Indice

1. Glossario

1.1. Stakeholder

Coloro che hanno interesse nel progetto. Includono almeno i clienti finali e coloro che lo stanno finanziando (committenti).

1.2. Elicitazione

Estrazione di informazioni dal modello mentale dello stakeholder, tradurre il modello mentale in modello di analisi. Si elicita intervistando il committente, guardando i competitor, facendo questionari, osservando l'ambiente, simulazioni, prototipizzazione, modellistica.

1.3. Valore

Produrre valore vuol dire interagire con un sistema software per giungere ad un obiettivo, determinato dal valore del prodotto.

1.4. Covarianza

La covarianza si verifica quando un sottotipo può essere utilizzato al posto del supertipo senza violare il sistema di tipi. In altre parole, un tipo più specifico può sostituire un tipo più generico.

1.5. Controvarianza

La controvarianza è l'opposto della covarianza: un tipo più generico può essere usato al posto di un tipo più specifico. È comune nei parametri di funzione, dove un parametro di tipo `T` può accettare un valore del suo supertipo.

2. Introduzione

Software engineering: costruzione di software multi-versione attraverso team di multi-persone. Per affrontare un sistema complesso si usa:

2.1. Astrazione

L'astrazione è il processo attraverso il quale si generalizzano delle idee in modo che siano applicabili a classi di idee. L'astrazione è utile per definire dei ragionamenti generali che non si applicano a singoli elementi, ma a classi generali. Il processo di sviluppo del software è una sequenza di attività ad alta astrazione a bassa astrazione

- Analisi
- Design
- Costruzione (codice sorgente/asset). Il codice è un modello che specifica ogni comportamento che il sistema deve avere.

2.2. Modelli

I modelli sono astrazioni o rappresentazioni della realtà in maniera più semplice rispetto a quella che è effettivamente. Sono rappresentazioni in grado di fornire risposte alle domande di un sistema. Il processo software è un raffinamento dei modelli (opposto dell'astrazione)

- Modello mentale (*problem space*): modelli che rispondono alla domanda "cosa deve fare il sistema?". In genere è incompleto.
- Modello di analisi (*problem space*): focalizzato sulla comprensione e sulla definizione dei requisiti del sistema. Si tratta di una rappresentazione astratta di *cosa* il sistema deve fare, senza entrare nei dettagli

implementativi. Usa modelli dei casi d'uso e diagrammi delle classi.
- Modello di design (*solution space*): traduce il modello di analisi in una rappresentazione più concreta, definendo *come* il sistema verrà implementato. Qui si passa da concetti astratti a soluzioni tecniche.
- Modello di costruzione (*solution space*): modello più dettagliato, completo e carico di informazioni.

Bisogna aggiungere informazioni a modelli astratti. Un modello deve *rispondere a delle domande*.

3. Modello di analisi

3.1. Obiettivi

- Capire cosa deve fare il sistema software realizzato
- Comunicare lo sviluppo al team di sviluppatori e agli azionisti
- Definire una serie di requisiti che possono essere validati dal software una volta costruito: *fornire gli strumenti di validazione*.

Qualsiasi forma di modello che consente di rispondere alle domande va bene. Il linguaggio naturale è ambiguo, va usato in maniera controllata.

3.2. Artefatti dei modelli di analisi

Servono per analizzare da punti di vista diversi i problemi dello sviluppo.
- Documenti delle richieste
- Glossario
- Modello dominante
- Specifiche supplementari

3.3. Requisiti

Servono per specificare quello che il software deve fare e i vincoli che deve rispettare.
Requirements engineering si occupare della gestione sistematica dei requisiti. La gestione dei requisiti richiede una serie di competenze
- Elicitazione
- Analisi
- Specificazione
- Validazione: definire i meccanismi per verificare che il sistema finito è coerente con i requisiti iniziali.

Il committente ha un modello mentale non coerente e spesso difficile da spiegare. I requisiti sono *indipendenti dall'implementazione*.

> I progetti software in genere falliscono quando la visione del committente non è allineata con quella dello sviluppatore. In genere ci si rende conto di questo solo quando il progetto è terminato.

