Appunti IS – Pietro

[Disclaimer: avrebbero meritato una revisione accurata, tante informazioni a volte ridondanti – anche colpa delle slide che le ritirano in mezzo secoli dopo – però questo è tutto il lavoro che ho voglia di fare, soprattuto considerando che lo sto facendo a fine febbraio]

Sommario

0 – Introduzione

[Tanto yapping sulle nostre capacità attuali, poi qualcosina di interessante, tipo queste due definizioni]

Lo sviluppo software è principalmente uno sforzo collaborativo e comunicativo, all'interno del quale la progettazione consente di raggiungere gli obiettivi prestabiliti con il committente.
L'obiettivo del corso, da cui l'uso della parola "ingegneria", è di fornire gli strumenti e le tecniche per lavorare e comunicare in modo efficace ed efficiente.

[Yapping sulla strutturazione del corso, il come funzionano i progetti e in cosa consiste l'esame]
[Pensiero generale sulle slide: casino spesso inconcludente che mischia italiano e inglese, in genere prendendo il peggio di entrambi]

1 – Requirements Elicitation

Con **software development lifecycle** ci si rifersice a quell'insieme di attività e relazioni tra esse che compongono lo sviluppo di un sistema software. In generale, individuiamo le seguenti fasi, ciascuna con uno specifico output:

- ***Requirements Elicitation***
- ***Requirements Analysis***
- ***System Design***
- ***Object Design***
- ***Implementation***
- ***Testing***

Le prime due fasi sono molto simili e si rifanno al processo di individuazione dei requisiti del sistema: in particolare con l'elicitation vogliamo definire il sistema in termini compresi dal cliente, mentre nella seconda vogliamo la specifica tecnica.

Possiamo inserire i progetti in tre grandi categorie:

- **Greenfield Engineering,** quando lo sviluppo inizia da zero e non si hanno sistemi precedenti su cui basarsi. Sono mossi da necessità degli utenti.
- **Re-engineering,** quando si effettua un re-design e/o una re-implementazione di un sistema esistente tramite nuove tecnologie.
- **Interface Engineering,** quando il progetto non modifica le funzionalità sottostanti di un sistema già esistente, ma si limita a migliorarne le interfacce sia uomo-macchina che tra sistema e terze parti.

Lo **statement of work** definisce il problema da risolvere, in particolare specificando: la situazione attuale, le funzionalità che dovrebbe introdurre il sistema, l'ambiente di lavoro, eventuali release parziali da mostrare, la data di consegna ed i criteri di accettazione.
I requisiti si dividono in diverse categorie:

- ***funzionali,*** cioè quelli che descrivono le interazioni tra il sistema e l'ambiente in cui si trova;
- ***non funzionali,*** cioè aspetti visibili all'utente che non sono strettamente legati alle funzioni; necessitano di metriche chiare. Categorie popolari sono: usabilità, affidabilità, performance e supportabilità;
- ***pseudo-requisiti*** o ***vincoli,*** cioè imposizioni del cliente o dell'ambiente di sviluppo, come l'uso di un determinato linguaggio.

La validazione dei requisiti è un passaggio cruciale nello sviluppo software e ruota attorno a diversi aspetti:

- ***correttezza,*** se i requisiti rappresentano la visione del cliente;
- ***completezza,*** se i requisiti descrivono ogni possibile modo in cui si può interagire con il sistema, comportamenti eccezionali inclusi;
- ***consistenza,*** se non ci sono requisiti che si contraddicono;

- ***realismo,*** se i requisiti possono essere implementati e consegnati;
- ***tracciabilità,*** se ogni funzione del sistema può essere ricollegata ad un requisito corrispondente.
    - Vale la pena differenziare tra *raggiungibili,* cioè se è possibile conseguire l'obiettivo nel contesto attuale, e *realizzabili,* cioè se è tecnicamente fattibile.

Ad ogni requisito si assegna inoltre una priorità: alta, media, bassa; va notato che in scenari reali i requisiti cambiano molto spesso.

Esistono svariate tecniche per raccogliere i requisiti, ciascuna con pro e contro:

- *Questionari,* che possono raggiungere moltissime persone in poco tempo, ma richiedono grande accuratezza nella scelta delle domande.
- *Interviste individuali,* che possono guidare facilmente l'intervistato sui temi che portano maggior informazioni, ma questi potrebbe non rispondere sinceramente.
- *Focus group,* che fanno emergere aree di consenso e conflitto, ma richiedono conduzione esperta per non avere figure che monopolizzano la discussione.
- *Osservazioni sul campo,* che permettono di vedere realmente un prodotto in uso, ma dispendiose per risorse e tempo.
- *Suggerimenti spontanei degli utenti,* dai costi irrisori, ma episodici.
- *Analisi della concorrenza e delle best practices,* per non reinventare la ruota e cercare di ottenere vantaggio competitivo, ma molto costose.

A riempire il gap tra utenti e sviluppatori troviamo:

- **Scenari,** cioè descrizioni narrative di cosa fanno le persone quando interagiscono con il sistema. Ne troviamo diverse sottocategorie:
    - *As-is,* che descrivono la situazione corrente.
    - *Visionary,* che descrivono un sistema futuro.
    - *Evaluation,* che servono a valutare il sistema rispetto a delle task utente.
    - *Training,* che guidano passo passo all'uso del sistema.
- **Use cases,** che descrivono il flusso di eventi nel sistema, eccezioni incluse. Hanno una condizione di inizio ed una di fine e gli attori che partecipano sono ben definiti.

Gli use cases possono essere legati tra loro tramite delle associazioni:

- **Include** rappresenta l'uso di uno use case da parte di un altro e lo si può vedere come una decomposizione funzionale. Lo use case base delega al secondo, il 'supplier', una parte delle operazioni da svolgere, per cui A non potrà esistere senza B.

- **Extend** rappresenta una estensione del problema originale: una relazione da A a B indica che B è uno use case completo di per sé, ma può essere esteso da A per includere un qualche scenario specifico.
- **Generalization** indica che uno use case astratto ha più specializzazioni o che più use case hanno degli elementi in comune e quindi possiamo 'tirarli fuori'. Ad esempio, 'Valida utente' è il padre di 'Verifica password' e 'Verifica impronta digitale'.

2 – Cicli di vita del software

Un modello di CVS è una caratterizzazione descrittiva o prescrittiva di come un sistema software viene sviluppato e, ad alto livello, possiamo dividerlo in tre fasi:

- **Definizione** (si occupa del cosa): comprende determinazione dei requisiti, informazioni da elaborare, comportamento del sistema, criteri di validazione, vincoli progettuali.
- **Sviluppo** (si occupa del come): comprende definizione del progetto, definizione dell'architettura software, traduzione del progetto nel linguaggio di programmazione, collaudi.
- **Manutenzione** (si occupa delle modifiche): comprende miglioramenti, correzioni, prevenzione, adattamenti.

Nel **modello a cascata** il processo è diviso in fasi sequenziali senza ricicli, in modo da controllare meglio sia tempi che costi. Le fasi vengono definite e separate per minimizzare le sovrapposizioni ed ognuna produce un semilavorato con annessa documentazione. Tali prodotti, inoltre, non possono essere modificati durante il processo di elaborazione successiva.
Le prime tre fasi compongono la "fase alta" del processo, le altre quella "bassa":

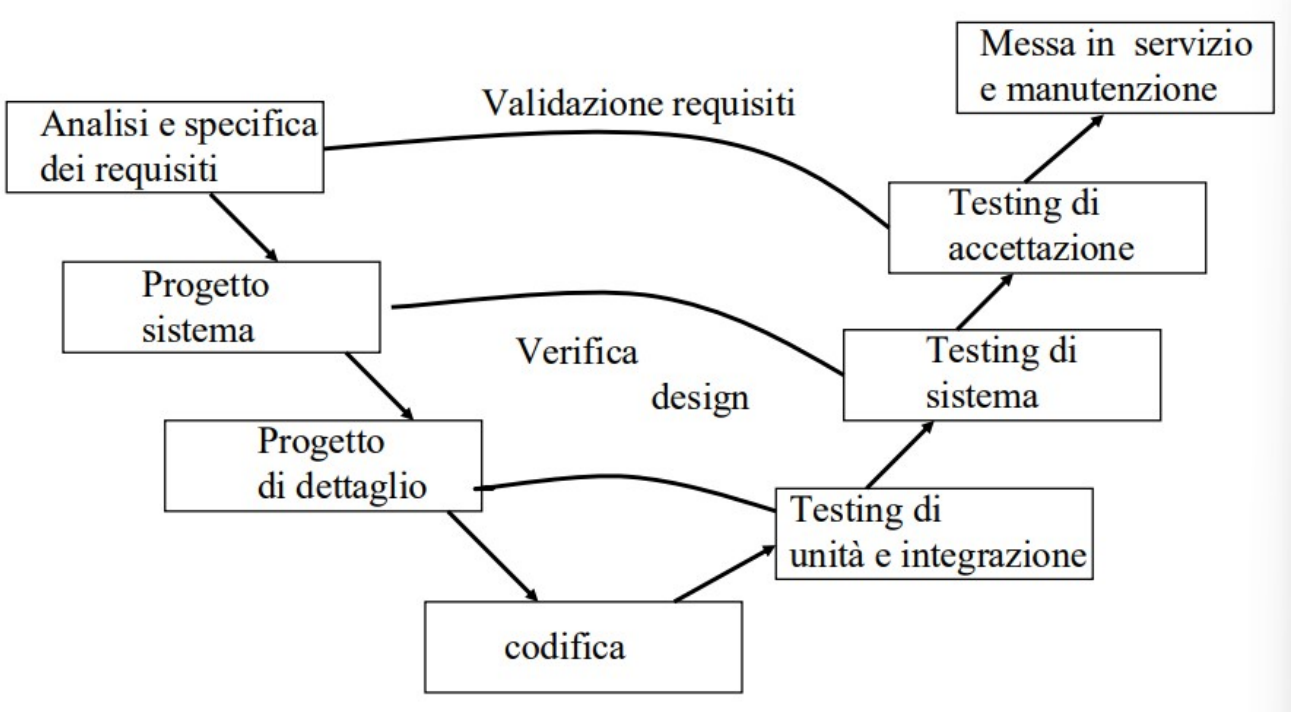
- **Studio di fattibilità**: Effettua una valutazione preliminare di costi e requisiti assieme al committente. L’obiettivo è quello di decidere la fattibilità del progetto, valutandone costi, tempi necessari e modalità di sviluppo. Output: documento di fattibilità.
- **Analisi e specifica dei requisiti**: Vengono analizzate le necessità dell’utente e del dominio d’applicazione del problema. Output: documento di specifica dei requisiti.
- **Progettazione**: Viene definita la struttura del software e il sistema viene scomposto in componenti e moduli. Output: definizione dei linguaggi e formalismi.
- **Programmazione e test di unità**: Ogni modulo viene codificato e testato separatamente dagli altri.

