Chapter 6,
System Design
Lecture 1

Design

La progettazione di un sistema (System Design) è la trasformazione del modello di analisi nel modello di progettazione del sistema

Scopo del system design

- 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
  - La gestione delle condizioni limite

Output del system design:

Un **modello** del sistema che include la decomposizione del sistema in sottosistemi e una chiara descrizione di ognuna delle strategie.

Perché il system design è così difficile?

- *Analisi:* si focalizza sul dominio di applicazione
- *Design:* si focalizza sul dominio di implementazione
  - **Gli sviluppatori devono raggiungere dei compromessi fra i vari obiettivi di design che sono spesso in conflitto gli uni con gli altri**
  - **Non possono anticipare tutte le decisioni relative alla progettazione non avendo una chiara idea del dominio della soluzione**

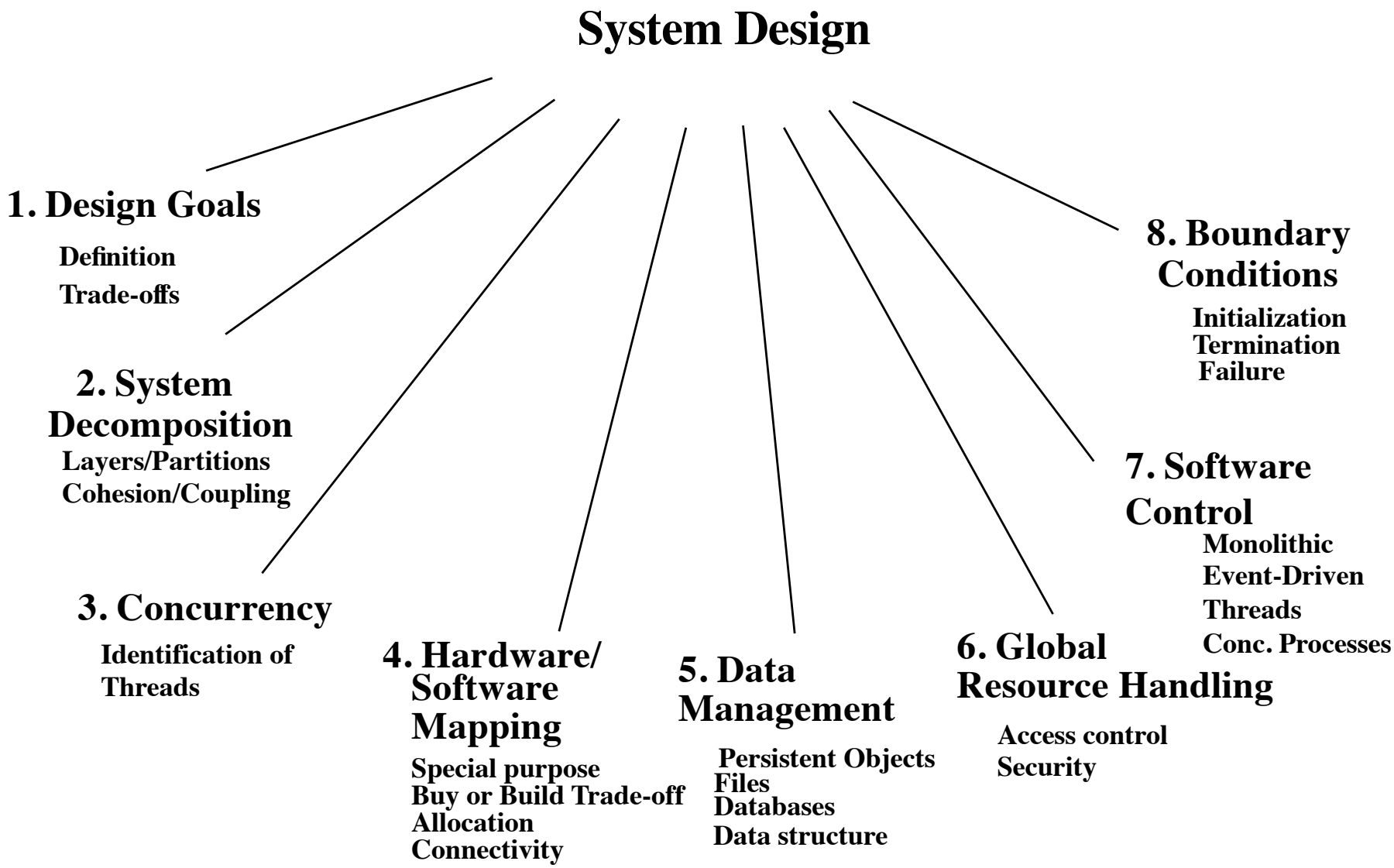
Attività di system design

Il system design è costituito da una serie di attività:

- ***Identificare gli obiettivi di design*:**
  - gli sviluppatori identificano quali caratteristiche di qualità dovrebbero essere ottimizzate e definiscono le priorità di tali caratteristiche
- ***Progettazione della decomposizione del sistema in sottosistemi*:**
  - basandosi sugli use case ed i modelli di analisi, gli sviluppatori decompongono il sistema in parti più piccole. Utilizzano stili architetturali standard.
- ***Raffinare la 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.

Per il momento ci concentriamo sulle prime due attività

System Design



I risultati dell'analisi dei requisiti

- Il modello di analisi descrive il sistema dal punto di vista degli attori e serve come base fra il cliente e gli sviluppatori.

- Non contiene informazioni sulla struttura interna del sistema, la sua configurazione hardware e, in generale, su come il sistema dovrebbe essere realizzato.

I risultati dell'analisi dei requisiti (2)

L'analisi fornisce in output il modello dei requisiti descritto dai seguenti prodotti:

- **Requisiti non funzionali e vincoli => quali tempo di risposta massimo, minimo throughput, affidabilità, piattaforma per il sistema operativo, etc.**
- **Use Case model => descrive le funzionalità del sistema dal punto di vista degli attori**
- **Object model => descrive le entità manipolate dal sistema**
- **Dynamic model => Un sequence diagram mostra la sequenza di interazioni fra gli oggetti che partecipano ad un caso duso**

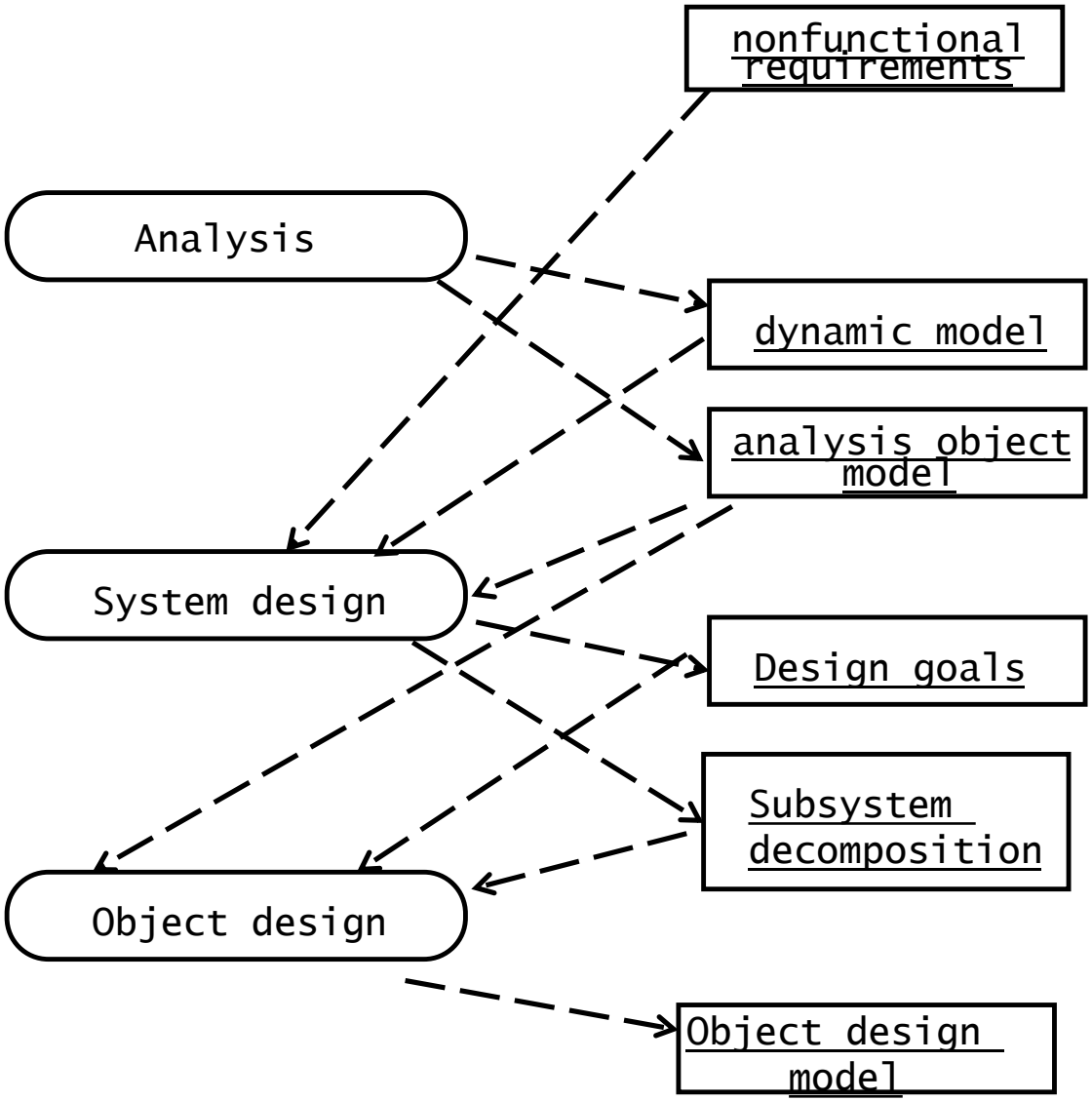
Prodotti del system design

- **Obiettivi di design**, descrivono la qualità del sistema (vengono derivati dalle richieste non funzionali).
- **Architettura software**, descrive:
  - **la decomposizione del sistema in termini delle responsabilità dei sottosistemi. Così ogni sottosistema può essere assegnato ad un team e realizzato indipendentemente.**
  - **le dipendenze fra sottosistemi,**
  - **l' hardware associato ai vari sottosistemi**
  - **le decisioni (politiche) relative a:**
    - **Control flow**
    - **Controllo degli accessi**
    - **Memorizzazione dei dati**
- **Boundary use case**, descrivono la configurazione del sistema, le scelte relative allo startup, allo shutdown ed alla gestione delle eccezioni.

