Lezione 7 29/10/2024

Understanding quality attributes

Vediamo il concetto di qualità e come questo può essere descritto dal punto di vista architetturale.

Lo **scenario** è il modo in cui si descrive una cerca situazione nel sistema e il come riesco a garantire che quella situazione avvenga con una certa qualità garantita.

- Questo deve essere **testabile**, bisogna poter valutare che la situazione possa essere gestita con quel livello di qualità.
- è difficile capire a quale qualità si fa riferimento, se per esempio ho una system failure per colpa di un ddos, bisogna capire se questo è un aspetto di disponibilità, performance, sicurezza, usabilità? Bisogna capire in che contesto ci troviamo e quale attributo di qualità stiamo considerando.
- Gli attributi possono avere interpretazioni diverse, bisogna non avere ambiguità, usare definizioni condivise da tutti.

Gli **scenari** permettono di andare a rappresentare un insieme di situazioni in cui il sistema si può trovare, e andare a specificare quali sono le tattiche che posso mettere in gioco per fare in modo che la qualità attesa è esattamente quella che otterrò a runtime.

Esistono due categorie di **qualità**:

- Quelle che percepisco solo a **runtime**, quindi solo quando il sistema è in esecuzione;
- Quelli legati alla **staticità** del sistema, quindi legati allo sviluppo del software: modificabilità, testabilità...

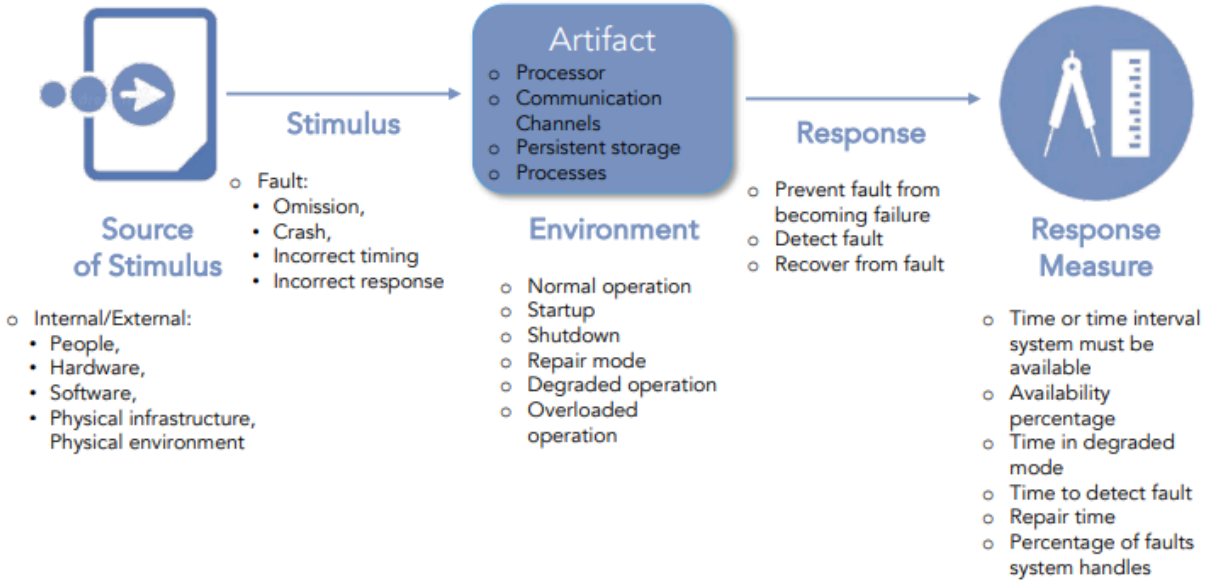
Noi adatteremo un metodo generale per descrivere queste situazioni.

Quindi modello un contesto e poi specifico, attraverso l'applicazione di tattiche, quelle decisioni architetturali che mi permetteranno di ottenere un particolare requisito di qualità (tipo availability) con un certo livello atteso.

Lo scenario permette di descrivere:

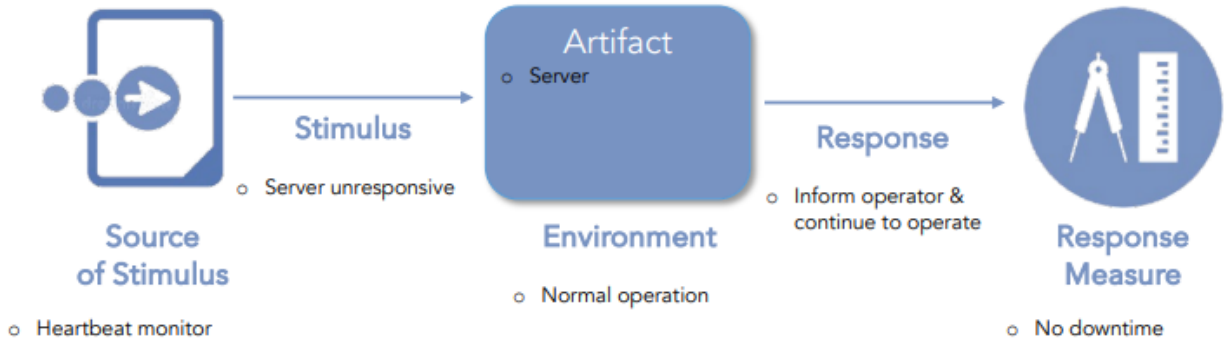
Characteristic	Description
Stimulus	• The stimulus is a condition that requires a **response** when it arrives • For runtime qualities a stimulus can be: - an event (for **performance**), a user operation (for **usability**), an attack (for **security**) • For development qualities a stimulus can be: - a request for a modification (for **modifiability**), the completion of a development phase (for **testability**)
Stimulus source	• This is some **entity** (a human, a computer system, etc.) that generated the stimulus
Response	• The response is the **activity undertaken** as the result of the arrival of the stimulus - By the system for runtime qualities - By the developer for development qualities
Response measure	• The response should be **measurable** in some fashion so that the quality requirement can be tested - latency or throughput (for **performance**) - the time required to make, test, and deploy the modification (for **modifiability**)
Environment	• The set of **circumstances** (the context) in which the scenario takes place - A request for a modification that arrives after the code has been frozen for a release may be treated differently than one that arrives before the freeze
Artifact	• The target of the stimulus - This may be the whole system (or project), or some pieces of it

