Ingegneria del Software

Progettazione architetturale

Antonino Staiano
e-mail: **antonino.staiano@uniparthenope.it**

Progettazione

- La progettazione di un sistema (System Design) è l'insieme dei task svolti dall'ingegnere del software/analista che permettono di trasformare il modello di analisi nel modello di design del sistema
- Obiettivo ultimo è definire un nuovo documento, scritto in linguaggio tecnico/formale, da dare ai programmatori affinché questi siano in grado di implementare il software descritto nel Documento dei requisiti software

Scopo della progettazione del sistema

- Definire gli obiettivi di design del progetto
- Decomporre il sistema in sottosistemi più piccoli che possono essere realizzati da team individuali
- Selezionare le strategie per costruire il sistema, quali:
  - Strategie hardware/software
  - Strategie relative alla gestione dei dati persistenti
  - Il flusso di controllo globale
  - Le politiche di controllo degli accessi

Output della progettazione del sistema

- Un documento contenente:
  - Un modello del sistema che include la decomposizione del sistema in sottosistemi
    - Formalizzato con Class, Sequence, Statechart diagram …etc di UML
  - Una chiara descrizione di ognuna delle strategie elencate nella slide precedente

Progettazione del sistema

- Si cambia radicalmente punto di vista
  - Finora si è lavorato per interagire con il cliente
    - Linguaggio per profani
  - Dal system design gli interlocutori sono le squadre di programmatori che dovranno implementare il sistema
    - Linguaggio per esperti di informatica
- Analisi: si focalizza sul dominio di applicazione
- Design: si focalizza sul dominio di implementazione
  - Si devono raggiungere dei compromessi fra i vari obiettivi di design che spesso sono in conflitto gli uni con gli altri
  - Si devono anticipare molte decisioni relative alla progettazione pur non avendo una chiara idea del dominio della soluzione

Attività del system design

- Il system design è costituito da tre macro-attività:

  1. Identificazione degli obiettivi di design: gli analisti identificano e definiscono le priorità dei criteri di qualità del sistema

  2. Progettazione della decomposizione del sistema in sottosistemi
     - Utilizziamo linee guida e stili architetturali standard.

  3. Raffinamento della decomposizione in sottosistemi per rispettare gli obiettivi di design: La decomposizione iniziale di solito non soddisfa gli obiettivi di design. Gli sviluppatori la raffinano finché gli obiettivi non sono soddisfatti

Input che provengono dalla fase di specifica dei Requisiti Software

- Il modello di analisi descrive il sistema dal punto di vista degli attori
- Non contiene informazioni sulla struttura interna del sistema, la sua configurazione hardware e, in generale, su come il sistema dovrebbe essere realizzato
- L'analisi fornisce in output il modello dei requisiti descritto dai seguenti prodotti:

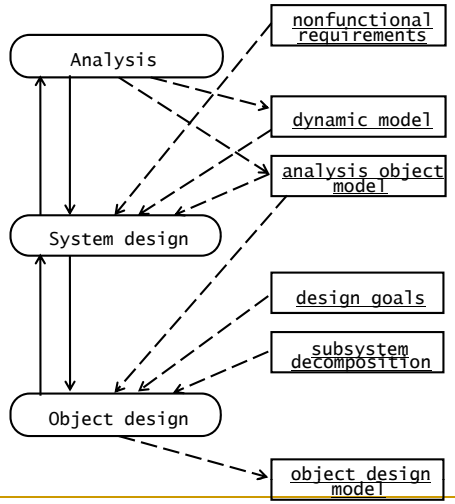
  1. Use Case model => descrive le funzionalità del sistema dal punto di vista degli attori

  2. Object model => descrive le entità manipolate dal sistema

  3. Un sequence diagram per ogni use case significativo => mostra la sequenza di interazioni fra gli oggetti che partecipano al caso d'uso

  4. Requisiti non funzionali e vincoli => quali tempo di risposta massimo, minimo throughput, affidabilità, piattaforma per il sistema operativo, etc.

Input e Output



Identificare gli obiettivi qualitativi del sistema

- E' il primo passo del system design
- Identifica i criteri di qualità su cui deve essere basata la progettazione del sistema
- Molti design goal possono essere ricavati dai requisiti non funzionali o dal dominio di applicazione, altri sono forniti dal cliente
- E' importante formalizzarli esplicitamente poiché ogni importante decisione di design deve essere fatta seguendo lo stesso insieme di criteri

Criteri di design

- Si possono selezionare gli obiettivi di design da una lunga lista di qualità desiderabili
- I criteri sono organizzati in cinque gruppi:
  - Prestazioni
  - Affidabilità
  - Costi
  - Mantenimento
  - Criteri End User

Criteri di prestazione

- Includono i requisiti imposti sul sistema in termini di spazio e velocità
- Tempo di risposta: con quali tempi una richiesta di un utente deve essere soddisfatta dopo che la richiesta è stata inoltrata?
- Throughput: quanti task il sistema deve portare a compimento in un periodo di tempo prefissato?
- Memoria: quanto spazio è richiesto al sistema per funzionare?