- **Integrazione e test di sistema**: I moduli vengono integrati tra loro e vengono testate le loro interazioni: per farlo al meglio si rilascia una alpha (se interna) o una beta (se esterna).
- **Deployment**: Rilascio del prodotto al cliente.
- **Manutenzione**: Gestione dell'evoluzione del software.

Si tratta di un modello facile sia da comprendere che da contemplare, ma limita l'interazione con il cliente, che avviene soltanto all'inizio ed alla fine. Anche i requisiti dell'utente sono scoperti alla fine e, se non sono tutti rispettati, bisogna ripartire da capo.

Svariati modelli sono stati proposti nel tempo per sopperire ai punti critici del modello a cascata: troviamo ad esempio delle varianti chiamate V&V e Feedback, che includono la possibilità di avere ricicli.
Con V&V aggiungiamo **Verification**, cioè stabiliamo se ci sia corrispondenza tra un prodotto software e la sua specifica, e **Validation**, cioè stabiliamo l'appropriatezza di un prodotto software rispetto alla sua missione operativa.
Il modello con **Feedback** invece prevede che da una certa fase sia possibile inviare dei feedback ad una qualunque delle fasi precedenti.

Il **modello a V** prevede una correlazione tra le fasi antecedenti alla codifica e quelle seguenti:



se si trova un problema in una delle fasi a destra, si ritorna alla corrispettiva fase a sinistra.

Altri modelli fanno invece ricorso ai **prototipi**, che sono usati per aiutare a comprendere i requisiti o per valutare la fattibilità di un approccio. Possiamo distinguere vari tipi di prototipazione:

- i **mock-up** prevedono la produzione completa dell'interfaccia utente e consentono di definire con completezza e senza ambiguità i requisiti.
- Con le **breadboards** si implementano sottinsiemi di funzionalità critiche per avere feedback sulla loro implementazione.
- Gli **usa e getta** sono usati per giungere ad una migliore comprensione dei requisiti, per cui si usano per le parti meno chiare.
- I prototipi **esplorativi** sono usati per giungere ad un prodotto finale partendo dai requisiti meglio compresi e lavorando a stretto contatto con il committente.

I prototipi vengono usati come base per il **modello evolutivo**, che è efficace per sistemi medio-piccoli o con cicli di vita brevi poiché comporta una perdita di visibilità del processo ed un prodotto finito scarsamente strutturato.

Nel **modello trasformazionale** lo sviluppo viene visto come una sequenza di passi che trasformano formalmente una specifica in un'implementazione. Può essere usato congiuntamente ad un altro modello per migliorare il processo di sviluppo di sistemi critici, ma richiede grande esperienza e rende difficile la specifica formale di aspetti come le interfacce.

I **modelli di sviluppo a componenti** si basano sul riuso sistematico di componenti esistenti sviluppate durante le varie fasi del ciclo di vita. In generale, in un **full reuse model** si ha repository da cui si prendono componenti già sviluppate e, durante lo sviluppo di un nuovo sistema, vi si caricano le nuove. È particolarmente adatto in un contesto object-oriented.

Poiché nel corso dello sviluppo è possibile che dei requisiti cambino, ci sono delle iterazioni di rework, applicabili in qualsiasi modello di processo di sviluppo. Da questo derivano due approcci correlati: lo sviluppo incrementale e quello a spirale.

Il **modello incrementale** viene usato spesso nella creazione di grandi software in tempi ristretti e si basa sul rilascio di versioni funzionanti che non soddisfano del tutto i requisiti del cliente. In genere, le fasi alte sono completamente realizzate e si scompone il sistema progettato in vari sottosistemi, ciascuno con una priorità diversa. Ciascun incremento corrisponde al rilascio di una parte delle funzionalità ed agisce inoltre come prototipo per individuare i requisiti successivi.

I **modelli iterativi** prevedono più versioni successive del sistema, in modo che in un certo istante esista una versione N in esercizio ed una N+1 in sviluppo. Ogni versione raffina o migliora le funzionalità, che sono tutte presenti fin dalla prima iterazione.

Il **modello a spirale** è una formalizzazione del concetto di iterazione, avendo come fondamento il riciclo. Il processo viene rappresentato come una spirale e ad ogni giro corrisponde una fase, articolata in 4 parti: determinazione di obiettivi e vincoli della fase, identificazione e riduzione dei rischi con valutazione di alternative, sviluppo e verifica, pianificazione della fase successiva. Abbiamo quindi un meta-modello risk-driven che ad ogni ciclo produce un deliverable. Rendere esplicita la gestione dei rischi richiede un aumento sia dei tempi di sviluppo che dei costi ed una gestione maggiore del team, ma comporta anche una certa attenzione al riuso, la determinazione rapida di errori ed aiuta a considerare gli aspetti della qualità.

L'**extreme programming** è un approccio recente allo sviluppo software basato su iterazioni veloci che rilasciano piccoli incrementi di funzionalità. Si basa su 12 regole:

- Progettare con il cliente.
- Test funzionali e unitari.
- Refactoring (riscrivere il codice senza alterarne le funzionalità esterne).
- Progettare al minimo.
- Descrivere il sistema con una metafora, anche per la descrizione formale.
- Proprietà del codice collettiva (contribuisce alla stesura chiunque sia coinvolto nel progetto).
- Scegliere ed utilizzare un preciso standard di scrittura del codice.
- Integrare continuamente i cambiamenti al codice.
- Il cliente deve essere presente e disponibile a verificare (sono consigliate riunioni settimanali).
- Open Workspace.
- 40 ore di lavoro settimanali.
- Pair Programming (due programmatori lavorano insieme su un solo computer).

Va da sé che è possibile combinare diversi approcci a seconda del caso specifico, in modo da prendere soltanto gli aspetti positivi (ad esempio, prototipazione per comprendere i requisiti per bene e poi waterfall).

3 – Introduzione UML

UML (Unified Modeling Language) è un **linguaggio di modellazione** che definisce una notazione standard per semplificare l'astrazione di un progetto software e nascondere i dettagli non necessari alla comprensione della struttura generale. È nato dall'unificazione dei metodi Booch-93, OMT ed OOSE e accogliendo varie idee di altri metodologi, tuttavia UML è indipendente da metodi, tecnologie e produttori. Si indica UML come linguaggio universale in quanto può specificare, costruire, visualizzare e documentare gli artefatti di un qualsiasi sistema, dai legacy ai moderni. UML basa la notazione su un meta-modello integrato, cioè definiscele relazioni tra i vari elementi, ed assume un processo basato sui casi d'uso, incentrato sull'architettura, interattivo ed incrementale. UML ha lo scopo di fornire all'utente un linguaggio di specifica espressivo, visuale e pronto all'uso, offrendo meccanismi di estensibilità e specializzazione del linguaggio e l'integrazione dei migliori approcci. Sono previsti diversi tipi di diagramma:

- Use Case diagrams, che descrivono i comportamenti funzionali del sistemi dal punto di vista dell’utente.
- Class diagrams, che descrivono la struttura statica del sistema (oggetti, attributi, associazioni).
- Sequence diagrams, che descrivono i comportamenti dinamici tra gli oggetti del sistema.
- Statechart diagrams, che descrivono il comportamento dinamico di un solo oggetto.
- Activity diagrams, che descrivono il comportamento dinamico del sistema, in particolare il suo flusso di lavoro.

Class – object; ultimi tre – dynamic

Uno **Use Case Diagram,** come già accennato precedentemente, mostra le modalità di uso del sistema, i suoi utilizzatori e le relazioni che intercorrono tra i due. Vi è almeno un attore presente, cioè un'entità che esterna che interagisce con il sistema, ed ogni use case ha una o più condizioni di entrata ed uscita, un flusso di eventi e delle eventuali condizioni eccezionali. Fa parte del ***modello funzionale*** del sistema.

I **diagrammi delle classi** rappresentano la struttura di un sistema, cioè le classi di oggetti del sistema e le relazioni tra loro. In UML una classe è composta dal nome, che la indica ed è univoco, i suoi attributi, dotati di un tipo, e le sue operazioni, dotate di firma; è possibile rappresentare una classe anche solo con la sezione nome, che può essere "simple" oppure "path", se preceduto dal package. Caratteristiche fondamentali:

- Un'**istanza** rappresenta un fenomeno, ha forma: nomeIstanza:nomeClasse e gli attributi sono rappresentati con i loro valori.
- Le **associazioni** servono a collegare le classi tra loro ed è possibile indicare la **molteplicità** dell'associazione per specificare il numero di istanze comprese: 1 a 1, 1 a Molti, Molti a Molti (molti indicato con un *).
  - Un'**aggregazione** è una relazione non forte che collega una classe padre a più componenti, dette classi figlie; si usa un rombo vuoto.
  - La **composizione** è invece un'associazione forte in cui le parti non esistono singolarmente, indicata con un rombo pieno.
  - L'**ereditarietà** crea una gerarchia "di tipo" in cui le classi figlie ereditano dalla classe padre gli attributi e le operazioni e si indica con un triangolo. La generalizzazione identifica concetti astratti a partire da quelli di più basso livello, mentre la specializzazione identifica concetti spiù pecifici da quelli di più alto livello.

