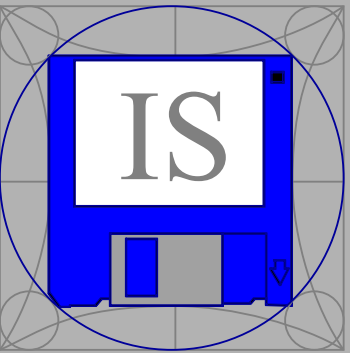
03/10/2022: Introduzione: metodo e concetti chiave di sviluppo del software (Vardanega)

Cose utili:

* Tullio usa solo il Moodle e mette il materiale sul Moodle o sul calendario stesso (tende ad usare siti esterni, comunque accessibili dal Moodle stesso). Non mette tutte le slide (una lezione sì ed un’altra no)
* Legenda del calendario:
  + T 🡪 Teoria
  + P 🡪 Pratica
  + PD 🡪 Progetto Didattico
  + E 🡪 Esercitazione
* Glossario (termini specifici del dominio di interesse) delle parole chiave:
  + Protocollo: in informatica, un insieme di regole o procedure per la trasmissione di dati tra dispositivi elettronici, come i computer. Affinché i computer possano scambiarsi informazioni, deve esistere un accordo preesistente su come le informazioni saranno strutturate e su come ciascuna parte le invierà e le riceverà.
  + Progetto (software) 🡪 Visto (esaminato) da chi lo crea e da chi lo usa. Idee chiave:
    - Sviluppo continuativo
    - Professionale
    - Valutazione qualitativa
    - Collaborativo
    - Monitoraggio

Esso deve essere un mezzo e non un “obiettivo” (goal/aim).

Quest’ultimo, infatti, va compreso; infatti, è visibile a pochi da subito ma l’obiettivo deve essere chiaro a tutti. Non si intende programmazione, quanto piuttosto la *realizzazione di obiettivi* per mezzo dello stesso progetto.

* + Sviluppo: Percorso di realizzazione tra obiettivi *detti* ed obiettivi *raggiunti* per il tramite del software. Questi ultimi rispondono principalmente a dei *bisogni (needs)*, essendo essi stessi il *mezzo* per realizzarli. Si intende tutto ciò attraverso il percorso di progettazione (*design*), di cui il progetto è un prodotto. L’insieme delle regole che lo realizzano è il *paradigma*, seguito dagli informatici che realizzano software e applicano queste regole.
    - Es. Zoom 🡪 Architettura client (inoltrano richieste) – server (soddisfano le richieste presenti) non bastevole; infatti, tutte le richieste sono soddisfatte da un server centrale (senza che l’utente sappia nulla di tutto ciò). Esso è uno strumento

web-based (quindi, una risposta ad un bisogno) 🡪 risultando *portatile*.

Essa è una *piattaforma* (un aggregato che, raduna tanti main(), idealmente compiendo tante cose insieme).

Per approfondire:

<https://www.lavivienpost.com/how-zoom-works/>

Si deriva dalla concezione del Web 2.0 (utenti che creano contenuti); un’architettura tradizionalmente intesa come client-server ha dovuto evolvere per rispondere a nuovi bisogni.

* + - Per realizzare in maniera coesa (interconnessa, espressa come unica entità) le azioni di tanti programmi distinti.
    - Il ruolo dell’informatico è distinguere i bisogni tecnici e i bisogni utente creando un prodotto di *qualità* (quindi seguendo una chiara idea di realizzazione (protocollo), comodo da usare, accessibile, ecc.), risultando dunque professionale. Si cerca di raggiungere lo stato dell’arte (inteso come la qualità in un senso personale) attraverso un costante *autoapprendimento*. Colui che ha gli strumenti e crea è il *practitioner*, continuando ad imparare in autonomia.

04/10/2022: Introduzione (slide omonima): obiettivi, metodo, concetti chiave

(Eventuali approfondimenti listati dal prof:

<http://worrydream.com/refs/Brooks-NoSilverBullet.pdf>

<https://www.acm.org/binaries/content/assets/membership/images2/fac-stu-poster-code.pdf>)

Apprendiamo metodi e pratiche di lavoro alla base della professione informatica.

* Gestire il tempo 🡪 Risorsa scarsa, in quanto soggetto a termini/deadlines
  + Disponibilità, scadenze, conflitti, priorità
* Collaborare 🡪 I compiti da svolgere richiedono molto tempo per un singolo individuo e quindi devono essere suddivisi
  + Fissare obiettivi, dividersi compiti, verificare progressi, riportare difficoltà
* Assumersi responsabilità 🡪 Incarico per garantire una partecipazione equa, di qualità e prendere consapevolezza dei propri doveri
  + Fare quanto pattuito, agire al meglio delle proprie capacità, auto-valutarsi prima di valutare
  + Usare strumenti allo stato dell’arte (quindi, strumenti consolidati e validi nell’ambito di lavoro e di utilizzo)
* Auto-apprendere 🡪 Cercare di “stare al passo” con le tecnologie e con le competenze richieste (avvicinandosi allo stato dell’arte, come detto poco fa)
  + “Imparare a imparare”, essenziale competenza trasversale, soprattutto tramite il

problem-solving, quindi adattarsi a rispondere ad esigenze specifiche di un certo tipo

L’obiettivo è cercare di avvicinarci a un modo di lavorare (way of working) professionale, cioè «operante allo stato dell’arte», per conoscenze tecnologiche e metodologiche, colmando quanto ci manca con il costante auto-apprendimento.

La conoscenza passa dalla comprensione profonda e sperimentata dei significati di quanto si sta facendo

* Non ricordare, ma riconoscere nel corso del tempo quale funzionalità offrano le singole componenti di un software

Vogliamo fissare tali conoscenze in un glossario

* Raccolta di termini/concetti centrali al dominio SWE (Software Engineering, da ora in avanti abbreviato; spesso, tali termini sono in inglese e si offre quando e dove utile, una traduzione, una definizione ed un chiarimento)
* Registrati in modo da facilitarne la localizzazione
* Corredati dalla nostra personale specifica del loro significato e ogni altra informazione utile a riconoscerli

Vogliamo raffinarne costantemente la comprensione

* Legando la teoria (quanto ascoltato) con la pratica (quanto riscontrato)

Questo nel nostro corso si realizza tramite un progetto didattico collaborativo (a scopo di learning, rispetto ad un futuro professionale)

* Promosso da un proponente esterno (aziende terze che portano un’idea)
* Con esigenze e obiettivi funzionali innovativi
* Complesso, impegnativo, visionario
* Tecnologicamente avanzato

Occorre fare attenzione alle norme e regole, per *non inabissarsi e sparire dai radar* (come un sottomarino e partire con un’idea, non riuscire a realizzarla perdendosi in pratiche inutili, buon paragone usato dal prof).

In generale (segue una definizione di Harold Kerzner, autore di “Project Management” ed esperto mondiale in materia) del termine:

Progetto: Insieme di attività che

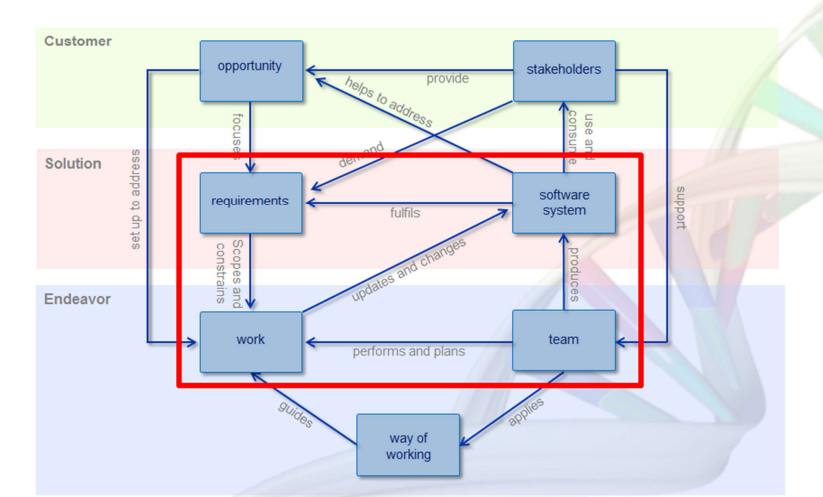
* + - Devono raggiungere determinati obiettivi a partire da determinate specifiche (rispondendo ai *needs-bisogni*)
    - Hanno una data d’inizio e una data di fine fissate
    - Dispongono di risorse limitate (persone, tempo, denaro, strumenti)
    - Consumano tali risorse nel loro svolgersi

L’uscita di un progetto è un prodotto composito

* + SW (software) sorgente/eseguibile, librerie, documenti, manuali

I costituenti di un progetto sono:

* Pianificazione 🡪 Studiare la natura del problema, capire come impostare le attività sulla base della domanda e pianificare l’organizzazione delle risorse
  + Gestire risorse (persone, tempo, denaro, strumenti) in modo responsabile, in funzione degli obiettivi
* Analisi dei requisiti 🡪 I needs sono tradotti in prodotti informatici (non sapendo come fare, si chiede e dialoga con una serie di interlocutori)
  + Definire cosa bisogna fare, normalmente compreso tramite interazione continuativa tra le parti in gioco (quelli che in project management sono chiamati *stakeholders*, quindi “portatori di interesse”)
* Progettazione (design) 🡪 Trasformare l’idea di comprensione (profonda) del problema in modo attuabile e con una soluzione sensata ed accettabile. Qui non si ha una sola soluzione, ma ce ne possono essere tante
  + Definire come farlo
* Realizzazione (implementation) 🡪 Creare un *utensìle* (cioè, avere un prodotto *usabile*; quindi, una volta creato compie correttamente la sua funzione e sia utile allo scopo voluto)
  + Farlo, perseguendo qualità (cioè, il grado di bontà oggettiva delle azioni eseguite)
  + Accertando l’assenza di errori od omissioni
  + Accertando che i risultati soddisfino le attese



Vedere la possibilità di creare

un prodotto utile allo scopo possibile

Figura - Proveniente da https://semat.org/

Chi ha cosa da dire significative sulle opportunità, sui bisogni e sulla soluzione/implementazione della stessa.

Possono essere clienti, committenti, venture capitalists (investitori)

Stato dell’arte, adattato al proprio livello sulla base delle regole presenti

Collassare design ed implementazione, esempio scrivendo codice, è tipico di chi progetta male.

Prima si pensa bene a cosa fare e l’implementazione risulta essere conseguenza (non premessa) del design.

In tal modo, mi assicuro di soddisfare le aspettative di chi lo richiede, chiedendo ed interagendo spesso.

