: persona che gestisce il Backlog, in contatto con i clienti (non è il capo).
- Scrum master: organizza le riunioni, fa da mediatore.
- Development team.

Eventi per ogni sprint:

- Sprint panning: riunione in cui si decide cosa fare.
- Daily scrum: meeting in piedi, deve durare poco.
- Sprint review: alla fine dello sprint, si mostra l'incremento agli stakeholders.
- Sprint retrocspective: dopo la review, è una riunione interna al team.

Definizione 1.2.11: Kanban

Si vuole mantenere il flusso di lavoro. Non si mette più lavoro di quello che si riesce a fare.

Principi di Kanban:

- Visualizzazione: si vede il proprio lavoro attraverso delle lavagne su cui vengono appiccicati post-it.
- WIP limit: si fanno un certo numero di cose contemporaneamente (non più di 3-4).
- Pull system[2]: le cose vengono spostate dal to do al doing quando si libera un posto.
- Continuous delivery: si integra la feature implementata e la si consegna.

Board:

- Backlog.
- To do: roba da fare.
- Doing (WIP limit): roba che si sta facendo.
- Done: roba fatta.

Definizione 1.2.12: Scrumban

Scrum: ha i ruoli, il product Backlog e PBI, daily meeting, sprints.
Kanban: il flusso è pull-based e usa i WIP limit, le lavagne e i Continuous delivery.

[2] Gacha moment.

Siloed evoluta:

- Biz team: relazioni con gli stakeholders, marketing e vendite.
- Dev team: requisiti, sviluppo, testing e comunicazione con il biz team.
- Ops team: deploy, setting, validazioni.

Note:-

La divisione c'è ancora, ma c'è più comunicazione tra i vari team.

Definizione 1.2.13: Service-Oriented Architecture

Si inizia a ragionare sul fatto che l'integrazione debba avvenire mediante moduli che forniscono servizi l'uno all'altro.

Domanda 1.2

Cos'è un servizio?

Corollario 1.2.3 Servizio

Un servizio è una capacità di business autocontenuta che viene esposta secondo un contratto standard (un'interfaccia).

I servizi:

- *Coarse-grained:* ogni servizo implementa tutto (più pesanti dei microservizi).
- Condivide dati e funzioni attraverso interfacce (o API).
- *Scopribili:* i servizi si scoprono attraverso nomi e non IP.

SOA:

- Comunicazione attraverso applicazioni apposta o protocolli basati su HTTP.
- Le infrastrutture hanno un ruolo importante nel comporre i servizi in funzioni.
- Deployment centralizzato.

Definizione 1.2.14: Web Services

Istanza di Service-Oriented Architecture che stabilisce:

- Protocollo di comunicazione (SOAP):
  - XML su HTTP.
  - Consente sia comunicazione sincrona che asincrona.
- Service registry: UDDI
  - Elenco di servizi registrati secondo le loro features generali.
  - Consente ai servizi di essere scopribili.
  - Comunicazione mediante SOAP.
- Contratto (WSDL, Web Service Description Language):
  - Fornisce informazioni per contattare effettivamente un servizio.
  - La struttura dei messaggi.
  - Le strutture dati.
  - Protocollo e indirizzo.

Osservazioni 1.2.2 Sui Web Services

Idealmente:

- Il client cerca il servizio su UDDI.
- Ottiene il link dal WSDL del servizio.
- Utilizzando WSDL collega dinamicamente il servizio alle operazioni.

In Pratica:

- La scoperta di servizi "in tempo reale" era impraticabile.
- I WSDL erano in maggioranza statici.
- Le informazioni venivano salvate in file di configurazione.

1.2.3 JavaEE e Cloud

Definizione 1.2.15: Enterprise Java Beans (EJB)

Gli EJB sono oggetti resi disponibili dinamicamente. Offrivano:

- Gestione del lifecycle.
- RMI.
- Sicurezza basata sui ruoli.
- Persistenza tramite Object-relational mapping (ORM).
- Gestione delle transazioni ACID.

I Java Beans erano pesanti:

- Oggetti collegati alla JVM (al container).
- Molto accoppiati all'ambiente di esecuzione.
- Necessitavano un java application server.
- Molto codice boiler-plate.
- Annotazioni XML.
- Non portabili.