Criteri di affidabilità

- Quanto sforzo deve essere speso per minimizzare i crash del sistema e le loro conseguenze?
- Rispondono alle seguenti domande:
  - Robustezza. Capacità di sopravvivere ad input non validi immessi dall'utente
  - Attendibilità. Differenza di comportamento specificato e osservato
  - Disponibilità. Percentuale di tempo in cui il sistema può essere utilizzato per compiere normali attività
  - Tolleranza ai fault. Capacità di operare sotto condizioni di errore
  - Sicurezza. Capacità di resistere ad attacchi di malintenzionati
  - Fidatezza. Capacità di evitare di danneggiare vite umane

Criteri di costi

- Includono i costi per sviluppare il sistema per metterlo in funzione e per amministrarlo
- Quando il sistema sostituisce un sistema vecchio, è necessario considerare il costo per assicurare la compatibilità con il vecchio o per transitare verso il nuovo sistema
- I criteri di costo:
  - Costo di sviluppo del sistema iniziale
  - Costo relativo all'installazione del sistema e training degli utenti
  - Costo per convertire i dati del sistema precedente. Questo criterio viene applicato quando nei requisiti è richiesta la compatibilità con il sistema precedente (backward compatibility)
  - Costo di manutenzione. Costo richiesto per correggere errori hw/sw

Criteri di mantenimento

- Determinano quanto sia più o meno complesso modificare il sistema dopo il suo rilascio
- Estendibilità. Quanto è facile aggiungere funzionalità o nuove classi al sistema?
- Modificabilità. Quanto facilmente possono essere cambiate le funzionalità del sistema?
- Adattabilità. Quanto facilmente può essere portato il sistema su diversi domini di applicazione?
- Portabilità. Quanto è facile portare il sistema su differenti piattaforme?
- Leggibilità. Quanto facile deve essere capire il sistema dalla lettura del codice?
- Tracciabilità dei requisiti. Quanto è facile mappare il codice nei requisiti specifici?

Criteri End User

- Includono qualità che sono desiderabili dal punto di vista dell'utente, ma che non sono state coperte dai criteri di performance e affidabilità
- Criteri:
  - Utilità: il sistema quanto bene dovrà supportare il lavoro dell'utente?
  - Usabilità: quanto dovrà essere facile l'utilizzo del sistema per l'utente?

Trade-off di progettazione

- Quando definiamo gli obiettivi di progetto, spesso solo un piccolo sottoinsieme di questi criteri può essere tenuto in considerazione
  - Es.: non è realistico sviluppare software che sia simultaneamente sicuro e poco costoso
- Gli sviluppatori devono dare delle priorità agli obiettivi di design, tenendo anche conto di aspetti manageriali, quali il rispetto dello schedule e del budget

Trade-off di progettazione

- Esempi:
- Spazio vs. velocità. Se il software non rispetta i requisiti di tempo di risposta e di throughput, è necessario utilizzare più memoria per velocizzare il sistema (es. Caching, più ridondanza). Se il software non rispetta i requisiti di memoria, può essere compresso a discapito della velocità
- Tempo di rilascio vs. funzionalità. Se i tempi di rilascio sono stringenti, possono essere rilasciate meno funzionalità di quelle richieste, ma nei tempi giusti
- Tempo di rilascio vs. qualità. Se i tempi di rilascio sono stretti, il projetc manager può rilasciare il software nei tempi prefissati con dei bug e, in tempi successivi, correggerli, o rilasciare il software in ritardo, ma con meno bug
- Tempo di rilascio vs. staffing. Può essere necessario aggiungere delle risorse al progetto per accrescere la produttività

Concetti di base di progettazione

- Descriveremo le decomposizioni del sistema e le loro proprietà in maggiore dettaglio
  - Definiamo prima il concetto di sottosistema e la sua relazione con le classi
  - Successivamente, vedremo le interfacce dei sottosistemi
    - I sottosistemi forniscono servizi ad altri sottosistemi
    - Un servizio è un insieme di operazioni tra loro collegate finalizzate ad un obiettivo comune
  - Esamineremo poi, due proprietà dei sottosistemi: la coesione e l'accoppiamento
    - Coesione: misura la dipendenza tra le classi in un sottosistema
    - Accoppiamento: misura la dipendenza tra due sottosistemi
    - Una decomposizione ideale dovrebbe minimizzare l'accoppiamento e massimizzare la coesione