Fa parte del ***modello ad oggetti*** del sistema.

Il **Sequence diagram** è utilizzato per definire la sequenza di eventi di un caso d'uso ed è uno dei principali input per l'implementazione dello scenario. Mostra gli oggetti coinvolti specificando la sequenza temporale dei messaggi scambiati, pertanto rappresenta un diagramma di interazione. Vi rappresentiamo:

- le istanze tramite dei rettangoli;
- gli attori come omini stilizzati;
- le lifelines con linee tratteggiate;
- i messaggi con delle frecce;
- le attivazioni con rettangoli stretti.

I Sequence diagram rappresentano il comportamento del sistema tramite le interazioni e fanno da complemento ai Class diagram, essendo utili per trovare oggetti mancanti. Rientrano nel ***modello dinamico*** del sistema.

Gli **State** (o StateChart) **diagrams** specificano il ciclo di vita di un oggetto, rappresentandone il comportamento in termini di: eventi a cui gli oggetti (la classe) sono sensibili, azioni prodotte e transizioni di stato. Graficamente, indichiamo lo stato iniziale con un cerchio nero e quello finale con un cerchio nero bordato, mentre gli altri intermedi sono dei rettangoli; le frecce che li collegano sono le transizioni.
Uno **stato** è una situazione in cui un oggetto si trova, in attesa di qualcosa o mentre esegue un azione, e presenta diversi attributi, tutti opzionali: nome, entry/exit actions, transizioni interne, attività (interrompibili) e sottostati, sia disgiunti che concorrenti.
I *sottostati sequenziali* si ottengono con una scomposizione OR ed ereditano le transizioni dei loro superstati. I *sottostati concorrenti* si ottengono con una

scomposizione AND e con loro il flusso di controllo si dirama in due percorsi paralleli che poi si riuniscono.
Le **transizioni** servono ad indicare un passaggio da uno stato iniziale ad uno finale (è possibile che siano uguali) e possiedono vari attributi opzionali: evento, che può essere un segnale o un messaggio da altri oggetti o ancora un cambiamento, condizione di guardia, un boolean da verificare in entrata, azione, eseguita durante la transizione e non interrompibile. Una transizione triggerless si verifica o quando la condizione di guardia diventa vera o, se questa non è presente, quando l'attività interna allo stato iniziale termina.
Anche questi appartengono al ***modello dinamico.***

Gli **Activity diagrams** forniscono la sequenza di operazioni che formano un'attività più complessa, permettondo di rappresentare processi paralleli e la loro sincronizzazione. Sono un retaggio della scomposizione funzionale e possono essere associati a classi, use case ed implementazioni di operazioni. Sono molto utili quando bisogna modellare dei flussi di lavoro, dei sistemi distribuiti, dei comportamenti sequenziali e la concorrenza. Abbiamo:

- attività, esecuzioni non atomiche entro uno stato che possono essere gerarchiche;
- transizione, il flusso di controllo tra due attività successive;
- guard expression, un espressione booleana che deve essere verificata per attivare una transition;
- branch, una diramazione che specifica percorsi alternativi in base a espressioni booleane. Ha una unica transition in ingresso e due o più transition in uscita;
- synchronization bar, usata per sincronizzare flussi concorrenti;
- fork, usata per dividere un flusso su più transition verso action state concorrenti;
- join, usate per unificare più transition da più action state concorrenti in uno solo;
- gli activity state sono stati non atomici (decomponibili ed interrompibili). Sono a loro volta rappresentabili con un activity diagram;
- gli action state rappresentano azioni atomiche (non decomponibili né interrompibili) e possono essere considerate come un caso particolare di activity state.

Anche questi fanno parte del ***modello dinamico.***

Altri diagrammi che possiamo trovare sono i seguenti:

Le **swimlanes** sono costrutti grafici che rappresentano un insieme partizionato di action/activity, identificando le responsabilità relative alle diverse operazioni. Le swimlane hanno un nome univoco ed ogni oggetto responsabile di un

action/activity appartiene ad una di esse; le transition possono attraversarne molteplici per raggiungere un certo stato. Gli object flow sono associazioni tra action/activity state ed object e possono essere input/output di una action, essere presenti più volte nello stesso diagramma ed essere manipolati da un qualsiasi numero di action. Sotto al nome dell'object è possibile rappresentarne lo stato indicandolo tra parentesi quadre.

I **package diagrams** aiutano a organizzare i modelli UML creati nelle fasi precedenti riducendo la complessità del sistema tramite un'organizzazione in sottosistemi. Un package raccoglie un insieme di classi che si occupano della parte funzionale del sistema, ma può raccogliere anche altri diagrammi. Si hanno dipendenze tra packages se esiste almeno una dipendenza tra classi appartenenti a packages differenti, motivo per cui in fase di Testing e Manutenzione bisogna fare attenzione quando si apportano modifiche.

4 – Analisi dei requisiti

L'**analisi dei requisiti** ha l'obiettivo di fornire un modello del sistema che sia corretto, completo, consistente e privo di ambiguità. In questa fase gli sviluppatori formalizzano la specifica delle richieste prodotta durante la fase di raccolta dei requisiti, chiariscono eventuali errori presenti nella specifica delle richieste ed esaminano più in dettaglio le condizioni limite ed i casi eccezionali. Viene esteso il modello di analisi (composto dal modello funzionale, quello ad oggetti e quello dinamico) per descrivere come gli attori ed il sistema interagiscono per manipolare il dominio di applicazione. Vengono definite tre tipologie di oggetti per classificarli all'interno del modello degli oggetti di analisi:

- **entity**, che rappresentano l'informazione persistente, cioè gli oggetti del dominio di applicazione (quelli di business);
- **boundary**, che rappresentano le interazioni tra attori e sistema, come oggetti relativi alle interfacce;
- **control**, che si occupano di realizzare gli use case e rappresentano il controllo dei task eseguiti dal sistema;

Questo approccio three-object-type rende i modelli più flessibili e facili da modificare.
UML fornisce il meccanismo degli stereotipi per consentire di aggiungere questa meta-informazione e si seguono alcune convenzioni sui nomi: gli oggetti Control possono avere il suffisso "Control", mentre per gli oggetti Boundary si dovrebbe scegliere qualcosa che ricordi elementi dell'interfaccia.

Le attività che consentono di trasformare gli use case e gli scenari della raccolta dei requisiti in un modello di analisi sono guidate da euristiche e comprendono:

- l'identificazione di oggetti entity, boundary, control;
- mappare gli use case in oggetti con sequence diagram;
- identificare associazioni, attributi, aggregati;
- modellare il comportamento dipendente dallo stato degli oggetti individuali;
- modellare le relazioni di ereditarietà;
- rivedere il modello di analisi.

La base del modello di analisi è formata dagli oggetti partecipanti, la cui individuazione è fatta esaminando gli use case. L'**euristica di Abbott** si basa sull'analisi del linguaggio naturale, mappando parti del parlato per modellare componenti:

- nome proprio – istanza;
- nome comune – classe;
- verbo fare/azione – operazioni;
- verbo essere – gerarchia;
- verbo avere – aggregazione;
- aggettivo – attributo;

date le imprecisioni del linguaggio naturale, cioò che si ottiene è una lista iniziale di candidati. Oltre a questa, esistono altre euristiche: sostantivi ricorrenti, termini che bisogna chiarire per comprendere gli use case, entità o attività del mondo reale da considerare, sorgenti o destinazioni di dati.
Ad ogni oggetto identificato vanno poi assegnati un nome univoco, in genere lo stesso usato da utenti e specialisti del dominio applicativo, ed una breve descrizione.

Gli **oggetti boundary** rappresentano l'interfaccia del sistema con gli attori, senza descriverne gli aspetti visuali, e, in genere, ogni attore interagisce con almeno uno di essi. Servono a collezionare informazioni dall'attore, traducendole in una forma che può essere usata sia da oggetti control che entity, e sono sviluppate, solitamente, in modo prototipale ed iterativo. Alcune euristiche per l'identificazione: identifica i controlli della UI di cui l'utente ha bisogno per iniziare lo use case e per inserire i dati nel sistema, identifica avvisi/messaggi con cui il sistema risponde, non modellare aspetti visuali con oggetti boundary (meglio mock-up).
Vanno sempre usati i termini dell'utente finale per descrivere l'interfaccia, mai quelli del dominio di implementazione.

In generale, gli **oggetti control** non hanno controparte nel mondo reale: vengono creati all'inizio dello use case e cessano di esistere al loro termine. Alcune euristiche: identificane uno per ogni use case o per ogni attore in uno use case, la vita di un oggetto control dovrebbe corrispondere alla durata di una

sessione utente e se è difficile identificare l'inizio o la fine vuol dire che entry o exit condition non sono ben definite.

Un **Sequence Diagram** mostra come il comportamento di uno use case è distribuito tra i suoi oggetti partecipanti tramite l'assegnazione di un insieme di operazione a ciascun oggetto.

