

Lezione 12 26/11/2024

Architecture documentation

Anche la migliore architettura sarà inutile se le persone che ne hanno bisogno non sanno cosa sia, non riescono a capirla abbastanza bene da usarla, costruirla o modificarla, oppure se la fraintendono e la applicano in modo errato.

Creare un'architettura non è sufficiente. Deve essere **comunicata** in modo tale da permettere ai suoi stakeholder di utilizzarla correttamente per svolgere il proprio lavoro.

La documentazione parla per l'architetto oggi, quando l'architetto dovrebbe occuparsi di altre attività invece di rispondere alle domande, e domani, quando l'architetto avrà dimenticato i dettagli o avrà lasciato il progetto.

La documentazione dell'architettura deve:

- Essere sufficientemente **trasparente e accessibile** per essere rapidamente compresa dai nuovi dipendenti.
- Essere sufficientemente **concreta** da fungere da modello per la costruzione.
- Contenere **informazioni** sufficienti per servire come base per la sua analisi.

Comprendere come gli stakeholder utilizzano la documentazione dell'architettura è essenziale: tali utilizzi determinano le informazioni da catturare.

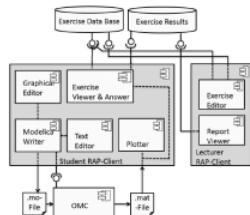
La documentazione dell'architettura può essere usata in 3 modi:

- **Educazione:** introduzione al sistema per nuovi membri del team, analisti o valutatori esterni. Oppure per un nuovo architetto.
- **Veicolo principale per la comunicazione tra gli stakeholder.** Soprattutto tra architetto e sviluppatori, e tra l'architetto attuale e quello futuro.
- **Base per l'analisi e la costruzione del sistema.** L'architettura indica agli implementatori cosa implementare. Ogni modulo definisce le interfacce fornite ad altri moduli e quelle richieste da altri moduli. La documentazione funge da base per la valutazione dell'architettura.

Notazione

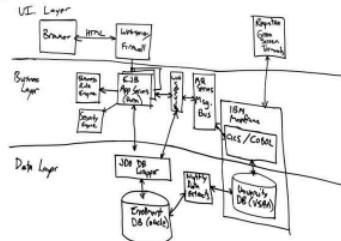
- **Informal notations**

- Views are depicted (often graphically) using general-purpose tools
- The **semantics** of the description are characterized in natural language
- They cannot be formally analyzed



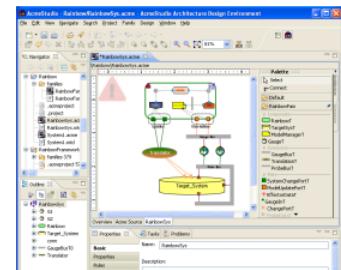
- **Semiformal notations**

- Standardized notation that prescribes graphical elements and rules of construction
- Lacks a complete semantic treatment of the meaning of those elements
- Rudimentary analysis can be applied
- UML is a semiformal notation in this sense



- **Formal notations**

- Views are described in a notation that has a **precise** (usually mathematically based) **semantics**
- Formal analysis of both syntax and semantics is possible
- Architecture description languages (ADLs)
- Support automation through associated tools



Tipicamente, le notazioni più formali richiedono più tempo ed effort per essere create e comprese, ma offrono meno ambiguità e maggiori opportunità di analisi. Invece, le notazioni più informali sono più facili da creare, ma forniscono meno garanzie.

Esempi:

- Un diagramma UML delle classi non ti aiuterà a ragionare sulla schedulabilità.
- Un diagramma di sequenza non ti dirà molto sulla probabilità che il sistema venga consegnato in tempo.

Views

Le **view** sono il concetto più importante associato alla documentazione.

Una **view** è una rappresentazione di un insieme di elementi del sistema e delle relazioni tra di essi (cioè le strutture). Le view ci permettono di suddividere un'architettura software in diverse rappresentazioni interessanti e gestibili del sistema.

Principio della documentazione dell'architettura:

Documentare un'architettura significa rappresentarla attraverso viste distinte ma complementari, che catturano aspetti specifici e rilevanti del sistema.

Le diverse view supportano obiettivi e utilizzi differenti, quindi le view da documentare dipendono dagli usi previsti della documentazione.

Le diverse view evidenziano qualità differenti in misura variabile:

- Una **view dei moduli** permette di ragionare sulla manutenibilità del sistema.
- Una **view di deployment** consente di analizzare le prestazioni e l'affidabilità del sistema.

La scelta delle view è guidata dalla necessità di documentare uno specifico pattern nel tuo design. Alcuni pattern sono composti da moduli, altri da componenti e connettori, e altri ancora presentano considerazioni di allocazione.

Model views

- **Elements**
 - *Modules*, which are implementation units of software that provide a coherent set of responsibilities
- **Relations**
 - *Is part of*, which defines a part/whole relationship between the submodule—the part—and the aggregate module—the whole
 - *Depends on*, which defines a dependency relationship between two modules
 - *Is a*, which defines a generalization/specialization relationship between a more specific module—the child—and a more general module—the parent
- **Constraints**
 - Different module views may impose *specific topological constraints*, such as limitations on the visibility between modules
- **Usage**
 - Blueprint for construction of the code
 - Change-impact analysis (dependencies among modules)
 - Planning incremental development
 - Communicating the functionality of a system and the structure of its code base
 - Supporting the definition of work assignments, implementation schedules, and budget information
 - Showing the structure of information that the system needs to manage

La documentazione di qualsiasi architettura software contiene almeno una view dei moduli. Le view dei moduli sono comunemente mappate su view di componenti e connettori. Le unità di implementazione mostrate nelle view dei moduli hanno una mappatura sui componenti che vengono eseguiti durante il runtime.