3.3.1. Formalizzare i requisiti

Standard ISO che indica come vanno rappresentati i requisiti e i documenti che li raccolgono:

3.3.1.0.1. *Stakeholder requirements specification document*: StRS

Requisiti dello stakeholder: requisiti per un sistema che possono fornire i servizi richiesti dagli utenti e altri stakeholder.

3.3.1.1. *System & software requirements specification document*: SyRS/SRS

Requisiti del sistema: specificazioni da parte del fornitore delle caratteristiche, attributi e requisiti di performance del sistema affinché possa soddisfare i requisiti dello stakeholder.

3.3.2. Scrivere un requisito

Si usa un linguaggio il meno ambiguo possibile (linguaggio controllato). Deve essere steso con uno o più template. Ad esempio `[Condition] [Subject] [Action] [Object] [Constraint]` o `[Condition] [Action or Constraint] [Value]`.

3.3.2.1. Caratteristiche di un requisito

Un requisito deve essere:
- non ambiguo
- coinciso
- consistente

- completo: deve specificare una funzionalità del sistema nella sua interezza
- singolare: un qualunque requisito deve essere comprensibile senza l'utilizzo degli altri
- fattibile: deve rappresentare funzionalità che si è effettivamente in grado di realizzare
- tracciabile: si devono avere meccanismi che permettono di capire da dove emergono i requisiti
- verificabile: deve essere possibile definire un meccanismo che permette di affermare che un requisito è stato rispettato

3.3.2.2. Caratteristiche di un insieme di requisiti

I requisiti devono essere:

- completi
- consistenti
- convenienti: specifiche minime necessarie che garantiscono il funzionamento e le prestazioni richieste, rimanendo all'interno di un determinato budget.
- vincolati: anche se non è al momento completo, l'insieme dei requisiti sarà completato. Vincoli di budget, tecnici e temporali.

3.3.2.3. Metadati dei requisiti

- Identificativo
- Priorità dello stakeholder: quali sono le funzionalità più critiche, quale requisito ha più valore agli occhi dello stakeholder
- Dipendenze
- Rischi: le funzionalità da implementare sono più o meno facili da implementare nel contesto di tempo e risorse disponibili
- Sorgente: rende il rischio tracciabile
- Razionale
- Difficoltà: quanto è difficile realizzare software che implementi il richiesto
- Tipo

3.3.2.3.1. Tipi dei requisiti

- *Funzionali*: descrivono come ci aspettiamo che funzioni il sistema in relazione all'ambiente in quale opera. Lega l'azione, lo stato del sistema e la reazione. Indicano *cosa* fa il sistema.
- *Non funzionali*: proprietà misurabili/percettibili dei sistemi non direttamente collegate agli aspetti funzionali. Indicano *come* il sistema lo fa.

	Funzionale	**Non Funzionale**
Obiettivo	"Descrive cosa fa il prodotto"	"Descrive come funziona il prodotto"
Risultato finale	"Definisce le funzionalità del prodotto"	"Definisce le proprietà del prodotto"
Focus	"Si concentra sui requisiti dell'utente"	"Si concentra sulle aspettative dell'utente"
Origine	"Generalmente definito dall'utente"	"Generalmente definito dagli sviluppatori o altri esperti tecnici"
Testing	"Prima dei test non funzionali"	"Dopo i test funzionali"

3.3.3. Tassonomia FURPS+ per i requisiti

- *Functional*
- *Usability*: sforzo cognitivo di un utente per giungere ad un obiettivo
- *Reliability*
- *Performance*: tempi di attesa, uso delle risorse, accuratezza, disponibilità
- *Supportability*: mantenutibilità

- +: implementazione, interfaccia, packaging

3.3.4. Processo di analisi dei requisiti

Lo scopo è rendere la vista basata su requisiti dello stakeholder dei servizi richiesti in una vista tecnica di un prodotto richiesto che può fornire quei servizi. Se ci sono vincoli, vanno documentati all'interno del modello.

3.3.5. I Requisiti vanno memorizzati

I requisiti devono essere registrati in una forma adeguata per la gestione lungo tutto il ciclo di vita del progetto e oltre. Per progetti complessi, è consigliato l'uso di uno strumento di gestione dei requisiti, che permetta di tracciare i collegamenti tra di essi per evidenziare le relazioni.

