Riassunti di Ingegneria del Software

Simone Montali

1 novembre 2019

1 T1 - Software Development Process

Anche la signora seduta all'angolo di via Cavour, sa che oggi più che mai i software sono parte di tutti i processi che fanno girare il mondo. L'Ingegneria del software è però molto di più che scrivere codice. Infatti, è piuttosto un concetto di risoluzione dei problemi del mondo reale, sfruttando software. I requirements sono sempre più stringenti: tempi brevi, sistemi complessi, molte funzionalità richieste. Un buon software deve avere ottime **maintainability, dependability, efficiency, acceptability**. Problemi e soluzioni sono complessi, ma il software offre estrema flessibilità. Esso è un sistema discreto. Alcuni problemi tipici possono essere le scadenze, i budget, le performance, la manutenzione. Le sfide principali sono rappresentate da **eterogeneità, delivery, trust.** L'attività di problem solving è composta da due fasi: l'analisi e la sintesi.

1.1 Quindi? Cos'è l'ingegneria del software?

Siamo giunti al nocciolo della questione: **di cosa stiamo parlando?** L'ingegneria del software è un insieme di **tecniche, metodologie, strumenti** che aiutano nella produzione di software di alta qualità dati un budget, una scadenza, e delle modifiche continue. La sfida principale è quindi quella di avere a che fare con complessità elevate. Siamo di fronte anche a un aumento delle responsabilità: un ingegnere del software non deve solo scrivere codice, ma piuttosto lavorare con competenze e confidenzialità, applicando un'etica.

1.1.1 Processo del software

In seguito ad una rappresentazione astratta, si procede con un set di attività strutturato: specifica dei **requirements**, **design**, **implementazione**, **validazione**,**evoluzione**.

1.2 Modelli di sviluppo del software

Distinguiamo tra **plan-driven** e **agile** development. Nel primo, prima si pianificano i requirement, e solo in seguito si sviluppa il software. Nel secondo si sviluppa il software un pezzo alla volta, a stretto contatto col cliente.

1.2.1 Modello a cascata

In questo modello, **plan-driven**, le specifiche e lo sviluppo sono separati. I pro sono, ad esempio, un'ottima documentazione e manutenzione semplice. D'altro canto, però, le specifiche vengono *"congelate"* dopo la prima fase, il cliente viene poco coinvolto, ed i tempi sono più lunghi.

1.2.2 Modello a spirale

Nel modello a spirale, abbiamo diverse fasi che si susseguono a spirale; il *risk handling* viene gestito tramite prototipazione, che permette di testare i prodotti contro i requirements. Alcuni pro possono essere l'elevata prevenzione dei rischi, la completezza della documentazione, la flessibilità. È, però, un modello costoso, riservato ad esperti ed a progetti costosi e richiedenti molta sicurezza. La spirale può allargarsi all'infinito. Il **prototipo** è un'implementazione limitata del sistema, rappresentando solo alcuni aspetti. È utilizzato in varie fasi dello sviluppo. Porta però vari vantaggi, come un'elevata usabilità, un buon design, una grande facilità di manutenzione, ed un ridotto costo di sviluppo.

1.2.3 Incremental development

L'incremental development consiste in una prima fase di raccolta dei requirement, da cui nasce la versione iniziale, una fase di design, ed una fase di implementazione, che produce la versione finale. Alcuni pro possono essere la naturale presenza di prototipi ad ogni aggiunta di feature, un basso rischio di fallimento progettuale, una quantità di testing variabile in base alla priorità. Alcuni contro: bassa *process visibility*, sistemi mal strutturati, skill speciali necessarie. Esso è adatto per progetti piccoli, o parti di progetti grandi.

1.2.4 Test driven development

Qui, i test vengono scritti prima dell'implementazione, rendendo note le difficoltà da subito. Rende il debug più semplice. Si aggiunge un test, si prova il codice vecchio con il test nuovo, si aggiunge la feature e si verifica che il test sia ancora positivo.