Component&component views

- **Elements**

- **Components.** Principal processing units and data stores. A component has a set of *ports* through which it interacts with other components (via connectors)
- **Connectors.** Pathways of interaction between components. Connectors have a set of *roles* (interfaces) that indicate how components may use a connector in interactions

- **Relations**

- **Attachments.** Component ports are associated with connector roles to yield a graph of components and connectors

- **Constraints**

- Components can only be attached to connectors
- Connectors can only be attached to components
- Attachments can only be made between compatible ports and roles
- Connectors cannot appear in isolation: it must be attached to a component

- **Usage**

- Show how the system works
- Guide development by specifying structure and behavior of runtime elements
- Help reason about runtime system quality attributes, such as *performance* and *availability*

Allocation views

- Allocation views describe the mapping of **software units** to **elements of an environment** in which the software is developed or in which it executes

- **Elements**

- **Software element** and **environmental element**
 - A software element has properties that are required of the environment
 - An environmental element has properties that are provided to the software

- **Relations**

- **Allocated to.** A software element is mapped (allocated to) an environmental element

- **Constraints**

- Varies by view

- **Usage**

- Reasoning about *performance*, *availability*, *security*, and *safety*
- Reasoning about *distributed development* and allocation of work to teams
- Reasoning about concurrent access to *software versions*
- Reasoning about the *form* and *mechanisms* of *system installation*

Esempi di views:

- Module Views
 - Decomposition View
 - Uses View
 - Layered View
 - Generalization View
- Component-and-Connector (C&C) Views
 - Shared-Data View
 - Client/service View
 - Pipe-and-Filter View
 - Microservices View
- Allocation Views
 - Deployment View
 - Implementation View

Quality views

Le **view dei moduli**, **C&C** (componenti e connettori), e **di allocazione** sono **view strutturali**. Mostrano principalmente le strutture che soddisfano i requisiti funzionali e delle qualità.

Tuttavia, nei sistemi in cui alcuni attributi di qualità sono particolarmente importanti, le **view strutturali** potrebbero non essere il miglior modo per presentare la soluzione architettonica a tali esigenze, per esempio la soluzione potrebbe essere distribuita, e quindi difficile da rappresentare.

Una **view di qualità** si forma estraendo i pezzi rilevanti dalle **view strutturali** e confezionandoli insieme.

Esempi:

-
- **Security view**
 - Show all the architectural measures taken to provide security
 - Show components that have **security role or responsibility**, how those components communicate, any **data repositories** for security information, **repositories** that are of security interest, ...
 - **Communications view**
 - Especially helpful for systems that are globally **dispersed** and **heterogeneous**
 - Show all the network component-to-component channels, quality-of-service parameter values, areas of concurrency, ...
 - **Exception or error-handling view**
 - Show how components **detect**, **report**, and **resolve** faults or errors
 - It would help identify the sources of errors and appropriate corrective actions for each
 - **Reliability view**
 - Model mechanisms such as replication and switch-over
 - Depicts timing issues and transaction integrity
 - **Performance view**
 - Shows those aspects of the architecture useful for inferring the system's performance
 - Show network traffic models, maximum latencies for operations, and so forth

These and other quality views reflect the documentation philosophy of ISO/IEC/IEEE standard 42010:2011, which prescribes creating views driven by the concerns of the architecture's stakeholders

Documenting Behavior

Un'architettura potrebbe richiedere una documentazione sul comportamento che descriva come gli elementi interagiscono tra loro.

La documentazione sul comportamento permette di ragionare su:

- Il potenziale di deadlock di un sistema.
- La capacità di un sistema di completare un compito nel tempo desiderato.
- Il consumo massimo di memoria.
- ...

Esistono due tipi di notazioni:

- **Lingue orientate ai tracciati**
- **Lingue comprensive**

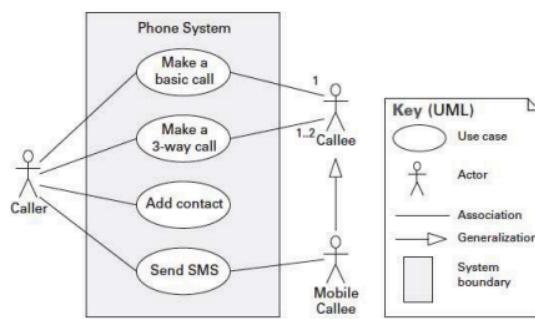
Trace-oriented languages

I **tracciati** sono sequenze di attività o interazioni che descrivono la risposta del sistema a uno stimolo specifico quando il sistema si trova in uno stato specifico.

Esempi:

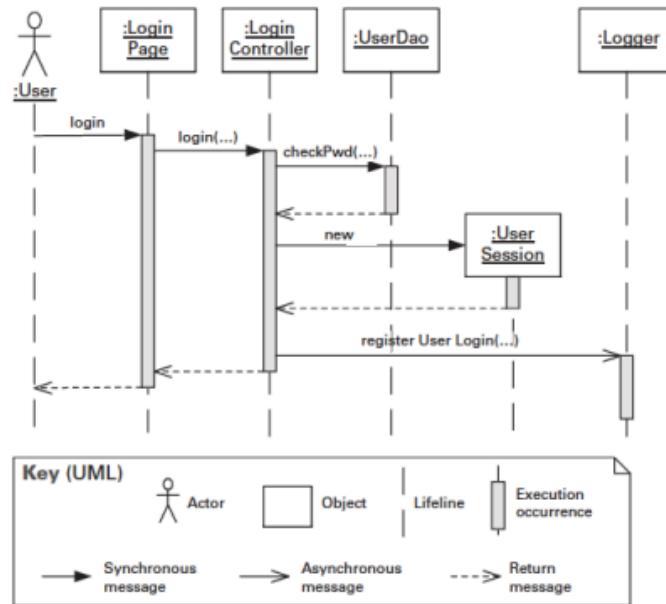
- Casi d'uso
- Diagrammi di sequenza
- Diagrammi di comunicazione
- Diagrammi di attività
- Diagrammi di sequenza dei messaggi
- Diagrammi di temporizzazione
- ...

Esempio use cases:

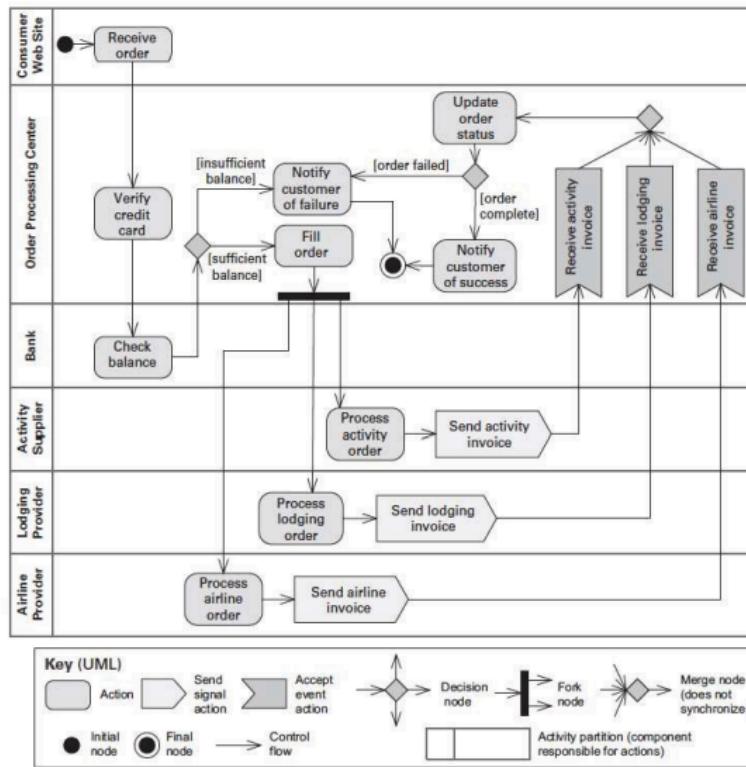


Name: Make a basic call
Description: Making a point-to-point connection between two phones.
Primary actors: Caller
Secondary actors: Callee
Flow of events:
 The use case starts when a caller places a call via a terminal, such as a cell phone. All terminals to which the call should be routed then begin ringing. When one of the terminals is answered, all others stop ringing and a connection is made between the caller's terminal and the terminal that was answered. When either terminal is disconnected—someone hangs up—the other terminal is also disconnected. The call is now terminated, and the use case is ended.
Exceptional flow of events:
 The caller can disconnect, or hang up, before any of the ringing terminals has been answered. If this happens, all ringing terminals stop ringing and are disconnected, ending the use case.

Esempio sequence diagram:



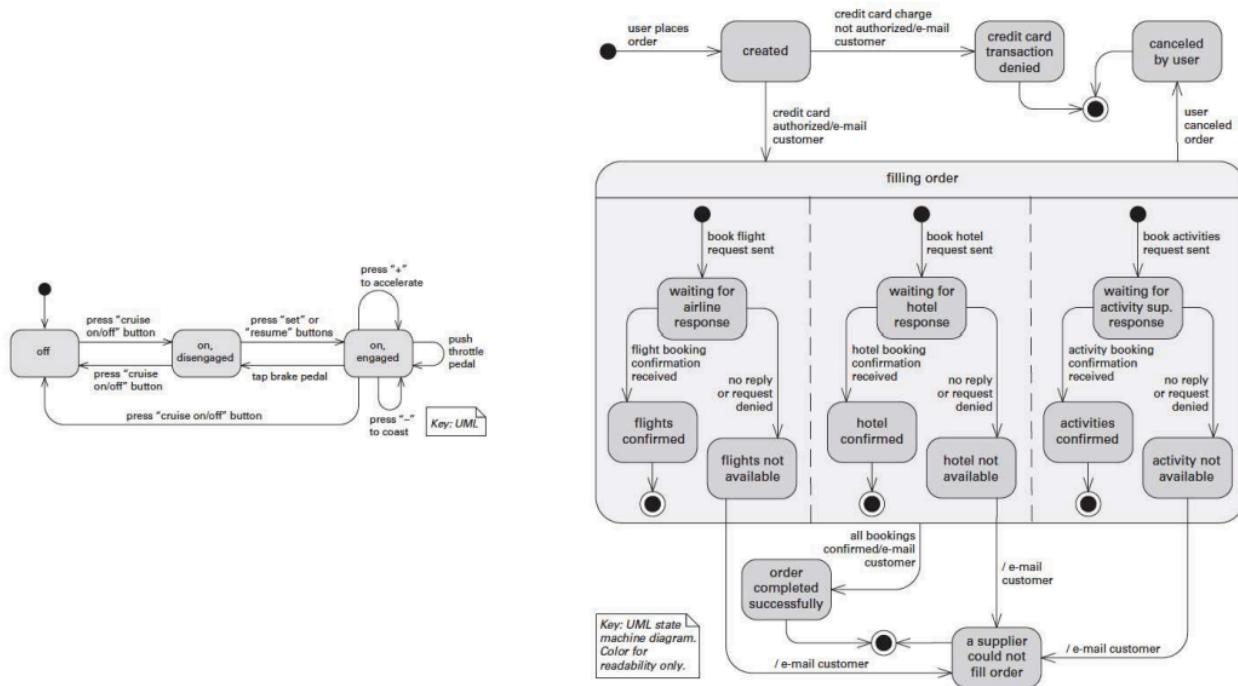
Esempio activity diagram:



Comprehensive Languages

Mostrano il comportamento completo degli elementi strutturali. Con questo tipo di documentazione, è possibile inferire tutti i possibili percorsi dallo stato iniziale allo stato finale.

Esempio: formalismo della macchina a stati. Le lingue delle **macchine a stati** consentono di completare una descrizione strutturale degli elementi del sistema con vincoli sulle interazioni e reazioni temporizzate sia a stimoli interni che esterni.



Choosing the Views

Per determinare quali viste sono necessarie, quando crearle e quanto dettaglio includere, è importante conoscere quanto segue:

- Le persone disponibili e le relative competenze
- Gli standard con cui è necessario conformarsi
- Il budget a disposizione
- La programmazione (schedule)
- Le necessità informative degli stakeholder principali
- I requisiti di qualità predominanti

- Le dimensioni approssimative del sistema

Come minimo, si prevede di avere almeno una vista del modulo, almeno una vista C&C (Componenti e Connettori) e, per sistemi di maggiori dimensioni, almeno una vista di allocazione nel documento architetturale.

Oltre a questa regola di base, esiste un metodo in tre fasi ...

Fase 1. Creare una tabella Stakeholder/Vista

Righe: Elencare gli stakeholder per la documentazione dell'architettura software del progetto

Colonne: Elencare le viste applicabili al sistema

Alcune viste (come decomposizione, utilizzi e assegnazione del lavoro) si applicano a ogni sistema

Altre (come varie viste C&C o la vista stratificata) si applicano solo a determinati sistemi

Compilare ogni cella descrivendo il livello di dettaglio richiesto dallo stakeholder per ciascuna vista: nessuno, solo panoramica, dettaglio moderato o dettaglio elevato

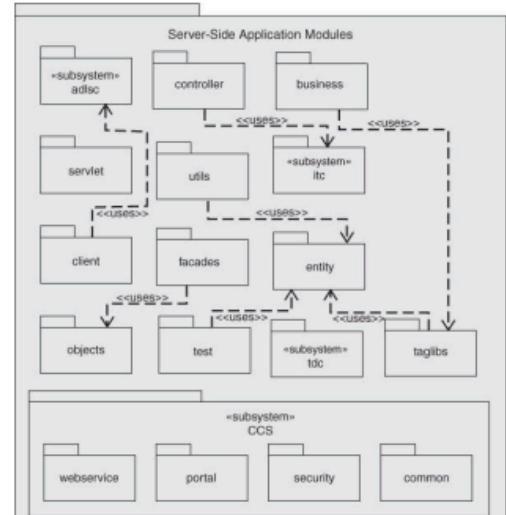
Fase 2. Combinare le viste per ridurne il numero

Identificare le viste marginali nella tabella, ovvero, quelle che richiedono solo una panoramica o che servono un numero molto limitato di stakeholder

