Lezione 3 08/10/2024

The importance of Software Architecture

1. An architecture will inhibit or enable a system's quality attributes
2. The decisions made in an architecture allow you to reason about and manage change as the system evolves
3. The analysis of an architecture enables early prediction of a system's qualities
4. A documented architecture enhances communication among stakeholders
5. The architecture is a carrier of the earliest and hence most fundamental, hardest-to-change design decisions
6. An architecture defines a set of constraints on subsequent implementation
7. The architecture dictates the structure of an organization, or vice versa
8. An architecture can provide the basis for evolutionary prototyping
9. An architecture is the key artifact that allows the architect and project manager to reason about cost and schedule
10. An architecture can be created as a transferable, reusable model that form the heart of a product line
11. Architecture-based development focuses attention on the assembly of components, rather than simply on their creation
12. By restricting design alternatives, architecture channels the creativity of developers, reducing design and system complexity
13. An architecture can be the foundation for training a new team member

7. significa che per esempio le scelte possono dipendere dalle competenze degli sviluppatori

Per approfondire:

> 03_ArcSw-13 Reasons (chapter 02)_Blended.pdf

Structures, Views, Viewpoints

Le attività di design, motivazione e comunicazione dell'architettura possono essere supportate dalla **modellazione** dell'architettura.

Una **descrizione architetturale** è un insieme di **modelli** usati per descrivere un'architettura.

Una **descrizione architetturale** è un insieme di **documenti** che descrivono l'architettura. Uso quindi modelli e viste per descrivere l'architettura, anche per valutare l'architettura scelta (se è corretta, etc).

Queste **views** devono descrivere:

- Interessi degli stakeholders
- Scopo del sistema, requisiti funzionali, vincoli, principi rilevanti, proprietà di qualità
- Scelte che sono state scartate, e perchè
- Una giustificazione logica dell'architettura

Ciascuna view descrive un aspetto **diverso** dell'architettura. Insieme, descrivono l'intero sistema e descrivono come il sistema può raggiungere i suoi obiettivi.

L'architettura software è un insieme di **strutture**. Ciascuna struttura definisce degli elementi e le proprietà. Non ho solo strutture che modellano componenti software, ma anche l'ambiente (anche i team di sviluppo). Una **view** è la descrizione (rappresentazione) di una **struttura** (relazione 1-1) ed è usata da alcuni stakeholder.

Ciascuna view cerca di descrivere un insieme di elementi e relazioni, rilevanti su uno specifico interesse.

Ci sono 3 **categorie di strutture**, che modellano diversi aspetti del sistema. Queste corrispondono a tre tipi di decisioni riguardanti la struttura del sistema.

Strutture a moduli

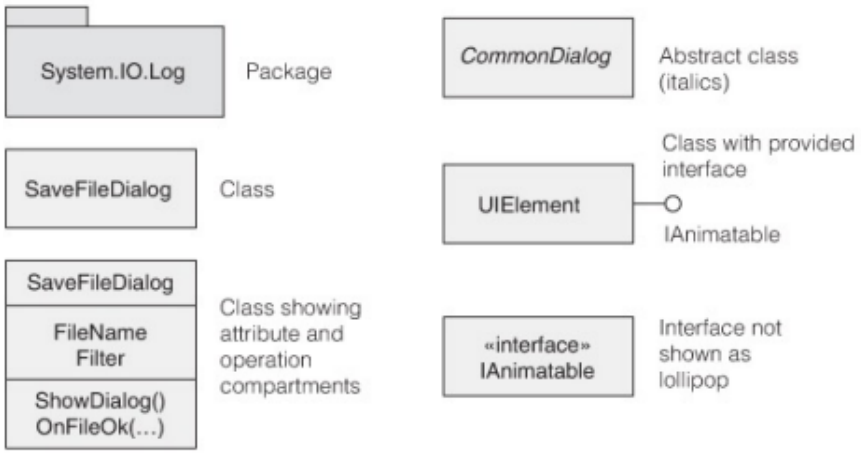
La **struttura a moduli** si concentra sull'organizzazione del software in unità **statiche** di codice chiamate moduli. Quindi non prende in considerazione la dinamicità del sistema.