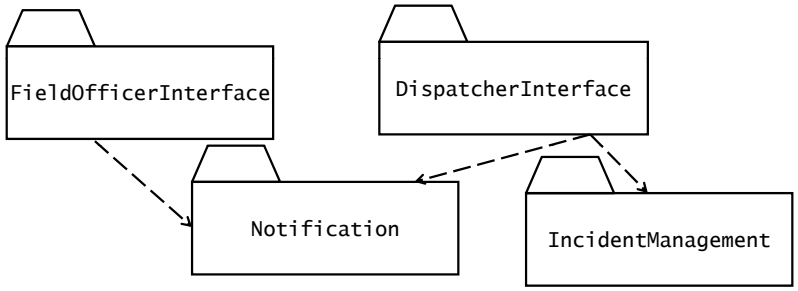
Sottosistemi e classi

- Per ridurre la complessità del dominio della soluzione, si decompone il sistema in parti più semplici, chiamate sottosistemi, che sono costituite da un certo numero di classi del dominio della soluzione
- Un sottosistema tipicamente corrisponde alla quantità di lavoro che un singolo sviluppatore o un singolo team di sviluppo può affrontare
  - Decomponendo il sistema in sottosistemi relativamente indipendenti, vari team di sviluppo possono lavorare concorrentemente sui singoli sottosistemi con un minimo di overhead per la comunicazione
  - Nel caso di sistemi complessi, si applica ricorsivamente questo principio e si decompone un sottosistema in sottosistemi più semplici

Esempio

- Il sistema di gestione degli incidenti può essere decomposto in
  - sottosistema *DispatcherInterface*, che realizza l'interfaccia utente per il *Dispatcher*
  - sottosistema *FieldOfficerInterface*, che realizza l'interfaccia utente per il *FieldOfficer*
  - sottosistema *IncidentManagement*, responsabile di tener traccia delle risorse disponibili (Mezzi dei vigili del fuoco, ambulanze, etc.)
  - sottosistema *Notification*, che implementa la comunicazione tra i terminali dei *FieldOfficer* e le stazioni dei *Dispatcher*

Decomposizione in sottosistemi per un sistema di gestione incidenti



Decomposizione in sottosistemi

- Numerosi linguaggi di programmazione forniscono costrutti per modellare i sottosistemi (tipo i package in Java)
- In altri linguaggi, come C o C++, i sottosistemi non sono modellati esplicitamente, per cui gli sviluppatori usano delle convenzioni per raggruppare le classi
  - un sottosistema può essere rappresentato come una directory che contiene tutti i file che implementano il sottosistema
- A prescindere dal linguaggio usato, gli sviluppatori devono documentare con attenzione la decomposizione in sottosistemi poiché i sottosistemi solitamente sono realizzati da team di sviluppo differenti

Servizi e Interfacce dei sottosistemi

- Un sottosistema è caratterizzato dai servizi che esso fornisce ad altri sottosistemi
- Un servizio è un insieme di operazioni collegate che condividono uno scopo comune
  - Un sottosistema che fornisce un servizio di notifica, ad esempio, definisce operazioni per inviare le notifiche, cercare canali di notifica, e sottoscrivere e annullare la sottoscrizione ad un canale
- L'insieme di operazioni di un sottosistema che sono disponibili agli altri sottosistemi costituiscono l'interfaccia del sottosistema

Servizi e Interfacce dei sottosistemi

- L'interfaccia del sottosistema include il nome delle operazioni, i relativi parametri, il loro tipo, e i valori di ritorno
- La progettazione del sistema si focalizza sulla definizione dei servizi forniti da ogni sottosistema, ovvero, enumerare le operazioni, i loro parametri ed il comportamento ad alto livello
- La progettazione degli oggetti si focalizza sulle application program interface (API), che rifiniscono ed estendono le interfacce dei sottosistemi

Servizi e Interfacce dei sottosistemi

- La definizione di un sottosistema in termini di servizi che esso fornisce aiuta a mettere a fuoco l'interfaccia piuttosto che l'implementazione
- Quando si scrive una interfaccia di un sottosistema, ci si dovrebbe sforzare di minimizzare la quantità di informazioni interenti l'implementazione
  - Ad esempio, un'interfaccia di sottosistema non dovrebbe fare riferimento alle strutture dati interne, come liste concatenate, array o tavole hash
    - Ciò consente di minimizzare l'impatto delle modifiche quando rivediamo l'implementazione di un sottosistema