Combinare ogni vista marginale con un'altra vista che abbia un pubblico più ampio o più significativo

CONSOLIDACY

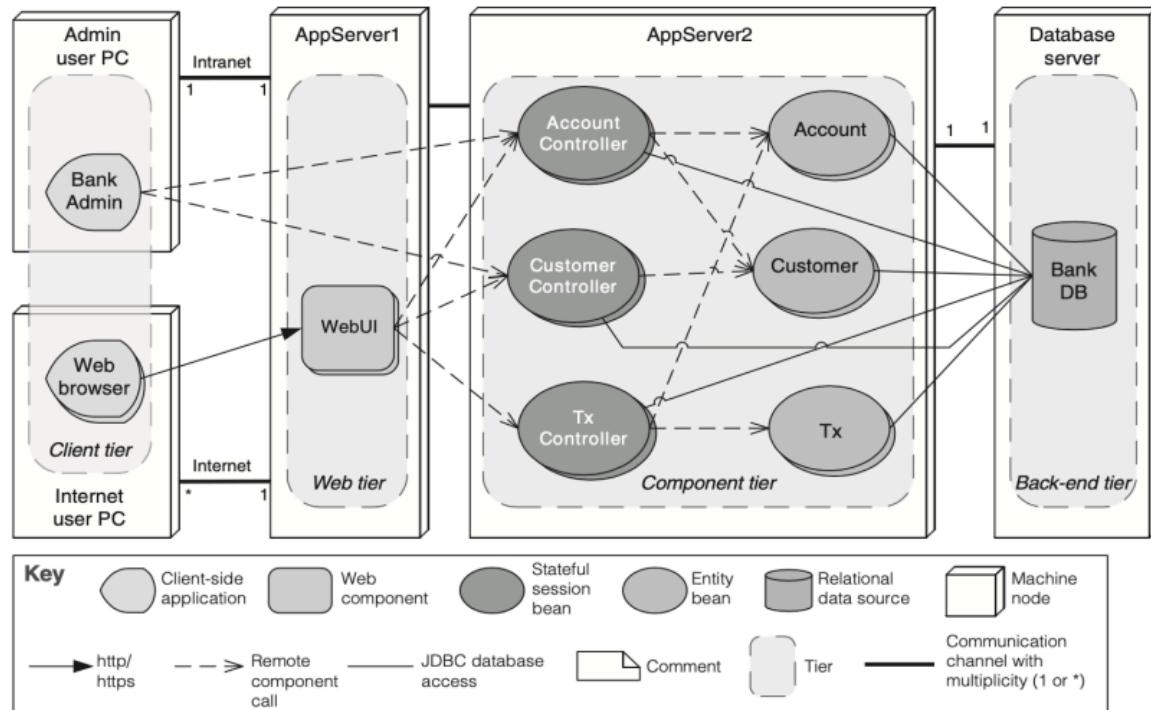
The easiest way to merge views is to create an **overlay** that combines the information that would otherwise have been in two separate views



A volte è conveniente mostrare una vista combinata con elementi e relazioni provenienti da due o più altre viste.

Il modo più semplice per unire le viste è creare una sovrapposizione che combini le informazioni che altrimenti sarebbero apparse in due viste separate.

Esempio:



Fase 3. Dare Priorità

È necessario decidere quale vista sviluppare per prima

Dipende dal progetto, ma alcuni aspetti da considerare:

- La vista di decomposizione è spesso facile da progettare e particolarmente utile
- Il project manager può iniziare a formare i team di sviluppo, avviare la formazione, decidere quali parti esternalizzare e cominciare a elaborare budget e pianificazioni

Non è necessario essere esaustivi

- Fornire l'80% delle informazioni potrebbe essere "sufficiente" affinché gli stakeholder svolgano il proprio lavoro
- Verifica con gli stakeholder se un sottoinsieme di informazioni è adeguato

Non è obbligatorio completare una vista prima di iniziare un'altra

- Le persone possono fare progressi anche con informazioni di livello panoramico
- Un approccio in larghezza prima della profondità è spesso il migliore

Architecture Stakeholders

Principali portatori di interesse di un'architettura e le prospettive che li interessano includono:

Project Manager (Responsabili di Progetto):

- Interessi principali: pianificazione, assegnazione delle risorse, ecc.
- Per creare un piano, necessitano informazioni sui moduli da implementare, sull'ordine in cui devono essere sviluppati e sulla loro complessità, inclusa la lista delle responsabilità e le dipendenze con altri moduli.

Membri del team di sviluppo: Interessi principali:

- Comprendere la visione generale del sistema.
- Sapere quali elementi sono stati loro assegnati per l'implementazione.

- Dettagli sugli elementi assegnati, come il modello di dati con cui devono operare.
- Le interfacce con gli altri elementi.
- Gli asset di codice disponibili per l'utilizzo.
- I vincoli da rispettare, come attributi di qualità, interfacce con sistemi legacy e budget disponibili.

Tester e integratori:

- **Tester black-box (a scatola nera):** necessitano di accedere alla documentazione delle interfacce dell'elemento.
- **Integratori e tester di sistema:** hanno bisogno di consultare collezioni di interfacce, specifiche di comportamento e una vista di utilizzo per lavorare con sottogruppi incrementali.

Manutentori, quando effettuano modifiche, necessitano di:

- Una vista di decomposizione che permetta di individuare i punti in cui devono essere apportati cambiamenti.
- Una vista di utilizzo per supportare un'analisi dell'impatto e definire l'intero ambito degli effetti delle modifiche.
- Il razionale del design (design rationale).

Utenti finali:

- Possono ottenere informazioni utili sul sistema, su cosa fa e su come utilizzarlo in modo efficace esaminando l'architettura.

Analisti:

- **Interessi principali:** verificare se il design soddisfa gli obiettivi di qualità del sistema.
- Necessitano delle informazioni architettoniche necessarie per valutare gli attributi di qualità.

Personale di supporto all'infrastruttura:

- Si occupano di configurare l'infrastruttura che supporta gli ambienti di sviluppo, integrazione, staging e produzione.

- Una guida sulla variabilità è particolarmente utile per configurare l'ambiente di gestione delle configurazioni software.

Futuri architetti:

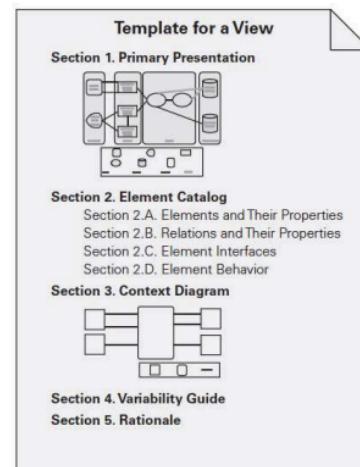
- Sono i lettori più interessati della documentazione architettonica, con un forte interesse in ogni dettaglio.
- ...