- La colonna più a sinistra rappresenta l'attore che inizia lo use case, la seconda l'oggetto boundary con cui interagisce per iniziare e la terza è per l'oggetto control che gestisce il resto dello use case e si occupa di creare altri oggetti boundary.
- Le frecce orizzontali rappresentano i messaggi scambiati e la loro ricezione determina l'attivazione di operazioni. Queste sono graficamente rappresentate da rettangoli la cui lunghezza indica il tempo durante il quale l'operazione è attiva.
- La linea tratteggia che scorre dal basso verso l'alto indica il tempo in cui l'oggetto può ricevere messaggi.
- Gli oggetti che vengono creati durante l'interazione sono illustrati con il messaggio <<create>>, mentre quelli distrutti sono evidenziati da una croce.
- Gli oggetti entity non accedono mai agli oggetti control e boundary, cosa che rende più facile condividerli tra più use cases.

La **specifica delle responsabilità** consiste di:

- precondizioni, cioè condizioni per cui le operazioni possono essere eseguite e fornire un risultato corretto;
- postcondizioni, cioè condizioni che sono vere dopo l'esecuzione di un'operazione;
- class invariant, cioè condizioni che devono sempre restare vere;

Ci si riferisce a queste 3 con il termine ***contract.***

I sequence diagrams possono essere usati per fare **cross-checking**, cioè aiutano a controllare la completezza e la correttezza di use case model e class diagrams.

Il **class diagram** serve a mostrare le relazioni tra gli oggetti, sotto forma di associazioni, aggregazioni ed ereditarietà, consentendo di identificare i casi limite. Rappresentano la struttura del sistema e sono usati durante per modellare sottosistemi, interfacce e classi.
Un attore è un'entità esterna che interagisce con il sistema, una classe è un'astrazione che modella un'entità del problem domain e un oggetto è una specifica istanza di una classe.

Le associazioni sono sostanzialmente mapping bidirezionali, mentre i link sono connessioni tra istanze di oggetti (come una tupla). Euristiche: esaminare i verbi, eliminare le associazioni derivabili da altre, preoccuparsi della molteplicità solo quando l'insieme è stabile.
Gli attributi indicano proprietà individuali degli oggetti e vanno identificati solo quelli rilevanti al sistema. Sono caratterizzati da nome, breve descrizione e tipo, che descrive i valori possibili che può assumere. Hanno priorità minore delle associazioni e possono essere aggiunti anche dopo che l'analisi è finita. Euristiche: frasi possessive o aggettivi, rappresentare lo stato memorizzato come un attributo di un oggetto entity, guardare i contratti delle operazioni, descrivere i dettagli a modello stabile.

Gli **statechart diagram** sono utili da costruire per gli oggetti con un ciclo di vita di durata significativa, quindi spesso i control, a volte gli entity e mai i boundary.

Il **modello di analisi** viene costruito in modo incrementale ed iterativo e solo quando è stabile si esegue una riesamina prima tra sviluppatori e poi assieme al cliente. La revisione è necessaria a stabilire se la specifica risulta corretta, completa, consistente e chiara. Una volta documetata l'analisi tramite ***RAD,*** si procede ad assegnare le responsabilità e si giunge infine ad un accordo con il cliente.

Ci sono tre tipi di ruoli quando si parla di responsabilità: **generazione di informazione**, **integrazione** e **revisione**.
L'utente finale è esperto del dominio di applicazione e genera informazioni sul sistema corrente, mentre il cliente (integrazione) definisce lo scopo del sistema sulla base delle richieste che riceve dall'utente.
L'analista è lo sviluppatore con la maggiore conoscenza del dominio di applicazione e genera le informazioni sul sistema da sviluppare; ciascuno si occupa di almeno uno use case. L'architetto (integrazione) unifica use case ed oggetti dal punto di vista del sistema.
Il manager delle configurazione è responsabile di mantenere la storia delle revisioni e la tracciabilità del RAD con gli altri documenti.
Infine il reviewer valida il RAD per correttezza, completezza, consistenza e chiarezza.

L'accordo con il cliente rappresenta l'accettazione del modello di analisi: ci si accorda non solo su funzioni e caratteristiche del sistema ma anche sullo scheduling delle attività, il budget, le priorità e la lista di criteri per accettare o meno il sistema.

5 – Modelli dinamici

I **modelli dinamici** sono modelli operazionali che descrivono il comportamento del sistema in funzione del tempo e sono utili soprattuto per sistemi orientati al controllo. Comprendono Interaction diagrams, cioè sequence e collaboration diagram, e statechart diagrams, di cui un caso speciale sono gli activity diagrams.
Una interazione è un comportamento che comprende un insieme di messaggi scambiati tra un insieme di oggetti nell'ambito di un contesto per raggiungere uno scopo.
Un **messaggio** è una specificazione di una comunicazione tra oggetti e trasmette informazioni utili per un'attività; la loro ricezione può essere considerata istanza di un evento. È necessario specificare ricevente e messaggio e possono esserci informazioni aggiuntive.

In un sequence diagram i messaggi sincroni sono rappresentati da una freccia chiusa ed il ritorno, sempre opzionale, è rappresentato da una freccia tratteggiata. I messaggi asincroni sono invece indicati da una freccia aperta e descrivono interazioni concorrenti. Per l'esecuzione condizionata di un metodo si indica [condizione] prima del nome del metodo e, a meno che non sia esplicitamente modellato ogni caso, il diagramma non dice cosa succede se essa non è verificata.
Gli oggetti creati durante l'interazione vanno posti all'altezza della action di creazione.
È possibile rappresentare un ciclo raggruppando con un riquadro i messaggi da iterare, aggiungendo un eventuale condizione in alto a sinistra tra parentesi quadre. I costrutti che è possibile indicare su un box sono Loop, Alt (il-then-else) e Opt (if-then).
Per descrivere un oggetto che invoca un proprio metodo si usa un'auto-chiamata, indicata con una "freccia circolare" che rimane entro il lifetime di uno stesso metodo.
Distinguiamo due strutture nei sequence diagram: **fork** e **stair** diagrams. Nel primo caso la maggior parte del dinamismo si concentra su un singolo oggetto, in genere il control, mentre nel secondo il dinamismo viene diviso e ciascun oggetto delega alcune delle responsabilità ad altri. La struttura centralizzata consente l'inserimento di nuove operazioni, in quanto il loro ordine è modificabile, mentre la decentralizzata fornisce una forte connessione tra la operazioni che non consente cambi d'ordine.

Come passare da use case a sequence diagram:
il punto di partenza è una coppia attore – oggetto, in cui questo rappresenta l'intero sistema visto dall'esterno. Si rappresenta il messaggio con cui il caso

d'uso viene attivato e, in generale, trattare il caso base tralasciando le eccezioni, che possono essere aggiunte in seguito.
Bisogna assicurarsi che i metodi indicati siano gli stessi definiti nelle classi corrispondenti, con stesso numero e tipo di parametri. Si documenta ogni assunzione tramite delle note o delle condizioni. Si sceglie un titolo per ogni diagramma e, in generale, si ricorre sempre a nomi espressivi.
Non bisogna rappresentare tutto ciò che sarà presente nel codice e anzi è meglio evitare di inserire troppi dettagli, eventualmente scomponendo i più complessi in blocchi più semplici.

Un **collaboration diagram** specifica gli oggetti che collaborano tra loro in uno scenario ed i messaggi che si indirizzano. Qui si mette in evidenza il legame tra gli oggetti ed è possibile numerare i messaggi per visualizzarne meglio l'ordine sequenziale. Sono adatti per concorrenza, thread ed invocazioni innestate.

6 – System Design

Con **System Design** si indica la progettazione di un sistema, cioè la trasformazione del modello di analisi nel modello di progettazione del sistema. È costituito da diverse attività:

- Identificare gli obiettivi di design del progetto, in cui si delineano le caratteristiche di qualità che devono essere ottimizzate e qual è la loro priorità.
- Decomposizione del sistema in sottosistemi, usando stili architetturali standard e sulla base di use cases e modello di analisi.
- Raffinare la decomposizione.

Gli **obiettivi di design,** o design goals, descrivono la qualità del sistema e vengono derivati dai requisiti non funzionali.
L'**architettura software** descrive:

- la decomposizione del sistema in termini delle responsabilità dei sottosistemi, in modo da realizzarli in modo indipendente su più team, con le relative dipendenze e l'hardware associato;
- le decisioni relative a *control flow, controllo degli accessi* e *memorizzazione dei dati.*

I **boundary use case** descrivono la configurazione del sistema e le scelte relative a startup, shutdown e gestione delle eccezioni.

Un **sottosistema** è un insieme di un certo numero di classi del dominio della soluzione e, tipicamente, corrisponde ad una porzione di lavoro che può essere

svolta da un sottoteam di sviluppatori. Quando questi sono relativamente indipendenti è possibile lavorare con un overhead di comunicazione minimo. L'insieme delle operazioni fornite forma l'**interfaccia** del sottosistema ed include il nome delle operazioni, i loro parametri, il loro tipo ed i loro valori di ritorno. Due proprietà importanti dei sottosistemi sono:

- **Accoppiamento,** che indica il numero di dipendenze tra due sottosistemi. L'obiettivo è ridurlo al minimo (loosely coupled e non strongly coupled), poiché così i sistemi saranno indipendenti e le modifiche su uno impatteranno poco l'altro. Renderlo estremamente basso rischia di aggiungere un numero eccessivo di livelli di astrazione, che impattano su tempi di sviluppo ed elaborazione, ed andrebbe perseguito solo in sistemi con elevate probabilità di cambiamento nei sottosistemi.
- **Coesione,** che indica quanto è compatto un sottosistema, cioè le sue dipendenze interne. Vogliamo sottosistemi con alta coesione (high cohesion e non low cohesion) in modo che le classi realizzino compiti simili e siano collegate le une alle altre.

