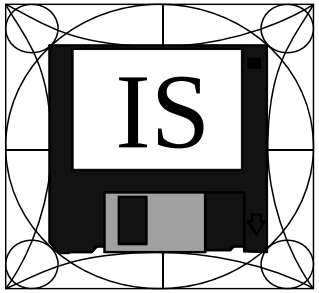
Progettazione *software*



Ingegneria del Software

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

Aggiornamenti di: T. Vardanega (UniPD)

Progettare prima di produrre

- La progettazione precede la codifica
  - Perseguendo la correttezza per costruzione
  - Preferendola alla correttezza per correzione

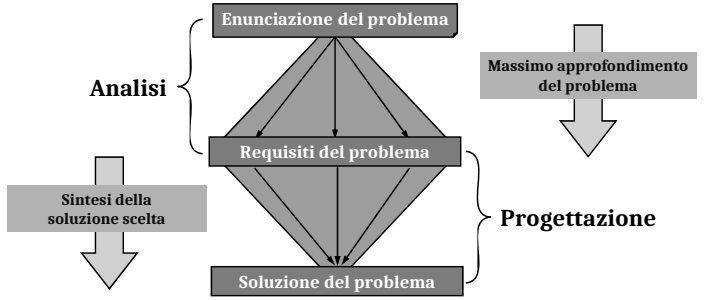
- Progettare per
  - Dominare la complessità del prodotto ("*divide-et-impera*")
  - Organizzare e ripartire le responsabilità di realizzazione
  - Produrre in economia (efficienza)
  - Garantire qualità (efficacia)

Dall'analisi alla progettazione – 1

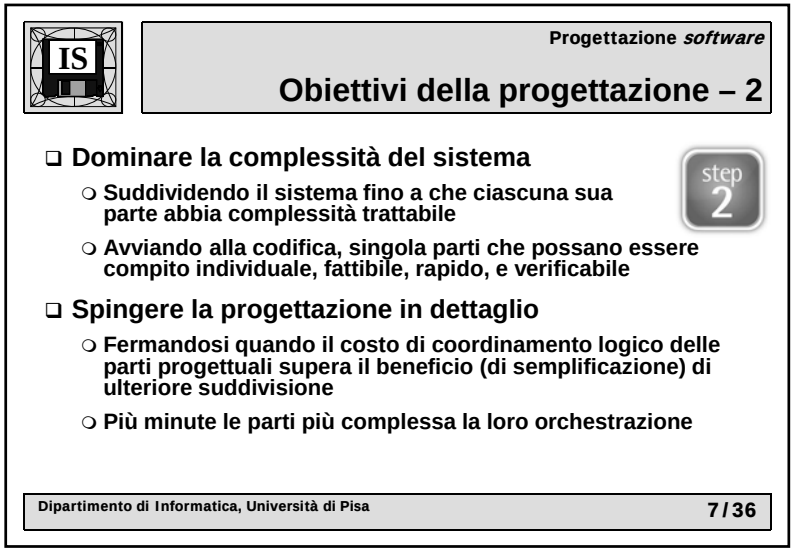
- L'analisi risponde alle domande:
  quale è il problema, quale la cosa giusta da fare?
  - Rispondere richiede comprensione del dominio e discernimento di obiettivi, vincoli e requisiti
  - L'attività di analisi attua un approccio investigativo

- La progettazione risponde alla domanda:
  come fare la cosa giusta?
  - Ricercando una soluzione soddisfacente per tutti gli *stakeholder*
  - Prima di pensare al codice, fissare l'architettura del prodotto
  - L'attività di progettazione attua un approccio sintetico