Building a Documentation Package

Il nostro compito è documentare le viste pertinenti e le informazioni che si applicano a più di una vista. Pertanto, il pacchetto di documentazione consiste in:

- Viste (Views).
- Documentazione oltre le viste (Documentation beyond views).

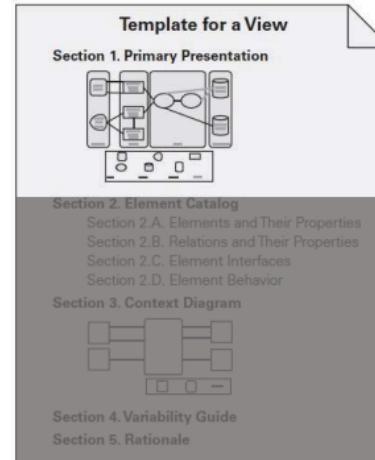
No matter what the view, the documentation for a view can follow a standard organization consisting of **five sections**...



- **Section 1: The Primary Presentation**

- Should contain the **information** you wish to transfer about the system
- Includes the primary elements and **relations** but might not include all of them
 - For example, it shows the elements and relations during normal operation but relegate error handling or exception processing
- It is most often **graphical**
 - It might be a diagram you have drawn in an informal, semiformal, or formal notation
 - Make sure to include a key that explains the notation
 - Lack of a key is the most common mistake that we see in documentation in practice
- Occasionally it will be **textual**, such as a table or a list
 - If that text is presented according to **certain stylistic rules**, these rules should be stated, as the analog to the graphical notation key

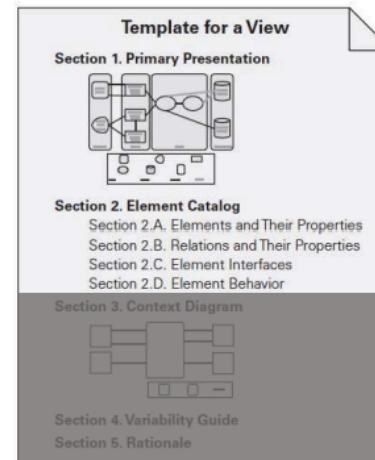
Software Architecture



- **Section 2: The Element Catalog**

- It details the **elements** depicted in the primary presentation
 - For instance, if a diagram shows elements A, B, and C, then the element catalog needs to **explain** what A, B, and C are
 - If elements or relations relevant were omitted from the primary presentation, they should be introduced and explained in the catalog
- Parts of the catalog:
 - **Elements and their properties**. Names each element in the view and lists the properties of that element
 - For example, elements in a decomposition view might have "responsibility" and elements in a communicating-processes view might have "timing"
 - **Relations and their properties**. It documents the relations the elements
 - **Element interfaces**. It documents element interfaces
 - **Element behavior**. It documents element behavior that is not obvious from the primary presentation

Software Architecture



- **Section 3: Context Diagram**

- It shows **how** the system (or portion of the system depicted in the view) relates to its **environment**
 - Entities in the environment may be **humans**, other **computer systems**, or **physical objects**, such as sensors or controlled devices

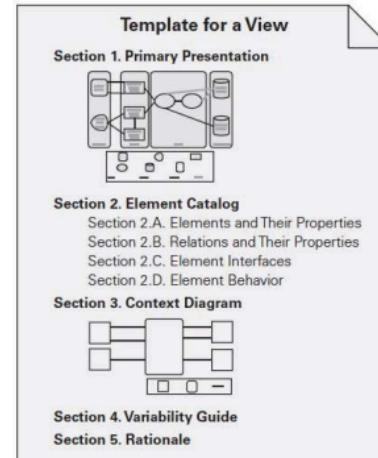
- **Section 4: Variability Guide**

- It shows where to exercise any variation
 - e.g., product line

- **Section 5: Rationale**

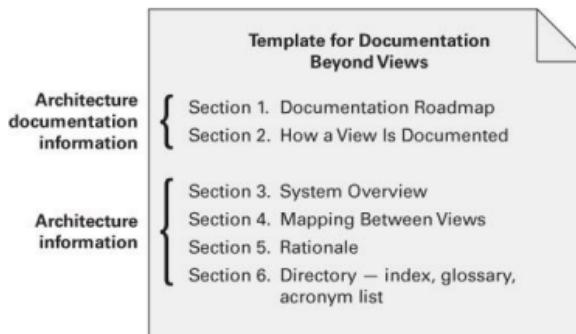
- It explains why the design is as it is and provides a **convincing argument** that it is sound
 - The choice of a **pattern** in this view should be justified here by describing the architectural problem that the chosen pattern solves and the rationale for choosing it over another

Software Architecture



La documentazione oltre le viste può essere suddivisa in due parti principali:

1. Informazioni sulla documentazione dell'architettura, che spiegano come è strutturata e organizzata la documentazione.
2. Informazioni sull'architettura, che descrivono l'architettura del sistema e come le viste sono interconnesse.



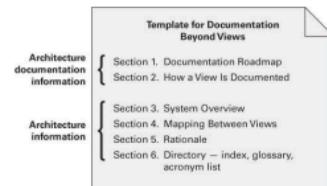
- **Section 1: Documentation Roadmap**

- Tells the reader **what information** is in the documentation and **where** to find it
- **Scope and summary**
 - Explain the purpose of the document and briefly summarize what is covered
- **How the documentation is organized**
 - For each section in the documentation, give a short synopsis of the information that can be found there
- **View overview**
 - Describes the views included. For each view:
 - The name of the view and what pattern it instantiates, if any
 - A description of the view's element types, relation types, and property types
 - A description of language, modeling techniques, or analytical methods used in constructing the view
- **How stakeholders can use the documentation**
 - Shows how various stakeholders might use the documentation to help address their concerns
 - Includes short scenarios, such as "A maintainer wishes to know the units of software that are likely to be changed by a proposed modification, then s/he has to consult section ..."