3.4. Modelli

Un modello è una rappresentazione astratta della realtà, sono usati per catturare proprietà rilevanti di un sistema. Permette di catturare astrazioni e condividere conoscenze all'interno di un progetto (modelli *prescrittivi*, non descrittivi). I modelli possono essere espressi con linguaggi diversi da quello naturale. I linguaggi sono utilizzati per caratterizzare le entità del modello, le loro proprietà e relazioni.

3.5. Paradigma *Object Oriented*

Si caratterizza un sistema (software) non focalizzandosi sui singoli dati, ma cercando di definire la struttura in termine di entità autonome caratterizzate da uno stato e un comportamento. Uno stato è l'insieme di informazioni memorizzate nell'oggetto. Invocando metodi si attivano i comportamenti.

3.6. Principi *Object Oriented*

Astrazione: meccanismo attraverso il quale ci si focalizza sulle caratteristiche essenziali di un elemento, senza essere a conoscenza degli elementi strutturali che non ci interessano. Ci interessa solo l'interfaccia dell'oggetto, ma non come funzione internamente. *Incapsulamento*: non ci interessa come vengono ottenute le informazioni. Sono incapsulate nel codice..

Ereditarietà: il comportamento e lo stato possono essere specializzati (con l'*override* di parte dei comportamenti della classe base). Si riusa un prototipo o classe esistente. L'istanza di una sottoclasse è un'istanza della superclasse. Un oggetto può essere sia impiegato che persona. Un'oggetto può appartenere a più tipi diversi (polimorfismo). *Polimorfismo*: il comportamento degli oggetti dipende da che oggetti sono. Il polimorfismo è legato al concetto di ereditarietà. Un oggetto può essere di due o più tipi.

3.7. *Object Oriented Modelling* con UML

UML è un linguaggio di modellazione, che usa un approccio object oriented sia per l'analisi che per la progettazione/design. L'analisi si concentra sull problema, la progettazione sulla soluzione. UML è un linguaggio grafico semi-formale: gli artefatti sono grafici. Semi-formale vuol dire che ha una *sintassi e semantica ben definita* (nonostante la sua natura grafica). UML è stato soggetto a una serie di revisioni per ridurre l'ambiguità nel suo utilizzo. Le regole sintattiche definiscono come creare diagrammi validi. Le regole semantiche indicano come creare diagrammi con significato.

3.7.1. I 13 Diagrammi UML

Sono divisi in 2 famiglie

- Diagrammi strutturali
- Diagrammi comportamentali

Le frecce indicano l'ereditarietà

3.7.2. Le primitive UML

Sono elementi uguali indipendentemente dal tipo del modello. *Package*: raggruppano elementi e li da un namespace. *Annotazioni*: commenti che non hanno effetto sul modello *Relazioni*: linee che collegano due o più elementi di un modello

- Associazione
- Generalizzazione
- Dipendenza: se cambia l'oggetto sulla punta della freccia, può riflettersi sull'oggetto da dove parte la freccia.

- Realizzazione

La loro semantica cambia in base a come vengono usate. *Elementi*: forme geometriche *Stereotipi*: parole chiave che permettono di specializzare concetti esistenti. Permettono di estendere il vocabolario di UML per creare nuovi elementi del modello. Specificano la semantica di un elemento.

3.7.3. Object Constraint Language

OCL è un linguaggio dichiarativo utilizzato per specificare vincoli che si applicano a UML. Questi vincoli possono essere

- invarianti
- pre-condizioni
- post-condizioni

4. UML *use case* (casi d'uso)

4.1. Diagramma dei casi d'uso

È un diagramma comportamentale che serve per descrivere un'insieme di azioni (caso d'uso) che un certo sistema deve realizzare in collaborazione con uno o più entità esterne, chiamate attori. Ogni *use case* dovrebbe fornire un risultato osservabile e di valore agli attori o ad altri stakeholder. Ci saranno azioni che gli attori potranno operare sul sistema, e il sistema risponderà a queste azioni. I casi d'uso si limitano sempre ad interazioni osservabili. Il concetto di caso d'uso è indipendente dal diagramma dei casi d'uso. Il diagramma non è in grado da solo di catturare tutto quello che si vuole sapere riguardo ad un caso d'uso.

