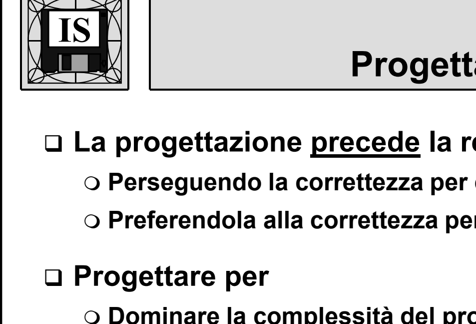


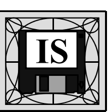
Progettazione *software*

Progettare prima di produrre

- La progettazione <u>precede</u> la realizzazione
  - Perseguendo la correttezza per costruzione
  - Preferendola alla correttezza per correzione

- Progettare per
  - Dominare la complessità del prodotto ("*divide-et-impera*")
  - Organizzare e ripartire le responsabilità di realizzazione
  - Produrre in economia (efficienza)
  - Garantire qualità (efficacia)

Dipartimento di Informatica, Università di Pisa 2/36

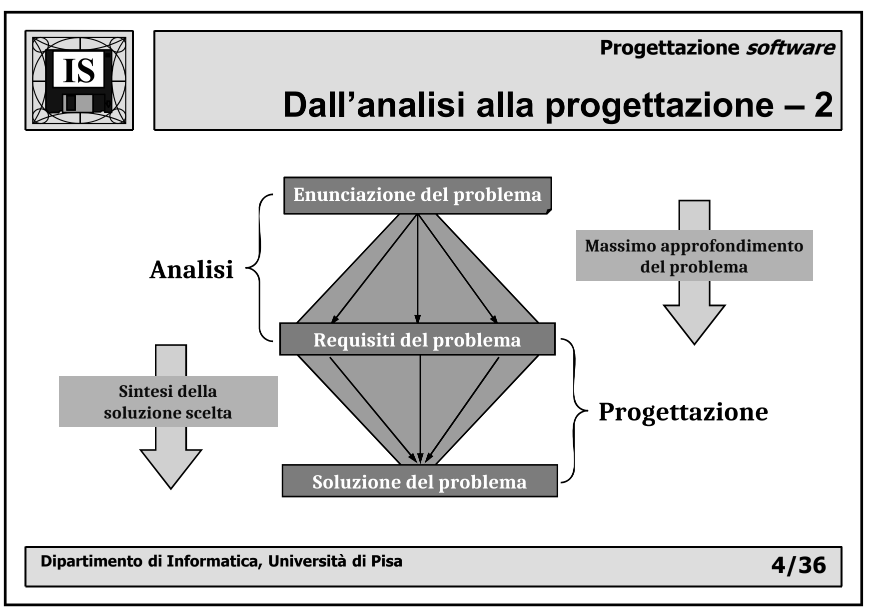


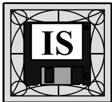
Progettazione *software*

Dall'analisi alla progettazione – 1

- L'analisi risponde alle domande: quale è il problema, quale la cosa giusta da fare?
  - Rispondere richiede comprensione del dominio e discernimento di obiettivi, vincoli e requisiti
  - L'attività di analisi attua un <u>approccio investigativo</u>
- La progettazione risponde alla domanda: come fare la cosa giusta?
  - Ricercando una soluzione soddisfacente per tutti gli *stakeholder*
  - Prima di pensare al codice, descrivere l'architettura del prodotto
  - L'attività di progettazione attua un <u>approccio sintetico</u>

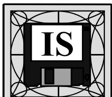
Dipartimento di Informatica, Università di Pisa 3/36





Dall'analisi alla progettazione – 3

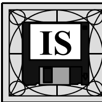
- In "*On the role of scientific thought*" (1982), Edsger W. Dijkstra dice
  - *The task of "making a thing satisfying our needs", as a single responsibility, is split into two parts*
  1. *Stating the properties of a thing, by virtue of which it would satisfy our needs, and*
  2. *Making a thing that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



Obiettivi della progettazione – 1

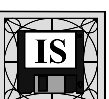
- Soddisfare i requisiti con un sistema di qualità
- Definendo l'architettura logica del prodotto
  - Impiegando parti con specifica chiara e coesa
  - Realizzabili con risorse sostenibili e costi fissati
  - Organizzate in modo da facilitare cambiamenti futuri