Un euristica dice che gli sviluppatori possono trattare, ad ogni livello di astrazione, un numero di concetti pari a 7+-2, dunque c'è un problema se ci sono più di 9 sottosistemi ad un certo livello di astrazione o se un sottosistema fornisce più di 9 servizi.

La decomposizione in sottosistemi si può fare in due modi:

- **Layer,** cioè una decomposizione gerarchica in un insieme ordinato di strati. Ciascun layer è un raggruppamento di sottosistemi che forniscono servizi correlati, può dipendere solo da quelli di livello più basso e non ha conoscenza di quelli ai livelli più alti. Si può differenziare tra:
  - ***architettura chiusa,*** se ogni layer può accedere solo al layer immediatamente sottostante. Forniscono alta portabilità e manutenibilità, ma aggiungono overhead di tempo e memoria.
  - ***architettura aperta,*** se un layer può accedere a tutti i layer sottostanti. Forniscono efficienza a runtime.
- **Partition,** cioè una suddivisione in sottosistemi **peer,** o pari fra loro, ognuno responsabile di differenti classi di servizi.

Gli **stili architetturali** che possono essere usati sono i seguenti:

- **a Repository,** in cui i sottosistemi accedono e modificano una singola struttura dati chiamata repository, che non ha conoscenza degli altri sottosistemi. Sono adatti per applicazioni con task di elaborazione dati che cambiano frequentemente ed è semplice definire sottosistemi addizionali. Il repository può facilmente diventare un collo di bottiglia ed inoltre il forte accoppiamento con ogni sottosistema rende difficile cambiarlo.
- **MVC,** in cui si classificano i sottosistemi in 3 tipi differenti:

    - ***Model,*** che mantiene la conoscenza del dominio di applicazione, cioè fornisce i metodi per accedere ai dati utili.
    - ***View,*** che fa visualizzare all'utente gli oggetti del dominio dell'applicazione, cioè si occupa dell'interazione con l'utente.
    - ***Controller,*** che è responsabile della sequenza di interazione con l'utente, ricevendone i comandi tramite View, e modifica lo stato degli altri due componenti.
    - Si tratta di un caso particolare di architettura di tipo repository, in cui il Model implementa la struttura dati centrale, il Controller gestisce il control flow ed il View visualizza il Model e viene notificato ad ogni suo cambiamento. È appropriato per sistemi interattivi ed è diffuso per le interfacce grafiche dei sistemi object oriented.
- **Client/Server,** in cui un sottosistema, detto server, fornisce specifici servizi ad un insieme di client che li richiedono attraverso una rete. I client conoscono l'interfaccia del server, ma non è vero il contrario; il flusso di controllo nei due è indipendente. Spesso usato nei sistemi di database.
- **Peer-to-Peer,** in cui ogni sottosistema può agire sia come client che come server, cioè richiede e fornisce servizi. Il flusso di controllo di ciascun sottosistema, eccezion fatta per la sincronizzazione sulle richieste, è indipendete dagli altri, ma c'è la possibilità di deadlock.
- **Three-tier,** in cui i sottosistemi sono separati in tre gruppi:
    - ***Interface layer,*** che include i *boundary object* che interfacciano con l'utente.
    - ***Application logic layer,*** che include tutti i *control object* e le entità che realizzano elaborazione, verifica e notifica delle richieste.
    - ***Storage layer,*** che effettua memorizzazione, recupero ed interrogazione degli oggetti persistenti.
- **Pipeline,** (o a flusso di dati), in cui ogni componente della catena lavora in modo indipendente dagli altri per trasformare i dati in input in una serie di dati in output.

Il primo passo del system design è l'individuazione dei design goal, che vanno formalizzati in modo esplicito poiché ogni decisione deve essere presa con lo stesso insieme di criteri. Le qualità desiderabili sono organizzate in gruppi:

- **Performance,** che includono i requisiti imposti al sistema in termini di spazio e velocità.
- **Dependability,** che riuniscono robustezza, affidabilità, disponibilità, tolleranza agli errori e sicurezza.
- **Cost,** che includono i costi per sviluppare il sistema, installarlo, mantenerlo e amministrarlo, oltre che i costi per addestrare il personale e gli eventuali costi dovuti al passaggio dal vecchio sistema a questo.

- **Maintenance,** che riguardano la difficoltà nel modificare il sistema dopo il suo rilascio. Troviamo estensibilità, modificabilità, adattabilità, portabilità, leggibilità e tracciabilità dei requisiti.
- **End User,** che riguardano le qualità desiderabili per un utente e che però non rientrano in performance o dependability, come utilità ed usabilità.

Ai design goal vanno assegnate delle priorità, che dipendono dai trade-off a cui gli sviluppatori devono far fronte, ad esempio: favorire una tra la velocità o il risparmio di memoria, rilasciare il prodotto nei tempi richiesti ma con meno funzionalità di quante ci si aspettava o richiedere tempo aggiuntivo per terminare, rilasciare il prodotto in tempo ma con dei bug conosciuti o rilasciarlo in ritardo ma corretto.

[Parte 2]

Il **component diagram,** o diagramma delle componenti, modella la struttura statica del sistema, cioè quella a design e compile time. È un grafo che rappresenta le interconnessioni tra le componenti software con relazioni di dipendenza.
Il **deployment diagram** modella la struttura dinamica del sistema, cioè quella a runtime. È un grafo in cui gli archi sono associazioni di comunicazione ed i nodi sono box 3D che contengono istanze di componenti, che a loro volta possono contenere oggetti.

Il **mapping hardware/software** consiste nel prendere decisioni riguardo le piattaforme su cui far girare il sistema e successivamente mapparvi le componenti. È possibile che vengano identificati nuovi oggetti e sottosistemi per spostare informazioni tra i nodi, ma questo, di contro, può implicare l'insorgere di problematiche, come quelle legate alla sincronizzazione.

La **gestione dei dati persistenti** consiste nell'identificare gli oggetti persistenti e scegliere il tipo di infrastruttura adatta a memorizzarli. Gli oggetti Entity sono dei buoni candidati di partenza, ma non è detto che tutti diventino persistenti, in quanto alcuni potrebbero non sopravvivere dopo una singola esecuzione del sistema. Tra i mezzi a disposizione per salvare i dati abbiamo:
- **File,** semplici ed economici, ma essendo di basso livello richiedono codice aggiuntivo per fornire gli opportuni livelli di astrazione.
- **DB relazionali,** organizzato in tabelle in cui gli oggetti devono essere mappati per essere salvati.
- **DB object-oriented,** in cui gli oggetti vengono mappati così come sono, riducendo i tempi di setup iniziale, ma richiedendo query più complesse.

Nella fase di **controllo di accesso** si stabilisce a quali informazioni e a quali funzioni può accedere ogni tipologia di attore. Si descrivono i diritti di accesso e si definiscono i meccanismi di autenticazione. Una **matrice di accesso** modella i diritti di accesso su una classe: le righe rappresentano gli attori, le colonne le classi e ad ogni entry corrispondono le operazioni consentite su un'istanza.

- **Tabella di accesso globale,** che rappresenta esplicitamente tutti gli accessi tramite una tupla attore-classe-operazione, pertanto richiede molto spazio.
- **Access Control List,** o ACL, che associa una lista di coppie attore-operazione per ogni classe.
- **Capability,** che associa una coppia classe-operazione ad un attore.

Nella fase di **controllo del flusso globale** viene descritta la sequenza di azioni ed interazioni con il sistema, sia che si basino su eventi esterni generati da attori che sul trascorrere del tempo.

- **Procedure-driven control,** se i controlli risiedono nel codice del programma. È semplice e facile da costruire per sistemi procedurali.
- **Event-driven control,** se il ciclo principale attende un evento esterno per rimandarlo all'oggetto appropriato. Il controllo risiede in un dispatcher che richiama funzioni di sottosistema. È flessibile ed usato spesso per le interfacce utente.
- **Threads,** che rappresentano una versione concorrente del Procedure-driven, poiché il sistema può creare un numero arbitrario di thread da usare per risondere ad eventi differenti.

È necessario identificare i processi concorrenti per analizzarne i problemi relativi e, in generale, oggetti concorrenti dovrebbero essere assegnati a thread di controllo differenti.

Nella fase di **identificazione delle condizioni limite** si studia il comportamento del sistema in fase di avvio e terminazione e si individuano e gestiscono i casi eccezionali. In generale, gli use case vengono identificati esaminando sottosistemi ed oggetti persistenti:

- **Configurazione:** per ogni oggetto persistente, si esaminano gli use case in cui viene creato o distrutto e per ogni operazione mancante si aggiunge uno use case invocato dall'admin.
- **Avvio e terminazione:** per ogni componente si aggiungono gli use cases *start, shutdown* e *configure.*
- **Gestione eccezioni:** per ogni tipo di fallimento di componente si decide come il sistema debba reagire.

Anche il system design è un'attività iterativa in cui è necessaria un'attività di revisione ad opera di manager e sviluppatori. Bisogna inoltre assicurarsi che il system design sia:

- **Corretto,** cioè se il modello di analisi vi può essere mappato.
- **Completo,** cioè se ogni requisito e ogni caratteristica è stata portata a compimento.
- **Consistente,** cioè se non contiene contraddizioni (ad esempio design gol che violano requisiti non funzionali).
- **Realistico,** cioè se il sistema può essere realizzato ed è possibile rispettare problemi di concorrenza e tecnologie.
- **Leggibile,** cioè se i nomi e le descrizioni scritte sono sufficientemente dettagliate.

I **liaisons** sono rappresentanti dei team che lavorano ai diversi sottosistemi, raccogliendo informazioni da e verso il loro team di appartenenza. Negoziano inoltre i cambiamenti di interfacce, in quanto le modifiche vanno effettuate con attenzione.