L'**obiettivo** è quello di definire il sistema software finale, come è organizzato in moduli. Ogni modulo **rappresenta** una parte del sistema, con le sue responsabilità. I moduli sono collegati da dipendenze, ereditarietà o import. (per esempio un'applicazione web potrebbe avere moduli: uno per fare l'UI, quindi la

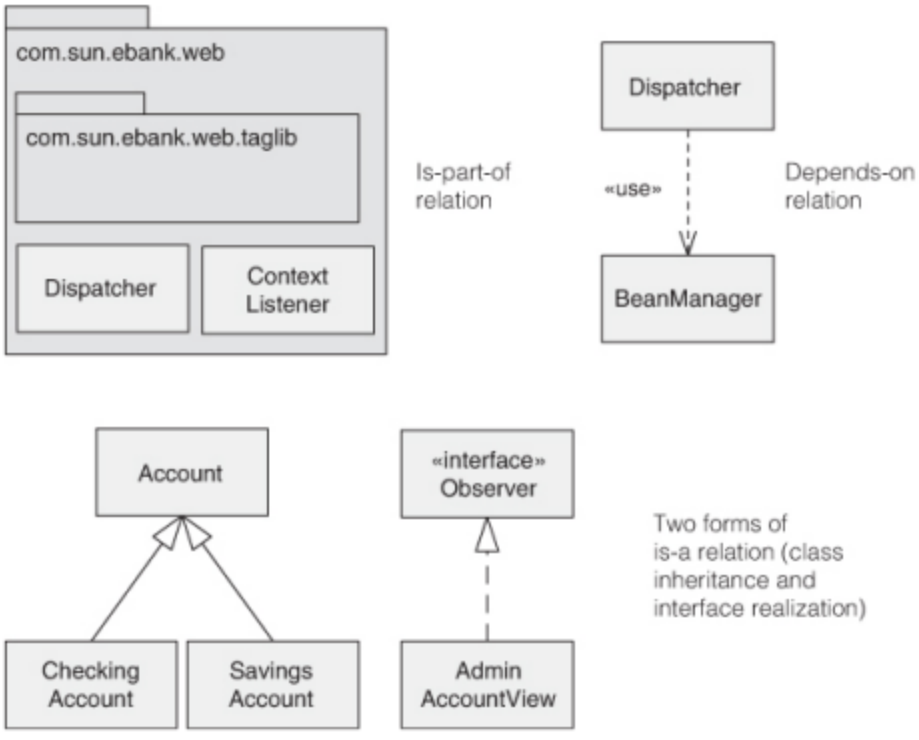
presentazione dei dati, uno per il data processing e un altro che fa l'accesso al database).

Gli elementi di queste strutture a moduli si chiamano moduli. Ne abbiamo di diversi tipi: classi, layers, divisi per funzionalità.

Rappresentazione grafica di una struttura a moduli:



E poi ci sono le relazioni:



Posso usare le classi come formalismi, ma rappresentano i moduli. Quindi non vuol dire che se ho una classe Dispatcher allora poi avrò la classe su java chiamata Dispatcher, può essere, ma di base no. Avrò un insieme di classi che in maniera organica realizzano il dispatcher. Uso il formalismo delle classi giusto perchè è noto, potrei usare un formalismo di forme diverso, inventato.

Quando si inizia a progettare l'idea di massima di architettura, l'idea iniziale, di solito si inizia con una struttura a moduli (suddividendo le responsabilità), applicando un pattern architetturale (per esempio a layer o a microservizi).

La struttura a moduli definisce qual è la responsabilità primaria di ciascun modulo, quali altri elementi può utilizzare, da quali altri moduli dipende, quali moduli sono in relazioni con quali e che relazioni sussistono (generalizzazione, specializzazione, ...).