1.2.5 Agile development

*"Tutti fanno Agile, nessuno fa Agile."*L'agile è basato sulla **continuous delivery**, con dei requirements in continuo cambiamento. Il cliente è direttamente coinvolto, aggiungendo requirements man mano che il progetto va avanti. È una metodologia semplice nella quale il team si auto organizza, ma ha svariati rischi, come la mancanza di planning, la necessità di team esperti, la documentazione scarna o spesso errata.

1.2.6 Extreme Programming

Suona più badass di quanto sia realmente. /s L'XP è utilizzato in situazioni in cui i requirements variano velocemente, i team sono ridotti e "affiatati"(spesso si ricorre al **pair programming**). È un tipo di programmazione agile, basato su design semplice, release minori, refactoring continuo, alta semplicità.

1.3 Reusable software

È spesso comodo lavorare per **microservizi** atomici, in modo da poterli riutilizzare. Parliamo, ad esempio, di API. Questo riduce i costi e i tempi di sviluppo, al costo di leggeri sacrifici sul lato dei requirements, e un mancato controllo sull'evoluzione del software → funzionalità che variano il loro comportamento. *Parlo a lei, signor React (ah, già, Mark), che cambia tutto ad ogni versione.*

2 T2 - Coding, Debugging, Testing

Un buono stile di coding è fondamentale perché i programmi vengono scritti una volta, ma letti molte. Alcuni aspetti importanti sono il layout, i nomi, i commenti.

2.1 Legge di Ambler per gli standard

Più uno standard è utilizzato, più è facile comunicare tra membri del team. Si possono inventare standard quando necessario, ma attenzione a non perdere tempo in qualcosa che non verrà riutilizzato. Tutti i linguaggi hanno standard reperibili. *Consiglio per la vita: quando vi servono, esistono gli standard di Google, gente più affidabile di me.* È consigliabile avere standard aziendali. Uno standard consiste di nomenclature, formattazione, formato e contenuto dei commenti. È necessario documentare tutte le volte in cui si infrangono gli standard. *(È davvero necessario infrangerlo?)*

2.2 Coding practices

Alcune pratiche sono rappresentate da:

- **indentazione**, che oltre ad essere importante per la leggibilità, a volte fa parte delle regole di compilazione. *e.g. Python.*
- **whitespaces**, migliorano la leggibilità. *Però, non fate le bestie di satana mettendo uno spazio dopo l'apertura della parentesi. Non fatelo.*
- **Naming, commenting**, rendono la comprensione del codice molto più facile.

I commenti sono uno strumento fondamentale da tenere sempre vicino al codice. Non pensiate però di essere autorizzati a scrivere *spaghetti code*, se commentato. Refactorate. Fondamentale è spiegare i compiti di classi, funzioni, variabili o blocchi di codice complessi. *In generale, se avreste bisogno di spiegarlo a Guido Soncini, commentatelo.* Utilizzare uno standard permette ai vostri colleghi di non dover riscrivere il vostro codice perché non gli piace. Rendete più semplice agiungere funzioni, o creare la documentazione. È, insomma, tempo ben speso.

2.3 Dealing with errors

Distinguiamo tra prima, durante e dopo: **prevention, detection, recovery**. Alcune fonti di errori possono essere un design errato, una mancanza di isolazione, o typos. Per esempio, errori di "confini" negli array, errori di *off-by-one*, errori di input errati. Per debuggare, bisogna riconoscere l'esistenza di un bug, isolarne la fonte, identificarne la causa, trovare un fix, applicarlo, e **testarlo**. *Lo scrivo in grassetto perché mi capita spesso di romperepiù di quello che metto a posto.* Riconoscere un bug spesso è complicato, soprattutto quando accade solo in determinate situazioni, o se il software è difficile da testare. Per trovare i bug, potete usare dei print statement, molto veloci da usare ma spesso incompleti e poco pratici. Per questo, esistono gli strumenti di debug, che permettono di bloccare il codice in determinati punti, analizzare le variabili, *e bestemmiare con calma.* Spesso gli errori sono dovuti al design piuttosto che all'implementazione.