- **Section 2: How a View is Documented**

- Explain the standard organization used to document views

Software Architecture



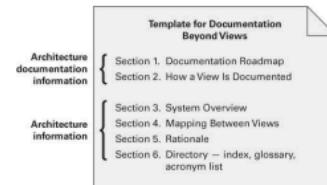
- **Section 3: System Overview**

- Short prose description of the system's **function**, its **users**, and any important **constraints**

- **Section 4: Mapping Between Views**

- Helps in
 - understanding the associations between views
 - gaining an insight into **how the architecture works** as a **unified conceptual whole**
- View-to-view associations can be captured as **tables**
 - Tables should name the **correspondence** between the elements across the two views
 - Example
 - "included in" for mapping from a decomposition view to a layered view

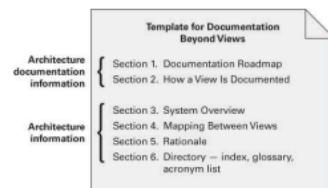
Software Architecture



- **Section 5: Rationale**

- Documents the architectural decisions that apply to more than one view
 - Documentation of **constraints** or **major requirements** that led to decisions at the system level
 - Decisions about which **fundamental architecture patterns** are used
- Indeed, when designing, you make decisions to achieve your goals
- These decisions include:
 - Selecting a design concept from several alternatives
 - Creating structures by instantiating the selected design concept
 - Establishing relationships between elements and defining interfaces
 - Allocating resources (e.g., people, hardware, computation)
- When you study an architecture diagram, you only see the end product of a thought process

Software Architecture



- Recording design decisions **beyond** the chosen elements, relationships, and properties is fundamental to help understand how you arrived at the result
- For example, you might record:
 - What evidence was produced to justify decisions?
 - Who did what?
 - Why were shortcuts taken?
 - Why were tradeoffs made?
 - What assumptions did you make?

Design Decisions and Location	Rationale and Assumptions (Include Discarded Alternatives)
Introduce concurrency (tactic) in the TimeServerConnector and FaultDetectionService	Concurrency should be introduced to be able to receive and process several events (traps) simultaneously.
Use of the messaging pattern through the introduction of a message queue in the communications layer	Although the use of a message queue imposes a performance penalty, a message queue was chosen because some implementations have high performance and, furthermore, this will be helpful to support quality attribute scenario QA-3.
...	...

- **Section 6: Directory**

- Set of reference material that helps readers find more information quickly
 - Index of terms
 - Glossary
 - Acronym list

Practical considerations

La documentazione di un sistema può essere strutturata come pagine web collegate. Utilizzando strumenti come i wiki, è possibile creare un documento condiviso a cui molti portatori di interesse possono contribuire.

Il piano di sviluppo del progetto dovrebbe specificare il processo per mantenere aggiornata la documentazione importante, inclusa quella architettonale. Gli artefatti documentali dovrebbero essere soggetti a controllo di versione, come qualsiasi altro artefatto importante del progetto.

L'architetto dovrebbe pianificare il rilascio della documentazione in corrispondenza delle principali milestone del progetto.

Summary

- Architectural documentation supports communication among various stakeholders, up the management chain, down to the developers, and across to peers
- You must understand the uses to which the documentation is to be put and its audience
- An architecture is a complicated artifact, best expressed by focusing on views
- You must choose the views to document, the notations, and a set of views that is both minimal and adequate
- There are other practical considerations, such as choosing a release strategy, choosing a dissemination tool, and creating documentation for architectures that change dynamically

Managing Architecture Debt

Il debito architetturale è una forma importante e altamente costosa di debito tecnico. Questo si verifica quando prendiamo **decisioni architetturali (ADs)** errate o subottimali.

Sintomi del debito architetturale:

Un **architectural smell** è una decisione architetturale comunemente adottata che impatta negativamente sulla qualità del sistema. È qualcosa che rende difficile modificare il codice.

Il debito architetturale si accumula quando non eseguiamo **rifattorizzazioni architetturali (ARs)**.

Esempi:

1. Needless dependencies

- As the software evolves, the earlier design decisions on using external or third-party libraries are often not revisited
- Over time, the unnecessary or old dependencies keep accumulating and they continue to be part of the deployable
- It's risky to remove dependencies (though not referred from the code anymore)

2. Layering violations

- A large fraction of software follow layered structure
- If left unchecked, the layers get tightly coupled

3. Cyclic dependencies

- When packages or components depend on each other (directly or indirectly), they get tightly coupled
- As the software evolves, unless refactored, the cyclically dependent packages can only be used, reused, tested, and deployed together

Introduciamo un processo per identificare e gestire i debiti architetturali esistenti nei sistemi. Il processo di identificazione del debito richiede tre tipi di informazioni:

- **Codice sorgente**: utilizzato per determinare le dipendenze strutturali.
- **Storico delle revisioni**, estratto dal sistema di controllo delle versioni di un progetto, utilizzato per determinare la co-evoluzione delle unità di codice.

- **Informazioni sui problemi**, estratte da un sistema di controllo dei problemi, utilizzate per determinare le ragioni dei cambiamenti.

Il processo funziona attraverso i seguenti passaggi:

1. **Identificazione degli elementi architettonicamente connessi**, con potenziali relazioni progettuali problematiche.
2. **Identificazione delle aree di debito architetturale (hotspot)**, tipicamente segnalate da un numero insolitamente alto di modifiche e bug.
3. **Fornitura di un'analisi basata sul ROI**, che consente all'architetto di giustificare il refactoring come una necessità al manager.

1) Identifying architecturally connected elements

Come posso determinare se un gruppo di file è architetturalmente connesso?

- Identificare le dipendenze statiche tra i file nel progetto. Come? Utilizzando uno strumento di analisi statica del codice.
- Catturare le dipendenze evolutive tra i file nel progetto.
 - Una dipendenza evolutiva si verifica quando due file vengono modificati insieme.
 - Come? Estrarre queste informazioni dal sistema di controllo delle revisioni.

Rappresentiamo le dipendenze tra i file utilizzando una Design Structure Matrix (DSM).

I file sono disposti sulle righe della matrice e, nello stesso ordine, sulle colonne.

Le celle della matrice possono essere annotate per indicare il tipo di dipendenza. Esempi:

Informazioni che mostrano che il file sulla riga:

- Eredita dal file nella colonna o chiama il file nella colonna (dipendenza strutturale).
- Cambia insieme al file nella colonna (dipendenza evolutiva o di storia).