Accoppiamento e coesione

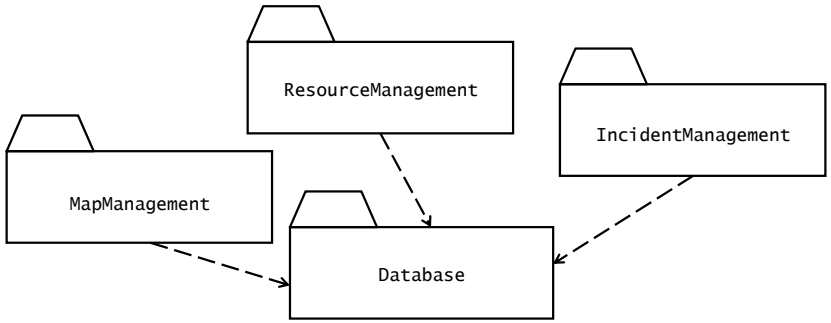
- L'accoppiamento è il numero di dipendenze tra due sottosistemi
- Se due sottosistemi sono debolmente accoppiati, essi sono relativamente indipendenti
  - La modifica ad un sottosistema avrà un leggero impatto sull'altro sottosistema
- Se due sottosistemi sono fortemente accoppiati, la modifica ad uno di essi avrà verosimilmente un certo impatto anche sull'altro sottosistema
- Una proprietà desiderabile della decomposizione è che i sottosistemi siano (ragionevolmente) debolmente accoppiati in modo da minimizzare l'impatto che errori o cambiamenti futuri su un sottosistema possono avere sugli altri sottosistemi

Esempio: sistema gestione incidenti

- Durante la progettazione decidiamo di memorizzare tutti i dati persistenti (cioè, tutti i dati che vivono oltre una singola esecuzione del sistema) in un database relazionale
  - Introduciamo un sottosistema aggiuntivo chiamato *Database*
  - Inizialmente, progettiamo l'interfaccia del sottosistema database in modo che i sottosistemi che necessitano di memorizzare i dati inviano comandi nel linguaggio di query nativo del database, come SQL
  - Ciò porta ad una situazione di forte accoppiamento tra il sottosistema *Database* e i tre sottosistemi client (*IncidentManagement*, *ResourceManagement* e *MapManagement*) poiché ogni modifica nel modo in cui i dati sono memorizzati richiederà delle modifiche nei sottosistemi client
  - Per ridurre l'accoppiamento tra questi quattro sottosistemi, decidiamo di creare un nuovo sottosistema, chiamato *Storage*, che scherma il database dagli altri sottosistemi. I tre sottosistemi client usano i servizi forniti da *Storage* che è responsabile di inviare le query in SQL al sottosistema *Database*
  - Se decidiamo di cambiare tipo di database dobbiamo solo cambiare il sottosistema *Storage*