Corrispondenza fra gli output della fase di analisi e il processo di design

- Nonfunctional requirements =>
  - **Activity 1: Design Goals Definition**
- Use Case model =>
  - **Activity 2: System decomposition (Selection of subsystems based on functional requirements, coherence, and coupling)**
- Object model =>
  - **Activity 4: Hardware/software mapping**
  - **Activity 5: Persistent data management**
- Dynamic model =>
  - **Activity 3: Concurrency**
  - **Activity 6: Global resource handling**

Le attività del system design



Concetti di System Design

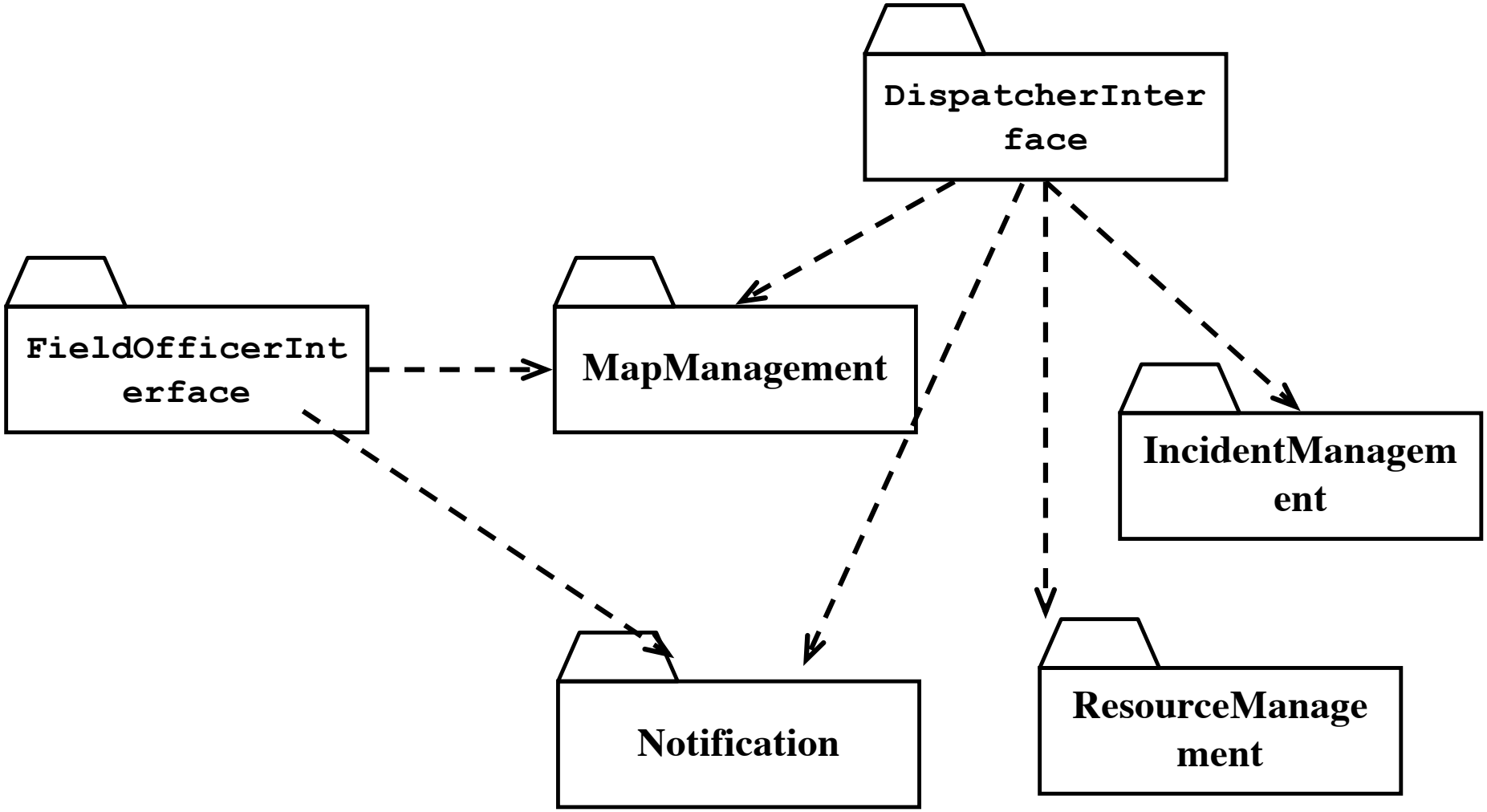
Concetti di System Design

- Descriviamo in dettaglio:
  - Il concetto di **sottosistema** e come è collegato alle classi
  - Le **interfacce** dei sottosistemi poichè i sottosistemi forniscono dei **servizi** agli altri sottosistemi
  - **Servizio** è un insieme di operazioni correlate che condividono uno scopo comune
- Due proprietà dei sottosistemi: coupling e cohesion
  - **Coupling**, misura la dipendenza fra due sottosistemi
  - **Cohesion**, misura la dipendenza fra classi entro un sottosistema
- Esaminiamo due tecniche per collegare fra loro due sottosistemi: layering e partitioning
  - **Layering,** consente ad un sistema di essere organizzato come una gerarchia di sottosistemi in cui ognuno fornisce servizi al sistema di livello superiore utilizzando i servizi forniti dal sistema al livello inferiore
  - **Partitioning**, organizza sottosistemi come peer che mutuamente forniscono differenti servizi agli altri sottosistemi
- Descriveremo alcune architetture software che sono largamente utilizzate

Sottosistemi e classi

- Per ridurre la complessità della soluzione, decomponiamo il sistema in parti più piccole, chiamate sottosistemi.
- Un **sottosistema** è costituito da un certo numero di classi del dominio della soluzione
- Un **sottosistema** tipicamente corrisponde alla parte di lavoro che può essere svolta da un singolo sviluppatore o da un team di sviluppatori
- Decomponendo il sistema in sottosistemi relativamente indipendenti, i team di progetto possono lavorare sui sottosistemi individuali con un minimo overhead di comunicazione (è importante che siano definiti gli obiettivi di design chiari a tutti i sottoteam).
- Nel caso di sottosistemi complessi, applichiamo ulteriormente questo principio e li decomponiamo in sottosistemi più semplici.

Esempio: decomposizione in sottosistemi



Si notino le dipendenze di UML package

Modellazione di sottosistemi

- Subsystem (UML: Package)
  - **Collezioni di classi, associazioni, operazioni e vincoli che sono correlati**

- JAVA fornisce i package che sono costrutti per modellare i sottosistemi

Servizi e interfacce di sottosistemi

- Un **sottosistema** è caratterizzato dai **servizi** che fornisce agli altri sottosistemi
- Un **servizio** è un insieme di **operazioni** correlate fornite dal sottosistema per uno specifico scopo
  - **ES. un sottosistema che fornisce un servizio di gestione delle risorse, definisce le operazioni per verificare la disponibilità di una risorsa, assegnarla a chi ne ha fatta richiesta, registrare la restituzione della risorsa, ecc.**
- L'insieme di operazioni di un sottosistema che sono disponibili agli altri sottosistemi forma l'**interfaccia** del sottosistema
  - **L'interfaccia** di un sottosistema include **il nome delle operazioni, i loro parametri, il loro tipo ed i loro valori di ritorno.**
- Il **system design** si focalizza sulla definizione dei **servizi** forniti da ogni sottosistema in termini di:
  - **Operazioni**
  - **Loro parametri**
  - **Loro comportamento ad alto livello**
- **L'Object design** si focalizza sulle **operazioni**, l'Application Programmer Interface (API), che raffina ed estende le interfacce, definendo i tipi dei parametri ed i valori di ritorno di ogni operazione.

Definizione dei sottosistemi in termini dei servizi

- Aiuta a concentrarci sull'interfaccia piuttosto che sulla implementazione
- Quando si descrive un'interfaccia si dovrebbero cercare di ridurre info sull'implementazione (non riferirsi alle strutture dati interne, liste linkate, hash table…)
- Ciò consente di ridurre l'impatto dei cambiamenti di un sottosistema sugli altri

Coupling e Cohesion: Proprietà dei Sottosistemi

- We want highly *cohesive* classes within each subsystem and loosely *coupled* subsystems.
- This has an impact on comprehensibility, maintainability, the ease with which teams can work concurrently on subsystems, the integration of subsystems, etc.
- *Coupling:* A measure of how closely two classes or subsystems are connected
- *Cohesion:* A measure of how well a class or subsystem is tied together

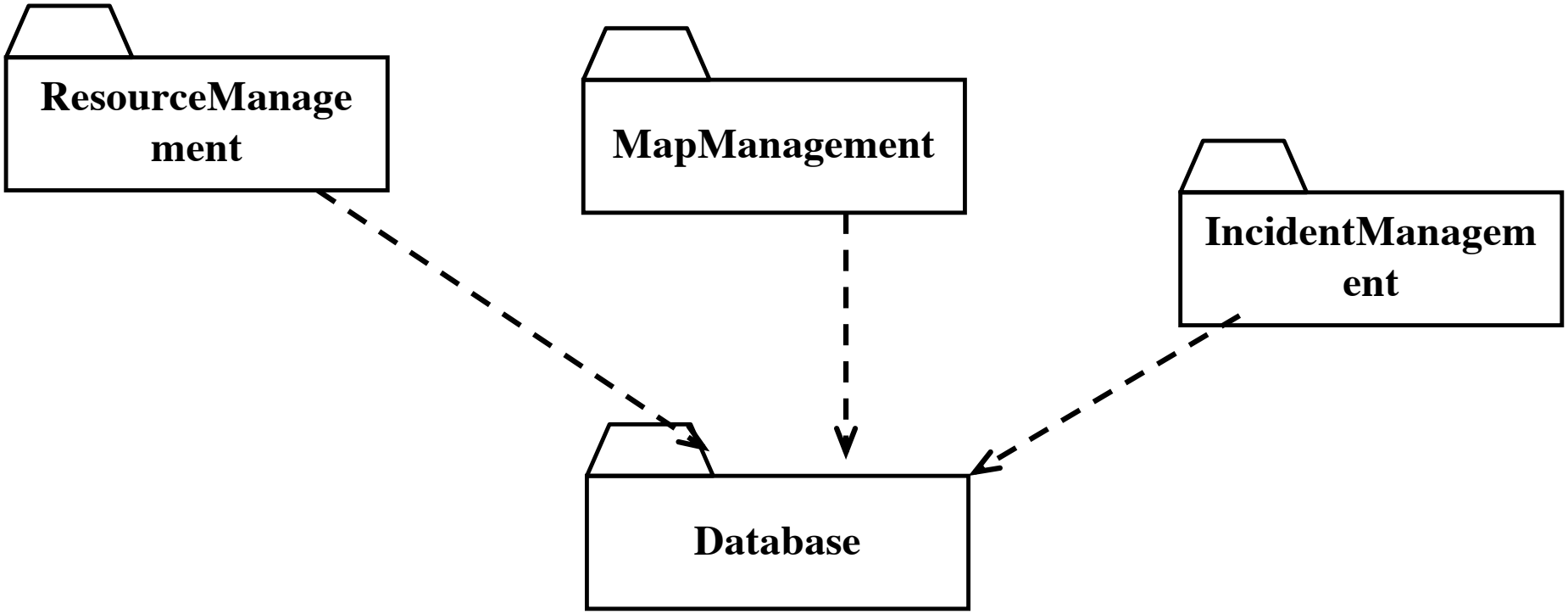
Coupling

- **Obiettivo**: ridurre la complessità
- **Coupling** è il numero di dipendenze fra due sottosistemi
  - **Connections among classes/subsystems: aggregation, specialization, calling operations**