7 – Object Design

Durante la fase di **object design** vengono aggiunti dettagli all'analisi dei requisiti, si prendono decisioni di implementazione e si raffinano gli oggetti esistenti. L'object designer sceglie tra i diversi modi di implementare il modello di analisi con l'obiettivo di minimizzare tempi di esecuzione, costi, ecc.
Vengono selezionate librerie e componenti utili per le strutture dati ed i servizi di base e si scelgono i Design Pattern necessari per risolvere problemi comuni.

Durante la **specifica delle interfacce** si raffina ulteriormente la suddivisione in sottosistemi effettuata durante il system design, ponendo l'attenzione sui dettagli di basso livello degli oggetti e descrivendo l'interfaccia di ciascuno di essi.
Vengono specificate signature e visibilità di ogni operazione e, tramite *Object Constraint Language,* precondizioni, postcondizioni ed invarianti (predicati sempre veri per tutte le istanze di una classe).
Il risultato è una specifica completa delle interfacce per ogni sottosistema, spesso chiamate "API del sottosistema".

Durante la fase di **ristrutturazione** il modello di sistema viene modificato per aumentare il riuso del codice o soddisfare ulteriori design goal, come mantenimento e leggibilità. Alcune attività includono fusioni o decomposizioni di classi e trasformazioni delle associazioni.

Durante la fase di **ottimizzazione** ci si occupa di soddisfare i requisiti di performance del modello di sistema, tramite attività quali riduzione delle molteplicità nelle associazioni o aggiunta di informazioni ridondanti.

L'**Object Design Document,** o ODD, serve sia a scambiare informazioni sulle interfacce tra i team che a fare da riferimento per la fase di testing:

- la sezione ***introduzione*** contiene una descrizione dei trade-offs e le convenzioni e le linee guida per il miglioramento della comunicazione;
- la sezione ***packages*** descrive la decomposizione del sottosistema in packages e l'organizzazione in file del codice, mostrando dipendenze ed uso;
- la sezione ***class interfaces*** descrive le classi e le loro interfacce pubbliche.

È possibile generare queste ultime due sezione in automatico tramite **Javadoc,** usando annotazioni specifiche sul codice sorgente.

[Riuso e flessibilità – Ereditarietà, Composizione e Design Patterns]

Gli obiettivi dell'**attività di riuso** sono realizzare nuove funzionalità a partire da quelle già disponibili e risolvere i problemi correnti grazie alle conoscenze precedentemente acquisite. Il primo punto richiede ereditarietà e composizione, mentre per il secondo ci sono i Design Pattern.
L'uso dell'**ereditarietà** permette di ridurre le ridondanze e di migliorare l'estendibilità e la modificabilità del codice. È detta riuso *White-Box*, in quanto la struttura interna delle classi è visibile alle sottoclassi.

- Con *ereditarietà di specifica* si definisce la possibilità di usare un oggetto al posto di un altro e la sua definizione formale è "se un oggetto di tipo S può essere sostituito ovunque ci si aspetta un oggetto di tipo T, allora S è un sottotipo di T".
- Con *ereditarietà di implementazione* si definisce l'implementazione di un oggetto in funzione dell'implementazione di un altro oggetto.

L'uso della **composizione** permette di ottenere funzionalità tramite aggregazione ed è detta riuso *Black-Box*, in quanto i dettagli implementativi interni agli oggetti non sono visibili all'esterno.

- Con la *delega* due oggetti collaborano nel gestire una richiesta: il Receiver ottiene la richiesta del Client e ne delega l'esecuzione ad un oggetto Delegate, in modo che quest'ultimo non possa essere usato in modo scorretto.

La delega rende le componenti più resistenti rispetto al fare subtyping e si guadagna in flessibilità, ma si perde in estendibilità e velocità a runtime.

I **Design Patterns** descrivono problemi ricorrenti presentando soluzioni generali applicabili in moltissimi casi. In genere sono composti da poche classi legate tramite delegazione ed ereditarietà, in modo da essere robuste e modificabili per essere adattate al problema specifico che si sta affrontando.
Sono descritti tramite un *nome,* una *descrizione del problema,* che indica le situazioni in cui può essere usato, una *soluzione,* che descrive gli elementi che costituiscono il progetto, ed un *insieme di conseguenze,* che descrive i risultati e i vincoli che si ottengono applicandolo. Li dividiamo in tre grandi gruppi:

- **Pattern strutturali,** che si occupano del modo in cui le classi e gli oggetti formano strutture complesse e che riducono l'accoppiamento tra classi tramite l'incapsulamento e la creazione di classi astratte che facilitino estensioni future.
- **Pattern comportamentali,** che si occupano dell'assegnazione delle responsabilità tra oggetti che collaborano tra loro e che caratterizzano i flussi di controllo complessi.
- **Pattern creazionali,** che forniscono un'astrazione del processo di creazione degli oggetti e aiutano a rendere un sistema indipendente dai meccanismi di creazione, composizione e rappresentazione degli oggetti.

Il **Composite pattern** compone gli oggetti in strutture ad albero ad ampiezza e profondità variabili. Nelle gerarchie formate i nodi foglia rappresentano oggetti individuali e quelli interni invece oggetti composti ed è possibile trattare tutti in maniera uniforme attraverso un'interfaccia comune.
L'oggetto *Composite* ha un'associazione di aggregazione con l'interfaccia *Component* ed implementa ogni servizio iterando su ogni Component che contiene, mentre i servizi *Leaf* si occupano del lavoro effettivo.
È possibile modificare e aggiungere con facilità i Leaf.

Il **Façade pattern** fornisce un'unica interfaccia per accedere ad un insieme di oggetti che compongono un sottosistema. Facilita l'ottenimento di un'architettura chiusa, evita l'uso improprio degli elementi dei sottosistemi e favorisce il disaccoppiamento tra essi e il client.

L'**Adapter pattern** consente di convertire l'interfaccia di una classe in un'altra, in modo da incapsulare componenti esistenti o off the shelf ed usarle.
Per implementarlo c'è bisogna di due componenti: un'interfaccia *Target* i cui metodi sono implementati in termini di richieste alla componente esterna ed una classe *Adapter* che la implementa e che si occupa di delegare le richieste ed eventualmente eseguire le conversioni tra strutture dati.

Il **Bridge pattern** separa l'astrazione dall'implementazione, in modo da poter interfacciare un insieme di oggetti anche quando questi non sono del tutto noti o

completi. È utile sia per testare diverse implementazioni di una stessa interfaccia che per estendere un sottosistema anche dopo la sua consegna.

Il **Proxy pattern** consente di ridurre i costi di accesso agli oggetti usando delle controfigure, in modo da creare ed inizializzare soltanto quelli necessari quando vengono richiesti dall'utente.

L'**Observer pattern** è un behevioral pattern che si occupa della gestione degli eventi tramite diversi oggetti: una classe *Subject* è l'oggetto osservato da più *Observer,* interfacce astratte che rappresentano una vista dell'oggetto e gestiscono un certo evento ad esso legato. Queste interfacce sono implementate dai *ConcreteObserver,* che si occupano effettivamente delle azioni da intraprendere.

L'**Abstract factory** è un creational pattern che ha lo scopo di fornire un'interfaccia per la creazione di famiglie di oggetti correlati, senza specificare quali siano le loro classi concrete.
L'interfaccia astratta *AbstractProduct* rappresenta l'interfaccia di un concetto singolo, mentre *AbstractFactory,* dichiara le operazioni per creare ogni singolo prodotto. *ConcreteFactory* si occupa invece della creazione vera e propria di un certo *Product.*
Rende il client indipendente dalle classi utilizzate effettivamente per l'implementazione degli oggetti, promuovendo la coerenza nell'uso dei prodotti, ma rendendo difficile l'aggiunta di nuovi.

[Mapping Models to Code]

Dopo aver specificato le interfacce ed aver raffinato le relazioni esistenti tra le classi è possibile implementare il sistema che realizza gli use case specificati nelle fasi di analisi e system design. Ci sono però svariati problemi a cui uno sviluppatore va in contro, come parametri non documentati o assenti ed il non supporto dei contratti o del concetto di associazione da parte del linguaggio di programmazione scelto. Tutte queste modifiche locali non sono particolarmente complesse, ma introducono il rischio di errori.

Una **trasformazione** ha lo scopo di migliorare un aspetto del modello, pur continuando a preservarne tutte le altre proprietà. In genere è localizzata, impatta un numero ristretto di classi, attributi ed operazioni e viene eseguita in una serie di passi semplici. Ve ne sono 4 tipi:

- **Trasformazioni di modello,** che operano sul modello ad oggetti, ad esempio convertendo un attributo semplice in una classe. Le

trasformazioni includono aggiunta e rimozione di classi, associazioni, attributi o informazioni dal modello o le attività di rinomina.
- **Refactoring,** che opera sul codice sorgente per renderlo più leggibile o facilmente modificabile. Richiede testing ad ogni passo, in quanto bisogna evitare di cambiare il comportamento del sistema.
- **Forward engineering,** che produce un template del codice sorgente corrispondente al modello ad oggetti. Si riduce il numero di errori introdotti in fase di implementazione (in base all'accuratezza della progettazione) e si riducono gli sforzi implementativi.
- **Reverse engineering,** in cui si produce, con un certo aiuto da parte degli sviluppatori, il modello ad oggetti corrispondente al codice sorgente. Viene creata una classe UML per ciascuna classe, aggiungendo un attributo per ciascun campo ed un'operazione per ciascun metodo.

Per evitare che le trasformazioni introducano errori difficili da rilevare, è necessario che ciascuna riguardi un singolo design goal ed effettui pochi cambiamenti per volta, in modo isolato e con validazione successiva.