La struttura a moduli è cruciale per poter rispondere a domande relative alle proprietà strutturali del sistema, come la **modificabilità**.

Strutture a componenti e connettori

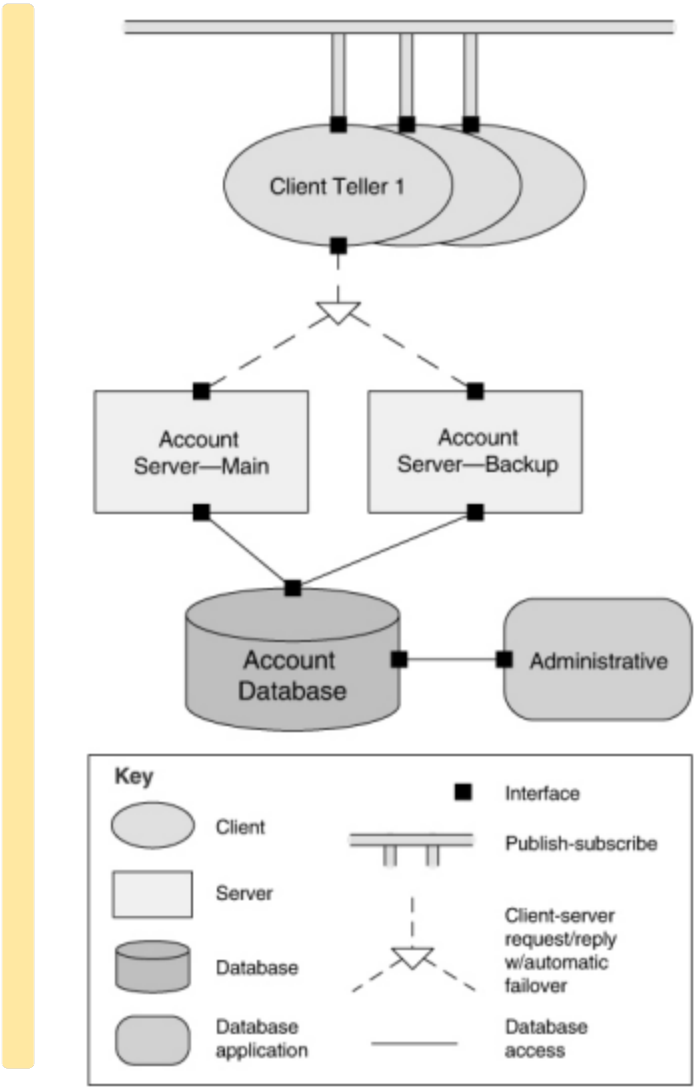
Descrive le interazioni a runtime (dinamiche) tra i componenti del sistema (non più moduli) e i connettori che gestiscono la comunicazione, l'attivazione o il flusso dati. Un componente è un'unità indivisibile di deployment, è come se fosse un .exe, sono unità funzionali come servizi. Noi andremo a dire che un programma che gira su un nodo computazionale è fatto da più componenti che interagiscono tra di loro.

I **connettori** rappresentano i meccanismi attraverso i quali i componenti interagiscono (chiamate di metodo, eventi, messaggi). L'**obiettivo** è quello di mostrare come i componenti interagiscono tra di loro a runtime. **Relazioni**: i connettori gestiscono le comunicazioni fra i componenti, che quindi possono avvenire attraverso protocolli, messaggi, chiamate api... per esempio in un'architettura a microservizi, i componenti (services) comunicano tramite REST API o sistemi di messaggio (connettori).

Quindi se in una struttura a moduli, io mostro le dipendenze tra i moduli, quando parlo di componenti e connettori, vado a dire le istanze che posso avere a runtime dei moduli e vado a dire in maniera esplicita come questi componenti interagiscono.

I componenti possono avere una relazione 1 a molti rispetto ad un modulo, per esempio nell'autenticazione, che avrà una relazione "uses".