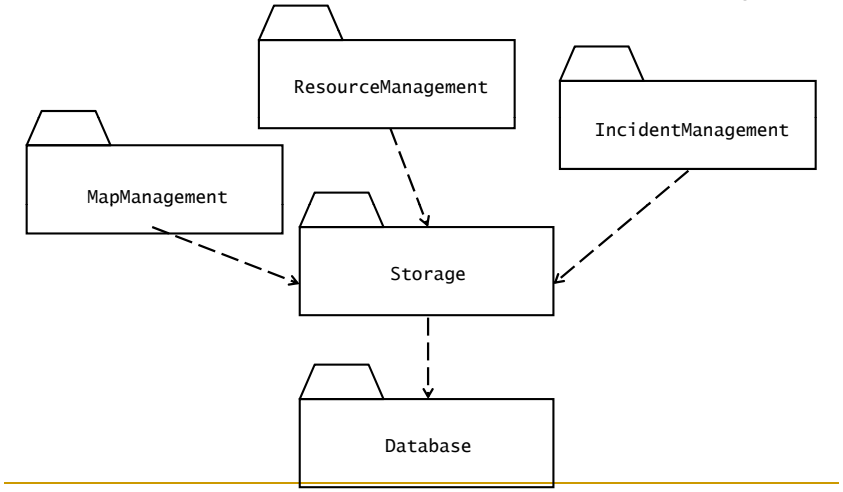
Riduzione dell'accoppiamento

Alternativa 1: Accesso diretto al sottosistema Database



Riduzione dell'accoppiamento

Alternativa 2: Accesso indiretto al Database tramite un sottosistema Storage



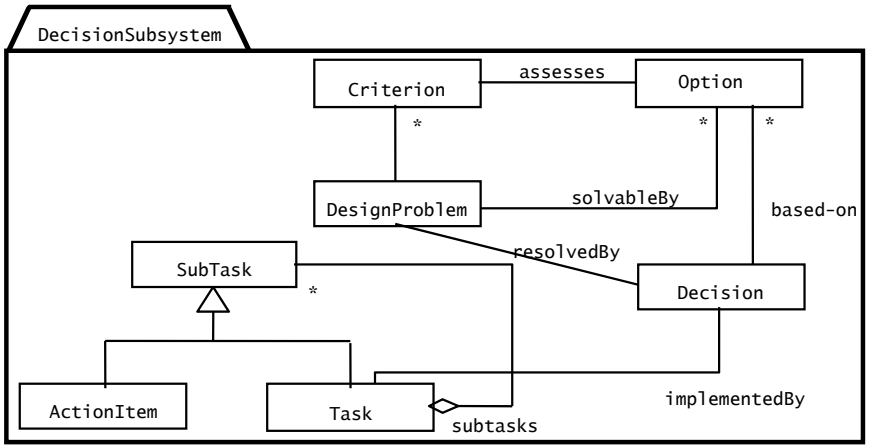
Coesione

- La coesione è il numero di dipendenze all'interno di un sottosistema
- Se un sottosistema contiene molti oggetti che sono tra loro in relazione ed ognuno esegue compiti simili, la sua coesione è elevata
- Se un sottosistema contiene un numero di oggetti non legati tra loro, la sua coesione è bassa
- Una proprietà desiderabile della decomposizione è che porti a sottosistemi con elevata coesione

Esempio

- Consideriamo un sistema di tracking di decisioni per registrare problemi di progettazione, discussioni, valutazioni alternative, decisioni e le rispettive implementazioni in termini di compiti
- *DesignProblem* e *Option* rappresentano l'esplorazione dello spazio di progettazione: formuliamo il sistema in termini di un numero di *DesignProblem* e documentiamo ogni *Option* che essi esplorano
- La classe *Criterion* rappresenta le qualità in cui siamo interessati. Una volta che abbiamo valutato le *Option* esplorate rispetto ai *Criteria* desiderabili, implementiamo *Decisions* in termini di *Task*
- *Task* sono ricorsivamente decomposti in *Subtask* abbastanza piccoli da essere assegnati a singoli sviluppatori. Chiamiamo i task atomici *ActionItems*
- Il sistema è sufficientemente piccolo per cui possiamo raggruppare tutte le classi in un unico sottosistema chiamato *DecisionSubsystem*

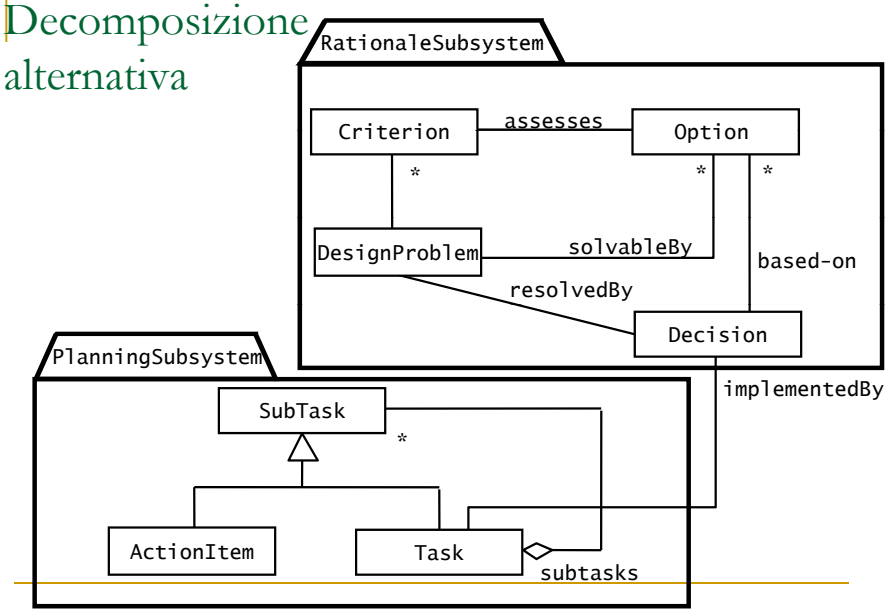
Sistema di tracking di Decisioni



Esempio: Sistema di tracking di Decisioni

- Il modello delle classi, però, può essere partizionato in due sottografi
  - *RationaleSubsystem*, contiene le classi *DesignProblem*, *Option*, *Criterion* e *Decision*
  - *PlanningSubsystem*, contiene Task, *Subtask* e *ActionItem*
- Entrambi i sottosistemi hanno un'elevata coesione rispetto al sottosistema originale *DecisionSubsystem*
  - Ciò ci consente di riusare ogni parte in modo indipendente poiché gli altri sottosistemi necessitano solo della parte di pianificazione o la parte delle logica sottostante.
  - Inoltre, i sottosistemi risultanti sono più piccoli rispetto all'originale, possono quindi essere assegnati a singoli sviluppatori
  - L'accoppiamento tra i sottosistemi è relativamente basso, con solo un'associazione tra i due sottosistemi