Questo è un insieme che i vari elementi dello scenario possono avere, per l'availability:



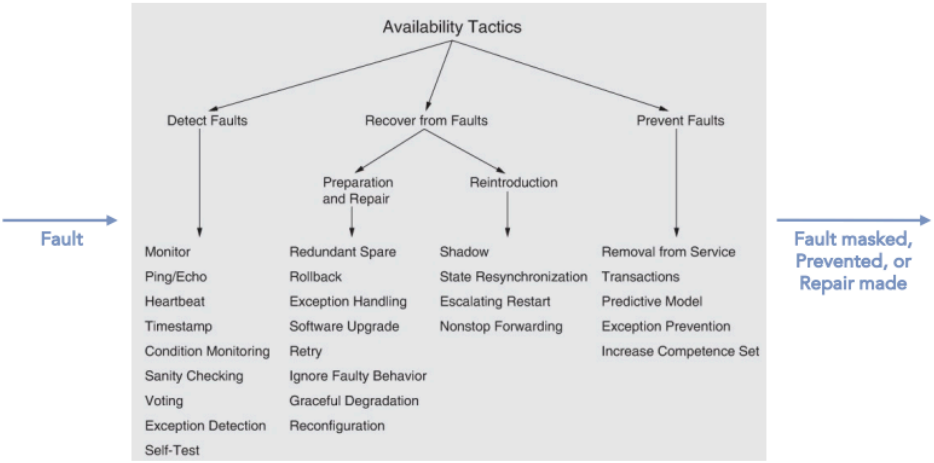
L'availability è uno degli aspetti più rilevanti sui sistemi di software attuali.

Ecco un esempio:



La sorgente è un monitor che manda dei segnali per capire se un servizio è attivo, lo stimolo è il fatto che non arriva la risposta, l'artefatto è il server in una condizione di normal operation, la risposta che ci si aspetta che il sistema faccia è di fermare l'operatore e fare in modo che si continui ad operare (come switchare ad un server di backup), di modo che non ci sia downtime.

Questo è un esempio, se voglio fare un miglioramento del tempo di risposta (latenza), se ho un elemento che deve processare delle code di messaggi allora posso usare un load balancer che divide il flusso di dati in due elementi. Questa è una tattica. Dipende dalle frequenze che mi aspetto.



Analizzare quanto bene sono stati raggiunti gli attributi di qualità è una parte fondamentale del compito di progettare un'architettura. Non bisogna aspettare che il tuo progetto sia "completo" prima di iniziare a farlo.

I questionari basati su tattiche possono essere utili per ottenere informazioni sulla capacità dell'architettura di supportare gli attributi di qualità necessari.

Per ciascuna domanda del questionario, l'architetto registra le seguenti informazioni:

1. Se ciascuna tattica è supportata dall'architettura del sistema.
2. Se vi sono rischi evidenti nell'uso (o nel mancato uso) di questa tattica.
3. Se la tattica è stata utilizzata, registrare come viene realizzata nel sistema, o come si intende realizzarla (ad esempio, tramite codice personalizzato, framework generici o componenti prodotti esternamente).
4. Le specifiche decisioni di progettazione prese per realizzare la tattica e la posizione della sua implementazione.
5. Qualsiasi motivazione o ipotesi fatte nella realizzazione di questa tattica.

Sommario

I requisiti per un sistema rientrano in tre categorie:

- **Funzionali**: Questi requisiti sono soddisfatti includendo un insieme adeguato di responsabilità all'interno del design.
- **Attributi di qualità**: Questi requisiti sono soddisfatti tramite le strutture e i comportamenti dell'architettura.
- **Vincoli**: Un vincolo è una decisione di progettazione che è già stata presa.

Per esprimere un requisito di attributo di qualità, utilizziamo uno **scenario di attributo di qualità**, che include:

- **Fonte dello stimolo**
- **Stimolo**
- **Ambiente**
- **Artefatto**
- **Risposta**
- **Misura della risposta**

Una **tattica architetturale** è una decisione di progettazione che influisce sulla risposta di un attributo di qualità. L'obiettivo di una tattica è concentrarsi su una singola risposta di un attributo di qualità.

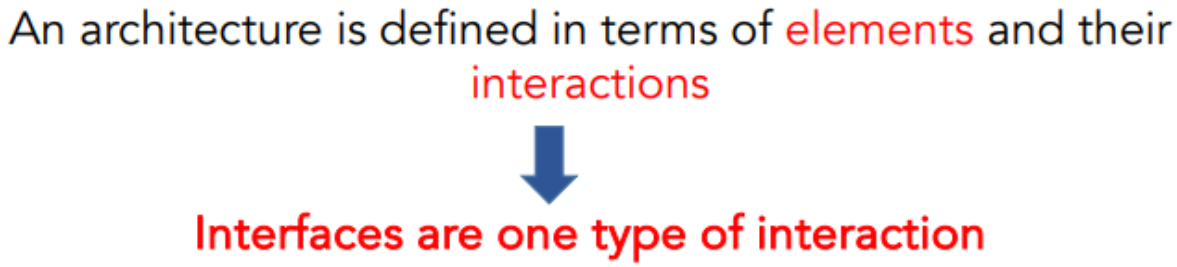
I **pattern architetturali** possono essere considerati come "pacchetti" di tattiche.

Un analista può comprendere le decisioni prese in un'architettura attraverso l'uso di una **checklist basata su tattiche**.

Questa tecnica leggera di analisi dell'architettura può fornire informazioni sui punti di forza e di debolezza dell'architettura in un tempo molto breve.

Interfacce software

Oggi i sistemi sono spesso costruiti usando **risorse virtualizzate** che risiedono in un **cloud** e devono fornire e dipendere da **interfacce esplicite**. Questi sistemi sono spesso mobili.



Le **interfacce** sono un meccanismo di astrazione fondamentale necessario per connettere insieme gli elementi. Esse influenzano la **modificabilità**, **usabilità**, **testabilità**, **prestazioni**, **integrabilità** di un sistema, e altro ancora.

Un'interfaccia permette di astrarre da come il componente realizza le funzionalità che offre, e da come utilizza i servizi che gli altri offrono.

Un'**interfaccia** è un confine attraverso il quale gli elementi interagiscono, comunicano e si coordinano.

- **Attori**: sono gli altri elementi, utenti o sistemi con cui un elemento interagisce.
- **Ambiente**: è l'insieme di attori con cui un elemento interagisce.
- Le **interazioni** possono assumere diverse forme, sebbene la maggior parte comporti il trasferimento di controllo e/o dati.

