Dispense di Ingegneria del software

Leonardo De Faveri

A.A. 2021/2022

Indice

Capitolo Nr.1

Introduzione

1.1 Definizione e finalità dell'Ingegneria del software

Iniziamo questa trattazione con una definizione formale di *Ingegneria del Software* (*IS*):

Definizione 1 - Ingegneria del software.

L'Ingegneria del software si definisce come:

1. Applicazione di una strategia sistematica, disciplinata e misurabile allo sviluppo, esercizio e manutenzione di un software;
2. Studio delle strategie di cui al punto (1);

L'obiettivo finale dell'*IS* è quindi lo sviluppo di un software di qualità.

1.1.1 L'IS come tecnologia

Per comprendere meglio cosa sia questa disciplina, possiamo immaginarla come una tecnologia suddivisa in livelli.

Fig. 1.1: Schema degli strati dell'*Ingegneria del software*

Alla base abbiamo l'obiettivo, ovvero lo sviluppo di un prodotto di qualità. Per fare ciò, esistono diversi *processi* che definiscono e organizzano le attività da svolgere. Similmente, il modo in cui le attività devono essere svolte e collegate tra loro, è stabilito dai *metodi*. Infine, per rendere efficiente il corso delle attività, esistono dei supporti automatizzati detti *strumenti*.

Per rendere più chiaro quanto appena detto possiamo anche dire che:

- *Processi*: definiscono e organizzano le attività da svolgere per realizzare il prodotto finito;
- *Metodi*: indicano come portare a termine le singole attività e anche come mettere in relazione attività diverse (e.g. analisi dei requisiti, progettazione dell'architettura, scrittura di documentazione);
- *Strumenti*: permettono di rendere più efficiente il lavoro (e.g. software per lo sviluppo del codice, linguaggi di modellazione, tool per il testing, ...);

Capitolo Nr.2

Processi di sviluppo

Definizione 2 - Processo di sviluppo.

Un *processo di sviluppo* stabilisce quando e come qualcuno deve fare cosa, per raggiungere un obiettivo.

Chiaramente, a seconda del progetto, può essere più conveniente seguire un certo tipo di *processo* invece di un altro. È anche possibile far valutare il proprio processo di sviluppo, così da garantire ai propri clienti che verranno mantenuti determinati livelli di qualità.

2.1 Modelli per processi di sviluppo

Vediamo ora una carrellata dei più diffusi *modelli per processi di sviluppo*.

2.1.1 Modello a cascata

Il *modello a cascata*, o *waterfall*, descrive l'approccio classico allo sviluppo di un software.

Con questo tipo di *processo*, una volta che i requisiti sono noti, il lavoro procede in maniera lineare e si conclude con la consegna del prodotto finito al cliente.

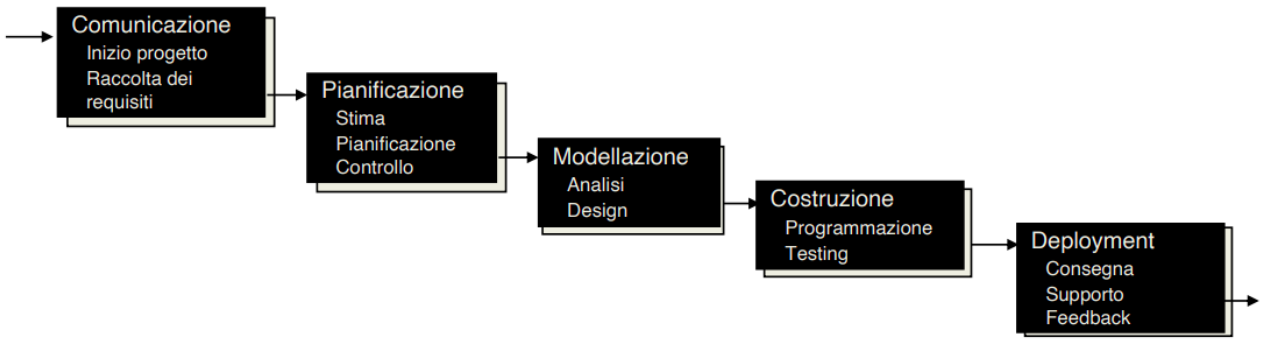


Fig. 2.1: Schema del *modello a cascata*

Il problema con questo tipo di approccio è che il mancato coinvolgimento del cliente (o in generale di parte degli stakeholder) nelle fasi intermedie, porta la maggior parte dei prodotti a disattendere le richieste del cliente e quindi al fallimento del progetto. Inoltre, qualora si rendesse necessario una modifica a quanto fatto in una fase di sviluppo precedente, sarebbe necessario un impiego di tempo e risorse non indifferente.

Per queste ragioni, il *modello a cascata* non viene più utilizzato, se non in progetti di dimensioni molto ridotte.