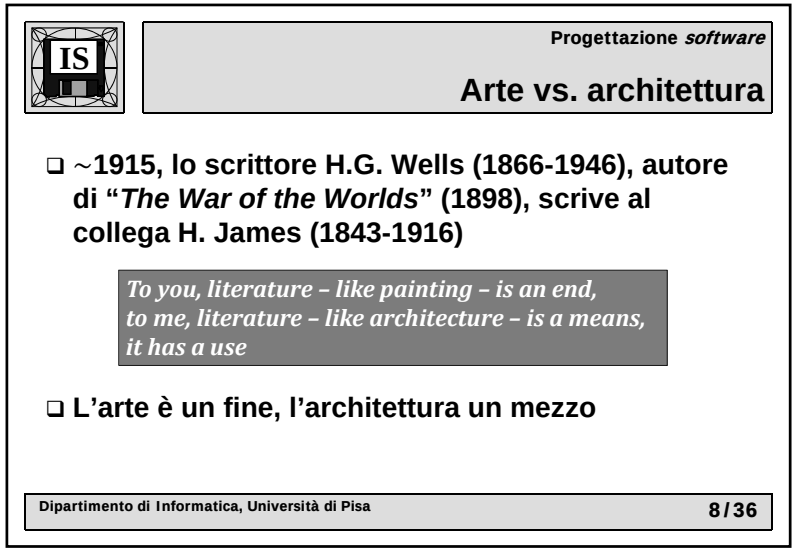
Dall'analisi alla progettazione – 2




IS
Progettazione *software*
Dall'analisi alla progettazione – 3
Dice Edsger W. Dijkstra in:
"*On the role of scientific thought*" (1982)
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
Dipartimento di Informatica, Università di Pisa
5 / 36


IS
Progettazione *software*
Obiettivi della progettazione – 2
Dominare la complessità del sistema
Suddividendo il sistema fino a che ciascuna sua parte abbia complessità trattabile
Avviando alla codifica, singola parti che possano essere compito individuale, fattibile, rapido, e verificabile
Spingere la progettazione in dettaglio
Fermandosi quando il costo di coordinamento logico delle parti progettuali supera il beneficio (di semplificazione) di ulteriore suddivisione
Più minute le parti più complessa la loro orchestrazione
step 2
Dipartimento di Informatica, Università di Pisa
7 / 36


IS
Progettazione *software*
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
Ricercando soluzioni architetturali utili al caso e con parti riusabili
La scelta di una buona architettura è determinante al successo del progetto
step 1
Dipartimento di Informatica, Università di Pisa
6 / 36


IS
Progettazione *software*
Arte vs. architettura
~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
Dipartimento di Informatica, Università di Pisa
8 / 36

Glossario: architettura

- La <u>decomposizione</u> del sistema in componenti
- L'<u>organizzazione</u> di tali componenti
  - Ruoli, responsabilità, interazioni (chi fa cosa e come)
- Le <u>interfacce</u> necessarie all'interazione tra le componenti tra loro e con l'ambiente
  - Collaborazione tra le componenti
- I <u>paradigmi di composizione</u> delle componenti
  - Regole, criteri, limiti, vincoli (anche a fini di manutenibilità)

ISO/IEC/IEEE 42010:2011 Systems and software engineering Architecture description

Criteri guida

- Esistono più stili architetturali
  - Aderire a uno stile garantisce coerenza progettuale (*design*)
  - Determina l'organizzazione dello stato logico e fisico del prodotto e come esso rifletta l'interazione delle sue componenti
- Molta letteratura sul tema

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*

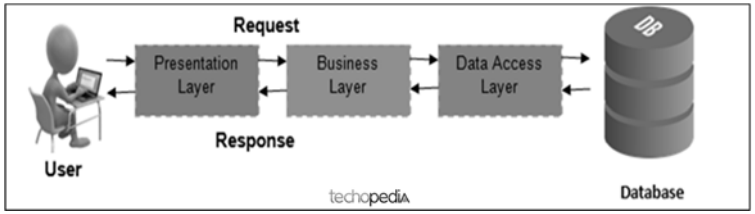
P.es.: https://msdn.microsoft.com/en-us/library/ee658098.aspx