In qualsiasi struttura componenti-e-connettori, gli elementi sono componenti di un certo tipo: unità funzionali, come servizi, oggetti o processi. Le relazioni sono concretizzati da chiamate di metodo, eventi o messaggi.



può essere utile mettere una legenda. Bisogna essere coerenti al disegno utilizzato.

Le strutture componente-e-connettore rispondono a **domande** come:

- Quali sono i componenti più importanti e come interagiscono a runtime?

- Quali sono i principali dati condivisi?
- Quali sono le parti del sistema che sono replicate?
- Il dato come progredisce nel sistema?
- Quali parti del sistema sono in parallelo?

Le strutture **componente-e-connettore** sono di cruciale importanza per farsi domande riguardanti le proprietà del sistema a **runtime**, come la **performance, sicurezza e disponibilità**.

Strutture di allocazione

Permette di mappare gli elementi software ad altri elementi non software come **risorse fisiche o virtuali** (servers, containers, CPU), **entità organizzative** (teams) e **storage** (file systems).

L'**obiettivo** è quello di mostrare la distribuzione dei componenti software nell'ambiente nella sua complessità, includendo anche le strutture organizzative e dove i file sono memorizzati.

Le **relazioni** mostrano come gli elementi software sono collegati alle loro controparti fisiche o organizzative.

Per esempio un servizio di accesso ai dati potrebbe essere allocato ad un server che sta su cloud, che è gestito da un team devops, con i suoi file di configurazione che sono memorizzati su una sezione dedicata del filesystem.

Le strutture di allocazione permettono di rispondere alle seguenti **domande**:

- su quale processore è eseguito ciascun elemento?
- in quale file o directory sono memorizzati i file?
- qual è l'assegnamento ai team di sviluppo dei vari moduli?

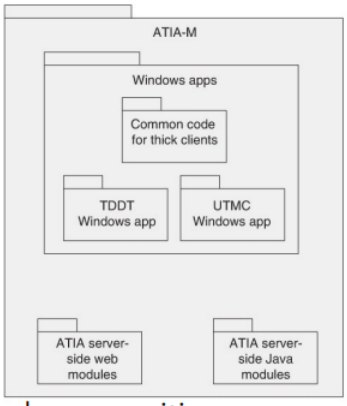
Le **strutture di allocazione** sono importanti per le proprietà runtime come le **performance, security e disponibilità**, ma anche per le proprietà organizzative come il **costo**.

Quindi potrei avere load balancers, potrei avere server replicati...

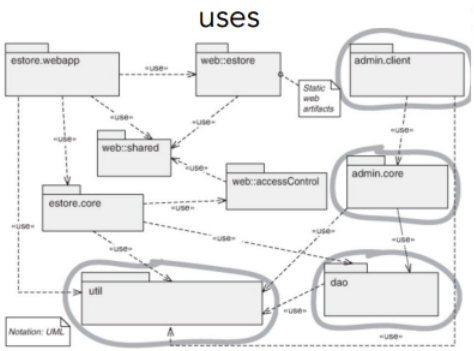
Some useful model, C&C and allocation structures

