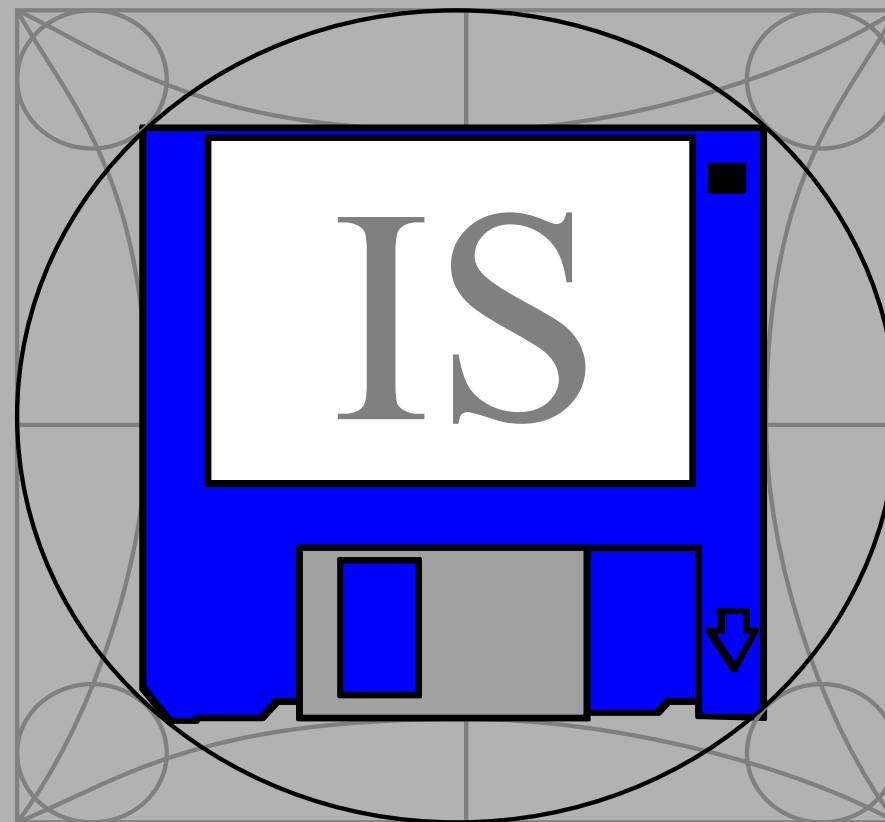


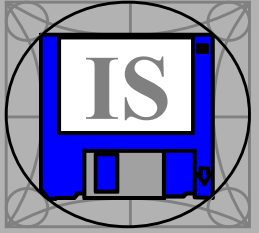
Progettazione *software*

Ingegneria del Software

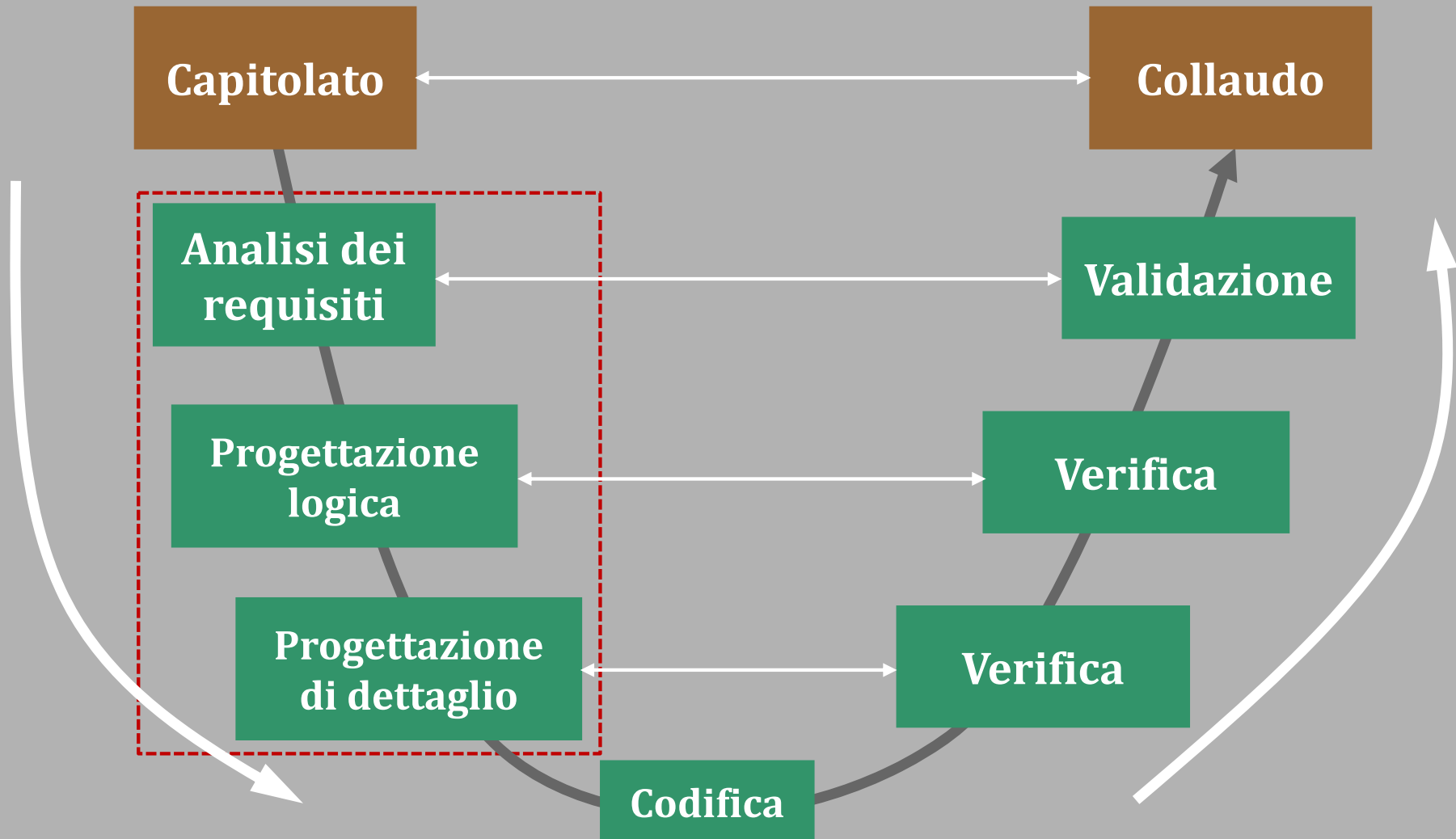
V. Ambriola, G.A. Cignoni
C. Montangero, L. Semini