4.2. Uso dei diagrammi dei casi d'uso

- Usati per catturare i requisiti di un sistema: alternativa a documenti di requisiti software.
- La funzionalità offerta da un soggetto: cosa può fare il sistema.
- Requisiti che il soggetto specificato applica nel suo ambiente, come l'ambiente dovrebbe interagire on il soggetto affinché possa eseguire i suoi servizi.

L'elemento *attore* si può rappresentare con una qualunque icona che rappresenti gli attori del sistema. L'attore è un entità esterna al sistema, che ci interagisce scambiando dati. Con gli attori si vuole catturare le classi di utenti. Il *soggetto* è il sistema da analizzare o progettare, sul quale si applicano gli use case. Può essere un sistema fisico o software, ed è rappresentato da un rettangolo, il cui nome è in un angolo in alto. Gli attori sono al di fuori di questo rettangolo. Ogni *caso d'uso* è usato per descrivere una completa funzionalità che il sistema fornisce. Sono rappresentati con un'ellissi. La relazioni tra gli elementi del diagramma fanno parte della sintassi UML. Tra attori può esserci solo la relazione di *generalizzazione* (legato all'ereditarietà).

4.2.1. Generalizzazione

La relazione di generalizzazione serve a identificare che un elemento è più generico di un altro. Le frecce puntano verso l'elemento più generale, e si legge come "A è un B". A fa tutto quello che fa B, più qualcos'altro. Questa struttura è tipica della programmazione object-oriented. Al posto di fare *override* di un metodo, si fa *override* del caso d'uso.

> Non si possono riportare diagrammi che abbiano relazioni diverse da quella di generalizzazione tra attori.

4.2.2. Associazione

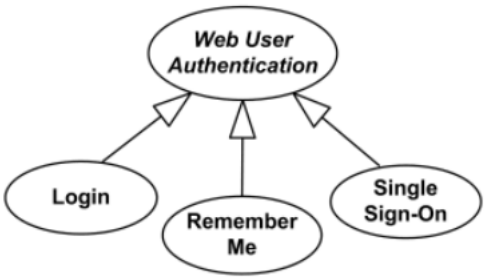
Un attore è *associato* a un caso d'uso solamente da una relazione di associazione. Quando il soggetto è tutto il sistema, viene lasciato implicito. Questa è l'unica relazione che può esserci tra attori e casi d'uso (relazione binaria). La rappresentazione grafica è un segmento.

4.3. Relazioni tra casi d'uso

4.3.1. Generalizzazione

Il caso d'uso è una sequenza di passi di interazione che sottintendono un obiettivo. Un attore interagisce con il sistema quando ha un

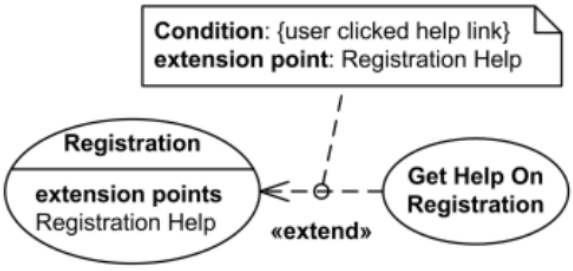
obiettivo. Il caso d'uso è sempre identificato da un obiettivo.



I nomi dei casi d'uso vengono definiti in base al loro obiettivo. La specializzazione (operazione inversa della generalizzazione) sono operazioni che hanno lo stesso obiettivo.

4.3.2. Estensione

Definisce un comportamento supplementare *opzionale* (e non obbligatorio come l'inclusione). Se un caso d'uso definisce il suo comportamento con passi di interazione, estendere un caso d'uso vuol dire aggiungere passi se è soddisfatta una certa condizione. In UML deve essere il caso base ad essere disposto ad estendere il proprio comportamento, definendo ***extension points***. La rappresentazione grafica di un estensione usa uno stereotipo *extends*.