Con **ottimizzazione del modello di Object Design** ci si riferisce ad una serie di operazioni per migliorare le prestazioni del modello di sistema:
- ***Ottimizzare i cammini di accesso,*** aggiungendo associazioni per semplificare le operazioni più frequenti, trasformando le associazioni di tipo "molti" in associazioni qualificate e si verifica che non ci siano attributi mal collocati.
- ***Collassare gli oggetti in attributi,*** nel caso alcune classi presentino pochi attributi e comportamenti e siano legate soltanto ad un'altra classe.
- ***Ritardare le elaborazioni costose,*** nel caso si voglia ritardare la creazione di oggetti costosi fin quando non siano necessari.
- ***Memorizzare il risultato di elaborazioni costose,*** come nel caso di metodi invocati diverse volte ed il cui risultato cambia raramente.

Le **associazioni** sono concetti UML che denotano collezioni di link bidirezionali tra due o più oggetti, ma i linguaggi OO non li forniscono e bisogna quindi effettuare delle trasformazioni.
- Le associazioni uno a uno unidirezionali diventano riferimenti dalla classe chiamante alla chiamata, mentre quelle bidirezionali hanno bisogno di valutazioni specifiche e beneficiano di getter e setter per ridurre gli impatti delle modifiche.
- Le associazioni uno a molti richiedono l'uso di oggetti di tipo *Collection* e sono realizzate in modo bidirezionale; per le molti a molti ci sono oggetti *Collection* in entrambe le classi.
- Le classi associative, che raccolgono attributi ed operazioni legate su un'associazione, diventano classi che fanno da tramite tra le due classi.

Un **contratto** è un vincolo su una classe che deve essere soddisfatto primi di poterla utilizzare e, per gestirne le violazioni, si ricorre al meccanismo delle eccezioni. Specificare tutti i contratti rischia di introdurre errori, oltre che peggiorare le prestazioni globali, per cui è possibile omettere i controlli sia sulle postcondizioni che sugli invarianti, così come per metodi privati e protetti a patto che siano ben definite le interfacce. Le componenti su cui concentrarsi davvero sono quelli che durano a lungo, come gli Entity, e si dovrebbe cercare di riusare il codice per quanto possibile.

L'ultima fase consiste nel **mapping** dell'Object Model in uno schema di memorizzazione persistente. Poiché i linguaggi OO non forniscono metodi efficienti, è necessario usare strutture dati che possono essere memorizzate in sistemi di gestione: nel caso dei DB OO non c'è bisogno di alcuna trasformazione, mentre per DB relazionalii e file è necessaria un'infrastruttura per effettuare le conversioni.

8 – Testing

Il testing consiste nell'identificare le differenze tra comportamento atteso e comportamento osservato e ne troviamo diverse tipologie, a seconda di cosa si stia analizzando nello specifico.
L'obiettivo di questa fase è progettare una serie di test in modo da trovare il maggior numero possibile di errori, così da correggerli.
Con il **software quality management** ci si occupa di garantire che il livello di qualità richiesto venga raggiunto tramite la definizione di standard e procedure di qualità e la verifica della loro applicazione.
Le parole chiave in questa fase sono:

- **affidabilità,** cioè la misura di successo con cui il comportamento del sistema è conforme ad una certa specifica;
- **fallimento,** o failure, cioè qualsiasi deviazione del comportamento;
- **stato di errore,** cioè uno stato tale che ogni ulteriore elaborazione da parte del sistema porta ad un fallimento;
- **difetto,** detto anche bug o fault, cioè la causa di un failure.

Non tutti i fault generano un failure, un failure può essere generato da più fault ed un fault può generare diversi failure.
È possibile usare **defect** quando non si fa distinzione tra fault e failure.

Un **test case** contiene un insieme di dati di input per testare una funzionalità assieme al risultato atteso, mentre una **test suite** è un insieme di casi di test.

L'esecuzione del test consiste nell'eseguire il programma per tutti i casi di test della suite e si considera un successo quando il test rileva almeno un failure. Con **oracolo** si indica il risultato atteso per un test case e può essere sia scritto da un umano che generato automaticamente sulla base di specifiche formali.

I fault possono essere classificati in diverse tipologie:

- **Fault nella specifica delle interfacce,** se c'è un disallineamento tra quello che il client ha bisogno e ciò che gli viene offerto o tra la specifica e l'esecuzione.
- **Algorithmic fault,** se ci sono inizializzazioni mancanti, errori di ramificazione, prove mancanti per zero, ecc.
- **Mechanical fault,** se la documentazione non è aderente alle condizioni reali ed alle procedure operative.
- **Errori,** che possono scaturire da sovraccarichi o essere legati a performance, throughput o capacità.

Esistono diverse tecniche per aumentare l'affidabilità di un sistema software:

- **Fault avoidance,** che include tecniche che tentano di prevenire l'inserimento di difetti nel sistema prima della sua realizzazione.
- **Fault detection,** che include tecniche per identificare stati di errore e trovare il difetto che li causa.
  - Il ***debugging*** è il processo di individuazione e correzione dei bug dopo che è stato rilevato un errore. Parte dal presupposto che un failure non pianificato evidenzi un problema nel codice o nella logica.
  - Il ***testing*** è una tecnica che tenta di creare stati di fallimento o errore in modo pianificato e proattivo.
  - La ***review*** è un'ispezione manuale di alcuni aspetti del sistema, senza eseguirlo. È possibile suddividere ulteriormente tra *Walkthrough,* se lo sviluppatore presenta informalmente API, codice e documentazione al team di review, ed *Inspection,* se la presentazione è formale e lo sviluppatore può intervenire solo per delle richieste di chiarimento.
- **Fault tolerance,** che include tecniche per gestire i fault a runtime effettuando il recovery.

Soltanto un testing esaustivo può dimostrare la correttezza, ma è impraticabile per motivi di tempo e spesso banale. Il programma si reputa testato a sufficienza quando si raggiunge il limite di tempo o costo prefissato oppure il criterio di copertura, cioè una percentuale predefinita degli elementi definita nei criteri di accettazione.

Una **componente** è una parte isolabile del sistema che può essere testata.
Un **test case** si compone di 5 attributi:

- **Name,** per distinguerlo dagli altri.
- **Location,** per descrivere dove può essere trovato (path o URL).
- **Input,** che descrive l'insieme di dati di input o comandi che l'attore deve inserire.
- **Oracle,** che descrive il comportamento atteso, cioè la sequenza di dati in output che la corretta esecuzione dovrebbe avere.
- **Log,** cioè l'insieme delle correlazioni del comportamento osservato con quello atteso.

I test case possono essere legati da due tipi di relazioni, *aggregation*, nel caso possano essere decomposti in subset, e *precedence*, se devono essere eseguiti in un certo ordine; è preferibile che ve ne siano poche per velocizzare il processo.
Dividiamo inoltre i test case in **blackbox,** se testano principalmente I/O, e **whitebox** se invece ci si focalizza sulla struttura interna della componente.

Poiché le componenti vanno isolate per essere testate in modo indipendente, è necessario sostituire le parti mancanti: un **test stub** è un'implementazione parziale di componenti da cui la parte testata dipende, mentre un **test driver** è per le componenti che dipendono dalla parte testata.

Una **correzione** è un cambiamento di una componente per riparare un fault. Poiché quest'attività potrebbe introdurre nuovi difetti, abbiamo diverse tecniche per gestire la situazione:

- **Problem tracking,** che include la documentazione di ogni fallimento, la sua correzione e la revisione delle componenti coinvolte.
- **Regression testing,** che richiede la riesecuzione di tutti i test precedenti dopo il cambiamento.
- **Rational maintenance,** che include la documentazione delle motivazioni alla base dei cambiamenti e le relazioni dietro le motivazioni della revisione della componente.

Con **managing testing** ci si riferisce a tutte le tecniche e gli accorgimenti necessari a gestire le attività di testing per minimizzare le risorse necessarie. Sono necessari 4 tipi di documenti:

- **Test plan,** che si focalizza sugli aspetti manageriali. Documenta approccio e risorse da usare assieme a requisiti e componenti da testare.
- **Test case specification,** in cui si documentano i test case. Troviamo input, drivers, stubs ed output attesi dei test.
- **Test incident report,** in cui si documenta l'esecuzione di ogni test case e, per ogni test fallito, fornisce informazioni sul risultato effettivo rispetto a quello atteso e sul motivo del fallimento.

- **Test report summary,** che elenca tutti i fallimenti rilevati e che devono essere investigati.

Oltre allo unit testing, esistono molti altri tipi di testing:
- **Integration testing,** fatto dagli sviluppatori, in cui si testano gruppi di sottosistemi sempre più grandi, fino all'intero sistema, per testare le interfacce.
- **System testing,** fatto dagli sviluppatori, in cui si testa l'intero sistema per determinare se rispetti i requisiti funzionali.
- **Acceptance testing,** fatto dal cliente, per verificare che rispetti i requisiti del cliente e sia pronto all'uso.

Tra le attività di testing troviamo:
- **Component inspection,** che trova i fault in una componente individuale tramite l'ispezione manuale del sorgente. Il processo proposto da *Parnas* snellisce quello di *Fagan* e prende il nome di *Active Design Review:* nella fase *preparation* i revisori analizzano l'implementazione della componente ed individuano i fault, per poi compilare un questionario, e successivamente l'autore incontra separatamente ciascuno di loro piuttosto che fare un meeting generale.
- **Usability testing,** che testa la comprensibilità del sistema analizzando il comportamento dell'utente che interagisce con le interfacce in base a degli obiettivi preposti. Distinguiamo in *scenario test,* in cui si presenta agli utenti uno scenario visionario e li si fa interagire con mock-up e storyboard, *prototype test,* in cui viene presentata una parte del software che implementa gli aspetti chiave del sistema, e *product test,* in cui si usa una versione funzionale del sistema.