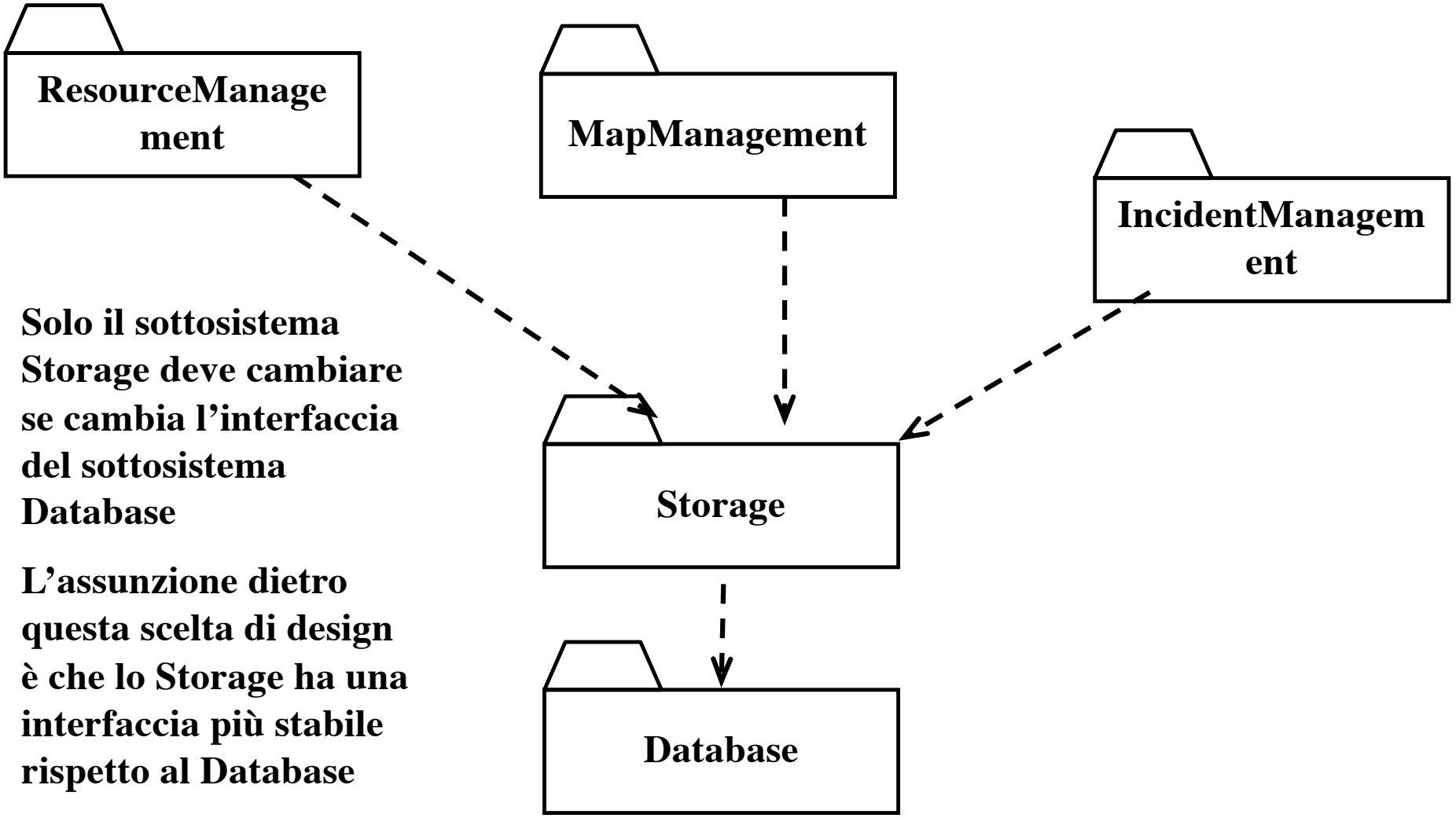
- Se due sistemi sono scarsamente accoppiati (**loosely coupled**) sono relativamente indipendenti: modifiche su uno dei sottosistemi avranno probabilmente pochi impatti sull'altro.
- Se due sistemi sono fortemente accoppiati (**strongly coupled**), una modifica su di un sottosistema probabilmente avrà impatti sull'altro.
- Proprietà desiderabile di una decomposizione di sottosistemi: che siano loosely coupled

Esempio: accesso diretto al database

Tutti i sottosistemi accedono al database direttamente, rendendoli vulnerabili ai cambiamenti dell'interfaccia del sottosistema Database



Esempio: accesso al database attraverso un sottosistema di Storage



Solo il sottosistema Storage deve cambiare se cambia l'interfaccia del sottosistema Database

L'assunzione dietro questa scelta di design è che lo Storage ha una interfaccia più stabile rispetto al Database

Coupling: osservazioni

- Nell'esempio proposto abbiamo ridotto il coupling fra i quattro sottosistemi.
- Abbiamo aumentato la complessità.
- Con l'obiettivo di ridurre il coupling si rischia di aggiungere livelli di astrazione che consumano tempo di sviluppo e tempo di elaborazione.
- Un coupling estremamente basso va perseguito solo se ci sono elevate probabilità che qualche sottosistema cambi.

COESIONE

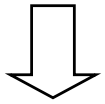
- Classes:
  - Le operazioni costituiscono un “intero” funzionale
  - Gli attributi e le strutture dati descrivono gli oggetti in stati ben definiti, che sono modificati dalle operazioni
  - Le operazioni si usano a vicenda
- Sottosistemi:
  - Le classi dei sottosistemi sono concettualmente correlate
  - Le relazioni strutturali tra le classi sono fondamentalmente generalizzazioni e aggregazioni (formano cluster)
  - Le operazioni chiavi sono eseguite all’interno dei sottosistemi

Cohesion (coesione)

- Misura le dipendenze entro un sottosistema
- **Alta coesione (high cohesion)**: le classi di un sottosistema realizzano compiti simili e sono collegate le une alle altre (attraverso associazioni)
- **Bassa coesione (low cohesion)**: il sottosistema contiene un certo numero di oggetti non correlati

Cohesion and Coupling trade-off

- Aumentando la coesione $\Longrightarrow$ aumenta il numero di sottosistemi



- Aumenta il numero di interfacce $\Longrightarrow$ Aumenta il coupling

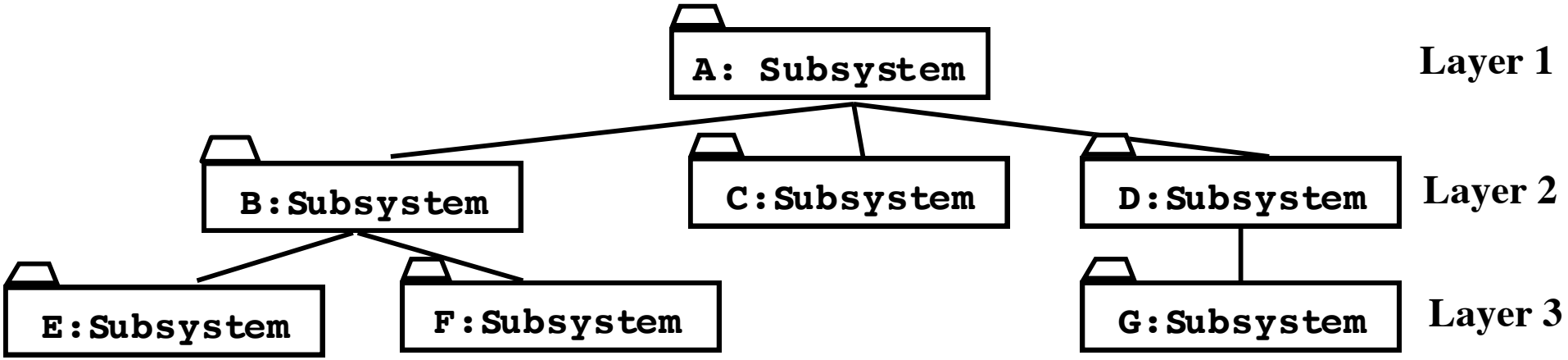
- **Euristica**: gli sviluppatori possono trattare ad ogni livello di astrazione un numero di concetti pari a 7 ± 2
  - **Qualcosa non va se:**
    - **ci sono più di 9 sottosistemi ad un livello di astrazione**
    - **Un sottosistema fornisce più di 9 servizi**

Layers e partizioni

Layers e partizioni

- Un sistema grande è di solito decomposto in sottosistemi usando layer e partizioni.
- Una **decomposizione gerarchica** di un sistema consiste di un insieme ordinato di **layer** (strati).
- Un layer è un raggruppamento di sottosistemi che forniscono servizi correlati, eventualmente realizzati utilizzando servizi di altri layer.
- Un layer può dipendere solo dai layer di livello più basso
- Un layer non ha conoscenza dei layer dei livelli più alti
- **Architettura chiusa**: ogni layer può accedere solo al layer immediatamente sotto di esso
- **Architettura aperta**: un layer può anche accedere ai layer di livello più basso

Decomposizione di sottosistemi in Layer



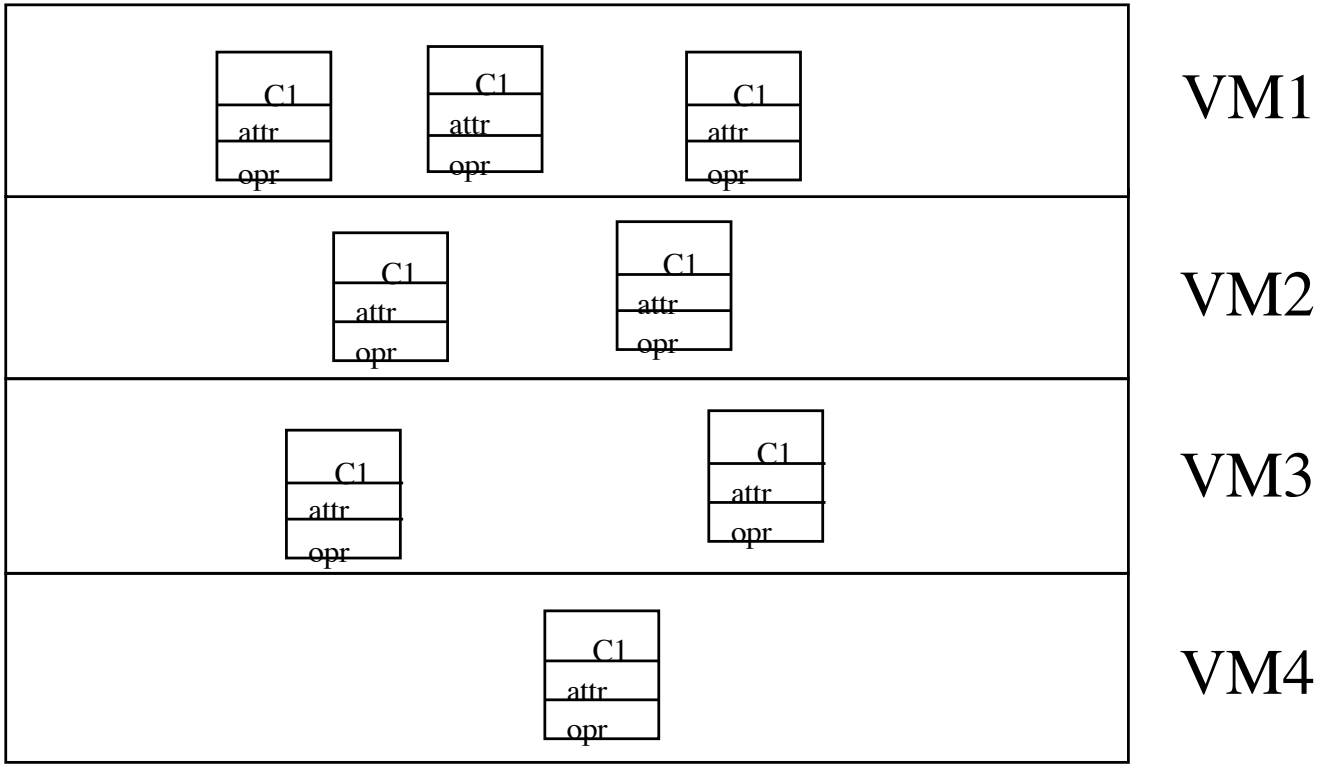
- Euristiche per la decomposizione di sottosistemi::
- Non più di 7±2 sottosistemi per layer
  - **Più sottosistemi accrescono la cohesion ma anche la complessità (più servizi)**
- Non più di 5±2 layer

Proprietà dei Layered Systems

- I Layered system sono gerarchici. La gerarchia riduce la complessità.
- Le architetture chiuse sono più portabili
- Le architetture aperte sono più efficienti.
- Se un sottosistema è un layer, spesso è chiamato macchina virtuale.