Cosa non è un progetto:

* Si è accecati dalla fondamentale inutilità di fondo dei loro prodotti, dal senso di successo che si prova nel farli funzionare (il fatto che compili/funzioni ci toglie dall’esaminare ulteriormente un prodotto)
* In altre parole, i loro difetti fondamentali di progettazione sono completamente nascosti dai loro difetti di design difetti superficiali di progettazione (errori di sintassi, segmentation fault, ecc.)

Esso dovrebbe essere l’espressione di realizzare sulla base di regole un prodotto preciso e professionale, valorizzando l’esperienza e creandole al meglio delle regole esistenti.

L’opportunità è il progetto stesso; il bisogno c’è ed esiste e si cerca di anticipare la richiesta e soddisfare il bisogno.

Altri termini del glossario:

* Teamwork
  + Lavoro collaborativo che punta a raggiungere un obiettivo comune in modo efficace ed efficiente
  + I membri del team sono inter-dipendenti
  + La gestione di questa inter-dipendenza richiede il rispetto di regole e di buone pratiche
    - Comunicazioni aperte e trasparenti: risoluzione dei conflitti
    - Costruzione e preservazione delle fiducia reciproca: condivisione e collaborazione
    - Assunzione di responsabilità: coordinamento
    - Condivisione dei rischi
  + La sua base è un solido way of working
* Stakeholder (portatore di interesse)
  + Tutti coloro che a vario titolo hanno influenza sul prodotto e sul progetto
    - La comunità degli utenti (che usa il prodotto)
    - Il committente (che compra il prodotto)
    - Il fornitore (che sostiene i costi di realizzazione)
    - Eventuali regolatori (che verificano la qualità del lavoro)
* Way of working
  + Come organizzare al meglio le attività di progetto (in modo professionale)

Per svolgere un progetto potendo confidare nel suo successo serve ingegneria, quindi usando principi noti ed autorevoli (best practices), basato sulla software engineering (SWE), per garantire i migliori risultati in circostanze note e specifiche.

* Disciplina per la realizzazione di prodotti SW così impegnativi da richiedere il dispiego di attività collaborative
* Capacità di produrre “in grande” e “in piccolo”
* Garantendo qualità: efficacia (misura della capacità di raggiungere l'obiettivo prefissato)
* Contenendo il consumo di risorse: efficienza (misura dell'abilità di raggiungere l’obiettivo impiegando le risorse minime indispensabili)
* Lungo l’intero periodo di sviluppo e di uso del prodotto: ciclo di vita (Gli stati che il prodotto SW richiesto assume dal suo concepimento (bisogni 🡪 needs) all’uso e poi eventualmente al ritiro

Un sistema SW è tanto più utile quanto più è usato.

* Metrica: integrale della sua intensità d’uso nel tempo

Più lunga la vita d’uso di un prodotto, maggiore il suo costo di manutenzione

* Manutenzione: insieme di attività necessarie a garantire l’uso continuativo del prodotto
  + Reattivamente (per correzione dopo malfunzionamento) o preventivamente

Il costo di manutenzione ha varie componenti

* Mancato guadagno, perdita di reputazione, recupero o reclutamento esperti, sottrazione di risorse ad altre attività

I principi SWE puntano ad abbassare tali costi

* Sviluppando SW più facilmente manutenibile

L’obiettivo è raccogliere, organizzare, consolidare la conoscenza (body of knowledge) necessaria a realizzare progetti SW con efficacia ed efficienza (collezione e manutenzione migliorativa di best practice) e quindi applicare principi ingegneristici calati nella produzione del SW. Il modo di applicazione è:

* Sistematico
  + Modo di lavorare metodico e rigoroso
  + Che conosce, usa ed evolve le best practices di dominio
* Disciplinato
  + Che segue le regole che si è dato
* Quantificabile
  + Che permette di misurare l’efficienza e l’efficacia del suo agire

Distinguiamo infatti la figura del programmatore da quella del software engineer:

* Il programmatore scrive programmi in modo tecnico e personalizzato
* Il software engineer realizza parte di un sistema complesso con la consapevolezza che potrà essere usato, completato e modificato da altri. Egli comprende il contesto in cui si colloca il sistema cui contribuisce e cerca di attuare dei compromessi intelligenti e lungimiranti tra costi – qualità,

risorse – disponibilità, esperienza utente – facilità di realizzazione

Il metodo di acquisizione delle competenze avanza in corrispettivo alle sfide proposte, ciascuna con un punto di partenza e di arrivo, creandosi la condizione giusta per apprendere ed entrare nel flow (idealmente, Zone of Proximal Flow, quindi creandosi gli strumenti, apprendere e mettere in pratica nel concreto in maniera continuata e migliorante).

L’obiettivo è creare un *MVP (Minimum Viable Product)*, il prodotto fa quello che deve fare in un modo accettabile; chi lo decide è un accordo tra il committente e l’incaricato. Essa è una soglia minima e punta ad arrivare verso ciò che è fattibile.

06/10/2022: Processi di ciclo di vita (Vardanega)

(Eventuali approfondimenti:

<https://www.math.unipd.it/~tullio/IS-1/2010/Approfondimenti/A03.pdf>

<https://www.math.unipd.it/~tullio/IS-1/2009/Approfondimenti/ISO_12207-1995.pdf>)

Un sistema SW, tanto più è usato, dovrebbe essere sottoposto sempre a manutenzione, la quale permette di risalire ad uno storico di vita di un SW. L’uso permette di scoprire necessità, difetti, adattandosi alle nuove esigenze e creandosi nuovi obiettivi.

Lo sviluppo di per sé ha un fine esecutivo (crei un software e finita lì); la manutenzione è il continuare in parte lo sviluppo, definendo un ciclo di vita per gli utenti che imparano a volerlo usare.

Gli stessi numeri di versione definiscono quanti cambiamenti: ovviamente, più grande è, più il software è cambiato. A noi interessa farla bene, dando una qualità crescente al prodotto, dando il giusto ordine alle operazioni (individuando il giusto e lo sbagliato in quanto fatto/eseguito).

Questa idea è il controllo di versione. Esso è un attributo decisivo in base alle esigenze dell’utente, che magari funzionano meglio in una versione precedente; ovviamente, occorre un giusto controllo tra le versioni, affinché non si perda tempo a sviluppare usando versioni sconnesse tra di loro.

Un software non è un oggetto statico; per esempio, un software scritto in un solo linguaggio per un solo programma non è manutenibile. Non è possibile rimettere insieme i pezzi dopo la creazione del software, in quanto deve essere fatto sin da subito (modularizzare); il software ben fatto è costituito di parti, le quali sono disposte secondo un preciso ordine.

L’idea chiave è che: se cambia il sorgente, allora cambia anche il prodotto. L’insieme delle parti del software dà luogo ad un prodotto stabile.   
Alla base, quindi, si cerca di compiere degli aggiustamenti a qualcosa già stabile di per sé.

La decisione di riconoscere quali parti usare e come tenerle insieme è detto configurazione.

Da ciò, consegue sia il citato controllo di versione, integrato anche con il controllo di configurazione.

Le parti che cambiano molto sono quelle molto esposte agli utenti e le parti che cambiano poco sono dei protocolli software che rimangono fissi durante la creazione del prodotto. La storia di un prodotto è definita da maturazioni progressive, definite come stati. Le transizioni tra i singoli stati e le attività svolte sono gli archi. Quali/quanti stati/regole attivino/abilitino gli archi dipende da:

* Vincoli/obblighi contrattuali, impegni (way of working), opportunità

*Altra definizione di progetto*:

Ogni progetto ha il compito di cercare di spingere un prodotto verso una nuova fase del suo ciclo di vita, con il solito obiettivo di creare qualcosa di usabile (implementation sulla base del design). In effetti, il ciclo di per sé intende l’usuale compiere attività ripetute comunemente; ciò può essere una conseguenza della manutenzione, quindi la ripetizione di attività iniziali.

Più nello specifico, il ciclo di vita del SW intende gli stati che il prodotto SW assume tra concepimento e ritiro in conseguenza delle attività svolte su di esso.

Il cambiamento di stato viene causato dalla comprensione dei *needs* e la progettazione del *design* sia per motivazioni interne che esterne allo stato di sviluppo. Ciò significa che:

* prendo un software non esistente, lavorando sulla base dei needs
* prendo un software esistente e cerco di mantenermi ai needs iniziali, qualificando il design realizzandolo progressivamente con l’implementation, mantenendo i principi cardine ed evolvendolo

Definiamo in generale:

* i *processi* di ciclo di vita, che raggruppano e codificano le transizioni nel ciclo di vita di un SW. In generale, sono insiemi di attività correlate e coese che trasformano ingressi (bisogni) in uscite (prodotti) secondo regole date e consumando risorse allo stesso tempo per farlo (definizione standard ISO 9000).
* i *modelli* del ciclo di vita, capendo quali stati/transizioni privilegiare, quali processi attivare ed attuare aderendo ad un modello e pianificare le attività, eseguendole e controllandole

Le regole dello stato dell’arte hanno stabilito quali attività privilegiare seguendo la logica del ciclo di vita, in cui ogni attività va opportunamente configurata.

07/10/2022: Object Oriented Programming Principles (Cardin)

Il metodo di produzione del software segue dei principi di massima, cosiddetti ingegneristici, affinché la produzione di qualcosa di astratto come un software sia un processo ripetibile. Il ciclo è come segue; nel corso vediamo soprattutto la parte di Manutenzione (può essere molto dannoso chiunque ci metta mano, portando ad una potenziale regressione se fatta male, oppure non è possibile aggiungere nuove funzionalità, portando alla morte immediata del software):

Manutenzione & Evoluzione

Produzione

Programmazione

Ideazione

Il processo di scrittura del software è un viaggio lungo e ricco di errori; ovviamente, questi sono la principale forma di apprendimento. La progettazione di un software deve idealmente comprendere sempre direttamente lo stakeholder, tale che si abbia un confronto diretto con chi conosce il dominio applicativo di riferimento. Caso per caso, non si ha una soluzione universale: dipende sempre dal contesto.

Parliamo quindi della programmazione ad oggetti, che considera *oggetti* ovviamente. L’idea di “oggetto” risulta essere forzata, perché noi intendiamo gli oggetti come componenti. I tre principi cardine della programmazione ad oggetti (incapsulamento, ereditarietà e polimorfismo) non sono perfetti; almeno uno di questi viene rotto. Come detto, il problema sono gli oggetti in sé, in quanto le stesse gerarchie sono insensate; si parte normalmente da dati grezzi, strutturati in modo fisso come *record*, normalmente senza connessione tra dati e funzioni.