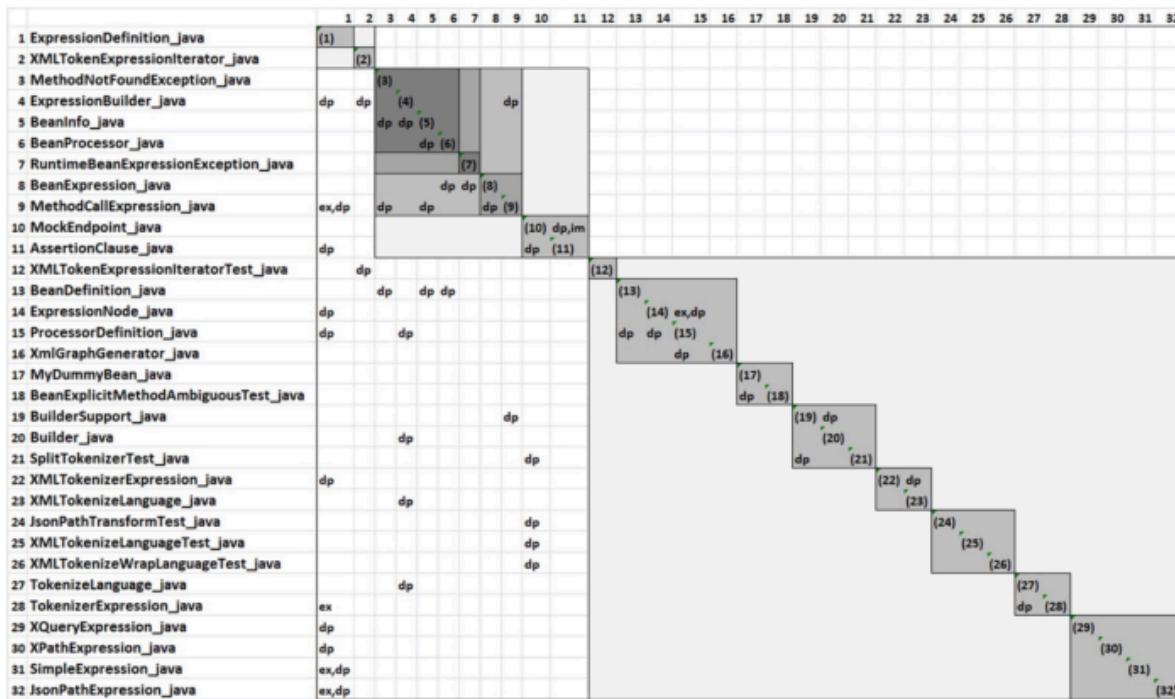
Se il sistema ha un basso accoppiamento, ci si aspetta che la DSM sia:

- **Sparsa**, cioè, ogni file dipende da un numero limitato di altri file.

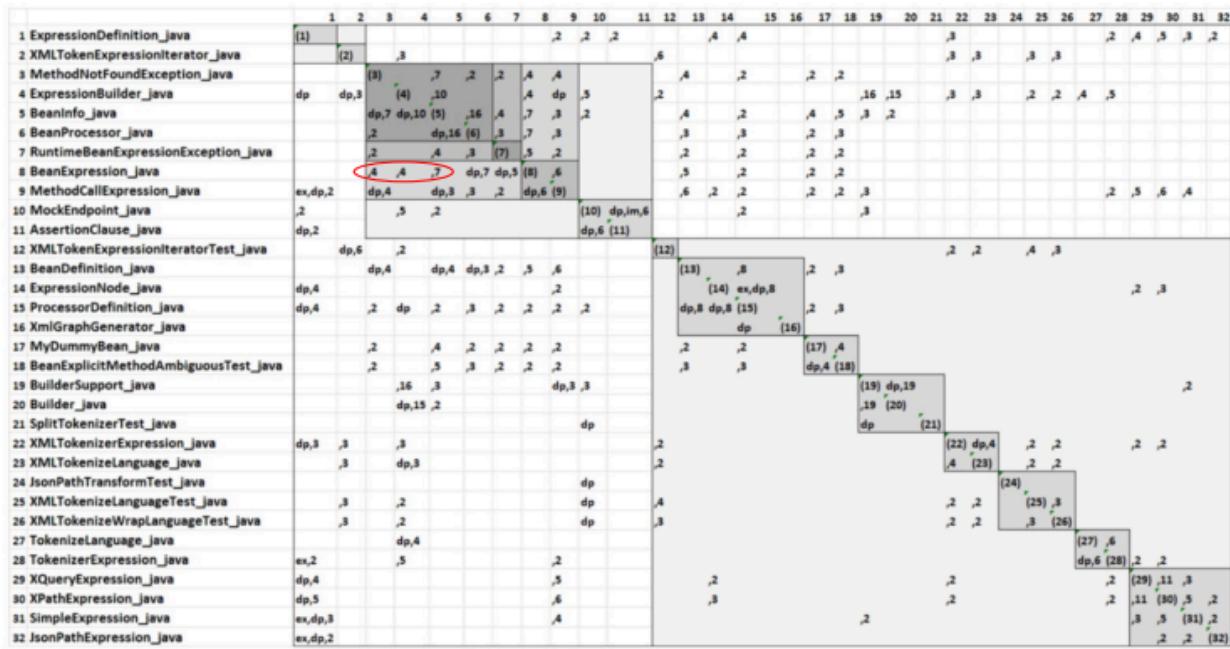
- **Diagonale inferiore**, cioè:

- Un file dipende solo da file di livello inferiore, non da quelli di livello superiore.
- Non ci sono dipendenze cicliche nel sistema.

1) Identifying architecturally connected elements – Example: Apache Camel, Structural



1) Identifying architecturally connected elements – Example: Apache Camel, Evolutionary



2) identifying areas of arch debt

Per affrontare il debito architettonico, è necessario identificare i file specifici e le loro relazioni problematiche.

Chiamiamo **hotspot** gli insiemi di elementi che contribuiscono in modo sproporzionato ai costi di manutenzione di un sistema.

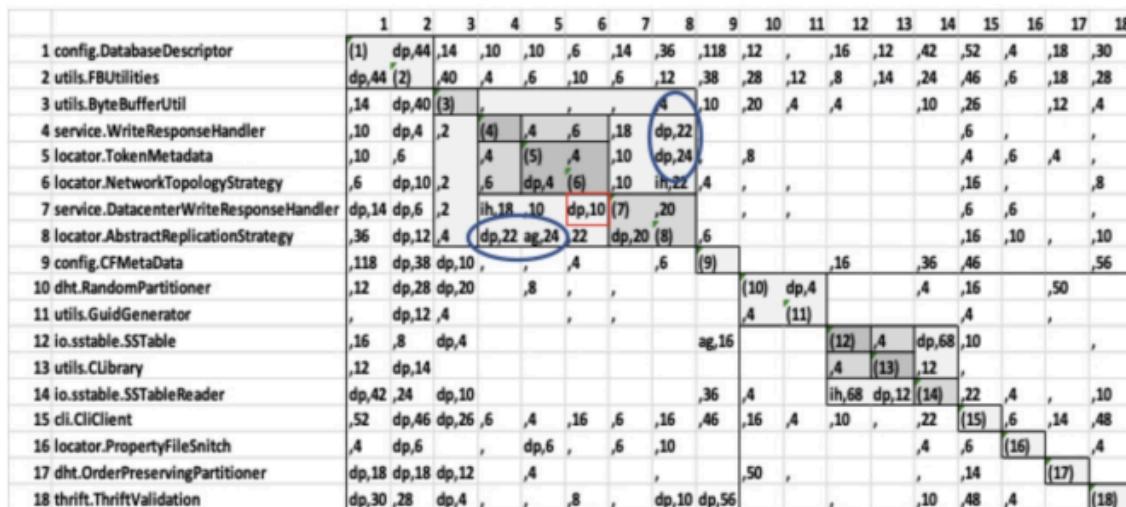
Per identificare gli hotspot:

- Cerchiamo anti-pattern che favoriscono un elevato accoppiamento e una bassa coesione.
- Li ponderiamo.