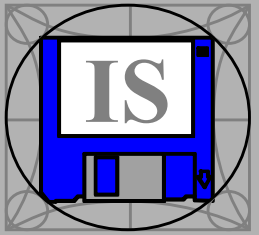
Aggiornamenti di: T. Vardanega (UniPD)



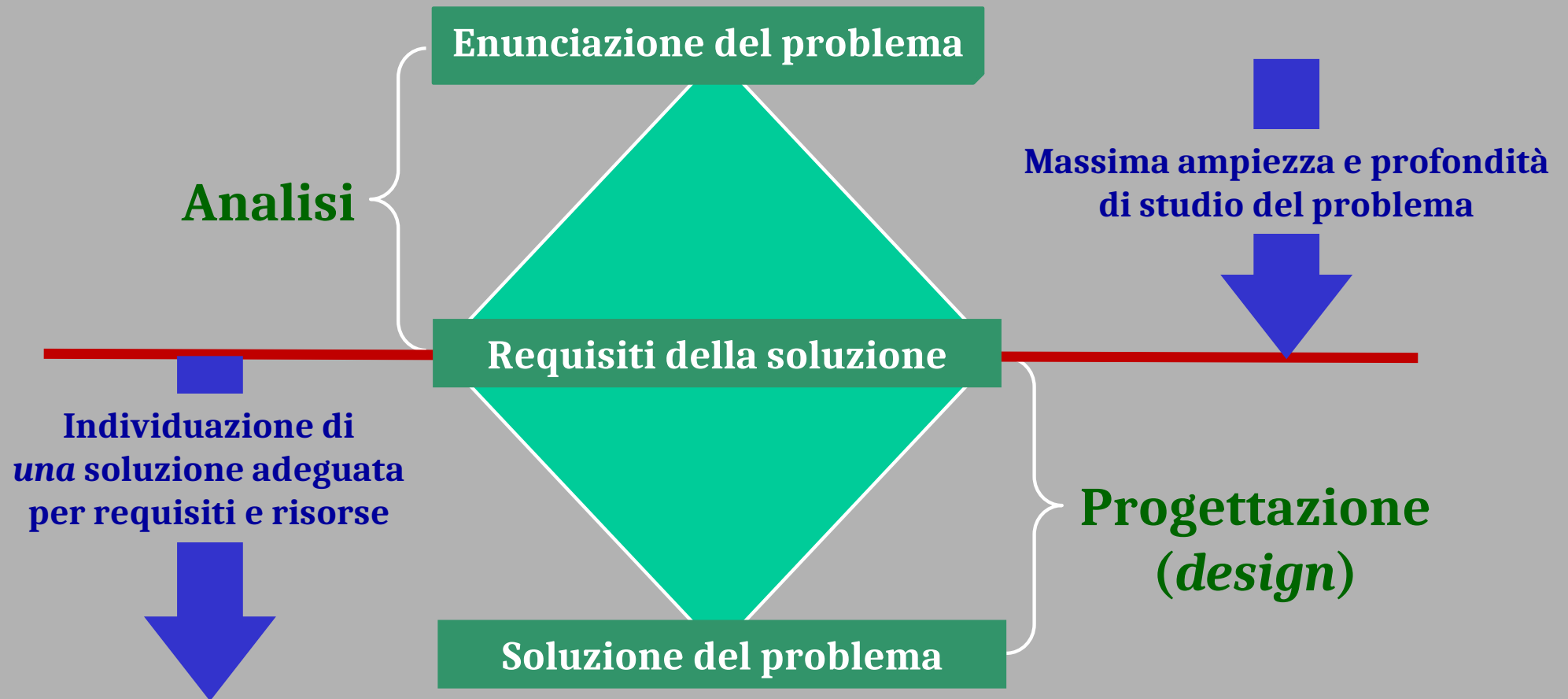


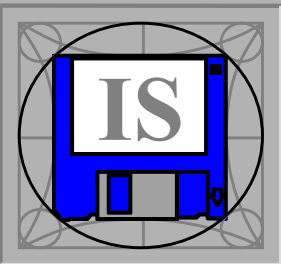
Il modello a V





Da analisi a progettazione – 1/3





Da analisi a progettazione – 2/3

□ L'attività di analisi risponde alla domanda

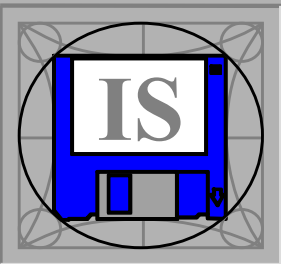
- Qual è il problema, qual è la cosa giusta da fare?
- Comprendere il dominio e discernimento di obiettivi, vincoli, e requisiti tecnici e funzionali



□ L'attività di progettazione risponde alla domanda

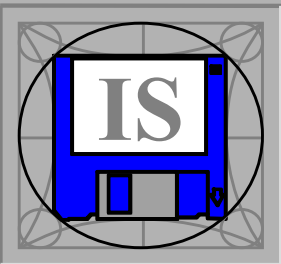
- Come fare la cosa giusta (quella di cui c'è bisogno)?
- Ricercare una soluzione realizzativa soddisfacente per tutti gli *stakeholder*
- Fissare l'**architettura** del prodotto prima di passare alla sua codifica





Da analisi a progettazione – 3/3

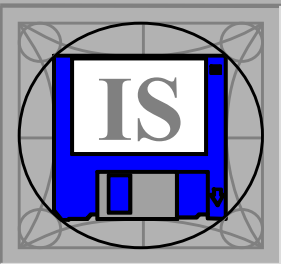
- ❑ Edsger W. Dijkstra (1982) in *“On the role of scientific thought”*
 - *The task of “making a thing satisfying our needs”, as a single responsibility, is split into two parts:*
 1. *Stating the properties of a thing, by virtue of which, it would satisfy our needs, and*
 2. *Making a thing that is guaranteed to have the stated properties*
- ❑ La parte (1) di tale responsabilità è dell'analisi
- ❑ La parte (2) è di progettazione e codifica



