Progettazione e Validazione di Sistemi Software

Alessio Gjergji

Indice

Capitolo 1

Agile

1.1 Agile

1.1.1 Approccio Agile

Costruire sistemi safety-critical, governativi o gestire molti team distribuiti sul territorio richiede una coordinazione efficace. Verso la fine degli anni '90, il contesto di sviluppo software è cambiato radicalmente. I piani rigidi non erano più sempre efficaci, specialmente considerando le richieste di consegne sempre più rapide. Con team piccoli, l'overhead generato da processi pesanti sarebbe stato un costo eccessivo, portando così all'adozione di un approccio Agile.

Il requisito fondamentale dell'approccio Agile è rispondere rapidamente ai cambiamenti anziché seguire pedissequamente un piano preciso. Data l'incertezza sulle esigenze finali e la loro probabile evoluzione nel corso dello sviluppo, l'Agile propone una strategia per gestire tali cambiamenti. In questo contesto, si evita di congelare i requisiti iniziali, ma si cerca piuttosto di capire meglio cosa si vuole ottenere.

Nell'approccio Agile, la specifica, il design e l'implementazione procedono in parallelo e sono strettamente collegati. Il software finale viene gestito attraverso una serie di versioni, con ciascuna versione successiva che rappresenta un incremento del prodotto. È cruciale coinvolgere il cliente e gli stakeholder durante tutto il processo di sviluppo Agile in modo da assicurare un feedback costante e un allineamento continuo con le aspettative.

Di solito, si stabilisce un intervallo di tempo limitato per lo sviluppo di ciascuna versione, che va da 2 a 4 settimane. La documentazione è mantenuta al minimo per evitare discrepanze e inconsistenze tra la documentazione e il codice stesso. La comunicazione all'interno del team è spesso informale ma frequente, promuovendo un flusso costante di informazioni.

Per automatizzare il processo di testing, è fondamentale adottare strumenti per il testing automatico, in quanto il testing manuale risulterebbe troppo lento e inefficiente data la necessità di coprire numerosi scenari di test. Alcuni strumenti chiave per l'approccio Agile includono:

- Testing automatico

- Gestione della configurazione
- Integrazione continua
- Produzione automatica dell'interfaccia utente

L'obiettivo è rilevare e correggere gli errori il prima possibile, in modo da evitare che si propaghino attraverso il sistema.

1.1.2 Manifesto Agile

L'Agile si basa su alcuni principi chiave che sono riassunti nel Manifesto Agile:

- L'importanza è spostata dal processo formale all'individuo e alle interazioni.
- È più importante avere un software funzionante rispetto a una documentazione esaustiva.
- Si promuove la collaborazione con il cliente, che dovrebbe essere parte integrante del team di sviluppo piuttosto che un soggetto esterno con cui contrattare.
- L'attenzione è posta sulla capacità di rispondere al cambiamento in modo rapido e flessibile, piuttosto che sull'aderenza rigorosa a un piano prestabilito.

1.1.3 Implementazione dell'Approccio Agile

Per realizzare appieno i principi dell'Agile, è fondamentale coinvolgere attivamente il cliente nel processo di sviluppo. Il coinvolgimento del cliente deve andare oltre il semplice fornitore di specifiche, incoraggiando il cliente a partecipare attivamente alle discussioni e al processo decisionale. Questo assicura che le funzionalità sviluppate siano allineate con le aspettative del cliente e che i feedback siano integrati tempestivamente nel processo di sviluppo.

Un altro aspetto chiave è lo sviluppo incrementale, dove le funzionalità vengono fornite in maniera graduale e integrata nel sistema in crescita. In questo contesto, è fondamentale avere team con competenze solide e diversificate, che si fidino a vicenda per garantire un flusso di lavoro efficiente e un'efficace condivisione delle responsabilità.

Nell'approccio Agile, è importante prepararsi al cambiamento continuo dei requisiti nel corso del progetto. La capacità di adattarsi rapidamente ai nuovi requisiti e di integrarli nel processo di sviluppo è un fattore cruciale per il successo.

Il principio cardine per lo sviluppo Agile è quello di mantenere la semplicità. Ciò implica resistere alla tentazione di aggiungere funzionalità non essenziali che potrebbero introdurre una complessità eccessiva nel sistema. L'obiettivo è quello di adottare strategie che semplifichino il processo e riducano al minimo il rischio di complicazioni impreviste durante lo sviluppo.