2) identifying areas of arch debt: anti-pattern

Name	How to identify anti-pattern? Search for ...
Unstable interface	... a file with many dependents that is modified frequently with other files
Modularity violation	... two or more structurally independent files that change together frequently
Unhealthy inheritance	... <ul style="list-style-type: none"> a parent depends on its child class a client of the class hierarchy depends on both the parent and one or more of its children
Cyclic dependency or clique	... sets of files that form a strongly connected graph, where there is a structural dependency path between any two elements of the graph
Package cycle	... packages that form a strongly connected graph
Crossing	... a file that has both high fan-in and fan-out with other files and that has substantial co-change relations with these other files

2) identifying areas of arch debt: anti-pattern - Example: Apache Cassandra, example of Clique



2) identifying areas of arch debt: anti-pattern - Example: Apache Cassandra, example of Unhealthy Inheritance

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1 config.DatabaseDescriptor	(1)	dp,44	,14	,10	,10	,6	,14	,36	,118	,12	,	,16	,12	,42	,52	,4	,18	,30
2 utils.FBUtilities	dp,44	(2)	,40	,4	,6	,10	,6	,12	,38	,28	,12	,8	,14	,24	,46	,6	,18	,28
3 utils.ByteBufferUtil	,14	dp,40	(3)	,	,	,	,	,4	,10	,20	,4	,4	,	,10	,26	,	,12	,4
4 service.WriteResponseHandler	,10	dp,4	,2	(4)	,4	,6	,18	dp,22							,6	,	,	
5 locator.TokenMetadata	,10	,6		,4	(5)	,4	,10	dp,24							,4	,6	,4	,
6 locator.NetworkTopologyStrategy	,6	dp,10	,2	,6	dp,4	(6)	,10	ih,22	,4						,16	,	,8	
7 service.DatacenterWriteResponseHandler	dp,14	dp,6	,2	ih,18	,10	dp,10	(7)	,20							,6	,6	,	
8 locator.AbstractReplicationStrategy	,36	dp,12	,4	(dp,22 ag,24)	22	dp,20	(8)	,6							,16	,10	,	,10
9 config.CFMetaData	,118	dp,38	dp,10	,	,	,4	,	,6	(9)						,16	,36	,46	,56
10 dht.RandomPartitioner	,12	dp,28	dp,20	,8	,	,				(10)	dp,4				,4	,16	,50	
11 utils.GuidGenerator	,	dp,12	,4		,	,				(11)					,4	,		
12 io.sstable.SSTable	,16	,8	dp,4					ag,16			(12)	,4	dp,68	,10				
13 utilz.CLibrary	,12	dp,14									,4	(13)	,12					
14 io.sstable.SSTableReader	dp,42	,24	dp,10					,36	,4		ih,68	dp,12	(14)	,22	,4	,	,10	
15 cli.CliClient	,52	dp,46	dp,26	,6	,4	,16	,6	,16	,46	,16	,4	,10	,	,22	(15)	,6	,14	,48
16 locator.PropertyFileSnitch	,4	dp,6	,	dp,6	,	,6	,10								,4	,6	(16)	,4
17 dht.OrderPreservingPartitioner	dp,18	dp,18	dp,12	,4				,50							,,	,14		(17)
18 thrift.ThriftValidation	dp,30	,28	dp,4	,	,8	,	dp,10	dp,56							,10	,48	,4	(18)

La maggior parte delle segnalazioni in un sistema di tracciamento riguardano correzioni di bug o miglioramenti di funzionalità.

Le correzioni di bug, così come la volatilità legata ai bug e ai cambiamenti, sono fortemente correlate agli anti-pattern e agli hotspot.

Possiamo assegnare una penalità a ciascun anti-pattern utilizzando queste informazioni.

Come pesare gli anti-pattern:

Per ogni file in ciascun anti-pattern (ovvero, hotspot), determiniamo:

- Il numero totale di correzioni di bug e modifiche.
- Il volume totale di volatilità (churn) che il file ha subito.

Successivamente, sommiamo le correzioni di bug, le modifiche e la volatilità dei file all'interno di ciascun anti-pattern.

- Questo ci fornisce un peso per ogni anti-pattern in termini del suo contributo al debito architettonurale.

3) providing ROI-based analysis...

Con questa analisi, una strategia di riduzione del debito (attraverso il refactoring) diventa semplice. Conoscere i file coinvolti nel debito, insieme alle loro relazioni problematiche, consente all'architetto di elaborare e giustificare un piano di refactoring.

Esempi:

- Se esiste una **clique**, è necessario rimuovere o invertire una dipendenza per interrompere il ciclo di dipendenze.
- Se è presente un'**ereditarietà malsana**, alcune funzionalità devono essere spostate, tipicamente da una classe figlia a una classe padre.
- ...

Tool support

- This analysis process requires the following tools:
 - A tool to extract a set of issues from an **issue tracker**
 - A tool to extract a log from a **revision control system**
 - A tool to **reverse-engineer the code base**, to determine the syntactic dependencies among files
 - A tool to build **DSMs** from the extracted information and walk through the DSM looking for the anti-patterns
 - A tool that **calculates the debt** associated with each hotspot
- The only specialized tools needed for this process are the ones to build the DSM and analyze the DSM
 - E.g., DV8 (www.archdia.com)
- Projects likely already have issue tracking systems and revision histories, and plenty of reverse-engineering tools are available, including open-source options

Summary

- Architecture debt is an important and costly form of technical debt. Compared to code-based technical debt, architecture debt is harder to identify because its root causes are distributed among several files and their relationships
- The process involves gathering information from the project's issue tracker, its revision control system, and its source code
- Architecture anti-patterns can be identified and grouped into hotspots, and the impact of these hotspots can be quantified
- This architecture debt monitoring process can be automated and built into a system's continuous integration tool suite
- Once architecture debt has been identified, if it is bad enough, it should be removed through refactoring
- The output of this process provides the quantitative data necessary to make the business case for refactoring to project management