Note:-

Tutto questo fino al 2006 in cui la terza edizione di EJB li fa diventare più leggeri:

- Annotazioni Java al posto di XML.
- POJOs (Plain Old Java Objects).
- JPA (Java Persistence).
- Si integrano con web service.
- Introduzione della *dependency injection*: design pattern per collegare due o più moduli tramite l'ambiente di sviluppo stesso.

Definizione 1.2.16: Cloud

Insieme di risorse sia comptazionali, sia di storage, sia di networking. Queste risorse sono rese disponibili come servizi mediante API.

Osservazioni 1.2.3

- Il cloud è un'astrazione che nasconde la struttura fisica delle macchine.
- C'è un livello simile a un OS.
- I servizi sono offerti su base dichiarativa: diventa possibile avere un servizio che si conformi alle proprie necessità.

Modelli:

- *Infrastructure as a Service (IaaS):* l'azienda mette a disposizione macchine virtuali, di storage o sottoreti visibili a chi compra il servizio.
- *Platform as a Service (PaaS):* si acquista una piattaforma che nasconde cose e ottimizza.
- *Function as a Service (FaaS):* si carica su una piattaforma una serie di funzioni e si sviluppa solo il front end.

Applicazioni native sul cloud:

- Moduli molto leggeri e loosely-coupled.
- Deployment contenerizzato (impacchettato e Platform independent), orchestrazione (ignorante rispettto all'architettura ma che può operare su essa) e elasting scaling (cambiare il livello dei servizi).
- Dev Cycle features: integrazione continua, continuous delivery, deploy, infrastrutture dichiarative, consistenza tra dev/test/prod, anticipare i test sulla sicurezza, l'applicazione deve essere osservabile.
- NFRs: scalabilità, portabilità, sicurezza, evoluzione, mantenibilità, affidabilità.

Definizione 1.2.17: DevOps

Gestione del sistema mediante l'utilizzo di tools e pratiche basato sui principi Lean.

1.3 Analisi dei Requisiti

1.3.1 Architetture Monolitiche

Domanda 1.3

Ma ci serve un'architettura a microservizi?

Definizione 1.3.1: Architettura Monolitica

Un'architettura monolitica è semplice da sviluppare, non è distribuita, non ha integrazioni complesse, è facile da deployare e scalare, non ha coupling ed è facile da scalare.

Note:-

Può essere utile se piccola (non è vero, ma facciamo finta che lo sia). Il threshold è da 5 a 10 persone.

Corollario 1.3.1 Monolithic Hell

Il monolithic hell è un termine utilizzato per descrivere i rischi di un'applicazione monolitica.

Monolithic Hell:

- Teams che crescono e la coordinazione diventa ingestibile.
- Frammentazione della conoscenza: nessuno comprende tutto il sistema.
- I cambiamenti diventano rischiosi e costosi.
- All-or-Nothing update: o si cambia tutto o non si cambia nulla.
- I deployement possono essere lenti e causare downtime.
- Un fail su una funzione può buttare giu tutto il sistema.

	SOA	Microservices
Scope	business high level function (coarse)	subdomain / single purpose (fine grained)
Communication	Smart pipes (e.g. Enterprise Service Bus), heavyweight protocols	Dumb pipes (e.g. message brokers or service-to-service with lightweight protocols - REST, gRPC)
Storage	Shared databases	Database-per-service
Deploy	Tightly coupled services sharing same stack	Loosely coupled "polyglot" services
Scalability	Difficult, due to shared dependencies	Highly scalable, each service scales independently

Figure 1.4: SOA vs. Microservizi.

Microservizi e Miniservizi:

- I microservizi propriamente detto dovrebbe implementare una sola funzione.
- I miniservizi svolgono un'unità funzionale coesa e coerente, ma non necessariamente una sola funzione.

Note:-

Per chi fa questa distinzione quelli che vedremo nel corso sono considerati miniservizi.

1.3.2 Microservizi e DevOps

Nella figura 1.5:

- L'architettura a microservizi permette un'organizzazione autonoma e teams polifunzionali.
- L'organizzazione in questi teams permette a sua volta il continuous delivery e il continuos deployement.
- I teams devono essere piccoli in modo che possano essere eventualmente riorganizzati.
- In sostanza: dividere un'app in microservizi (invece che monolitica) consente un deployement costante. Per gestire i microservizi si fa affidamento su piccoli teams autonomi, che vanno a rinforzare il DevOps.

Definizione 1.3.2: API

Le API sono un insieme di regole e protocolli che permettono a software diversi di "parlare" tra di loro.