2.1.2 Modello incrementale

Nel *modello incrementale* lo sviluppo del software procede per aggiunta di funzionalità a una versione base del software. Può essere visto come uno sviluppo in contemporanea di più *processi con modello a cascata*.

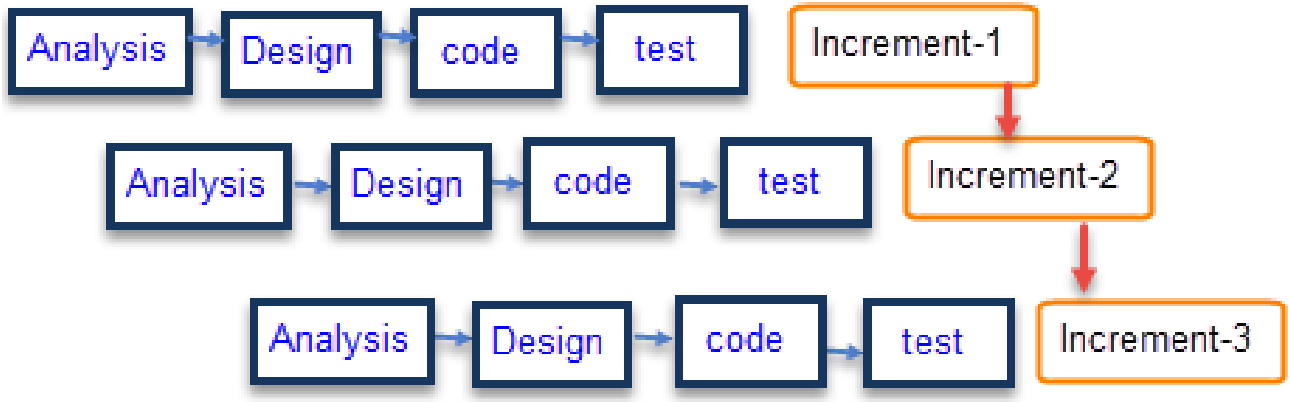


Fig. 2.2: Schema del *modello incrementale*

Questo modello argina parzialmente alcuni dei problemi del precedente, in quanto ogni incremento inizia con il coinvolgimento del cliente e, se le aggiunte sono minime, si riducono anche i costi in caso di modifiche o errori.

2.1.3 Modello a prototipi

Questo tipo di approccio va bene quando il cliente non ha un'idea chiara del prodotto che vuole realizzare. Il team di sviluppo produce quindi un prototipo basandosi sulle poche indicazioni ricevute e lo sottopone al cliente.

Se il cliente valida il prototipo, questo viene fatto evolvere per arrivare a un prodotto finito, altrimenti si ricomincia con un nuovo prototipo.

Prototipo

Definizione 3 - Prototipo.

Un *prototipo* è una rappresentazione di un prodotto o di un sistema, o di una sua parte, che, anche se in qualche modo limitata, può essere utilizzata a scopo di valutazione.

Il *prototipo* non deve necessariamente essere un prodotto funzionante, può anche essere un modello "finto" che dia soltanto un'idea di quali potrebbero essere le funzionalità e il *look and feel* del prodotto finito.

Per realizzare il prodotto finito è meglio non partire dal prototipo, in quanto, spesso, nel suo sviluppo, si sono sacrificati gli aspetti relativi alla qualità per prediligere invece la velocità di realizzazione. È quindi preferibile ricominciare lo sviluppo usando il *prototipo* come riferimento per le funzionalità e le caratteristiche da implementare.

