

Ingegneria del software

Silviu Filote

January 31, 2022

Contents

1	Introduzione	6
1.1	Che cosa é software engineering ?	7
1.2	Phases in the development of software	8
1.2.1	Requirements Engineering	8
1.2.2	Design	9
1.2.3	Implementation	9
1.2.4	Testing	9
1.2.5	Maintenance	9
1.2.6	Global distribution of effort	10
1.3	Software Engineering Ethics - Principles	11
1.4	Recent developments	11
2	Software Engineering Management	12
2.1	Contents of project plan	12
2.2	Project control	14
2.3	Managing	15
3	The Software Life Cycle	16
3.1	Simple life cycle model	16
3.2	Waterfall Model	17
3.3	V-Model	18
3.4	Activity versus phase	18
3.5	Agile Methods	19
3.5.1	Prototyping	20
3.5.2	Incremental development	22
3.5.3	RAD: Rapid Application Development and DSDM . .	23
3.5.4	XP – Programmazione estrema	26
3.5.5	SCRUM	29

3.6	RUP - Rational Unified Process	30
3.7	Differenze	33
3.8	Model driven architecture/engeneering	34
3.9	Maintenance or Evolution	35
3.10	Process modeling	36
3.10.1	Petri-net view of the review process - vedi slides	37
3.10.2	Avvertenze sulla modellazione dei processi	37
4	Configuration Management	38
4.1	Tasks and responsabilities	38
4.2	Configuration Control Board	39
4.3	Tool support for configuration management	40
4.4	Functionalities of SCM tools (SCM: Software Configuration Manager)	41
4.5	Models of configurations	41
4.6	Evolution of SCM tools	42
4.7	Configuration Management Plan	42
4.8	Github - example	43
5	People Management, People Organization	44
5.1	Mintzberg's coordination mechanisms	44
5.2	Reddin's management styles	45
5.3	Team Organization	46
5.3.1	Hierarchical organization	46
5.3.2	Matrix organization	47
5.3.3	SWAT team	47
5.3.4	Agile team	47
5.3.5	Open Source Software Development	49
6	Managing Software Quality	51
6.1	How to measure “complexity”?	51
6.2	Framework measurement	53
6.3	Representation condition	54
6.4	More on measures	54
6.5	A taxonomy of quality attributes	55
6.5.1	Quality attributes (McCall) - Model quality	55
6.5.2	Taxonomy of quality attributes (ISO 9126) - Model quality	56
6.6	ISO 9001	58
6.7	SQA and IEEE standard 730	58
6.8	Capability Maturity Model (CMM)	59

7 Requirements Engineering	62
7.1 Requirement (IEEE610)	63
7.2 Natural language specs are dangerous	63
7.3 market-driven vs customer-driven	63
7.4 Requirements engineering, main steps	63
7.4.1 Elicitation	64
7.5 Requirements Engineering Paradigms	65
7.6 Elicitation techniques	65
7.6.1 Asking	66
7.6.2 Task Analysis	66
7.6.3 Scenario-Based Analysis	67
7.7 Types of links between customer and developer	68
7.8 Structuring a set of requirements - Goals and Viewpoints . . .	68
7.9 Prioritizing requirements (MoSCoW)	69
7.10 Prioritizing requirements (Kano model)	69
7.11 Crowdsourcing	69
7.12 COTS selection	70
7.13 Requirements documentation and management	70
7.14 Requirements specification techniques	71
7.15 Types of requirements	72
7.16 Validation of requirements	72
8 Modeling	73
8.1 Specifica dei requisiti	73
8.2 Qualità delle specifiche	74
8.3 Linguaggi per la specifica	74
8.4 Diversi formalismi	75
8.5 Choosing a modeling notation	76
8.6 System Modeling Techniques	76
8.6.1 Entity-Relationship Modeling (ER)	77
8.6.2 Finite state machines	77
8.6.3 Data flow diagrams	78
8.6.4 CRC: Class, Responsibility, Collaborators	78
8.7 Intermezzo: what is an object?	78
8.8 Objects and attributes	79
8.9 Relations between objects	79
8.10 Unified Modeling Language (UML)	80
8.10.1 Viste in UML	80

9 Software Architecture	82
9.1 Architecture in the life cycle	83
9.2 Why Is Architecture Important?	83
9.3 Software architectur, definitions	84
9.4 Other points of view	84
9.5 Architectural Structures	85
9.6 Software Architecture & Quality	85
9.7 Attribute-Driven Design (ADD)	85
9.8 Generalized model	86
9.9 Global workflow in architecture design	86
9.10 Design issues, options and decisions	87
9.11 Software design in UML	89
9.12 IEEE model for architectural descriptions	89
9.13 Kruchten's 4+1 view model	90
9.14 Architectural views from Bass et al	92
9.14.1 Module views	92
9.14.2 Component and connector views	93
9.14.3 Allocation views	93
9.15 Architectural styles	93
9.16 Using schemas: pattern, framework, idioms	93
9.17 Components and Connectors	94
9.17.1 Types of components	94
9.18 Types of connectors	94
9.19 Framework for describing architectural styles	95
9.20 Main-program-with-subroutines style	95
9.21 Abstract-data-type style	96
9.22 Pipes-and-filters style	96
9.23 Repository style	97
9.24 Layered style	97
9.25 Model-View-Controller (MVC) style	98
10 Software Design	99
10.1 Design principles	99
10.1.1 Abstraction	99
10.1.2 Modularity	101
10.1.3 Information hiding	103
10.1.4 Complexity	103
10.2 Design methods	112
10.2.1 Functional decomposition	112
10.2.2 Data flow design	112
10.3 OOAD methods	113

10.4 Booch' method	114
10.5 Fusion method	114
10.6 RUP method	115
10.7 Classification of design methods	115
10.8 Design pattern	115
10.9 Antipatterns	115
10.10Design documentation	116
11 Software testing	117
11.1 How then to proceed?	117
11.1.1 Exhaustive testing	118
11.2 Classification of testing techniques	118
12 Terminologia base del testing	118
12.1 Error, fault, failure	119
12.2 V&V and testing	119
12.3 Testing process	120
12.4 Testing models	120
12.4.1 Demonstration	120
12.4.2 Destruction	120
12.5 Testing and the life cycle	121
12.6 Test-Driven Development (TDD)	122
12.7 Test Stages	122
12.8 Verification and validation planning and documentation	122
13 Software Maintenance	124
13.1 Kinds of maintenance activities	124
13.2 Shift in type of maintenance over time	125
13.3 Major causes of maintenance problems	125
13.4 Laws of Software Evolution	125
13.5 Reverse engineering	125
13.6 Reengineering (renovation)	126
13.7 Migration	126
13.8 Refactoring	127

1 Introduzione

Le applicazioni al giorno d'oggi:

- tendono ad essere molto grandi;
- è il prodotto generato dalla collaborazione e sviluppo di team;
- possono durare diversi anni;

Il termine "software engineering" è stato coniato nel 1968/1969 dalla NATO
⇒ è possibile sviluppare software così come si costruiscono i ponti, partendo da una solida base teorica costituita da **diverse tecniche di design?**

Errori all'interno dei software:

- Conseguenze: di ambito finanziario;
- Soluzione: **qualità e produttività**

Costo del software:

- I costi di sviluppo non sono gli unici costi imputabili;
- ma esistono delle spese per mantenere il software una volta che è stato deploioato;

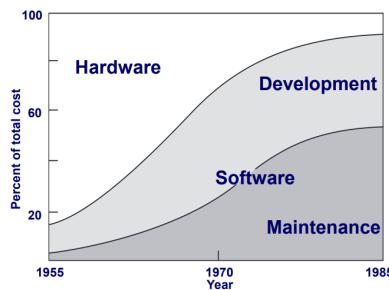


Figure 1: Relative distribution of hardware/software costs.

1.1 Che cosa è software engineering ?

- Una disciplina che si occupa della costruzione di sistemi software così grandi da essere costruito da un team o da team di ingegneri;
- Multi-person construction of multi-version software;
- Una disciplina il cui scopo è la produzione di software:
 - privo di errori;
 - consegnato in tempo;
 - rispettando un budget;
 - e che soddisfi le esigenze dell’utente;

Inoltre, il software deve essere facile da modificare quando l’utente vuole fare un cambiamento.

- È dove puoi progettare grandi cose ed essere creativo
- **Defizione (IEEE):** L’ingegneria del software è l’applicazione di un sistematico, disciplinato, quantificabile approccio allo:
 - sviluppo;
 - funzionamento;
 - e manutenzione del software;

Temi centrali:

- Problematiche allo sviluppo di grandi software;
- Lo sviluppo deve essere efficiente;
 - La complessità rappresenta un problema;
 - Il software evolve e i problemi che ne derivano;
- SE implica diverse discipline;

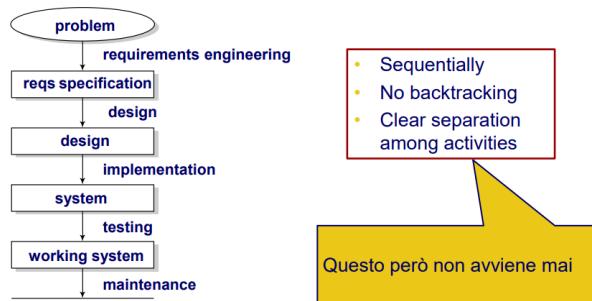
Programming versus software engineering

Small project	Large to huge project
You	Teams
Build what you want	Build what they want
One product	Family of products
Few sequential changes	Many parallel changes
Short-lived	Long-lived
Cheap	Costly
Small consequences	Large consequences

← Programming → Software engineering

Figure 2: differences

1.2 Phases in the development of software



1.2.1 Requirements Engineering

Fornisce una descrizione del sistema desiderato:

- funzionalità;
- possibili estensioni;
- produzione della documentazione necessaria;
- performance requirements;

Tale attività include:

- uno studio di fattibilità del sistema desiderato;
- **documentazione prodotta:** requirements specification (specificazione dei requisiti)

1.2.2 Design

- prime decisioni di progettazione ⇒ scelta della software architecture;
- decomposizione in parts/componenets:
 - che funzionalitá devono avere ?
 - e che interfacce ?
- **enfasi sul cosa piuttosto che sul come;**
- **documentazione prodotta:** specification

1.2.3 Implementation

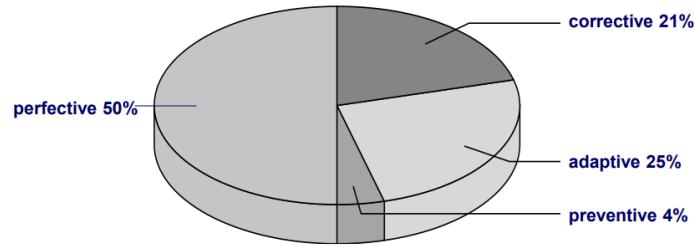
- concentrarsi sui singoli componenti → tramite moduli / classi;
- **Obiettivo:** realizzazione di un pezzo di sfotware: funzionante, robusto, flessibile, ..

1.2.4 Testing

- does the software do what it is supposed to do?
- are we building the right system? (validation)
- are we building the system right? (verification)
- start testing activities in phase 1, on day 1;

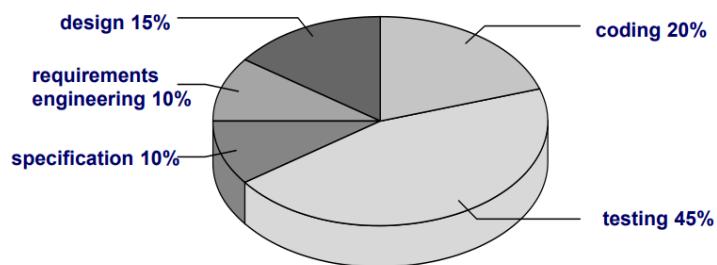
1.2.5 Maintenance

- correcting errors found after the software has been delivered;
- adapting the software to changing requirements, changing environments;
- Tipi:
 - **corrective maintenance:** correzione errori;
 - **adaptive maintenance:** adattarsi ai cambiamenti di ambiente (sia hardware che software);
 - **perfective maintenance:** adattarsi al cambiamento requisiti dell’utente;
 - **preventive maintenance:** aumentare la futura manutenibilità del sistema;



1.2.6 Global distribution of effort

- rule of thumb / (40–20–40 rule): distribution of effort
 - 20% coding;
 - 40% requirements engineering, specification and design;
 - 40% testing;
- When we consider the total cost of a software system over its lifetime maintenance alone consumes 50–75%



1.3 Software Engineering Ethics - Principles

- Tipo di comportamento da mantenere;
- Agire coerentemente con l'interesse pubblico;
- Agire in un modo che sia nel migliore interesse del cliente e del datore di lavoro
- Garantire che i prodotti siano ad un alto livello;
- Mantenere l'integrità in giudizio professionale;
- Managers shall promote an ethical approach;
- Sii onesto e collaborativo con i colleghi;

1.4 Recent developments

- Rise of agile methods;
- Shift from producing software to using software;
- Success of Open Source Software;
- Software development becomes more heterogeneous;

2 Software Engineering Management

I progetti di sviluppo software spesso coinvolgono più persone per un periodo di tempo prolungato. Tali progetti devono essere attentamente pianificati e controllati.

La pianificazione del progetto è il primo passo da intraprendere. Parte di questo processo di pianificazione consiste nell'identificare le caratteristiche del progetto e il loro impatto sul processo di sviluppo.

Il risultato della fase di progettazione è stabilito in un documento, **project plan** che mira a fornire un quadro chiaro del progetto sia ai clienti e il team di sviluppo.

2.1 Contents of project plan

1. **Introduction:** nell'introduzione al progetto vengono specificati

- La storia;
- Il background;
- Gli obiettivi da raggiungere;
- I risultati raggiunti;
- I nomi dei responsabili;
- Un riassunto;

2. **Process model:** viene scelto un modello di sviluppo da seguire per la realizzazione del prodotto finito.

3. **Organization of the project:** il tipo di relazione che si instaura con i diversi partecipanti del progetto.

All'interno del team di progetto si possono identificare diversi ruoli: project manager, tester, programmatore, analista, ecc...

Bisogna delineare chiaramente questi ruoli e identificare le responsabilità di ciascuno di essi. Se ci sono lacune nella conoscenza richiesta per svolgere uno di questi ruoli, la formazione e l'istruzione necessarie per colmare queste lacune devono essere identificati.

4. **Standards, guidelines, procedures** Occorre quindi una forte disciplina di lavoro, in cui ogni persona coinvolta segue gli standard, le linee guida e le procedure concordate. Oltre ad essere riportate su carta, molte di queste possono essere supportate o imposte da strumenti.

5. **Management activities:** Ad esempio, la direzione dovrà presentare rapporti regolari sullo stato e lo stato di avanzamento del progetto. Dovrà anche seguire alcuni priorità nel bilanciamento di requisiti, tempistiche e costi.

6. **Risks:** devono essere identificati il prima possibile

- l'hardware potrebbe non essere consegnato in tempo;
- mancanza di persone qualificate;
- mancanza di informazioni essenziali;
- ecc....

Quanto più incerti sono i vari aspetti del progetto, tanto maggiori sono i rischi.

7. **Staffing:** il progetto richiederá diverse persone con diffenti skills. In questa voce vengono indicate le competenze e le categorie delle persone coinvolte.

8. **Methods and techniques:** Sotto questa voce vengono resi noti, i metodi, le tecniche applicate durante

- Requirements engineering;
- Design;
- Implementation;
- Testing;

Si produrrá una quantitá ingente di informazioni e si dovrá scegliere come documentarla. Inoltre tutte decisioni prese nelle 4 elencate sopra dovranno risiedere in questa sezione

- The necessary test environment and test equipment is described;
- The order in which components are integrated and tested has to be stated explicitly.
- The procedures to be followed during acceptance testing, i.e. the testing under user supervision, have to be given

9. **Quality assurance:** Quale organizzazione e procedure saranno utilizzate per assicurare che il software in fase di sviluppo soddisfi i requisiti di qualità dichiarati

10. **Work packages:** I progetti più grandi devono essere suddivisi in attività, pezzi gestibili che possono essere assegnati ai singoli membri del team.

Tutte le attività devono essere elencate.

11. **Resources:** hardware, software (tools), e persone.

12. **Budget and schedule:**

- come viene allocato il budget rispetto alle attività dichiarate;
- come vengono tracciate le spese e stimati i costi;
- rispettare tempi dichiarati;

13. **Changes:** i cambiamenti sono inevitabili e devono essere garantiti, inoltre possono introdurre costi che devono essere stimati.

Tutte attività da intraprendere:

- documentazione chiara;
- codice pulito;
- ect ...

14. **Delivery:** Le procedure da seguire per la consegna del sistema al cliente.

Il **project plan** mira a fornire un quadro chiaro del progetto sia ai clienti e il gruppo di progetto. Se gli obiettivi non sono chiari, non saranno raggiunti. Nonostante un'attenta pianificazione, le sorprese si presenteranno comunque durante il progetto. Tuttavia, un'attenta pianificazione precoce porta a meno sorprese e rende meno vulnerabili a queste sorprese.

2.2 Project control

- **Time**, both the number of man-months and the schedule;
- **Information**, mostly the documentation;
- **Organization**, people and team aspects;
- **Quality**;
- **Money**;

2.3 Managing

- Managing Information
 - Documentation: Technical documentation, Current state of projects, Changes agree upon, ...
 - Agile projects: meno attenzione all'esplicito documentazione, più sulla conoscenza tacita detenuta da persone;
- Managing people
 - Managing expectations;
 - Building a team;
 - Coordination of work;
- Managing quality
 - Quality must be guaranteed;
 - Quality requirements often conflict with each other;
 - Requires frequent interaction with stakeholders;
- Managing cost
 - Which factors influence cost?
 - What influences productivity?
 - Relation between cost and schedule;

3 The Software Life Cycle

3.1 Simple life cycle model

- **document driven** also known as planning driven or heavyweight;
 - planning driven → il processo è pianificato in fasi ben circoscritte;
 - heavyweight → because of the emphasis placed on the process;
 - al contrario di **lightweight agile** → not ‘waste’ time on expensive planning and design activities early on, but to deliver something valuable to the customer as quickly as possible
- milestones are reached if the appropriate documentation is delivered (requirements specification, design specification, program, test document);
- much planning upfront, often heavy contracts are signed;
- problems:
 - feedback is not taken into account;
 - maintenance does not imply evolution;

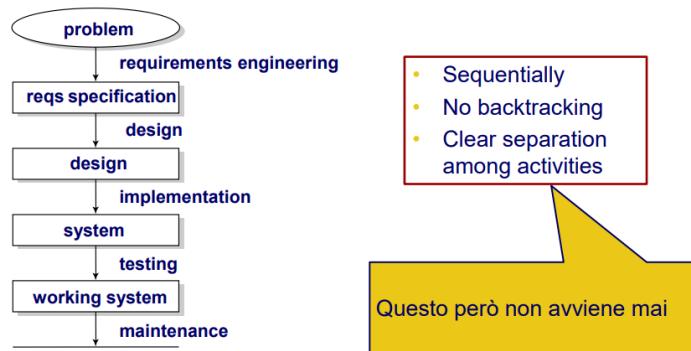


Figure 3: Simple life cycle model

3.2 Waterfall Model

- V&V stands for Verification and Validation.
 - **Verification:** asks if the system meets its requirements (are we building the system right) and thus tries to assess the correctness of the transition to the next phase;
 - **Validation:** asks if the system meets the user's requirements (are we building the right system);
- emphasis on a careful analysis before the system is actually built;
- identify the user's requirements as early as possible;
 - it is difficult in practice, if not impossible
- includes iteration and feedback;
- In each phase we have to compare the results obtained against those that are required;
- The waterfall model of software development is unrealistic
 - Ci sono ampie prove quantitative che il classico modello document-driven ha molte carenze;
 - la rigida sequenza di fasi sostenute dal modello a cascata non vengono effettivamente rispettate;

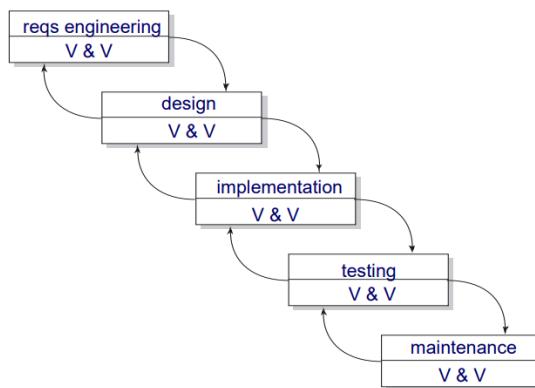


Figure 4: Waterfall Model

3.3 V-Model

- Another waterfall model;
- Different types of testing are related to the various development phases;
- Includes iteration and feedback;
- user requirements are fixed as early as possible;
- problems
 - too rigid;
 - developers cannot move between various abstraction levels;

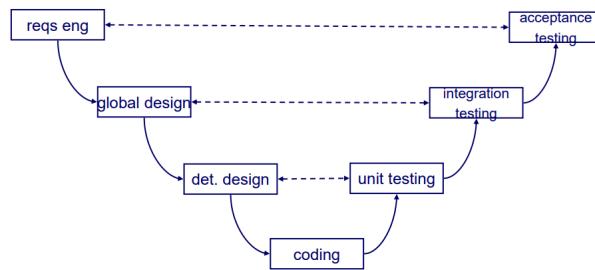


Figure 5: V-Model

3.4 Activity versus phase

Activity \ Phase	Design	Implementation	Integration testing	Acceptance testing
Integration testing	4.7	43.4	26.1	25.8
Implementation (& unit testing) CODING	6.9	70.3	15.9	6.9
Design	49.2	34.1	10.3	6.4

Design during testing?