Decomposizione alternativa



Trade-off coesione - accoppiamento

- In generale, c'è un compromesso tra coesione e accoppiamento
- Possiamo incrementare la coesione decomponendo il sistema in sottosistemi più piccoli
- Tuttavia ciò incrementa l'accoppiamento poiché aumentano il numero di interfacce

Concetti di Progettazione

Organizzazione dei sottosistemi in architetture

Definizione di Architettura SW

- L'architettura software è l'organizzazione di base di un sistema, espressa dai suoi componenti, dalle relazioni tra di loro e con l'ambiente, e i principi che ne guidano il progetto e l'evoluzione [IEEE/ANSI 1471–2000]
- Informalmente, un'architettura software è la **struttura del sistema, costituita dalle varie parti** che lo compongono, con le relative relazioni.

Architetture

- L'architettura di un sistema software viene definita nella prima fase di progettazione (*progettazione architetturale)*
- Lo scopo primario è la scomposizione del sistema in sottosistemi:
  - la realizzazione di più componenti distinti è meno complessa della realizzazione di un sistema come monolito
  - Permette di parallelizzare lo sviluppo
  - Favorisce modificabilità, riusabilità, portabilità, etc…

Architetture

- Definire un'architettura significa mappare funzionalità su moduli
  - Es: Modulo di interfaccia utente, modulo di accesso a db, modulo di gestione della sicurezza, etc…
- Anche la definizione delle architetture deve seguire i concetti di Alta Coesione e Basso Accoppiamento
  - Cambia il livello di astrazione, ma non i concetti sottostanti
  - Ogni sottosezione dell'architettura dovrà fornire servizi altamente legati tra loro, cercando di limitare il numero di altri moduli con cui è legato

Definizione dell'architettura

- La definizione dell'architettura viene di solito fatta da due punti di vista diversi, che portano alla soluzione finale:
  - Identificazione e relazione dei sottosistemi
  - Definizione politiche di controllo
- Entrambe le scelte sono cruciali:
  - è difficile modificarle quando lo sviluppo è partito, poiché molte interfacce dei sottosistemi dovrebbero essere cambiate

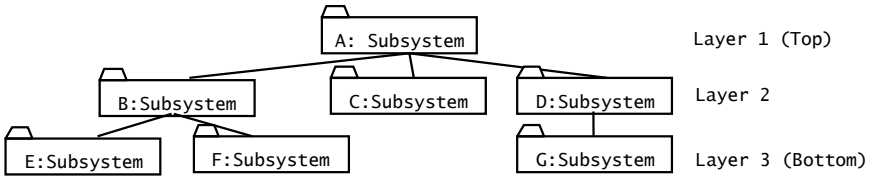
Concetti di Progettazione

Identificare i sottosistemi

Strati e Partizioni

- Una decomposizione gerarchica di un sistema porta ad un insieme ordinato di strati (o layer)
- Uno strato è il raggruppamento di sottosistemi che forniscono servizi tra loro in relazione, eventualmente usando servizi da un altro strato
- Gli strati sono ordinati nel senso che ciascuno di essi dipende solo dagli strati di livello inferiore e non ha conoscenza degli strati al di sopra
  - Lo strato che non dipende da nessun altro strato è chiamato strato bottom
  - Lo strato che non è usato da alcuno strato è chiamato strato top
  - In un'architettura chiusa, ogni strato può accedere solo allo strato immediatamente al di sotto di esso
  - In un'architettura aperta, uno strato può anche accedere a strati ai livelli più bassi

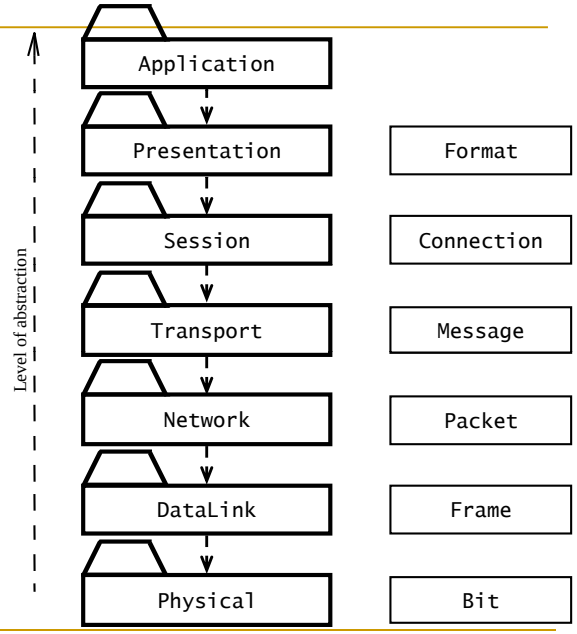
Decomposizione in sottosistemi di un sistema in tre layer