Posso far si che un elemento, attraverso un'interfaccia, controlli un altro elemento ma senza mandare un dato, manda uno stimolo che attiva l'altro elemento. Oppure un componente può inviare o rendere disponibili dati ad altri attori.

Ci sono 2 macro tipi di interazioni (noi ci concentriamo su quelle dirette):

Interazioni dirette

- Abilitate da risorse che forniscono punti di interazione diretta con un elemento.
- Alcune interazioni sono supportate da costrutti standard dei linguaggi di programmazione, come ad esempio:
  - Chiamate di procedura locali o remote (RPC),
  - Flussi di dati,
  - Memoria condivisa,
  - Passaggio di messaggi.

Interazioni indirette

- Ad esempio, il fatto che l'uso della risorsa X sull'elemento A lasci l'elemento B in uno stato particolare è qualcosa che altri elementi che utilizzano la risorsa potrebbero dover sapere, se questo influisce sul loro processamento, anche se non interagiscono mai direttamente con l'elemento A.

(skippa quelle indirette)

Esempio: se ho l'interfaccia animale, questa mangia, dorme e fa versi. Nel momento in cui definisco delle classi che implementano quest'interfaccia, mi aspetto che siano animali, e quindi che facciano quelle cose. Però loro danno un'implementazione specifica in base all'animale.

Quindi data un'interfaccia io poi posso avere diverse implementazioni.

Abbiamo 3 **concetti principali** legati all'interfaccia:

- Tutti gli **elementi** hanno interfacce.
- Le **interfacce** sono bidirezionali.
- Un elemento può interagire con più **attori** tramite la stessa interfaccia.

Interfacce multiple

È possibile suddividere un'unica interfaccia in **più interfacce**. Ognuna di queste ha uno scopo logico correlato e serve una diversa classe di attori.

Per esempio abbiamo un componente unico che permette di visualizzare prodotti, fare ricerche, aggiungere elementi al carrello e fare il checkout. Potrei mettere tutto questo in un'unica interface, ma generalmente si splitta in più interfacce, dove ciascuna contiene le risorse che servono.

Multiple interfaces offrono una sorta di **separazione dei concerns**.

- Un attore potrebbe richiedere solo un sottoinsieme della funzionalità disponibile.
- Al contrario, il fornitore di un elemento potrebbe voler concedere agli attori diversi diritti di accesso o implementare una **politica di sicurezza**.

Risorse

Le risorse sono il punto di interazione diretta.

La sintassi è la “**firma**” del metodo, che include il nome della risorsa, i nomi e i tipi di dati degli argomenti se ce ne sono.

Semantica delle risorse: qual è il risultato dell'invocazione di questa risorsa?

- Assegnazione di valori ai dati che l'attore che invoca la risorsa può accedere.
- Assunzioni sui valori che attraversano l'interfaccia.
- Modifiche nello stato dell'elemento causate dall'uso della risorsa.
- Eventi che saranno segnalati o messaggi che saranno inviati a seguito dell'uso della risorsa.
- Come si comporteranno diversamente le altre risorse in futuro a causa dell'uso di questa risorsa.
- Risultati osservabili a livello umano.
- ...

Operations, Events, and Properties

Abbiamo detto che le risorse permettono di controllare e trasferire dati, sono il punto di contatto, controllano l’attivazione di un elemento e nel caso trasferiscono i dati.

Di base (raffinando la definizione), le **risorse** consistono in:

- **Operazioni:** invocate per trasferire controllo e dati all'elemento per l'elaborazione.
- **Eventi**
  - **Eventi in uscita**: elementi attivi producono eventi in uscita utilizzati per notificare i listener.
  - **Eventi in ingresso**: possono essere la ricezione di un messaggio o l'arrivo di un elemento di flusso da consumare.
- **Metadati/Proprietà** :come diritti di accesso, unità di misura o assunzioni di formattazione.

Interface evolution

Un **interfaccia** è un **contratto** tra gli **elementi** e i suoi **attori**.

Le tre tecniche per modificare un'interfaccia sono:

- **Deprecazione** → Rimozione di un'interfaccia
  - La migliore pratica quando si depreca un'interfaccia è dare ampio preavviso agli attori dell'elemento.
- **Versioning** → Aggiunta di più interfacce (sostituendo quella precedente)
  - Supporta l'evoluzione mantenendo l'interfaccia vecchia e aggiungendone una nuova.
  - L'interfaccia vecchia può essere deprecata quando non è più necessaria o non più supportata.
- **Estensione** → Lascia l'interfaccia originale invariata e aggiunge nuove risorse all'interfaccia per incorporare le modifiche desiderate.

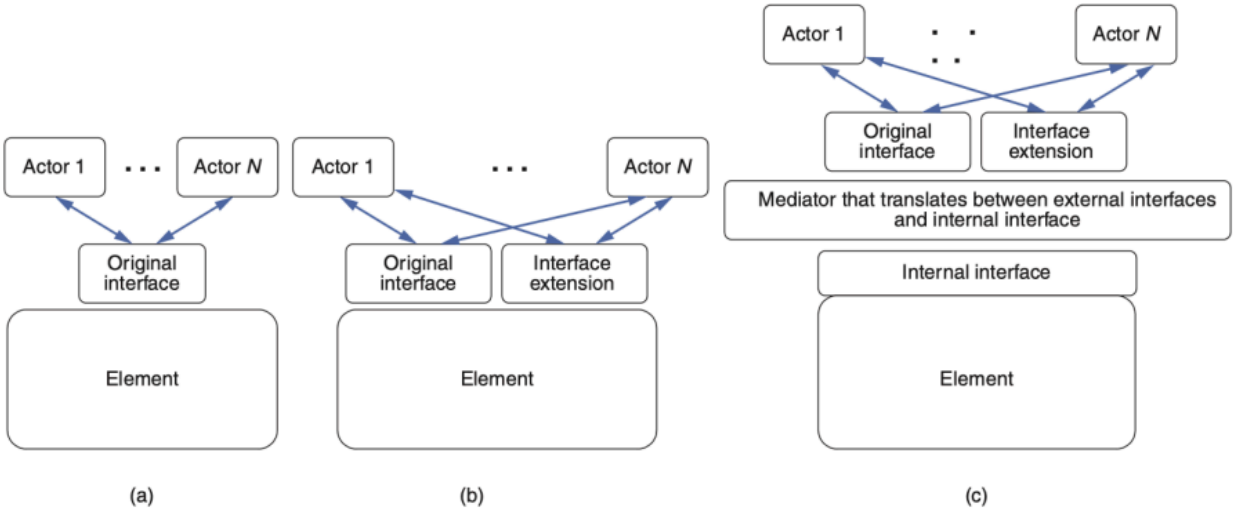


FIGURE 15.1 (a) The original interface. (b) Extending the interface. (c) Using an intermediary.