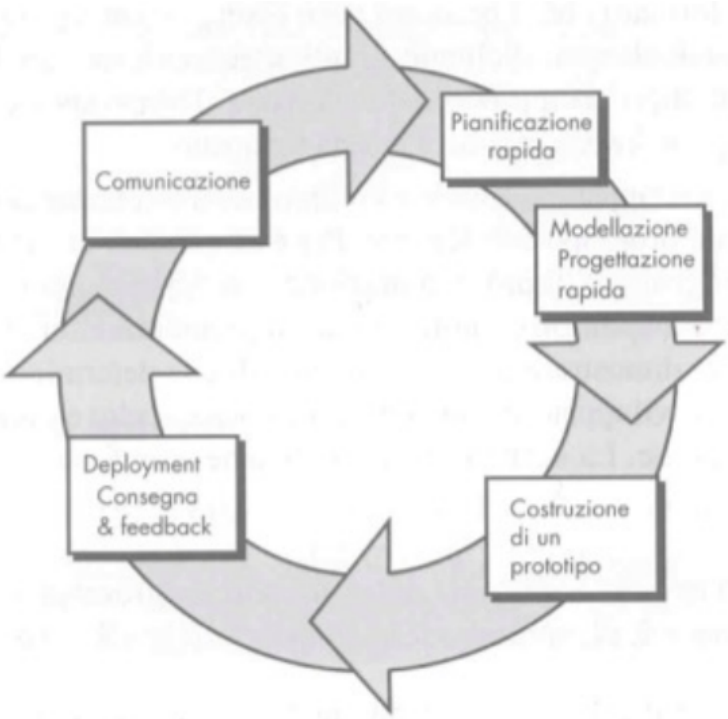
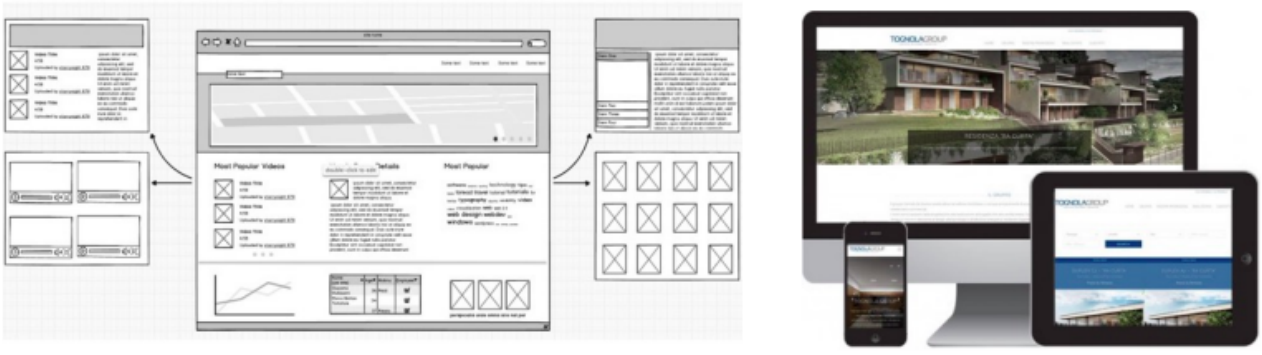


Fig. 2.3: Schema del *modello a prototipi*

Wireframe o mock-up Come detto il *prototipo* non deve necessariamente essere un qualcosa di funzionante, ma può limitarsi a dare un'idea delle funzioni e dall'aspetto del prodotto finito. In questo contesto, si inseriscono due strumenti: i *wireframe* e i *mock-up*.

- *Wireframe*: permette di capire quale sarà l'esperienza utente, la cosiddetta *user experience*. Si tratta di una bozza grafica, fatta anche su carta, priva di tutti gli elementi di design, cioè mancano colori, immagini e quant'altro;
- *Mock-up*: contiene gli elementi di design e serve a dimostrare quello che potrebbe essere il *look and feel* del prodotto;



(a) *Wireframe* (b) *Mock-up*

Fig. 2.4: *Wireframe* e *mock-up*

2.1.4 Modello a spirale

Nel *modello a spirale* si segue approccio iterativo: ad ogni "giro" viene consegnata una versione del prodotto. Questo è simile a quanto avviene nel *modello incrementale*, ma mentre nel *modello incrementale* si costruisce su di una versione base, qui è sempre possibile ricominciare da capo. Spesso, infatti, si inizia con dei prototipi e quando un prototipo soddisfa le richieste del cliente si procede con lo sviluppo del prodotto vero e proprio.

Si può dire che questo approccio unisca l'iteratività del *modello a prototipi* alla *sistematicità* di quello a *cascata*.