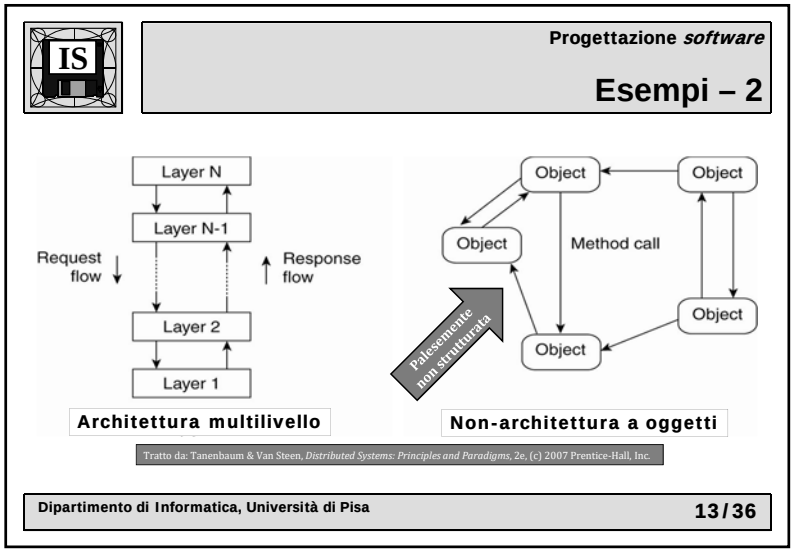
Stili architetturali – 1

- 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*

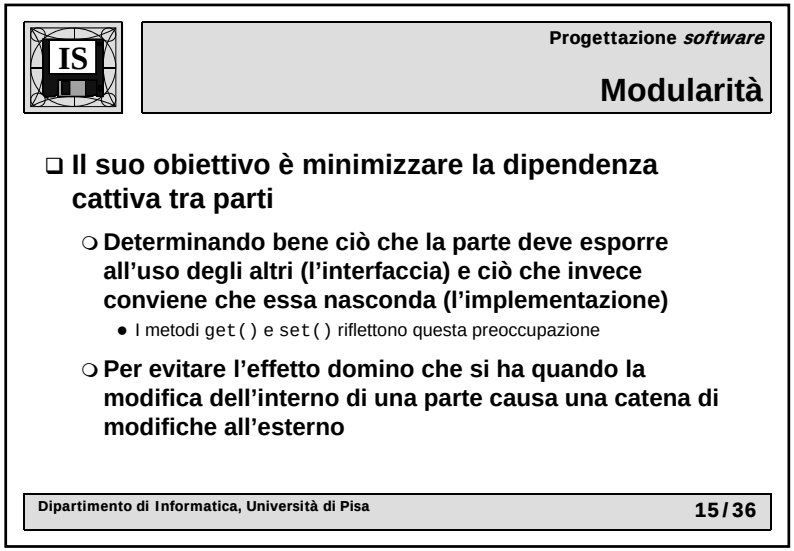
Esempi – 1

Architettura a 3 livelli




Progettazione *software*
Esempi – 2
Layer N
Layer N-1
Request flow
Response flow
Layer 2
Layer 1
Architettura multilivello
Object
Object
Object
Method call
Object
Object
Palesemente non strutturata
Non-architettura a oggetti
Tratto da: Tanenbaum & Van Steen, *Distributed Systems: Principles and Paradigms*, 2e, (c) 2007 Prentice-Hall, Inc.
Dipartimento di Informatica, Università di Pisa
13 / 36


Progettazione *software*
Qualità di una buona architettura – 1
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
Due vie per "modularizzare":
1. Suddividere l'attività nei suoi blocchi logici principali (p.es. gli stadi di una *pipeline*)
2. Perseguire *information hiding*
La seconda via riduce i cambiamenti esterni causati da modifiche interne. La prima non ne è capace!
D. Parnas, "*On the Criteria to be Used in Decomposing Systems into Modules*", CACM 15(12):1053-1058 (1972)
Dipartimento di Informatica, Università di Pisa
14 / 36


Progettazione *software*
Modularità
Il suo obiettivo è minimizzare la dipendenza cattiva tra parti
Determinando bene ciò che la parte deve esporre all'uso degli altri (l'interfaccia) e ciò che invece conviene che essa nasconda (l'implementazione)
I metodi `get()` e `set()` riflettono questa preoccupazione
Per evitare l'effetto domino che si ha quando la modifica dell'interno di una parte causa una catena di modifiche all'esterno
Dipartimento di Informatica, Università di Pisa
15 / 36

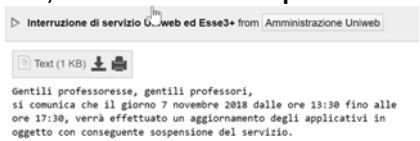

Progettazione *software*
Qualità di una buona architettura – 2
Flessibilità
Permette modifiche a costo contenuto al variare dei requisiti
Riusabilità
Le sue parti possono essere impiegate in altre applicazioni
Efficienza
Nel tempo, nello spazio, nelle comunicazioni
Affidabilità (*reliability*)
È probabile che svolga bene il suo compito quando utilizzata
Dipartimento di Informatica, Università di Pisa
16 / 36

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 manutenzione
- **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**

Availability

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



Qualità di una buona architettura – 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**
  - **Le parti che stanno insieme hanno gli stessi obiettivi**
- **Basso accoppiamento**
  - **Parti distinte dipendono poco o niente le une dalle altre**

Proprietà rafforzative della modularità

Semplicità

- **William Ockham (1285-1347/49)**
  - **"*Pluralitas non est ponenda sine necessitate*"**
  - **Principio del «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, scegli la più semplice**