Macchina Virtuale (Dijkstra, 1965)

- Un sistema dovrebbe essere sviluppato da un insieme di **macchine virtuali**, ognuna costruita in termini di quelle al di sotto di essa.

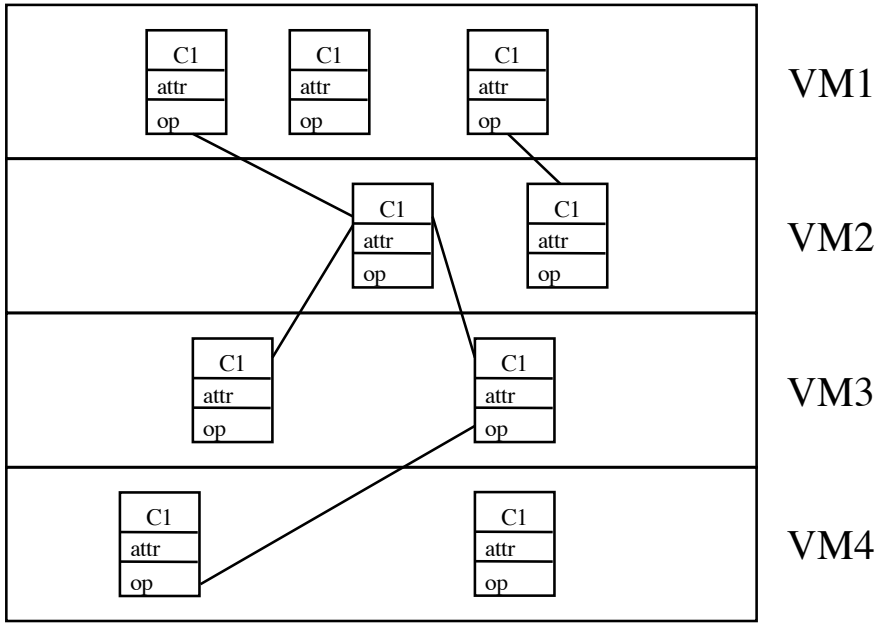


Macchina Virtuale (2)

- Una macchina virtuale è un'astrazione che fornisce un insieme di attributi e operazioni
- Una macchina virtuale è un sottosistema connesso a macchine virtuali di livello più alto e più basso attraverso associazioni "provides services for".
- Le macchine virtuali possono implementare due tipi di architetture software: architetture chiuse e architetture aperte.

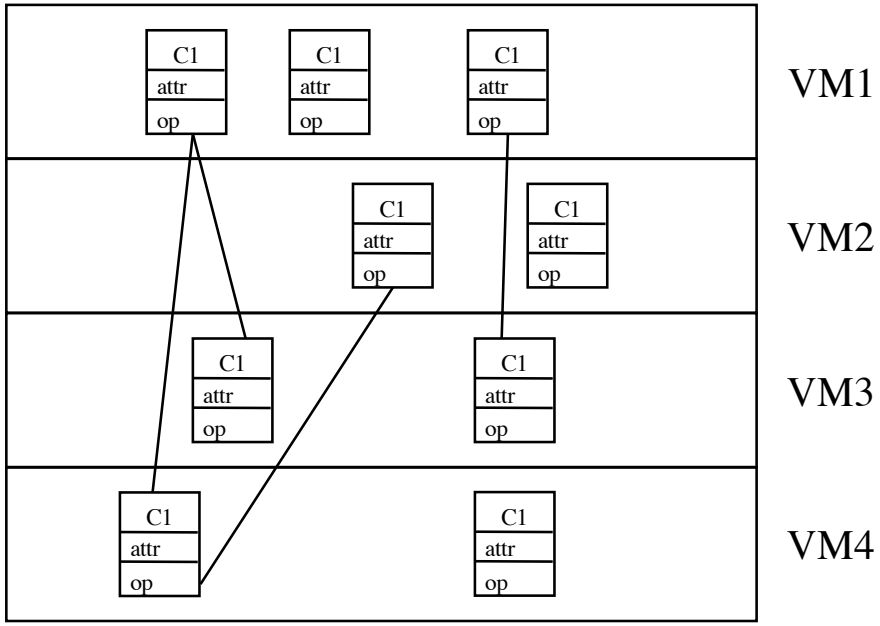
Architettura Chiusa (Opaque Layering)

- Una macchina virtuale può solo chiamare le operazioni dello strato sottostante
- Design goal: alta manutenibilità



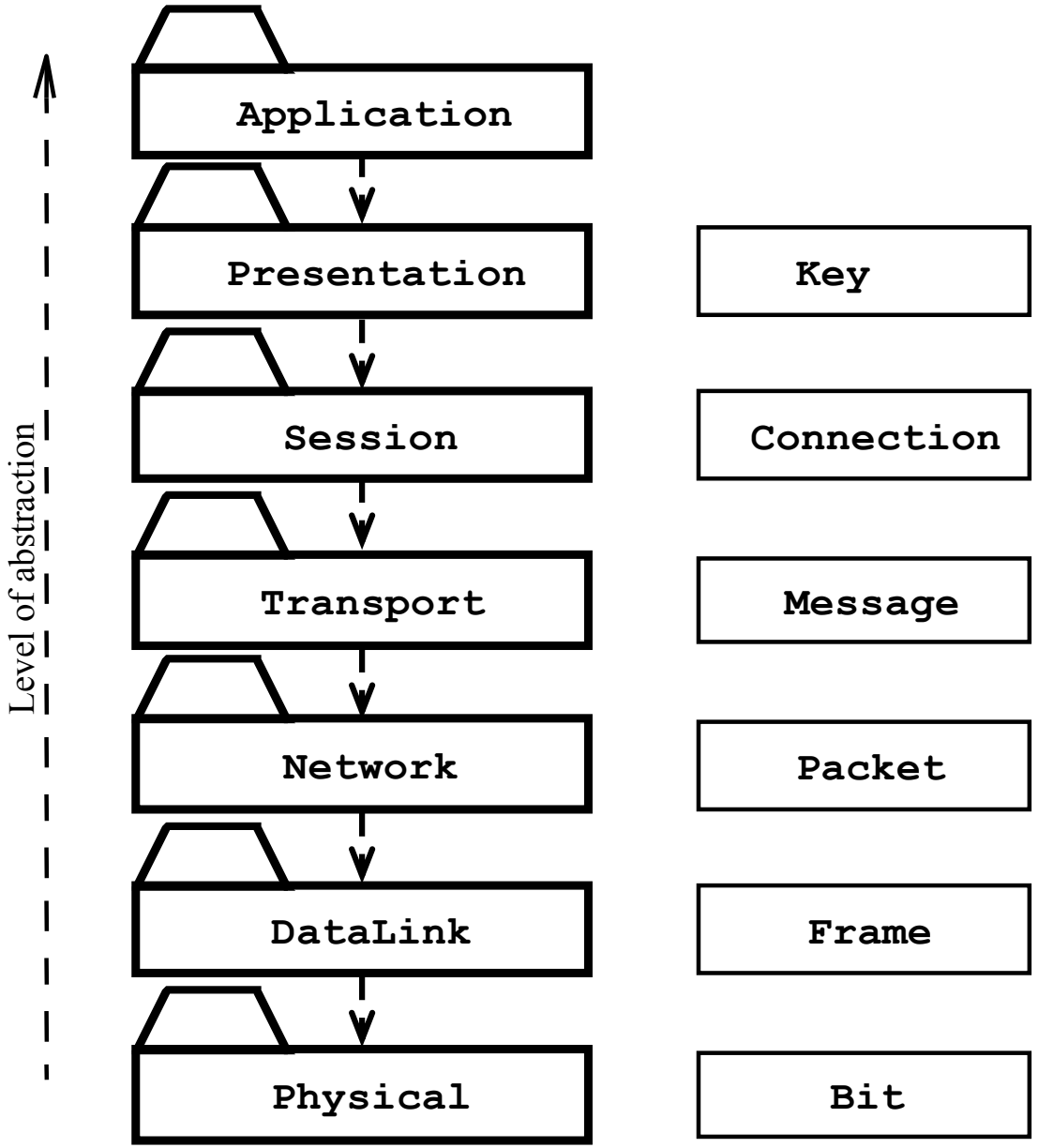
Architettura Aperta (Transparent Layering)

- Una macchina virtuale può utilizzare i servizi delle macchine dei layer sottostanti
- Design goal: efficienza relativa al tempo di esecuzione (runtime)



Esempio di Closed Architecture

- ISO's OSI Reference Model
  - **ISO = International Standard Organization**
  - **OSI = Open System Interconnection**
- Il modello di riferimento definisce 7 layer di protocolli di rete e metodi di comunicazione tra i layer.



Vantaggi e svantaggi delle architetture chiuse

- Vantaggi
  - **Basso accoppiamento tra le componenti**
  - **Integrazione e testing incrementale**
- Svantaggi
  - **Ogni livello aggiunge overhead in termini di tempo e memoria**
  - **Diventa difficile soddisfare alcuni obiettivi di design (performance)**
  - **Aggiungere funzionalità al sistema può essere difficile**

Basic Layer pattern

Un sistema di Base contiene tipicamente i sottosistemi

- **Interface**
- **Function**
- **Model**.

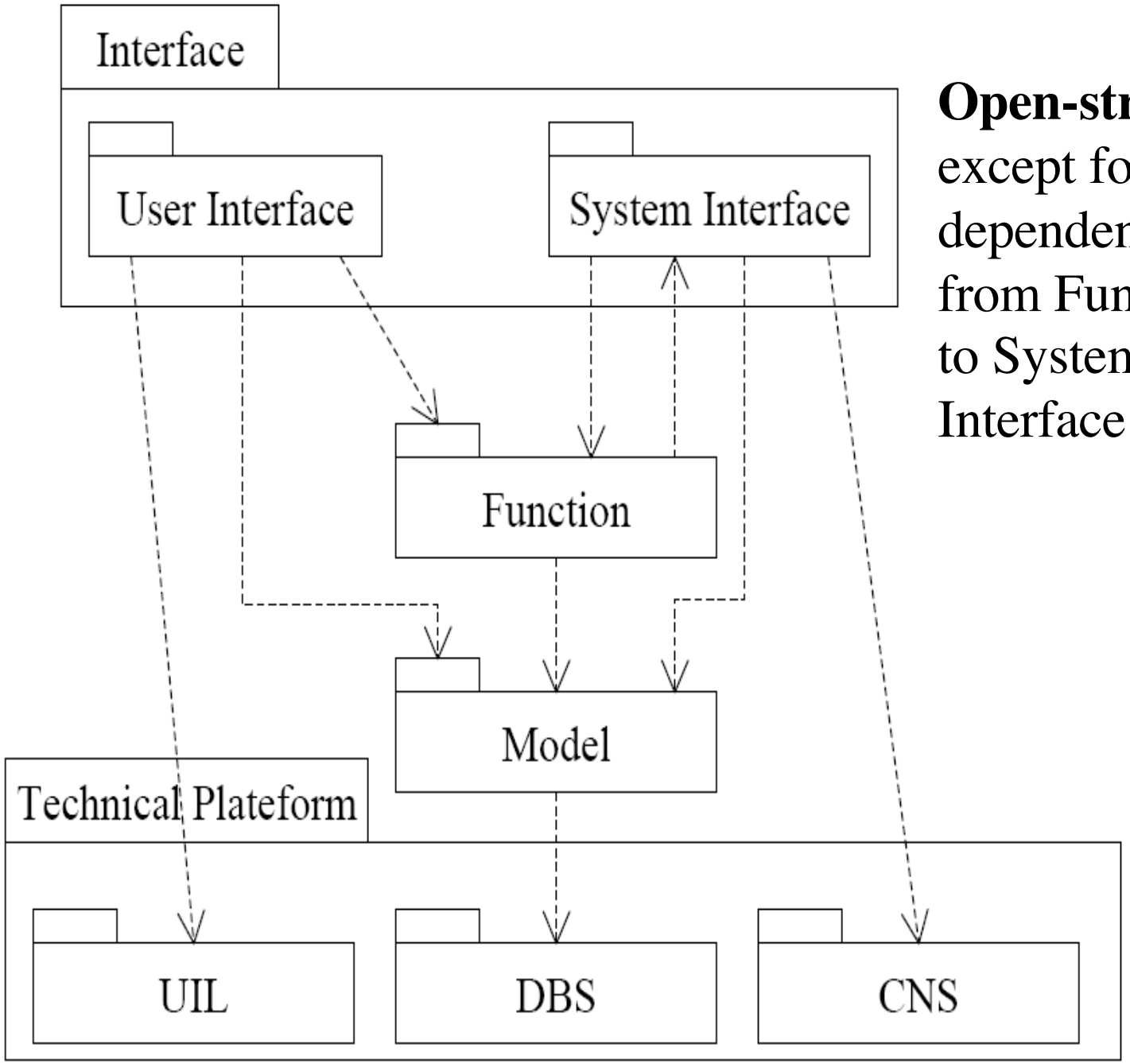
✓ Il sottosistema **Model** contiene gli oggetti **entity** del dominio di applicazione.