Progettare prima di produrre

- ❑ La progettazione (*design*) precede la codifica
 - Perseguendo **correttezza per costruzione**
 - In luogo di **correttezza per correzione**
- ❑ La progettazione (*design*) serve a
 - Dominare la complessità del prodotto (“*divide-et-impera*”)
 - Organizzare e ripartire le responsabilità di realizzazione
 - Produrre in economia (efficienza)
 - Garantire qualità (efficacia)

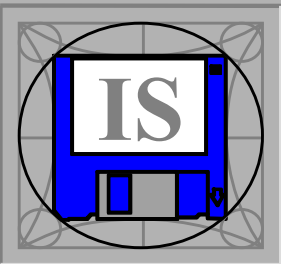




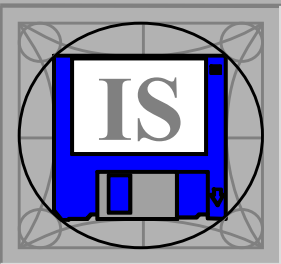
Obiettivi della progettazione – 1/2



- ❑ Soddisfare i requisiti con un sistema di qualità
- ❑ Definendo l'**architettura** (*design*) del prodotto
 - Individuando parti componibili coerenti con i requisiti, e dotate di specifica chiara e coesa
 - Realizzabili con risorse sostenibili e costi contenuti
 - Organizzate in modo da facilitare cambiamenti futuri
- ❑ La scelta di una buona architettura è determinante al successo del progetto



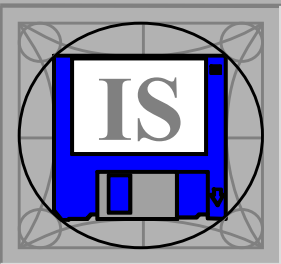
- ❑ ISO/IEC/IEEE 42010:2011 *Systems and software engineering – Architecture description*
- ❑ Decomposizione del sistema in parti componibili
 - Componenti
- ❑ Organizzazione di tali componenti
 - Ruoli, responsabilità, interazioni (chi fa cosa e come)
- ❑ Interfacce necessarie all'interazione tra le componenti tra loro e con l'ambiente di esecuzione
 - Come le componenti collaborano e interagiscono
- ❑ Paradigmi di composizione delle componenti
 - Regole, criteri, limiti, vincoli (anche a fini di manutenibilità)



Obiettivi della progettazione – 2/2

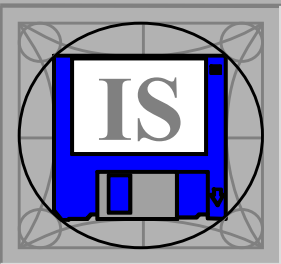


- ❑ **Dominare la complessità del sistema**
- ❑ **Spingendo il *design* in profondità: progettazione di dettaglio**
 - **Suddividere il sistema fino a che ciascuna sua parte abbia bassa complessità individuale**
 - **La codifica di ogni singola parte diventa compito individuale, fattibile, rapido, e verificabile**
 - **Fermare la decomposizione quando il costo di coordinamento tra le parti ne supera il beneficio**



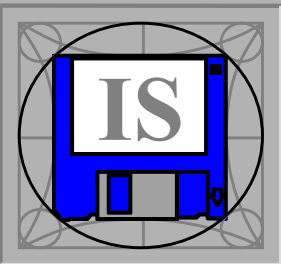
Progettazione di dettaglio – 1/2

- ❑ Le «parti» della progettazione di dettaglio sono chiamate **unità** architeturali
 - Unità funzionali (o di responsabilità) ben definite, realizzabili da un singolo programmatore
- ❑ A una singola unità architeturale possono corrispondere uno o più **moduli** di codice
 - La corrispondenza Unità – Modulo è determinata dalle caratteristiche del linguaggio di programmazione utilizzato per la realizzazione
 - P.es., una classe Java, modulo sintattico del linguaggio, può ben corrispondere a una unità architeturale



Progettazione di dettaglio – 2/2

- ❑ Le unità architettureali realizzano le componenti dell'architettura logica
 - La decomposizione facilita il lavoro di realizzazione
- ❑ **Tracciare l'architettura nel codice** porta due benefici
 - Aiuta la verifica di copertura dei requisiti
 - Guida l'integrazione (dalle parti al tutto)
- ❑ La specifica di ogni unità architettureale deve essere ben documentata affinché la sua programmazione possa svolgersi in modo autonomo e disciplinato
 - Assicurando tracciamento di requisiti da e verso ogni singola unità
- ❑ La responsabilità di realizzare unità ne include la verifica
 - Per questo il SW si misura in termini di ***delivered source lines of code***

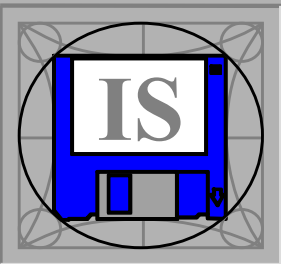


Arte vs. architettura

- ❑ Nel 1915, lo scrittore H.G. Wells (1866-1946), autore di “*The War of the Worlds*” (1898), scrive al collega H. James (1843-1916)

*To you, literature – like painting – is an end
To me, literature – like architecture – is a means, it has a use*

- ❑ L'arte è un fine
- ❑ L'architettura un mezzo per un fine di utilizzo



Approcci di progettazione

❑ Procedimento *top-down*

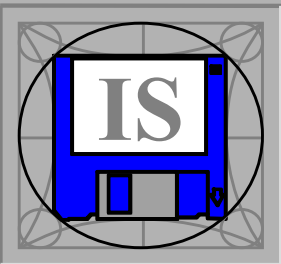
- Studio il sistema immaginando le parti in cui può essere decomposto
- Senza elementi preconcepiuti: esplorazione funzionale

❑ Procedimento *bottom-up*

- Concepisco il sistema ipotizzando le parti che possono comporlo
- Tipico dell'OOP, fortemente orientato a riuso e specializzazione

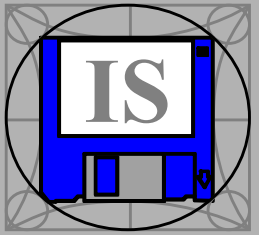
❑ Procedimento *agile*

- Perseguendo consolidamento progressivo (ciclo *feedback* – *refactoring*)
- Nella cattura dei requisiti e nella realizzazione del prodotto
- Per farlo serve una architettura base «accogliente» !