Il mediatore è utile quando cambio l'interfaccia, l'original interface rimane quella, e poi la richiesta viene mutata ed inviata alla nuova interfaccia.

Designing an interface

Le **risorse** che devono essere visibili esternamente dipendono dalle necessità degli attori che le utilizzano.

Aggiungere risorse a un'interfaccia implica l'impegno a mantenerle **immutate**, quindi dopo che ho definito un'interfaccia, non posso andare a cambiarla. Una volta che gli attori iniziano a dipendere da una risorsa, i loro elementi si romperanno se la risorsa viene modificata o rimossa.

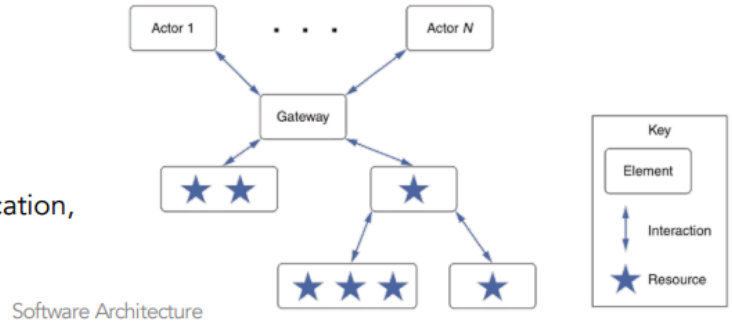
Principi

Alcuni principi di progettazione aggiuntivi per le interfacce sono:

- **Principio della minima sorpresa**
  - Le interfacce dovrebbero comportarsi in modo coerente con le aspettative dell'attore.
- **Principio delle interfacce ridotte**
  - Se due elementi devono interagire, devono scambiarsi la minore quantità di informazioni possibile.
- **Principio dell'accesso uniforme**
  - Evitare di far trapelare dettagli dell'implementazione attraverso l'interfaccia. L'attore non deve sapere da dove arriva il dato.
- **Principio DRY (Don't Repeat Yourself)**
  - Le interfacce dovrebbero offrire un insieme di primitive componibili, piuttosto che molteplici modi ridondanti per raggiungere lo stesso obiettivo. Le interfacce verranno eventualmente messe insieme da chi le usa.
- Una **interazione** di successo con un'interfaccia richiede un accordo sui seguenti aspetti:
  - Scope dell'interfaccia (quali risorse verranno messe disponibili)
  - Stile di interazione (sincrono, asincrono, push, pull...)
  - Rappresentazione e struttura dei dati scambiati (json, xml, binario...)
  - Gestione degli errori

Interaction styles

- The scope of an interface defines the resources available to the actors
- Actors may need access to, or be restricted to, specific subsets of the resources
- A common pattern for constraining and mediating access to resources is to establish a ***gateway*** element
- Gateways are useful for the following reasons:
  - A gateway can translate between elements and actors
  - The specifics of the resources may change over time, and the gateway can provide a more stable interface
    - number, protocol, type, location, properties



Le **interfacce** connettono gli elementi affinché possano comunicare (trasferire dati) e coordinarsi (trasferire il controllo).

Esistono molti modi in cui queste interazioni possono avvenire, a seconda di:

- la combinazione tra comunicazione e coordinazione,
- se gli elementi saranno co-locati o distribuiti in remoto.

Gli stili più ampiamente utilizzati sono **RPC** e **REST**.

Il **Remote Procedure Call (RPC)** è modellato sulle chiamate di procedura (metodo) nei linguaggi imperativi, con la differenza che il metodo chiamato si trova altrove su una rete.

Modalità di funzionamento:

- Il programmatore scrive la chiamata come se si trattasse di un metodo locale (con qualche variazione sintattica).

- La chiamata viene tradotta in un messaggio inviato a un elemento remoto dove viene invocato il metodo effettivo.
- I risultati vengono inviati indietro come messaggio all'elemento chiamante.

Il **Representational State Transfer (REST)** è uno stile architetturale per i servizi web, non un protocollo.

- Un sistema **RESTful** impone sei vincoli sulle interazioni tra elementi:
  - **Interfaccia uniforme**
    - Tutte le interazioni usano la stessa forma (tipicamente HTTP) e le risorse sono specificate tramite URI.
  - **Pattern client-server**
    - Gli attori sono i client, mentre i fornitori di risorse sono i server.
  - **Senza stato (stateless)**
    - Tutte le interazioni client-server sono senza stato.
  - **Cacheable**
    - Le risorse possono essere memorizzate nella cache, quando applicabile.
  - **Architettura a livelli**
    - Il “server” può essere suddiviso in più elementi, che possono essere distribuiti in modo indipendente.
  - **Codice su richiesta (opzionale)**
    - È possibile per il server fornire codice al client affinché lo esegua.

RESTful è un'implementazione pratica che segue i principi REST.

Quando diciamo che un'API è RESTful, intendiamo che aderisce ai principi definiti dal modello REST, come:

- l'uso corretto dei metodi HTTP (GET, POST, PUT, DELETE) per le operazioni CRUD,
- l'uso degli URI per identificare le risorse,

- il mantenimento dello stateless.

REST è lo stile architetturale teorico, mentre **RESTful** si riferisce a sistemi o API che implementano tali principi.

Representation and Structure of Exchanged Data

Ogni **interfaccia** consente di astrarre la **rappresentazione interna** dei dati in una **rappresentazione** più adatta:

- a essere scambiata tra diverse implementazioni di linguaggi di programmazione e
- a essere inviata attraverso la rete.

La conversione dalla rappresentazione interna a quella esterna viene chiamata **serializzazione**, **marshalling** o **traduzione**.