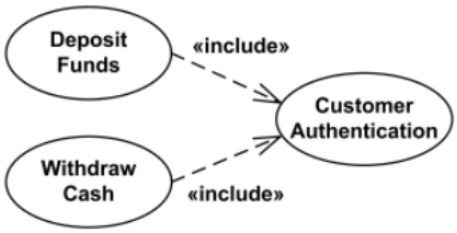
L'extension point viene definito all'interno del caso d'uso. La freccia va dal comportamento addizionale verso il comportamento che viene esteso.

4.3.3. Inclusione

Il comportamento addizionale avviene sempre, non c'è una condizione. L'inclusione si usa quando:

- ci sono parti comuni a più casi d'uso.
- si vuole semplificare un caso d'uso dividendolo in più parti.

Nella rappresentazione grafica, la freccia va dal comportamento base verso il caso d'uso incluso.



4.3.4. Come identificare i casi d'uso

- rappresentano la più piccola unità di attività che produce risultati tangibili
- sono definiti da obiettivi raggiungibili in una sessione d'uso, *immediatamente perseguibili*
- hanno al massimo una dozzina di passi
- quando un caso d'uso ha più attori con più obiettivi (ad esempio bisogna aspettare la reazione di un altro attore/utente), deve essere modellato con più casi d'uso. I casi d'uso non possono restare in attesa per un tempo indefinito. Il tempo può essere un attore.

Possono esserci dei comportamenti del sistema che il caso d'uso non è in grado di catturare. Per i processi automatizzati si usa il tempo come attore, rappresentato da una clessidra. Non ci interessa come gli attori agiscono al di fuori del sistema.

4.4. Modello dei casi d'uso

Il diagramma UML degli use case non è un modello degli use case. Il diagramma è un riassunto. I casi d'uso cominciano con l'attore che interagisce per primo. Non è mai il sistema che fa qualcosa per primo. Le *pre-condizioni* sono condizioni che devono essere vere affinché si possa realizzare il caso d'uso. Le *post-condizioni* sono i cambiamenti allo stato del sistema dopo che il caso d'uso è concluso.

4.5. Notazioni più semplici per i casi d'uso

- Id
- Attori
- Pre-condizioni
- Sequenza principale
- Sequenze alternative
- Post-condizioni

4.6. Scenari

Uno scenario è un esempio possibile del caso d'uso, un possibile modo in cui il caso d'uso ha luogo. Sono *istanze* dei casi d'uso. Si può verificare la correttezza dei casi d'uso generando dei test di accettazione. Un modello è soggetto a più raffinamenti iterativi.

- Stabilire l'obiettivo
- Capire la struttura
- Completare la prima storia
- Completare un numero sufficiente di storie
- Completare tutte le storie

4.7. Esercizio

> A blog is a web application presenting a collection of date-tagged messages (posts) on miscellaneous topics. Messages are posted by the blog owner who puts them online. The author can associate messages to one or more categories (expressed using keywords). Blog's visitors can comment messages; the comments, if approved by a moderator (usually the blog's owner), appear in a specific section under the original message.

1. Fare il diagramma UML
    1. Definire il sistema
    2. Identificare gli attori (blog owner, visitor, author, moderator). Eventuali attori omessi non possono essere "immaginati", si contatta il committente. Lo stesso fare per ambiguità: blog owner e author sono la stessa categoria di utente.
    3. Riscrivere il documento di specifica in base alle informazioni che emergono.
    4. Identificare i casi d'uso: si guardano gli obiettivi degli attori. Contattare il committente per risolvere eventuali ambiguità (il visitatore si deve autenticare per diventare autore).
2. Usare un template per descrivere un use case (esempio commento)
    - ID: UC3
    - Actors: Visitor
    - Precondizione: il visitatore si trova nella pagina di visualizzazione del messaggio.
    - Post-condizione: il messaggio è aggiunto alla lista di moderazione
    - Sequenza principale
        1. il visitatore seleziona l'opzione per inserire il commento (interazione osservabile)
        2. sistema mostra area per inserire il commento
        3. visitatore inserisce commento
        4. il sistema mostra messaggio di successo
    - Sequenze alternative: in questo caso non ci sono