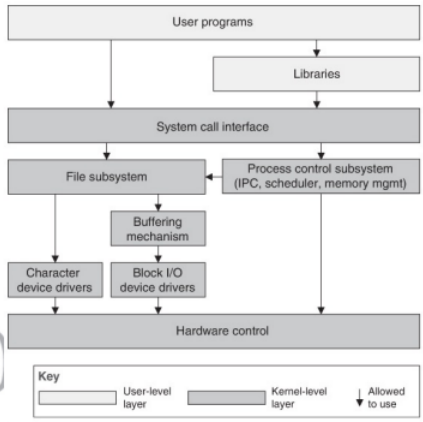
	Software Structure	Element Types	Relations	Useful for	Quality Concerns Affected
Module structures	Decomposition	Module	Is a submodule of	Resource allocation and project structuring and planning; encapsulation	Modifiability
	Uses	Module	Uses (i.e., requires the correct presence of)	Designing subsets and extensions	"Subsetability," extensibility
	Layers	Layer	Allowed to use the services of; provides abstraction to	Incremental development; implementing systems on top of "virtual machines"	Portability, modifiability
	Class	Class, object	Is an instance of; is a generalization of	In object-oriented systems, factoring out commonality; planning extensions of functionality	Modifiability, extensibility
	Data model	Data entity	{one, many}-to-{one, many}; generalizes; specializes	Engineering global data structures for consistency and performance	Modifiability, performance
C&C structures	Service	Service, service registry	Attachment (via message-passing)	Scheduling analysis; performance analysis; robustness analysis	Interoperability, availability, modifiability
	Concurrency	Processes, threads	Attachment (via communication and synchronization mechanisms)	Identifying locations where resource contention exists, opportunities for parallelism	Performance
Allocation structures	Deployment	Components, hardware elements	Allocated to; migrates to	Mapping software elements to system elements	Performance, security, energy, availability, deployability
	Implementation	Modules, file structure	Stored in	Configuration control, integration, test activities	Development efficiency
	Work assignment	Modules, organizational units	Assigned to	Project management, best use of expertise and available resources, management of commonality	Development efficiency

Model structures examples:



decomposition

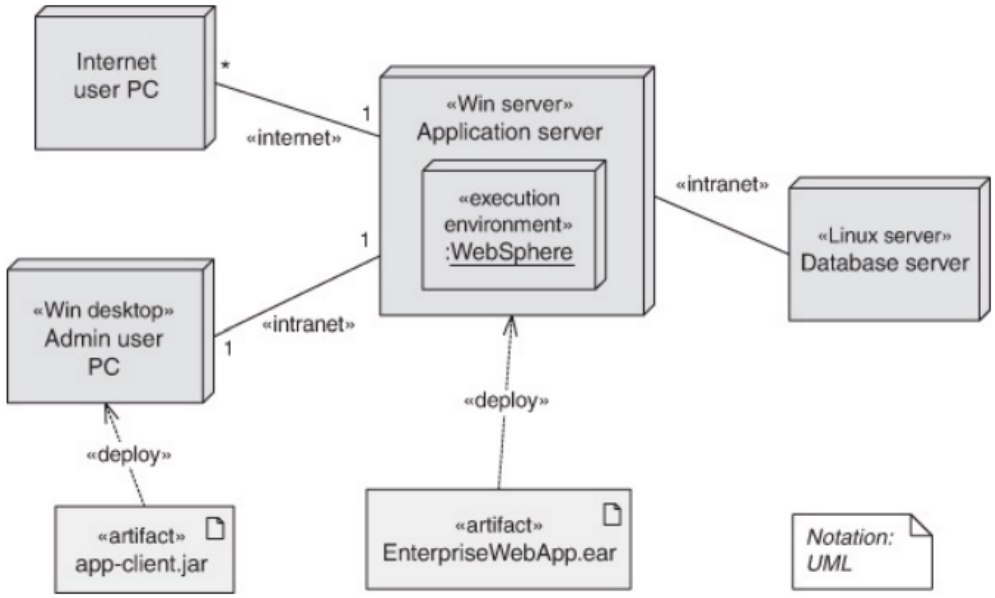




layer

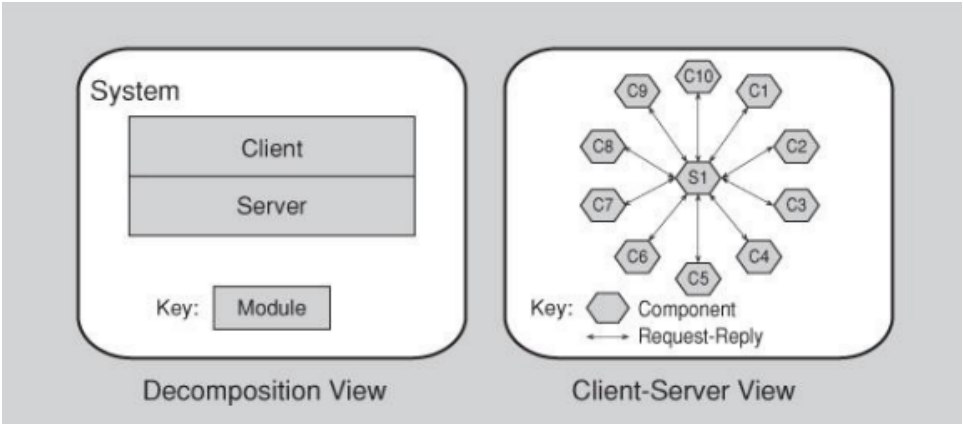
Allocation structure example:

deployment



Relazioni tra strutture

Le diverse strutture che uso mi permettono di andare a trattare e ragionare sul mio sistema da diversi punti di vista. Sto sempre modellando lo stesso sistema, quindi ci devono essere delle relazioni tra le strutture.



Viewpoints

Ora il problema è quello di quali viste creare e come creare ciascuna vista. La risposta a questa domanda sono i **viewpoints** (punti di vista). Questi forniscono

un’approccio strutturato per la progettazione e l’analisi dell’architettura di un sistema. Permettono di guidare nel realizzare quell’insieme di viste che sono particolarmente rilevanti rispetto ad un punto di vista che si vuole andare a modellare.

- Each viewpoint is defined in terms of:
  - The key concerns addressed by that viewpoint
  - The models that can be used – including the elements and relationships involved
  - Activities and guidelines for creating these models
  - Common problem situations
  - Applicability and stakeholders
  - References

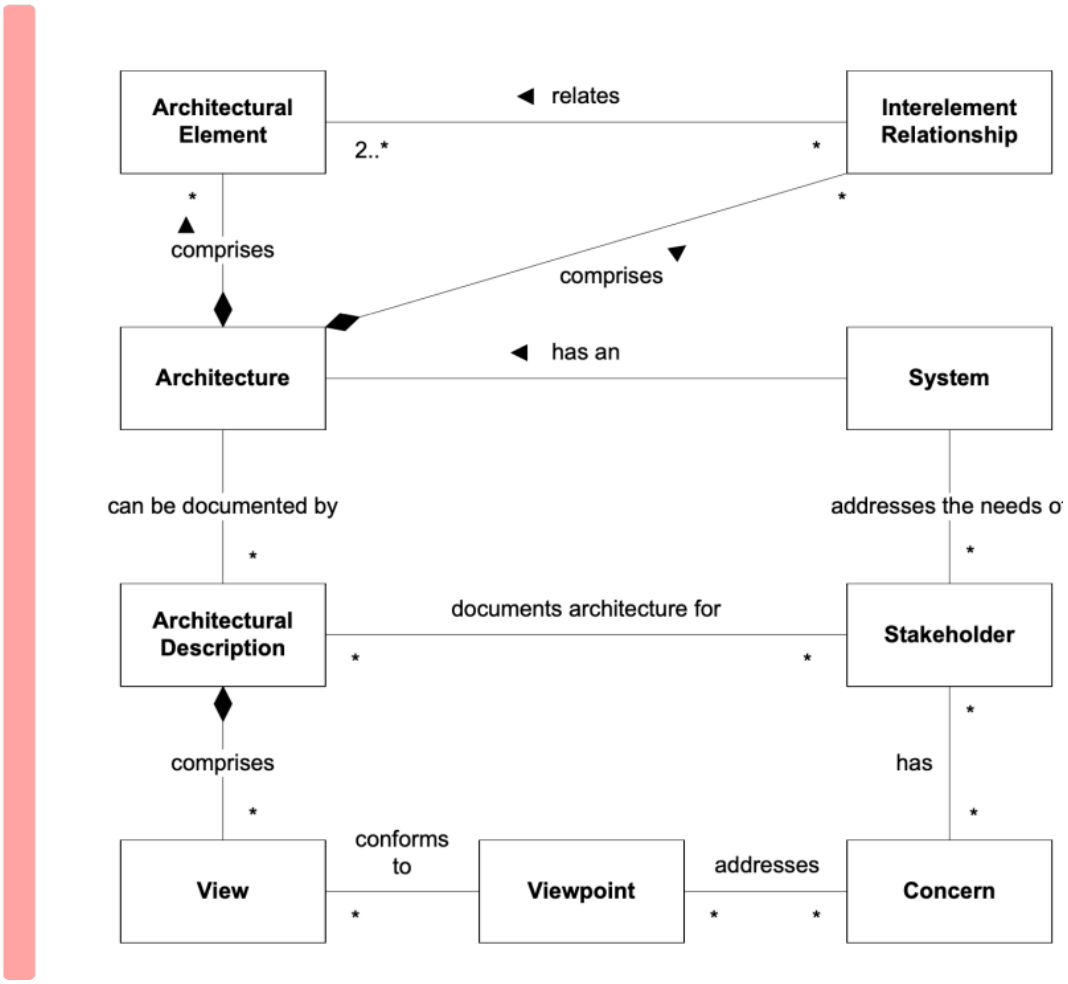
Ogni viewpoint definisce i principi, i modelli e le linee guida per creare viste architetturali rispetto ad uno specifico interesse.