Scegliere come rappresentare i dati scambiati ha le seguenti dimensioni:

- Espressività: La rappresentazione può serializzare strutture di dati arbitrarie? È ottimizzata per alberi di oggetti? Deve trasportare testo scritto in lingue diverse?
- Interoperabilità: La rappresentazione utilizzata dall'interfaccia corrisponde a ciò che i suoi attori si aspettano e sanno come analizzare?
- Prestazioni: La rappresentazione scelta consente un utilizzo efficiente della larghezza di banda di comunicazione disponibile?
- Accoppiamento implicito: Quali sono le assunzioni condivise che potrebbero portare a errori e perdita di dati durante la decodifica dei messaggi?
- Trasparenza: È possibile intercettare i messaggi scambiati e osservare facilmente il loro contenuto?

---

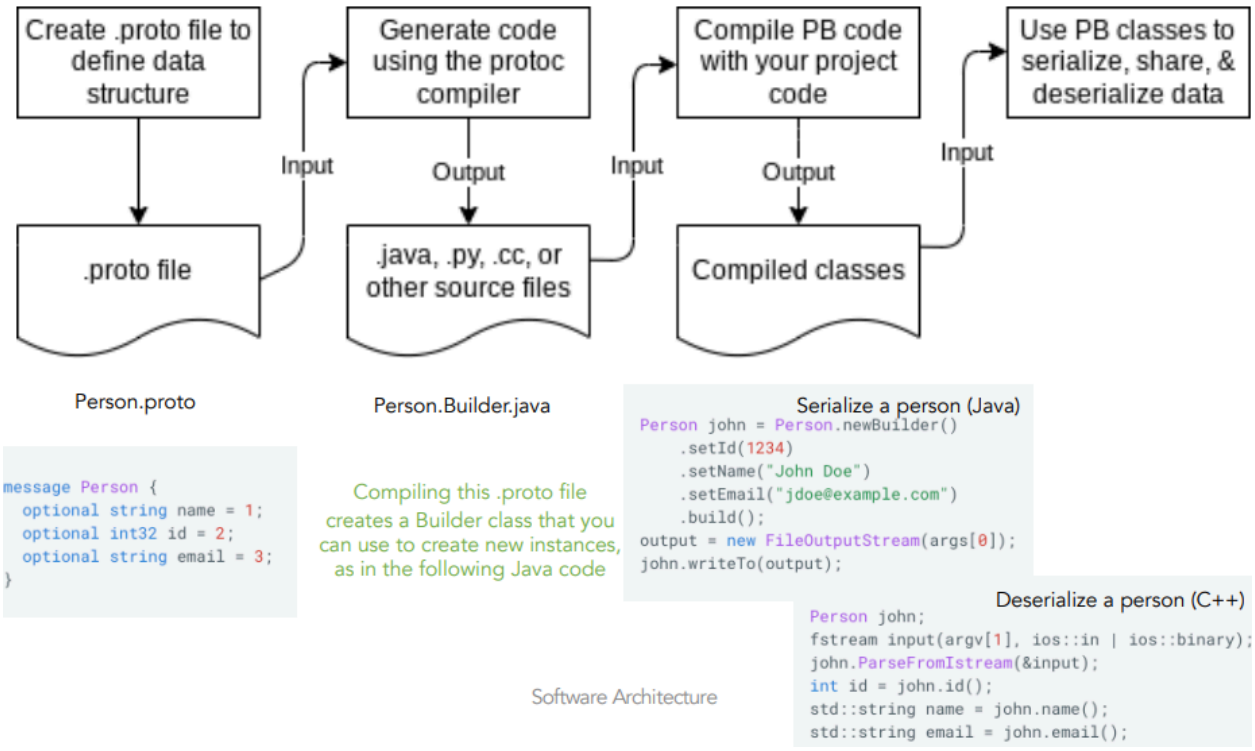
skip XML, JSON

Protocol Buffers

Come il **JSON**, i **Protocol Buffers** utilizzano tipi di dati che sono vicini ai tipi di dati dei linguaggi di programmazione, rendendo la serializzazione e la deserializzazione efficienti.

Come con l'**XML**, i messaggi dei Protocol Buffers hanno uno schema che definisce una struttura valida.

Tuttavia, a differenza sia dell'XML che del JSON, i **Protocol Buffers** sono un formato binario, quindi sono estremamente compatti ed efficienti.



Error handling

Quando si progetta un'interfaccia, gli architetti si concentrano naturalmente su come dovrebbe essere utilizzata nel **caso nominale**.

Ma non ci troviamo in un mondo perfetto, pertanto, un sistema ben progettato deve prendere le opportune misure di fronte a **circostanze indesiderate**.

- Cosa succede quando un'operazione viene chiamata con parametri non validi?
- Cosa succede quando una risorsa richiede più memoria di quella disponibile?
- Cosa succede quando una chiamata a un'operazione non restituisce mai un risultato, perché è fallita?
- ...

Gli **attori** devono sapere se l'elemento sta funzionando correttamente, se la loro interazione ha avuto successo e se si è verificato un errore.

Le strategie per farlo includono:

- Le operazioni fallite possono generare un'**eccezione**.
- Le operazioni possono restituire un **indicatore di stato** con codici predefiniti, che devono essere testati per rilevare esiti errati.
- Possono essere utilizzate **proprietà** per memorizzare dati che indicano se l'ultima operazione è stata eseguita con successo o meno, o se elementi a stato sono in uno stato errato.
- Possono essere attivati **eventi di errore** (come un timeout) per interazioni asincrone fallite.
- Il **registro degli errori** può essere letto collegandosi a un flusso di dati di output specifico.

La specifica di quali **eccezioni**, quali **codici di stato**, quali **eventi** e quali informazioni vengono utilizzati per descrivere esiti errati diventa parte dell'interfaccia di un elemento.

Le **fonti comuni di errori** includono:

- Informazioni **errate**, **non valide** o **illecite** inviate all'interfaccia.
    - Ad esempio, chiamare un'operazione con un parametro di valore null che non dovrebbe essere null.
- L'elemento si trova nello **stato sbagliato** per gestire la richiesta.
    - Ad esempio, 1) invocare un'operazione o leggere una proprietà prima che l'inizializzazione dell'elemento sia completata; 2) scrivere su un dispositivo di archiviazione che è stato messo offline dall'operatore umano del sistema.
- Si è verificato un **errore hardware o software** che ha impedito all'elemento di eseguire correttamente.
    - Ad esempio, guasti del processore, guasti della rete e impossibilità di allocare più memoria.
- L'elemento non è **configurato correttamente**.