3.5 Agile Methods

True agile methods view the world as fundamentally chaotic. They assume change is inevitable.

Their focus is to deliver value to the customer as quickly as possible, and not bother about extensive plans and processes that won't be followed anyway.

The essence of agile methods is laid down in the Manifesto for Agile Software Development, published in 2001 by a group of well-known pioneers in this area.

The key values of the agile movement are:

- **Individuals and interactions** over processes and tools;

They emphasize the human element in software development. Team spirit is considered very important. Team relationships are close.

Often, an agile team occupies one big room. The users are on site as well. Agile methods have short communication cycles between developers and users, and among developers.

- **Working software** over comprehensive documentation;

Why spend time on something that will soon be outdated? Rather, agile methods rely on the tacit knowledge of the people involved. If you have a question, ask one of your friends. Do not struggle with a large pile of paper that quite likely will not provide the answer anyway.

- **Customer collaboration** over contract negotiation;

- **Responding to change** over following a plan;

Agile methods involve the users in every step taken. The development cycles are small and incremental. The series of development cycles is not extensively planned in advance, but the new situation is reviewed at the end of each cycle. This includes some, but not too much, planning for the next cycle.

- Agile methods involve the users in every step taken;

Lightweight (agile) approaches:

- prototyping;
- incremental development;
- RAD: DSDM;
- XP;
- SCRUM;

3.5.1 Prototyping

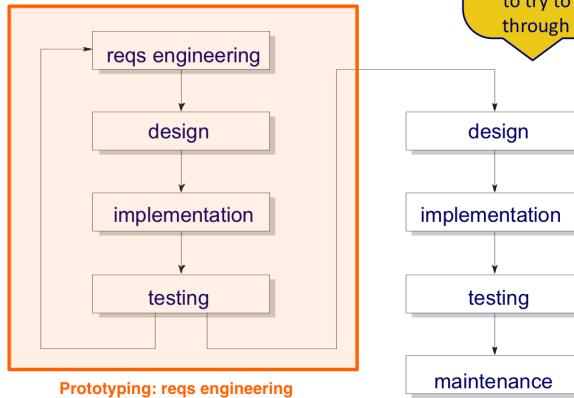
- l'elicitazione dei requisiti è difficile
 - Una delle principali difficoltà per gli utenti è esprimere le proprie esigenze precisamente. È naturale cercare di chiarirli attraverso la prototipazione. Questo può essere ottenuto sviluppando rapidamente l'interfaccia utente;
In questo modo, l'utente può ottenere un'visione di ciò che il sistema futuro sarà prima di grandi investimenti per realizzare il sistema.
– tuttavia, la nuova situazione auspicabile è ancora sconosciuta;
 - la prototipazione viene utilizzata per ottenere i requisiti di alcuni aspetti del sistema;
 - la prototipazione dovrebbe essere un processo relativamente economico;
 - utilizzare linguaggi e strumenti di prototipazione rapida;
 - non tutte le funzionalità devono essere implementate;
 - la qualità della produzione non è richiesta;

Il lato sinistro della figura riguarda le fasi di prototipazione, l'iterazione corrisponde al processo di convalida dell'utente, per cui requisiti nuovi o modificati attivano il ciclo successivo.

Il lato destro riguarda la produzione effettiva del sistema operativo. La differenza tra i due rami è che, usando diverse tecniche e strumenti, il lato sinistro può essere attraversato molto più rapidamente e a fronte di costi molto inferiori.

Prototyping thus becomes a tool for requirements engineering.

Prototyping as a tool for requirements engineering



Tipologie di Prototyping:

- **throwaway prototyping:** non si riporta il prodotto software dalle fasi di prototipazione alla fase di produzione vera e propria, ma si butta via esplicitamente dopo che le fasi sono terminate.
- **evolutionary prototyping:** l'utente porta nuovi/modificati requisiti i quali producono una nuova versione del prototipo. Dopo una serie di tali iterazioni, l'utente è soddisfatto e l'ultima versione sviluppata è il prodotto da consegnare.

Advantages
- The resulting system is easier to use - User needs are better accommodated - The resulting system has fewer features - Problems are detected earlier - The design is of higher quality - The resulting system is easier to maintain - The development incurs less effort
Disadvantages
- The resulting system has more features - The performance of the resulting system is worse - The design is of lesser quality - The resulting system is harder to maintain - The prototyping approach requires more experienced team members

3.5.2 Incremental development

- un sistema software viene consegnato in piccoli incrementi, evitando così l'effetto Big Bang;
 - per molto tempo non succede nulla e poi, all'improvviso, c'è una situazione completamente nuova.
- il waterfall model viene impiegato in ogni fase;
- l'utente è strettamente coinvolto nella direzione dei passi successivi;
- lo sviluppo incrementale previene la sovrafunzionalità
 - l'attenzione viene prima focalizzata sulle caratteristiche essenziali;
 - La funzionalità aggiuntiva è inclusa solo se e quando è necessaria.
- Con l'approccio incrementale, le parti più difficili vengono spesso affrontate per prime, o le parti che presentano i maggiori rischi rispetto al buon esito del progetto.

Seguendo questa linea di pensiero, Boehm (1988) propone un **modello a spirale** della processo di sviluppo del software, in cui ogni convoluzione della spirale dà origine a le seguenti attività:

- identificare il sottoproblema a cui è associato il rischio più elevato;
- trovare una soluzione per quel problema;

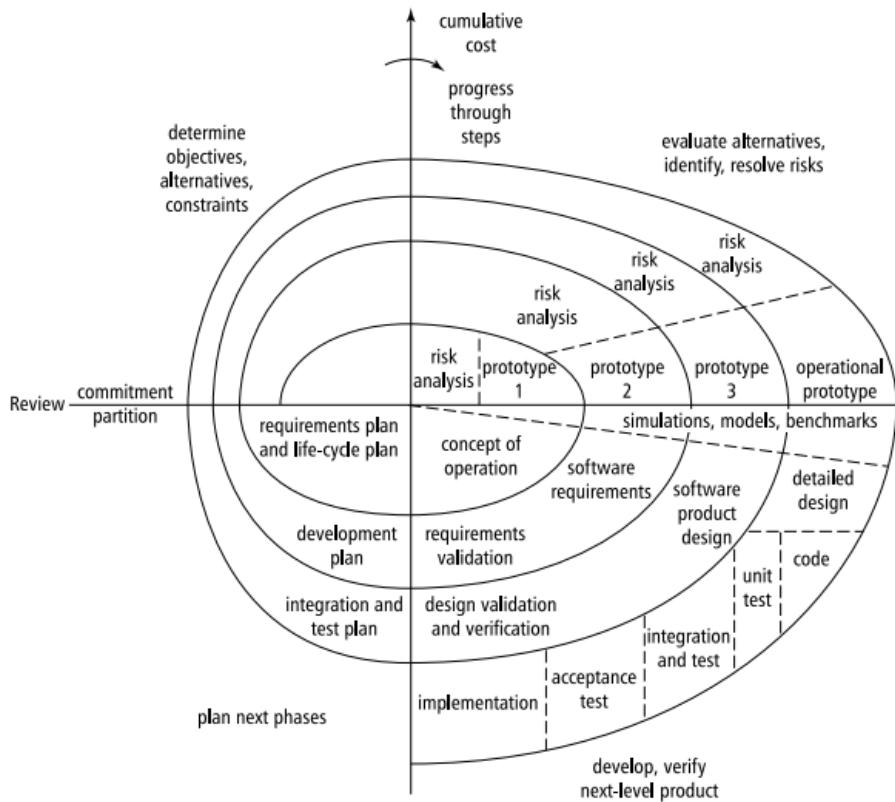


Figure 6: The spiral model

3.5.3 RAD: Rapid Application Development and DSDM

Il Rapid Application Development (RAD) ha molto in comune con altri iterativi modelli di processo di sviluppo. Sottolinea il coinvolgimento dell'utente, la prototipazione, il riutilizzo, il utilizzo di strumenti automatizzati e piccoli team di sviluppo.

- sviluppo evolutivo, con time boxes: tempi fissi entro i quali vengono svolte le attività;
- prima si decide la tempistica, poi si cerca di realizzare quanto più possibile entro quella tempistica;
- Le principali tecniche utilizzate in queste fasi: **JRP** (Joint Requirements Planning) e **JAD** (Joint Application Design);

- Entrambe queste tecniche fanno un uso massiccio di workshop in cui gli sviluppatori e i potenziali utenti lavorano insieme;
 - L’obiettivo del workshop JRP è ottenere i requisiti giusti la prima volta; (Questa priorità dei requisiti è nota come *triage*)
 - * Triage di solito significa un processo utilizzato su sul campo di battaglia e nei pronto soccorso per ordinare i feriti in gruppi con necessità/cure mediche simili.
- In RAD, il triage viene utilizzato per assicurarsi che i requisiti più importanti vengano affrontati per primi. Il risultato di questo processo è spesso una prioritizzazione indicata dall’acronimo MoSCoW:
 - **Must haves** sono requisiti assolutamente necessari;
 - **Should haves** sono requisiti importanti, ma non assolutamente necessari per un sistema utilizzabile;
 - **Could haves** sono requisiti che vengono implementati solo se il tempo lo consente;
 - **Won’t haves** sono requisiti che verranno lasciati per la prossima iterazione;
- sviluppo in un team SWAT: un team altamente qualificato di circa quattro persone

A well-known framework that builds on RAD is **DSDM** (Dynamic Systems Development Method):

- Idea fondamentale: fissare tempo e risorse (timebox), regolare la funzionalità di conseguenza;
- DSDM è un framework senza scopo di lucro, gestito dal Consorzio DSDM;
- Il set completo delle pratiche DSDM è disponibile solo per i membri del DSDM Consorzio.
- Il processo DSDM ha cinque fasi:
 - **Feasibility study:** si valuta l’idoneità di DSDM per il progetto. Il DSDM è appropriato per questo progetto? Se sì, si consegna relazione di fattibilità e piano di massima, eventualmente prototipo veloce (poche settimane)

- **Business study:** si traduce in una descrizione di alto livello dei processi aziendali rilevante per il sistema. Questi sono determinati utilizzando workshop facilitati (come JRP). In questa fase viene definita l'architettura.
- **Functional model iteration:** si traduce in modelli di analisi, prototipi e implementazione delle principali componenti funzionali. L'iterazione viene eseguita in time boxes di solito da due a sei settimane. Ogni iterazione è composta da quattro attività:
 1. identificare cosa farai;
 2. concordare su come lo farai;
 3. farlo;
 4. controlla di averlo fatto.;**Enfasi su cosa costruire.**
- **Design and build iteration:** il sistema è progettato in modo sufficientemente alta qualità. Anche qui, il lavoro viene time boxes di tempo da due a sei in genere settimane e vengono eseguite le stesse quattro attività. In questa sezione si focalizza a costruire da un punto di vista ingegneristico le componenti funzionali determinate nella fase precedente.
- **Implementation:** trasferimento in produzione ambiente;

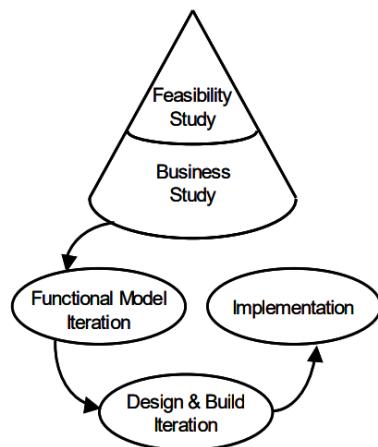


Figure 7: The DSDM process

- Pratiche DSDM:
 - Il coinvolgimento attivo dell’utente è imperativo;
 - Team potenziati;
 - Consegna frequente dei prodotti;
 - Sviluppo iterativo e incrementale;
 - Tutte le modifiche sono reversibili;
 - È essenziale un approccio collaborativo e cooperativo condiviso da tutte le parti interessate;

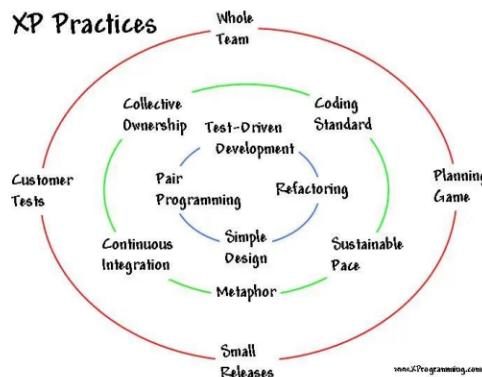
3.5.4 XP – Programmazione estrema

- Tutto si fa a piccoli passi;
- Il sistema compila sempre, esegue sempre;
- Cliente come centro del team di sviluppo;
- Gli sviluppatori hanno la stessa responsabilità rispetto al software e alla metodologia;

Extreme Programming, XP in breve, è un metodo agile puro. XP si basa su un numero di buone pratiche e le porta all'estremo.

13 practices of XP

1. Whole team: client part of the team
2. Metaphor: common analogy for the system
3. The planning game, based on user stories
4. Simple design
5. Small releases (e.g. 2 weeks)
6. Customer tests
7. Pair programming
8. Test-driven development: tests developed first
9. Design improvement (refactoring)
10. Collective code ownership
11. Continuous integration: system always runs
12. Sustainable pace: no overtime
13. Coding standards



- Pratiche XP

1. **Whole team:** vengono incluse persone con differenti competenze: analisi, progettazione, programmazione e test;
2. **Metaphor:**
 - Ogni progetto è guidato da una metafora condivisa da responsabili e sviluppatori;
 - La metafora è una descrizione sintetica del sistema nel suo complesso e fornisce un vocabolario comune a tutte le persone coinvolte, senza scendere nei dettagli implementativi;
3. **Planning game:** Lo sviluppo dell'applicazione è accompagnato dalla stesura di un piano di lavoro. Il piano è definito e continuamente aggiornato, a intervalli brevi e regolari dai responsabili del progetto, secondo le priorità aziendali e le scadenze dei programmatore;
4. **Simple design:** L'architettura dell'applicazione deve essere la più semplice possibile. A fronte di nuove situazioni, saranno progettati nuovi componenti o riprogettati quelli esistenti;
5. **Small releases:**
 - Ogni rilascio realizza un insieme di casi d'uso;
 - Ogni rilascio è la conclusione di un'iterazione di sviluppo e l'inizio di una nuova pianificazione;
 - Ogni iterazione dovrebbe durare non più di qualche settimana;
6. **Customer test:**
 - Il cliente deve essere coinvolto nello sviluppo;
 - Il cliente partecipa alla stesura dei test di sistema e verifica che il sistema realizzato corrisponda alle proprie esigenze;
7. **Pair programming**
 - La scrittura del codice è fatta da coppie di programmatore che lavorano al medesimo terminale;
 - Le coppie non sono fisse, ma si compongono associando le migliori competenze per la risoluzione di uno specifico problema;

- Il lavoro in coppia permette, scambiandosi periodicamente i ruoli, di mantenere mediamente più alto il livello d'attenzione;

8. Test driven development:

- L'applicazione è verificata sia a livello di sistema (test di sistema);
- sia a livello del singolo metodo (test di unità);
- I test di sistema sono basati su casi d'uso;
- I test di unità sono rieseguibili automaticamente;

9. Refactoring: L'applicazione è periodicamente ristrutturata per eliminare parti superflue o semplificare la struttura;

10. Collective code ownership: Il codice dell'applicazione può essere liberamente manipolato da qualsiasi sviluppatore (che deve essere messo nelle migliori condizioni di comprenderlo);

11. Continuous integration: Il codice sviluppato è frequentemente integrato in modo da avere una versione funzionante a disposizione di tutti i programmati che cresce.

12. Sustainable pace:

- Lo sviluppo di applicazioni con i metodi dell'eXtreme Programming è un'attività che richiede grande concentrazione;
- Gli straordinari sono da evitare per non provocare un deterioramento della qualità dell'impegno;

13. Coding standard:

- Il codice deve essere scritto sulla base di regole;
- Tutti gli sviluppatori devono essere in grado di capire e modificare ogni linea di codice scritto da altri;

3.5.5 SCRUM

- Il **Product Backlog** è di proprietà del **Product Owner**, è un documento in continua evoluzione e ha la finalità di raggiungere un obiettivo a lungo termine per il prodotto: il **Product Goal**.
 - Il Product Backlog è un artefatto ufficiale di Scrum che consiste in un elenco di attività (Product Backlog Item) ordinato per priorità;
 - Il Product Backlog viene costantemente rivisto e riordinato dal Product Owner in base alle necessità degli utenti o del cliente, le aspettative degli stakeholder, nuove idee, o in seguito alle esigenze di mercato – ma anche in base a suggerimenti da parte del team;
 - In fase di pianificazione, quindi durante lo Sprint Planning, i Developer prendono in esame un sottoinsieme del Product Backlog. In particolare tutti i Product Backlog Item con priorità più alta, e che quindi si trovano in cima all'elenco.
- Lo **Sprint Planning** è un meeting in cui il team pianifica il lavoro che deve essere svolto e portato a termine durante lo Sprint.
 - lo Sprint Planning prevede un maggior livello di dettaglio e opzionalmente una suddivisione in task;
 - Il risultato dello Sprint Planning è lo **Sprint Backlog**.
 - La squadra ha una certa cifra di tempo, uno **sprint**, per completare la sua lavoro – di solito dalle due alle quattro settimane;
 - Alla **fine dello sprint**, il lavoro dovrebbe essere potenzialmente spedibile, come in pronta consegna a un cliente, metti su un negozio scaffale o mostrare a uno stakeholder;
 - The sprint ends with a sprint **review** and retrospective.
 - As the next sprint begins, the team chooses another chunk of the product backlog and begins working again.
- Lungo la strada, lo ScrumMaster mantiene il team concentrato sul proprio obiettivo;
- Il ciclo si ripete fino a quando non è stato raggiunto un numero sufficiente di elementi nel product backlog è stato completato, il budget è esaurito o arriva una scadenza;

- Scrum assicura che il lavoro più prezioso sia stato completato quando il progetto finisce;

3.6 RUP - Rational Unified Process

- Complement to UML (Unified Modeling Language);
- Iterative approach for object-oriented systems, strongly embraces use cases for modeling requirements
- Tool-supported (UML-tools, ClearCase);
- Might be viewed as somewhat intermediate between document-driven and agile methods;
- Ha un processo ben definito e include attività iniziali di ingegneria dei requisiti ragionevolmente estese, ma enfatizza il coinvolgimento delle parti interessate attraverso la sua natura basata sui casi d'uso;
- RUP distingue quattro fasi:
 - **inception:** : stabilire l'ambito, i confini, i casi d'uso critici, le architetture candidate, la pianificazione e le stime dei costi;
 - **elaboration:** fondamenti dell'architettura, stabilire il supporto degli strumenti, ottenere tutti i casi d'uso;
 - **construction:** processo di fabbricazione, uno o più rilasci;
 - **transition:** rilascio alla comunità degli utenti, spesso diverse versioni;

In una seconda dimensione, RUP distingue nove cosiddetti **workflows**. Questi workflows di lavoro raggruppano attività logiche e possono estendersi a tutte le fasi con diversi livelli di attenzione.

Questa struttura ci permette di distinguere che iterazioni diverse hanno un'enfasi diversa.

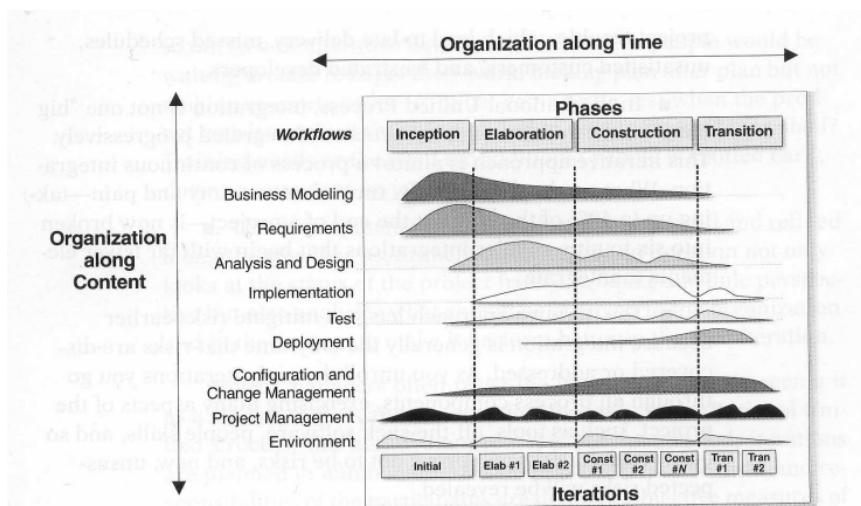


Figure 8: Two-dimensional process structure of RUP

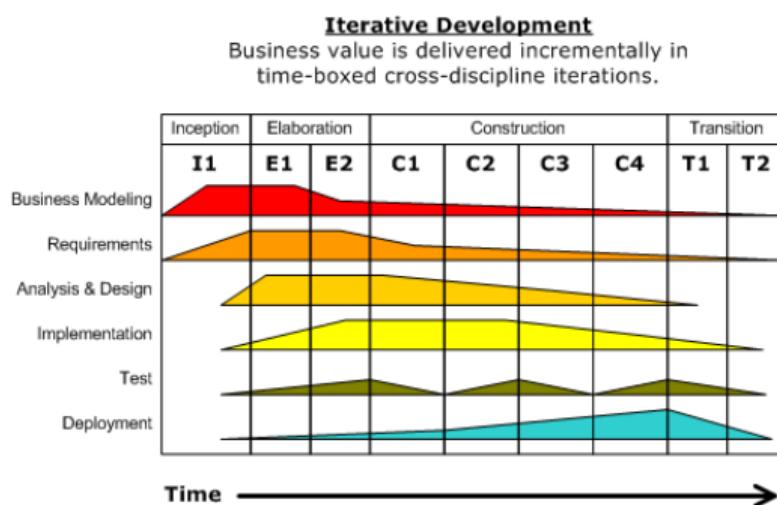


Figure 9: Iterative development in RUP

Best practice	Description
Iterative development	Systems are developed in an iterative way. This is not an uncontrolled process. Iterations are planned, and progress is measured carefully.
Requirements management	RUP has a systematic approach to eliciting, capturing and managing requirements, including possible changes to these requirements.
Architecture and Use of components	The early phases of RUP result in an architecture. This architecture is used in the remainder of the project. It is described in different views. RUP supports the development of component-based systems, in which each component is a nontrivial piece of software with well-defined boundaries.
Modeling and UML	Much of RUP is about developing models, such as a use-case model, a test model, etc. These models are described in UML.
Quality of process and product	Quality is not an add-on, but the responsibility of everyone involved. The testing workflow is aimed at verifying that the expected level of quality is met.
Configuration and change management	Iterative development projects deliver a vast amount of products, many of which are frequently modified. This asks for sound procedures to do so, and appropriate tool support.
Use-case-driven development	Use cases describe the behaviour of the system. They play a major role in various workflows, especially the requirements, design, test and management workflow.
Process configuration	No size fits all. Though RUP can be used "as-is", it can also be modified and tailored to better fit specific circumstances.
Tool support	To be effective, a software development needs tool support. RUP is supported by a wide variety of tools, especially in the area of visual modeling and configuration management.