2.4 Testing

Il testing permette di scovare errori e bug, ma non la loro assenza. È purtroppo impossibile testare ogni caso. Il tester deve conoscere il sistema e le tecniche di testing. **Il tester non dovrebbe essere il programmatore.** Spesso il programmatore ha in mente il modo corretto di far funzionare il programma, e quindi difficilmente trova casi in cui il suddetto si rompe.

2.4.1 Unit testing

Lo unit testing permette di testare singole unità di codice, trovando falle negli algoritmi, i dati, la sintassi. Un set di test cases viene creato e poi utilizzato.

2.4.2 Integration testing

L'integration testing prova un gruppo di sottosistemi, o anche l'intero software. Viene eseguito dai programmatori, il goal è testare le interfacce oltre ai sottosistemi. L'intero sistema è visto come un insieme di sottosistemi, l'obiettivo è quello di testare tutte le interfacce e l'interazione tra sottosistemi. La strategia determina il modo in cui i sottosistemi vengono testati. Molte falle sono date da problemi nell'interazione tra sottosistemi. Le falle non intercettate in questa fase diventeranno molto più costose.

2.4.3 System Testing

Il system testing testa l'intero sistema, per verificare che rispetti i requirements funzionali e non, oltre alle prestazioni.

2.4.4 Functional Testing

Questo tipo di testing viene svolto per verificare la funzionalità del sistema. I test cases vengono ideati a partire dai requirement del progetto, ed il sistema è trattato come una black box.

2.4.5 Performance Testing

Questo tipo di testing tenta di provare il sistema in situazioni estreme, come alti carichi, input errati, grandi volumi di dati. Alcuni esempi sono stress testing, security testing, volume testing, recovery testing.

2.4.6 Acceptance Testing

Questo tipo di testing prova che il sistema sia effettivamente pronto per la fase di production. I test vengono scelti ed effettuati dal cliente. Questi sono i famosi **alpha e beta tests.** Nel primo, il software è ancora nell'environment di sviluppo. Nella beta, l'environment è quello del cliente e l'utilizzo effettuato è realistico.

3 T3 - System Modeling and UML

Il system modeling fornisce rappresentazioni astratte a problemi reali, tramite notazione grafica. Un modello funzionale dovrebbe introdurre i componenti essenziali, utilizzare una notazione *consistente*, ed utilizzare tool al supporto della creazione. Il modello esprime quindi la realtà, adattata a dei modelli standard. Il system modeling deve essere **predictive**, in quanto deve essere svolto prima del development. Dev'essere **extracted** da un sistema esistente, tramite analisi delle proprietà del software. Deve essere **prescriptive**, ossia definire un set di regole e limiti per l'evoluzione del software.

3.1 UML

Unified Modeling Language nasce per unire diversi standard/linguaggi di modellazione, con diagrammi multipli e interoperabilità. UML è **semplice, espressivo, utile, consistent, estensibile**. Alcuni esempi di views sono:

- Use Case view
- Structural view
- Behavioral view

- Implementation view
- Environment view

4 T4 - Requirements Engineering

Gli scopi del requirements engineering sono **identificare** i servizi necessari ed i constraint, **definire** offerta e contratto, **ottenere** tutte le informazioni necessarie al design. I desiderata sono:

- **Validi**, esprimendo le reali necessità
- **Non ambigui**, leggibili in un solo modo
- **Completi**
- **Comprensibili** da tutte le persone coinvolte
- **Consistent**, non possono contraddirsi
- **Prioritizzati**, a volte bisogna scegliere
- **Verificabili**, con test
- **Modificabili** senza difficoltà
- **Traceable**, la loro origine è chiara

4.1 Some requirements classifications

4.1.1 Functional requirements (System Feature)

Descrivono funzionalità di sistema o di servizi, come l'input di dati, operazioni svolte, workflow, dati in output, autorizzazioni. Ad esempio, in una biblioteca, un functional requirement può essere la ricerca di libri da parte di un socio.