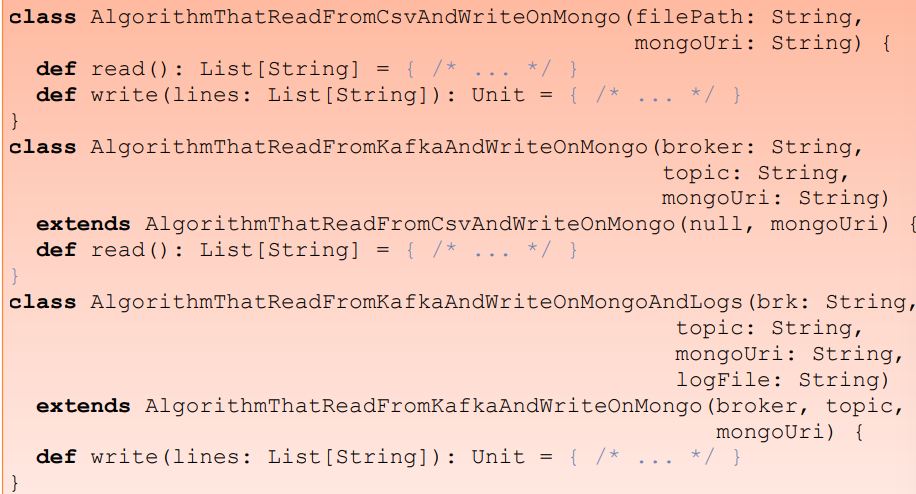
Il problema è il fatto che nessuna delle tre può esistere implicitamente senza le altre; l’incapsulamento permette di separare i dati e di modellare sistemi sulla base di questo, l’ereditarietà compone dei tasselli e cerca di farlo sulla base dei sottotipi, mentre il polimorfismo cerca di considerare una classe e adattarla alla bisogna, capendo la funzionalità implicita e andando “sopra/sotto” nella gerarchia.

Più le variabili, con questo paradigma, hanno scope ampio, più problemi abbiamo; se cambia valore (side effects), il programma alla lunga non è più adatto per determinati compiti; tanto più se si ha a che fare con delle gerarchie, dove una modifica si ripercuote su tutti gli altri oggetti facenti parte.

Con tanti parametri, molto lunghi da scrivere (*verbose*), difficili da mantenere, senza *information hiding*, senza autorizzazione o giusta limitazione, il testing è molto difficile.

I dati sono implementati con dei comportamenti, nascondendo lo stato interno all’esterno del programma e senza un vero collegamento tra procedure interne e dati veri e propri.

Si tende ad andare verso le *interfacce* per nascondere le implementazioni, viste le funzioni che condividono input comuni, da cui si prende un insieme minimo e si creano strutture comuni per accedervi, formando i *tipi,* con comportamenti definiti dai dati privati.



*L’ereditarietà* è il problema più grande usando le classi, con dei *tipi* (richieste a cui rispondere), con l’obiettivo di risparmiare codice (*code reuse*) ereditando il comportamento (*subtyping*) e nascondendo le implementazioni. Esempio di codice che cerca di adattarsi ad una serie di funzionalità multiple (leggere formati di file e framework e relativi percorsi/permessi):

Questo tipo di programmazione ad oggetti è totalmente ridondante, dovendo sempre trasportarsi dietro un contesto, dato che le classi figlie hanno una dipendenza forte dalle classi parent/genitori.

Ogni metodo figlio sbaglia; rispetto alla classe padre, avendo fatto riutilizzo del codice, si manca il vero obiettivo, dato che non si ha qualcosa di scalabile. *Noi non riutilizziamo comportamento, ma cerchiamo solo di riutilizzare il codice* (in questo caso, questo riutilizzo aumenta la *dipendenza* delle classi).

Infatti, *una classe e una responsabilità* (ereditarietà solo da tipi astratti).

La dipendenza tra classi è definita come: A 🡪 B

(se modifico B, ho probabilità di modificare anche A; se modifico A, allora si modifica anche B).

Il problema della dipendenza rende difficile la manutenzione, in quanto è anche difficile da testare (soprattutto perché quando il codice risulta *accoppiato* allora più modifiche comportano più ridefinizioni).

Per le classi, si cerca di evitare che il programmatore faccia casino con i modificatori di accesso (anche con i *final* che permettono ad altre classi di non avere problemi sull’ereditarietà) al fine di non aumentare la dipendenza tra le classi.

Quindi, si cerca di evitare l’ereditarietà da classi concrete (quelle che *devono* ridefinire le classi astratte, queste ultime classi che hanno già alcuni metodi ridefiniti e, soprattutto, *non si devono istanziare*/usare).

Se una classe *implementa* (non eredita, diverso) da un’altra, essa condivide un *contratto*, quindi metodi e funzioni che può utilizzare ai propri scopi. Il codice condiviso tra le interfacce è *minima* (*loose coupling*), dato che il codice è lascamente accoppiato con il tipo specifico e, grazie al polimorfismo, non serve sapere il tipo specifico. Il riutilizzo è definito a scatola chiusa (*black box reuse*), quindi si assemblano funzionalità in nuove caratteristiche senza dettagli interni delle classi, usando la tecnica della composizione dei tipi.

Immagine che contiene testo

Descrizione generata automaticamente

L’idea è di progettare quando serve (per contratto) e soprattutto definire delle implementazioni che abbiano senso rispetto alla classe base, rappresentandone alla bisogna una naturale estensione.

I *cambiamenti*/*changes* in un componente influiscono sulle dipendenze (esterne/interfacce ed interne/implementazioni). La *dipendenza/dependency* è una misura della probabilità di cambiamenti tra componenti dipendenti, distinguendo tra alta e bassa probabilità di cambiamenti (difficile farlo senza introdurre dei bug). Distinguiamo alcuni termini:

* Dipendenza/Dependency
  + Quando gli oggetti di una classe lavorano brevemente con oggetti di un'altra classe
* Associazione/Association
  + Quando gli oggetti di una classe lavorano con gli oggetti di un'altra classe per un periodo di tempo prolungato
* Aggregazione/Aggregation
  + Quando una classe possiede ma condivide un riferimento ad oggetti di un'altra classe.
* Composizione/Composition
  + Quando una classe contiene oggetti di un'altra classe
* Ereditarietà/Inheritance
  + Quando una classe è un tipo di un'altra classe

La *dipendenza più banale (weakest dependency)* è data dal numero di righe di codice condiviso tra A e B (*limitata nel tempo*, quindi per l’esecuzione di un metodo, *limitata a codice condiviso*, quindi a livello di interfaccia). Una classe contiene un riferimento ad un oggetto, magari con legame definito per tutta la durata di vita di un oggetto (permanente), con tutti i comportamenti di una classe condivisi.

Nell’aggregazione, un tipo possiede l’altro e la creazione/cancellazione non è cosa semplice, in quanto occorre ridefinire i comportamenti. La *composizione* evita la condivisione dei componenti.

La *dipendenza più forte* (*strongest dependency)* viene data dal non riutilizzo di codice privato e ogni cambio di codice della classe genitore può distruggere i figli. Quanto più codice condiviso, più forte la dipendenza. Formalmente, sarebbe definita come nell’immagine seguente (dove SLOC, intende Source Lines Of Code), portando un codice facile da mantenere.

Immagine che contiene testo

Descrizione generata automaticamenteL’accoppiamento è proporzionale alla probabilità di cambiamento mutuale tra le componenti, risultando come una sorta di media.

Cerchiamo a tutti gli effetti di avere una soluzione ottima, intesa come minimizzazione (sull’accoppiamento).

10/10/2022: Processi di ciclo di vita (Vardanega): continuazione e conclusione slide

Tornando alla definizione di *processi*, essi sono definiti come parti di progetto che attengono ad uno specifico di questi, producono una parte dei risultati e misurano l’efficacia sulla base dei needs, minimizzando le risorse utilizzate (efficienza), vista quest’ultima come dimensione dello spreco (quanto risorse servirebbero, più o meno, per fare una certa cosa).

Idealmente, si ha una trasformazione (dai needs ai prodotti), verificando in uscita la risposta ai needs (se abbiamo soddisfatto le attese, facendo il meglio e sprecando il meno possibile).

Un insieme di attività è *efficiente* quando fa quel che deve fare, non sprecando risorse

* Metrica: produttività (i.e., efficienza produttiva): rapporto tra quantità di prodotto realizzato e risorse utilizzate

Un insieme di attività è *efficace* quando raggiunge gli obiettivi attesi

* Metrica: grado di raggiungimento obiettivi interni (del fornitore) o esterni (gradimento del cliente/degli utenti)

Grazie a delle strade collaudate/metodi utili (idealmente, cammini minimi), cerchiamo di arrivare alla realizzazione sprecando meno risorse possibili, usando delle risorse esterne che controllano le nostre azioni. La combinazione di efficienza ed efficacia si chiama «*economicità*».

Premesse:

1. Compito di un progetto è «spingere» un prodotto SW attraverso un segmento del suo ciclo di vita
2. I processi di ciclo di vita specificano quali attività svolgere per attuare corrette transizioni di stato in modo efficiente ed efficace
   * Conseguenza 🡪 Lo svolgimento di un progetto attua un insieme organico di processi (la parte del ciclo di vita che serve per il progetto), quindi non un insieme causale e disordinato di attività

In questo modo, applichiamo dei limiti al nostro campo d’azione; questi limiti sono standard, cataloghi di processi già formati che aiutano a raggiungere l’economicità.

* Standard generali (base per tutti coloro che creano software)
  + ISO/IEC 12207:1995 e sue evoluzioni (Software life cycle processes)
  + Modello di riferimento del dominio SWE
  + <https://www.math.unipd.it/~tullio/IS-1/2009/Approfondimenti/ISO_12207-1995.pdf>
* Standard settoriali: per specifici domini applicativi (di seguito alcuni esempi)
  + IEC 880 : settore nucleare
  + RTCA DO-178 : settore aeronautico
  + ECSS E40: settore spaziale

Ogni standard ha due funzioni complementari:

1. Modello di azione
   1. Fissa quali attività svolgere e come farlo: visione *prescrittiva*
   2. Specifica cosa serva fare, lasciando libertà sul come: visione *descrittiva*
2. Modello di valutazione
   1. Identifica “*best practice*” di riferimento: la regola d’arte
   2. Permette di misurare quanto un particolare *way of working* disti da essa (soprattutto, chi cerca di avvicinarsi alle regole allo stato dell’arte, sperando di saperlo prima)

Se non si ha way of working, il lavoro è qualificato.