Figure 10: Best practices of RUP

3.7 Differenze

- Differences for developers
 - **Agile:** molto semplice da imparare, basato sulla collaborazione, di facile collocazione come metodo;
 - **Heavyweight:** plan-driven, adequate skills, access to external knowledge;
- Differences for customers
 - **Agile:** si costruisce intorno al cliente;
 - **Heavyweight:** lo trascura e si focalizza sulla documentazione;
- Differences for requirements
 - **Agile:** largely emergent, rapid change;
 - **Heavyweight:** knowable early, largely stable;
- Differences for architecture
 - **Agile:** designed for current requirements;
 - **Heavyweight:** progettato per requisiti prevedibili;
- Differences for size
 - **Agile:** smaller teams and products;
 - **Heavyweight:** larger teams and products;
- Differences for primary objective
 - **Agile:** rapid value;
 - **Heavyweight:** high assurance (garanzia/sicurezza/certezza);

3.8 Model driven architecture/engeneering

- In traditional software development approaches, we build all kinds of models during requirements engineering and design.
 - Code = ultimate model
- Any evolution is accomplished by changing the last model in the chain, → **the source code**;
- requirements and design models are often not updated, so that they quickly become outdated. ⇒ **Is there a better bay to develop a system? da modello a codice**

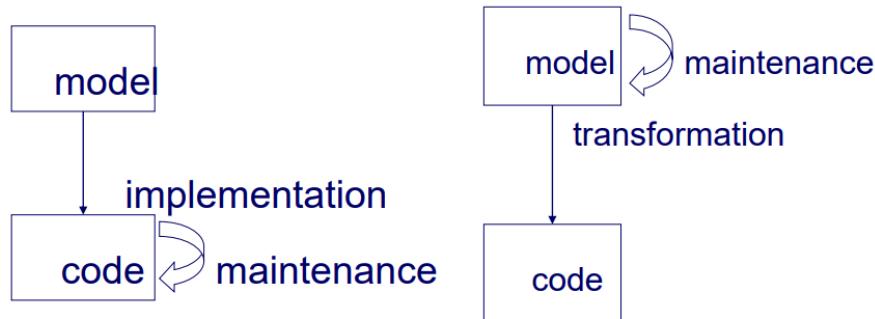


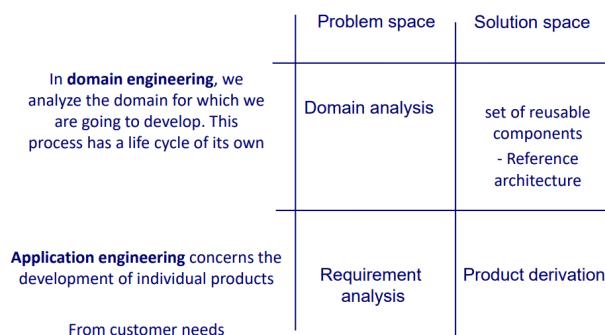
Figure 11: Best approach

- MDA é:
 - **computation-independent model (CIM)**: must contain business requirements for the software system;
 - **Platform Independent Model (PIM)**: is a model that is independent of the specific technological platform used to implement it;
 - **Model transformation and refinement**;
 - **Resulting in a Platform Specific Model (PSM)**: definisce le funzionalità in modo specifico per una particolare piattaforma di implementazione;

3.9 Maintenance or Evolution

- Osservazioni:
 - systems are not built from scratch → solitamente si parte da qualcosa che già esiste;
 - there is time pressure on maintenance → perché il sistema non funziona durante la manutenzione;
 - the system's documentation may not be updated;
- **the five laws of software evolution:**
 - **law of continuing change:** un sistema subisce continue modifiche, fino a quando non sarà giudicato più conveniente ristrutturare il sistema o sostituirlo da una versione completamente nuova;
 - **law of increasingly complexity:** le modifiche ad un sistema aumentano l'entropia, ossia il sistema diventa sempre meno strutturato e più complesso. Si deve spesso investire affinché si eviti l'aumento di complessità;
 - **Law of self regulation:** I processi di evoluzione del software sono autoregolanti e promuovere una crescita regolare del software;
 - **law of invariant work rate:** il processo di sviluppo di un sistema rimane sempre lo stesso;
 - **Law of conservation of familiarity:** un sistema subisce una costante crescita per mantenerne la familiarità (ridurne la complessità). Quando questo l'incremento viene superato problemi relativi alla qualità e all'utilizzo;
 - **Law of continuing growth:** La funzionalità di un sistema deve continuamente aumentare al fine di mantenere la soddisfazione degli utenti;
 - **Law of declining quality:** La qualità di un sistema diminuisce, a meno che non sia attivamente mantenuto e adattato al suo ambiente mutevole;
 - **Law of system feedback:** L'evoluzione del software deve essere vista come un sistema di feedback al fine di ottenere miglioramenti;

- Software Product Lines
 - developers are not inclined to make a maintainable and reusable product, it has additional costs;
 - two processes result: domain engineering, and application engineering;



3.10 Process modeling

Potremmo descrivere un processo di sviluppo software, o parti di esso, sotto forma di "programma".

- Il linguaggio definisce una serie di regole ben definite che ci permettono di stare entro limiti;
- Il linguaggio stesso puó essere anche simbolico;
- the **review process** is described in terms of the successive activities to be performed;
- We may also describe the process in terms of the states it can be in;
- **Petri nets** provide yet another formalism to describe process models;
- facilitates **understanding** and communication by providing a shared view of the process;
- supports **management** and improvement; it can be used to assign tasks, track progress, and identify trouble spots;
- serves as a basis for **automated** support (usually not fully automatic)

3.10.1 Petri-net view of the review process - vedi slides

- Vedi slide Cap 3 da pag 69

3.10.2 Avvertenze sulla modellazione dei processi

- not all aspects of software development can be caught in an algorithm;
- a model is a model, thus a simplification of reality;
- progression of stages differs from what is actually done;
- some processes tend to be ignored;

4 Configuration Management

Occorrono procedure attente per gestire il vasto numero di elementi (codice, documentazione, richieste di modifica, ecc.) che vengono creati e aggiornati nel corso della vita di un grande sistema software. Questo è particolarmente vero in progetti di sviluppo distribuito. Si chiama **configuration management**.

4.1 Tasks and responsibilities

- **baseline:** una versione ufficiale del set completo di documenti relativi al progetto;
- items contained in the baseline
 - source code components;
 - the requirements specification;
 - the design documentation;
 - the test plan, test cases, test results;
 - the user manual;
- Initially the baseline will contain a requirements specification. As time goes on, elements will be added;
- Any proposed change to the baseline is called a **change request (CR)**;
- gestire le modifiche e creare elementi di configurazione disponibile durante il ciclo di vita del software, di solito attraverso un **Configuration Control Board (CCB)**;
 - Un gruppo di persone qualificate con la responsabilità di regolare l'approvazione di modifiche all'hardware, firmware, software e documentazione durante tutto lo sviluppo e ciclo di vita operativo di un sistema informativo;
- keeping track of the **status** of all items (including the change requests);

4.2 Configuration Control Board

- ensures that every change to the baseline (change request - CR) is properly authorized and executed;
- CCB needs certain information for every CR:
 - such as who submits it?
 - how much it will cost?
 - eccetera
- CCB valuta il CR. Se è approvato, risulta in un pacchetto di lavoro che deve essere schedulato;
- so, **configuration management** is not only about keeping track of changes, but also about workflow management;

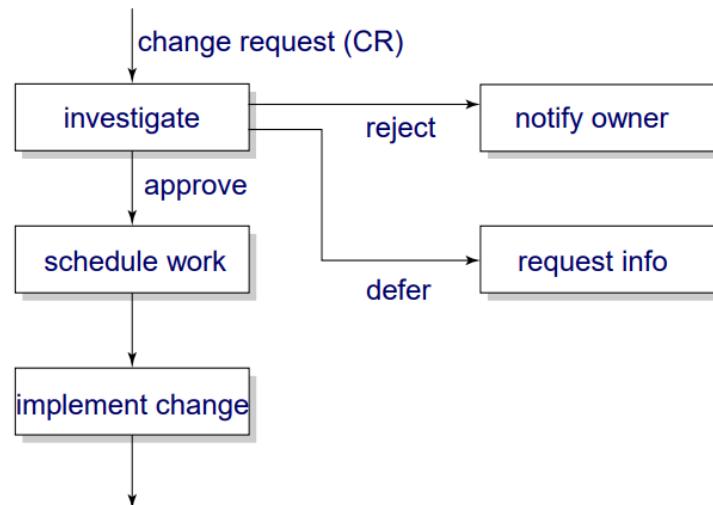


Figure 12: Workflow of a change request

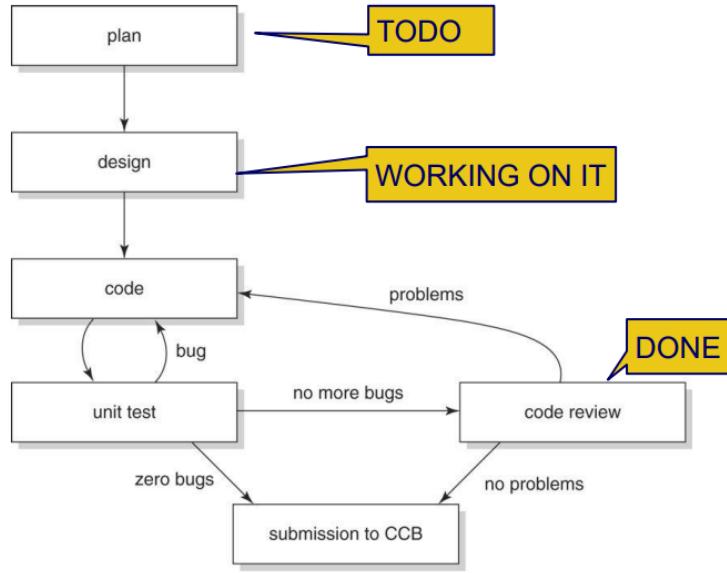
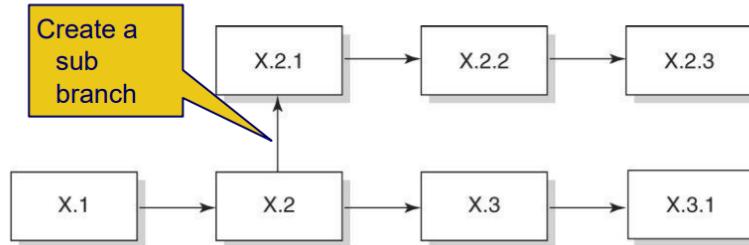


Figure 13: development activities (example)

4.3 Tool support for configuration management

- se un item deve essere cambiato, una persona ne tiene una copia e nel frattempo è lockata a tutte le altre persone;
- i nuovi elementi possono essere aggiunti alla linea di base solo dopo test approfonditi;
- cambiamenti allo status di un item (es: code finished) triggerà ulteriori attività (es: start unit testing);
- le versioni vecchie dei componenti vengono comunque mantenute con versione $X.1, X.2..$;
- we may even create different branches of revisions $X.2.1, X.2.2$ and $X.3.1, X.3.2$;

▪ When working in parallel



4.4 Functionalities of SCM tools (SCM: Software Configuration Manager)

- Components (storing, retrieving, accessing, . . .);
- Structure (representation of system structure);
- Construction (build an executable);
- Auditing (follow trails, e.g. of changes);
- Accounting (gather statistics);
- Controlling (trace defects, impact analysis);
- Process (assign tasks);
- Team (support for collaboration);

4.5 Models of configurations

- **version-oriented:** il cambiamento fisico si traduce in un nuovo versione, quindi le versioni sono caratterizzate dalla loro differenza, cioè delta;
- **change-oriented:** basic unit in configuration management is a logical change;
- identification of configuration becomes different:
*baseline X plus ‘fix table size problem’ rather than
”X3.2.1 + Y2.7 + Z3.5 + ...”*

4.6 Evolution of SCM tools

- **Early tools:** emphasis on product-oriented tasks;
- **Nowadays:** support for other functionalities too. They have become a (THE) major tool in large, multi-site projects;
- **Agile projects:** emphasis on running system: daily builds;

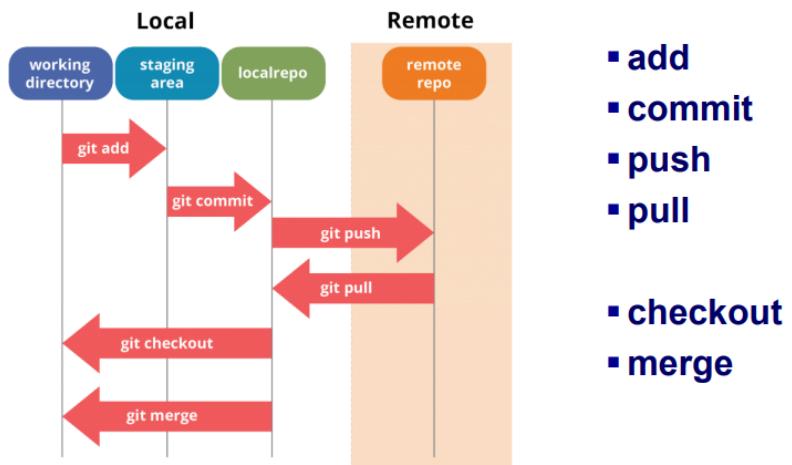
4.7 Configuration Management Plan

- **Management section:** organization, responsibilities, standards to use, etc;
- **Activities:** identification of items, keeping status, handling CRs

-
1. *Introduction*
 - a. Purpose
 - b. Scope
 - c. Definitions and acronyms
 - d. References
 2. *SCM management*
 - a. Organization
 - b. SCM responsibilities
 - c. Applicable policies, directives and procedures
 3. *SCM activities*
 - a. Configuration identification
 - b. Configuration control
 - c. Configuration status accounting
 - d. Configuration audits and reviews
 - e. Interface control
 - f. Subcontractor/vendor control
 4. *SCM schedules*
 5. *SCM resources*
 6. *SCM plan maintenance*
-

Figure 14: configuration management plan

4.8 Github - example



5 People Management, People Organization

5.1 Mintzberg's coordination mechanisms

- **Simple:** supervisione diretta;
- **Machine bureaucracy:** standardization of work in processes;
- **Divisionalized form:** standardization of work products;
- **Professional bureaucracy:** standardization of worker skills;
- **Adhocracy:** aggiustamento reciproco;
- **Simple structure**
 - ci possono essere uno o pochi manager e un nucleo di persone che fanno il lavoro. Il corrispondente meccanismo di coordinamento è chiamato supervisione diretta. Questa configurazione si trova spesso in nuovi, organizzazioni relativamente piccole. C'è poca specializzazione, formazione e formalizzazione. La coordinazione spetta a poche persone, che sono responsabili del lavoro degli altri.
- **Machine bureaucracy**
 - La produzione di massa e le linee di montaggio sono esempi tipici di questo tipo di configurazione. C'è poca formazione e molta specializzazione e formalizzazione. Il coordinamento si ottiene attraverso la standardizzazione dei processi di lavoro.
- **Divisionalized form**
 - a ciascuna divisione (o progetto) è concessa una notevole autonomia su come devono essere gli obiettivi dichiarati raggiunto. I dettagli operativi sono lasciati alla divisione stessa. Il coordinamento è raggiunto attraverso la standardizzazione degli output di lavoro. Il controllo viene eseguito misurando regolarmente le prestazioni della divisione. Questo meccanismo di coordinamento è possibile solo quando viene specificato il risultato finale preciso.

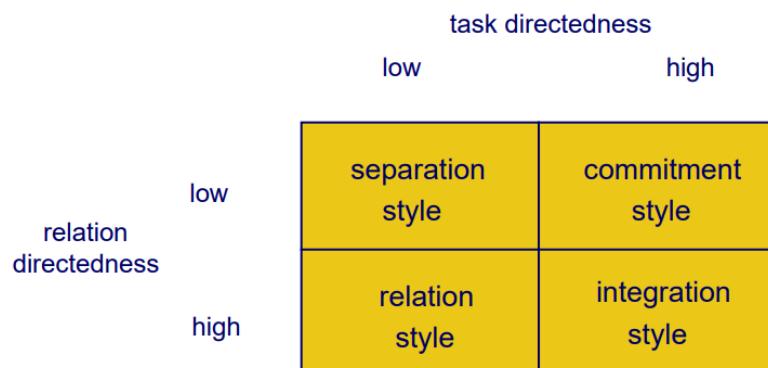
- **Professional bureaucracy**

- Se non è possibile specificare né il risultato finale né i contenuti del lavoro, il coordinamento può essere raggiunto attraverso la standardizzazione delle competenze dei lavoratori. In una burocrazia professionale, i professionisti qualificati sono dati considerevoli libertà su come svolgere il proprio lavoro.

- **Adhocracy**

- Nei progetti di grandi dimensioni o di natura innovativa, il lavoro è diviso tra molti specialisti. Potremmo non essere in grado di dire esattamente cosa dovrebbe fare ogni specialista, o come dovrebbero svolgere i compiti loro assegnati. Il successo dipende dalla capacità del gruppo nel suo insieme di raggiungere un obiettivo non specificato. Il coordinamento si ottiene attraverso la mutua regolazione tra compagni.

5.2 Reddin's management styles



5.3 Team Organization

- Hierarchical organization;
- Matrix organization;
- Chief programmer team;
- SWAT team;
- Agile team/Extreme Programming (XP);
- Open Source Development;

5.3.1 Hierarchical organization

- salendo aumenta la responsabilità così come il salario;
- diminuisce le attività da informatico e aumentano quelle di coordinamento;
- facendo carriera e salendo, le nuove persone non hanno la stessa esperienza; (impoverimento di personale specializzato)

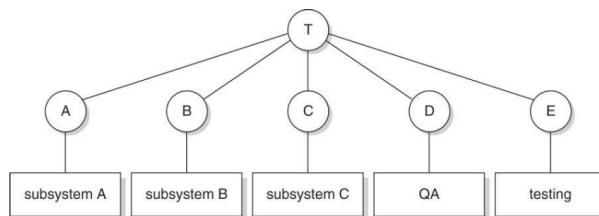
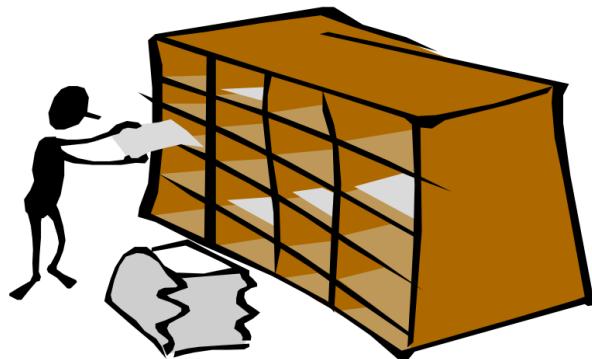


Figure 15: Vari leader a capo delle squadre

5.3.2 Matrix organization



	Real-time program ming	Graphics	Databas e s	QA	Testing
Project C	X			X	X
Project B	X		X	X	X
Project A		X	X	X	X

5.3.3 SWAT team

- A SWAT team is relatively small;
- It typically has four or five members;
- occupies one room;
- I canali di comunicazione sono molto brevi. La squadra non ha tempo incontri formali con verbali formali. Piuttosto, utilizza laboratori e sessioni di brainstorming con lavagna.