4.1.2 Non-functional requirements (System Feature)

Descrivono constraints di parti del sistema e del suo sviluppo. Specificano criteri per giudicare l'operato del sistema. Con l'esempio di prima, i libri devono avere un codice che rispetti lo standard ISBN, e il sistema non deve rilasciare informazioni sensibili sui soci a determinati autorizzati. Alcune metriche per questi requirements possono essere velocità, dimensione, facilità d'uso, affidabilità, robustezza, portabilità.

4.1.3 Domain requirements (System Feature)

I **domain requirements** derivano dal dominio dell'applicazione, ossia l'ambito in cui si lavora. Ad esempio, sviluppando un sistema per Trenitalia *(qualcuno lo fa veramente? Non sono scritti da scimmiette?)*, il dominio sarebbe quello della gestione dei treni. Il problema sta nel fatto che il programmatore non ha conoscenze al riguardo dei treni, e deve utilizzare quindi un linguaggio, implicazioni e conoscenze che non gli appartengono.

4.1.4 Volatile Requirements (Static/Dynamic Nature)

I **mutable requirements** sono requirements destinati a cambiare, come normative o tasse. Gli **emergent requirements** cambiano quando il cliente capisce di più sul sistema. I **consequential requirements** emergono con l'informatizzazione di un sistema che non lo era. I **compatibility requirements** emergono dal doversi interfacciare con altri sistemi appartenenti all'organizzazione.

4.2 Rischi

Alcuni rischi nella scrittura dei requirements possono essere:

- Imprecisioni
- Conflitti tra più requirements

4.3 Documento di specifica dei requirements

Il **documento di specifica dei requirements** specifica i requirement di sistema, includendone una definizione e una specifica. È detto **System Specification** se include direttive su hardware e software, **Software Requirements Specification** (SRS) se include il solo software. Dovrebbe seguire lo standard IEEE 830. Un SRS deve avere un'**introduzione**, una **descrizione generale** ed infine **le feature e i requirement.** Dovrebbe avere un formato stratificato, notazioni grafiche e termini consistenti, acronimi chiari, indice, glossario, ed uno stile non ambiguo. A tal proposito, il linguaggio naturale spesso nasconde delle insidie: mancanza di chiarezza, ambiguità, troppa flessibilità... Bisogna quindi inventare uno standard di utilizzo del linguaggio naturale, con sintassi fissa, termini chiari. Alcune keyword sono: **shall, should, can, must, may, will, might, expected to, could.** Alcune alternative al linguaggio naturale possono essere un linguaggio naturale strutturato, linguaggi di descrizione del design, notazioni grafiche, notazioni formali. I rischi del processo di specifica sono una mancanza di comprensione, requirements che cambiano rapidamente, impreicisione nella stesura del documento, difficoltà nel conciliare conflitti.

5 T5 - Requirement engineering and UML

5.1 Use case diagram

Nello use case diagram includiamo tutti i casi d'uso del sistema, da parte di diversi **attori**, rappresentanti utenti, ma anche servizi o sistemi. Il **system boundary** divide l'esterno e l'interno del sistema, gli use case dagli attori. Un attore può generalizzare un altro attore. Idem per gli use case. Uno use case può contenere la funzionalità di un altro use case. Uno use case può essere usato per estendere il comportamento di un altro use case, anche con condizioni. Opzionalmente, possiamo includere le molteplicità delle relazioni.

5.2 Class diagram

Una classe è rappresentata da un rettangolo che mostra il nome della classe, e opzionalmente il nome degli attributi e delle operazioni. Il nome della classe, gli attributi e le operazioni sono separati in compartimenti. Il simbolo che precede attributi e operazioni ne indica la visibilità: + se pubblico, − se privato, # se protetto, se package. Un'**interfaccia** è una specifica di comportamento che deve essere implementato o, più semplicemente, un **contract.** Un **template** definisce un pattern i cui parametri rappresentano tipi, e può essere applicato a classi, packages, operazioni.