1.1.4 Applicabilità dell'Approccio Agile

L'approccio Agile è particolarmente adatto in contesti in cui il cliente è disponibile e desideroso di partecipare attivamente al processo di sviluppo del software. È efficace in ambienti

in cui le normative sono meno restrittive e permettono una maggiore flessibilità nel processo di sviluppo. I processi di sviluppo Agile sono spesso implementati in combinazione con metodologie di gestione progetti agili, come ad esempio Scrum.

1.2 Extreme Programming (XP)

Spesso, nell'ambito dell'approccio Agile, si fa riferimento a **Extreme Programming**, che spinge all'estremo alcune delle caratteristiche fondamentali dell'Agile. Ad esempio, in Extreme Programming, sono comuni varie versioni del software consegnate al cliente quotidianamente, con incrementi che vengono immediatamente messi in produzione. Un principio chiave è che i casi di test devono superare con successo per ogni build, pertanto è necessario assicurarsi che tutti i casi di test siano funzionanti prima di procedere con una nuova build.

Nel contesto di Extreme Programming, viene dato ampio risalto alla pratica di scrivere i test prima di scrivere il codice effettivo. Il refactoring è una pratica costante con l'obiettivo di semplificare il codice sorgente per renderlo più leggibile e mantenibile nel tempo. Altro aspetto fondamentale di XP include il pair programming, in cui due sviluppatori lavorano insieme su un singolo codice, garantendo una maggiore qualità e condivisione delle conoscenze. Inoltre, si mira a mantenere un ritmo di sviluppo sostenibile nel tempo, evitando eccessive accelerazioni che potrebbero compromettere la qualità del software.

1.3 User Stories

Nel contesto dell'Agile, i requisiti vengono spesso raccolti in maniera informale tramite user stories, che sono narrazioni in prosa che descrivono le interazioni tra l'utente e il sistema. Questo metodo aiuta a mantenere i requisiti concreti e facilita l'elicitazione dei requisiti stessi poiché le user stories sono facili da scrivere e da comprendere.

Le user stories vengono successivamente suddivise in parti più piccole, comunemente chiamate task cards. Per ogni task, è necessario fornire una stima temporale per la sua realizzazione. L'obiettivo è quello di coinvolgere attivamente il cliente nella priorizzazione delle varie attività, poiché, in un contesto di sviluppo incrementale, non è possibile affrontare tutto contemporaneamente.

Tuttavia, con questo tipo di approccio, non si ha la certezza di implementare tutte le funzionalità richieste, poiché alcune di esse potrebbero non emergere in determinati scenari o iterazioni di sviluppo.

1.4 Test-Driven Development (TDD)

Nel Test-Driven Development (TDD), i test vengono scritti prima del codice stesso, e tali test possono essere eseguiti durante la fase di scrittura del codice. Questo approccio consente di individuare errori il prima possibile, valutando ogni singola microcomponente prima di passare a quella successiva. Il codice risultante è di alta qualità in quanto viene costantemente testato e non presenta bug evidenti.

I test, in questo contesto, rappresentano una documentazione degli scenari previsti e consentono di pensare al comportamento del sistema prima ancora di implementarlo effettivamente. Ciò porta a una maggiore consapevolezza del sistema e del suo funzionamento da parte del team di sviluppo.

Inoltre, è importante coinvolgere attivamente l'utente nella fase di verifica, in modo da sviluppare i cosiddetti acceptance test per le varie user stories. Questo è possibile solo grazie all'utilizzo di framework di test automatizzati. Di conseguenza, il numero di test generati è notevolmente elevato, garantendo la coerenza e la stabilità del sistema, specialmente in un contesto di sviluppo incrementale.

1.4.1 Refactoring

Il refactoring rappresenta il processo di riscrittura del codice al fine di renderlo più leggibile e mantenibile nel lungo periodo. Spesso, ogni singolo incremento potrebbe richiedere alcuni compromessi che, per essere integrati nel sistema, richiedono una sorta di "degradazione" del sistema. Il refactoring permette di affrontare tali problematiche e di mantenere l'integrità del sistema nel tempo, assicurandone la stabilità e la longevità. Il refactoring risulta essere una pratica altrettanto valida anche nel contesto di sviluppo pianificato (*plan-driven*).