- La scelta di una buona architettura facilita il successo
- Ricercando soluzioni architetturali utili al caso e con parti riusabili



Obiettivi della progettazione – 2

- Dominare la complessità del sistema
  - Suddividendo il sistema fino a che ciascuna sua parte abbia complessità trattabile
  - Perché la codifica di ogni singola parte possa essere compito rapido, fattibile e verificabile di un singolo individuo
- Conviene spingere la progettazione nel dettaglio
  - Fermandosi nel momento in cui il costo di coordinamento delle parti supera il beneficio (di semplificazione) della ulteriore suddivisione
  - Più minute le componenti più complessa la loro orchestrazione

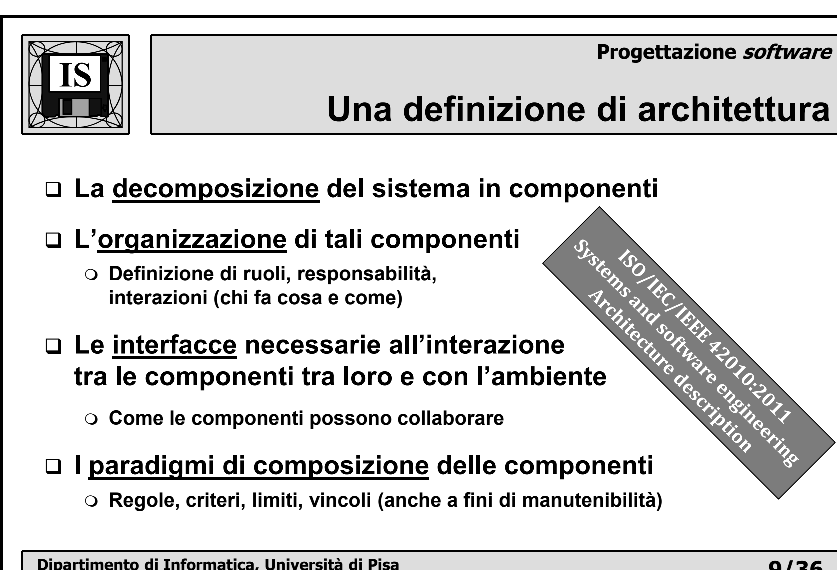


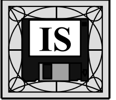
Arte vs. architettura

- Ca. 1915, lo scrittore H.G. Wells (1866-1946), autore, tra l'altro, 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





Progettazione *software*

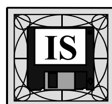
Criteri guida

- Esistono più stili architetturali
  - Aderire a uno stile garantisce coerenza e consistenza progettuale
  - Le scelte architetturali determinano l'organizzazione dell'informazione (di stato) e l'interazione tra le parti
- Molta letteratura sul tema

  P.es.: https://msdn.microsoft.com/en-us/library/ee658098.aspx

> *An architectural style is a named collection of architectural design decisions that*
> - *are applicable in a given development context*
> - *constrain architectural design decisions that are specific to a particular system within that context*
> - *elicit beneficial qualities in each resulting system*

Dipartimento di Informatica, Università di Pisa — 10/36

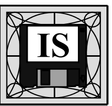


Progettazione *software*

Qualità di una buona architettura – 2

- Flessibilità
  - Permette modifiche a costo contenuto al variare dei requisiti
- Riusabilità
  - Le parti possono essere utilmente impiegate in altre applicazioni
- Efficienza
  - Nel tempo, nello spazio, nelle comunicazioni
- Affidabilità (*reliability*)
  - È probabile che svolga bene il suo compito quando utilizzata

Dipartimento di Informatica, Università di Pisa — 12/36

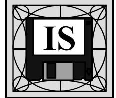


Progettazione *software*

Qualità di una buona architettura – 3

- **Disponibilità (*availability*)**
  - Necessita di poco tempo di indisponibilità totale per manutenzione
  - Esempio: non tutto il sistema deve essere interrotto se qualche sua parte è sotto intervento
- **Sicurezza rispetto a malfunzionamenti (*safety*)**
  - È esente da malfunzionamenti gravi
  - Esempio: il sistema dispone di un sufficiente grado di ridondanza per restare utilmente operativo anche in presenza di guasti locali
- **Sicurezza rispetto a intrusioni (*security*)**
  - I suoi dati e le sue funzioni non sono vulnerabili a intrusioni