    - Ad esempio, la stringa di connessione al database fa riferimento al server di database sbagliato.

Documentation

La **documentazione dell'interfaccia** indica cosa devono sapere gli altri sviluppatori per utilizzare un'interfaccia.

Quando si documenta l'interfaccia di un elemento, è importante tenere a mente i seguenti ruoli degli **stakeholder**:

- Sviluppatore dell'elemento
- Manutentore
- Sviluppatore di un elemento che utilizza l'interfaccia
- Integratore di sistemi e tester
- Analista
- Architetto in cerca di risorse da riutilizzare in un nuovo sistema.

Sommario

Gli **elementi architettonici** hanno interfacce, che sono confini attraverso i quali gli elementi interagiscono. La progettazione dell'interfaccia è un compito architettonico.

Un uso principale di un'interfaccia è quello di **incapsulare un'implementazione**, in modo che possa cambiare senza influenzare altri elementi.

Le interfacce specificano quali risorse l'elemento fornisce ai suoi attori e quali risorse l'elemento necessita dal suo ambiente.

Le interfacce hanno **operazioni**, **eventi** e **proprietà**; queste sono le parti di un'interfaccia che l'architetto può progettare. Per farlo, l'architetto deve decidere riguardo a:

- **Ambito dell'interfaccia**
- **Stile di interazione**
- **Rappresentazione, struttura e semantica dei dati scambiati**
- **Gestione degli errori**.

Alcuni di questi problemi possono essere affrontati con mezzi standardizzati. Ad esempio, lo scambio di dati può utilizzare meccanismi come **XML**, **JSON** o **Protocol Buffers**.

- Tutto il software evolve, comprese le interfacce. Tre tecniche che possono essere utilizzate per modificare un'interfaccia sono la **deprecazione**, il **versioning** e l'**estensione**.
- La **documentazione dell'interfaccia** indica cosa devono sapere gli altri sviluppatori per utilizzare un'interfaccia in combinazione con altri elementi.
- Documentare un'interfaccia implica decidere quali operazioni, eventi e proprietà dell'elemento esporre agli attori dell'elemento e dettagliare la sintassi e la semantica dell'interfaccia.

Virtualizzazione

La virtualizzazione nasce per un problema di mancanza di risorse.

Le **macchine virtuali** e, successivamente, i **container** sono emersi per gestire la **condivisione delle risorse**.

L'obiettivo delle macchine virtuali e dei container è quello di **isolare** un'applicazione da un'altra, pur continuando a **condividere** risorse.

La condivisione delle risorse può ridurre notevolmente i costi di distribuzione di un sistema.

Come architetto:

- potresti essere chiamato a utilizzare una forma di **virtualizzazione** per distribuire il software che crei.
- Se sei tenuto a distribuire su **hardware specializzato**, la virtualizzazione consente di eseguire test in un ambiente molto più accessibile rispetto all'hardware specializzato.

Ci sono quattro risorse di cui ci preoccupiamo tipicamente nella condivisione:

- Unità di elaborazione centrale (CPU)
- Memoria

- Archiviazione su disco
- Connessioni di rete

Come possiamo condividere le risorse in modo sufficientemente "isolato" affinché le diverse applicazioni non siano consapevoli dell'esistenza l'una dell'altra?

Central processing

Condivisione:

- La **condivisione del processore** è ottenuta attraverso un meccanismo di **pianificazione dei thread**.
- Lo **scheduler** seleziona e assegna un thread di esecuzione a un processore disponibile, e quel thread mantiene il controllo fino a quando il suo processore non viene ripianificato.
- La **ripianificazione** avviene quando il thread cede il controllo del processore, quando scade un intervallo di tempo fisso o quando si verifica un'interruzione.

Isolamento:

- Nessun thread di applicazione può ottenere il controllo di un processore senza passare attraverso la **pianificazione**.

Memoria:

Condivisione:

- Con l'aumentare delle dimensioni delle applicazioni, tutto il codice e i dati non potevano essere caricati nella memoria fisica.
- L'**unità di gestione della memoria** (MMU) partiziona lo spazio di indirizzamento di un processo in **pagine** e scambia le pagine tra la memoria fisica e la memoria secondaria secondo necessità.
- Le pagine che si trovano nella memoria fisica possono essere **accessibili immediatamente**, mentre le altre pagine sono memorizzate nella memoria secondaria fino a quando non sono necessarie.

Isolamento:

- La **MMU** associa le pagine a un solo processo e solo la MMU può gestire il loro **scambio**.

Archiviazione su Disco:

Condivisione:

- I **meccanismi hardware** consentono di condividere lo spazio su disco tra più processi.

Isolamento:

- Il **sistema operativo** etichetta i thread in esecuzione e i contenuti del disco con informazioni (come un ID) e limita l'accesso confrontando le etichette del thread che richiede l'accesso e i contenuti del disco.
- I dischi fisici possono essere accessibili solo attraverso un **controller del disco**, che garantisce che i flussi di dati da e verso ciascun thread siano consegnati in sequenza.

Rete:

Condivisione:

- I **protocolli di rete** gestiscono la condivisione dei canali di comunicazione.

Isolamento:

- L'**isolamento della rete** è ottenuto attraverso l'identificazione dei messaggi, supportato da:
  - **IP:**
    - Ogni macchina virtuale (VM) o container ha un **indirizzo Internet Protocol (IP)**, utilizzato per identificare i messaggi provenienti o destinati a quella VM o container.
    - In sostanza, l'indirizzo IP viene utilizzato per instradare le risposte alla VM o al container corretto.
  - **Porta:**
    - Ogni messaggio destinato a un servizio ha un **numero di porta** associato.