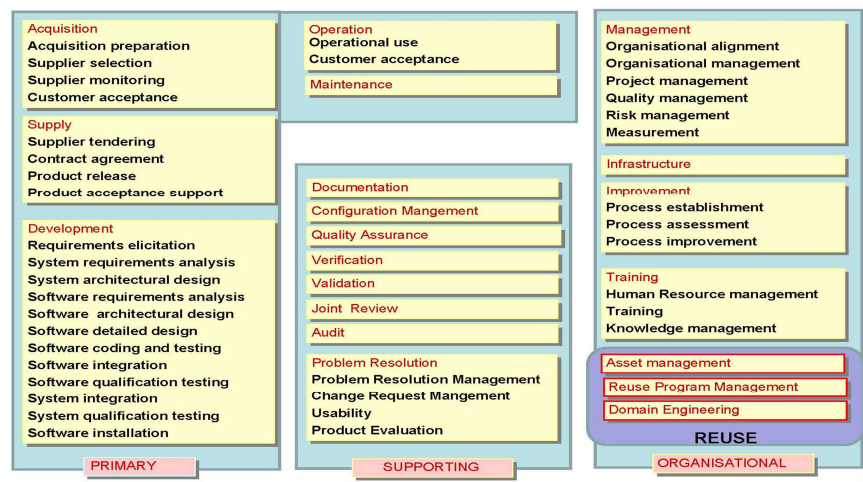
Come detto, ISO/IEC 12207:1995 è il modello più noto e riferito, che ha natura descrittiva e di cui esistono altri modelli specializzati di dominio sono prevalentemente prescrittivi.

Esso è “ad alto livello”, che permette di liberarsi da vincoli inutili alla creazione:

* identifica i processi di ciclo di vita del SW
* ha struttura modulare che richiede specializzazione
* specifica le responsabilità sui processi
* identifica i prodotti dei processi

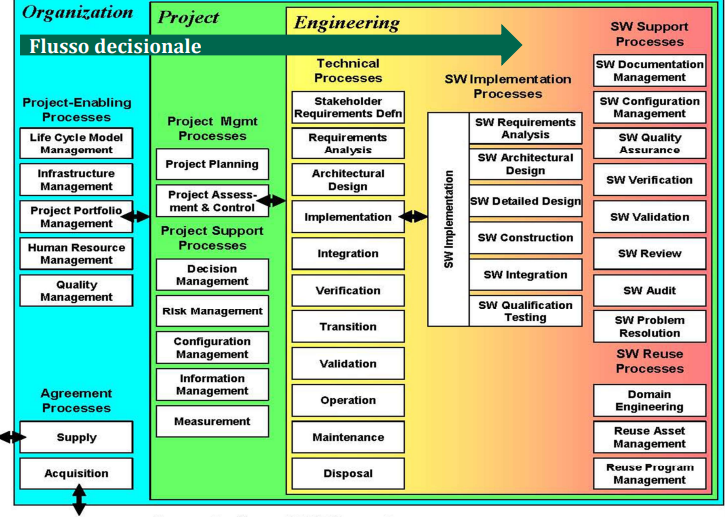
Graficamente, qui di seguito:

* i processi sono identificati a categorie (primario, a sostegno, organizzativo)
  + se esiste un progetto, esiste sempre un processo primario, fatto di sviluppo, acquisizione e fornitura delle risorse
  + i processi di supporto servono a mantenere l’integrità del processo primario (validazione, verifica, documentazione, manutenzione, revisione, bilancio)
  + i processi organizzativi cercano di capire i compiti da assegnare ai singoli, gestendo l’infrastruttura ed operando una formazione a fine di miglioramento

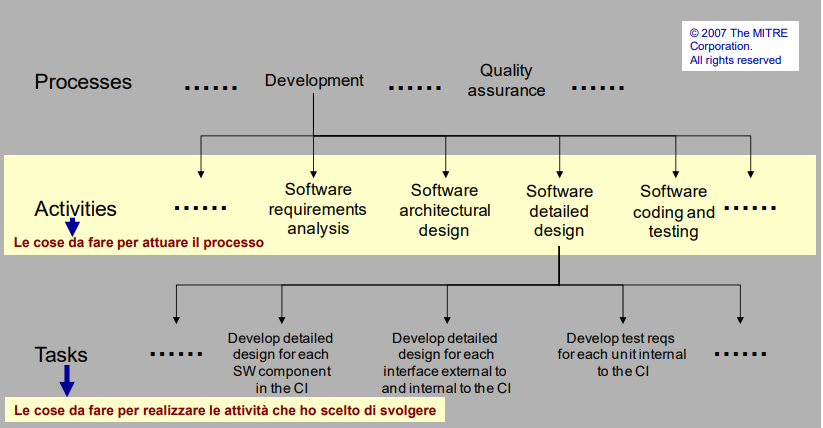


Lo sviluppo è un processo e non una singola attività, in quanto formato da molteplici compiti riuniti in modo organizzato. Lo scrivere codice, ad esempio, è una singola attività. Tra le attività di supporto, il progetto ha sicuramente bisogno di *documentazione*, per capire come le attività sono svolte e cosa è stato fatto. Similmente, nelle attività organizzative si ha la *formazione/training*, tale che chi prende parte ad un progetto richiede che tutti si formino continuamente, evolvendo in modo costante.

Riassumendo (*a* *titolo visivo, più che altro, impensabile ricordarseli nel dettaglio*):

* Processi primari
  + Acquisizione
    - Gestione dei propri sotto-fornitori
  + Fornitura
    - Gestione dei rapporti con il cliente
  + Sviluppo
  + Gestione operativa (utilizzo)
    - Installazione ed erogazione dei prodotti e/o servizi
  + Manutenzione
    - Correzione, adattamento, evoluzione
* Processi di supporto
  + Documentazione
  + Gestione della configurazione
  + Accertamento della qualità
  + Verifica
  + Validazione
  + Revisioni congiunte con il cliente
  + Verifiche ispettive interne
  + Risoluzione dei problemi (gestione dei cambiamenti)
* Processi organizzativi
  + Gestione dei processi
    - Debugging dei processi
  + Gestione delle infrastrutture (tecniche)
    - Organizzazione degli strumenti di supporto ai processi
  + Miglioramento del processo
    - Manutenzione migliorativa dei processi
  + Formazione del personale
    - Aiuto, stimolo e complemento all’auto-formazione

Le attività di sviluppo software comprendono, in sé, più compiti, dato che il progetto si dice collaborativo per definizione. Anche i processi sono svolti collaborativamente (connessione sia con altre attività sia con processi insiti alle attività stesse). Se più persone collaborano per lo svolgimento delle attività, esse non si devono sovrapporre. Quando questo succede, è possibile che l’efficienza diminuisca grandemente (si spreca tempo, in quanto qualche componente non ha “nulla da fare”). Ecco perché per mantenere l’efficienza si può pensare di scomporre le attività in pezzi più piccoli, rendendoli elementari (atomici, non divisibili). Più lunga è l’attesa, peggio è.



Dunque, un progetto deve essere diviso in questo modo, tale da svolgere più attività in poco tempo.

Keywords:

* Codifica e prova dei componenti SW
* Integrazione dei componenti e del sistema
* Collaudo del SW e del sistema

Gli standard:

* da un lato danno una guida su *come fare* le attività (modello)
* dall’altro, servono come strumenti di valutazione sulla *way of working* rispetto allo stato dell’arte

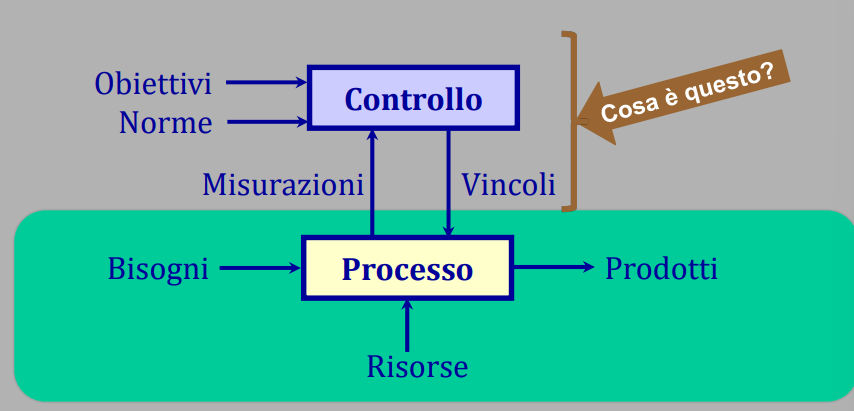
La misura della way of working si chiama qualità, quindi la “vicinanza alle best practices”, misurando principalmente la distanza tra quanto fatto e l’ideale stato dell’arte che gli standard forniscono.

Fattori di specializzazione:

* Dimensione e complessità del progetto
* Tipo e intensità dei rischi di progetto
  + Nel dominio applicativo, nel rapporto con clienti e utenti, nella maturità/complessità delle tecnologie in uso
* Competenza ed esperienza delle risorse umane

Associare ai processi un sistema di qualità aiuta a migliorarli e a garantire *conformità*

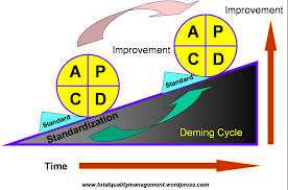
* Maturità = qualità misurata delle prestazioni
* Conformità = adesione alle aspettative e agli obblighi



Il controllo di processo contente di attuare manutenzione migliorativa al proprio way of working (fare le cose al meglio possibile).

Si ha il *principio del miglioramento continuo* (a salita continua), in cui ogni attività (poiché misurabile) è confrontabile con le attese.

Per realizzare ciò, si ha un ciclo a 4 stadi nato per apportare migliorie, noto come *Shewhart-Deming’s Learning-and-Quality Cycle/PDCA Cycle*, così articolato (in quanto costituito da pezzi ripetibili):

* Pianificare (Plan)
  + Definire attività, scadenze, responsabilità, risorse per raggiungere specifiche obiettivi di miglioramento. Esso non corrisponde alla pianificazione delle normali attività, ma alla variazione esplorativa di parti del way of working per ottenere migliore qualità (di processo e quindi anche di prodotto).
* Eseguire (Do)
  + Eseguire le attività secondo P (Pianificare/Plan)
* Valutare (Check)
  + Verificare l’esito delle azioni di miglioramento rispetto alle attese (se le variazioni attuate abbiano conseguito il miglioramento atteso, cosa che non si fa tramite testing sul SW prodotto)
* Agire (Act)
  + Consolidare il buono e cercare modi per migliorare il resto

Eseguendo e facendo, si eseguono piccoli e costanti passi di miglioramento, utili a livello produttivo, sempre spostandosi in avanti. Si individuano in particolare le debolezze, pianificando le attività di miglioramento e identificando dove migliorare.