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

Incapsulazione

- Criterio chiave per l'individuazione di componenti architetturali
  - Da rendere "*black box*" per l'esterno, che ne vede solo l'interfaccia
  - La loro specifica deve nascondere 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
  - Minori le dipendenze indotte sull'esterno, maggiore il potenziale di riuso

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 funge da criterio di decomposizione ma anche da limite inferiore
- Va massimizzata per ottenere
  - Maggiore manutenibilità e riusabilità
  - Minore interdipendenza fra componenti
  - Maggiore comprensibilità dell'architettura del sistema

Coesione – 2

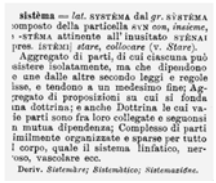
- Vi sono svariati tipi di coesione buona
  - Funzionale: quando le parti concorrono allo stesso compito
    - Esempio: suddivisione in ruoli (come produttore / consumatore)
  - Sequenziale: quando alcune azioni sono «vicine» ad altre per ordine di esecuzione
    - Esempio: *pipeline*
  - Informativa: quando le parti agiscono sulla stessa unità di informazione
    - Esempio: get() e set() su una struttura dati
- La migliore tra esse è quella che produce maggiore incapsulazione

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
  - Grave mancanza di coesione, che nasce da scadente progettazione e copia-incolla pigro nella codifica
  - Questa difetto complica molto la manutenzione
    - Una correzione locale non sana tutte le occorrenze del problema
    - Un aggiornamento locale può confliggere con parti preesistenti e questo scoraggia l'evoluzione
- Quali rimedi?
  - Coesione
  - Incapsulazione

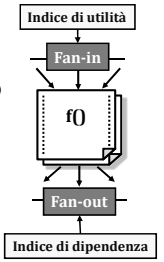
Accoppiamento – 1

- Quando parti diverse hanno dipendenze reciproche cattive
  - Facendo assunzioni dall'esterno su come le cose siano fatte all'interno (variabili, tipi, locazioni, …)
  - Imponendo dall'esterno vincoli sull'interno (ordine di azioni, uso di certi dati, formati, valori)
  - Agendo su *alias* della stessa risorsa logica
- 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.* SYSTĒMA dal *gr.* SÝSTĒMA composto della particella SYN *con, insieme,* e -STĒMA attinente all'inusitato STÉNAI pres. ISTĒMI *stare, collocare* (*v. Stare*).
Aggregato di parti, di cui ciascuna può esistere isolatamente, ma che dipendono o une dalle altre secondo leggi e regole sse, e tendono a un medesimo fine; Aggregato di proposizioni su cui si fonda na dottrina; e anche Dottrina le cui varie parti sono fra loro collegate e organizzate n mutua dipendenza; Complesso di parti imilmente organizzate e sparse per tutto i corpo, quale il sistema linfatico, nervoso, vascolare ecc.
Deriv. *Sistemàtico; Sistematizzare.*

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
- La buona progettazione produce componenti con SFIN elevato

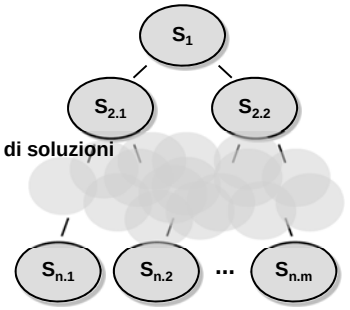


Riuso

- 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 agevole, perché non deve modificare le componenti riusate, affinché esse non perdano valore
- Nel breve periodo, il riuso è solo costo: diventa risparmio nel medio termine
  - Quindi è un investimento

Progettazione architetturale

- *Top-down* ↓
  - Decomposizione di problemi
  - Stile funzionale
- *Bottom-up* ↑
  - Composizione e specializzazione di soluzioni
  - Stile *object-oriented*
- *Meet-in-the-middle* ↑↓
  - Approccio intermedio
  - Il più frequentemente usato



Framework

- Insieme integrato di componenti SW prefabbricate
  - Erano librerie nel mondo pre-OO
  - Sono *bottom-up* perché sono fatti di codice già sviluppato
  - Sono anche *top-down* se impongono uno stile architetturale
- Base facilmente riusabile di diverse applicazioni entro un dato dominio
  - Molti importanti esempi nel mondo *enterprise*
    - Spring (http://www.springsource.org/about)
    - Struts (http://struts.apache.org/)
    - Swing, Qt, …

Design pattern architetturali

- Soluzione progettuale a problema ricorrente
  - Organizzazione architetturale con proprietà note, ottenibili solo con buona istanziazione e coerente implementazione
  - 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

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

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

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**