5.3.4 Agile team

- Esempio: **SCRUM team**
- **Product owner**
 - knows what the customer wants and the relative business;
 - He or she can then translate the customer's wants and values back to the Scrum team;

- must know the business case for the product and what features the customers' wants;
- He must be available to consult with the team to make sure they are correctly implementing the product vision;
- the authority to make all decisions necessary to complete the project;
- is responsible for managing the Product Backlog which includes:
 - * Expressing Product Backlog items clearly;
 - * Ordering the Product Backlog items to best achieve goals and missions;
 - * Optimizing the value of the work the Team performs;
 - * Ensuring that the Product Backlog is visible, transparent, and clear to all, and shows what the Team will work on further;
 - * Ensuring that the Team understands items in the Product Backlog to the level needed;

- **A Scrum Team**

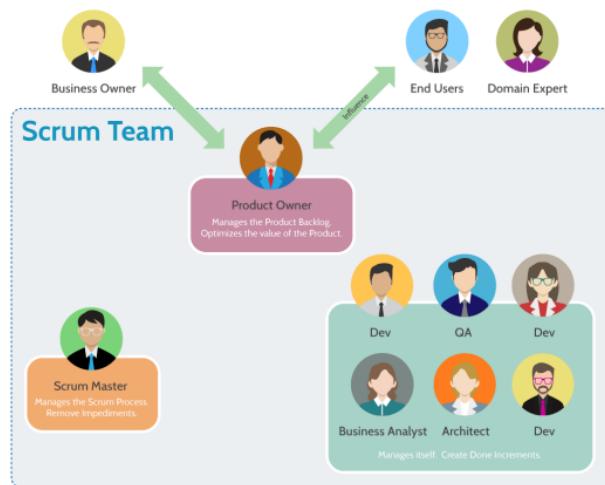
- typically between five and nine members;
- working together to deliver the required product increments;
- The Scrum framework encourages a high level of communication among team members;

- **Scrum master**

- aiuta a mantenere la squadra responsabile nei confronti dei propri impegni per l'azienda e rimuovere anche eventuali ostacoli che potrebbe ostacolare la produttività del team;
- si incontrano con il team su base regolare per rivedere il lavoro e i risultati finali, la maggior parte spesso con cadenza settimanale;
- Il ruolo di uno scrum master è quello di allenare e motivare i membri del team, non imporre loro delle regole;
- ensure the process run smoothly;
- remove obstacles that impact productivity;
- organize the critical events and meeting;

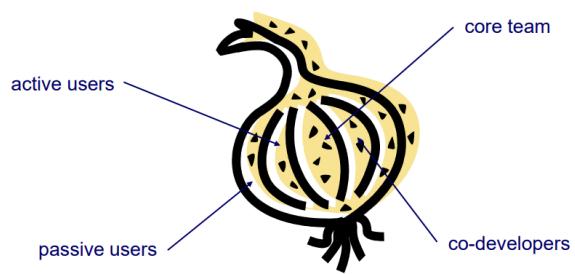
- **Development Team**

- They are self-organizing. No one (not even the Scrum Master) tells the Development Team how to turn Product Backlog into Increments of potentially releasable functionality;
- are cross-functional, with all the skills as a team necessary to create a product Increment;
- no sub-teams;
- Individual Development Team members may have specialized skills and areas of focus, but accountability belongs to the Development Team as a whole;



5.3.5 Open Source Software Development

- Use fewer, and better, people;
- Fit tasks to people;
- Help people to get the most out of themselves;
- Look for a well-balanced team;
- If someone doesn't fit the team: remove him;



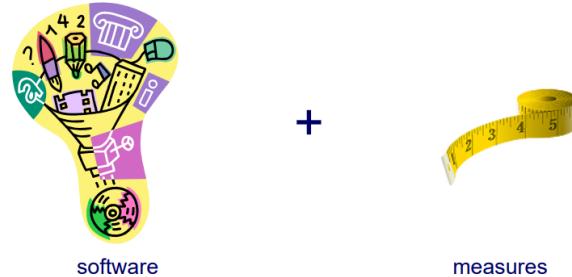
6 Managing Software Quality

- Qualità del prodotto contro qualità del processo;
- Verificare se (prodotto o processo) è conforme a certe norme;
- Migliorare la qualità migliorando il prodotto o i processi;

	Conformance	Improvement
Product	ISO 9126	'best practices'
Process	ISO 9001 SQA	CMM SPICE Bootstrap

Figure 16: Different approaches to quality

- What is quality?



6.1 How to measure “complexity”?

- The length of the program?
- The number of goto's?
- The number of if-statements?
- The sum of these numbers?
- Yet something else?

```

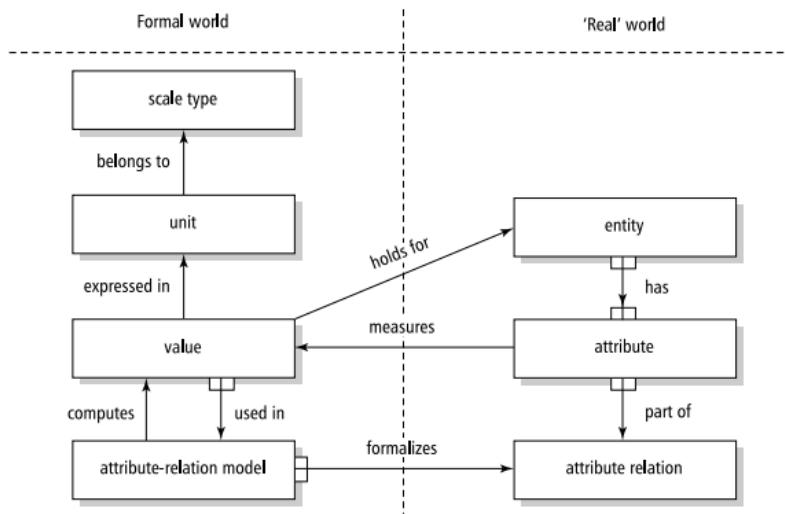
procedure bubble
  (var a: array [1..n] of integer; n: integer);
  var i, j, temp: integer;
begin
  for i:= 2 to n do
    j:= i; (a)
    while j > 1 and a[j] < a[j-1] do
      temp:= a[j];
      a[j]:= a[j-1];
      a[j-1]:= temp;
      j:= j-1;
    enddo
  enddo
end;

procedure bubble
  (var a: array [1..n] of integer; n: integer);
  var i, j, temp: integer;
begin
  for i:= 2 to n do
    if a[i] >= a[i-1] then goto next endif;
    j:= i;
    loop: if j <= 1 then goto next endif;
    if a[j] >= a[j-1] then goto next endif;
    temp:= a[j];
    a[j]:= a[j-1];
    a[j-1]:= temp;
    j:= j-1;
    goto loop;
  next: skip;
  enddo
end;

```

Figure 17: Which is better?

- using a **measurement framework**: Una serie di aspetti rilevanti della misurazione, come attributi, unità e tipi di scala possono essere introdotti e correlati tra loro utilizzando il measurement framework. Questo ci permette di indicare come le metriche possano essere utilizzate per descrivere e prevedere le proprietà dei prodotti e processi, e come convalidare queste previsioni;



6.2 Framework measurement

Il framework measurement è caratterizzato da 7 costituenti

- **Entity:** Un'entità è un oggetto nel mondo "reale" di cui vogliamo conoscere o prevedere determinate proprietà;
- **Attribute:** Le entità hanno determinate proprietà che chiamiamo attributi;
- tipi di scale:
 - **Nominal:** Gli attributi sono semplicemente classificati: il colore dei miei capelli è grigio, bianco o nero.
 - **Ordinal:** Esiste un ordinamento (lineare) nei possibili valori di un attributo: un tipo di materiale è più duro di un altro, un programma è più complesso di un altro;
 - **Interval:** Uguale all'ordinale, ma la "distanza" tra valori successivi di un attributo è lo stesso, come in un calendario, o la temperatura misurata in gradi Fahrenheit;
 - **Ratio:** Lo stesso dell'intervallo, con il requisito aggiuntivo che ci esiste un valore 0, come la temperatura misurata in gradi Kelvin;
 - **Absolute:** In questo caso contiamo semplicemente il numero di occorrenze, come in il numero di errori rilevati in un programma;
- **Unit:** Ovviamente questo valore è espresso in una certa unità, come metri, secondi o righe di codice;
- **Value:** vogliono caratterizzare formalmente questi oggetti misurando attributi, cioè assegnandoli dei valori.
- **Attribute relation:** È possibile mettere in relazione diversi attributi di una o più entità. Inoltre, un attributo di un'entità può essere correlato a un attributo di un'altra entità.
- **Attribute-relation model:** Se esiste una relazione tra attributi diversi di entità possibilmente diverse nel mondo "reale", possiamo esprimere tale relazione in un modello formale: il modello attributo-relazione. Questo modello calcola (predice) il valore di un attributo a cui siamo interessati dai valori di uno o più altri attributi del modello;

6.3 Representation condition

La **misurazione** è una mappatura dal mondo empirico, "reale" a quello formale, relazionale. Una **misura** è il numero o il simbolo assegnato a un attributo di un'entità da questa mappatura;

- Le relazioni empiriche tra gli oggetti nel mondo reale dovrebbero essere preservate nel sistema di relazioni numeriche che usiamo;
- Se osserviamo per esempio che l'auto A va più veloce dell'auto B, allora vorremmo che la nostra funzione S che mappa la velocità osservata risulti $S(A) > S(B) \rightarrow$ This is called **the representation condition**;
- Se la misura rispetta la representation condition, si dice che essa è una **valid measure**;
- if we measure complexity as the number of ifstatements, then:
 - Two programs with the same number of if-statements are equally complex;
 - If program A has more if-statements than program B, then A is more complex than B;

6.4 More on measures

- Direct versus indirect measures:
 - Spesso non possiamo misurare **direttamente** il valore di un attributo;
 - La velocità viene quindi misurata **indirettamente**, da prendendo il quoziente di due misure dirette. In questo caso, il modello attributo-relazione formalizza la relazione tra la distanza percorsa, il tempo e la velocità;
- Internal versus external attributes:
 - **External attributes:**
 - * di un'entità sono quelli che possono essere misurati solo rispetto a come quell'entità si riferisce al suo ambiente;
 - * Maintainability and usability are examples of external attributes;

- * can only be measured indirectly;
- **Internal attributes:**
 - * sono attributi che appartengono direttamente all'entità;
 - * Modularity, size, defects encountered, and cost are typical examples;

6.5 A taxonomy of quality attributes

6.5.1 Quality attributes (McCall) - Model quality

Correctness: The extent to which a program satisfies its specifications and fulfills the user's mission objectives.

Reliability: The extent to which a program can be expected to perform its intended function with required precision.

Efficiency: The amount of computing resources and code required by a program to perform a function.

Integrity: The extent to which access to software or data by unauthorized persons can be controlled.

Usability: The effort required to learn, operate, prepare input, and interpret output of a program.

Maintainability: The effort required to locate and fix an error in an operational program.

Testability: The effort required to test a program to ensure that it performs its intended function.

Flexibility: The effort required to modify an operational program.

Portability: The effort required to transfer a program from one hardware and/or software environment to another.

Reusability: The extent to which a program (or parts thereof) can be reused in other applications.

Interoperability: The effort required to couple one system with another.

- **Product operation:** all'uso del software dopo di esso è diventato operativo
- **Product revision:** la manutenibilità del sistema
- **Product transition:** la terza classe contiene fattori che riflettono la facilità con cui si passa a un nuovo ambiente.

Product operation:	
Correctness	Does it do what I want?
Reliability	Does it do it accurately all of the time?
Efficiency	Will it run on my hardware as well as it can?
Integrity	Is it secure?
Usability	Can I run it?
Product revision:	
Maintainability	Can I fix it?
Testability	Can I test it?
Flexibility	Can I change it?
Product transition:	
Portability	Will I be able to use it on another machine?
Reusability	Will I be able to reuse some of the software?
Interoperability	Will I be able to interface it with another system?

6.5.2 Taxonomy of quality attributes (ISO 9126) - Model quality

The ISO quality characteristics strictly refer to a **software product**. Their definitions do not capture process quality issues.

- La ISO 9126 misura la "qualità in uso": la misura i quali gli utenti possono raggiungere il loro obiettivo
 - Effectiveness
 - Productivity
 - Safety
 - Satisfaction

Effectiveness: The capability of the software product to enable users to achieve specified goals with accuracy and completeness in a specified context of use.

Productivity: The capability of the software product to enable users to expend appropriate amounts of resources in relation to the effectiveness achieved in a specified context of use.

Safety: The capability of the software product to achieve acceptable levels of risk of harm to people, business, software, property or the environment in a specified context of use.

Satisfaction: The capability of the software product to satisfy users in a specified context of use.

Per maggiori informazioni guardare pagina 120 libro (130 pdf)

Characteristic	Subcharacteristics
Functionality	Suitability Accuracy Interoperability Security Functionality compliance
Reliability	Maturity Fault tolerance Recoverability Reliability compliance
Usability	Understandability Learnability Operability Attractiveness Usability compliance
Efficiency	Time behavior Resource utilization Efficiency compliance
Maintainability	Analyzability Changeability Stability Testability Maintainability compliance
Portability	Adaptability Installability Co-existence Replaceability Portability compliance

Figure 18: Quality sub-characteristics of the external and internal quality model of ISO 9126

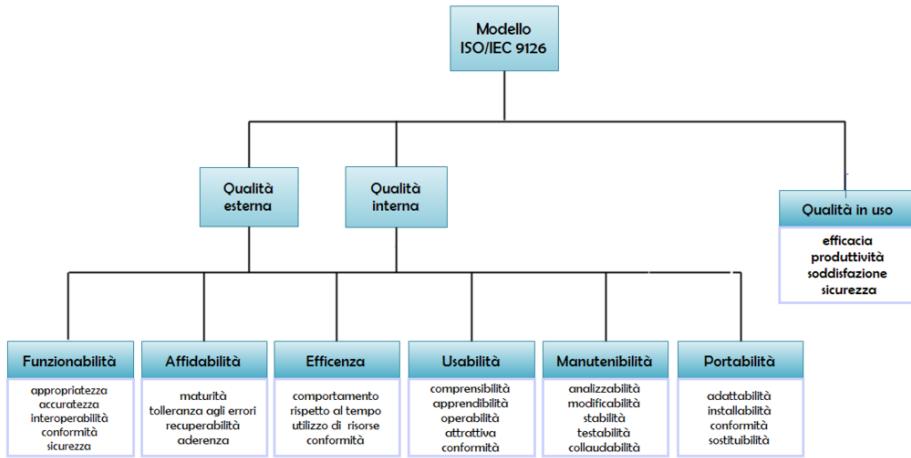


Figure 19: ISO/IEC 9126

6.6 ISO 9001

- ISO 9000 is a series of standards for quality management systems → **Processes**;
- ISO 9001 is a generic standard that can be applied to any product;
- Model for quality assurance in design, development, production, installation and servicing;
- Premessa di base: fiducia nel prodotto la conformità può essere ottenuta mediante un'adeguata dimostrazione delle capacità del fornitore in processi (progettazione, sviluppo, . . .);
- Registrazione ISO da parte di un ente ufficialmente accreditato, reimmatricolazione ogni tre anni;

6.7 SQA and IEEE standard 730

- The purpose of Software Quality Assurance (SQA) is to make sure that work gets done the way it is supposed to be done;
- IEEE Standard 730 offers a framework for the contents of a Quality Assurance Plan for software development;

6.8 Capability Maturity Model (CMM)

- Al fine di misurare il processo di miglioramento del processo di sviluppo del software, Watts Humphrey ha sviluppato un **framework** di maturità del software che si è evoluto in il Capability Maturity Model (CMM).
- software development process can be controlled, measured, and improved;
- In CMM, il processo software è caratterizzato in uno dei cinque livelli di maturità/livelli evolutivi verso il raggiungimento di un processo software maturo.
 - **Initial:** Lo sviluppo del software a questo livello può essere caratterizzato come ad hoc. Le prestazioni possono essere migliorate istituendo un project management di base
 - **Repeatable:** fornisce il controllo sul modo in cui vengono stabiliti i piani e gli impegni. Attraverso precedenti esperienze nel fare lavoro simile. Lo sviluppo di una nuova tipologia di prodotto e grandi cambiamenti organizzativi tuttavia rappresentano ancora grandi rischi a questo livello;
 - **Defined level:** un insieme di processi standard per il lo sviluppo e la manutenzione del software è in atto. L'organizzazione ha hanno raggiunto una solida base e possono ora iniziare a esaminare i loro processi e decidere come migliorarli;
 - **Quantitatively managed level:** i dati quantitativi vengono raccolti e analizzati su base routinaria. Tutto è sotto controllo e l'attenzione può quindi passare dall'essere reattiva - cosa succede al presente progetto? – per essere proattivi – cosa possiamo fare per migliorare progetti futuri? L'attenzione si sposta sulle opportunità di miglioramento continuo;
 - **Optimizing level:** Mentre l'attenzione agli altri livelli è focalizzata su modi per migliorare il prodotto, l'enfasi a livello di ottimizzazione si è spostata da il prodotto al processo. I dati raccolti possono ora essere utilizzati per migliorare il stesso processo di sviluppo del software;

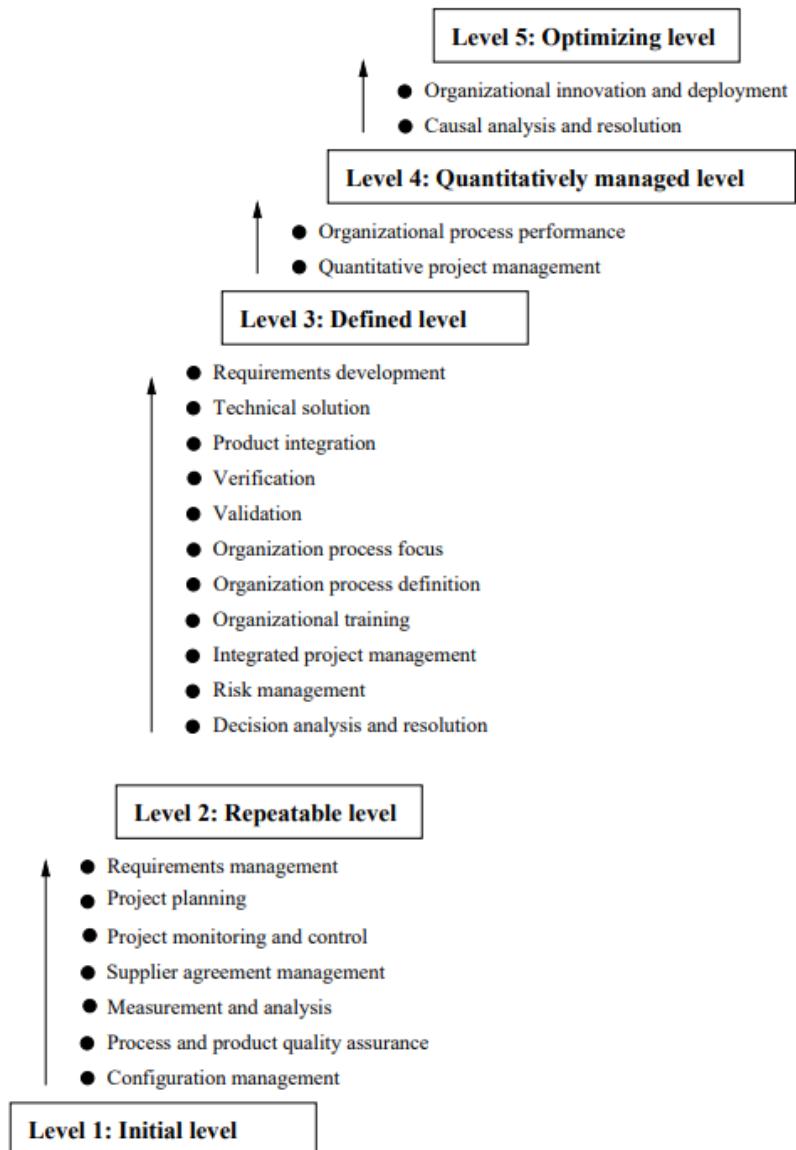


Figure 20: Maturity levels and associated process areas of CMM

- **CMM: critical notes**

- Most appropriate for big companies;
- Il puro approccio CMM può soffocare la creatività;
- Crude 5-point scale;

7 Requirements Engineering

This chapter covers requirements engineering, the first major phase in a software development project. The most challenging and difficult aspect of requirements engineering is to get a complete description of the problem to be solved. We discuss a number of techniques for eliciting requirements from the user. Following elicitation, these requirements must be negotiated, validated, and documented.