*Il ciclo PDCA* non è la normale sequenza di attività di progetto, in quanto non *opera* sul prodotto, ma

*sul way of working*, per migliorarlo.

11/10/2022: Telemetria (da T01, slide 19) e Modelli di sviluppo software (Vardanega)

(Eventuali approfondimenti:

<https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9121630>

<https://www.agilealliance.org/agile101/>

<https://www.ivarjacobson.com/files/field_iji_file/article/scrum_teams_improving_with_essence_0.pdf>)

Il SWE spiega come svolgere le attività in un modo che abbia sempre 3 caratteristiche (qui classificate non in ordine di importanza):

1. Sistematico 🡪 Parte di quello che deve essere fatto per un certo fine a prescindere di chi lo faccia (quindi, modo stabile da usare sempre in un certo contesto), rendendo l’azione una procedura (azioni ripetibili)
2. Disciplinato 🡪 Insieme di regole applicate ad un dominio, indice di seguire le regole, in modo oggettivo (a regola d’arte)
3. Quantificabile 🡪 Per aderire alle regole dello stato dell’arte, si ha un controllo misurabile

Idealmente, si vede come “cruscotto”, cioè esaminiamo pezzo per pezzo tutte queste caratteristiche, verificando quando siamo distanti dall’obiettivo (telemetria).

Il ciclo di vita di un progetto è articolato in varie fasi:

* Concezione (Elaborazione a priori di tutto ciò che il progetto deve fare)
* Sviluppo (Trasformazione dei needs in un utensile tramite l’implementation)
* Utilizzo (Stadio di vita del prodotto, quindi chi ha fornito il prodotto garantisce che questo continui ad essere usato tramite la manutenzione)
* Ritiro (Usare qualcosa che tecnicamente funziona ma non è più adatto)

L’obiettivo di un progetto è progredire come stato di avanzamento di prodotto SW.

Il termine fase corrisponde alla permanenza del prodotto software in uno stato o in una transizione, in particolare un tempo continuo e contiguo in cui si esegue una stessa azione.

Esistono *molteplici* modi per passare e spostarsi tra i cicli di vita, definiti come *modelli* di sviluppo; quindi, qualcosa che collega e designa modi di fare/agire, al fine di arrivare allo stesso obiettivo secondo la

*way of working*. Lo stadio di «evoluzione» (manutenzione) innesca nuovi cicli di sviluppo.

Un modello viene definito come insieme di specifiche che descrivono un fenomeno di interesse (astratto / concreto), in modo che non dipende dall’osservatore ed è dimostrata corretta/vera (empiricamente o per teorema). I modelli aiutano a studiare, comprendere, misurare, trasformare l’oggetto di interesse.

* Il modello specifica *cosa* esso sia
* L’architettura interna (design) specifica *come* esso funzioni
* L’analisi spiega *perché* fa quel che fa nel modo in cui lo fa

La prima scrittura del software era un processo difficoltoso, poiché veniva fatta senza organizzazione e in autonomia, generando quindi progetti caotici e difficili da gestire 🡪 *cowboy coding*.

Quello stile causò la crisi del SW, in quanto non si aveva un vero e proprio standard; questo portò alla nascita della disciplina SWE. Infatti, per molto tempo si è scritto software ignorando l’utente davanti a sé,

il contesto di apprendimento e di utilizzo. Esistono una serie di modelli organizzati come segue:

Immagine che contiene testo

Descrizione generata automaticamente

Glossario:

* Iterazione
  + Raffinamenti o rivisitazione: distruttivo
* Incremento
  + Aggiunte successive a un impianto base: costruttivo
* Prototipo
  + Provare e scegliere soluzioni: usa-e-getta o per incrementi
* Riuso
  + Copia-incolla opportunistico (occasionale: basso costo, scarso impatto)
  + Sistematico (per progetto / famiglia di prodotti / ogni prodotto): maggior costo,

maggior impatto

1. Modello sequenziale (a cascata), ideato nel 1970 da Royce. Vede la realizzazione del software partendo dall’idea di realizzazione meccanica sulla base di precondizioni (*gates* per andare incontro ai needs, visti con l’analisi dei requisiti), trasformati concretamente all’interno del design. Il prodotto dovrebbe contenere tutto ciò che serve.
   * Esempio: Il Web HTTP, visto come protocollo di utilizzo della rete per rispondere al bisogno di scambio di informazioni

Le fasi sono *rigidamente sequenziali*, in cui il design è guida all’implementation, dicendoci esattamente cosa deve essere fatto. Esse sono distinte nel tempo.

* + Altrimenti, non siamo più quantificabili o disciplinate

I prodotti sono principalmente *documenti*, non tanto codice, fino poi ad includere il SW.

Ogni fase (stato/transizione) viene definita da:

* + Attività previste e prodotti attesi in ingresso (gates) e in uscita (dimostrabili con le

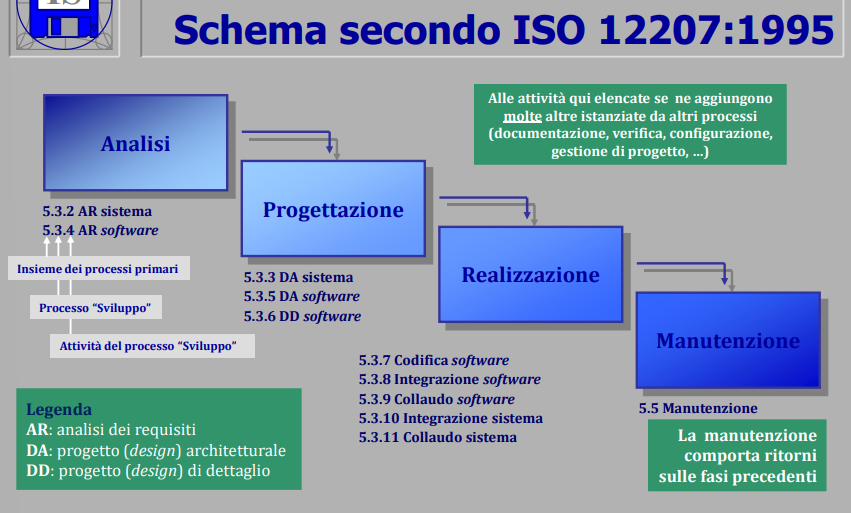
post-condizioni)

* + Contenuti e struttura di documenti che descrivono lo stato raggiunto e le attività svolte
  + Responsabilità e ruoli coinvolti nelle attività
  + Scadenze di consegna dei prodotti

Entrare, uscire, stazionare in una fase comporta lo svolgimento di determinate azioni

* + Realizzate come attività di specifici processi

Quanto tutti i needs sono ben definiti (*once and for all*), allora inizia *l’analisi dei requisiti*:



Il fatto che i needs siano definiti tutti all’inizio è utopistica; quando comincia ad emerge un concetto ed un eventuale prototipo, non si ha qualcosa di adattabile all’uso da parte degli utenti.

In particolare, tutte le funzioni realizzative sarebbero già state pensate (deve essere provato, altrimenti è idea solo sulla carta).

Da questo punto di vista, il modello risulta troppo fisso (*critica del modello sequenziale*).

* Difetto principale: eccessiva rigidità
  + Stretta sequenzialità: nessun parallelismo e nessun ritorno
  + Non ammette modifiche nei requisiti in corso d’opera
  + Visione rigida (burocratica) e poco realistica del progetto
* Correttivo 1: con prototipazione (avanzamento concettuale ma non di risultati)
  + Prototipi di tipo “usa e getta”
    - Per capire meglio i requisiti o le soluzioni (ma non come risolvere i problemi)
    - Strettamente all’interno di singole fasi
* Correttivo 2: con ritorni (si accorge che si sbaglia e, progressivamente, corregge)
  + Come «allenamenti» prima dell’atto definitivo, per imparare a fare sempre meglio ciò che serve a realizzare il prodotto

Questi ritorni dovrebbero condurre ad un miglioramento; tuttavia, è bene capire se trattasi di iterazione (sempre la stessa cosa, non convergendo all’avanzamento) oppure verso l’incremento (miglioramento).

Problemi particolarmente complessi richiedono di procedere a tentoni, spesso tramite iterazioni potenzialmente distruttive. Conviene, per Royce, procedere per passi incrementali, evitando di integrare il prodotto tutto-in-una-volta, mettendo tutti i moduli insieme (aka big-bang-integration), spendendo molto più tempo. Assai meglio adottare l’integrazione continua/continuous integration/CI, con iterazioni di avanzamento continue che portano a migliorare. Iterazione e incremento coincidono quando la sostituzione raffina ma non ha impatto sul resto.

Nei modelli incrementali (vantaggi):

* Possono produrre valore a ogni incremento
  + Un insieme crescente di funzionalità utili diventa presto e progressivamente disponibile
  + Magari a valle di una buona prototipazione, non usa-e-getta
* Procedere per incrementi riduce il rischio di fallimento
  + Senza però azzerarlo…
  + Come un ciclo *for*, da cui sappiamo quando usciremo, a meno di eccezioni
* Le funzionalità fondamentali (più necessarie) vanno sviluppate prima
  + Il loro uso frequente aiuta a verificare che siano solide

Nei modelli iterativi (vantaggi):

* Applicabili a qualunque modello di sviluppo
  + Ma comportando forte potenziale distruttivo
* Consentono maggior capacità di adattamento
  + Insorgere di problemi, cambio di requisiti, collasso tecnologico
* Ma comportano il rischio di non convergenza
  + Come un ciclo *while*, da cui non sappiamo per certo se e quando usciremo
* Tecniche di mitigazione
  + Decomporre il sistema in parti, lavorando prima su quelle più critiche, perché più complesse o con requisiti più incerti
  + Fissando un limite superiore al numero di iterazioni

In questi ultimi, però, ogni iterazione comporta un ritorno all’indietro nella direzione opposta all’avanzamento del tempo.