Come esempi di viewpoints abbiamo functional, security, performance...

Esempio di un viewpoint sulla modificabilità:

- Definition
  - Focuses on the ease of changing a system to adapt to new requirements or correct defects, ensuring long-term sustainability
- Key Components:
  - Interests:
    - Add functionalities with minimal impact
    - Correct bugs efficiently
    - Integrate evolving systems seamlessly
  - Models and Guidelines:
    - Microservices Architecture: Enables independent component updates
    - Plugin Design: Facilitates adding/removing features via plugins
    - APIs: Simplifies communication, allowing changes without extensive rewrites
  - View’s Elements:
    - Dependency Diagrams: Visualize interconnections to identify change impacts
    - Change Flows: Outline processes for implementing changes
- Conclusion
  - The modifiability viewpoint ensures efficient changes with minimal impact, enhancing system flexibility and resilience

recap di quello che abbiamo detto fin ora sull’architettura:



Rischi nell'uso dei viewpoints

C'è un rischio di inconsistenza che arriva dall'uso di multipli viewpoints, è difficile mantenere consistenza tra di loro.

Alcuni interessi (concerns) richiedono considerazione da parte di più viewpoints, e quindi diventa difficile considerarli in isolamento.

La soluzione sono gli **scenari**

Scenari

Lo **scenario** racconta una storia. Dicono per esempio, se accade questo evento nel sistema, viene catturato da questo componente, in una situazione tale per cui c'è la cpu totalmente libera, allora deve succedere questa cosa (response).

Attraverso gli scenari quindi tocco le diverse viste.

Gli ASR sono quegli scenari che modellano il soddisfacimento dei requisiti che sono architetturalmente significativi, perchè sono di particolare interesse per uno stakeholder.

Per esempio, devo fare un sistema con autenticazione. Se lo stakeholder è quello che mi compra il sistema e quindi conosce i suoi utente. Se lui sa che i suoi utenti utilizzano la mail google come autenticazione, questo diventa un requisiti architetturalmente significativo.

Uno scenario è una **situazione** in cui il sistema si andrà a trovare nel suo ambiente di produzione, insieme ad una **risposta** che ci si aspetta che il sistema dia.

Abbiamo scenari funzionali, che descrivono la sequenza degli eventi esterni a cui il sistema deve rispondere, e scenari non funzionali, che definiscono come il sistema deve reagire ad una situazione scatenata da un'evento esterno, con un certo livello di qualità (non funzionale).

Lo scenario supporta una vista comprensiva (nel suo insieme) dell'intero sistema.

Understanding quality attributes

Gli scenari sono importanti per poter capire come fare in modo che un certo requisito (funzionale o non) sia soddisfatto.