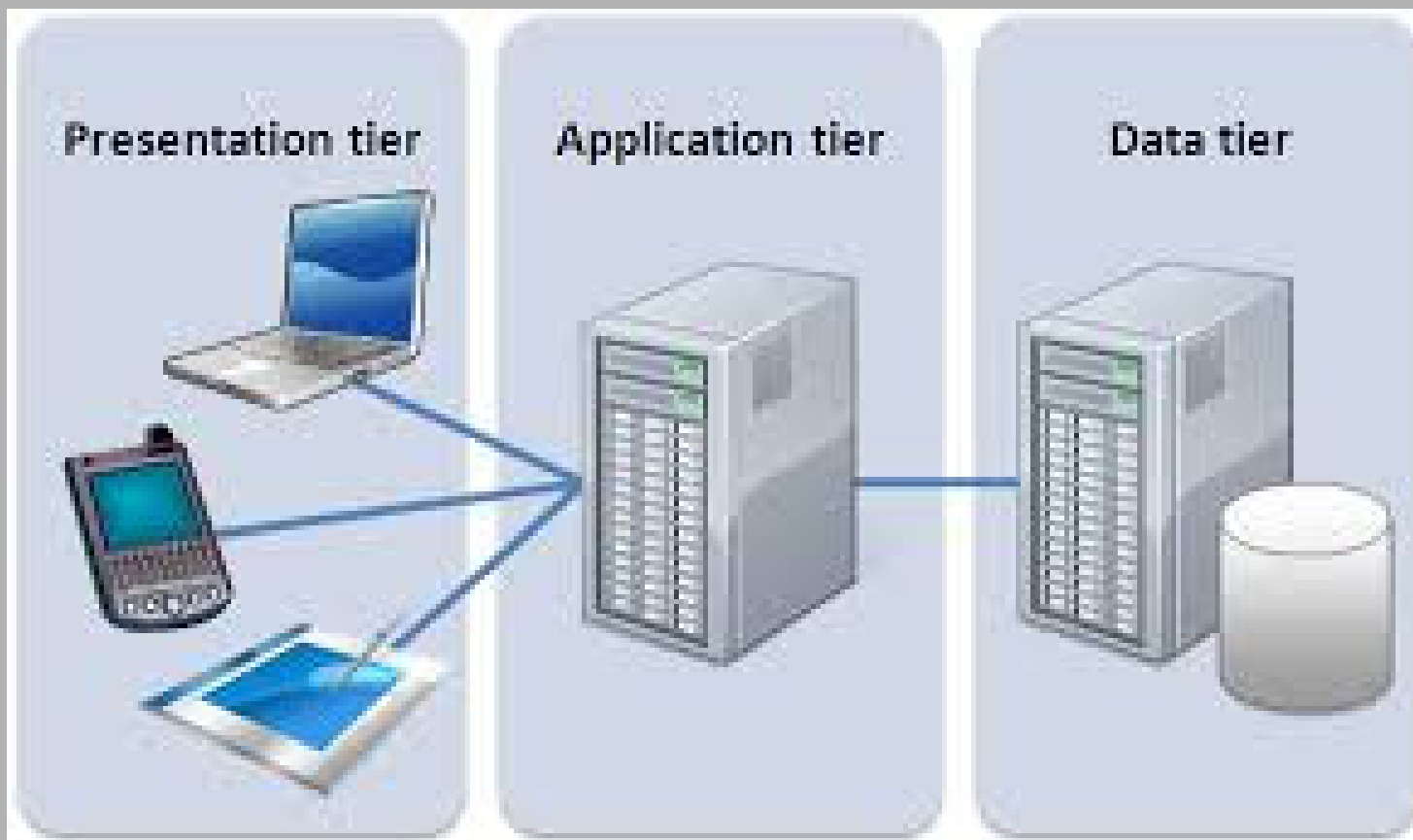


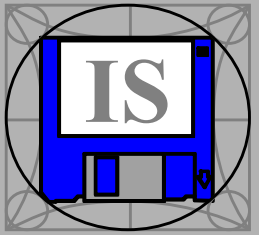
Stili architetturali

- ❑ L'attività di *design* apprende dall'esperienza per auto-migliorarsi
 - Attraverso conoscenza e consolidamento di **stili architetturali**
- ❑ Uno stile architettonico è un aggregato coerente
 - Catalogo di componenti standard (ricorrenti)
 - Regole che vincolano la composizione di tali componenti tra loro
 - Significato semantico di tali composizioni
 - Catalogo di verifiche di conformità su sistemi costruiti in tal modo