1.4.2 Pair Programming

Il pair programming è una pratica in cui due persone lavorano insieme su un singolo computer. Mentre una persona scrive il codice, l'altra controlla attivamente il lavoro. I ruoli si alternano regolarmente, e le decisioni cruciali vengono prese in modo collaborativo. I vantaggi del pair programming sono numerosi, inclusa la condivisione della responsabilità, il controllo continuo e un feedback costante. Inoltre, il continuo scambio di conoscenze e la code review continua portano a un miglioramento delle abilità individuali e della qualità del codice. Il refactoring continuo contribuisce ulteriormente a migliorare la comunicazione all'interno del team. Contrariamente a ciò che si potrebbe pensare, il pair programming non causa una diminuzione significativa della produttività, ma piuttosto conduce a un miglioramento generale delle prestazioni del team.

1.5 Project Management Agile

Nel contesto dell'approccio Agile, il project manager ha la responsabilità di assicurarsi che il software venga consegnato entro i tempi previsti, con tutte le funzionalità richieste e nel rispetto del budget stabilito. È responsabile del coordinamento del progetto e del monitoraggio dei progressi.

1.5.1 Scrum

Scrum è un metodo di project management agile che si basa sul lavoro per iterazioni, richiamando così il processo di sviluppo software agile. Le sue fasi principali includono:

1. Definizione di obiettivi generali e architettura ad alto livello del sistema.

2. Sprint planning, in cui si definiscono gli obiettivi per la prossima iterazione.
3. Fase conclusiva, che include la documentazione finale, la code review e la riflessione sulle lezioni apprese.

1.5.2 Terminologia

Alcuni termini chiave in Scrum includono:

- Team piccolo, con un massimo di 7 persone.
- Backlog, che rappresenta la lista di tutte le funzionalità richieste dal cliente.
- Product Owner, responsabile della priorizzazione delle funzionalità e delle decisioni su cosa è più necessario.
- Sprint, che rappresenta un periodo di lavoro di 2-4 settimane.
- Scrum Master, responsabile del monitoraggio e del supporto al processo di sviluppo.
- Velocità, che rappresenta la produttività del team e viene calcolata e aggiornata durante lo sviluppo per la pianificazione dei prossimi sprint.
- Sprint Review, una sessione per riflettere sull'iterazione e fornire input per migliorare i prossimi sprint.

1.5.3 Benefici

I benefici di Scrum includono la gestione efficiente di funzionalità nel breve termine, una conoscenza approfondita dei problemi e delle attività di ogni membro del team, nonché la puntualità delle consegne per il cliente. Scrum favorisce un clima di fiducia tra il cliente e il team di sviluppo, poiché il cliente ha la possibilità di visualizzare il sistema software funzionante e fornire un feedback continuo. Questo feedback aiuta a chiarire le esigenze del cliente e ad adattare il prodotto in modo efficace.

1.6 Scrum Distribuito

Lo Scrum distribuito si riferisce a situazioni in cui il team di sviluppo è distribuito in più sedi geografiche. In questo contesto, la comunicazione tra i membri del team avviene attraverso strumenti come chat in tempo reale, chiamate video e l'utilizzo di sistemi di continuous integration per garantire che il software sia sempre funzionante. Un piano di sviluppo comune aiuta a facilitare la comunicazione tra i vari team e sedi geografiche.

1.6.1 Scalabilità

La scalabilità di Scrum si riferisce alla sua capacità di adattarsi a progetti più grandi che coinvolgono più persone e team, o che richiedono un'implementazione distribuita. Alcuni punti chiave da considerare includono la necessità di mantenere le pratiche chiave come il Test Driven Development (TDD), la comunicazione e la consegna di incrementi funzionanti.

La coordinazione tra lo sviluppo Agile e la manutenzione del sistema è altrettanto importante, specialmente quando le persone coinvolte potrebbero cambiare nel corso del progetto.

1.6.2 Problemi e Resistenze

Alcuni problemi e resistenze comuni nell'adozione di metodi Agile come Scrum includono la mancanza di esperienza del management nell'ambito della metodologia Agile, la necessità di adattarsi a procedure di controllo della qualità preesistenti e le resistenze culturali all'interno dell'organizzazione. Il successo della metodologia Agile può essere dimostrato attraverso casi di studio e una maggiore stabilità dei requisiti nel tempo.