La **funzionalità** è definita come l'abilità di un sistema di andare a fare ciò per cui è stato sviluppato. I sistemi sono spesso ri-disegnati non perchè mancano di funzionalità ma perchè sono difficili da mantenere, scalare, o sono troppo lenti, o sono stati compromessi da hackers.

Un'**architettura** dovrebbe avere un design che considera un insieme di qualità che il sistema finale dovrebbe avere. Un sistema deve soddisfare un insieme di **requisiti**, dettati dagli **stakeholders**.

- System requirements can be categorized as

Type	Description
Functional requirements	• They state **what** the system must do, **how** it must behave or react to run-time *stimuli*

Quality attribute requirements	• (or better known as Non-functional req.)
	• They provide prescriptions regarding
	  • **functional requirements**
	  - *how fast* the function must be performed, *how resilient* it must be to erroneous input
	  • the **overall product**
	  - the *time to deploy* the product, *limitation* on operational cost

Constraints	• Is a **design decision** with **zero degrees** of freedom • it's a design decision that has already been made for you • Examples include the requirement to use a certain *programming language* or to *reuse a certain existing module*

posso quindi anche fare considerazioni architetturali, avendo già in mente una tecnologia.

IMPORTANTE! Un'**attributo di qualità** è una proprietà **misurabile** e **testabile** di un sistema che è usato per indicare quanto il sistema soddisfi i bisogni degli stakeholders.

- For example, if a functional requirement is "when the user presses the green button the Options dialog appears":
  - A performance QA annotation might describe
    - how quickly the dialog will appear
  - An availability QA annotation might describe
    - how often this function will fail, and **how quickly** it will be repaired
  - A usability QA annotation might describe
    - how easy it is to learn this function

- **Performance**
  - The ability of the system to execute predictably within the required performance profile
- **Security**
  - The ability of the system to resist unauthorized use while continuing to provide services to legitimate users
- **Modifiability**
  - The system's ability to be flexible in response to inevitable changes after the initial release, balanced against the costs of providing such flexibility
- **Reliability**
  - The ability of a system to perform its specified functions over a specified period, without failures
  - Often specified in terms of other qualities, such as availability, fault tolerance, and recoverability
- **Availability**
  - The ability of a system to be fully or partially operational as needed
- **Fault Tolerance/Resilience**
  - The ability of the system to function as required, even when hardware or software components fail
- **Recoverability**
  - The system's ability to recover from failures and restore operations and data within predefined acceptable times
- **Scalability**
  - The ability of the system to handle workload variations without affecting other qualities, especially performance and availability
- **Usability**
  - The ease with which users can operate the system to accomplish tasks, as well as the support the system provides to users
- **Interoperability**
  - The degree to which two or more systems can interact effectively, exchanging meaningful information through interfaces in a specific context, and correctly interpreting the exchanged data

- **Testability**
  - The ease with which software errors can be demonstrated and identified through testing
- **Monitorability/Observability**
  - The ability to observe the system's behavior while running in production environments
- **Deliverability**
  - The process of releasing new software versions to end-users, which must be fast, frequent, and reliable
- **Time to Market**
  - The time needed to develop and release a new system or service, critical under competitive pressure or time-sensitive opportunities
- **Cost**
  - The software development costs, which can be measured in various ways
- **Simplicity**
  - Systems designed with simplicity are easier to implement, test, and evolve

Nel momento in cui definisco l'architettura, questo è il momento di fare in modo che la soluzione riesca a garantire certi requisiti di qualità.

Un **pattern** architetturale permette di definire la struttura di massima del sistema, è il punto di partenza.

Se ad un certo punto la struttura che ho individuato attraverso il pattern architetturale non soddisfa a pieno i requisiti di qualità, io vado ad applicare delle **tattiche** per raffinare.

Una collezione di attività, tattiche e linee guida è usato per assicurarsi che un sistema sia in grado di esibire una certa proprietà di qualità. Questa viene ottenuta tramite il ragionamento su viste che sono scorrelate.