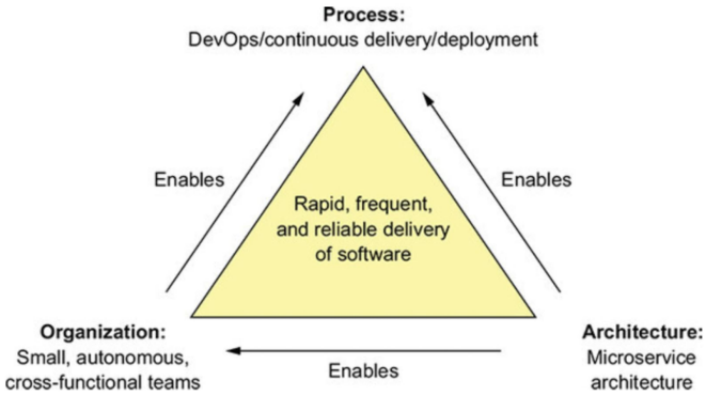

Figure 1.5: Microservizi e DevOps.

Architettura a microservizi:

- Normalmente viene eseguita su un server che espone al cliente delle cose (per esempio user interface).
- C'è un modulo dedicato che funge da gateway. Il front-end si collega a un indirizzo web tramite esso.
- Ogni servizio espone un API REST.

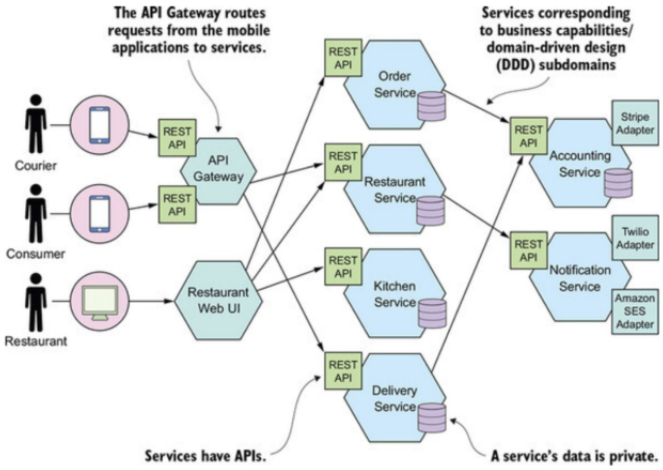

Figure 1.6: API

Definizione 1.3.3: Conway's Law

Le organizzazioni che producono softwares sono obbligate a produrre softwares che sono copie delle strutture comunicative dell'azienda.

Corollario 1.3.2 Inverse Conway Maneuver

L'idea è quella di strutturare la propria organizzazione in modo tale che la struttura rispecchi la propria architettura a microservizi. Così facendo i dev teams sono debolmente collegati ai servizi.

Note:-

"Spesso i softwares delle organizzazioni pubbliche come l'università non è granché", *citazione necessaria*.

Sfide dei microservizi:

- Complessità: un'architettura a microservizi è un sistema distribuito.
- Richiede ristrutturazione dell'organizzazione aziendale.
- Deve tenere conto della performance della rete:
  - Evitare *chatty service*: servizi che si scambiano tanti piccoli messaggi.
  - Evitare messaggi enormi.
  - Minimizzare la latenza.
- Misure di sicurezza per ogni servizio.

1.3.3 Domain-Driven Design

Definizione 1.3.4: Domain-Driven Design (DDD)

Il Domain-Driven Design è un approccio al design software che nasce intorno alla nozione di "modello di dominio":

- Il modello di dominio cattura in una maniera formale, ma concettuale, i concetti rilevanti, le entità, le relazioni e le regole di uno specifico business.
- Il modello di dominio è l'output dell'analisi dei requisiti ed è l'input della fase di design.
- Si utilizza un approccio AGILE.

Note:-

L'idea di questo approccio: tutto il DevOps ha un'idea chiara del dominio e delle sue regole.