✓ Il sottosistema **Function** contiene gli oggetti **control e la logica dell'applicazione.**

✓ Il sottosistema **Interface** contiene gli oggetti **boundary/interface**. E' ulteriormente decomposto in

- **user interface**
- **system interface** (devices e sistemi esterni).

Basic Layer Pattern



Open-strict, except for the dependency from Function to System Interface.

Basic Layer pattern - Discussion

- **UIL** stands for any library which is part of the technical platform that provide **GUI capabilities** (e.g., awt, swing in Java).
- **DBS** stands for **database subsystem** and represents any persistent data facility which is part of the platform, e.g., relational DBMS, object-oriented DBMS.
- **CNS** stands for **communication and network subsystem** and contains any capability part of the technical platform to communicate with other hosts/machines on a local/wide-area network, wireless link, etc.

Partition

- Un altro approccio per trattare con la complessità consiste nel partizionare (partition) il sistema in sottosistemi **pari** (**peer**) fra loro, ognuno responsabile di differenti classi di servizi.

Relazioni Layer e Partition tra Sottosistemi

- In generale, una decomposizione in sottosistemi è il risultato di entrambi partition e layering.

- Ogni sottosistema aggiunge overhead di elaborazione a causa della sua interfaccia verso gli altri sottosistemi.
- Eccessiva frammentazione accresce la complessità.

Architetture Software

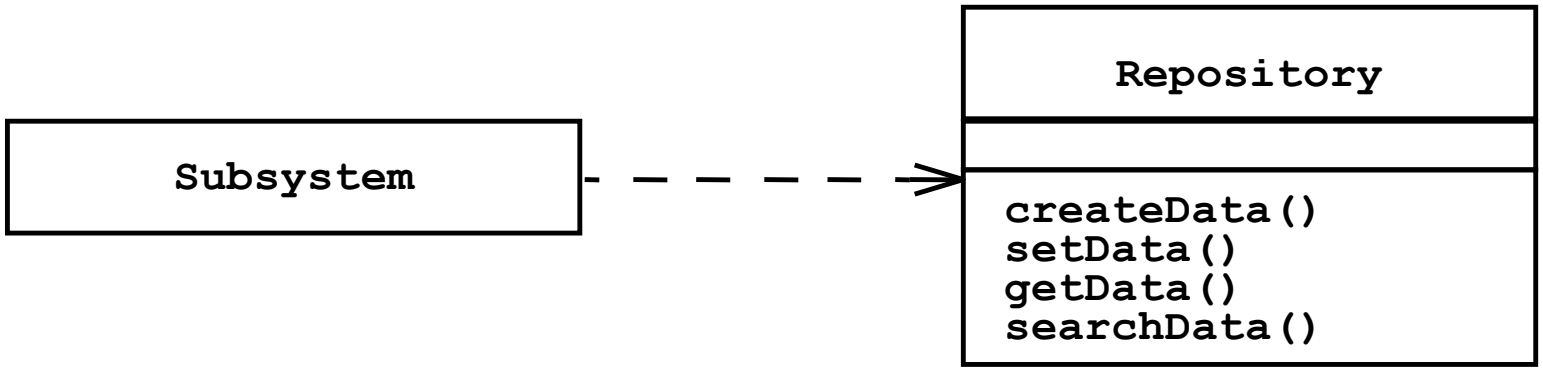
Architetture Software

- Decomposizione in sottosistemi
  - **Identificazione dei sottosistemi, servizi e relazioni fra di essi.**
- La specifica della decomposizione in sottosistemi è **critica**:

  è difficile modificarla quando lo sviluppo è partito, poiché molte interfacce dei sottosistemi dovrebbero essere cambiate.
- **Architettura Software**: decomposizione del sistema, flusso di controllo globale, gestione delle condizioni limite, protocollo di comunicazione tra sottosistemi
- Patterns per architetture software
- Stili architetturali che possono essere usati come base di architetture software:
  - **Architettura Client/Server**
  - **Architettura Peer-To-Peer**
  - **Architettura a Repository**
  - **Model/View/Controller**

Architettura a Repository

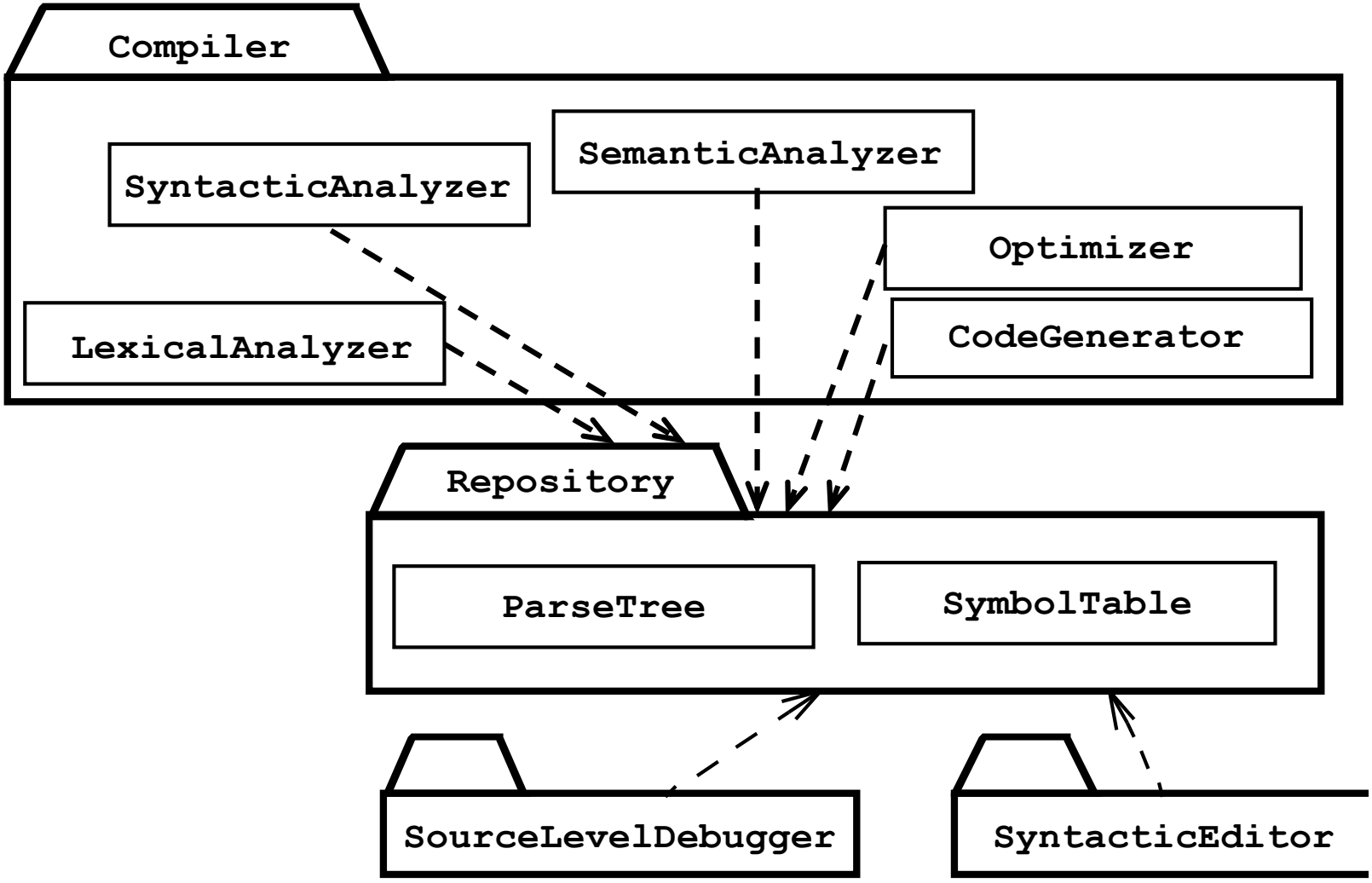
- I sottosistemi accedono e modificano una singola struttura dati chiamata **repository**.
- I sottosistemi sono loosely coupled (interagiscono solo attraverso il repository)
- Il repository non ha conoscenza degli altri sottosistemi



Architettura a Repository

- Esempi di sistemi con architettura a repository:
  - **Compilatori**
  - **Database Management Systems: bank systems, payroll systems**
  - **Software Development Environments**
- Il Control flow è determinato
  - **dai sottosistemi (locks, synchronization primitives)**
    - **Nel compilatore ogni tool è invocato dall'utente. Il repository assicura solo che gli accessi concorrenti siano serializzati**
  - **dal repository (triggers sui dati invocano i vari sottosistemi)**
    - **I sottosistemi possono essere invocati sulla base dello stato della struttura dati centrale (sistemi blackboard)**
      - **Es. Hearsy II: sistema di riconoscimento vocale**