Dipartimento di Informatica, Università di Pisa — 13/36

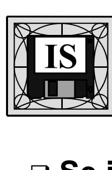


Progettazione *software*

Esempi: modularità

- **Un obiettivo della modularità è minimizzare la dipendenza cattiva tra parti**
  - Capire ciò che la parte deve esporre all'uso degli altri (l'interfaccia) e ciò che invece essa deve nascondere (l'implementazione)
    - I metodi `get()` e `set()` riflettono questa preoccupazione
  - Evitare l'effetto domino: quando una modifica in una parte causa una catena di modifiche all'esterno di sé

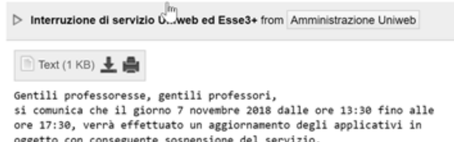
Dipartimento di Informatica, Università di Pisa — 14/36



Progettazione *software*

Esempi: *availability*

- **Se il mio sistema è un monolite, devo ricostituirlo tutto intero ogni volta che ne tocco una parte (per modifica, aggiunta, rimozione), e poi sostituire il vecchio con il nuovo**
  - Durante la sostituzione e le conseguenti verifiche di buon esito, il sistema non è disponibile

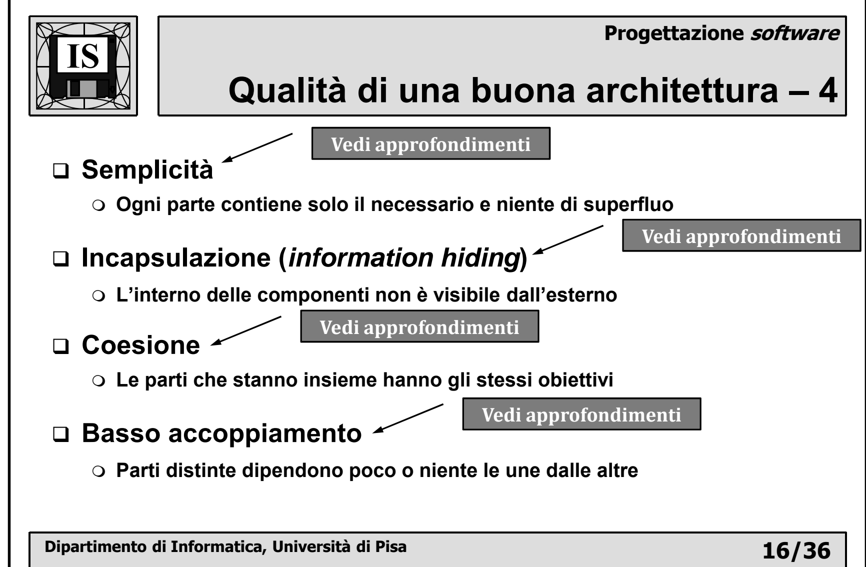


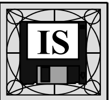
Interruzione di servizio U...web ed Esse3+ from Amministrazione Uniweb

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.

Dipartimento di Informatica, Università di Pisa — 15/36





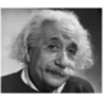
Progettazione *software*

Semplicità

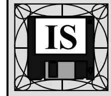
- William Ockham (1285-1347/49)
  - "Pluralitas non est ponenda sine necessitate"
  - Le entità usate [per spiegare un fenomeno] non devono essere moltiplicate senza ragione
  - Principio noto come "il rasoio di Occam"

- 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*"
  - Quando hai due soluzioni equivalenti rispetto ai risultati, scegli la più semplice

- Albert Einstein (1879-1955)
  - "*Everything should be made as simple as possible, but not simpler*"



Dipartimento di Informatica, Università di Pisa 17/36

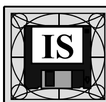


Progettazione *software*

Incapsulazione

- Un buon criterio guida per l'individuazione di moduli (componenti) architetturali
- Tali componenti sono "*black box*" per l'esterno, che ne vede solo l'interfaccia
- La loro specifica nasconde gli algoritmi e le strutture dati usati per l'implementazione
- Importanti benefici
  - L'esterno non può fare assunzioni sull'interno
  - Diventa più facile fare manutenzione sull'implementazione
  - Quante minori sono le dipendenze indotte sull'esterno quanto maggiore è il potenziale di riuso