Capitolo 2

Ingegneria dei requisiti

2.1 Requisiti

Il requisito è la descrizione delle funzionalità di un sistema software, nella condizione in cui questo sistema dovrebbe funzionare. Riflette le necessità del cliente e le sue aspettative su software. Sembra semplice, ma è critico.

2.1.1 User requirements

Parliamo dei user requirements, ovvero i requisiti utente. Questi sono requisiti ad alto livello, date dall'utente che esprimono il comportamento del sistema, sono tipicamente frase astratte e generiche, non prevedono una soluzione. Tipicamente scritti con il cliente.

2.1.2 System requirements

System requirements, ovvero i requisiti di sistema, sono requisiti più specifici, strutturati e formali. Tipicamente si va nel dettaglio, comprensibile anche dall'utente. I requisiti di sistema possono essere utili per l'utente per essere validati.

2.2 Stakeholders

Tutte le persone coinvolte nel progetto, che hanno un interesse nel progetto. Quindi utenti e tutti quelli che verranno toccati dal sistema software.

2.3 Requisiti funzionali e non funzionali

2.3.1 Requisiti funzionali

Quando parliamo di requisiti funzionali intendiamo requisiti che catturano funzionalità che il sistema deve fornire. Raccoglie scenari, come deve agire, come agisce, cosa fa, cosa non fa.

Ci sono diversi livelli di astrazione, alcuni descrivono l'alto livello.

Possono essere ambigui. Bisogna segnalare l'ambiguità per evitare errori.

Devo raggiungere l'obiettivo della completezza, ovvero che tutte le funzionalità devono essere catturate dai miei requisiti. Il secondo obiettivo è la consistenza, ovvero che non devono essere alterati.

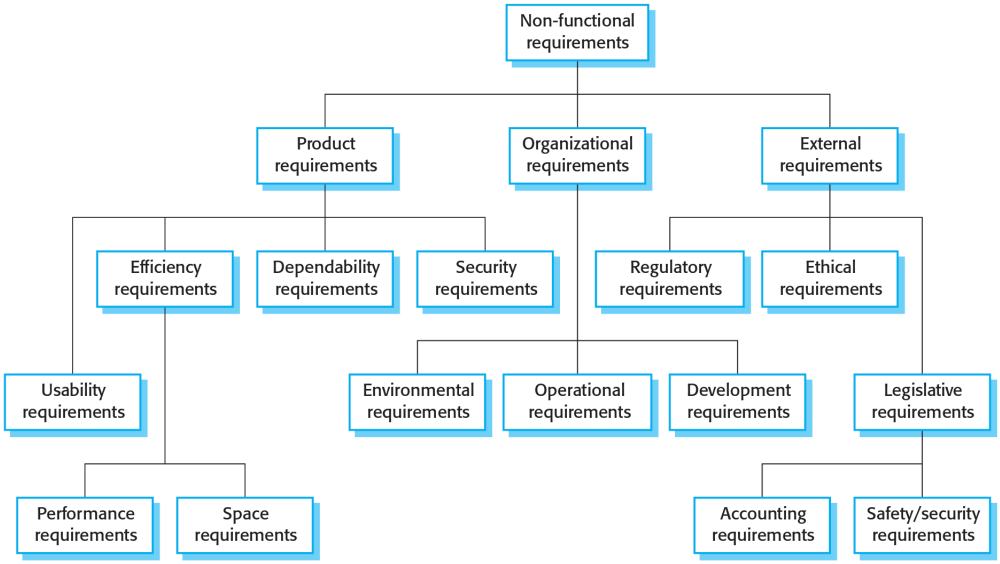
In pratica questi due obiettivi nella realtà non vengono mai raggiunti al 100%, quindi potrebbero esserci incomprensioni tra stakeholder. Alcuni problemi emergono in fase di sviluppo.

2.3.2 Requisiti non funzionali

Descrivono caratteristiche che non sono funzionalità, ma sono importanti per la soddisfazione del cliente. Coinvolgono tutti i componenti. Ambienti di sviluppo, standard da soddisfare, performance, requisiti di qualità, requisiti di interfaccia.