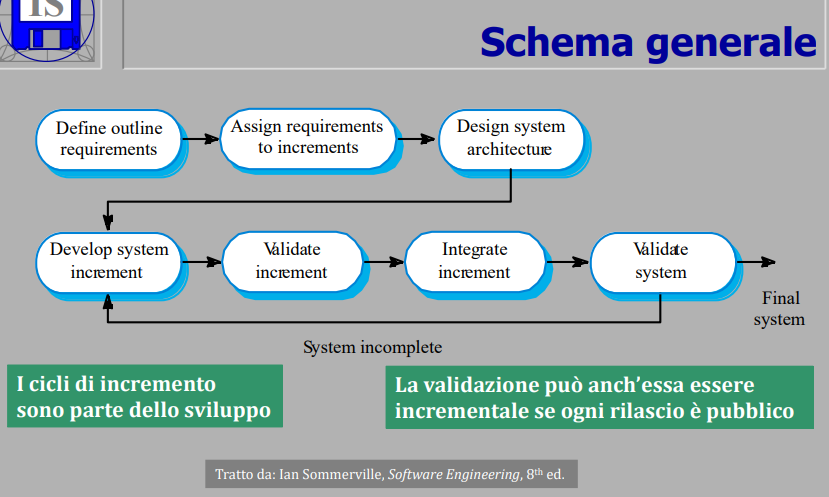
(Rischi modelli iterativi)

Quando si ha urgenza di avanzare in un progetto, magari non siamo sicuri e approssimiamo le attività “alla meglio” a causa di incertezze tecniche/limiti di tempo. Iterativamente, cerchiamo di eseguire il cosiddetto *refactoring*, “passando e migliorando iterativamente all’interno senza modificare l’esterno” sulla base del technical debt (qualcosa che rimane non sanato nello sviluppo e contamina tutte le altre attività; metterlo a posto dopo, è doloroso e porta a fallimenti). Ciò appare non appena si “smonta” la rigidità del modello sequenziale. A questo argomento è associato il [primo](https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9121630) dei tre link di approfondimento di questa lezione.

Qual è l’idea del modello incrementale? Esso itera, producendo un pezzo utile alla soluzione.

Esso prevede rilasci multipli e successivi, in cui ciascuno realizza un incremento di funzionalità.

I *requisiti* sono classificati e trattati in base alla loro importanza strategica (non realizzati una volta sola, si scoprono tutti avanzando nel design e nell’implementation).

* I primi incrementi puntano a soddisfare i requisiti più importanti/più evidenti sul piano strategico (nascondo idee interne al prodotto)
* Così i requisiti importanti diventano presto chiari e stabili, quindi più facilmente soddisfacibili
* Quelli meno importanti hanno invece più tempo per stabilizzarsi e armonizzarsi con lo stato del sistema

In questo modello, analisi dei requisiti e progettazione architetturale vengono svolte *una sola volta*: ciò è un’idea ambiziosa, perché richiede molta esperienza per poterlo fare bene. Ciò viene fatto:

* Per stabilizzare presto i requisiti principali
* Per stabilizzare presto *l’architettura* complessiva del sistema
* Per decidere preventivamente il numero di incrementi e i loro specifici obiettivi

La realizzazione è incrementale 🡪 Istanziazione del processo e numero di iterazioni prefissato

* Raffinando l’analisi dei requisiti e la progettazione di dettaglio, entro l’architettura adottata
* Il completamento dei primi incrementi serve a rendere disponibili le principali funzionalità

Un modello arrivato dopo e oggi utilizzato combatte l’insensatezza dello sviluppare partendo da zero (*from scratch*) cercando di usare cose già esistenti (*componenti*). Esso è il modello a componenti, in cui molto di quello che ci serve fare è già stato fatto (cercando qualcosa che serva al contesto che mi interessa). Tuttavia, molto di quello che faremo potrebbe servirci ancora.

* Analisi dei requisiti guidata dalla possibilità di riuso di quanto già esista
* Realizzazione che cerca di favorire riuso futuro

13/10/2022: Introduzione presentazione gruppi progetto 1o Lotto (Vardanega)

(Quantificazione *ideale*)

* 12 crediti - 300 ore di lavoro complessivo
* Gruppi di 6/7 Persone
* 95 ore in lezioni di teoria, pratica, monitoraggio attività, esercitazioni *(carrying out*, attività di completamento sulla base dei needs (*accomplishment*)
* 150 ore nel progetto didattico
  + 95 in attività rendicontate
  + 55 per auto-formazione su strumenti e metodi di lavoro utili al progetto
* 50 ore per studio personale in preparazione

La prima cosa è definire ruoli e azioni (way of working), passando da teaching e learning (apprendimento progressivo per cui la produttività è misurata in realizzazione, tale che in gruppo si possa incrementalmente comporre l’insieme di attività in modo ciclico e ripetuto, riducendo idealmente il tempo di esecuzione).

14/10/2022: Diagrammi delle classi (Cardin)

Il linguaggio naturale è poco espressivo nella creazione di un software; questo avviene soprattutto quando si ha a che fare con altre persone, è fondamentale farsi comprendere e comprendersi. Per poter parlare di software, usiamo UML (Unified Modeling Language) che è un linguaggio di modellazione standardizzato costituito da un insieme integrato di diagrammi, sviluppato per aiutare gli sviluppatori di sistemi e software a specificare, visualizzare, costruire e documentare gli artefatti dei sistemi software, nonché per la modellazione aziendale e di altri sistemi non software (più o meno complessi).

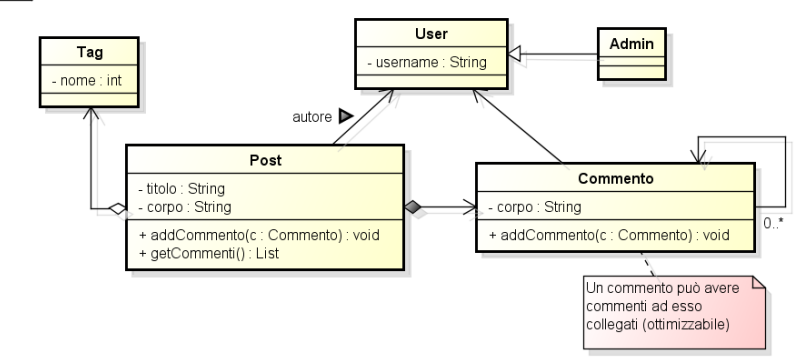
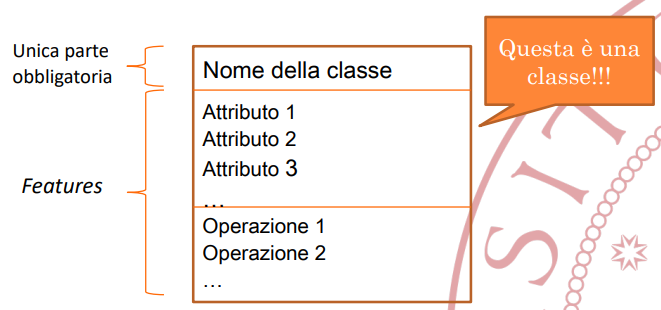
Ciò viene fatto principalmente per immagini, essendo forma efficace di comunicazione.

Vediamo i diagrammi delle classi, che descrivono i singoli oggetti, il loro tipo e gli attributi, tracciando anche la dipendenza tra le parti e le loro operazioni (con relazione statiche fra i tipi degli oggetti e per ognuno si ha una descrizione). Ne esistono di vari tipi: diagrammi di classe, di oggetti, di casi d’uso, di sequenza, di pacchetti, ecc.

Prendiamo uno strumento che ci permetta di realizzare diagrammi: <https://staruml.io/download>

Altri strumenti possibili:

* <https://app.diagrams.net/>
* LucidChart
* Draw.io

Per esempio, proviamo a modellare la classe Movie (nel modo che si vede sotto e a lato, come idea):

Gli attributi sono caratteristiche strutturali, ciascuno con una visibilità. L’associazione è rappresentata da una linea continua e orientata fra due classi e viene rappresentata la molteplicità; quindi, quanti oggetti possono far parte dell’associazione 1, 0..1, 0..\*, \*,… spesso interscambiabile con un attributo.

Ciò dipende dal contesto; ad esempio in quello che segue, la persona ha una sola città di nascita e dunque la molteplicità è uno.

Immagine che contiene tavolo

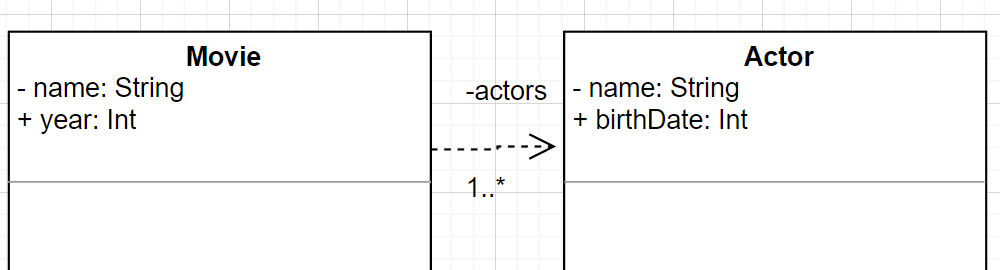
Descrizione generata automaticamente

Immagine che contiene tavolo

Descrizione generata automaticamenteEssi hanno una certa visibilità:

Per esempio, vogliamo inserire le *molteplicità* degli attributi, utile per le validazioni, tale da poter tracciare visualmente le dipendenze (nel nostro caso, Movie che dipende da Actor).

Graficamente, per ora:

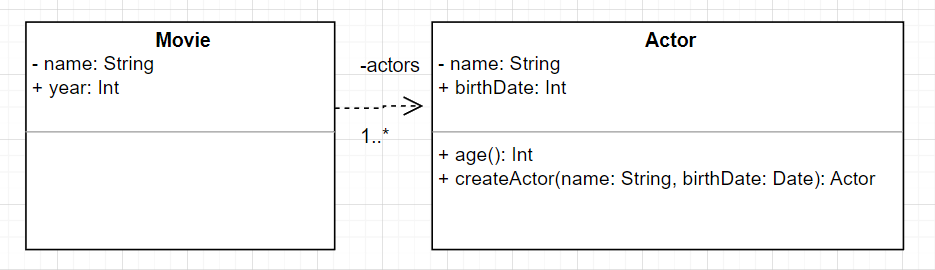


Gli attributi/metodi vengono chiamati *properties* (perché definiscono proprietà aggiuntive). Per questo, vengono definiti come privati/membri di classe; in aggiunta, per convenzione, possiamo avere dei metodi *getter* e *setter* per ogni attributo.

*In*: Attributo viene letto

*Out:* Attributo viene letto e modificato (*è più prone to error*, tale da generare più facilmente bug)

Le associazioni vengono rese tramite un *nome* (meglio fare così piuttosto che etichettare con un verbo) ed è meglio evitare le associazioni bidirezionali.