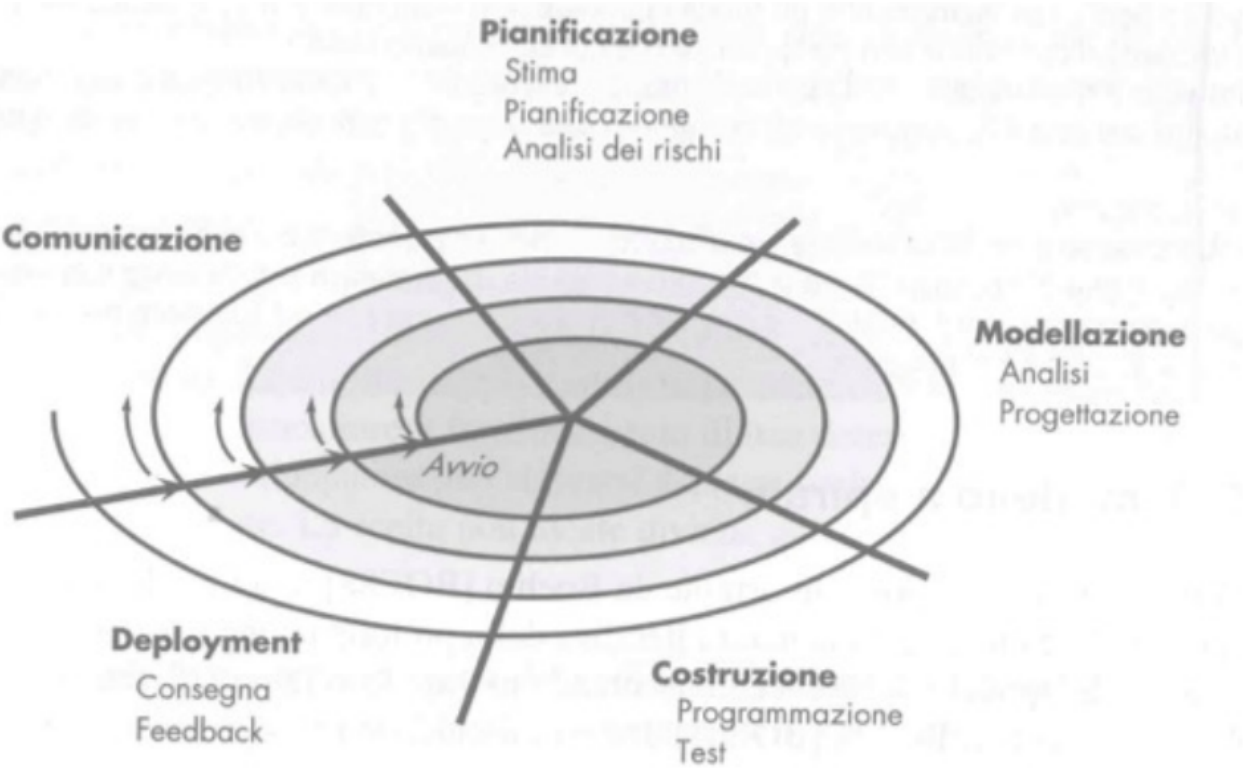


Fig. 2.5: Schema del *modello a spirale*

2.1.5 Modello di sviluppo a componenti

Vengono realizzati componenti software con funzionalità e interfacce ben definite. I singoli componenti vengono poi collegati per realizzare il prodotto finale.

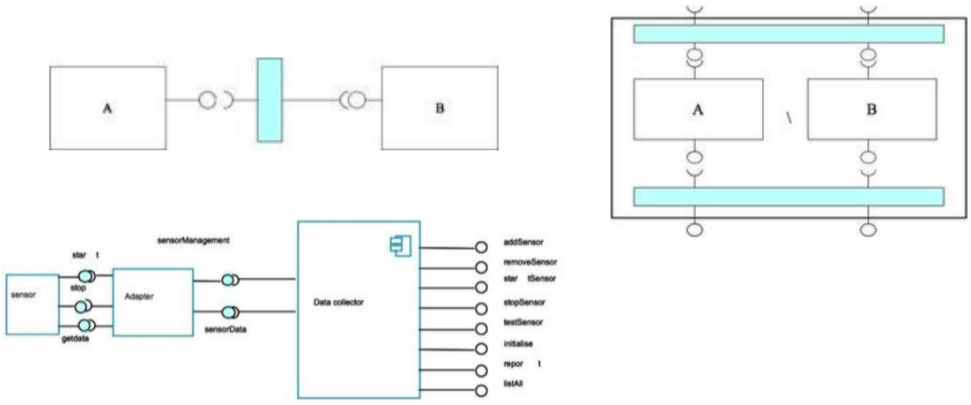


Fig. 2.6: Schema del *modello di sviluppo a componenti*

Questo approccio permette di dividere facilmente il lavoro in problemi più semplici e contemporaneamente di ridurre i costi per eventuali interventi correttivi o aggiunte. Tuttavia, spesso è necessario produrre del codice ausiliario detto *glue-code* per consentire i componenti di comunicare correttamente tra loro.

Architetture orientate ai servizi La *Service Oriented Architecture* (*SOA*) è il modello che sta alla base di particolari architetture, oggi sempre più diffuse, che si basano proprio sullo sviluppo di servizi distribuiti e indipendenti tra loro che vengono poi combinati per realizzare software complessi. I vantaggi di questo tipo di architetture sono la scalabilità e la modularità, che vengono favorite proprio dall'indipendenza dei singoli servizi.

Uno esempio di questo è l'*architettura a microservizi*, o *Microservices Architecture* (*MA*), nella quale componenti con funzionalità minime vengono combinate per realizzare software complessi. Questo tipo di sviluppo è particolarmente adatto per applicazioni che hanno la necessità di scalare ed evolversi rapidamente soprattutto sfruttando il cloud.

2.1.6 Model-driven developement

Secondo questo approccio, lo sviluppo del prodotto procede di pari passo con la modellazione del sistema software: i modelli evolvono con il procedere dello sviluppo. Si parte cioè, con la creazione di un modello dell'architettura nel suo insieme e, mano a mano che il lavoro procede, vengono aggiunte alla rappresentazione le descrizioni di componenti più specifici. Di conseguenza, il livello di dettaglio del modello andrà ad aumentare progressivamente.

Esistono linguaggi e strumenti appositi che permettono sia di definire i modelli che di implementarli in codice.