Richieste che vincolano l'architettura del sistema. Un requisito non funzionale potrebbe generare più requisiti funzionali. Non emergono dalla discussione.

Ad esempio i **requisiti di prodotto**, specificano comportamento del sistema e si specializza in diversi tipi di requisiti.



È opportuno segnalare più requisiti non funzionali, per evitare che vengano persi, quindi da qualitativo deve diventare quantitativo. Riduce la contestazione di un sistema, riducendo il grado di contestabilità, per togliere astrazione.

Le metriche per trasformarli in requisiti quantitativi sono:

- Velocità;
- Dimensione;

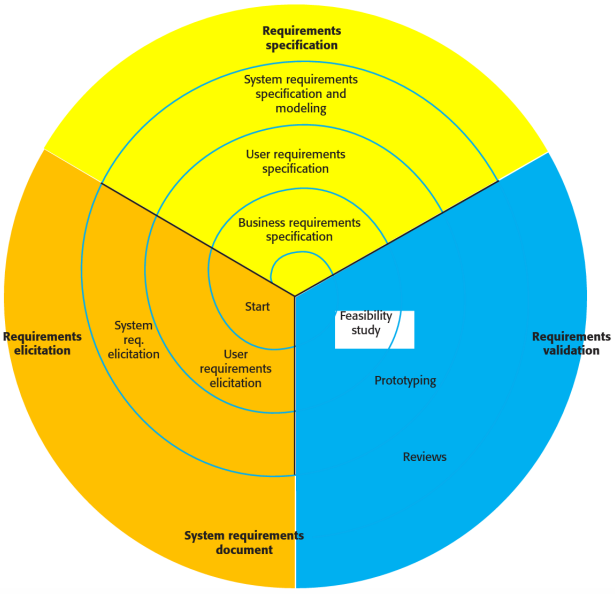
- Facilità di utilizzo;
- Reliability;
- Robustezza;
- Portabilità;

2.4 Processi di ingegneria dei requisiti

Ci sono tre fasi:

- Raccolta: ho la descrizione del sistema;
- Specifica: ho il documento di specifica dei requisiti;
- Validazione: ho il documento di validazione dei requisiti;

Ognuna di queste fasi ha un output. In generale la raccolta può essere visto come un processo a spirale.



2.4.1 Elicitazione dei requisiti

Devono capire il dominio applicativo, funzionalità che dovrà fornire, vincoli non funzionali come le performance, hardware con cui dovrà interagire. Ci potrebbero essere ostacoli, sistemi non realizzabilit, i clienti possono esporre i requisiti con termini non noti agli ingegneri del software o sottointendere requisiti per sono ovvi nel loro dominio applicativo.

Con differenti stakeholder, si possono avere differenti obiettivi, quindi potremmo raccogliere requisiti in conflitto tra loro. Potrebbero non collaborare se qualcuno non si sente rappre-

sentato. Potrebbero esserci fattori politici. Potrebbero cambiare nel corso dello sviluppo, o emergere. Le esigenze intrinseche del cliente potrebbero.

Il primo modo per raccogliere i requisiti è mediante intervista, con risposte chiuse o domande aperte. In genere mista. Gli stakeholders non forniscono requisiti dettagliati. Non è semplice condurre un'intervista in maniera efficace. Cercare di evitare pregiudizzio, prima di portarmi avanti con assunzioni verifico. Iniziare con domande trampolino, non iniziare con cose molto astratte.

2.4.2 Etnografia

Non intervistare le persone, ma lavorare insieme a loro, per capire il loro necessità. Questo aiuta molto la conoscenza implicita che potrebbe non emergere durante l'intervista. Questo tipo di studio può essere attuato un sistema software, o un processo in atto.

2.4.3 Storie e scenari

Sono dei modi concreti per descrivere un esempio reale del sistema software in un particolare contesto. Possono essere presentato allo stakeholder che può esprimere parere e dire se qualcosa va bene o no. Le storie sono testi narrativi, ad alto livello, che descrivono un'interazione e sono facili da comprendere. Lo scenario è un esempio di utilizzo del sistema, che descrive un'interazione tra utente e sistema. È più dettagliato e preciso (ho un input specifico).

Una storia raccoglie può quindi raccogliere più scenari.

2.4.4 Documento dei requisiti

È un documento che contiene tutti i requisiti del sistema software. i requisiti dell'utente devono essere raccolti dall'utente, i requisiti di sistema vengono catturati dagli ingegneri del software.