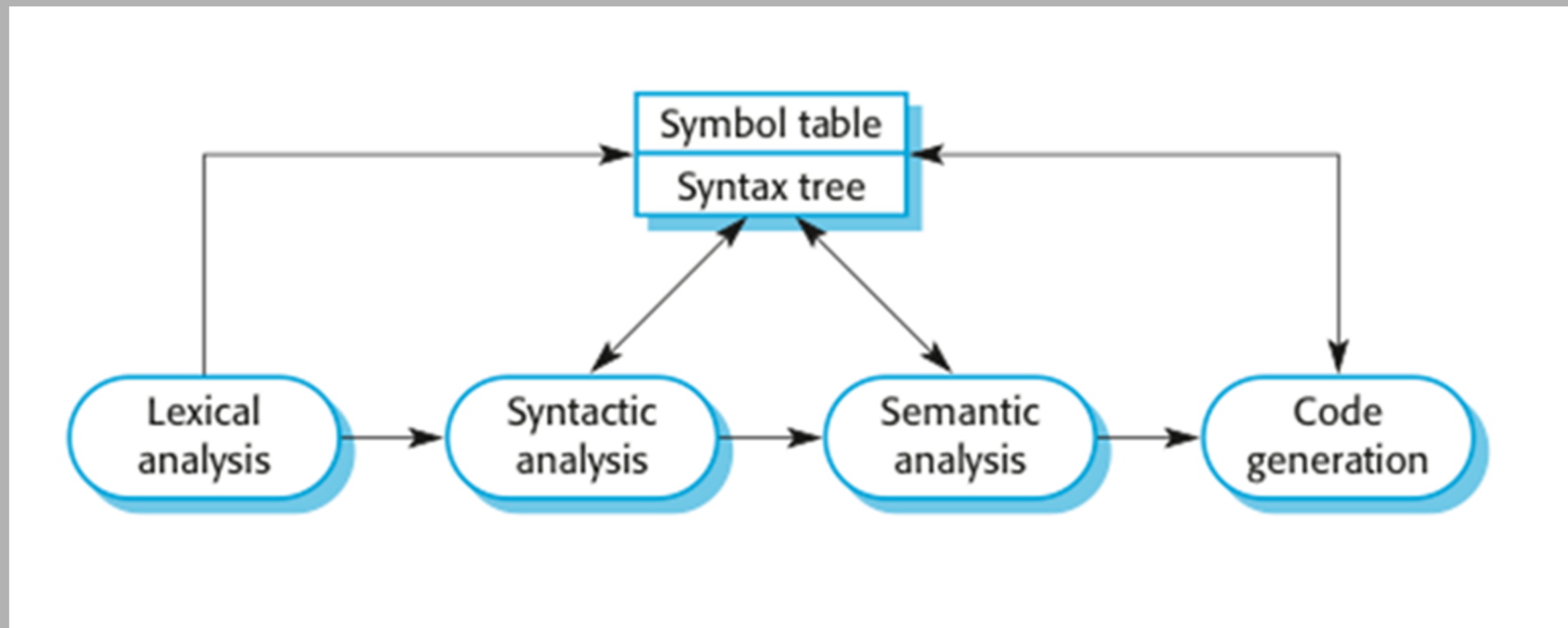


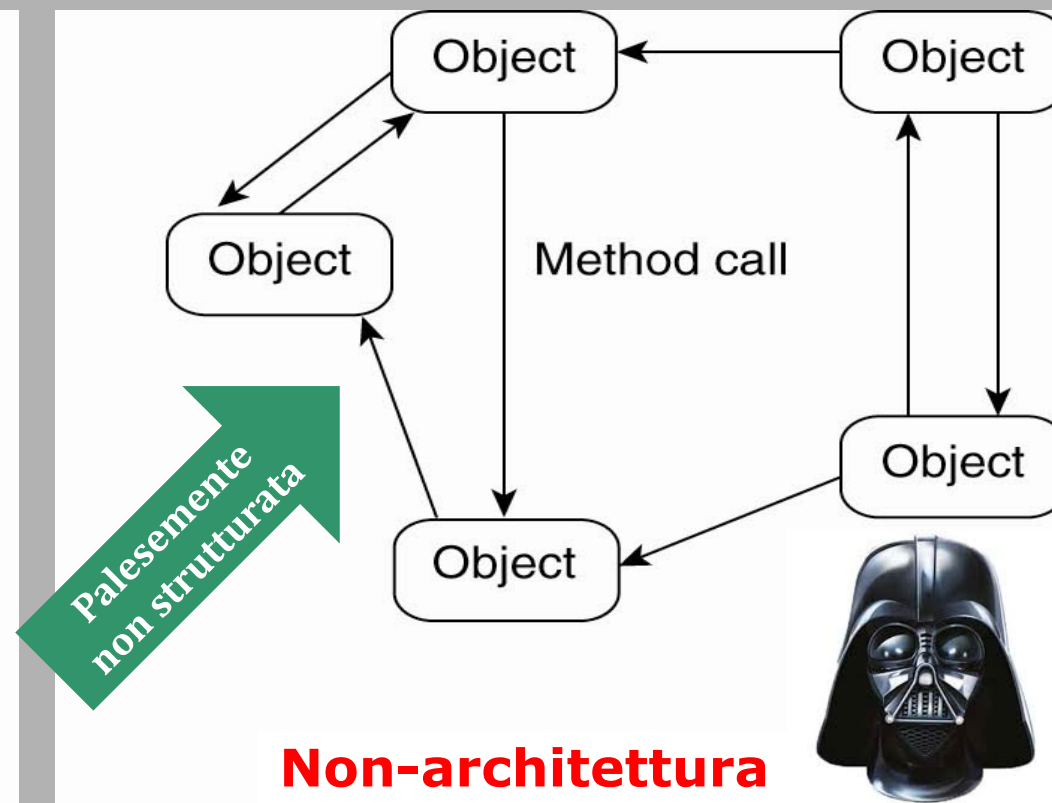
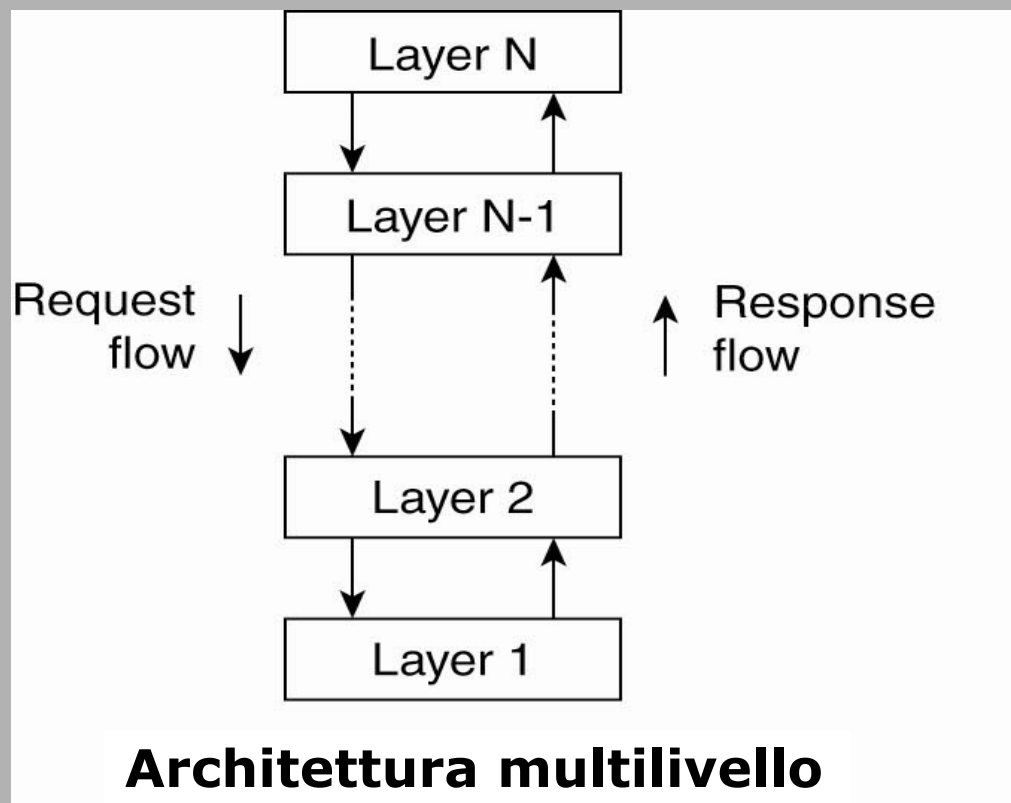
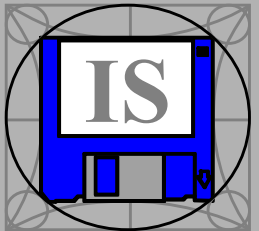
Architettura *Three-Tier*