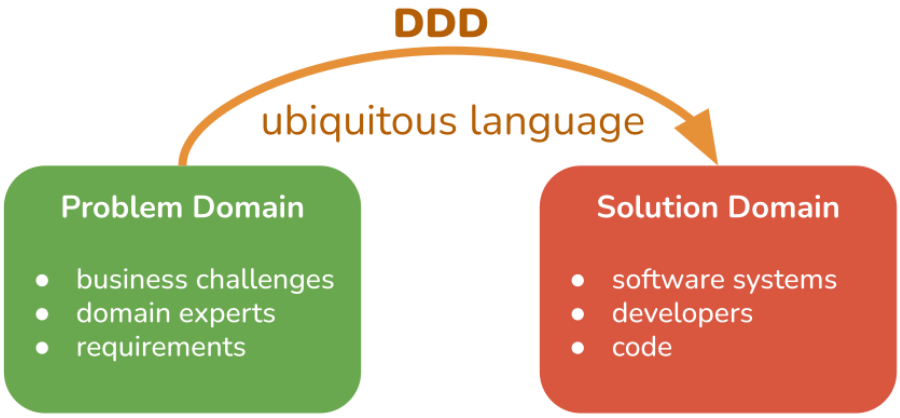

Figure 1.7: Dal problema alla soluzione.

Definizione 1.3.5: Ubiquitous Language

Un linguaggio comune costruito insieme e condiviso dagli esperti del dominio e dal development team. Deve essere usato:

- Nel glossario.
- In tutta la documentazione.
- Nel codice.

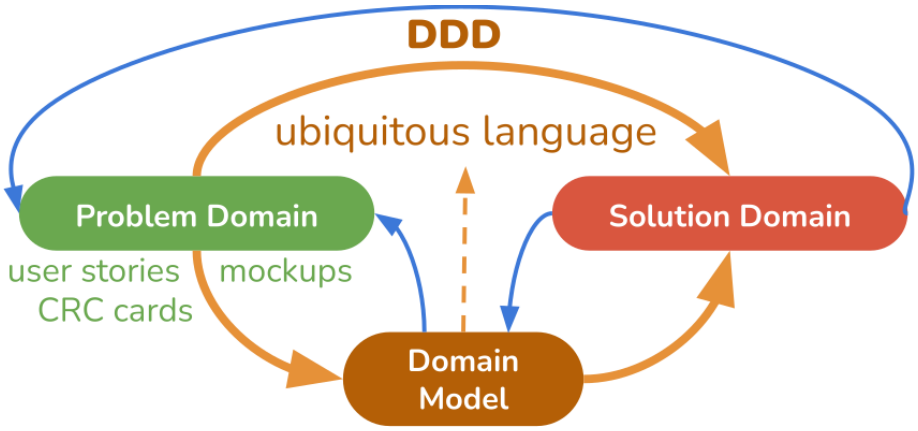

Figure 1.8: Connessione tra problema e soluzione.

Osservazioni 1.3.1

- Non ci si può aspettare che un modello di dominio rifletta completamente il mondo reale.
- È una rappresentazione selettiva della prospettiva del problema che si cerca di risolvere.
- Astrazioni, confini e conoscenza condivisa.
- "Tutti i modelli sono sbagliati, ma alcuni sono utili".

Figure 1.9: Quando si utilizza DDD.

1.3.4 Modeling Practice

Definizione 1.3.6: Event Storming

L'event storming è una tecnica di modellazione collaborativa in cui si esplora un dominio, si scrivono gli eventi rilevanti su dei post-it e li si attacca alla parete. Dopo di che si creano delle sequenze cronologiche per far capire quali sono i flussi.

Note:-

Ciò permette di chiarire eventuali ambiguità o fraintendimenti.

Definizione 1.3.7: Value Streams

Un value stream è una sequenza end-to-end di attività che un'organizzazione effettua per portare un valore a un cliente o a uno stakeholder.

Note:-

Deve mostrare chiaramente il valore, essere comprensibile alle altre entità coinvolte e deve essere in terza persona.

Corollario 1.3.3 Support Streams

Come i value streams ma interni all'azienda, non riguardano direttamente il cliente finale.

Definizione 1.3.8: User Stories

Una user story è una breve narrativa che descrive un processo o un goal dal punto di vista di un solo attore. Cattura le motivazioni, le azioni e il risultato desiderabile.

Note:-

Una variante sono le AGILE user stories che sono composte da una sola frase (As a [actor], I want [goal], so that [reason]).

1.4 Ripasso su Spring Boot e React

Note:-

DISCLAIMER: è il mio primo approccio alla programmazione web (dato che sono specializzata in robe teoriche e/o a basso livello) per cui potrei fare qualche imprecisione, sorry.