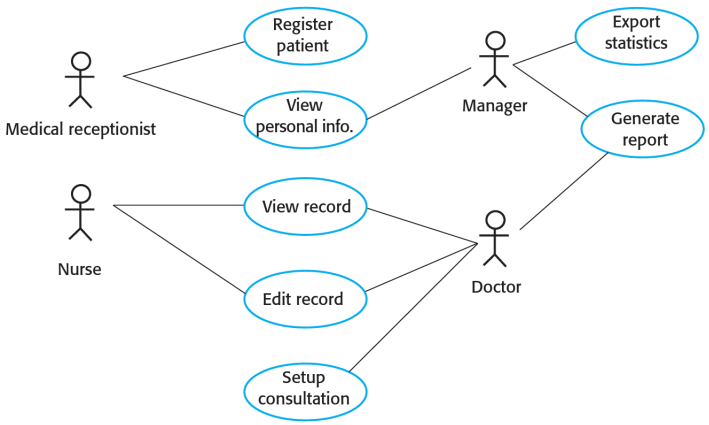
Scrivendo i requisiti emergono anche architetture o magari dalla descrizione capiamo che deve operare con altri sistemi. Oppure vincoli da alcuni documenti.

Il primo modo per raccogliere i requisiti è il linguaggio naturale, che però è intrinsecamente ambiguo. Quindi adoperare un linguaggio formale, utilizzando i verbi modali in modo corretto. Testi in grassetto per evidenziare parti chiave, mediante artefatto topografico. Tipicamente si inserisce il motivo per cui un requisito è necessario. Tipicamente si aggiungono anche identificativi univoci.

Un'altra tecnica è una forma più vincolata. Spesso su sistemi verificati da terze parti.

2.4.5 Use cases

L'use case è una descrizione di un insieme di sequenze di azioni che un sistema e serve per identificare gli attori e le funzionalità del sistema che fanno riferimento ai vari utenti.



2.4.6 Documento dei requisiti

Per specificare i requisiti parliamo di documento dei requisiti software. Può esserci variabilità su questo docomento, ci sono organizzazioni internazionali che possono definire standard (esempio IEEE).

IEEE stuttura

- prefazione
- Introduzione
- Glossario: la terminologia è un insieme di keyboard;
- Definizione dei requisiti utente;
- Vicoli per l'architettura;
- Requisiti di sistema;
- Modelli di sistema
- Evoluzione del sistema;
- Appendice;
- Indice;

Gli utenti interessati sono:

- Utenti: chi usa il sistema;
- Manager: possono stimare costi e rischi;
- Ingegneri del sistema: possono capire cosa devono fare;
- Ingegneri del software legati al test: per i test in determinati scenari.
- Ingegneri legati alla manutenzione: possono non essere legati al progetto originale;

2.5 Cambiamento dei requisiti

Possono evolvere durante lo sviluppo, bisogna dotarsi di un processo. Bisogna analizzare il costo che il cambiamento comporta. Tracciare i cambiamenti per poter propagare la modifica in tutto il sistema.

Il documento dei requisiti va modificato e poi il software. Quindi devono andare di pari passo.

2.6 Progetto

CTO dell'azienda, l'azienda ha l'incarico di questo software, abbiamo risorse limitate. soddisfi le esigenze dei committenti.

Documentazione con gli scenari. Sotto forma di lista puntata non va bene, intendiamo cose specifiche.

Capitolo 3

Refactoring

3.1 Introduzione

Refactoring

Il codice sorgente, soprattutto con approcci incrementali, tende a subire cambiamento nel tempo. Questi cambiamenti possono degradare la struttura interna del software, rendendolo difficile da comprendere e mantenere. Il refactoring è un processo di trasformazione del codice sorgente che non altera il comportamento esterno del software ma migliora la sua struttura interna.

Si tratta di una manutenzione preventiva, la qualità del codice è normale che degradi. Limitando la normale decadenza del codice, migliorare la leggibilità (*non muore una volta scritto, bisogna continuare a modificarlo*), non migliorando è più facile inserire bug.