Durante il blackbox testing per unit testing è possibile differenziare tra:
- **Equivalence testing,** in cui si riduce il numero di test case effettuando un partizionamento degli input per classi di equivalenza.
- **Boundary testing,** in cui si selezionano gli elementi ai confini della classe di equivalenza.

Il **boundary value testing** (BVT) è efficace per errori semplici ai confini del dominio valido, mentre il **robustness testing** (RT) estende quest'ultimo testando anche input non validi. Con **worst case testing** (WCT) ci si concentra sulle combinazioni di input che rappresentano i peggiori scenati possibili ed anche di questo esiste una versione **robust** che include valori non validi.

Il whitebox testing è a sua volta diviso in 4 tipologie:
- **Statement testing,** in cui si testano i singoli statement.

- **Loop testing,** in cui i test case si assicurano che il loop sia completamente saltato, poi eseguito una sola volta e poi eseguito più volte.
- **Path testing,** in cui ci si assicura che tutti i path vengano eseguiti.
- **Branch testing,** in cui ci si assicura che ogni possibile uscita da una condizione sia testata almeno una volta.

Ci sono diversi modi di approcciarsi all'integration testing:

- **Big Bang approach,** in cui le componenti sono testate tutte assieme come un unico sistema.
- **Bottom-up strategy,** in cui il testing viene effettuato a partire dal layer più basso della gerarchia.
- **Top-down strategy,** in cui sono testati prima i layer al top o i sottosistemi di controllo.
- **Sandwitch strategy,** in cui si uniscono bottom-up e top-down per testare i due layer attorno al target. La versione **modified** prevede una fase di testing individuale dei tre layer, seguita da due test che coinvolgono il target ed uno degli altri due.

Il system testing è diviso in più attività:

- **Functional testing,** che essenzialmente testa le funzionalità del sistema come nel metodo blackbox: i test case sono progettati sulla base dei requisiti e si focalizzano sulle richieste e le funzioni chiave.
- **Pilot testing,** in cui non viene fornita nessuna linea guida al gruppo di utenti selezionati per provarlo. Si parla di ***alpha test*** se effettuato nell'ambiente di sviluppo e ***beta*** se effettuato in quello di utilizzo.
- **Performance testing,** che fornisce valutazioni delle misure di performance del sistema, come volume, compatibilità, sicurezza, ecc.
- **Acceptance testing,** in cui il cliente valuta il sistema. Potrà usare un insieme di test case suoi che rappresentano le condizioni tipiche in cui si opera e confrontarlo con sistemi esistenti, sia concorrenti che legacy.
- **Installation testing,** in cui ci si accerta che il sistema installato nell'ambiento di utilizzo soddisfi in pieno le richieste del cliente.

[Parte 2]

L'obiettivo del functional testing è trovare discrepanze tra il comportamento del sistema e quello atteso. La mancanza di sistemicità può portare a sprechi di tempo e risorse ed un idea per svolgere il lavoro in modo efficace è il **category partition.** L'idea è quella di suddividere l'input domain in partizioni e selezionare per ciascuna classe dei dati di test.

La specifica del test case presenta le informazioni in modo conciso ed uniforme, è facilmente modificabile e fornisce al tester un modo per controllare il volume del test stesso. È possibile inoltre generarli in modo automatico tramite tool.

Il metodo individua i parametri di ogni "funzione" del sistema e per ogni parametro individua categorie distinte; oltre ad esse si possono considerare anche oggetti dell'ambiente. Le categorie sono a loro volta suddivise in scelte, proprio come si applica la partizione in classi di equivalenza. Vengono inoltre individuati i vincoli che esistono tra le scelte, in modo da capire come si influenzino a vicenda. Si generano così i *test frames,* che consistono di combinazioni valide di scelte nelle categorie, e li si converte in *test data.*

[Segue una lunga serie di esempi]

SCRUM

Scrum è un processo agile che punta a massimizzare il valore di business in tempi brevi. Consente di ispezionare rapidamente software funzionante, con cicli di sviluppo iterativi di 2-4 settimane, detti **sprint**. Le priorità vengono definite dal business ed i team si auto-organizzano per realizzare le funzionalità più importanti.

Creato da Jeff Sutherland nel 1993 e formalizzato con Ken Schwaber nel 1995. Utilizzato da aziende come Microsoft, Google, Nokia, BBC e molte altre per progetti di varia natura.

Come caratteristiche principali troviamo:

- Team auto-organizzati e multifunzionali.
- Sviluppo del prodotto in una serie di sprint, dalla durata di 2-4 settimane, in cui il prodotto viene pianificato, scritto e testato.
- Lista delle attività, o product backlog, che raccoglie i requisiti.

Ruoli in Scrum:

- **Product Owner,** che definisce e prioritizza le funzionalità e le date di rilascio e accetta o rifiuta i risultati del lavoro.
- **Scrum Master,** che rappresenta il ruolo manageriale del progetto; si assicura che il team sia funzionale e produttivo, elimina eventuali ostacoli o interferenze esterne e facilita la cooperazione.
- **Team,** che comprendono 5-9 membri con competenze complementari, responsabili della consegna.

Eventi in Scrum:

- **Sprint Planning:** riunione in cui si pianifica il lavoro da completare nello sprint. È diviso in *sprint prioritization,* in cui si decifono gli sprint goals sulla ase dello sprint backlog, e *sprint planning,* in cui si decide come raggiungere gli obiettivi, si creano le task nello sprint backlog e si fornisce una stima del tempo necessario.
- **Daily Scrum:** incontri giornalieri di 15 minuti per monitorare il progresso e non per fare problem-solving. Ognuno risponde a 3 domande: "Cosa ho fatto ieri?", "Cosa farò oggi?" e "Ci sono ostacoli che impediscono il progresso?"
- **Sprint Review:** l'intero team presenta il lavoro svolto, tipicamente con una demo. Non c'è una preparazione pesante, il tono è informale ed il focus è sui risultati pratici.
- **Sprint Retrospective:** effettuata dopo ogni sprint e dalla durata di 15-30 minuti. È una valutazione di ciò che ha funzionato e cosa invece va migliorato, discutendo in particolare di cosa iniziare, interrompere o continuare a fare.

La **capacità del team** indica realisticamente quante ore il team può dedicare allo sprint e si calcola moltiplicando il numero di membri del team, le ore lavorative giornaliere e i giorni lavorativi dello sprint. Bisogna però tenere in considerazione il **focus factor,** un fattore compreso tra 0.6 e 0.8 da aggiungere al prodotto, che serve a rimuovere dal totale il tempo perso in riunioni, distrazioni e cali di concentrazione. Si usa un FF minore, ad esempio, quando si hanno team nuovi o inesperti oppure progetti complessi, mentre uno maggiore per team maturi oppure in sprint con obiettivi e backlog chiari e definiti.

Con **user stories** si indicano requisiti ad alto livello scritti dal punto di vista dell'utente: "Come [ruolo], voglio [funzionalità] così da ottenere [beneficio]". Le più grandi e complesse vengono dette *epics* e sono divise in attività più piccole.

Andando nello specifico, gli artefatti in Scrum sono:

- **Product backlog:** lista ordinata delle attività da completare e dei requisiti, in cui le priorità di ciascuno sono rivalutate all'inizio di ogni sprint.
- **Sprint backlog:** attività selezionate per lo sprint attuale con una stima del tempo restante di completamento. Ciascun team member sceglie autonomamente quale svolgere e può aggiungere, cancellare e modificare ciascun elemento.
- **Burndown chart:** traccia l’andamento del lavoro rimanente, evidenziando il progresso giornaliero rispetto all'impegno pianificato. È possibile che si vada incontro a svariati problemi:

- Sovraccarico iniziale: il team prende troppo lavoro e deve ridurre il carico durante lo sprint.
- Lavoro sottostimato: il team finisce in anticipo ma non ha attività aggiuntive pronte.
- Mancanza di aggiornamenti: il grafico non riflette il progresso reale per disorganizzazione o problemi di comunicazione.

Scrum si adatta molto bene ai principi di **scalabilità,** definendo il concetto di "Scrum of Scrums", ossia di team multipli che collaborano.

Note su pull-based development e design patterns

Il software ad oggi è sviluppato con un modello a cipolla: al centro ci sono i core developers, che sono coloro che si assumono rischi e responsabilità e che impostano una visione per il progetto; i co-developers propongono soluzioni ed assistono i core developer nell'implementazione; la community è l'insieme delle persone attorno al progetto e si occupano di proporre feature e use cases; gli utenti sono coloro che usano il software e, in contesti open-source, possono evidenziare problemi/difetti/vulnerabilità.
Si opera principalmente in un contesto distribuito e distinguiamo tra **contributors**, che scrivono il codice, e **integrators**, che hanno la possibilità di fare le merge. GitHub ha conquistato fama offrendo un modo semplice di lavorare in questo contesto.
Gli integrators si basano su diversi fattori per capire quali nuovi cambiamenti abbiano la priorità: criticità, urgenza, taglia ed età.

//

Il DAO (Data Access Object) è un design pattern strutturale utile a separare la logica di accesso ai dati dalla logica applicativa, in modo da semplificare la gestione dei dati e migliorare la manutenibilità del codice. Questo metodo di gestione della persistenza consiste nel fornire un'interfaccia astratta per l’accesso al database, così da operare sui dati senza esporne i dettagli.

//

Il Façade è un design pattern strutturale che fornisce un'interfaccia semplificata e unificata per l'interazione con un insieme di sottosistemi complessi. Questo pattern permette di mascherare la complessità interna del sistema, offrendo agli utenti o ai componenti esterni un unico punto di accesso che nasconde i dettagli implementativi.