1.4.1 Maven

Dato che gli IDE moderni consumano un sacco di batteria e risorse includo anche una mini guida per setuppare un progetto java con Maven (in questo modo potete usare vim, gedit o nano se vi va). Se usate Intell*J, Vs C*de o altro potete saltare[3].

Definizione 1.4.1: Maven

Maven è un tool per creare automaticamente delle build di progetti java. Permette di compilare codice, fare testing, packaging, etc.

Note:-

Maven utilizza il *Project Object Model (POM)* per descrivere la configurazione di un progetto e gestire le dipendenze.

Domanda 1.4

Come si crea un progetto con Maven?

Listing 1.1: Creazione di un progetto Maven

```
mvn archetype:generate \
    -DgroupId=com.example \
    -DartifactId=myapp \
    -DarchetypeArtifactId=maven-archetype-quickstart \
    -DinteractiveMode=false
```

[3] Questi IDE possono utilizzare anche Maven, ma lo gestiscono loro.

Nello specifico:

- `DgroupID` indica il nome di una compagnia o di un'organizzazione.
- `DartifactId` indica il nome del progetto.
- `DarchetypeArtifactId` indica il template (in questo caso un semplice HelloWorld java).

Listing 1.2: Esempio di pom.xml per Spring Boot

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
         http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>3.3.4</version>
        <relativePath/>
    </parent>

    <groupId>com.example</groupId>
    <artifactId>myapp</artifactId>
    <version>1.0.0-SNAPSHOT</version>
    <packaging>jar</packaging>
    <name>myapp</name>

    <properties>
        <java.version>24</java.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
</project>
```

Spiegazione:

- Il `parent` imposta la versione di Spring Boot e le configurazioni di default.
- Le `dependencies` includono il modulo web e quello per i test.
- Il plugin `spring-boot-maven-plugin` permette di eseguire l'app con `mvn spring-boot:run`.

1.4.2 Gradle

Per alcune persone può essere più facile utilizzare Gradle (inclusa me), quindi aggiungo qualcosa anche per questo.

Definizione 1.4.2: Gradle

Come Maven, Gradle è un tool per creare automaticamente progetti java, C/C++, kotlin, etc. A livello di base ha le stesse funzionalità di Maven, le differenze principali sono il linguaggio utilizzato (Maven è basato su xml, Gradle su Groovy), velocità (Gradle è più veloce per build incrementali), etc.

Domanda 1.5

Come si crea un progetto Spring Boot con Gradle?

Listing 1.3: Creazione di un progetto Spring Boot con Gradle

```
curl https://start.spring.io/starter.tgz \
        -d type=gradle-project \
        -d dependencies=web \
        -d groupId=com.example \
        -d artifactId=test \
        -d name=test \
        -d packageName=com.example.test \
        -o test-gradle.tgz
  tar -xvf test-gradle.tgz
  cd test
```

Alcune osservazioni importanti:

- Di default il progetto creato usa java 17, per cambiarlo basta andare nel file `build.gradle`.
- Inizialmente darà errore perché non si sono definiti endpoint.

Listing 1.4: Avvio del progetto Spring Boot con Gradle