Dipartimento di Informatica, Università di Pisa 18/36

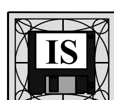


Progettazione *software*

Coesione – 1

- Proprietà interna di singole componenti
  - Funzionalità "vicine" devono stare nella stessa componente
  - La modularità spinge a decomporre il grande in piccolo
  - La ricerca di coesione fornisce un criterio di decomposizione e vi pone anche un limite inferiore
- Va massimizzata per ottenere
  - Maggiore manutenibilità e riusabilità
  - Minore interdipendenza fra componenti
  - Maggiore comprensibilità dell'architettura del sistema

Dipartimento di Informatica, Università di Pisa 19/36

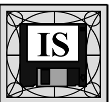


Progettazione *software*

Coesione – 2

- Vi sono svariati tipi di coesione buona
  - Funzionale, quando le parti concorrono al medesimo specifico compito
    - Esempio: suddivisione in ruoli
  - Sequenziale, quando alcune azioni sono «vicine» ad altre per ordine di esecuzione e dunque conviene tenerle insieme
    - Esempio: *pipeline*
  - Informativa, quando le parti agiscono sulla stessa unità di informazione
    - Esempio: nascondere la persistenza di tabelle dietro a oggetti (ORM)
- La migliore è quella che persegue *information hiding*

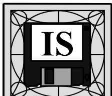
Dipartimento di Informatica, Università di Pisa 20/36



Progettazione *software*

Esempi: SIAGAS

- Un sistema in uso, sviluppato come progetto di IS nel 2007
- Molte parti del suo codice realizzano funzioni analoghe: fare calcoli, leggere/scrivere lo stesso dato
  - Questa caratteristica complica molto la manutenzione
    - Una correzione locale non risolve tutte le occorrenze del problema
    - Un aggiornamento locale può confliggere con parti preesistenti e questo scoraggia l'evoluzione
  - Questo grave difetto nasce da scadente/assente progettazione e da copia-incolla pigro di codice nell'implementazione
- Quali rimedi?
  - Coesione
  - Incapsulazione

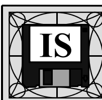


Progettazione *software*

Accoppiamento – 1

- Parti diverse hanno dipendenze reciproche cattive
  - Quando dall'esterno si fanno assunzioni su come certe cose stiano all'interno di altre (variabili, tipi, locazioni, …)
  - Quando dall'esterno si impongono vincoli sull'interno di una parte (per ordine di azioni, uso di certi dati, formati, valori)
  - Quando più parti condividono frammenti delle stesse risorse (p.es. di strutture dati)
- Un sistema è un insieme organizzato che ha bisogno di tutte le sue parti
  - E quindi ha sempre un po' di accoppiamento
  - La buona progettazione lo 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; Aggregato di proposizioni su cui si fonda una dottrina; e anche Dottrina le cui vale parti sono fra loro collegate e seguonsi n mutua dipendenza; Complesso di parti imilmente organizzate e sparse per tutto l corpo, quale il sistema linfatico, nervoso, vascolare ecc.
Deriv. *Sistemàre; Sistemàtico; Sistemazióne.*



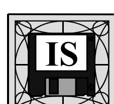
Progettazione *software*

Accoppiamento – 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

Indice di utilità
Fan-in
f()
Fan-out
Indice di dipendenza

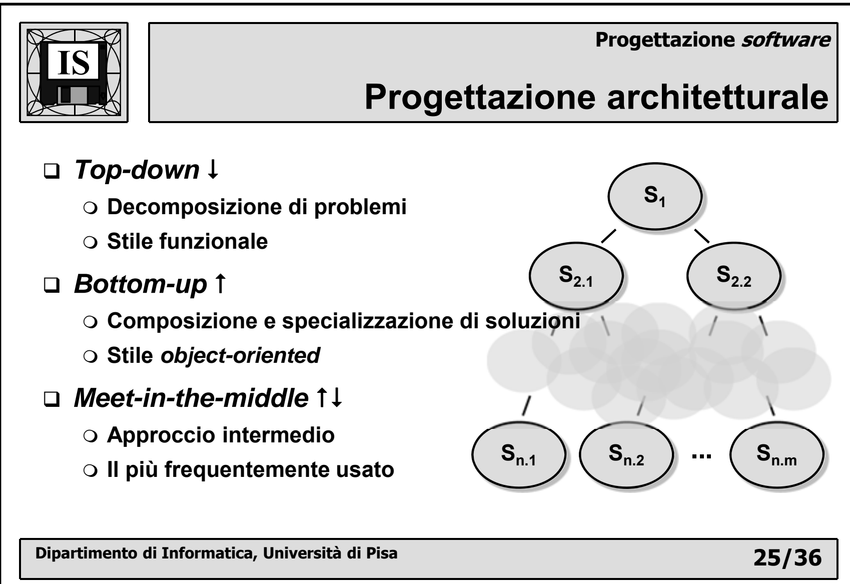
- La buona progettazione produce componenti con SFIN elevato