Esempio di Architettura a Repository

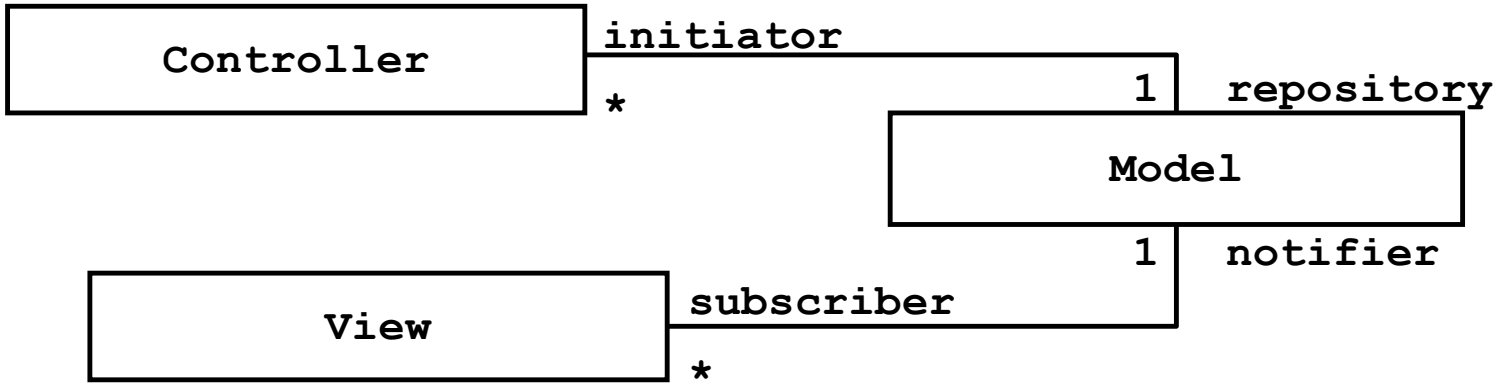


Repository: vantaggi e svantaggi

- I repository sono adatti per applicazioni con task di **elaborazione dati che cambiano di frequente**.
- Una volta che un repository centrale è stato definito, nuovi servizi nella forma di **sottosistemi addizionali** possono essere definiti facilmente
- **Problemi:**
  - **il repository centrale può facilmente diventare un collo di bottiglia per aspetti sia di prestazione, sia di modificabilità**
  - **Il coupling fra ogni sottosistema ed il repository è alto, così è difficile cambiare il repository senza avere un impatto su tutti i sottosistemi.**

Model/View/Controller

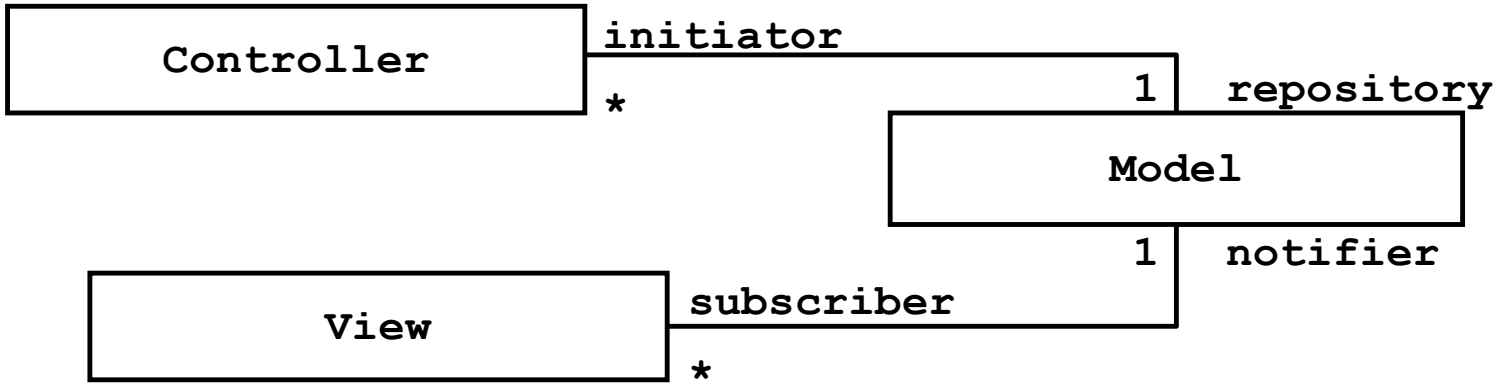
- In questo stile architetturale i sottosistemi sono classificati in 3 tipi differenti:
  - **Sottosistema Model**: mantiene la conoscenza del dominio di applicazione (fornisce I metodi per accedere ai dati utili all'applicazione)
  - **Sottosistema View**: visualizza all'utente gli oggetti del dominio dell'applicazione (i dati contenuti nel model e si occupa dell'interazione con utenti)
  - **Sottosistema Controller**: responsabile della sequenza di interazioni con l'utente (riceve i comandi dell'utente (in genere attraverso il view) e li attua modificando lo stato degli altri due componenti)



Model/View/Controller (2)

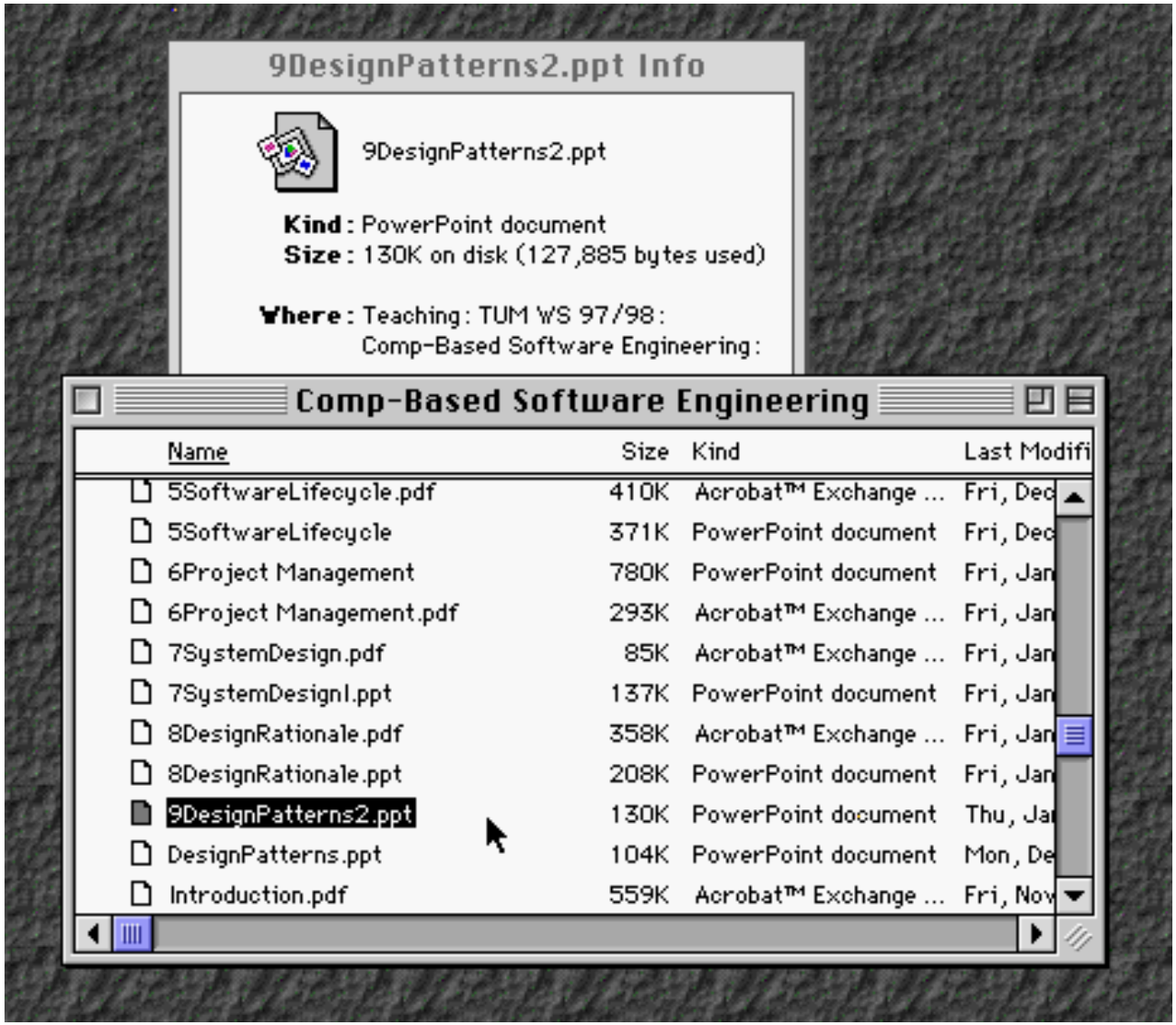
MVC è un caso particolare di architettura di tipo repositoty:

- **Il sottosistema Model implementa la struttura dati centrale,**
- **il sottosistema Controller gestisce il control flow: ottiene gli input dall'utente e manda messaggi al Model**
- **I sottosistemi View visualizzano il Model e sono notificati (attraverso un protocollo subscribe/notify) ogni volta che il Model è modificato**



Example of a File System based on MVC Architecture

Il "model" è il nome del file 9DesignPatterns2.ppt



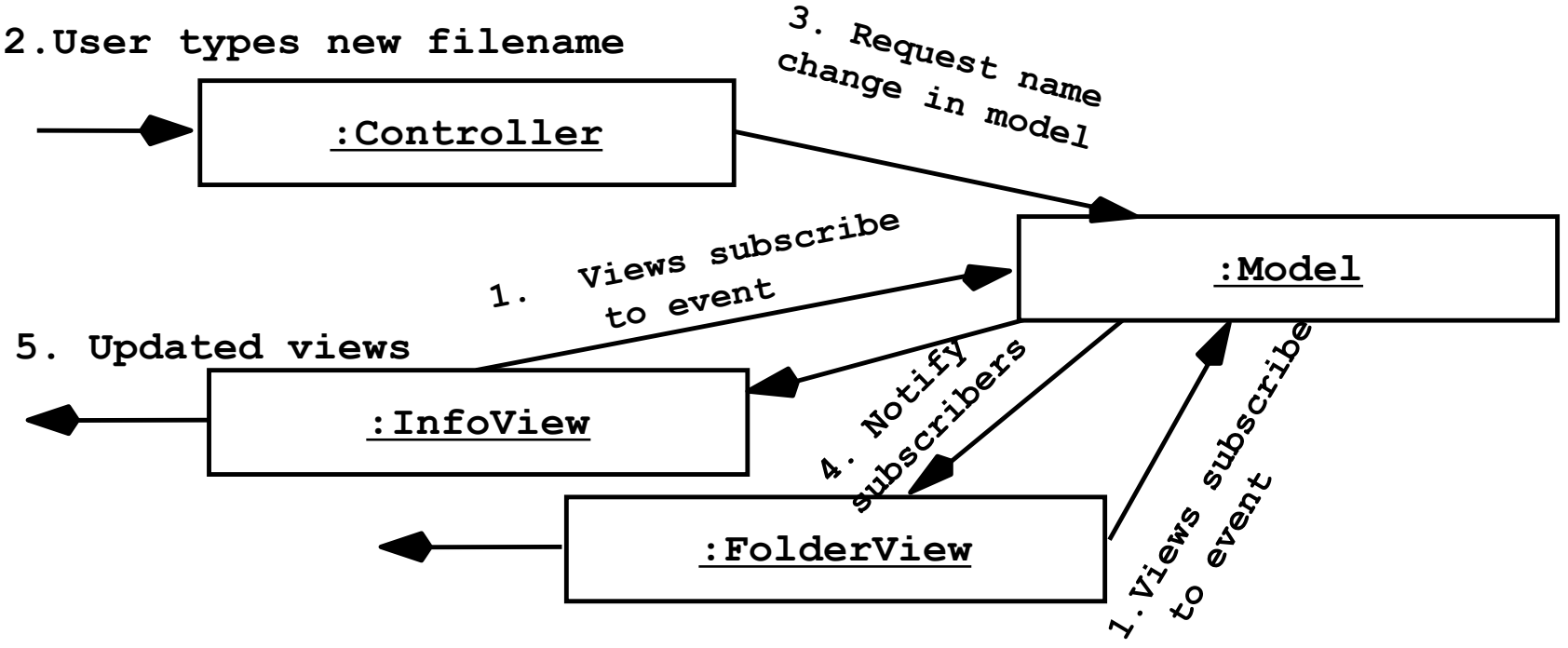
Due viste di un file system:

1. **La finestra in basso visualizza il folder Comp-Based Software Engineering**
2. **La finestra in alto le informazioni relative al file 9DesignPatterns2.ppt**