Il refactoring è diverso dal reingegnering. Il reingegnering è un processo che mira a riprogettare il software che è stato scritto per architetture datate o tecnologie obsolete (*soprattutto cambio di architetture, come per esempio da monolitiche e cloud*). Il refactoring invece è un processo che mira a migliorare la struttura interna e va in parallelo con lo sviluppo del software.

Debito tecnico

I debiti tecnici sono tutti i problemi di comprensione del codice che sono disposto ad avere per qualcosa che serve attualmente, ma probabilmente causerà in futuro.

Per ridurre il debito tecnico si può fare refactoring. Ci sono solamente in cui è opportuno fare refactoring, ad esempio quando si deve iniziare a sviluppare nuove funzionalità, in modo che sia facile nel codice recepire questa nuova funzionalità. Oppure quando si deve correggere un bug, o in caso di identificazione di *code smell*.

3.1.1 Code smell

Sensazione che qualcosa nel codice non va, e che gli sviluppatori esperti sanno che è un problema. Gli sviluppatori esperti hanno una sensazione di disagio quando vedono un code smell, lo sviluppatore non avrebbe fatto così.

Code smell comuni

Duplicazione di codice, quando avviene si utilizza una funzione per rimuovere la duplicazione. Preventivamente vengono evitati problemi, riducendo il costo di manutenzione futuro.

I metodi e le funzioni troppo lunghi, sono difficili da capire, e ci si accorge quando si tende a commentare il codice ad ogni step. In questo caso è utile spezzare il codice in più segmenti, già individuati dai commenti.

Spesso lo switch case rappresenta un code smell, soprattutto quando si utilizza in un linguaggi ad oggetti. In questo caso è meglio sfruttare il polimorfismo. In termine di oggetti è più naturale.

Data clumping quando delle strutture dati sono sempre usate insieme, è un code smell. Ha senso creare una classe che racchiude queste strutture dati.

Generalità speculativa, quando si crea un'interfaccia per qualcosa che non è ancora necessario, ma questo crea difficoltà nella complessità del codice.

3.1.2 Classi di smell

- Troppo codice:
  - **Large class**: una classe che ha troppe responsabilità
  - **Long method**: un metodo che è troppo lungo.
  - **Long parameter list**: una lista di parametri troppo lunga.
  - **Duplicated code**: codice duplicato.
  - **Dead code**: codice che non viene mai eseguito.
- Troppo poco codice:
  - **Classi con poco codice**: una classe che non ha abbastanza codice.
  - **Empty catch clause**: un blocco catch che non fa nulla.
  - **Classi di dati**: classi con solo getter e setter.
- Codice con troppi commenti: il codice non è autoesplicativo.

3.2 Software clone

Clone

Un clone è un frammento di codice che è simile ad un altro frammento di codice.

Spesso in copia incolla e vengono fatte modifiche, ma non sempre. Tuttavia abbiamo problemi di manutenzione, perché se si modifica un frammento di codice, bisogna ricordarsi di modificare anche il clone. Attorno al 5%. Esistono tool per identificare i cloni.

Tipi di cloni

- **Type 1**: cloni identici, copia incolla.
- **Type 2**: cloni simili, ma con modifiche, a meno di ridenominazioni di identificatori.
- **Type 3**: cloni simili, ma con modifiche e aggiunte, ad esempio quando i parametri sono di tipi diversi e anche il valore di ritorno.

3.3 Refactoring

Il fatto di avere i test automatizzati è importante, perché si può fare refactoring in modo sicuro. Test di non regressione, per verificare che il comportamento non cambi. Quando tutti i test passano, si può fare refactoring, si identifica il code smell e si determina il refactoring appropriato. Una volta completata questa modifica si rieseguono i test, se passano si può continuare.

Il ritmo da seguire è quello su scala piccola, si fa refactoring su una piccola parte di codice, si eseguono i test, si fa refactoring su un'altra piccola parte di codice, si eseguono i test, e così via. In questo modo si riduce il rischio di introdurre bug e ci si accorge subito se si è introdotto un bug.

3.3.1 Applicazione del refactoring

La ridenominazione è il refactoring più semplice, si cambia il nome di una variabile, di una funzione, di una classe, di un metodo. Per farlo possiamo utilizzare tool automatici, che ci permettono di fare refactoring in modo sicuro, e vengono gestite in automatico dipendenze.