2.2 Metodologie agili

Esistono modelli di sviluppo cosiddetti *agili*, o *agile*, che propongono paradigmi il cui focus è sull'interazione col cliente piuttosto che sul prodotto. Nelle *metodologie agili* infatti, il processo di sviluppo del software cerca di coinvolgere quanto più possibile il cliente, ottenendo in questo modo un prodotto maggiormente conforme alle sue richieste.

Proprio questa caratteristica fa si che i progetti realizzati in questo modo abbiano un tasso di successo molto superiore rispetto a quelli che usano il classico *modello a cascata*.

Non bisogna erroneamente pensare che i *processi agili* siano totalmente contrapposti ai modelli più tradizionali. Infatti, molti dei concetti di queste metodologie sono adattamenti dei migliori concetti dell'*ingegneria del software* tradizionale. Ad esempio, viene mantenuta l'iteratività caratteristica del *modello incrementale*.

2.2.1 Modello scrum

Il *modello scrum* (mischia) è un insieme di pratiche e regole che consentono di ridurre l'overhead amministrativo. In particolare, il lavoro viene diviso in task che sono assegnati a piccoli gruppi di lavoro, tipicamente di 3-4 componenti. Ogni team ha molta libertà nell'organizzazione del proprio lavoro e nella distribuzione delle responsabilità tra i propri componenti, ad esempio nella scelta del proprio responsabile.

Lo sviluppo del software procede poi in maniera incrementale e con attività continue di testing e manutenzione.

Interessante è il modo in cui è organizzato il progetto nel suo insieme: il progetto viene diviso in blocchi rapidi di lavoro detti *sprint* alla fine dei quali viene consegnata al cliente una versione

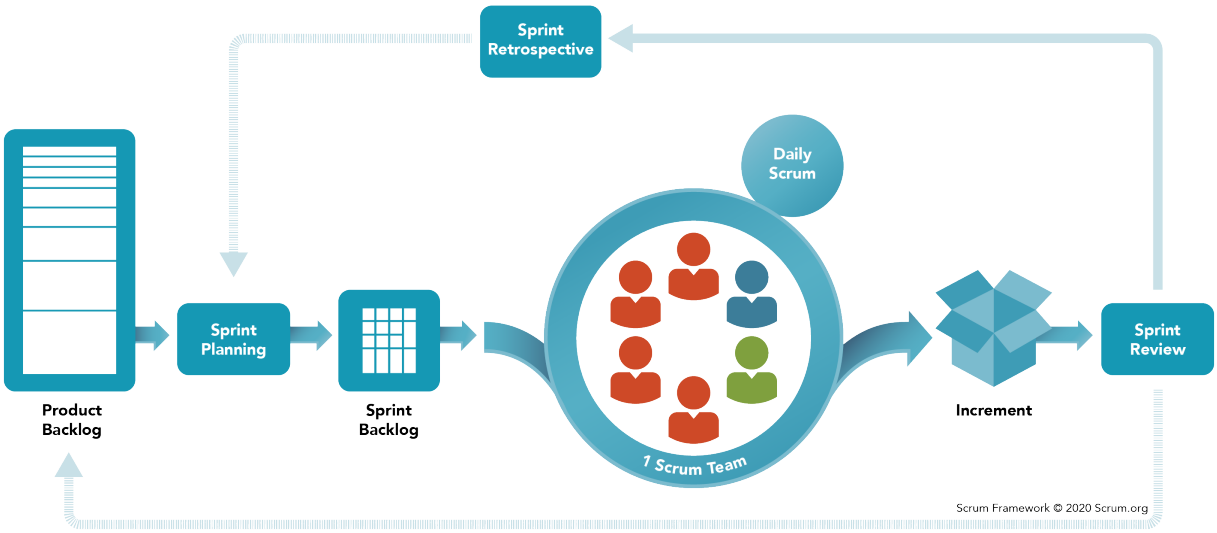


Fig. 2.7: Schema del *modello scrum*

del software. Vengono inoltre svolte giornalmente riunioni dei singoli team di sviluppo, dette *daily scrum* nelle quali ci si confronta sul lavoro svolto e quello rimanente. Il *modello scrum* prevede anche che vengano definiti i *backlog*, cioè i dettagli del lavoro da fare nell'immediato futuro. Questo permette di avere una visione estesa dello stato del progetto.