- the first step in finding a solution for a data processing problem;
- the results of requirements engineering is a **requirements specification (document)**;
- requirements specification:
 - contract for the customer;
 - starting point for design;
- the phase in which the user's requirements are analyzed and documented is also sometimes called the '**specification**' phase;
- Requirements engineering and design generally cannot be strictly separated in time;
 - Often, a preliminary design is done after an initial set of requirements has been determined;
 - Based on the result of this design effort, the requirements specification may be changed and refined;
- Il successo dipende da quanto bravi siamo in grado di descrivere il sistema desiderato;

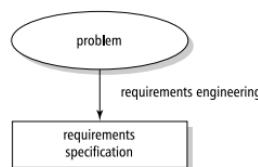


Figure 21: The first part of the software life cycle

7.1 Requirement (IEEE610)

- A requirement is ‘a condition or capability needed by a user to solve a problem or achieve an objective’;
- it has to be met? → La maggior parte dei requisiti sono negoziabili;

7.2 Natural language specs are dangerous

- fraintendibilitá dei termini;
- termini utilizzati in maniera errata o confusi;
- puó rappresentare un grosso problema;

7.3 market-driven vs customer-driven

- Much software developed today is market-driven rather than customer-driven;
- No real user can be questioned to build the requirements;

7.4 Requirements engineering, main steps

This is an iterative process

1. understanding the problem: **elicitation**;
2. describing the problem: **specification**
3. agreeing upon the nature of the problem: **validation**
4. agreeing upon the boundaries of the problem: **negotiation**

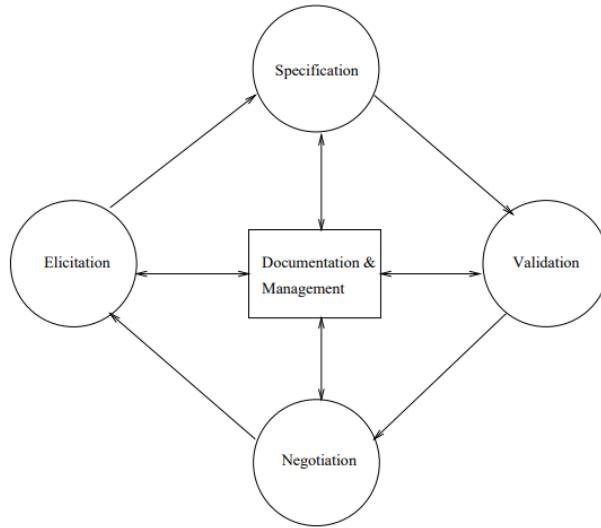


Figure 22: A framework for the requirements engineering process

7.4.1 Elicitation

- In generale, l'analista dei requisiti non è un esperto in il dominio da modellare. Attraverso l'interazione con specialisti di dominio, deve costruirsi un modello sufficientemente ricco;
- Pertanto, l'elicitazione dei requisiti riguarda la comprensione del problema;
- Diverse discipline siano coinvolte in questo processo, complica le cose;
- This process is also known as **Conceptual modeling**
- we are modeling part of reality → The part of reality in which we are interested is referred to as the **universe of discourse (UoD)**
- The model constructed during the requirements engineering phase is an **explicit conceptual model** of the UoD;
 - The adjective ‘explicit’ indicates that the model must be able to be communicated to the relevant people (such as analysts and users);
 - * analysis problems;
 - * negotiation problems;

7.5 Requirements Engineering Paradigms

- Most requirements engineering methods, and software development methods in general, are **Taylorian** in nature
 - which tasks are recursively decomposed into simpler tasks and each task has one ‘best way’ to accomplish it;
 - Con attente osservazioni ed esperimenti questo modo migliore può essere trovato e formalizzata in procedure e regole;
 - This view of software development is a functional, and rational one
 - Sebbene questo punto di vista abbia i suoi meriti nel redigere requisiti in termini puramente tecnici
 - * → **Problema:** molti UoD di interesse coinvolgono anche persone - persone che possono generare modelli incompleti, soggettivi, irrazionali e possono entrare in conflitto con la versione degli altri;

Necessità di persone specializzate in dominio che diano forma al UoD;

7.6 Elicitation techniques

- | | |
|--|--|
| <ul style="list-style-type: none">▪ Asking<ul style="list-style-type: none">▪ interview▪ Delphi technique▪ brainstorming session▪ task analysis▪ scenario analysis▪ ethnography▪ form analysis▪ analysis of natural language descriptions | <ul style="list-style-type: none">▪ synthesis from existing system▪ domain analysis▪ Business Process Redesign (BPR)▪ prototyping |
|--|--|

Figure 23: A sample of requirements elicitation techniques

Technique	Main info source		Strong on	
	Domain	User	Current	Future
Interview		X	X	
Delphi technique		X	X	
Brainstorming session		X		X
Task analysis		X	X	
Scenario (use-case) analysis		X	X	X
Ethnography	X		X	
Form analysis	X		X	
Analysis of natural language descriptions	X		X	
Synthesis of reqs from an existing system	X		X	
Domain analysis	X		X	
Use of reference models	X		X	
Business Process Redesign (BPR)	X		X	X
Prototyping		X		X

Figure 24: A sample of requirements elicitation techniques

7.6.1 Asking

- We may simply ask the users what they expect from the system;
- Un presupposto quindi è che l'utente sia in grado di consciere i limiti dei sistemi e sapere specificamente quello che si vuole;
 - **interviews**
 - **The Delphi technique:** è una tecnica iterativa in cui le informazioni vengono scambiate in forma scritta fino al raggiungimento di un consenso;

7.6.2 Task Analysis

- I dipendenti che lavorano in un dominio svolgono una serie di compiti;
- Le attività di livello superiore possono essere scomposte in attività secondarie;
- Task Analysis è una tecnica per ottenere una gerarchia di compiti e sottocompiti da svolgere da persone che lavorano nel dominio;
- Non ci sono chiare regole su quando interrompere le attività di decomposizione.
 - Un'euristica importante è che a un certo punto gli utenti tendono a "rifiutarsi" di scomporre ulteriormente le attività;

- Task analysis viene spesso applicata nella fase l'interazione uomo-macchina sono in fase di decisione. \Rightarrow **Parte grafica**
- Solitamente questa metodologia viene applicata quando esiste già un prodotto su cui intervenire. Tuttavia, può essere utilizzato come punto di partenza per un nuovo sistema:
 - users will refer to new elements of a system and its functionality;
 - scenario-based analysis can be used to exploit new possibilities;

7.6.3 Scenario-Based Analysis

- Fornisce una prospettiva di visualizzazione più orientata all'utente sul design e sviluppo di un sistema interattivo;
- Uno scenario è la descrizione di un'attività che l'utente svolge
 - la descrizione deve essere sufficientemente dettagliata in modo che il progetto implicazioni possono essere dedotte e ragionate.

The scenario view	The standard view
<ul style="list-style-type: none"> ▪ concrete descriptions ▪ focus on particular instances ▪ work-driven ▪ open-ended, fragmentary ▪ informal, rough, colloquial ▪ envisioned outcomes 	<ul style="list-style-type: none"> ▪ abstract descriptions ▪ focus on generic types ▪ technology-driven ▪ complete, exhaustive ▪ formal, rigorous ▪ specified outcomes

Figure 25: Scenario-Based Analysis

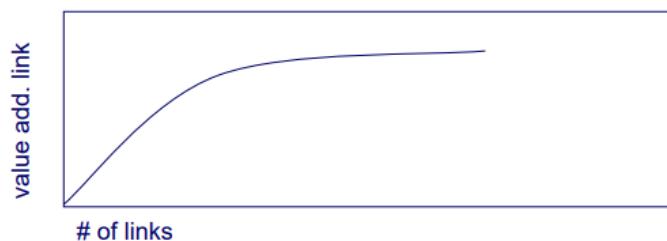
- **Application areas:**
 - requirements analysis
 - user-designer communication
 - design rationale
 - software architecture (& its analysis)
 - software design
 - implementation
 - verification & validation
 - documentation and training
 - evaluation
 - team building
- **Scenarios must be structured and managed**

7.7 Types of links between customer and developer



- | | |
|--|---|
| <ul style="list-style-type: none">▪ facilitated teams▪ intermediary▪ support line/help desk▪ survey▪ user interface prototyping▪ requirements prototyping▪ interview▪ usability lab | <ul style="list-style-type: none">▪ observational study▪ user group▪ trade show▪ marketing & sales |
|--|---|

- **lesson 1:** don't rely too much on indirect links (intermediaries, surrogate users)
 - indirect persons: sono persone esterne che non partecipano al progetto in maniera diretta e quindi programmano ...
- **lesson 2:** Tanti piú sono i modi per interagire con il cliente meglio é perché riusciamo a estrapolare piú informazioni in maniera precisa fino a un determinato punto;



7.8 Structuring a set of requirements - Goals and Viewpoints

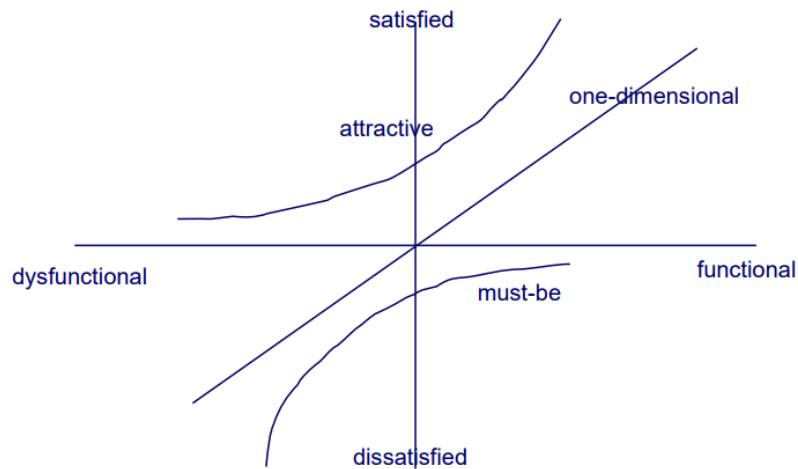
- **Hierarchical structure:** higher-level reqs are decomposed into lower-level reqs;
- Link requirements to specific stakeholders;
- In both cases, elicitation and structuring go hand in hand;

7.9 Prioritizing requirements (MoSCoW)

- **Must haves:** top priority requirements
- **Should haves:** highly desirable
- **Could haves:** if time allows
- **Won't haves:** not today

7.10 Prioritizing requirements (Kano model)

- **Attractive:** more satisfied if +, not less satisfied if -;
- **Must-be:** dissatisfied when -, at most neutral;
- **One-dimensional:** satisfaction proportional to number;
- **Indifferent:** don't care;
- **Reverse:** opposite of what analyst thought;
- **Questionable:** preferences not clear;



7.11 Crowdsourcing

- the requirements engineering process is outsourced to a large community of volunteers;
- Gives rise to new business model;

7.12 COTS selection

- **COTS: Commercial-Off-The-Shelf**

- the customer has to choose from what is available;

- **Iterative process:**

- Define requirements;
 - Select components;
 - Rank components;
 - Select most appropriate component, or iterate;

- **Simple ranking:** weight * score (WSM – Weighted Scoring Method)

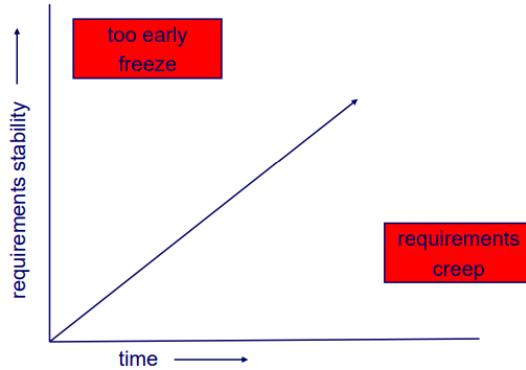
7.13 Requirements documentation and management

Come deve essere scritta la specifica.

Requirements Specification IEEE830

- **correct.**
- **unambiguous**
- **complete**
- **(internally) consistent**
- **it should rank requirements for importance or stability**
- **verifiable**
- **modifiable**
- **traceable**

1. <i>Introduction</i>	3. <i>Specific requirements</i>
1.1 Purpose	3.1 External interface requirements
1.2 Scope	3.1.1 User interfaces
1.3 Definitions, acronyms and abbreviations	3.1.2 Hardware interfaces
1.4 References	3.1.3 Software interfaces
1.5 Overview	3.1.4 Communications interfaces
2. <i>Overall description</i>	3.2 Functional requirements
2.1 Product perspective	3.2.1 User class 1
2.2 Product functions	3.2.1.1 Functional requirement 1.1
2.3 User characteristics	3.2.1.2 Functional requirement 1.2
2.4 Constraints	...
2.5 Assumptions and dependencies	3.2.2 User class 2
2.6 Requirements subsets	...
3. <i>Specific requirements</i>	3.3 Performance requirements
	3.4 Design constraints
	3.5 Software system attributes
	3.6 Other requirements



- Requirements identification (number, goal-hierarchy numbering, version information, attributes) → tracciabilità;
- Requirements traceability:
 - Where is requirement implemented?
 - Do we need this requirement?
 - Are all requirements linked to solution elements?
 - What is the impact of this requirement?
 - Which requirement does this test case cover?

7.14 Requirements specification techniques

Which notation?

using natural language?

- **noise**
- **silence**
- **overspecification**
- **contradictions**
- **ambiguity**
- **forward references**
- **wishful thinking (pia illusione)**



alternative given by Meyer is to first describe and analyze the problem using some formal notation and then translate it back into natural language

7.15 Types of requirements

- **functional requirements:** the system services which are expected by the users of the system;
- **non-functional (quality) requirements:** the set of constraints the system must satisfy and the standards which must be met by the delivered system.
 - speed;
 - size
 - facilità d'uso;
 - reliability;
 - robustness;
 - portability

7.16 Validation of requirements

- controllo della specifica del requisito per quanto riguarda la correttezza, completezza, coerenza, accuratezza, leggibilità e verificabilità;
- **alcuni aiuti:**
 - procedure dettagliate strutturate;
 - prototypes;
 - develop a test plan;
 - tool support for formal specifications;

8 Modeling

Nel corso di un progetto di sviluppo software, e in particolare durante ingegneria e progettazione dei requisiti, vengono utilizzate molte notazioni di modellazione. Questi vanno da schizzi molto informali di funzioni di sistema o layout dello schermo, a descrizioni altamente formali del comportamento del sistema.

8.1 Specifica dei requisiti

- La descrizione informale di un sistema SW è data in termini di **requisiti** (documento);
- La specifica dei requisiti è una descrizione completa e non ambigua dei requisiti
 - Descrive COSA un sistema software deve fare e non COME deve farlo;
- I requisiti si suddividono in:
 - Requisiti funzionali
 - Requisiti non funzionali
 - Requisiti del processo e manutenzione
- Una specifica va intesa come l' **accordo** tra il produttore di un servizio e il suo committente;
- Il 73% dei progetti SW vengono abbandonati o non rispondono alle aspettative a causa di requisiti errati;
- Le cause del fallimento sono dovute a:
 - difetti iniziali dei requisiti;
 - mancato coinvolgimento dell'utenza;
 - incapacità di gestire le variazioni in corso d'opera dei requisiti stessi;
- Lo sviluppo di una specifica richiede:
 - un'analisi iterativa e cooperativa del problema ⇒ attraverso colloquio con chi ha interesse al sistema (stakeholders)
 - l'impiego di linguaggi formali o semiformali;

8.2 Qualità delle specifiche

- **Chiarezza:** la specifica deve descrivere quanto più chiaramente possibile i termini e le operazioni coinvolte;
- **Non ambiguità:** la specifica non deve generare interpretazioni ambigue;
- **Consistenza:** la specifica non deve contenere contraddizioni;
- **Completezza (interna, esterna)**
 - **interna:** la specifica deve definire ogni concetto nuovo e ogni terminologia usata (definire un **glossario**);
 - **esterna:** la specifica deve essere completa rispetto ai requisiti, ossia deve documentare tutti i requisisti richiesti;
- **Incrementalità:**
- **Comprendibilità:** la specifica, in quanto contratto tra committente e produttore, deve essere intuitiva e comprensibile per il cliente
 - Possibilmente visuale;
 - Il più possibile succinta;

8.3 Linguaggi per la specifica

- **Informali:** Linguaggio naturale;
- **Semi-formali:** Possibilmente grafici (es: UML);
- **Formali:**
 - Formalismi operazionali;
 - Formalismi dichiarativi
 - * Formalismo è una notazione rigorosa;

Diversi modi di specifica

- **Informale:**

Il valore di x sarà tra 1 e 5, finché ad un certo momento diventa 7. In ogni caso non sarà mai negativo.

- **Formale:** (logica temporale)

$$(1 \leq x \leq 5 \cup x=7) \wedge [] x \geq 0$$

- **Grafico:**

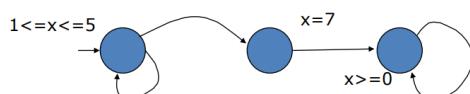


Figure 26: esempio

8.4 Diversi formalismi

- **Formalismi operazionali:**

- definiscono il sistema descrivendone il comportamento (normalmente mediante un modello) come se eseguito da una macchina astratta;
- fornisce una rappresentazione più intuitiva poiché più simile al modo di ragionare della mente umana;
- Permette facilità di realizzazione;
- Permette facilità di validazione;
- Esempi di formalismi operazioni:
 - * Abstract State Machines;
 - * B method;
 - * Z method;
 - * SCR (Software Cost Reduction);
 - * Reti di Petri;

- **Formalismi dichiarativi:**

- definiscono il sistema dichiarando le proprietà che esso deve avere;
- fornisce una rappresentazione che non si presta ad ambiguità, ma è più difficile da comprendere e sviluppare;

- Permette facilità di verifica;
- Esempi di formalismi dichiarativi:
 - * Logica temporale;
 - * Trio;
 - * Algebre dei processi ;

• Esempio: definizione di array ordinato

• Esempio: definizione di ellisse

– **Definizione operazionale**

E' l'insieme dei punti del piano che si ottiene muovendosi in modo che la somma delle distanze tra il punto e due punti fissi p1 e p2 rimanga invariata

– **Definizione operazionale**

Sia a un array di n elementi. Il risultato del suo ordinamento è un array b di n elementi tali che il primo elemento di b è il minimo di a (se diversi elementi hanno lo stesso valore, uno di essi è selezionato); il secondo elemento di b è il minimo dell'array di $n-1$ elementi ottenuto da a rimuovendo il suo elemento minima; e così via fino a che gli n elementi di a sono stati rimossi

– **Definizione dichiarativa**

$$ax^2 + by^2 + c = 0$$

E' l'insieme dei punti del piano che soddisfano l'equazione

– **Definizione dichiarativa**

Il risultato dell'ordinamento di un array a è un array b che è una permutazione di a ed è ordinato

Figure 27: esempio

8.5 Choosing a modeling notation

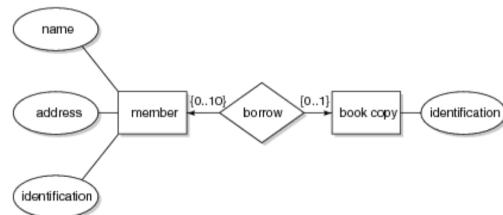
- The user → a document which speaks his language;
 - Natural language + terms from the domain;
- software engineer → often use some formal/informal language.
 - A requirements specification phrased in such a formal language may be checked using formal techniques, for instance with regard to consistency and completeness;

8.6 System Modeling Techniques

- Classic modeling techniques:
 - Entity-relationship modeling (ER);
 - Finite state machines;
 - Data flow diagrams;
 - CRC cards;
- Object-oriented modeling: variety of UML diagrams;

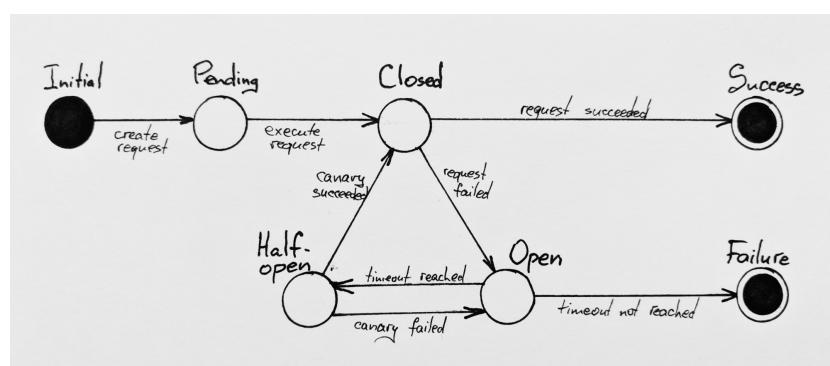
8.6.1 Entity-Relationship Modeling (ER)

- **entity**: distinguishable object of some type;
- **entity type**: type of a set of entities;
- **attribute value**: piece of information (partially) describing an entity;
- **attribute**: properties of an entity;
- **relationship**: association between two or more entities;

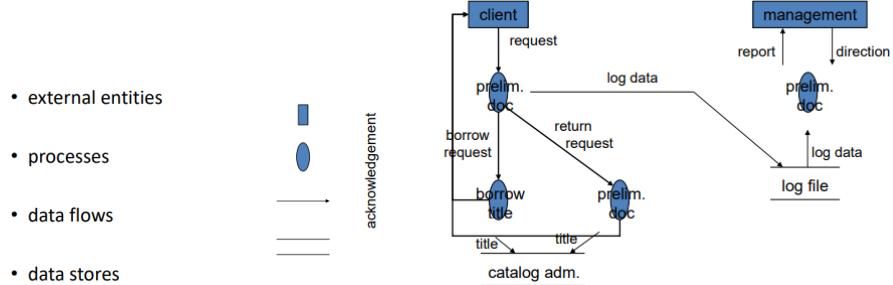


8.6.2 Finite state machines

- Models a system in terms of (a finite number of) states, and transitions between those states;
 - Each state is a bubble;
 - Each transition is a labeled arc from one state to another;



8.6.3 Data flow diagrams



8.6.4 CRC: Class, Responsibility, Collaborators

Class Reservations	Collaborators • Catalog • User session
Responsibility • Keep list of reserved titles • Handle reservations	

8.7 Intermezzo: what is an object?

- **Modeling viewpoint:** model of part of the world;
- **Philosophical viewpoint:** existential abstractions;
- **Software engineering viewpoint:** data abstraction;
- **Implementation viewpoint:** structure in memory;
- **Formal viewpoint:** state machine;