5.2.1 Associazione

Un'associazione è una relazione tra elementi, che implica che uno dei due sia una variabile dell'altro. Essa è rappresentata da un connettore che può includere ruoli, cardinalità, direzione e constraints. Per più elementi, si può usare un **diamond**.

5.2.2 Generalization e nesting

La generalizzazione indica **ereditarietà**, disegnata come una freccia che parte dal figlio e arriva al padre. Il connettore di nesting indica che una classe è nested nella classe dove arriva l'operatore. Con ciò, intendiamo che essa è definita all'interno del target.

5.2.3 Dipendenza e realizzazione

La **dipendenza** è una forma debole di relazione, e mostra un'interazione tra un client ed un supplier. La **realizzazione** è una relazione tra una specifica e la sua implementazione.

5.2.4 Aggregazione e composizione

L'aggregazione rappresenta elementi composti da elementi minori. È indicata da un diamante bianco che punta verso il contenitore. I componenti possono essere condivisi da più contenitori. La composizione è rappresentata da un diamante nero. Una classe di associazione rappresenta invece un'associazione più complessa, che ha operazioni e attributi.

5.3 Sequence diagram

Il sequence diagram indica, su una *lifeline (aka timeline)* verticale, l'interazione tra le classi. Se il nome della lifeline è *self*, la suddetta rappresenta il classifier a cui appartiene il diagramma. Una **lifeline** rappresenta un *partecipante* del sequence diagram. Possiamo avere diversi tipi di messaggi tra lifeline: sincroni, asincroni, risposte, persi, trovati, self (ricorsività). Per rappresentare logica procedurale come if, cicli, thread, usiamo i **fragment**.

5.4 Activity diagram

Un **activity diagram** è un diagramma di flusso che rappresenta un'operazione eseguita sul sistema, con decisioni, I/O, fork/join, timeout/segnali, concorrenza. Possiamo raggruppare alcune attività *related*. Alcuni esempi di activity diagrams strani:

- **State machine diagram**, che dà indicazioni a livello hardware
- **Choice and Junction pseudo state**
- **Compound state** tramite il quale possiamo separare alcune parti del diagramma e "includerle"
- **History state and concurrent regions**, con il primo indichiamo un salvataggio dello stato, col secondo due frazioni di diagramma che si svolgono contemporaneamente

5.5 Robustness diagram

Il **robustness diagram** è un UML semplificato che ha lo scopo di raffinare gli use case, verificandone correttezza, completezza e requisiti. Presenta tre **object nodes:**

- **Boundary**, che permette la comunicazione tra attori e sistema
- **Control**, intermediario tra boundary ed entity, implementa la logica che gestisce i vari elementi e le loro interazioni
- **Entity**, rappresenta un'unità informativa del sistema

6 T6 - Feasability and requirements elicitation

Per ottenere le informazioni necessarie, dobbiamo identificare le fonti, acquisire le informazioni, analizzarle e verificarle. Infine, vanno sintetizzate. I vari stakeholder *aka le persone interessate* possono essere così categorizzati con i loro interessi;

- **Stakeholder**, interessato nel progetto
- **Developer**, alta produttività, mancanza di errori, minore sforzo possibile
- **Marketing sales**, soddisfazione del cliente e vendite
- **Project management**, budget, scadenze
- **Investor**, velocizzazione del processo
- **Customer and user**, usabilità e workflow

6.1 Tecniche di elicitation

(elicitation = tirare fuori le informazioni)

6.1.1 Document analysis

Bisogna preparare i documenti che possono essere adatti e rilevanti, studiarli, annotare informazioni ed elencare le domande al riguardo. Infine, assieme agli stakeholder, verificare le note, organizzare i requirement e rispondere alle suddette domande. Questa tecnica può essere sfruttata quando non c'è presenza degli stakeholder, o per il cross-checking.