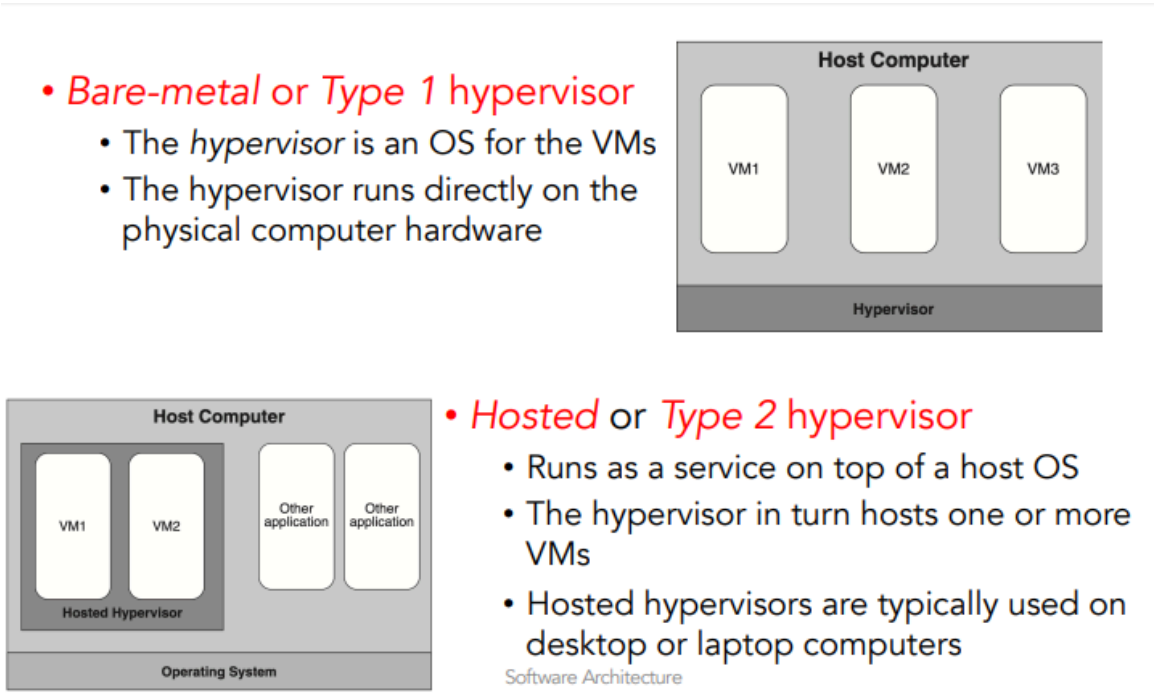
- Un servizio ascolta su una porta e riceve messaggi che arrivano al dispositivo su cui il servizio è in esecuzione, designati per la porta su cui il servizio sta ascoltando.

Virtual machines

Le **macchine virtuali** (VM) consentono l'esecuzione di più computer virtuali su un unico computer fisico.

- Il computer fisico è chiamato **computer host**.
- Le VM sono chiamate **computer guest**.
- L'**hypervisor** è un sistema operativo progettato per gestire le VM.

Tipi



Virtualbox è di tipo 2.

Un **hypervisor** richiede che le sue VM guest utilizzino lo stesso set di istruzioni della CPU fisica sottostante.

Gli **emulatori** supportano l'esecuzione su processori diversi.

- Un emulatore legge il codice binario per il processore target e simula l'esecuzione delle istruzioni guest sul processore host.
- L'emulatore spesso simula anche i dispositivi hardware di I/O guest.
- Ad esempio, l'emulatore open source **QEMU** può emulare un intero sistema PC, inclusi BIOS, processore x86 e memoria, scheda audio, scheda grafica e persino un'unità floppy disk.

Un hypervisor svolge due funzioni principali:

1. **Gestione del Codice in Ogni VM:**
    - Il codice che comunica al di fuori della VM accedendo a un disco o a un'interfaccia di rete viene **intercettato** dall'hypervisor e **eseguito** dall'hypervisor per conto della VM.
2. **Gestione delle VM Stesse:**
    - Le VM devono essere gestite, ovvero **create**, **distrutte** e **monitorate**.
    - Il processo di creazione di una VM prevede il **caricamento di un'immagine della VM**.
    - Dalla prospettiva del sistema operativo e dei servizi all'interno di una VM, sembra che il software stia eseguendo all'interno di una macchina fisica.
    - La VM fornisce una **CPU**, **memoria**, **dispositivi I/O** e una **connessione di rete**.

Uno degli aspetti critici delle VM è l'**overhead** introdotto dalla condivisione e dall'isolamento necessari per la virtualizzazione.

Implicazioni per l'Architetto

- **Prestazioni:**
    - La virtualizzazione comporta un costo di prestazione. Ciò significa che le applicazioni eseguite su VM possono avere una latenza e una reattività inferiori rispetto a quelle eseguite su hardware fisico diretto.
- **Separazione delle preoccupazioni:**
    - La virtualizzazione consente a un architetto di trattare le risorse di runtime come **commodities**, rimandando le decisioni di provisioning e deployment

a un'altra persona o organizzazione. Questo approccio può semplificare la progettazione e la gestione dell'architettura, consentendo una maggiore flessibilità e scalabilità.

Le **macchine virtuali (VM)** risolvono il problema della condivisione delle risorse e del mantenimento dell'isolamento, ma le immagini delle VM possono essere **grandi** e il loro trasferimento può richiedere tempo.

I **containers** mantengono la maggior parte dei vantaggi della virtualizzazione, riducendo tuttavia il tempo di trasferimento delle immagini e il tempo di avvio.

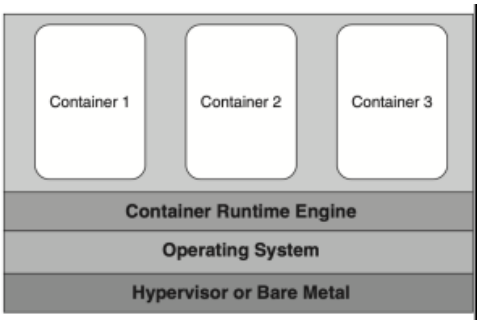
Poiché i contenitori **utilizzano** il sistema operativo della macchina host, riescono a essere più leggeri e più rapidi rispetto alle VM.

Come le VM (con le immagini delle VM), i contenitori sono **imballati** in immagini containerizzate eseguibili per il trasferimento.

I **contenitori** operano sotto il controllo di un **motore di runtime dei contenitori**, che funziona sopra un sistema operativo fisso.

Questo sistema operativo può essere caricato sia su una **macchina fisica bare-metal** che su una **macchina virtuale**.

I contenitori vengono allocati trovando un motore di runtime dei contenitori che dispone di **risorse inutilizzate sufficienti** per supportare un contenitore aggiuntivo.



What are the tradeoffs between delivering your service in a VM and delivering your service in a container?