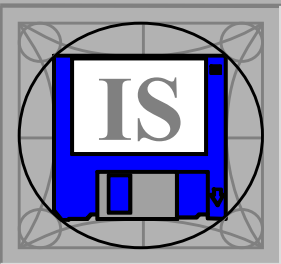


Architettura *Pipe-and-Filter*



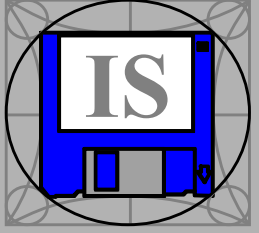


Tratto da: Tanenbaum & Van Steen, *Distributed Systems: Principles and Paradigms*, 2e, (c) 2007 Prentice-Hall, Inc.



Design pattern architeturali

- ❑ **Soluzione progettuale a problema realizzativo ricorrente**
 - Organizzazione architeturale con proprietà provate, ottenibili solo con buona contestualizzazione e coerente implementazione
 - Corrispondente architeturale degli algoritmi
- ❑ **Concetto promosso da C. Alexander, un vero architetto**
 - *The Timeless Way of Building*, Oxford University Press, 1979
- ❑ **Divenuto rilevante nel dominio SWE a partire dalla pubblicazione di “*Design Patterns*” della GoF (1995)**
 - Individuare i DP rilevanti ispira e guida riuso desiderabile
- ❑ **I DP contribuiscono a specifici stili architeturali**



Qualità di una buona architettura – 1/4

☐ Sufficienza

- Capace di soddisfare tutti i requisiti

☐ Comprensibilità

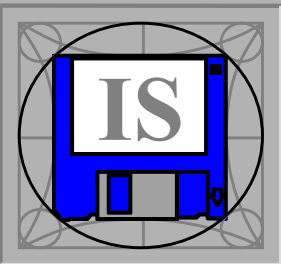
- Capita da tutti gli *stakeholder*

☐ Modularità

- Suddivisa in parti chiare e ben distinte

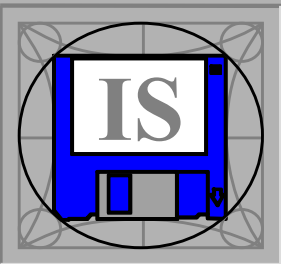
☐ Robustezza

- Capace di sopportare ingressi diversi (giusti, sbagliati, tanti, pochi) dall'utente e dall'ambiente



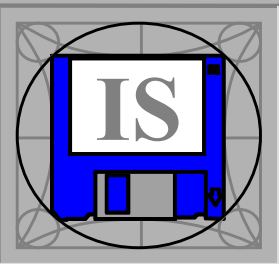
Modularità – 1/2

- ❑ **Minimizzare la dipendenza cattiva tra parti**
 - Determinando ciò che la parte deve esporre ai suoi utenti (l'interfaccia) e ciò che essa deve nascondere (l'implementazione)
 - Esporre metodi `get()` e `set()` come sola modalità di accesso a dati riflette questa preoccupazione
- ❑ **Evitando rischio di effetto domino**
 - Quando la modifica interna di una parte comporta modifiche all'esterno di sé



Modularità – 2/2

- ❑ Secondo D. Parnas, vi sono due vie per modularizzare
 1. Suddividere l'attività nei suoi blocchi logici principali (p.es. come stadi di una *pipeline*)
 2. Suddividere ricercando *information hiding* (aggregando strutture dati con le loro operazioni)
 - ❑ La soluzione 2. riduce i cambiamenti esterni causati da modifiche interne; la 1. non ne è capace!
- D. Parnas, “*On the Criteria to be Used in Decomposing Systems into Modules*”, CACM 15(12):1053-1058 (1972)



Qualità di una buona architettura – 2/4

☐ Flessibilità

- Permette modifiche a costo contenuto al variare dei requisiti

☐ Riutilizzabilità

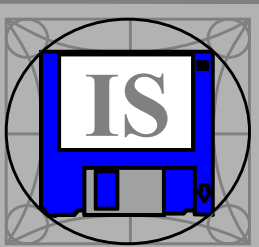
- Le sue parti possono essere impiegate in altre applicazioni

☐ Efficienza

- Nel tempo (CPU), nello spazio (RAM), nelle comunicazioni (banda)

☐ Affidabilità (*reliability*)

- È probabile che svolga bene il suo compito quando utilizzata



Qualità di una buona architettura – 3/4

☐ Disponibilità (*availability*)

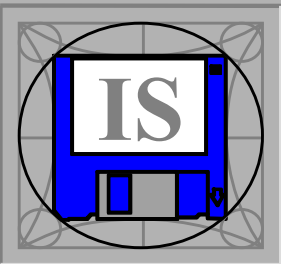
- La sua manutenzione causa poca indisponibilità totale

☐ Sicurezza rispetto a malfunzionamenti (*safety*)