Il nome del file 9DesignPatterns2.ppt appare in tre posti

Se il nome del file cambia tutte le view sono aggiornate dal controller

Sequenza di Eventi: Cambiamo il nome del file



1. **InfoView e FolderView sottoscrivono i cambi al modello File quando sono creati**
2. **L'utente digita il nuovo nome del file**
3. **Il Controller manda la richiesta al Model**
4. **Il Model cambia il nome del file e notifica i subscriber del cambiamento**
5. **Entrambi InfoView e FolderView sono aggiornati in modo tale che l'utente veda i cambi in modo consistente**

Model/View/Controller: motivazioni

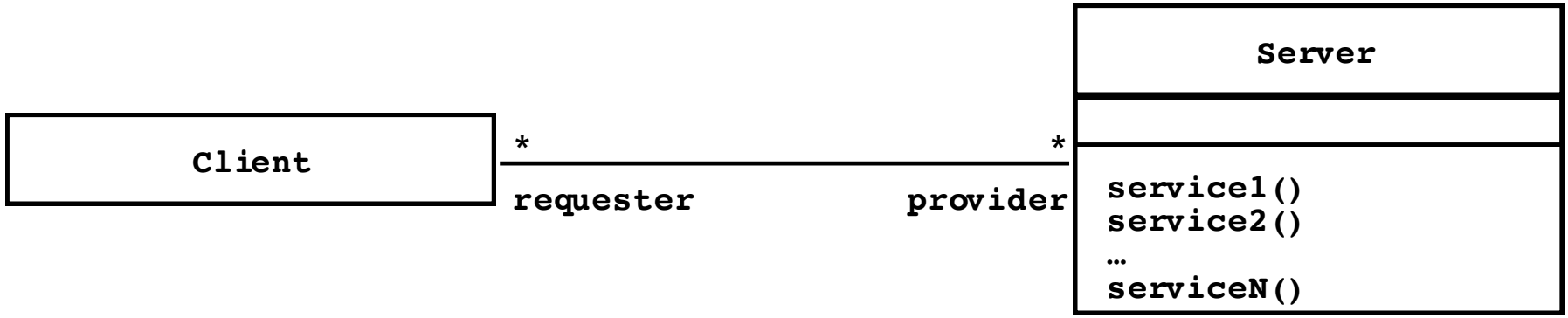
- Il motivo per cui si separano Model, View e Controller è che le interfacce utenti sono soggette a cambiamenti più spesso di quanto avviene per la conoscenza del dominio (il Model)
- MVC è appropriato per i sistemi interattivi, specialmente quando si utilizzano viste multiple dello stesso Model.
- Introduce lo stesso collo di bottiglia visto per lo stile architetturale a Repository

MVC

- è un **pattern architetturale** molto diffuso nello sviluppo di interfacce grafiche di sistemi software object-oriented.
- Originariamente impiegato dal linguaggio Smalltalk,
- E'stato esplicitamente o implicitamente sposato da numerose tecnologie moderne come framework basati su:
  - **PHP** (**Symfony**, **Zend Framework**),
  - **Ruby** (**Ruby on Rails**),
  - **Python** (**Django**, **TurboGears**, **Pylons**),
  - **Java** (**Swing**, **JSF** e **Struts**),
  - **.NET**

Client/Server Architecture

- Architettura originariamente sviluppata per gestire la **distribuzione tra alcuni processori geograficamente separat**i, e.g. sistema di autorizzazione di carte di credito
  - **Un sottosistema, detto *server*, fornisce specifici servizi, (stampa, gestione dei dati, ecc.)**
  - **Un insieme di client che richiedono questi servizi**
  - **Una rete che consente ai client di accedere ai server**
- I Client conoscono l'interfaccia del Server (i suoi servizi)
- I Server non conoscono le interfacce dei Client
- Gli utenti interagiscono solo con il Client
- Il flusso di controllo nei client e nei server è indipendente



Client/Server Architecture (2)

- Spesso usato nei sistemi di database:
  - **Front-end: applicazione utente (client)**
  - **Back end: accesso e manipolazione del Database (server)**
- Funzioni eseguite dal client:
  - **Fornire interfaccia utente personalizzata**
  - **Elaborazione front-end dei data, per verificare vincoli**
  - **Iniziare la transazione quando i dati sono stati collezionati**
- Funzioni eseguite dal server di database :
  - **Gestione centralizzata dei dati**
  - **Garantire integrità dei dati e consistenza del database**
  - **Garantire la sicurezza del Database**
  - **Gestire la concorrenza delle operazioni (multiple user access)**
  - **Elaborazioni centralizzate (per esempio archiviazione)**

Client/Server Architecture

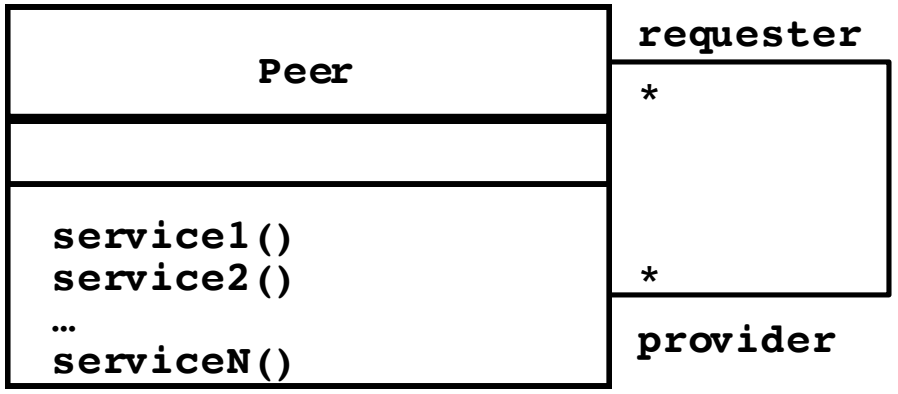
- L'esempio è un caso speciale di architettura a repository in cui la struttura dati centrale è gestita da un processo
- In generale nelle architetture client/server non abbiamo un singolo server!
  - **Sul Web (istanza di architettura client/server) un client può accedere a dati da migliaia di diversi server**
- Client Server architecture well suited for distributed systems that manage large amounts of data

Problems with Client/Server Architectures

- Peer-to-peer communication is often needed
- Example: Database receives queries from application but also sends notifications to application when data have changed

Peer-to-Peer Architecture

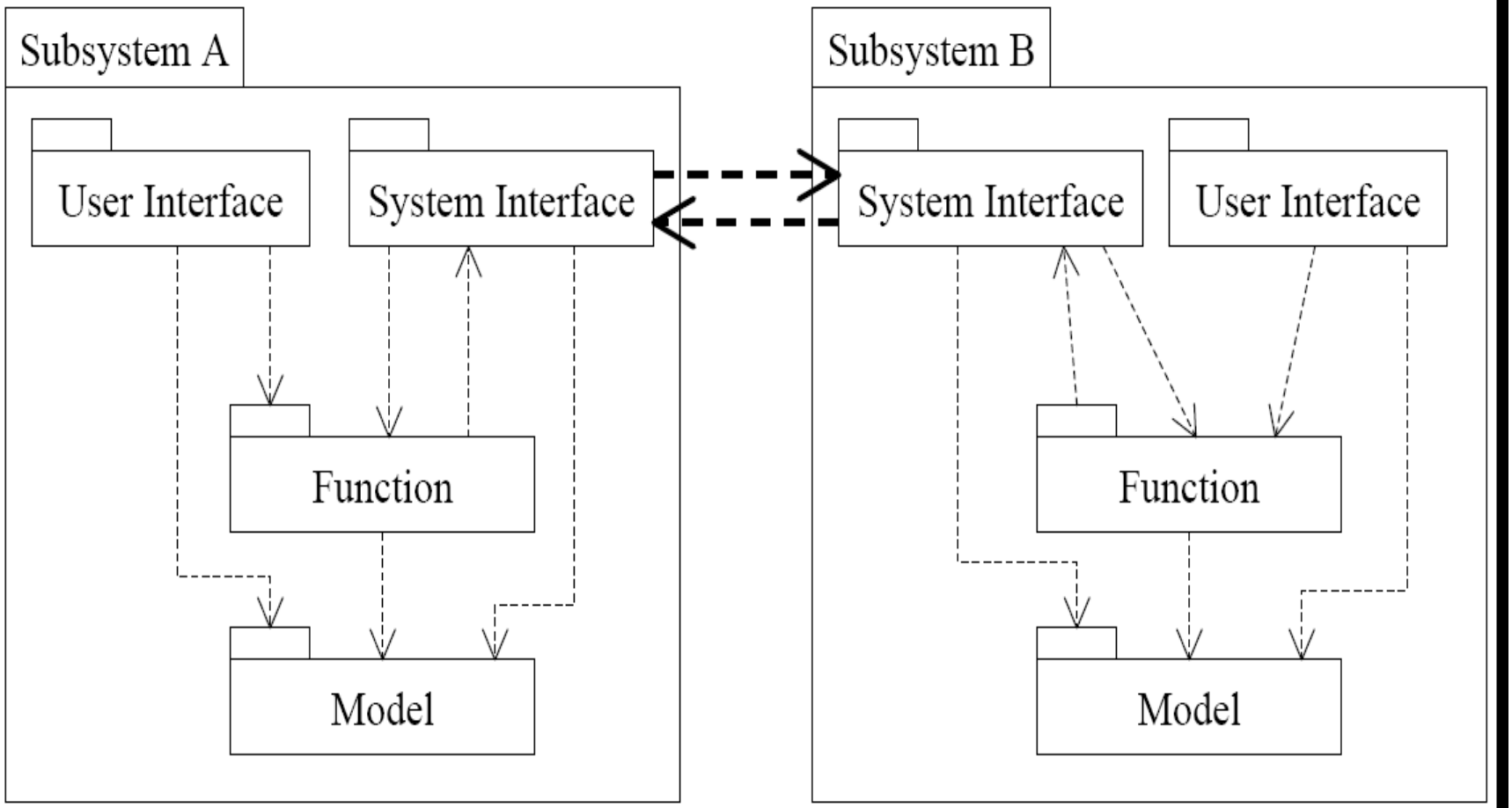
- E' una generalizzazione dell'Architettura Client/Server
- Ogni sottosistema può agire sia come Client o come Server: può richiedere e fornire servizi.
- Il control flow di ogni sottosistema è indipendente dagli altri, eccetto per la sincronizzazione sulle richieste
- Esempio: database che accetta richieste da una applicazione e notifica I cambiamenti sui dati all'applicazione
- Introducono la possibilità di deadlock e complicano il flusso di controllo



Large Systems

- Simple systems: Basic Layer Pattern
- For large systems, you must further decompose the system
- Several independent subsystems that communicate with each other
- Each subsystem is like an independent system with its own Model, Function, and Interface subsystems.
- The System interface subsystem provides a coherent interface to other subsystems for accessing the given subsystem's functionality.