I tool riescono anche a rilevare dipendenze anche in altri file, che possono sfuggire all'occhio umano. Lo svantaggio è che automatizzano solamente i refactoring più semplici, quelli più complessi vanno fatti manualmente. Ovviamente i casi di test devono esserci e devono coprire la funzionalità.

Di solito capita di dare nomi sbagliati alle variabili, soprattutto quando non è chiaro il dominio del problema. In questo caso è utile fare refactoring per rendere più chiaro il codice. Poiché le nuove funzionalità tenderanno a seguire la vita del progetto.

Estrazione di interfaccia

Una classe dipendente non dalla struttura, ma gli si da un'interfaccia che deve essere implementata. In questo modo si può cambiare la struttura della classe, ma la classe dipendente non cambia, riducendo il numero di cambiamenti da fare sulla classe. Si ragiona meglio in termini di funzionalità e non di implementazione in termini di cambiamento.

Pull up method

Si spostano dei metodi da una classe a una superclasse, in modo che le sottoclassi ereditino il metodo. In questo modo si evita di avere codice duplicato.

Extract method

Si estrae un metodo da una classe, in modo da avere un metodo che fa una sola cosa. In questo modo si evita di avere metodi troppo lunghi.

Move method

Si sposta un metodo da una classe a un'altra, in modo che il metodo sia più vicino ai dati che utilizza. In questo modo si evita di avere metodi che utilizzano dati di altre classi. Come ad esempio:

La persona partecipa al progetto	`p.partecipate(x)`
Il progetto contiene la persona	`x.partecipate(p)`

In questo modo il progetto viene controllato la classe progetto, poiché si tratta di una proprietà del progetto.

Replace temp

Si sostituisce una variabile temporanea con un metodo, in modo da rendere più chiaro il codice.

```
double basePrice = quantity * itemPrice;
if(basePrice > 1000)
    return basePrice * 0.95;
else
    return basePrice * 0.98;
```

Si crea il metodo `basePrice()` che restituisce il prezzo base, in questo modo il codice diventa più chiaro.

```
if(basePrice() > 1000)
    return basePrice() * 0.95;
else
    return basePrice() * 0.98;
```

Replace method with method object

Si sostituisce un metodo con un oggetto, in modo da avere un metodo che fa una sola cosa. In questo modo si evita di avere metodi troppo lunghi.

```
int basePrice = quantity * itemPrice;
double discountLvel = getDiscountLevel();
double finalPrice = computeFinalPrice(basePrice,
   discountLevel);
```

Si elimina il parametro discountLevel, chiamando il metodo al suo interno per ridurre la complessità del metodo.

3.3.2 Extract class

Si estrae una classe da un'altra, in modo da avere una classe che fa una sola cosa. In questo modo si evita di avere classi troppo grandi.

3.3.3 Replace inheritance with delegation

Si sostituisce l'ereditarietà con la delega, in modo da avere una classe che fa una sola cosa. In questo modo si evita di avere classi troppo grandi.

Pensando all'esempio dello stack, potremmo utilizzare un design dove lo stack che al posto di ereditare informazioni del vettore come, utilizzerà funzionalità di delega. incapsulando il vettore all'interno della classe. Una eredità selettiva e parziale.

```
public void push(Object elemento)
{
    _vector.insertElementAt(elemento, 0);
}
```

3.3.4 Replace conditional with polymorphism

Si sostituisce un costrutto condizionale con il polimorfismo, in modo da avere una classe che fa una sola cosa. In questo modo si evita di avere classi troppo grandi.

Prendiamo il calcolo dell'area:

```
private double a, b, r;
...
public double area()
{
    switch(type)
    {
        case SQUARE:
            return a * a;
        case RECTANGLE:
            return a * b;
```

```
            case CIRCLE:
                return Math.PI * r * r;
        }
    }
```

A seconda della forma geometrica, si calcola l'area in modo diverso. Ha più senso creare una classe astratta `Shape` che ha un metodo `area()` che viene implementato dalle classi `Square`, `Rectangle`, `Circle`.

```
interface class Shape
{
    public double area();
}

class Square implements Shape
{
    public double area()
    {
        return side * side;
    }
}
...
```

3.3.5 Separate domain from presentation

Separazione dal dominio della presentazione, in modo da avere una classe che fa una sola cosa. In questo modo si evita di avere classi troppo grandi. Dividendo quindi le responsabilità.