```
# Su Linux/macOS
./gradlew bootRun

# Su Windows
gradlew.bat bootRun
```

Note:-

Al primo avvio Gradle scaricherà tutte le dipendenze necessarie. Una volta completato, l'app sarà disponibile su `http://localhost:8080/`. Se non hai ancora definito controller o endpoint, vedrai la *Whitelabel Error Page*.

1.4.3 SpringBoot

Note:-

Non descriverò come fare un progetto su IntelliJ, se non ci riuscite è skill issue.

Definizione 1.4.3: SpringBoot

SpringBoot è un frameworks open-source per la programmazione di webapp.

Spring usa il pattern MVC:

- *Controller:* punto di ingresso delle richieste esterne.
- *Model:* consultato dal controller quando arriva una richiesta.
- *View:* prodotta dal controller.

Domanda 1.6

Come si fa a fare un controller in spring?

Definizione 1.4.4: Annotazioni java

Modo per aggiungere metadati nel codice java. Forniscono informazioni extra al compilatore.

Annotazioni in spring:

- `@RestController`: fa capire a spring che è un controller e quindi deve stare in attesa di richieste HTTP.
- `@RequestMapping(...)`: specifica dove devono arrivare le URL. Per esempio in una classe `TavoliController` può esserci l'annotazione `@RequestMapping("/tavoli")`.
- `@GetMapping(...)`: handler per le varie richieste. Il suo contenuto viene aggiunto dopo la root string specificata da `@RequestMapping(...)`. Per esempio `@GetMapping({id})` indica che si vuole un tavolo con un determinato menu.
- `@PathVariable`: si mette nella signature dei metodi per passare i parametri presi dal `@GetMapping(...)`. Per esempio `ResponseEntity<Tavolo> getTavoli(@PathVariable int id)` va a utilizzare l'id specificato in precedenza. Questo porta a due casi:
  - Se l'id c'è si restituisce una `ResponseEntity.ok(...)`.
  - Se non c'è viene restituito `ResponseEntity.notFound().build()`[4]. Il `notFound()` non dà una risposta definitiva, ma permette di aggiungere altre caratteristiche (è il `build()` che formula la risposta HTTP).
- `@PostMapping(...)`: il controller si aspetta che la richiesta abbia un body. Per esempio `@PostMapping("crea")` si occupa di creare un ordine.
- `@RequestBody`: si aspetta un body (quindi si usa nelle `@PostMapping(...)`), per cui un oggetto JSON che può essere deserializzato. Questo può portare a:
  - `ResponseEntity.badRequest().build()`: significa che l'utente ha sbagliato qualcosa nella richiesta (e.g. chiedere di entità che non esistono).
  - `ResponseEntity.InternalServerError().build()`: errore nell'esecuzione per cui il server non riesce a soddisfare la richiesta.
- `@Service`: l'istanza dell'oggetto viene creata automaticamente da spring. Viene creata a partire da un costruttore.

[4] Il famigerato 404.

Figure 1.10: Not found.

- `@PostConstruct`: metodi che vanno invocati subito dopo la costruzione del `@Service`.
- `@RestControllerAdvice`: per il `GlobalErrorHandler`, la gestione di errori di alto livello.
- `@ExceptionHandler(...)`: quando arriva un'eccezione la gestiscono.

Jakarta (persistence), per gestire i database:

- `@Entity`: la classe corrisponde a una tabella del database.
- `@Table(...)`: per specificare il nome della tabella (di default è il nome della classe).
- `@Id/@EmbeddedId`: campo chiave.
- `@GeneratedValue(...)`: per scegliere la strategia di generazione e altri parametri.
- `@Column(...)`: una colonna della tabella, si possono specificare varie opzioni come lunghezza o nullable.
- `@JoinColumn(...)`: per effettuare join, va inoltre specificto il tipo di relazione con un'altra annotazione (`@ManyToOne(...)`, `@OneToOne`, etc.).
- `@MapsId(...)`: quando ci sono più id che indicano la stessa cosa.
- `@Enumerate(...)`: di default gli enum vengono tradotti come numeri, con questa annotazione si possono specificare altri tipi come String.

Note:-

È importante che sia presente un costruttore vuoto per l'entità.

Per avere corrispondenza tra gli oggetti e le entità del database si devono definire dei repositories:

- Sono interfacce che estendono `JpaRepository<>`.
- Si specificano il tipo di oggetto e il tipo di id.
- Si può utilizzare l'annotazione `@Repository`, ma non ha sostanzialmente effetto.
- Questi repositories sono connessi ai services.
- `@Transactional`: importante per tenere traccia delle modifiche. Si usa nei metodi dei services.

Definizione 1.4.5: Applicazione Headless

Un'applicazione che non restituisce delle pagine, ma dei JSON. I files JSON possono essere intesi come view nel pattern MVC.

Spring Security:

- Appena inserito lo starter viene creato un utente fittizio.
- Da questo momento tutte le operazioni necessitano un'autenticazione, se non la si ha viene restituito 401 - Unauthorized.
- I metodi POST sono soggetti a Cross-Site Request Forgery (CSRF), per prevenire richieste false da parte di malintenzionati. Per risolvere bisogna configurare la Security.
- Si aggiunge al progetto la classe SecurityConfig con l'annotazione `@Configuration`.
- Il metodo `filterChain` è annotato come `Bean`, ossia un *singleton*.
- `@EnableMethodSecurity:` i controlli verranno fatti nel controller.
- `@PreAuthorize(...):` va a specificare condizioni tipo se l'user ha un ruolo specifico. Se non è presente i metodi sono accessibili a chiunque sia autenticato.

1.4.4 React