Strati

- Le architetture stratificate chiuse hanno proprietà desiderabili: portano ad un basso accoppiamento tra i sottosistemi ed i sottosistemi possono essere integrati e testati incrementalmente
- Ogni livello, comunque, introduce un overhead di velocità e memoria che può rendere difficile soddisfare i requisiti non funzionali
- Inoltre, aggiungere funzionalità al sistema in revisioni successive può essere difficile soprattutto quando le aggiunte non sono previste
- In pratica, raramente un sistema è decomposto in più di 3 o 5 strati

Un esempio di architettura chiusa: Il modello OSI



Partizioni

- Un altro approccio consiste nel partizionare il sistema in sottosistemi peer, ognuno responsabile di una diversa classe di servizi
  - Ad esempio, il sistema di bordo di una macchina potrebbe essere decomposto in un servizio di viaggio che fornisce le direzioni in tempo reale al guidatore, un servizio di preferenze individuale che ricorda la posizione del sedile del guidatore e la stazione radio preferita, ed un servizio del veicolo che tiene traccia dei consumi della macchina, le riparazioni e il piano di manutenzione
    - Ogni sottosistema dipende debolmente dagli altri, ma può operare anche da solo

Stratificazione e partizionamento

- In generale, la decomposizione di un sottosistema è il risultato sia del partizionamento che della stratificazione
  - Prima si partiziona il sistema in sottosistemi di livello superiore responsabili di specifiche funzionalità o che sono eseguiti su specifici nodi hardware
  - Ciascuno dei sottosistemi risultanti sono decomposti, a patto che la complessità lo giustifichi, in strati a livello gerarchico inferiore fino a che essi sono abbastanza semplici da essere implementati da un singolo sviluppatore
  - Ogni sottosistema aggiunge un certo overhead di elaborazione a causa dell'interfaccia con gli altri sottosistemi
  - Un eccessivo partizionamento o stratificazione può incrementare la complessità

Concetti di Progettazione

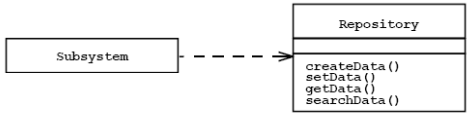
Stili architetturali

Principali Architetture Software

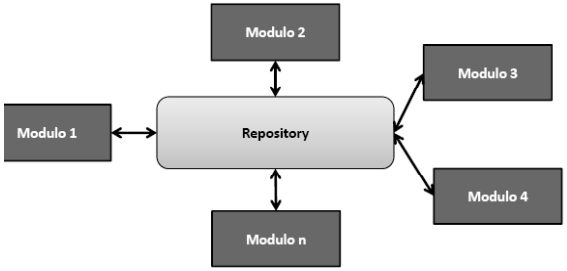
- Nell'ingegneria del sw sono stati definiti vari stili architetturali che possono essere usati come base di architetture software:
  - Architettura a Repository
  - Architettura Client/Server
  - Architettura Peer-To-Peer

Architettura repository

- I sottosistemi accedono e modificano una singola struttura dati chiamata **repository**
- I sottosistemi sono "relativamente indipendenti" (interagiscono solo attraverso il repository)
- Il flusso di controllo è dettato o dal repository (un cambiamento nei dati memorizzati) o dai sottosistemi (flusso di controllo indipendente)