- VM
  - The software that you run on the VM includes an entire OS
  - This allows you to run **multiple services** in the same VM
    - a desirable outcome when the services are tightly **coupled** or share large data sets
  - Run legacy or purchased sw
- Containers
  - Container instances share an OS
  - Containers generally run a **single service**, so the size of the container image is small

- VMs persist beyond the termination of services running within them; containers do not

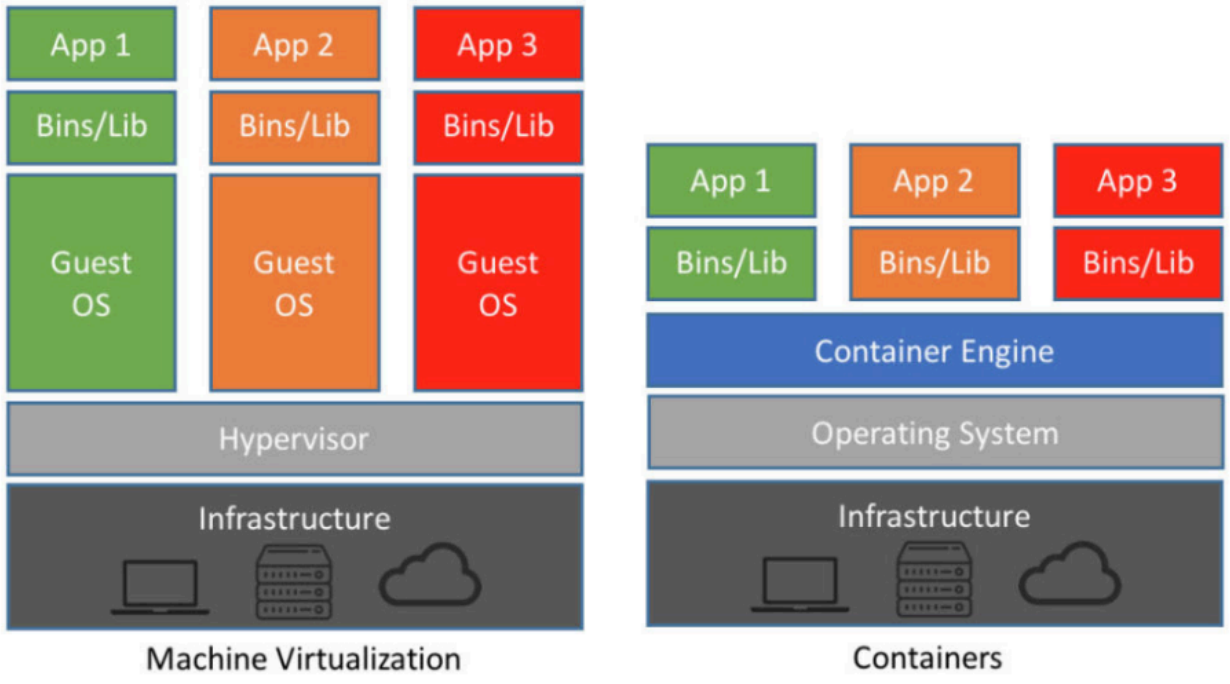
- Other differences between VMs and containers are as follows:
  - SO
    - VM can run any operating system
    - Containers are currently limited to Linux, Windows, or macOS
  - Services
    - Within VMs are started, stopped, and paused through **operating system functions**
    - Within containers are started and stopped through **container runtime engine functions**
  - Persistence
    - VMs persist beyond the termination of services running within them
    - Containers do not persist
  - …

Key Differences Between Containers and Virtual Machines

- A *container* is a lightweight virtualization technology that allows applications and their dependencies to run in isolated environments while sharing the host's OS kernel. Containers are fast, portable, and use fewer resources compared to virtual machines (VMs), which virtualize an entire operating system

1. **Architecture**:
    1. **Containers** share the host OS kernel and virtualize only the user space, making them lightweight.
    2. **VMs** use a hypervisor to run a full OS, including its own kernel, which consumes more resources.
2. **Performance**:
    1. **Containers** start quickly and use fewer resources.
    2. **VMs** are slower to start and require more resources due to the full OS virtualization.
3. **Isolation**:
    1. **Containers** offer process-level isolation, which is lighter but less secure since they share the host kernel.
    2. **VMs** provide stronger isolation as each has its own OS, making them more secure but heavier.
4. **Portability**:
    1. **Containers** are highly portable across environments (e.g., development, testing, production) if container runtime is supported.
    2. **VMs** are less portable and can require adjustments when moving between hypervisors.
5. **Use Cases**:
    1. **Containers** are ideal for microservices and cloud-native applications that need fast scalability and portability.
    2. **VMs** are suited for legacy or monolithic applications requiring full OS isolation.

- In summary, containers are faster, more portable, and resource-efficient, while VMs provide more complete isolation at the cost of higher resource usage and slower startup



Quindi i containers permettono di far girare più servizi sullo stesso sistema operativo.

Serverless architecture

Ci sono server che ospitano motori di esecuzione dei container.

I container vengono allocati dinamicamente con ogni richiesta di servizio.

Gli sviluppatori non sono responsabili per l'allocazione o la deallocazione di essi.

Le caratteristiche del provider di servizi cloud che supportano questa capacità sono chiamate funzione come servizio (FaaS).

Questo approccio è chiamato architettura senza server.

Questi container sono tipicamente senza stato.

Sommario

- La virtualizzazione è stata una benedizione per gli architetti di software e sistemi, poiché fornisce piattaforme di allocazione efficienti e convenienti per i servizi in rete (tipicamente basati sul web).
- La virtualizzazione hardware consente la creazione di diverse macchine virtuali che condividono la stessa macchina fisica. Questo avviene mentre viene garantita l'isolamento della CPU, della memoria, dello spazio di archiviazione e della rete.
- Di conseguenza, le risorse della macchina fisica possono essere condivise tra diverse VM, mentre il numero di macchine fisiche che un'organizzazione deve acquistare o affittare è ridotto al minimo.
- Un'immagine VM è l'insieme di bit che vengono caricati in una VM per abilitarne l'esecuzione.
- I container sono un meccanismo di impacchettamento che virtualizza il sistema operativo. Un container può essere spostato da un ambiente a un altro se è disponibile un motore di esecuzione dei container compatibile.
- Posizionare diversi container all'interno di un Pod significa che vengono allocati insieme e qualsiasi comunicazione tra i container può avvenire rapidamente.
- L'architettura serverless consente di istanziare rapidamente i container e sposta la responsabilità per l'allocazione e la deallocazione al fornitore di servizi cloud.