8.8 Objects and attributes

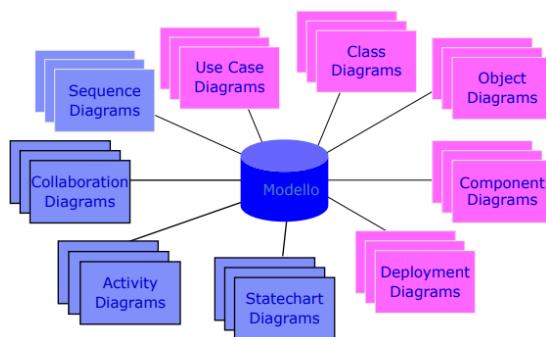
- Object is characterized by a set of attributes;
 - In ERM, attributes denote intrinsic properties; they do not depend on each other, they are descriptive;
 - In ERM, relationships denote mutual properties, such as the membership of a person of some organization;
 - In UML, these relationships are called associations;
 - Formally, UML does not distinguish attributes and relationships; both are properties of a class;
-
- **State:** set of attributes of an object;
 - **Class:** set of objects with the same attributes;
 - **Individual object:** instance;
 - Behavior is described by services, a set of responsibilities;
 - Service is invoked by sending a message;

8.9 Relations between objects

- **Specialization-generalization:** is a
 - es: A dog is an animal;
 - Expressed in hierarchy;
 - * Common attributes are defined at a higher level in the object hierarchy, and inherited by child nodes;
- **Whole-part:** has
 - es: A dog has legs;
 - **Aggregation** of parts into a whole;
- **Member-of:** has
 - es: A soccer team has players;
 - Relation between a set and its members (usually not transitive)

8.10 Unified Modeling Language (UML)

- UML: Linguaggio di Modellazione Unificato;
- Definisce una:
 - notazione grafica;
 - complesso di viste organizzate in diagrammi;
 - * strutturali e comportamentali;
 - semantica descritta in prosa → descrizione ridondante, frammentaria, ambigua;
- UML has 13 diagram types
 - I diagrammi statici rappresentano la struttura statica;
 - Dynamic diagrams show what happens during execution;



8.10.1 Viste in UML

- **Use-Case View:** Mostra le funzionalità che il sistema dovrebbe fornire così come sono percepite da attori esterni (black box view);
- **Logical View:** Analizza l'interno del sistema (struttura statica e dinamica) e descrive come le funzionalità sono fornite (white box view);
- **Component View:** Mostra l'organizzazione e le dipendenze delle componenti computazionali del sistema;
- **Deployment View:** Descrive l'allocazione delle parti del sistema software in una architettura fisica;

- **Use-Case View**
 - Diagramma dei casi d'uso
- **Logical View**
 - Diagramma delle classi e degli oggetti
 - Diagrammi di interazione
 - Diagrammi di sequenza e diagrammi di collaborazione
 - Macchine di stato
 - Diagramma di attività
- **Component View**
 - Diagramma delle componenti
- **Deployment View**
 - Diagramma di dislocamento

Static diagrams:

- Class
- Component
- Deployment
- Interaction overview
- Object
- Package

Dynamic diagrams:

- Activity
- Communication
- Composite structure
- Sequence
- State machine
- Timing
- Use case

9 Software Architecture

L'architettura del software riguarda la struttura su larga scala dei sistemi software. Questa struttura di grandi dimensioni riflette le prime ed essenziali scelte progettuali. Questo processo decisionale comporta la negoziazione e l'equilibrio tra funzionalità e qualità esigenze da un lato e possibili soluzioni dall'altro. Software Architecture non è una fase che segue strettamente l'ingegneria dei requisiti, ma i due sono intrecciati.

software architecture is synonymous with global design

- During the design phase, the system is decomposed into a number of interacting **components**;
- The top-level decomposition of a system into major components together with a characterization of how these components interact, is called its software architecture.

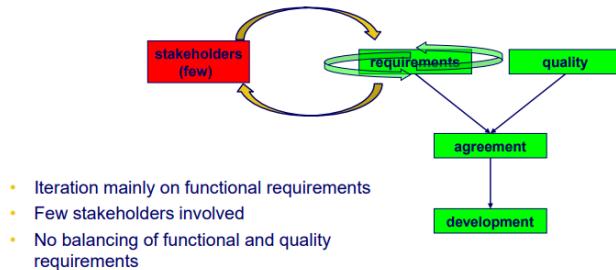


Figure 28: Pre-architecture life cycle

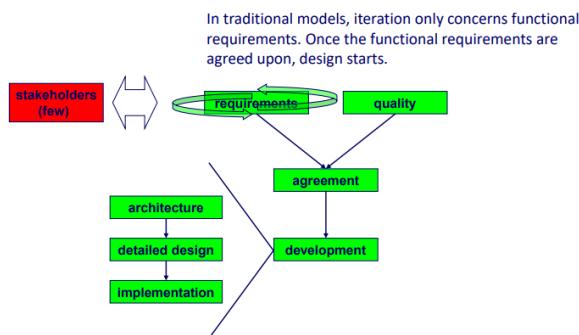
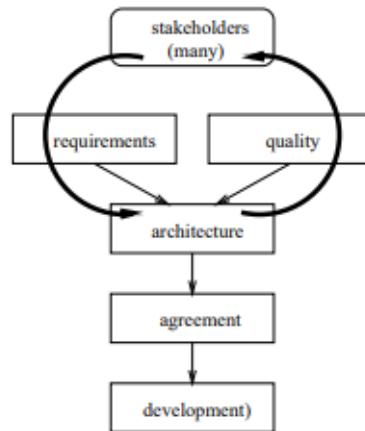


Figure 29: Adding architecture, the easy way

9.1 Architecture in the life cycle

In process models that include a software architecture phase, iteration involves both functional and quality requirements.

- Iteration on both functional and quality requirements;
- Many stakeholders involved;
- Balancing of functional and quality requirements;

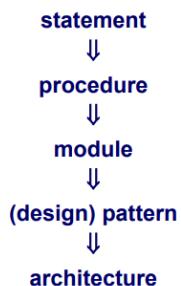


9.2 Why Is Architecture Important?

- Architecture is the vehicle for stakeholder communication;
- Architecture manifests the earliest set of design decisions;
 - Constraints on implementation;
 - Dictates organizational structure;
 - Inibisce o abilita gli attributi di qualità;
- Architecture is a transferable abstraction of a system;
 - Product lines share a common architecture;
 - Allows for template-based development;

9.3 Software architectur, definitions

- **Def:** The architecture of a software system defines that system in terms of computational components and interactions among those components;
- **Def:** The software architecture of a system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.
 - Important **issues** raised in this definition:
 - * *externally visible* (observable) properties of components.
 - * The definition does not include:
 - the process;
 - rules and guidelines;
 - architectural styles.



- **Def (IEEE):** L'architettura è l'organizzazione fondamentale di un sistema incarnato nelle sue componenti, le loro relazioni reciproche e con il ambiente e i principi che ne guidano la progettazione e l'evoluzione;

9.4 Other points of view

- Architecture is high-level design;
- Architecture is overall structure of the system;
- Architecture is the structure, including the principles and guidelines governing their design and evolution over time;
- Architecture is components and connectors;

9.5 Architectural Structures

- Includes:
 - module structure;
 - conceptual, or logical structure;
 - process, or coordination structure;
 - physical structure;
 - uses structure;
 - calls structure;
 - data flow;
 - control flow;
 - class structure;

9.6 Software Architecture & Quality

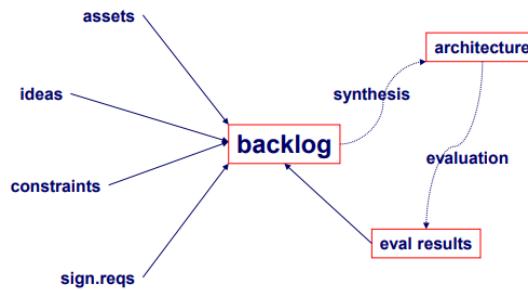
- The notion of quality is central in software architecting;
- Some qualities are observable via execution: performance, security, availability, functionality, usability;
- And some are not observable via execution: modifiability, portability, reusability, integrability, testability;

9.7 Attribute-Driven Design (ADD)

- a top-down decomposition process;
- Choose module to decompose;
- Refine this module:
 - choose architectural drivers (quality is driving force);
 - choose pattern that satisfies drivers;
 - apply pattern;
- Repeat steps;

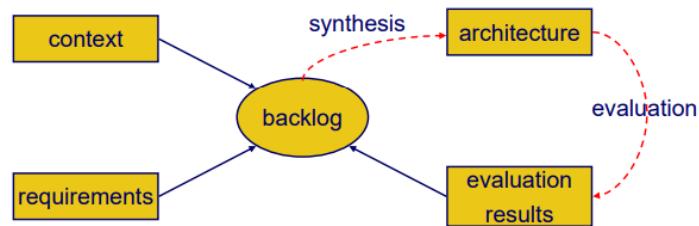
9.8 Generalized model

- Understand problem;
- Evaluate solution;
- Solve it;



9.9 Global workflow in architecture design

- **Backlog:** A backlog is a list of tasks required to support a larger strategic plan. The product team agrees to work on these projects next. Typical items on a product backlog include:
 - user stories;
 - changes to existing functionality;
 - bug fixes.
- Three inputs to the **backlog**:
 1. **Context:** idee che l'architetto può avere, risorse disponibili che possono essere utilizzate, vincoli impostati e così via;
 2. **requirements**
 3. **evaluation:** Il risultato di questa trasformazione viene valutato (di solito in modo piuttosto informale) e questa valutazione può in volta modificare il contenuto del backlog;



9.10 Design issues, options and decisions

- A designer is faced with a series of design issues:
 - These are sub-problems of the overall design problem;
 - Issues and options are not independent;
 - the designer makes a design decision to resolve each issue:
 - * This process involves choosing the best option from among the alternatives;
 - **Technical and non-technical issues** and options are intertwined
 - * Architects deciding on the type of database versus
 - * Management deciding on new strategic partnership or
 - * Management deciding on budget

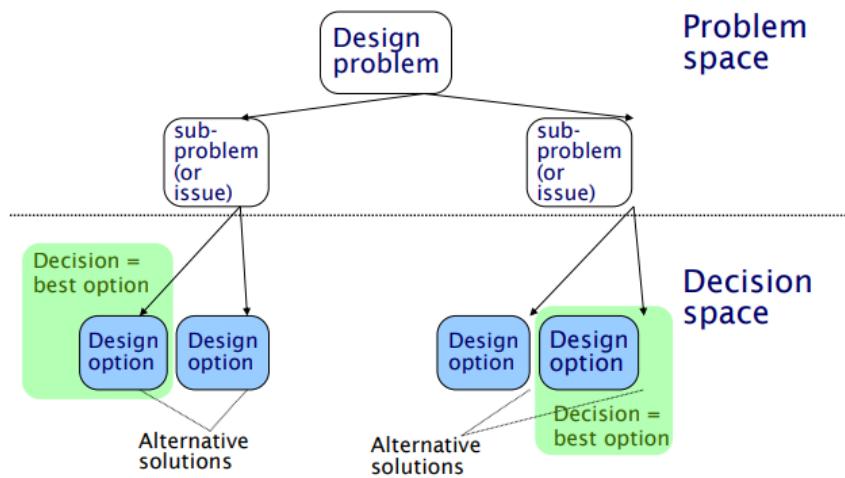
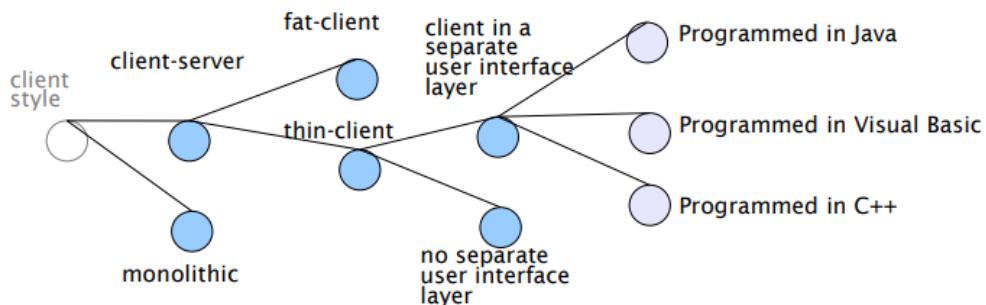


Figure 30: Taking decisions

- **Decision space:** The space of possible designs that can be achieved by choosing different sets of alternatives.
 - Types of decisions:
 - * **Implicit, undocumented:** Conoscenza inconsapevole, tacita, ovvia;
 - * **Explicit, undocumented:** Vaporizes over time;
 - * **Explicit, explicitly undocumented:** Tactical, personal reasons;
 - * **Explicit, documented**
 - Why is documenting design decisions important?
 - * Prevents repeating (expensive) past steps;
 - * Explains why this is a good architecture;
 - * Emphasizes qualities and criticality for requirements/goals
 - * Provides context and background;
 - Uses of design decisions:
 - * Identify key decisions for a stakeholder → Get a rationale, Validate decisions against reqs;
 - * Cleanup the architecture, identify important architectural drivers → best decision;



Element	Description
Issues	Design issues being addressed by this decision
Decision	The decision taken
Status	The status of the decision, e.g. pending, approved
Assumptions	The underlying assumptions about the environment in which the decision is taken
Alternatives	Alternatives considered for this decision
Rationale	An explanation of why the decision was chosen
Implications	Implications of this decision, such as the need for further decisions or requirements
Notes	Any additional information one might want to capture

Figure 31: Elements of a design decision

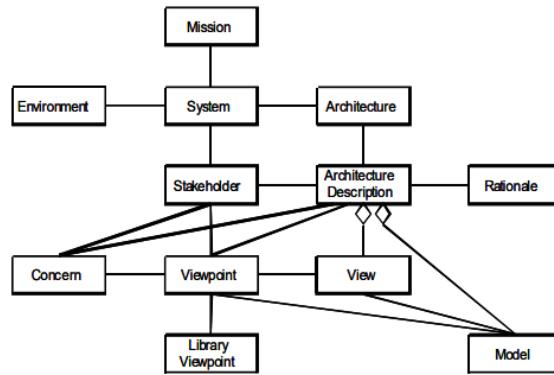
9.11 Software design in UML

- Who can read those diagrams?
- Which type of questions do they answer?
- Do they provide enough information?
- Different representations;
- For different people;
- For different purposes;
- Queste rappresentazioni sono sia descrittive che prescrittive;

9.12 IEEE model for architectural descriptions

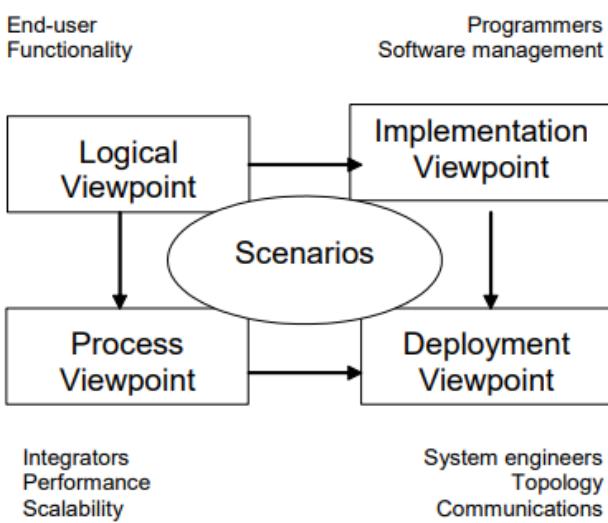
- **System stakeholder (parti interessate):** un individuo, una squadra o organizzazione (o classi di questa) con interessi o preoccupazioni relative a, un sistema;
- **View:** a representation of a whole system from the perspective of a related set of concerns;
- **Viewpoint:** Un punto di vista stabilisce:
 - gli scopi;
 - Stakeholders addressed;

- Preoccupazioni affrontate;
- le tecniche o i metodi impiegati nella costruzione di una vista;



9.13 Kruchten's 4+1 view model

- Many organizations have developed their own set of library viewpoints;
- A well-known set of library viewpoints is known as the '4 + 1 model' (Kruchten, 1995);



- **The logical viewpoint:** supports the functional requirements, that is the services the system should provide to its end users → solitamente classi;
- **Process Viewpoint:** Addresses concurrent aspects at runtime (tasks, threads, processes and their interactions)
 - It takes into account some non functional requirements, such as:
 - * performance;
 - * system availability;
 - * concurrency and distribution;
 - * system integrity;
 - * fault-tolerance;
- **Deployment Viewpoint:** definisce come i vari elementi identificati nei logical viewpoint, process viewpoint e di implementatio viewpoint, le reti, i processi, le attività e gli oggetti devono essere mappati sul vari nodi.
 - It takes into account some non functional requirements, such as:
 - * system availability;
 - * reliability (fault-tolerance);
 - * performance;
 - * scalability;
- **Implementation Viewpoint:** si concentra sull'organizzazione dei moduli software reali nell'ambiente di sviluppo software.
 - The software is packaged in small chunks-program libraries or subsystems-that can be developed by one or more developers
- **Scenario Viewpoint:** The scenario viewpoint consists of a small subset of important scenarios (ex: use cases) to show that the elements of the four viewpoints work together seamlessly.
 - Questo viewpoint è ridondante con gli altri (da cui il "+1"), ma svolge due ruoli critici:
 - * it acts as a driver to help designers discover architectural elements during the architecture design;
 - * it validates and illustrates the architecture design, both on paper and as the starting point for the tests of an architectural prototype;

9.14 Architectural views from Bass et al

view = representation of a structure

- **Module views**

- Module is unit of implementation;
- Decomposition, uses, layered, class;

- **Component and connector (C & C) views**

- These are runtime elements;
- Process (communication), concurrency, shared data (repository), client-server;

- **Allocation views**

- Relationship between software elements and environment;
- Work assignment, deployment, implementation;

Examples of viewpoint

- **Module viewpoints** give a static view of the system. They are usually depicted in the form of box-and-line diagrams where the boxes denote system components and the lines denote some relation between those components.
- **Component-and-connector** viewpoints give a dynamic view of the system, i.e. they describe the system in execution. Again, they are usually depicted as box-and-line diagrams.
- **Allocation** viewpoints give a relation between the system and its environment, such as who is responsible for which part of the system.

9.14.1 Module views

- **Decomposition:** units are related by “is a submodule of”, larger modules are composed of smaller ones;
- **Uses:** relation is “uses” (calls, passes information to, etc). Important for modifiability;
- **Layered:** is special case of uses, layer n can only use modules from layers \downarrow n;
- **Class:** generalization, relation “inherits from”;

9.14.2 Component and connector views

- **Process:** units are processes, connected by communication or synchronization;
- **Concurrency:** : to determine opportunities for parallelism (connector = logical thread);
- **Shared data:** shows how data is produced and consumed;
- **Client-server:** cooperating clients and servers

9.14.3 Allocation views

- **Deployment:** how software is assigned to hardware elements;
- **Implementation:** how software is mapped onto file structures;
- **Work assignment:** who is doing what;

9.15 Architectural styles

- An architectural style is a description of component and connector types and a pattern of their runtime control and/or data transfer.