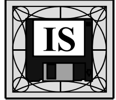


Progettazione *software*

Riuso

- Capitalizzare sottosistemi già esistenti
  - Impiegandoli più volte per più prodotti
  - Con minor costo realizzativo e minor costo di verifica
- Problemi
  - Progettare per riuso è difficile, come lo è anticipare bisogni futuri
  - Progettare con riuso non è immediato, perché deve minimizzare le modifiche alle componenti riusate, perché esse non perdano valore
- Nel breve periodo, il riuso è solo costo
  - Diventa risparmio nel medio termine (quindi è un investimento)



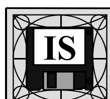


Progettazione *software*

Framework

- Insieme integrato di componenti SW prefabbricate
  - Nel mondo pre-OO erano chiamate librerie
  - Sono *bottom-up* perché sono fatti di codice già sviluppato
  - Sono anche *top-down* se impongono uno stile architetturale
- Preziosi come base facilmente riusabile di diverse applicazioni entro un dato dominio
  - Molti importanti esempi nel mondo J2EE, JS, …
    - Spring (http://www.springsource.org/about)
    - Struts (http://struts.apache.org/)
    - Swing per GUI, ecc.

Dipartimento di Informatica, Università di Pisa 26/36

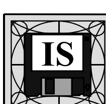


Progettazione *software*

Design pattern architetturali

- Soluzione progettuale a problema ricorrente
  - Suggeriscono una organizzazione architetturale con proprietà note, ottenibili solo con buona istanziazione e coerente implementazione
  - Sono il corrispondente architetturale degli algoritmi
- Concetto promosso da C. Alexander, un vero architetto
  - *The Timeless Way of Building*, Oxford University Press, 1979
- Rilevante nel SW a partire dalla pubblicazione di "*Design Patterns*" della GoF