6.1.2 Observation of the work environment

L'obiettivo è determinare **chi, cosa, dove, quando, perché e come.** Bisogna ottenere il permesso dai supervisori, informare gli osservati, prendere appunti, evitare di disturbare ma al tempo stesso non dare nulla per scontato. I dati ottenuti sono molto affidabili.

6.1.3 Questionario

Serve determinare i fatti e le opinioni necessarie, in secundis, da chi reperirli. Determinare quindi le domande, aperte o chiuse, da fare. Può essere conveniente "testare" il questionario su un gruppo piccolo, per poi metterlo a posto e presentarlo a tutti. Questa tecnica restituisce molti dati, è facile da attuare ed affidabile. Non c'è però interazione, e si rischiano risposte incomplete.

6.1.4 Interviste

Bisogna, prima di tutto, decidere chi intervistare. In seguito, preparare le domande e porle all'intervistato, con un linguaggio chiaro. Conviene prendere appunti e memo. Questo è un metodo ottimo perché permette di verificare i fatti, coinvolgendo gli end user. Non è però adatto a comprendere i domain requirements.

6.1.5 Scenarios and use cases

Essi sono esempi IRL di come il sistema verrà usato, basati su situazioni reali su cui gli stakeholder hanno senz'altro qualcosa da dire. Gli **scenari** sono semplicemente forme strutturare di user stories. Per gli use cases usiamo UML. Queste tecniche sono semplici ed utili, anche per sistemi complessi. È però difficile capire quando fermarsi, e soprattutto non sono utili a capire i **non-functional requirements.**

6.2 Attività di supporto all'elicitation

6.2.1 Brainstorming

Composto di due fasi, la prima di **storm** in cui si generano le idee, la seconda **calm** in cui vengono filtrate. Necessari due ruoli chiave: uno **scribe** ed un **moderatore**. Nel filtraggio delle idee è fondamentale: unire idee simili, applicare i criteri di accettabilità, votare con una soglia o votare con dei *campaign speeches.*

6.2.2 Focus group

Concetto simile al brainstorming ma più strutturato, esplora pro e contro di determinate opzioni. Fondamentale un moderatore.

6.2.3 Prototypes

Si mostra l'esecuzione di una task, si identificano le alternative, si cerca di estrapolare i possibili problemi.

7 Use Cases

Gli **use case** sono scenari che sfruttano diagrammi UML, descrivono le task del sistema e l'interazione con gli attori. Non sono mappati one-to-one coi requirements, ma ogni requirement deve essere coperto da almeno uno use case. Gli use case sono quindi composti da use case diagrams, descrizioni testuali, ed interaction diagrams (opzionali). Per identificare gli actor bisogna definire prima i boundary, poi gli utenti, l'hardware, i ruoli. Per identificare gli use case bisogna prima identificare il dominio. Vanno poi annotati come verbi rappresentanti le azioni. Per l'identificazione degli scenari, bisogna comprendere la situazione iniziale e il flusso di eventi. È utile capire cosa può andare male, e tutti i flussi alternativi. Infine, consideriamo la situazione finale. Uno use case deve avere un singolo attore iniziante, pochi step, non deve includere scelte implementative, può includere UML. Bisogna poi dare una priorità ai vari use case, in base all'impatto, la difficoltà, la necessità.

7.1 Componenti principali

Uno use case deve definire lo stato iniziale e le precondizioni. Deve definire l'ordine degli eventi, le alternative, le situazioni eccezionali, e i risultati. Deve inoltre menzionare gli attori coinvolti, i diagrammi related, i problemi di design. Alcune guidelines:

- Non pensare al lato implementativo

- Essere narrativi
- Elencare gli scenari funzionanti
- Elencare tutti i possibili use case
- Utilizzare un formato standard
- Utilizzare i verbi appropriati
- Documentare le eccezionali
- Non rappresentare singoli step come use cases

Note to self: metti un template qui