2.2.2 Modello extreme programming

Questa metodologia ha un approccio *object-oriented* allo sviluppo ed è caratterizzata da 4 valori e 12 pratiche. I valori sono:

1. *Semplicità*;
2. *Comunicazione*;
3. *Testing*;
4. *Coraggio*;

Le 12 pratiche possono invece essere raggruppate in 4 aree fondamentali:

1. *Feedback a scala fine*: la consegna del software avviene tramite frequenti rilasci che incrementano anche in minima parte le funzionalità del prodotto;
2. *Processo continuo*: l'integrazione delle modifiche nel sistema viene fatta frequentemente, così da limitare eventuali conflitti con il codice esistente. Questo tipo di processo è strettamente sequenziale, cioè un solo programmatore alla volta integra le proprio modifiche, testa e rilascia il software;
3. *Comprensione condivisa*: la pianificazione del lavoro è basata su un insieme di *user story* definite dal cliente;
4. *Benessere dei programmatori*: il codice viene modificato, anche nella sua architettura, per migliorarne le prestazioni e la comprensibilità;

Pianificazione a user story All'avvio del progetto, il cliente fornisce al team di sviluppo (nel suo intero o un gruppo specifico) una descrizione delle funzionalità e delle caratteristiche che dovrà avere il software. Questa descrizione di insieme viene suddivisa in componenti più semplici detti *user story*.

A ogni *user story* il cliente assegna un valore, in modo che venga data priorità alle *user story* col valore più alto. Il team di sviluppo valuta le *user story* e a ciascuna assegna un costo, descritto come tempo di sviluppo. Se il costo è troppo alto, viene chiesto al cliente di spezzare la *user story* in casi più semplici.

In fase di sviluppo il team può decidere come implementare le *user story* (in ordine di priorità, in ordine di costo, tutte insieme, ...), ciò che importa è che vengano concordate con il cliente le date di rilascio e quali *user story* dovranno essere incluse in ogni release.

Approccio al testing L'*Extreme Programming* stabilisce anche un approccio al testing. Vengono implementati degli *Unit test* che vanno a convalidare il funzionamento di ciascuna *user story*. I test non valutano l'implementazione complessiva delle *user story*, ma si assicurano soltanto che a ogni input corrisponda un output corretto.

Questa caratteristica permette agli sviluppatori di concentrarsi in prima battuta soltanto sull'aspetto *estensionale* del codice. Grazie a ciò, il tempo di rilascio della prima versione viene ridotto. Future attività di *refactoring* andranno a migliorare l'aspetto *intensionale* del codice e l'efficienza complessiva delle release successive.

L'attività di testing può essere automatizzata mediante l'utilizzo di *testing suite universali* col risultato che *integration* e *validation test* possono essere eseguiti quotidianamente. I cosiddetti *acceptance test*, o *customer test*, cioè l'attività di verifica dell'utente finale diventano quindi più semplici, in quanto si possono concentrare soltanto sulle funzioni e le caratteristiche globali del sistema.

Pair programming La programmazione avviene in coppia, cioè a ogni workstation lavorano due programmatori. Questo garantisce una maggiore qualità del software prodotto, perché vi è sempre un controllore e, inoltre, semplifica e velocizza la risoluzione dei problemi.

2.2.3 DevOps

Una figura che sta assumendo maggiore importanza nelle grandi aziende e quella del *DevOps* (*Developement + Operations*). In ambienti strutturati e complessi i componenti software vengono installati su infrastrutture altrettanto complesse. Il compito del *DevOps* è quello di coordinare il lavoro di sviluppo software con quello di manutenzione dell'infrastruttura, in modo da efficientare questa interazione e ridurre l'insorgere di problemi.

Capitolo Nr.3

Linguaggi di modellazione

Nello sviluppo di un software, prima di passare all'implementazione vera e propria, è consigliabile ragionare su una versione semplificata della realtà che si vuole realizzare. Tale visione semplificata è detta *modello*.

Definizione 4 - Modello.

Un *modello* è la rappresentazione semplificata di un'entità. Sono presenti solo le informazioni importanti e astrazioni dei concetti più complessi.

Un buon *modello*, oltre che rappresentare in modo semplice la realtà, deve anche poter essere compreso da soggetti terzi rispetto ai suoi creatori. Cioè, è necessario che venga creato usando un linguaggio standardizzato e diffuso.

Nell'*ingegneria del software* esistono diversi linguaggi a seconda di ciò che si vuole rappresentare ed è quindi importante saper scegliere quello corretto. Tra le caratteristiche di un buon linguaggio vanno considerate:

- *Semplicità di utilizzo*: deve consentire di esprimersi con facilità e contemporaneamente deve essere di facile comprensione;
- *Fruibilità diffusa*: deve poter essere compreso anche da soggetti non esperti in materia;
- *Formalità*: un linguaggio formale e preciso può essere usato per descrivere realtà e concetti complessi senza ambiguità e a livelli d'astrazione diversi;

Alcuni esempi di linguaggi di modellazione sono il linguaggio *Entity-Relationship* per la progettazione concettuale di database, il *BPMN* (*Business Process Modelling Notation*) per la rappresentazione dei processi di business.

3.1 Modellare i requisiti di sicurezza

Nella realizzazione di un software di qualità non si può non considerare l'aspetto della sicurezza. La definizione dei requisiti di sicurezza può essere realizzata con diversi linguaggi. Questo tipo di linguaggi rientra nella definizione di *Security Requirements Engineering* (*SRE*).