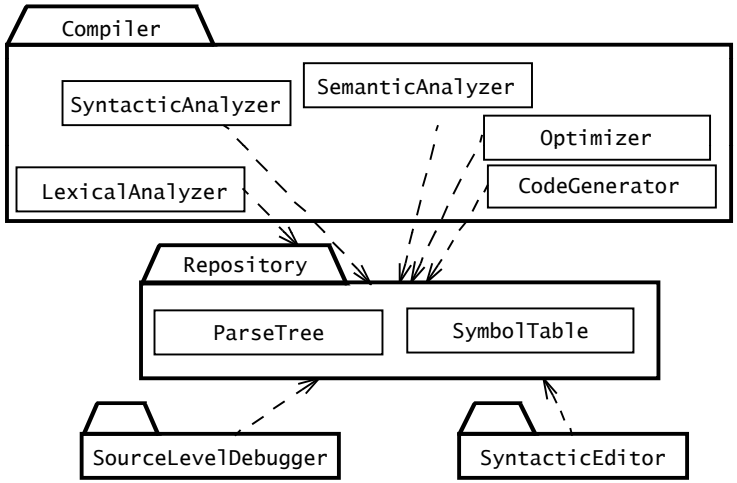
Architettura repository



Repository

- I repository sono usati tipicamente per i sistemi di gestione dei database
  - Sistemi pagamento personale
  - Sistema di gestione bancaria
- Lo posizione centrale dei dati semplifica il trattamento di questioni relative a concorrenza e integrità tra i sottosistemi
- I compilatori e gli ambienti di sviluppo del software seguono anch'essi uno stile architetturale a repository
  - I differenti sottosistemi di un compilatore accedono e aggiornano un parse tree centrale ed una symbol table
  - I debugger e gli editor di sintassi accedono anch'essi alla symbol table

Esempio di architettura repository



Caratteristiche della architettura a repository

- Vantaggi
  - Modo efficiente di condividere grandi quantità di dati: write once for all to read
  - Un sottosistema non si deve preoccupare di come i dati sono prodotti/usati da ogni altro sottosistema
  - Gestione centralizzata di backup, security, access control, recovery da errori...
  - Il modello di condivisione dati è rappresentato dallo schema del repository ==> facile aggiungere nuovi sottosistemi
- Svantaggi
  - I sottosistemi devono concordare su un modello dati di compromesso ==> minori performance
  - Data evolution: l'adozione di un nuovo modello dati è difficile e costosa: (a) esso deve essere applicato a tutto il repository, e (b) tutti i sottosistemi devono essere aggiornati
  - E' difficile distribuire efficientemente il repository su più macchine (continuando a vederlo come logicamente centralizzato): problemi di ridondanza e consistenza dati.

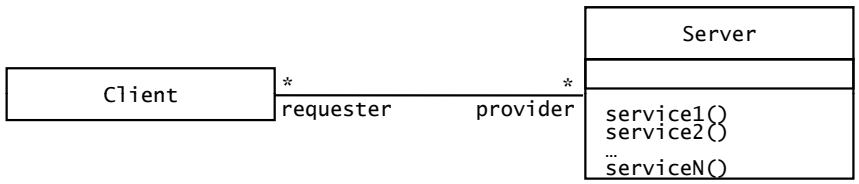
Architettura Client-server

- E' una architettura distribuita dove dati ed elaborazione sono distribuiti su una rete di nodi di due tipi:
  - I server sono macchine con processori potenti e dedicati: offrono servizi specifici come stampa, gestione di file system, compilazione, gestione traffico di rete, calcolo
  - I client sono macchine meno prestazionali sulle quali girano le applicazioni-utente, che utilizzano i servizi dei server

Architettura Client/Server

- Il sottosistema server, fornisce servizi ad istanze dei sottosistemi client che sono responsabili dell'interazione con l'utente
- I Client chiamano il server che realizza alcuni servizi e restituisce il risultato
  - I Client conoscono l'interfaccia del Server (i suoi servizi)
  - I Server non conoscono le interfacce dei Client
  - La risposta è in generale immediata
  - Gli utenti interagiscono solo con il Client
- La richiesta dei servizi è fatta solitamente mediante un meccanismo di remote procedure call

Architettura Client/Server



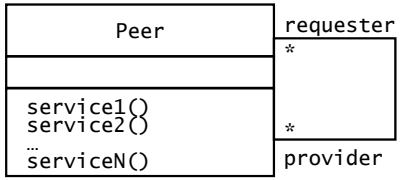
Caratteristiche client-server

- Vantaggi
  - La distribuzione dei dati è diretta
  - Usa in modo efficace i sistemi basati su rete. Può richiedere hardware poco costoso
  - Semplice aggiungere nuovi server o aggiornare server esistenti
- Svantaggi
  - Non c'è alcun modello di dati condivisi per cui i sottosistemi usano diverse organizzazioni dei dati. Lo scambio dei dati può essere inefficiente
  - Gestione ridondante in ogni server
  - Non c'è alcun registro centrale di nomi e servizi – può essere complicato scoprire quali server e servizi sono disponibili

Architettura Peer-to-Peer

- E' una generalizzazione dell'Architettura Client/Server
- Ogni sottosistema può agire sia come Client o come Server, nel senso che ogni sottosistema può richiedere e fornire servizi
- Il flusso di controllo di ogni sottosistema è indipendente dagli altri, eccetto per la sincronizzazione sulle richieste

Architettura peer-to-peer



Esempio

- Database che accetta le richieste dall'applicazione e notifica l'applicazione allorquando cambiano certi dati
- I sistemi peer-to-peer sono più difficili da progettare rispetto ai sistemi client/server poiché introducono la possibilità di deadlock e complicano il flusso di controllo

Esempio