- Abbastanza ridondante da funzionare anche in presenza di guasti



☐ Sicurezza rispetto a intrusioni (*security*)

- I suoi dati e le sue funzioni non sono raggiungibili da intrusi

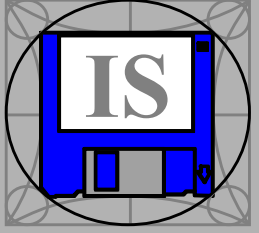


- ❑ Un sistema monolitico va ricostituito per intero a ogni piccolo cambiamento (modifica, aggiunta, rimozione), e poi il vecchio va sostituito dal nuovo
 - Durante la sostituzione e le conseguenti verifiche di buon esito, il sistema diventa indisponibile

► Interruzione di servizio Uniweb ed Esse3+ from Amministrazione Uniweb

Text (1 KB)  

Gentili professoresse, gentili professori,
si comunica che il giorno 7 novembre 2018 dalle ore 13:30 fino alle
ore 17:30, verrà effettuato un aggiornamento degli applicativi in
oggetto con conseguente sospensione del servizio.



Qualità di una buona architettura – 4/4

☐ Semplicità

- Ogni parte contiene solo il necessario e niente di superfluo

☐ Incapsulazione (*information hiding*)

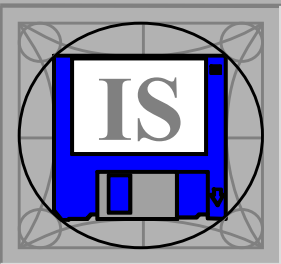
- L'interno delle componenti non è visibile dall'esterno

☐ Coesione

- Ciò che sta insieme concorre agli stessi obiettivi

☐ Basso accoppiamento

- Parti distinte dipendono poco o niente le une dalle altre



❑ William Ockham (1285-1347)



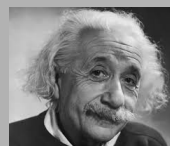
- “*Pluralitas non est ponenda sine necessitate*”
- Rasoio di Occam: gli elementi usati per la soluzione non devono mai essere più di quelli strettamente necessari

❑ Isaac Newton (1643-1727)

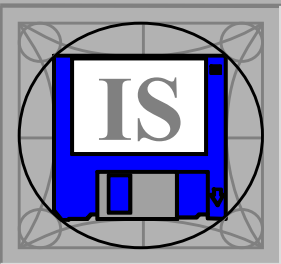


- “*We are to admit no more causes of natural things than such that are both true and sufficient to explain their appearances*”
- Tra due soluzioni equivalenti per risultato, preferire quella meno ricca di sfumature

❑ Albert Einstein (1879-1955)

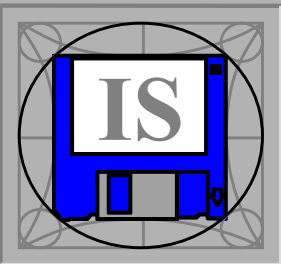


- “*Everything should be made as simple as possible, but not simpler*”



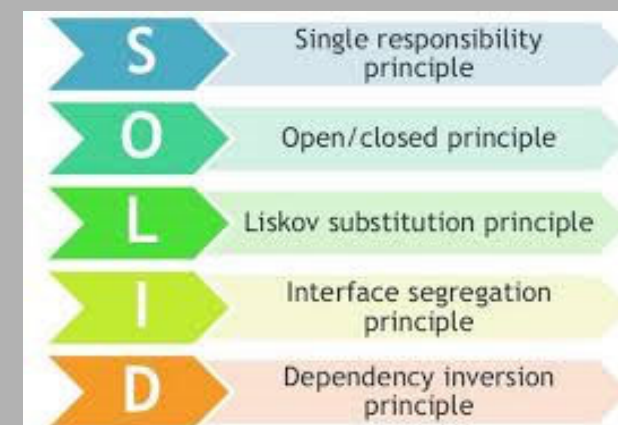
Incapsulazione

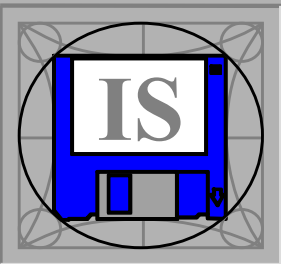
- ❑ **Rendere invisibile all'esterno l'interno delle componenti architetture (cioè la loro implementazione)**
 - Rendendole «*black box*» per l'esterno
- ❑ **E esporre solo l'interfaccia, nascondendo gli algoritmi e le strutture dati usate per realizzarla**
- ❑ **Questa porta importanti benefici**
 - L'esterno non può fare assunzioni sull'interno
 - Fare manutenzione sull'implementazione senza danneggiare gli utenti diventa più facile
 - Minori le dipendenze indotte sull'esterno (quello che da fuori si pensa di sapere sul funzionamento interno), maggiore il potenziale di riuso



Coesione

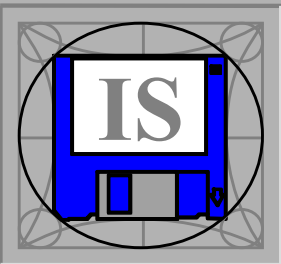
- ❑ Funzionalità “vicine” stanno nella stessa componente
 - Ciò che serve per soddisfare il contratto di interfaccia
- ❑ Va massimizzata per ottenere
 - Maggiore manutenibilità e riusabilità
 - Minore interdipendenza fra componenti
 - Architettura del sistema più comprensibile
- ❑ Ricercare modularità spinge a decomporre sempre di più: la ricerca di coesione limita questa spinta





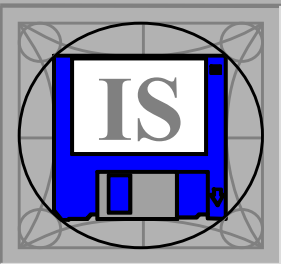
Tipi di coesione

- ❑ **Funzionale:** quando le parti concorrono allo stesso compito
 - Esempio: suddivisione in ruoli (come produttore / consumatore)
- ❑ **Temporale:** quando azioni sono «vicine» tra loro per ordine di esecuzione
 - Esempio: *pipeline*
- ❑ **Informativa:** quando le parti agiscono sulle stesse unità dati
 - Esempio: get() e set() su una struttura dati
- ❑ Secondo D. Parnas, la coesione migliore è quella che produce maggiore incapsulazione (*information hiding*)
 - Quindi quale?