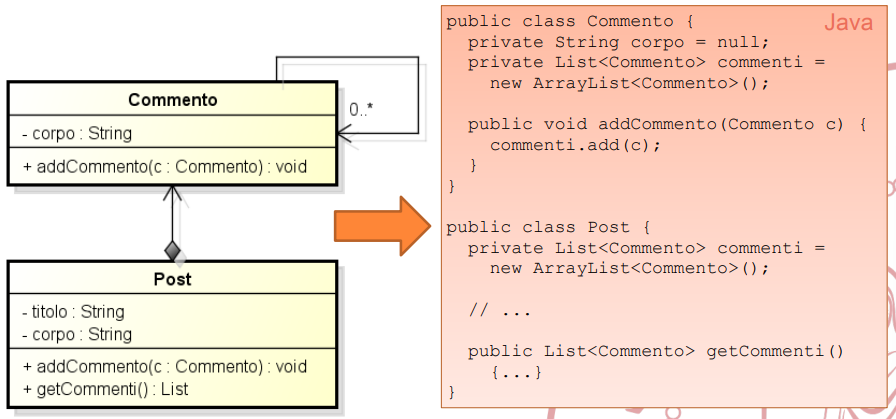
Gli attori sono visibili una volta sola; come tale, viene definito statico (quando questo succede, sottolineiamo il metodo). In StarUML viene fatto nel menù a lato spuntando “isStatic”.

La classe Movie viene definita *final* e sarebbe immutabile; tuttavia, estrapolando un qualsiasi oggetto dalla classe, si modifica. Inserendo un metodo *getActors()* viene reso immutabile,

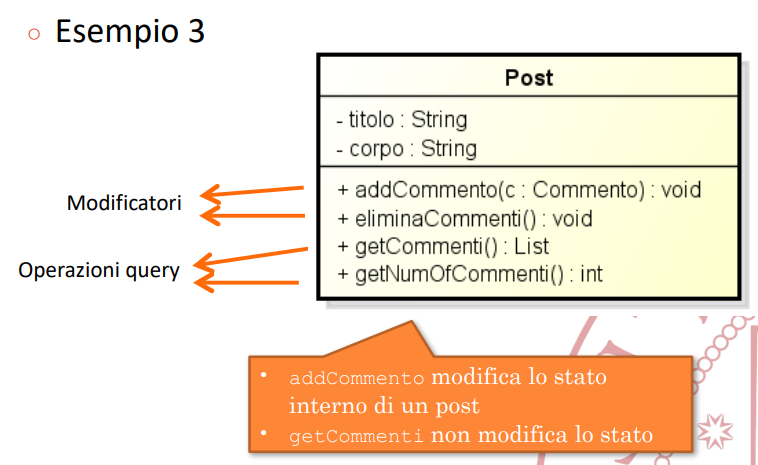
Immagine che contiene tavolo

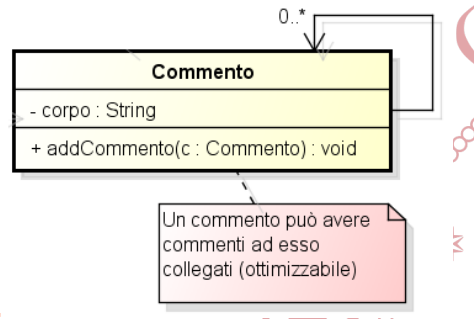
Descrizione generata automaticamente

Un esempio migliore di come vengano rese classi e relative proprietà segue:



Vengono definite *operazioni* le azioni che la classe sa eseguire (comportamento) oppure un *servizio* che ad istanza viene chiesto (query, modificatori); un’operazione *non è* un metodo (a meno che non si parli di polimorfismo). Un esempio di tipo di operazioni:

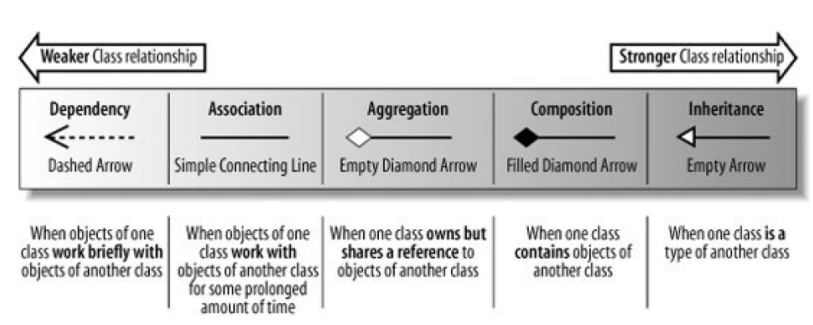




Per aggiungere dei commenti o delle note, esse vengono aggiunte come singole/solitarie e generalmente legate ad un elemento grafico (classe) (su StarUML vengono aggiunte nel menù a sx nella penultima categoria tra quelle a tendina (menù *Annotations*, selezionando *Note*). Piccolo esempio a lato.

Si ha una *relazione di dipendenza* quando “tra due elementi di un diagramma se la modifica alla definizione del primo (fornitore) può cambiare la definizione del secondo (client)”. UML permette di modellare ogni sorta di dipendenza e le dipendenze vanno minimizzate (da inserire solo quando danno valore aggiunto).

Ci sono vari tipi di relazione di dipendenza, come segue, ordinati come visto qui:



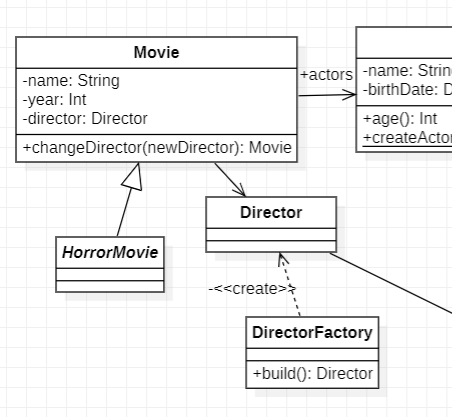
Maggiore è la quantità di codice condiviso tra due tipi, maggiore la dipendenza tra essi.

Listiamo anche le dipendenze UML, in base al loro significato:

Immagine che contiene tavolo

Descrizione generata automaticamente

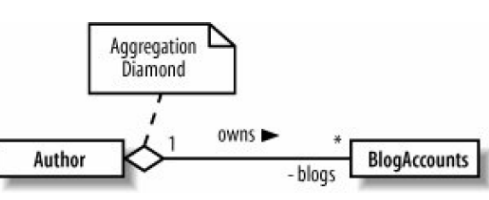
Aggiungiamo una classe Director ed una classe DirectorFactory è quella che costruisce i Direttori e introduciamo una dipendenza di tipo Create (selezionare sul menù Toolbox di sx la voce *Dependency* su StarUML, freccia tratteggiata), introducendo una cosa del genere:



Per scriverle *<<create>>,* occorre usare il menù a dx e scrivere manualmente sulla freccia tratteggiata tra *DirectorFactory* e *Director* all’interno di Properties la voce *<<create>>* sul campo *name*.

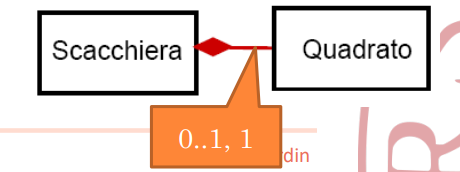
Ci possono essere vari tipi di relazione:

quella di *aggregazione* (part of/parte di, tale che gli attributi possano essere condivisi; la classe aggregata è responsabile della costruzione degli aggreganti):

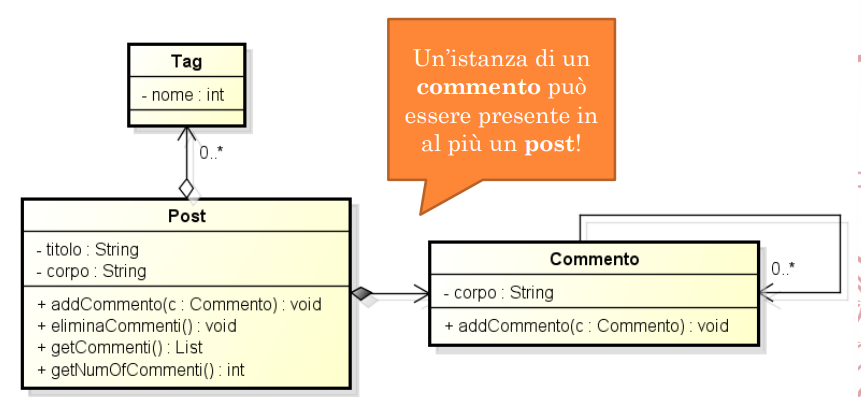


La *composizione* è simile all’aggregazione, ma gli aggregati appartengono ad un *solo aggregato*.

Solo l’oggetto intero può creare e distruggere le sue parti.



Un esempio secondo le slide di aggregazione e composizione è come segue:



Introduciamo la classe astratta *HorrorMovie*, con freccia piena (freccia di tipo *Generalization* in StarUML) e introduciamo l’interfaccia (*Interface* in StarUML, come si vede a lato):

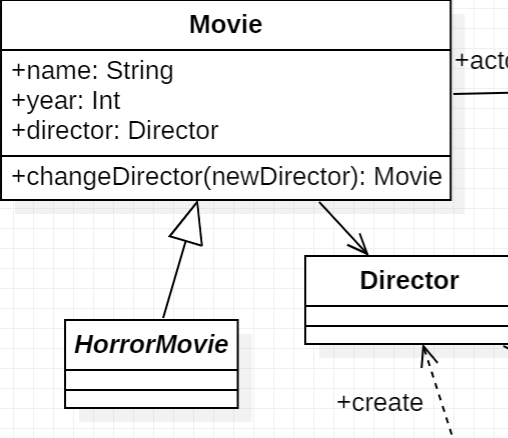
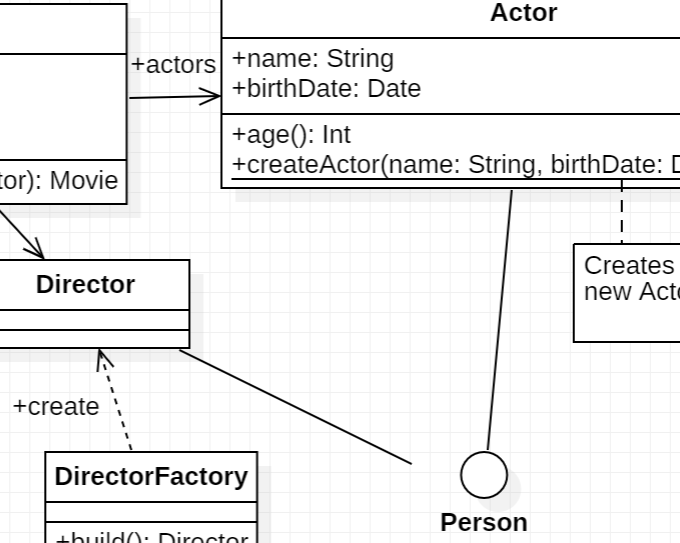