3.1.1 Misuse-case diagram

Questo linguaggio è basato sulla sintassi *UML* (*Unified Modeling Language*) e consente di rappresentare i casi d'uso improprio del software. Cioè, consente di descrivere quali utilizzi (percorsi

di utilizzo) del software potrebbero avere conseguenze sgradite. Visualizzare questi percorsi permette all'ingegnere del software di progettare un'architettura che li neutralizzi.

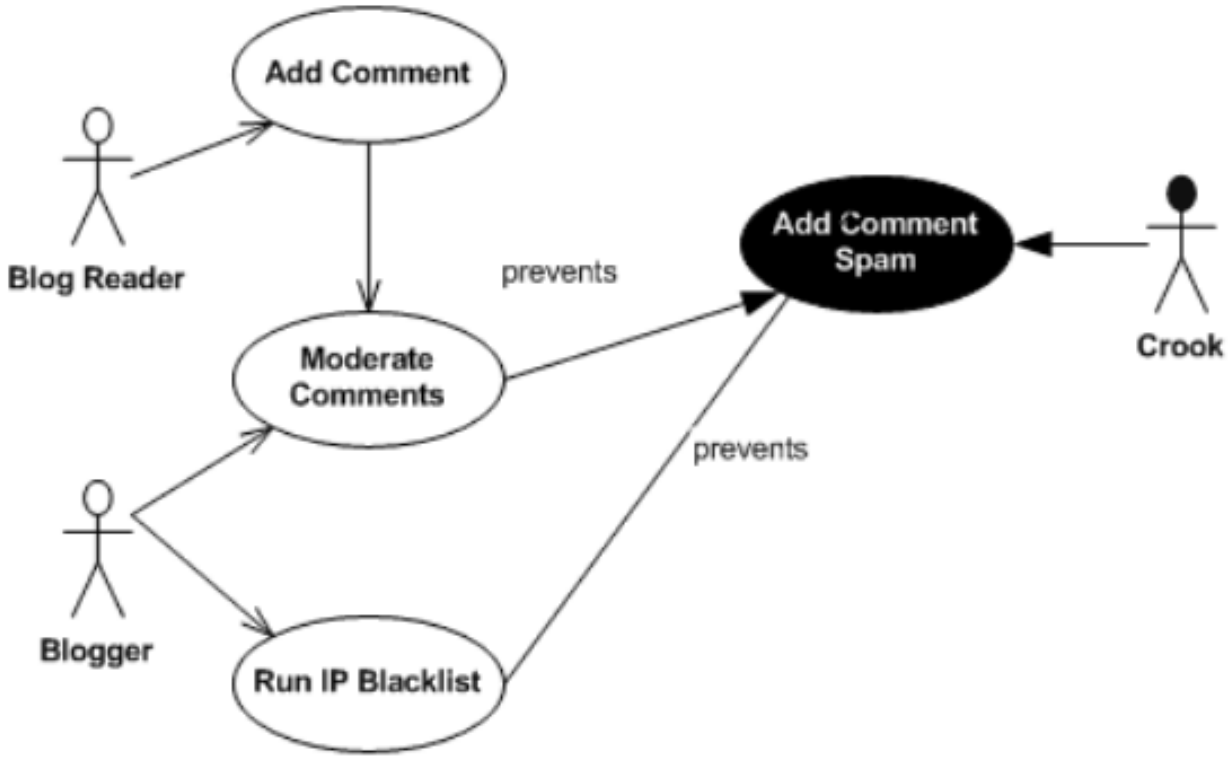


Fig. 3.1: Esempio di *misuse-case diagram*

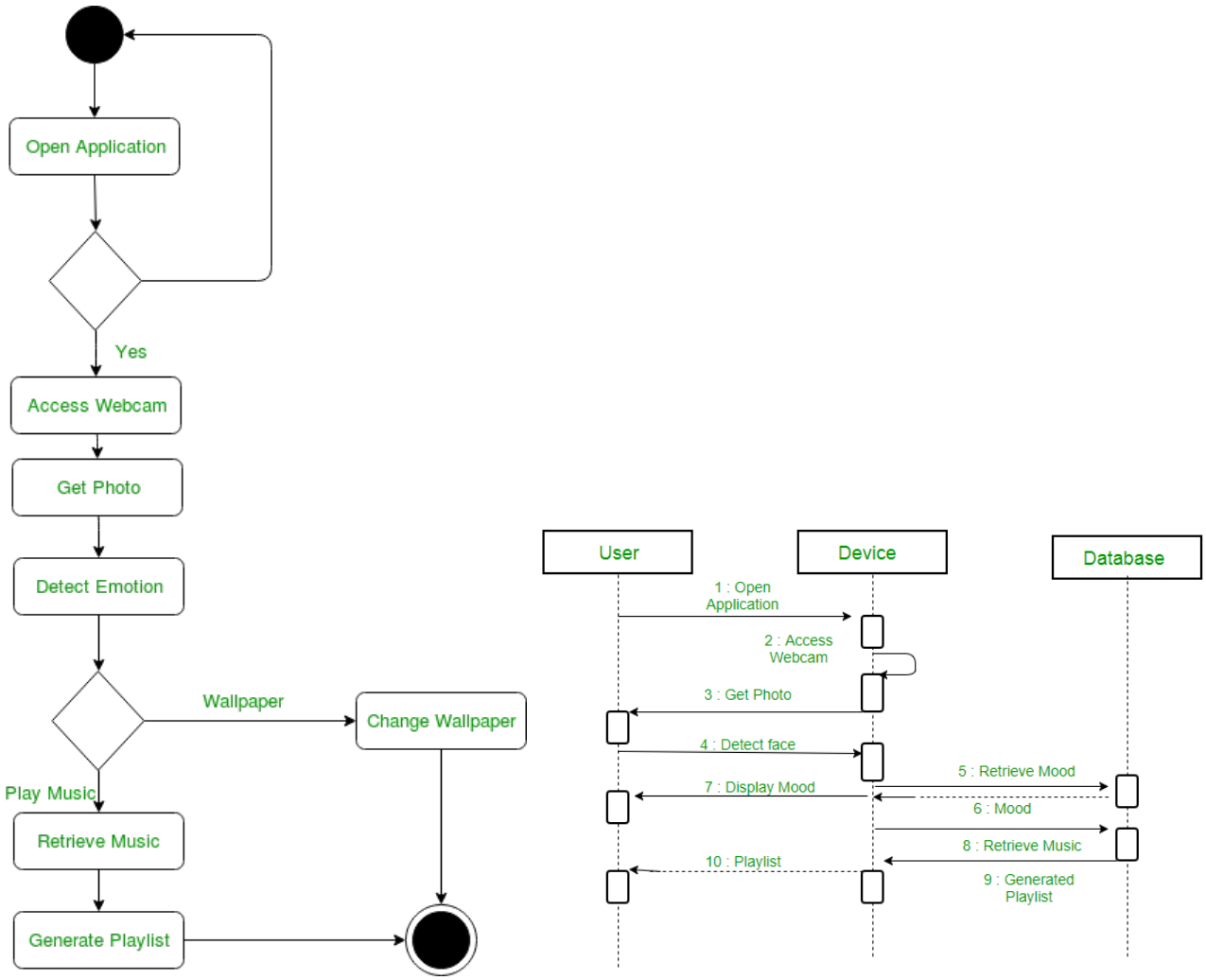
3.1.2 Linguaggi goal-oriented

Alcuni linguaggi si concentrano invece sull'ipotizzare quali potrebbero essere gli obiettivi di utenti malintenzionati, i cosiddetti *anti-goal*, così da poter poi a progettare il software in modo che quegli obiettivi non siano realizzabili.

3.2 Linguaggio UML

L'*UML* è certamente il linguaggio più usato nell'*ingegneria del software*. Grazie ai suoi dialetti può essere usato per modellare diversi aspetti di un software. Gli ambiti d'uso dell'*UML* sono:

- *Use-case diagram*: letteralmente il *diagramma dei casi d'uso*, rappresenta cioè quello che ci si aspetta possa fare l'utente;
- *Activity diagram*: descrive come ci si aspetta che vengano svolte le singole attività;