Dipartimento di Informatica, Università di Pisa 27/36

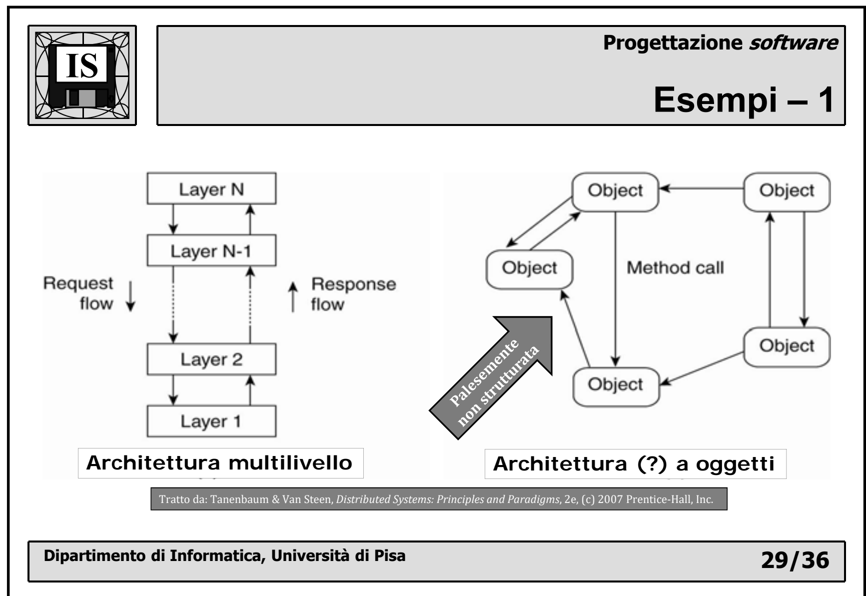


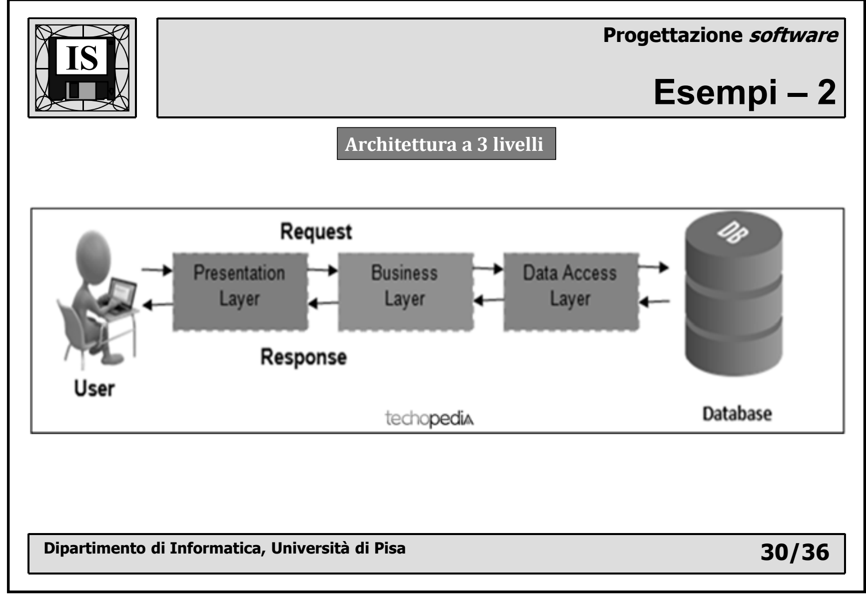
Progettazione *software*

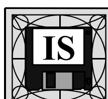
Stili architetturali – 1

- Architettura "*three-tier*"
  - Strato della presentazione (GUI)
  - Strato della logica operativa (*business logic*)
  - Strato dell'organizzazione dei dati (DB)
- Variante multilivello
  - Pile OSI e TCP/IP
- Architettura produttore-consumatore
  - Collaborazione a *pipeline*

Dipartimento di Informatica, Università di Pisa 28/36





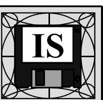


Progettazione *software*

Progettazione di dettaglio: attività

- **Definizione delle unità realizzative (moduli)**
  - **Un carico di lavoro realizzabile dal singolo programmatore**
  - **Corrispondente a una funzionalità (o responsabilità) ben definita**
    - Componente terminale (non decomponibile, una classe) o loro aggregato (un *package*)
    - Tipi, dati, operazioni strettamente correlate tra loro, raccolte in aggregato con chiara identità
- **Le unità possono essere insieme di moduli**
  - **La corrispondenza U-M è determinata dalle caratteristiche del linguaggio di programmazione utilizzato**
  - **Modulo: la più piccola entità progettuale che sia utile rappresentare**

Dipartimento di Informatica, Università di Pisa 31/36

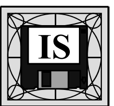


Progettazione *software*

Progettazione di dettaglio: obiettivi

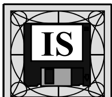
- **Le unità dell'architettura di dettaglio realizzano le componenti individuate nell'architettura logica**
  - **Decomposizione necessaria per organizzare il lavoro di programmazione**
  - **Tracciare le corrispondenze assicura conformità con l'architettura di sistema**
- **La specifica di ogni unità architetturale deve essere documentata**
  - **Perché la programmazione possa procedere in modo certo e disciplinato**
  - **Per assicurare tracciamento di requisiti alle unità**
- **La realizzazione delle unità è associata alla corrispondenti verifiche**
  - **Per ogni unità specificando i casi di prova che possano dimostrare il soddisfacimento dei requisiti a essa associati**

Dipartimento di Informatica, Università di Pisa 32/36



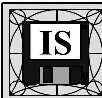
Documentazione

- **IEEE 1016:1998 *Software Design Document***
  - **Introduzione**
    - Come nel documento AR (*Software Requirements Specification*)
  - **Riferimenti normativi e informativi**
  - **Descrizione della decomposizione architetturale**
    - Componenti (visione statica), processi (visione dinamica), dati
  - **Descrizione delle dipendenze (tra componenti, processi, dati)**
  - **Descrizione delle interfacce (tra componenti, processi, dati)**
  - **Descrizione della progettazione di dettaglio**



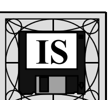
Stati di progresso per SEMAT – 1

- ***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

- ***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**