Esempi: SIAGAS

- ❑ **Sistema in uso per la gestione degli stage**
 - Sviluppato come progetto didattico di IS nel 2007
- ❑ **Molte parti del suo codice realizzano funzioni simili: fare calcoli, leggere/scrivere lo stesso dato**
 - **Questo difetto complica moltissimo la manutenzione**
 - Una correzione locale non sana tutte le occorrenze del problema e può causare effetti laterali inattesi su altre
 - **Progettazione non buona, realizzazione pigra**
- ❑ **Quali rimedi?**
 - **Coesione, incapsulazione**



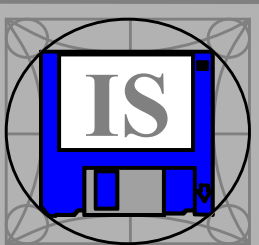
Accoppiamento – 1/2

- ❑ Quando parti diverse hanno dipendenze reciproche cattive
 - L'esterno fa assunzioni sul funzionamento dell'interno (variabili, tipi, indirizzi, ...)
 - L'esterno impone vincoli sull'interno (ordine di azioni, uso di certi dati, formati, valori)
 - Esterno e interno agiscono su *alias* della stessa entità
- ❑ Questo accoppiamento va minimizzato
- ❑ Un sistema è un insieme organizzato che ha bisogno di tutte le sue parti
 - Quindi ha sempre un po' di accoppiamento, che la buona progettazione tiene basso

sistèma = *lat. SYSTÈMA dal gr. SÝSTÈMA*
composto della particella *SYN* con, insieme,
-STÈMA attinente all'inusitato STÈNAI
pres. ISTÈMI| *stare, collocare* (v. *Stare*).

Aggregato di parti, di cui ciascuna può
sistere isolatamente, ma che dipendono
e une dalle altre secondo leggi e regole
isse, e tendono a un medesimo fine; Ag-
gregato di proposizioni su cui si fonda
una dottrina; e anche Dottrina le cui va-
rie parti sono fra loro collegate e seguonsi
in mutua dipendenza; Complesso di parti
inimilmente organizzate e sparse per tutto
il corpo, quale il sistema linfatico, ner-
voso, vascolare ecc.

Deriv. *Sistemàre; Sistemàtico; Sistemazione.*



Accoppiamento – 2/2

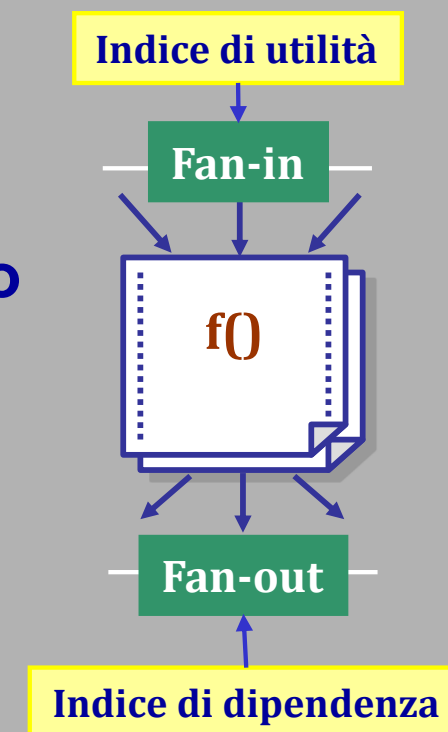
□ Proprietà esterna di componenti

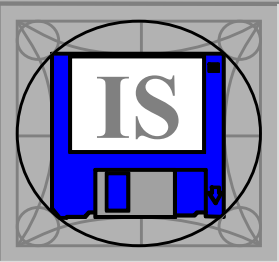
- Il grado U di utilizzo reciproco di M componenti
- $U = M \times M$ è il massimo grado di accoppiamento
- $U = \emptyset$ ne è il minimo

□ Metriche: *fan-in* e *fan-out* strutturale

- SFIN è indice di utilità → massimizzare
- SFOUT è indice di dipendenza → minimizzare

□ Buona progettazione ha componenti con SFIN elevato





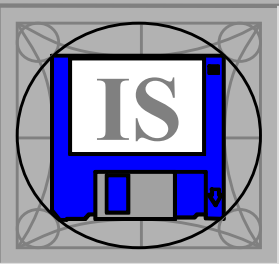
Stati di progresso per SEMAT – 1/2

❑ *Architecture selected*

- Selezione di una architettura tecnicamente adatta al problema: accordo sui criteri di selezione
- Selezione delle tecnologie necessarie
- Decisioni su *buy, build, make*

❑ *Demonstrable*

- Dimostrazione delle principali caratteristiche dell'architettura: gli *stakeholder* concordano
- Decisione sulle principali interfacce e configurazioni di sistema



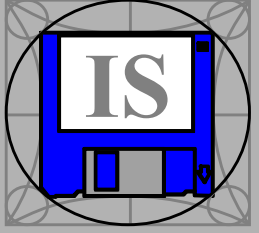
Stati di progresso per SEMAT – 2/2

□ *Usable*

- Il sistema è utilizzabile e ha le caratteristiche desiderate
- Il sistema può essere operato dagli utenti
- Le funzionalità e le prestazioni richieste sono state verificate e validate
- La quantità di difetti residui è accettabile

□ *Ready*

- La documentazione per l'utente è pronta
- Gli *stakeholder* hanno accettato il prodotto e vogliono che diventi operativo



Riferimenti

- ❑ D. Budgen, Software Design, Addison-Wesley
- ❑ C. Alexander, The origins of pattern theory, IEEE Software, settembre/ottobre 1999
- ❑ G. Booch, Object-oriented analysis and design, Addison-Wesley
- ❑ G. Booch, J. Rumbaugh, I. Jacobson, The UML user guide, Addison-Wesley
- ❑ C. Hofmeister, R. Nord, D. Soni, Applied Software Architecture, Addison-Wesley, 2000
- ❑ P. Krutchen, The Rational Unified Process, Addison-Wesley
- ❑ Y.K. Erinc, The SOLID Principles of Object-Oriented Programming Explained in Plain English, freeCodeCamp