- *Sequene diagram*: descrive in modo sequenziale l'interazione tra due oggetti;
- *Class diagram*: consente di definire la struttura di una o più classi nell'ambito della OOP;
- *Deploy diagram*: raffigura i requisiti e i passi da compiere per il rilascio del software;



(a) *Activity diagram* (b) *Sequence diagram*

Fig. 3.2: *Activity diagram* VS *Sequence diagram*

Activity diagram VS Sequence diagram Questi due diagrammi potrebbero sembrare simili, ma in realtà hanno una differenza piuttosto marcata.
Come si può vedere l'*activity diagram* descrive il flusso delle operazioni necessarie per passare da un'attività a un'altra, mentre il *sequence diagram* descrive lo scambio di messaggi tra un oggetto e un altro.

Si può comprendere meglio la differenza tra i due se si vede l'*activity diagram* come un *diagramma di flusso* e il *sequence diagram* come il diagramma descrittivo di un evento.

3.2.1 Linguaggio OCL

L'*OCL* (*Object Constraint Language*) è una versione estesa di *UML*. È un linguaggio funzionale e consente di rappresentare vincoli ed espressioni in modelli *object-oriented*.

Capitolo Nr.4

Requisiti

4.1 Requisiti non funzionali

I requisiti non funzionali definiscono le proprietà che deve avere il prodotto realizzato.