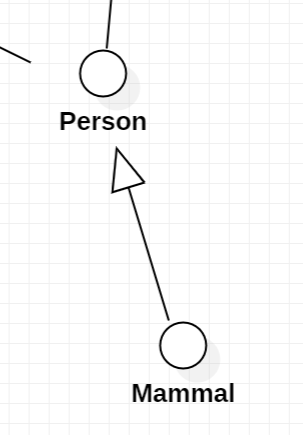
Immagine che contiene parete

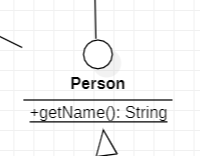
Descrizione generata automaticamente

Ora, colleghiamo per associazione *Person* verso *Actor* e *Director*:



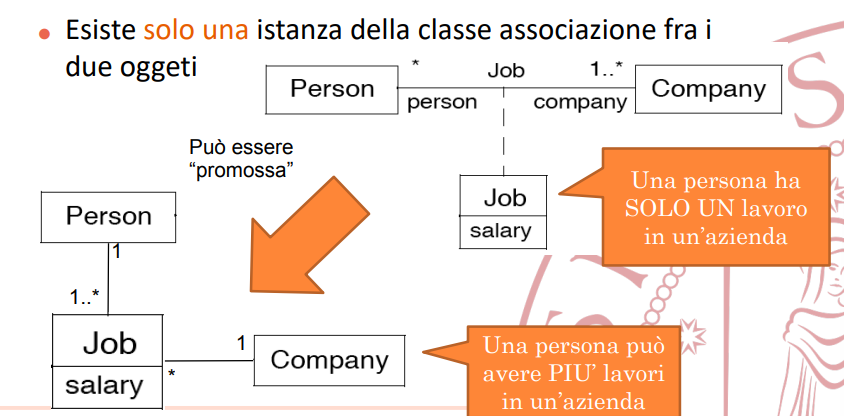
UML definisce solo interfacce e classi astratte; se ho implementazioni, essa è una classe astratta:

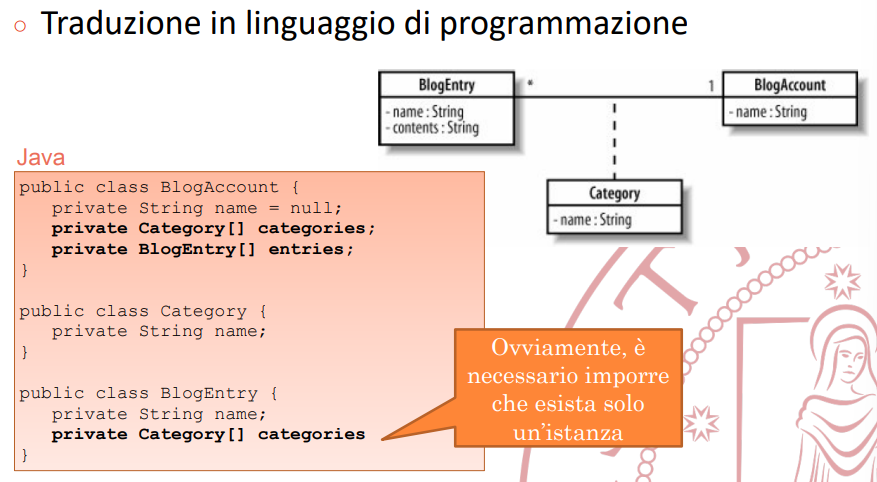




Le interfacce non hanno attributi; aggiungiamo il metodo *getName()* a Person; esso sarà un metodo virtuale puro (per aggiungere un metodo ad un’interfaccia, cliccando tasto dx sull’interfaccia Person e Format e togliere la spunta “Suppress Operations”, poi cliccando getName() e poi cliccando su isStatic sul menù a dx):

Un altro esempio sono le *classi di associazione*, che aggiungono attributi e operazioni alle associazioni.

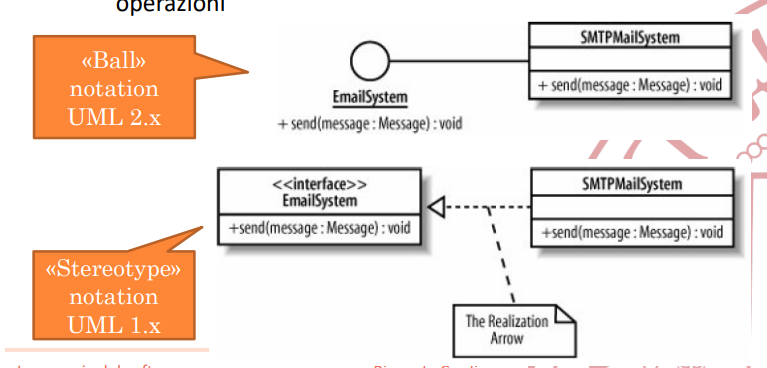


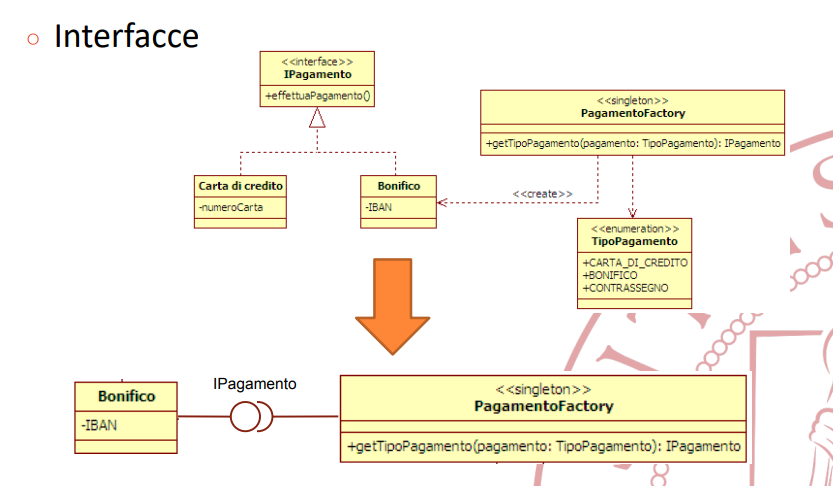


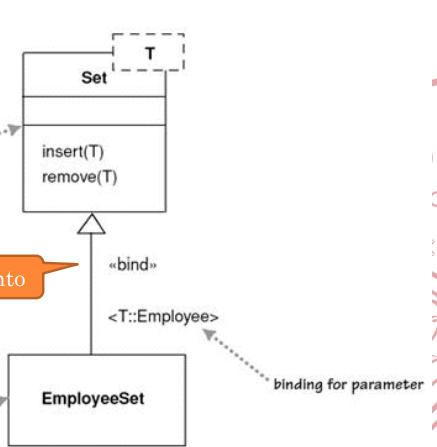
Similmente, si hanno *generalizzazioni* ad esempio quando con due classi A e B, ogni oggetto di B è anche un oggetto di A e ciò equivale all’ereditarietà dei linguaggi di programmazione. Le proprietà della superclasse non si riportano nel diagramma della sottoclasse (a meno di override); si ha il principio di sostituibilità (sottotipo != sottoclasse). Esempio pratico sono gli attributi protected, visibili anche dalle classi derivate.

Prendiamo poi le *classi astratte*, che ricordiamo essere classi che *non possono essere istanziate*; le operazioni astratte non hanno implementazione e queste vengono definite con nome in corsivo. Esse non devono essere molte, anzi, tendenzialmente sono poche (per non confondere l’UML).

Similmente, abbiamo le *interfacce*, classi prive di implementazioni; una classe realizza un’interfaccia se ne implementa le operazioni. Per come visto sopra; si ha una notazione a sfera/palla (da UML 2.x in avanti), precedentemente era indicata a rettangolo:

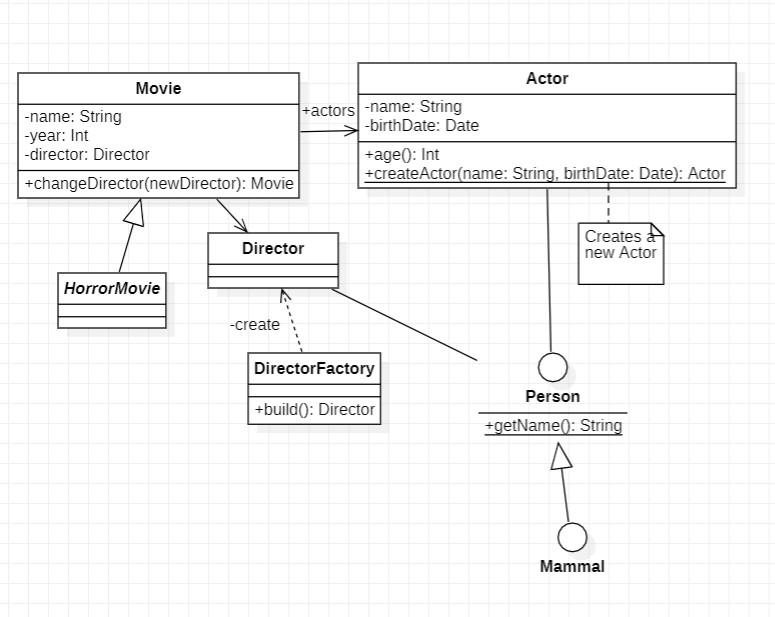




Un sottotipo è diverso da un *is-a*, dato che a livello generale hanno le stesse caratteristiche, ma un sottotipo normalmente viene solo istanziato, un *is-a* possiede sempre le caratteristiche che servono; ciò viene visto come dipendenza *<<istantiate>>*.

Le classi Parametriche hanno *T* come “segnaposto”; con parametrico si intende il concetto di *template* in C++ o di *generics* in Java, tale che per ereditarietà si possa per binding realizzare un’operazione collegata (es. setting dell’impiegato/employee, specificando dove sto realizzando l’operazione):

Il risultato finale dell’esempio mostrato è come segue:



Consigli:

- Creare diagrammi ricchi di concetti (cominciando prima dai concetti semplici)

- Una prospettiva concettuale permette di esplorare il linguaggio di un business, mantenendo una notazione semplice

- Concentrarsi sul disegno delle parti più importanti