Examples:

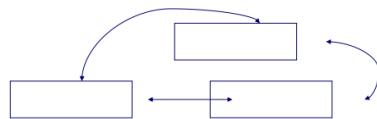
- main program with subroutines
- data abstraction
- implicit invocation
- pipes and filters
- repository (blackboard)
- layers of abstraction

9.16 Using schemas: pattern, framework, idioms

- **Design patterns** are collections of a few modules (or, in object-oriented circles, classes) that are often used in combination, and which together provide a useful abstraction. A design pattern is a recurring solution to a standard problem.
- **An application framework** is a partially complete system which needs to be instantiated to obtain a complete system.
- **Idiom:** is a low-level pattern, specific to some programming language;

9.17 Components and Connectors

- Components are connected by connectors;
- They are the building blocks with which an architecture can be described;
- No standard notation has emerged yet;



9.17.1 Types of components

- **computational:** does a computation of some sort.
 - ex: function, filter
- **memory:** maintains a collection of persistent data
 - ex: data base, file system, symbol table
- **manager:** contains state + operations. State is retained between invocations of operations.
 - ex: adt, server
- **controller:** governs time sequence of events.
 - ex: control module, scheduler

9.18 Types of connectors

- **procedure call (including Remote Procedure Call):** Control is transferred to another component;
- **data flow (e.g. pipes):** Data are transferred from one component to another
- **implicit invocation:** A component require a service when an event occur
- **message passing:** Both async and sync

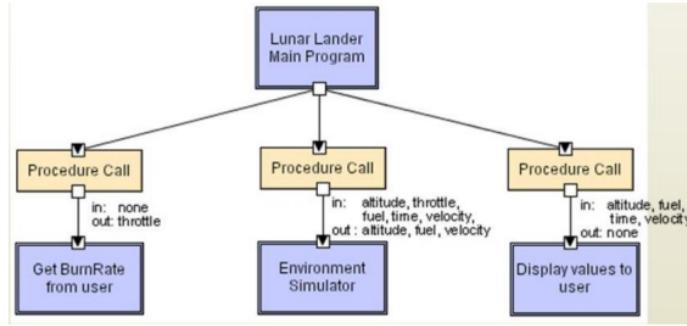
- shared data (e.g. blackboard or shared data base)
- Instantiation

9.19 Framework for describing architectural styles

- **problem:** tipo di problema affrontato dallo stile. Caratteristiche dei requisiti guida il designer nella scelta di uno stile particolare;
- **context:** caratteristiche dell'ambiente che vincolano il designer, i requisiti imposti dallo stile;
- **solution:** in terms of components and connectors (choice not independent), and control structure (order of execution of components);
- **variants**
- **examples**

9.20 Main-program-with-subroutines style

- **problem:** hierarchy of functions; result of functional decomposition, single thread of control;
- **context:** language with nested procedures;
- **solution:**
 - **system model:** modules in a hierarchy, may be weak or strong, coupling/cohesion arguments
 - **components:** modules with local data, as well as global data
 - **connectors:** procedure call
 - **control structure:** single thread, centralized control: main program pulls the strings
- **variants:** OO versus non-OO;
- **example:**



9.21 Abstract-data-type style

- **problem:** identify and protect related bodies of information. Data representations likely to change;
- **context:** OO-methods which guide the design, OO-languages which provide the class-concept;
- **solution:**
 - **system model:** component has its own local data (= secret it hides);
 - **components:** managers (servers, objects, adt's);
 - **connectors:** procedure call (message);
 - **control structure:** single thread, usually; control is decentralized;
- **variants:** caused by language facilities

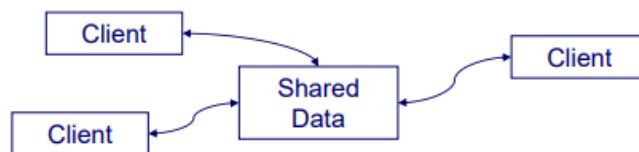
9.22 Pipes-and-filters style

- **problem:** independent, sequential transformations on ordered data. Usually incremental, Ascii pipes.
- **context:** serie di trasformazioni incrementali. Le funzioni del sistema operativo trasferiscono i dati tra i processi. Difficile la gestione degli errori;
- **solution:**
 - **system model:** continuous data flow; components incrementally transform data;

- **components:** filters for local processing;
- **connectors:** data streams (usually plain ASCII);
- **control structure:** data flow between components; component has own flow;
- **variants:** From pure filters with little internal state to batch processes;

9.23 Repository style

- **problem:** gestire informazioni riccamente strutturate, da manipolare in molti modi differenti. I dati sono longevi.
- **context:** dati condivisi su cui agire da più clienti;
- **solution:**
 - **system model:** centralized body of information. Independent computational elements;
 - **components:** one memory, many computational;
 - **connectors:** direct access or procedure call;
 - **control structure:** varies, may depend on input or state of computation;
- **variants:** traditional data base systems, compilers, blackboard systems;



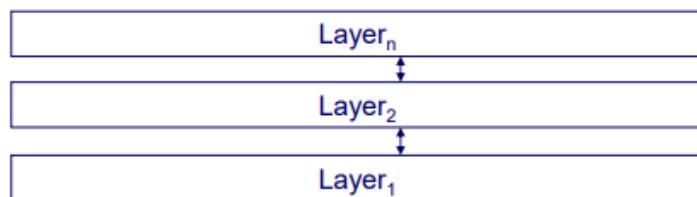
9.24 Layered style

- **problem:** distinct, hierarchical classes of services. “Concentric circles” of functionality;
- **context:** a large system that requires decomposition (e.g., virtual machines, OSI model);

- **solution:**

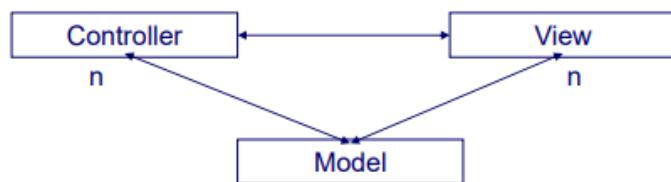
- **system model:** hierarchy of layers, often limited visibility;
- **components:** collections of procedures (module);
- **connectors:** (limited) procedure calls;
- **control structure:** single or multiple threads;

- **variants:** relaxed layering;



9.25 Model-View-Controller (MVC) style

- **problem:** la separazione dell’interfaccia utente dall’applicazione è auspicabile a causa dell’interfaccia utente prevista;
- **context:** applicazioni interattive con un’interfaccia utente flessibile;
- **solution:**
 - **system model:** UI (View and Controller Component(s)) is decoupled from the application (Model component);
 - **components:** collections of procedures (module);
 - **connectors:** procedure calls;
 - **control structure:** single thread;
- **variants:** Document-View;



10 Software Design

La progettazione del software riguarda la scomposizione di un sistema nelle sue parti. Un buon design è la chiave per l'implementazione e l'evoluzione di successo di un sistema. Un certo numero di principi guida per questa scomposizione aiutano a ottenere progetti di qualità.

Skip requirements engineering and design phases; start writing code

- programmer's approach:

- Design is a waste of time;
- We need to show something to the customer real quick;
- We expect or know that the schedule is too tight;

→ *The longer you postpone coding, the sooner you'll be finished*

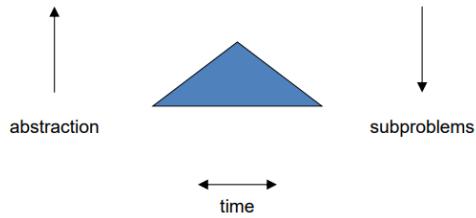
- There is no definite formulation;
- Solutions are not simply true or false;
- Ogni problema malvagio è sintomo di un altro problema;
- There is an interaction between requirements engineering, architecting, and design;

10.1 Design principles

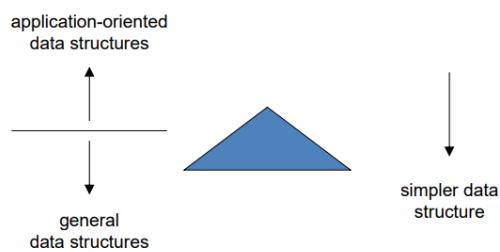
- **Abstraction;**
- **Modularity, coupling and cohesion;**
- **Information hiding;**
- **Limit complexity;**
- **Hierarchical structure;**

10.1.1 Abstraction

- **procedural abstraction:** conseguenza naturale dei passi di sviluppo;
- **refinement:** nome della procedura denota sequenza di azioni;

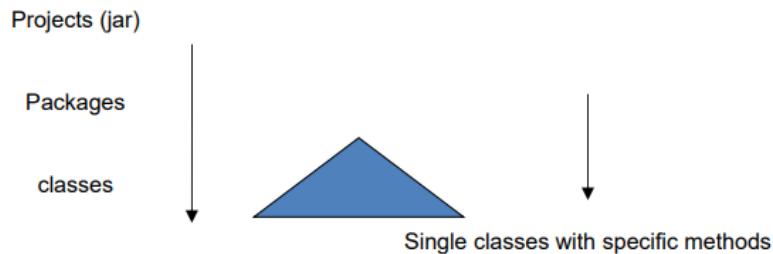


- **data abstraction:** finalizzato a trovare una gerarchia nei dati;



- Ex: Abstraction in OO projects Java

- Separate your application in modules (or projects) – packages – classes

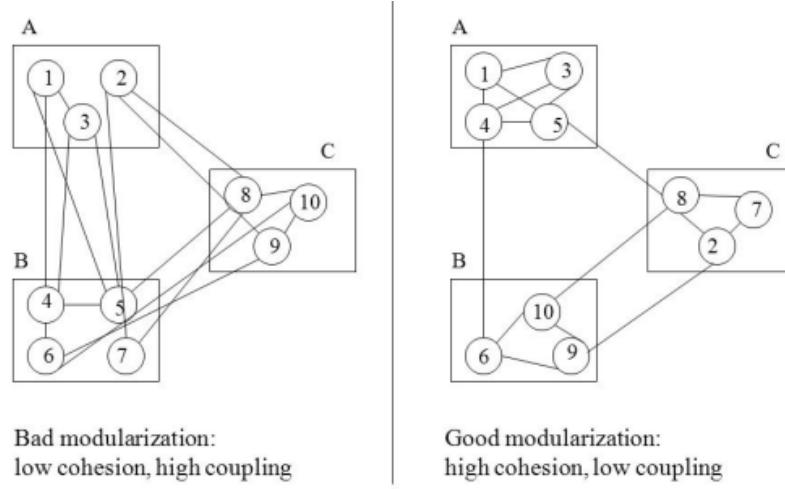


10.1.2 Modularity

- structural criteria which tell us something about individual modules and their interconnections;
- **cohesion and coupling**;
- **cohesion:** the glue that keeps a module together;
 - La coesione è una misura del grado i cui elementi del modulo sono funzionalmente correlati. È il grado in cui tutti gli elementi volti allo svolgimento di un unico compito sono contenuti nel componente.
 - Una buona progettazione del software avrà un'elevata coesione.
 - tipi di coesione:
 - * **coincidental cohesion:** Con coesione casuale, gli elementi sono raggruppati in moduli in modo casuale. Non vi è alcuna relazione significativa tra i elementi;
 - * **Logical cohesion:** Con la coesione logica, gli elementi realizzano compiti che sono logicamente correlati. Un esempio è un modulo che contiene tutte le routine di input. Queste routine non si richiamano e non trasmettono informazioni a l'un l'altro. La loro funzione è molto simile;
 - * **Temporal cohesion:** The elements are independent but they are activated at about the same point in time;
 - * **Procedural cohesion:** Un modulo presenta coesione procedurale se è costituito da un numero di elementi che devono essere eseguiti in un determinato ordine. Ad esempio, un modulo potrebbe dover prima leggere un dato, quindi cercare una tabella e infine stampa un risultato;
 - * **Communicational cohesion:** Questo tipo di coesione si verifica se gli elementi di un modulo opera sugli stessi dati (esterni). Ad esempio, un modulo può leggere alcuni dati da un disco, eseguire determinati calcoli su quei dati e stampare il risultato.
 - * **Sequential cohesion:** La coesione sequenziale si verifica se il modulo è costituito da una sequenza di elementi in cui l'output di un elemento funge da input per il elemento successivo;

- * **Functional cohesion:** In un modulo che mostra coesione funzionale tutti gli elementi contribuire ad un'unica funzione;
- Determinate il tipo di coesione:
 - * descrive il motivo del modulo in una frase;
 - * se la frase contiene una virgola o più di un verbo probabilmente ha più di una funzione: coesione logica o comunicativa;
 - * se la frase contiene parole come "first", "then", "after" → coesione temporale;
 - * se il verbo non è seguito da un oggetto specifico → probabilmente coesione logica;
 - * parole come "startup", "initialize" → coesione temporale
- **coupling:** la forza della connessione tra i moduli;
 - L'accoppiamento è la misura del grado di interdipendenza tra i moduli.
 - Un buon software avrà un basso accoppiamento.
 - Tipi di accoppiamenti:
 - * content coupling;
 - * common coupling;
 - * external coupling;
 - * control coupling;
 - * stamp coupling;
 - * data coupling;
 - **data coupling** assume scalari o array, non record;
 - **control coupling** assume passaggio di dati scalari;
 - nowadays:
 - * moduli possono passare strutture dati complesse;
 - * moduli possono consentire l'accesso a dati di altri moduli (così ci sono diversi livelli di visibilità);
 - * l'accoppiamento non deve essere commutativo (A può essere accoppiato a B, mentre B non è accoppiato a A);
- forte coesione e debole accoppiamento → semplici interfacce
 - comunicazione più semplice;

- simpler correctness proofs;
- changes influence other modules less often;
- reusability increases;
- comprehensibility improves;



10.1.3 Information hiding

- each module has a secret;
- design involves a series of decision: for each such decision, wonder who needs to know and who can be kept in the dark;
- information hiding is strongly related to:
 - **abstraction:** if you hide something, the user may abstract from that fact;
 - **coupling:** the secret decreases coupling between a module and its environment
 - **cohesion:** the secret is what binds the parts of the module together;

10.1.4 Complexity

- In senso molto generale, la complessità di un problema si riferisce al quantità di risorse necessarie per la sua soluzione;

- We may try to determine complexity in this way by measuring, say, the time needed to solve a problem. This is called an **external attribute**: we are not looking at the entity itself (the problem), but at how it behaves.
- Nel presente contesto, la complessità si riferisce agli attributi del software che influiscono sullo sforzo necessario per costruire o modificare un software.
 - These are **internal attributes**: they can be measured purely in terms of the software itself.
- measure certain aspects of the software:
 - lines of code;
 - ifstatements;
 - depth of nesting;
- utilizzare questi numeri come criterio per valutare un progetto o come guida per il design;
- **interpretation**: higher value → higher complexity → more effort required (= worse design);
- two kinds:
 - **intra-modular**: inside one module;
 - * attributes of a single module;
 - * two classes:
 - measures based on size;
 - measures based on structure;
 - **inter-modular**: between modules;

- **Sized-based complexity measures**

- counting lines of code:
 - * differences in verbosity;
 - * differences between programming languages;
- **software science:** essentially counting operators and operands
 - * n_1 : number of unique operators
 - The set of operators includes the arithmetic and Boolean operators, as well as separators (such as a semicolon between adjacent instructions) and (pairs of) reserved words.
 - * n_2 : number of unique operands
 - The set of operands contains the variables and constants used.
 - * N_1 : total number of operators
 - * N_2 : total number of operators

Example program	operator	# of Occurrences
<pre>public static void sort(int x []) { for (int i=0; i < x.length-1; i++) { for (int j=i+1; j < x.length; j++) { if (x[i] > x[j]) { int save=x[i]; x[i]=x[j]; x[j]=save } } } }</pre>	<pre>public sort() int [] [] for (++) if () = < ... n₁=17</pre>	<pre>1 1 4 7 4 4 2 1 5 2 N₁=39</pre>

Other software science formulas	operand	# of occurrences
<ul style="list-style-type: none"> size of vocabulary: $n = n_1 + n_2$ program length: $N = N_1 + N_2$ volume: $V = N \log_2 n$ level of abstraction: $L = V^*/V$ approximation: $L' = (2/n_1)(n_2/N_2)$ programming effort: $E = V/L$ estimated programming time: $T' = E/18$ estimate of N: $N' = n_1 \log_2 n_2 + n_2 \log_2 n_1$ for this example: $N = 68, N' = 89, L = .015, L' = .028$ 	 <pre>x length i j save 0 1</pre>	<pre>9 2 7 6 2 1 2 n₂=7</pre>

- * empirical studies: reasonably good fit;
- * critique:
 - explanations are not convincing;
 - is aimed at coding phase only;

- **Structure-based measures**

- based on:
 - * control structures;
 - * data structures;
 - * or both;
- example complexity measure based on data structures: numero medio di istruzioni tra successive riferimenti a una variabile;
- best known measure is based on the control structure: *McCabe's cyclomatic complexity*;
- **McCabe's cyclomatic complexity**
 - * La complessità ciclomatica di una sezione del codice sorgente è il numero percorsi indipendenti al suo interno;
 - * The complexity M is then defined as:

$$M = E - N + 2P$$

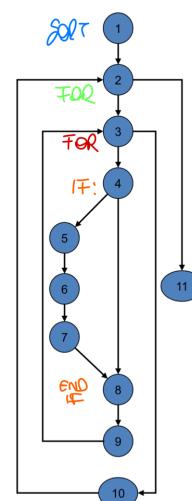
- E = il numero di archi del grafo
- N = il numero di nodi del grafo
- P = il numero di componenti collegati
- * We will use

$$M = E - N + P + 1 \quad \text{count each subgraph as separated}$$

Example program

```

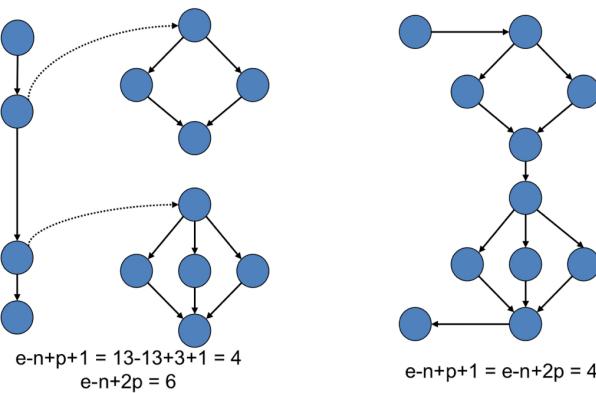
1 public static void sort(int x []) {
2   for (int i=0; i < x.length-1; i++) {
3     for (int j=i+1; j < x.length; j++) {
4       if (x[i] > x[j]) {
5         int save=x[i];
6         x[i]=x[j]; x[j]=save;
7       }
8     }
9   }
10 }
```



$$E = 13 \quad N = 11 \quad P = 1$$

$$CV = E - N + P + 1 = 4$$

Note: $CV = e-n+p+1$, $CV \neq e-n+2p$



- **Intra-modular complexity measures, summary**

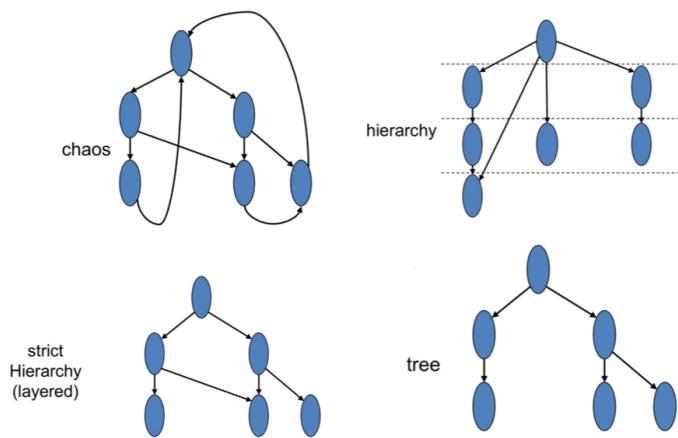
- for small programs, the various measures correlate well with programming time;
- complexity measures are not very context sensitive;
- it might help to look at the complexity density instead;

- **System structure: inter-module complexity**

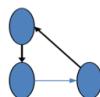
- looks at the complexity of the dependencies between modules;
- draw modules and their dependencies in a graph;
- then the arrows connecting modules may denote several relations, such as:
 - * A contains B;
 - * A precedes B;
 - * A uses B;

– The uses relation:

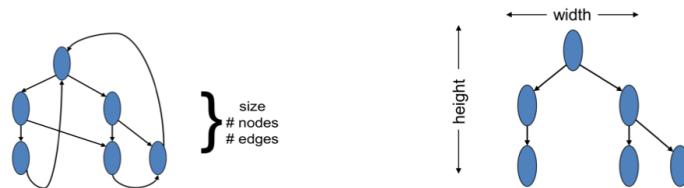
- * In a well-structured piece of software, the dependencies show up as procedure calls;
- * pertanto, questo grafico è noto come **call-graph**;
- * possible shapes of this graph:
 - chaos (directed graph);
 - hierarchy (acyclic graph);
 - strict hierarchy (layers);
 - tree;



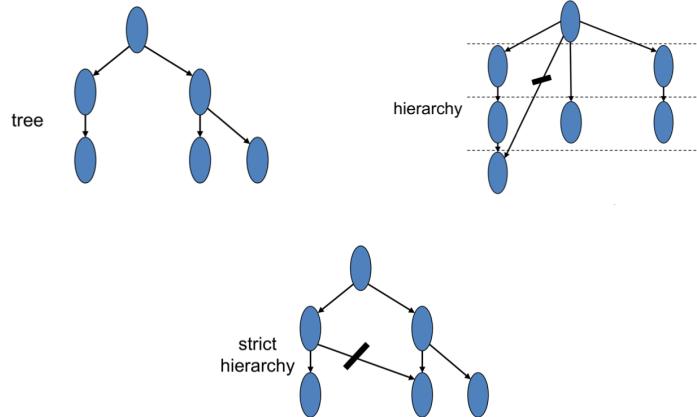
Problems with cycles



Measurements



Deviation from a tree



- Tree impurity metric

- * complete graph with n nodes has $\frac{n \cdot (n-1)}{2}$ edges;
- * a tree with n nodes has $(n - 1)$ edges;
- * tree impurity for a graph with n nodes and e edges:

$$m(G) = \frac{2(e - n + 1)}{(n - 1)(n - 2)}$$

- * Desirable properties of any tree impurity metric:
 - $m(G) = 0$ if and only if G is a tree
 - $m(G_1) > m(G_2)$ if $G_1 = G_2 + \text{edges}$

- Object-oriented metrics

- WMC: Weighted Methods per Class
- DIT: Depth of Inheritance Tree
- NOC: Number Of Children
- CBO: Coupling Between Object Classes
- RFC: Response For a Class
- LCOM: Lack of COhesion of a Method

- Weighted Methods per Class

- * measure for size of class
- * $WMC = \sum c(i) \quad i = 1, \dots, n$ (number of methods)
- * $c(i) \rightarrow$ complexity of method i
- * mostly, $c(i) = 1$

- **Depth of Class in Inheritance Tree**
 - * DIT = distance of class to root of its inheritance tree;
 - * DIT is somewhat language-dependent;
 - * **widely accepted heuristic:** strive for a forest of classes, a collection of inheritance trees of medium height;
- **Number Of Children**
 - * NOC: counts immediate descendants
 - * higher values NOC are considered bad:
 - possibly improper abstraction of the parent class
 - also suggests that class is to be used in a variety of settings
- **Coupling Between Object Classes**
 - * two classes are coupled if a method of one class uses a method or state variable of another class;
 - * CBO = conteggio di tutte le classi a cui una data classe è accoppiata
 - * high values: something is wrong
 - *
- **Package coupling**
 - * L'**afferent coupling** (C_a) di un pacchetto P è il numero di altri pacchetti a cui fanno riferimento alle classi di P (tramite eredità o associazioni). Indica la dipendenza di un pacchetto all'interno di un ambiente.

da package → a classes

- * L'**efferent coupling** (C_e) è il numero di pacchetti a cui fanno riferimento le classi all'interno di P. Questo indica la dipendenza dell'ambiente rispetto a un pacchetto.

da classes → a package

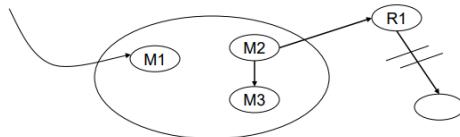
- * Adding these numbers together results in a total coupling measure of a package P
- * the ratio \mathbb{I} indica la relativa dipendenza dell'ambiente rispetto a P e il numero totale di dipendenze tra P e l'ambiente ⇒ si chiama **instability**

$$\mathbb{I} = \frac{C_e}{(C_e + C_a)}$$

- If $C_e = 0$, P does not depend at all on other packages and $\mathbb{I} = 0$ as well
- if $C_a = 0$, P only depends on other packages and no other package depends on P $\rightarrow \mathbb{I} = 1$
- \mathbb{I} thus can be seen as an instability measure for P. Larger values of \mathbb{I} denote a larger instability of the package.