Example


Subsystem A
User Interface
System Interface
Function
Model
Subsystem B
System Interface
User Interface
Function
Model

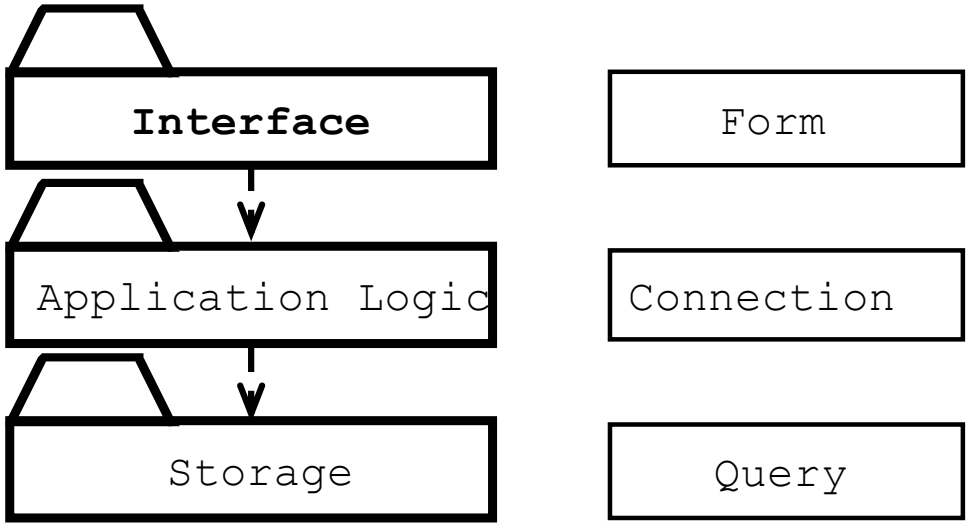
Distribution Patterns in CS Architectures

Client	Server	Architecture
U	U+F+M	Distributed presentation
U	F+M	Local Presentation
U+F	F+M	Distributed Functionality
U+F	M	Centralized Data
U+F+M	M	Distributed Data

Model (M), Function (F), and User Interface (U) subsystems

Three-tier Architecture

- I sottosistemi sono organizzati in tre strati
- *Interface layer*, include tutti i boundary object che interfacciano con l'utente
- *L'application logic layer*, include tutti gli oggetti relativi al controllo e alle entità che realizzano l'elaborazione, le regole di verifica e la notifica richiesta dall'applicazione
- Lo *storage layer* effettua la memorizzazione, il recupero e l'interrogazione di oggetti persistenti



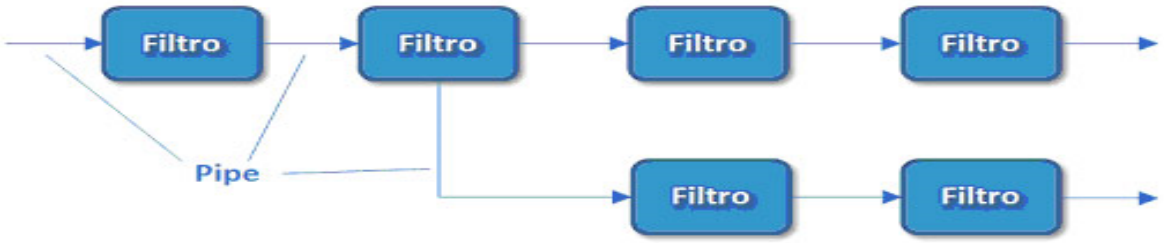
La separazione dell'interfaccia dalla logica applicativa consente di modificare e/o sviluppare diverse interfacce utente per la stessa logica applicativa

Architetture a flusso di dati (pipeline)

- Questo stile architetturale si rivela efficace nel caso in cui si abbia un insieme di dati in input da trasformare attraverso una catena di componenti e di filtri software al fine di ottenere una serie di dati in output.

- Ogni componente della catena lavora in modo indipendente rispetto agli altri, trasforma i dati in input in dati in output e delega ai componenti successivi le ulteriori trasformazioni.

- Ogni parte è inconsapevole delle modalità di funzionamento dei componenti adiacenti.
- Esempio: Shell di UNIX
- Non adatto per sistemi interattivi o che richiedono maggiore interazione tra le componenti



Scelta dei Sottosistemi

- Trovare I sottosistemi è simile ad individuare oggetti nell'analisi
- Criteri per la selezione dei sottosistemi: la maggior parte delle interazioni dovrebbe essere entro i sottosistemi, piuttosto che attraverso i limiti dei sottosistemi (alta coesione)
  - **Quale sottosistema chiama qualche altro per ottenere servizi?**
- Primary Question:
  - **Quale tipo di servizio è fornito dal sottosistema?**
- Secondary Question:
  - **Possono essere i sottosistemi ordinati gerarchicamente (layers)?**

Euristiche:

- **Tutti gli oggetti nello stesso sottosistema dovrebbero essere funzionalmente correlati.**
- **Assegnare gli oggetti identificati in un caso d'uso allo stesso sottosistema**
- **Creare un sottosistema dedicato per gli oggetti usati per muovere i dati fra i sottosistemi**
- **Minimizzare il numero di associazioni che attraversano i limiti dei sottosistemi**

Attività di system design

Identificare gli obiettivi di design

Identificare gli obiettivi di design

- E’ il primo passo del system design
- Identifica le qualità su cui deve essere focalizzato il sistema
- Molti design goal possono essere ricavati dai requisiti non funzionali o dal dominio di applicazione, altri sono forniti dal cliente, altri sono derivati da aspetti di management
- E’ importante formalizzarli esplicitamente poiché ogni importante decisione di design deve essere fatta seguendo lo stesso insieme di criteri

Criteri di design

- Possiamo selezionare gli obiettivi di design da una lunga lista di qualità desiderabili.
- I criteri sono organizzati in cinque gruppi:
  - **Performance**
  - **Dependability**
  - **Cost**
  - **Maintenance**
  - **End user criteria**

Criteri di Performance

- 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 immessa?**
  - **Troughput: quanti task il sistema deve portare a compimento in un periodo di tempo prefissato?**
  - **Memoria: quanto spazio è richiesto al sistema per funzionare?**

Dependability criteria

- Quanto sforzo deve essere speso per minimizzare i crash del sistema e le loro conseguenze?
- Rispondono alle seguenti domande:
  - **Robustness (robustezza).** Capacità di sopravvivere ad input non validi immessi dall'utente
  - **Reliability (affidabilità).** differenza fra comportamento specificato e osservato
  - **Availability (disponibilità).** Percentuale di tempo in cui il sistema può essere utilizzato per compiere normali attività
  - **Fault tolerance.** Capacità di operare sotto condizioni di errore
  - **Security**. Capacità di resistere ad attacchi di malintenzionati
  - **Safety.** Capacità di evitare di danneggiare vite umane, anche in presenza di errori e di fallimenti.

Cost criteria

- 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:
  - **Development Cost:** Costo di sviluppo del sistema iniziale
  - **Deployment cost:** costo relativo all'installazione del sistema e training degli utenti
  - **Upgrade cost:** costo di convertire i dati del sistema precedente. Questo criterio viene applicato quando nei requisiti è richiesta la compatibilità con il sistema precedente (backward compatibility)
  - **Maintenance cost (costo di manutenzione).** Costo richiesto per correggere errori sw o hw (bug)
  - **Administration cost (costo di amministrazione**). Costo richiesto per amministrare il sistema

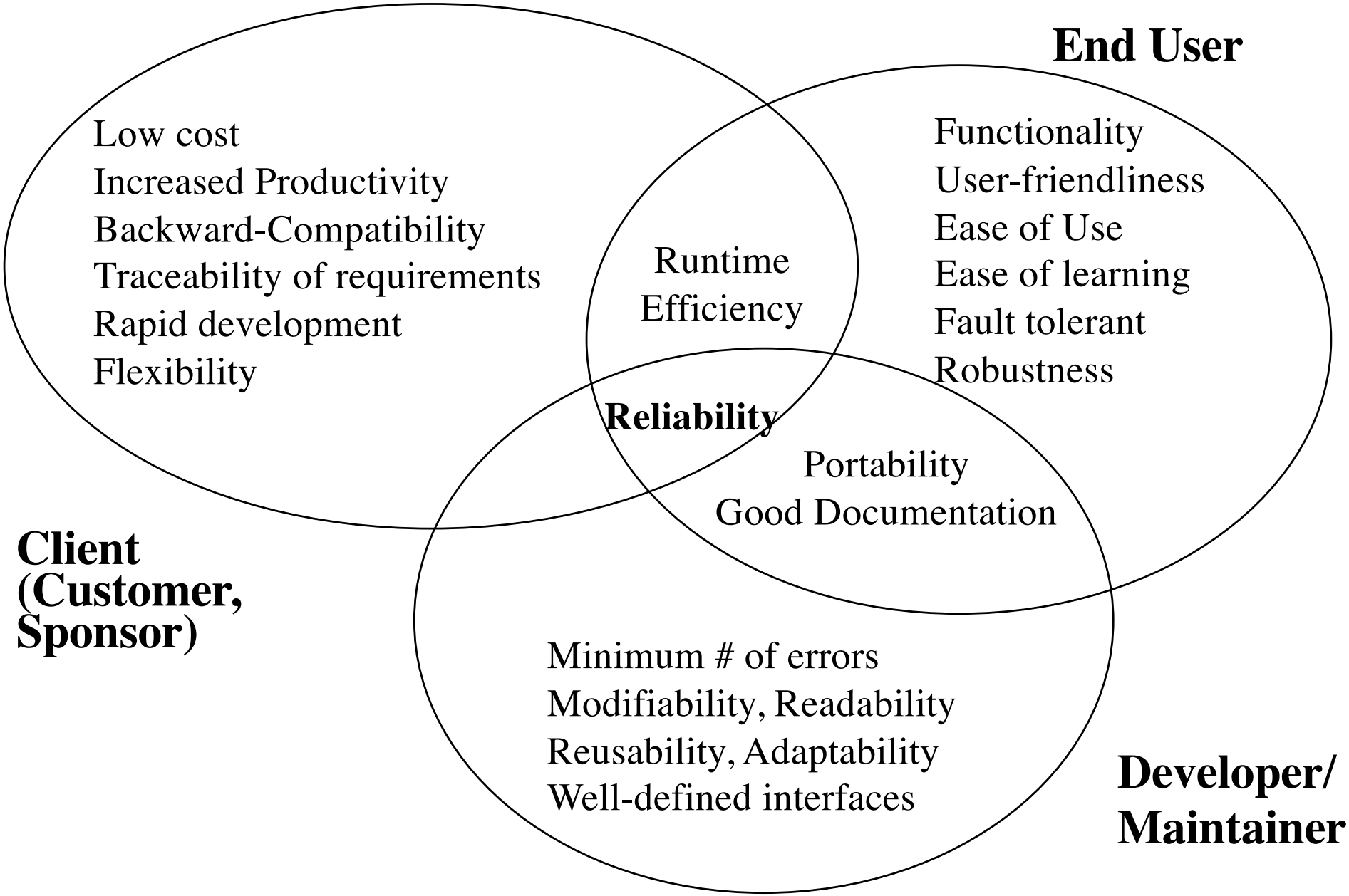
Maintenance criteria

- Determinano quanto deve essere difficile modificare il sistema dopo il suo rilascio
  - **Estensibilità.** 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 differenti domini di applicazione?
  - **Portabilità.** Quanto è facile portare il sistema su differenti piattaforme?
  - **Leggibilità.** Quanto è facile comprendere il sistema dalla lettura del codice?
  - **Tracciabilità dei requisiti.** Quanto è facile mappare il codice nei requisiti specifici?

End User Criteria

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

Relazioni tra Design Goals



Design Trade-offs

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

Design Trade-offs (2)

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 p.m. 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.

Sommario

- System Design
  - **Riduce il gap tra i requisiti e la macchina**
  - **Decompone l'intero sistema in parti che possono essere più facilmente gestite**
- Decomposizione in sottosistemi
  - **Fornisce un insieme di parti scarsamente dipendenti che costituiscono il sistema**
- Architettura Software
  - **Stili architetturali**
- Definizione dei Design Goals
  - **Descrive e assegna le priorità alle caratteristiche di qualità che sono importanti per il sistema**
  - **Definisce il valore del sistema rispetto alle varie opzioni**