– Response For a Class

- * RFC measures the “immediate surroundings” of a class
- * RFC = size of the “response set”
- * response set = $\{M\} \cup \{R_i\}$



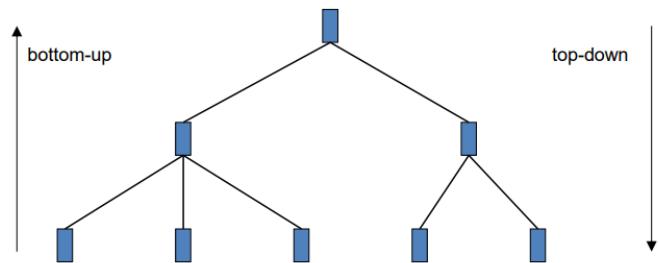
– Lack of Cohesion of a Method

- * **cohesion** = glue that keeps the module (class) together
- * if all methods use the same set of state variables: OK, & that is the glue
- * se alcuni metodi utilizzano un sottoinsieme delle variabili di stato, e altri usano un altro sottoinsieme, la classe manca di coesione
- * LCOM = numero di insiemi disgiunti di metodi in una classe
- * two methods in the same set share at least one state variable

10.2 Design methods

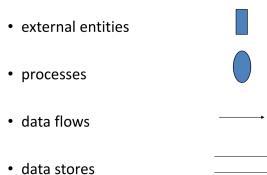
10.2.1 Functional decomposition

- Extremes: bottom-up and top-down
- Not used as such; design is not purely rational:
 - clients do not know what they want
 - changes influence earlier decisions
 - people make errors
 - projects do not start from scratch



10.2.2 Data flow design

- two-step process:
 - **Structured Analysis (SA)** resulting in a logical design, drawn as a set of data flow diagrams
 - **Structured Design (SD)** transforming the logical design into a program structure drawn as a set of structure charts

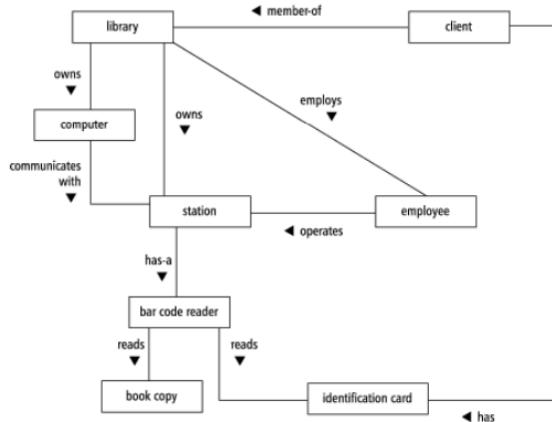


10.3 OOAD methods

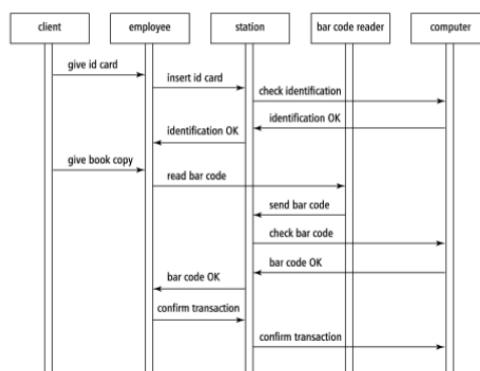
- three major steps:
 - identify the objects
 - determine their attributes and services
 - determine the relationships between objects

from text → select objects → determinate relationships between objects

Result: initial class diagram

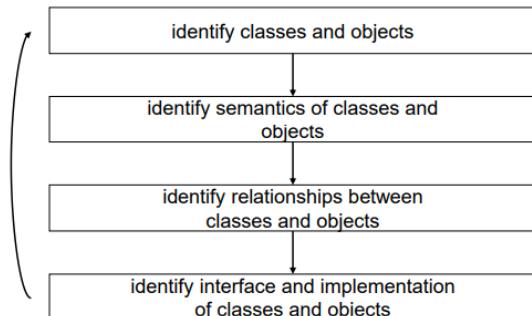


Usage scenario ⇒ sequence diagram

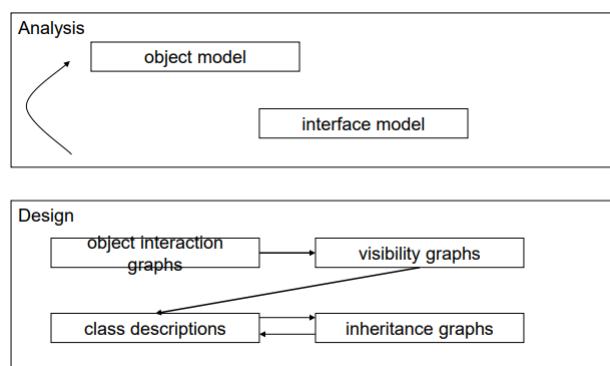


- OO as middle-out design:
 - First set of objects becomes middle level;
 - To implement these, lower-level objects are required, often from a class library;
 - A control/workflow set of objects constitutes the top level;
- OO design methods:
 - **Booch:** early, new and rich set of notations
 - **Fusion:** more emphasis on process
 - **RUP:** full life cycle model associated with UML

10.4 Booch' method



10.5 Fusion method



10.6 RUP method

- Nine workflows, a.o. requirements, analysis and design
- Four phases: inception, elaboration, construction, transition

10.7 Classification of design methods

- Simple model with two dimensions:
 - **Orientation dimension:**
 - * **Problem-oriented:** understand problem and its solution
 - * **Product-oriented:** correct transformation from specification to implementation
 - **Product/model dimension:**
 - * **Conceptual:** descriptive models
 - * **Formal:** prescriptive models

10.8 Design pattern

- Provides solution to a recurring problem
- Balances set of opposing forces
- Documents well-prove design experience
- Abstraction above the level of a single component
- Provides common vocabulary and understanding
- Are a means of documentation
- Supports construction of software with defined properties

10.9 Antipatterns

- Patterns describe desirable behavior
- Antipatterns describe situations one had better avoid
- In agile approaches (XP), refactoring is applied whenever an antipattern has been introduced

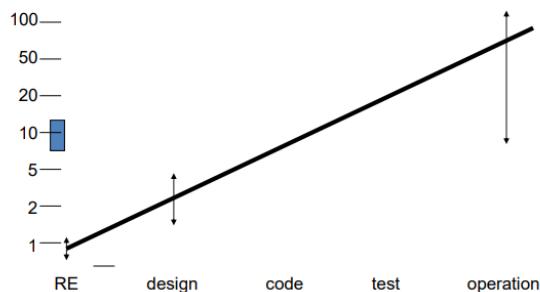
10.10 Design documentation

- Una specifica dei requisiti viene sviluppata durante l'ingegneria dei requisiti. Quel documento serve un certo numero di scopi. Specifica i requisiti degli utenti e come tale ha spesso un significato legale. È anche il punto di partenza per il design e quindi serve un'altra classe di utenti.
- Lo stesso vale per la documentazione progettuale. La descrizione del design serve diversi utenti, che hanno esigenze diverse. Una corretta organizzazione della documentazione progettuale è quindi molto importante.
- IEEE Standard 1016 distinguishes ten attributes. These attributes are minimally required in each project. The attributes from IEEE Standard 1016 are:
 - **Identification:** the unique name of the component, for reference purposes
 - **Type:** the kind of component, such as subsystem, procedure, class, or file
 - **Purpose:** the specific purpose of the component (this will refer to the requirements specification)
 - **Function:** what the component accomplishes (for a number of components, this information occurs in the requirements specification)
 - **Subordinates:** the components of which the present entity is composed (it identifies a static is-composed-of relation between entities)
 - **Dependencies:** a description of the relationships with other components
 - **Interface:** a description of the interaction with other components

11 Software testing

- 30-85 errors are made per 1000 lines of source code
- extensively tested software contains 0.5-3 errors per 1000 lines of source code
- testing is postponed, as a consequence: the later an error is discovered, the more it costs to fix it
- **error distribution:** 60% design, 40% implementation.
- 66% of the design errors are not discovered until the software has become operational

Relative cost of error correction



- Many errors are made in the early phases
- These errors are discovered late
- Repairing those errors is costly
- It pays off to start testing real early

11.1 How then to proceed?

- Testing software shows only the presence of errors, not their absence
- Il più delle volte non è fattibile un test esaustivo
- Random statistical testing does not work either if you want to find errors
- Therefore, we look for systematic ways to proceed during testing

11.1.1 Exhaustive testing

- Un test ideale è quello esaustivo in cui provo tutti gli input possibili
 - è fattibile? Considera questo esempio:

```
static int sum(int a, int b) return a + b;
```

un int è un numero binario di 32 bit, quindi ci sono solo $2^{32} \times 2^{32} = 2^{64}$ ca 10^{21} test ad un nanosecondo (10⁹) per test case circa 30000 anni.
 - se D è infinito? (ad esempio un sistema reattivo)
 - Il test esaustivo non è fattibile: $T \subset D$
 - Devo selezionare un sottoinsieme di D

11.2 Classification of testing techniques

- Classificazione in base al criterio di misura dell'adeguatezza di una serie di casi di test:
 - coverage-based testing;
 - fault-based testing;
 - error-based testing;
- Classificazione basata sulla fonte di informazione per derivare casi di test:
 - black-box testing (functional, specification-based)
 - white-box testing (structural, program-based)

12 Terminologia base del testing

- problemi di ambiguità, di uso errato dei termini
- Un **failure** o **guasto** o **malfunzionamento** è il funzionamento non corretto del programma
 - legato quindi al comportamento che si osserva durante l'esecuzione

Esempio:

```
// programma che dovrebbe restituire il doppio
// del valore passato come parametro
int raddoppia(int x) {
    return x*x ;
}
se chiamo raddoppia(3) noto un malfunzionamento
se chiamo raddoppia(2) non vi e' malfunzionamento
```

- Un **difetto** o **anomalia** o **fault** o **bug** è un elemento del programma sorgente non corrispondente alle aspettative
 - riguarda quindi la parte statica
 - uno o più difetti possono causare malfunzionamenti

*il difetto e' *x invece che *2.*

```
int raddoppia(int x) {
    return x*x ;
}
```

- un programma può avere molti difetti e non presentare alcun malfunzionamento.
- scopo del testing e' quello di evidenziare difetti mediante malfunzionamenti.

12.1 Error, fault, failure

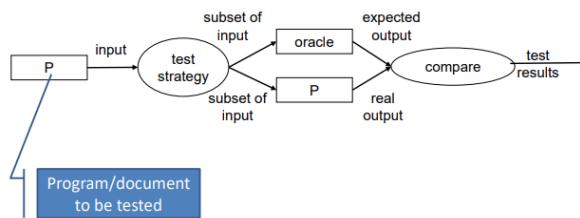
- an **error** is a human activity resulting in software containing a fault
- a **fault** is the manifestation of an **error**
- a **fault** may result in a failure

12.2 V&V and testing

- In this respect, a distinction is often made between ‘verification’ and ‘validation’.
- The IEEE Glossary defines:
 - **verification:** come il processo di valutazione di un sistema o componente, per determinare se i prodotti di una data fase di sviluppo soddisfano le condizioni imposte all’inizio di tale fase
 - * cerca quindi di rispondere la domanda: abbiamo costruito il sistema nel modo giusto?
 - **validation:** come il processo di valutazione di un sistema o componente durante o alla fine del processo di sviluppo per determinare se soddisfa i requisiti specificati.

- * poi si riduce alla domanda: abbiamo costruito il sistema giusto?
- Testing refers to both (sometimes we do verification sometimes validation)

12.3 Testing process



12.4 Testing models

- **Demonstration:** make sure the software satisfies the specs
- **Destruction:** try to make the software fail
- **Evaluation:** detect faults in early phases
- **Prevention:** prevent faults in early phases

12.4.1 Demonstration

- The primary goal is to make sure that the program runs and solves the problem
- If the software passes all tests from the test set, it is claimed to satisfy the requirements
- basarsi solo sui casi di test → pericoloso

12.4.2 Destruction

- A program should be tested with the purpose of finding as many faults as possible
- A test can only be considered successful if it leads to the discovery of at least one fault
- The test set is then judged by its ability to detect faults

12.5 Testing and the life cycle

- requirements engineering

- **criteri:** completezza, consistenza, fattibilità e testabilità
 - * **completezza:** attenzione all'omissione di funzioni o prodotti
 - * **consistenza:** componenti non sono in contraddizione tra loro
- **typical errors:** missing, wrong, and extra information
- determine testing strategy
- generate functional test cases
- test sulla specifica attraverso recensioni

- design

- test funzionali e strutturali possono essere concepiti sulla base della decomposizione applicata in questa fase
- the design itself can be tested (against the requirements)
 - * ossia il mapping tra elementi descritti nella specifica e quelli risultanti della scomposizione
- formal verification techniques
- the architecture can be evaluated

- implementation

- check consistency implementation and previous documents
- code-inspection and code-walkthrough
- all kinds of functional and structural test techniques
- extensive tool support
- formal verification techniques

- maintenance

- **regression testing:** either retest all, or a more selective retest

12.6 Test-Driven Development (TDD)

- First write the tests, then do the design/implementation
- Part of agile approaches like XP
- Supported by tools, eg. JUnit
- È più di una semplice tecnica di prova; include parte del lavoro di progettazione
- steps:
 1. Add a test
 2. Run all tests, and see that the system fails
 3. Make a small change to make the test work
 4. Run all tests again, and see they all run properly
 5. Refactor the system to improve its design and remove redundancies

12.7 Test Stages

- module-unit testing and integration testing
 - bottom-up versus top-down testing
- system testing
- acceptance testing
- installation testing

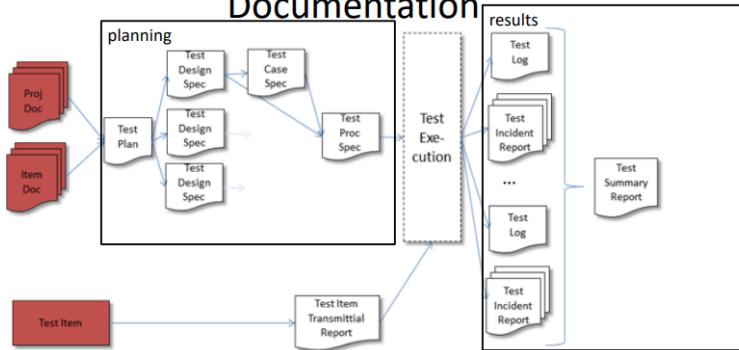
12.8 Verification and validation planning and documentation

- Like the other phases and activities, the testing activities need to be carefully planned and documented;
- Test documentation: **IEEE 928**
 - **Test plan:** è un documento che descrive l'ambito, l'approccio, le risorse e il programma delle attività di test previste. Descrive in dettaglio gli elementi del test, le funzionalità da testare, le attività di test, chi eseguirà ciascuna attività ed eventuali rischi che ciò comporta richiedono una pianificazione di emergenza.

- **Test design specification:** definisce, per ogni componente o caratteristica del software, i dettagli dell'approccio al test e identifica i test associati.
- **Test case specification:** definisce input, output previsti e condizioni di esecuzione per ciascun elemento del test
- **Test procedure specification:** definisce la sequenza di azioni per l'esecuzione di ogni test. Insieme a il Test Plan, questi documenti descrivono l'input per l'esecuzione del test
- **Test item transmittal report:** specifica quali elementi verranno testati. Elenca gli elementi, specifica dove trovarli e fornisce lo stato di ogni elemento.
- **Test log:** gives a chronological record of events
- **Test incident report:** documents all events observed that require further investigation In particular, this includes tests from which the outputs were not as expected
- **Test summary report:** dà una panoramica e valutazione dei risultati

IEEE 829 Standard for Software Test

Documentation



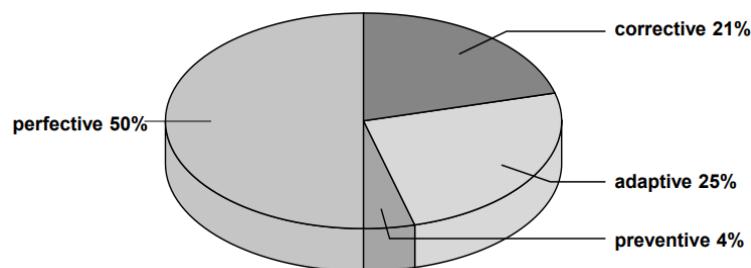
13 Software Maintenance

Il processo di modifica di un sistema software o componente dopo la consegna per correggere i guasti, migliorare le prestazioni o altri attributi, o adattarsi ad un ambiente mutato.

- correcting errors found after the software has been delivered
- adapting the software to changing requirements, changing environments, ...

13.1 Kinds of maintenance activities

- **corrective maintenance:** correcting errors
- **adaptive maintenance:** adapting to changes in the environment (both hardware and software)
- **perfective maintenance:** adapting to changing user requirements
- **preventive maintenance:** increasing the system's maintainability



- Higher quality → less corrective maintenance
- Anticipating changes → less adaptive and perfective maintenance
- Better tuning to user needs → less perfective maintenance
- Less code → less maintenance

13.2 Shift in type of maintenance over time

- **Introductory stage:** emphasis on user support
- **Growth stage:** emphasis on correcting faults
- **Maturity:** enfasi sui miglioramenti
- **Decline:** emphasis on technology changes

13.3 Major causes of maintenance problems

- Unstructured code
- Insufficient domain knowledge
- Insufficient documentation

13.4 Laws of Software Evolution

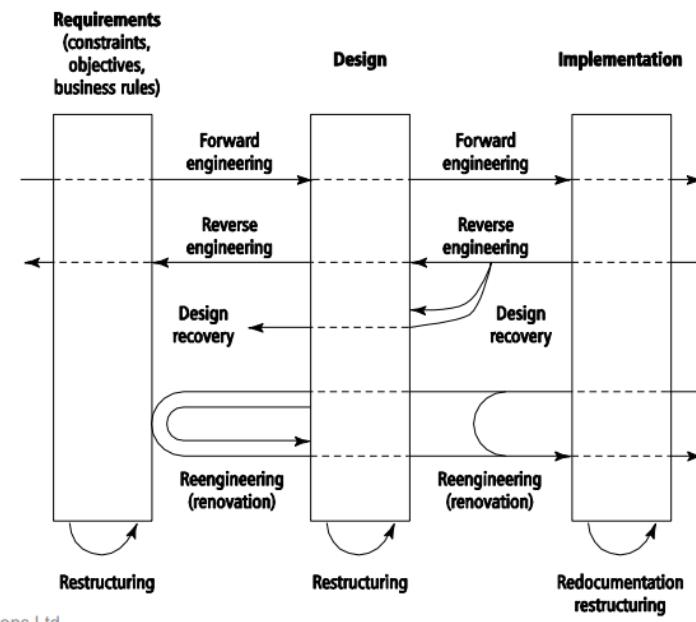
- Quelle che incidono maggiormente sulla manutenzione del software sono:
 - **Law of continuing change:** Un sistema che viene utilizzato subisce la continuazione cambiamento, fino a quando non sarà giudicato più conveniente ristrutturare il sistema o sostituirlo da una versione completamente nuova
 - **Law of increasing complexity:** Un programma che viene modificato, diventa sempre meno strutturato (l'entropia aumenta) e diventa così più complesso. Uno deve investire uno sforzo extra per evitare una crescente complessità
 - **Law of code rewriting:** riscrivere il codice da zero

13.5 Reverse engineering

the process of analyzing a subject system to identify the system's components and their interrelationships and create representations of the system in another form or at a higher level of abstraction.

- Non comporta alcun adeguamento al sistema
- Simile alla ricostruzione di un progetto
- **Design recovery:** result is at higher level of abstraction

- **Redocumentation:** result is at same level of abstraction
- Functionality does not change
- From one representation to another, at the same level of abstraction, such as:
 - From spaghetti code to structured code
 - Refactoring after a design step in agile approaches
 - **Black box restructuring:** add a wrapper
 - **With platform change:** migration



13.6 Reengineering (renovation)

- Functionality does change
- Then reverse engineering step is followed by a forward engineering step in which the changes are made

13.7 Migration

- These wrapping techniques do not change the platform on which the software is running.

- If a platform change is involved in the restructuring effort of a legacy system, this is known as **migration**.
- Migration to another platform is often done in conjunction with value-adding activities such as a change of interface or code improvements.

13.8 Refactoring

Refactoring is a white-box method, in that it involves inspection of and changes to the code.

- Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior.
- Each transformation (called a "refactoring") does little, but a sequence of these transformations can produce a significant restructuring.
- The system is kept fully working after each refactoring, reducing the chances that a system can get seriously broken during the restructuring
- Entropy is not only caused by maintenance
 - – In agile methods, such as XP, it is an accepted intermediate stage. These methods have an explicit step to improve the code.
- Refactoring in case of bad smells:
 - Long method
 - Large class
 - Primitive obsession
 - Data clumps
 - Switch statements
 - Lazy class
 - Duplicate code
 - Feature envy
 - Inappropriate intimacy
 - ...