5. *Software Process Model*

Processi: serie di attività coordinate che portano ad un obiettivo. Processo software: l'obiettivo è produrre, rilasciare, evolvere e mantenere. Con evoluzione si intende un software che viene arricchito con nuove funzionalità. Manutenzione: interventi di portata più piccola

5.1. Obiettivi del processo software

Pianificare e organizzare un progetto software con vincoli tipo: qualità, tempo, costi. Bisogna essere in grado di ottimizzare il processo e individuare i rischi. La gestione dei rischi consiste nel fargli emergere il prima possibile e vedere se sono mitigabili.

5.2. Ciclo di vita del software

1. Attività di specifica/analisi
2. Attività di progettazione e modello di design

3. Implementazione
4. Validazione (testing)
5. Evoluzione: piccole modifiche sul software

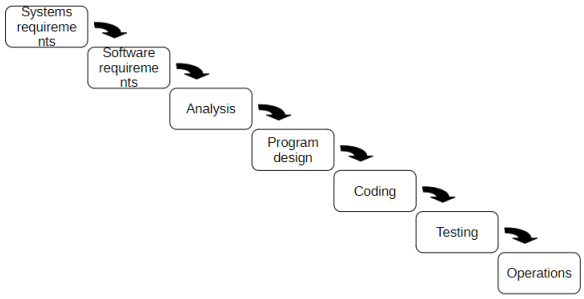
5.3. Artefatti (deliverables)

Dato che il software è intangibile, per compensare l'assenza di visibilità, si producono artefatti:

- documenti di design
- report
- incontri
- sondaggi

5.4. Modello a cascata

Descrive un processo in cui le diverse fasi che operano ai diversi livelli di astrazione sono originariamente sequenziali.



- Coding: si mettono gli algoritmi dentro ai metodi
- Testing: per vedere se soddisfa i requisiti
- Operations: si aggiorna l'applicazione web,...

Dopo aver applicato il modello a cascata può anche esserci il bisogno di risalire (da testing si può ritornare al coding). In un progetto vero e proprio è più comune risalire piuttosto che scendere. *Vantaggi*:

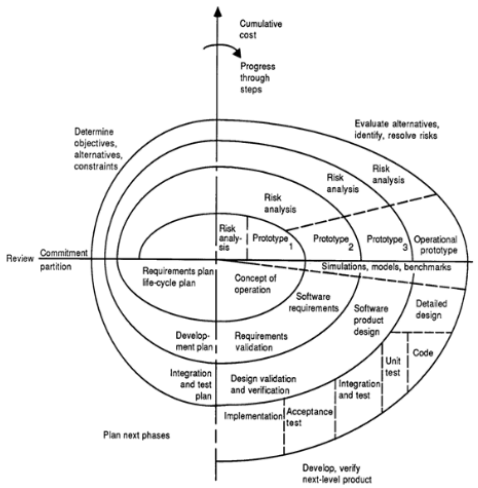
- Facile da capire
- Incita buone pratiche
- Artefatti identificabili
- Documentazione comprensiva

Svantaggi:

- Irrealistico (assume che le specifiche iniziali non cambino mai)
- Consegna solo del prodotto finito (alto rischio di disallineamento di idee tra committente e sviluppatore)
- Risk management inefficiente
- Difficile da gestire i cambiamenti
- Costo di manutenzione alto

5.5. Modello a spirale

Modello ormai non più utilizzato, basato su una famiglia di processi. Il processo di generazione del modello dipende dai rischi. Ha un approccio ciclico dato che è un modello iterativo. Non si fa tutto il modello di analisi all'inizio, ma solo una parte. Si prendono parte degli elementi per realizzare il modello: in questo modo si ha un prototipo.



Ogni ciclo inizia con obiettivi, alternative e limiti. Il passo successivo è determinato sulla base dei rischi rimanenti. Ogni ciclo termina con una revisione dagli stakeholder.

Vantaggi:

- riflette la natura iterativa dello sviluppo software
- buona visibilità
- comprensione dei rischi

Svantaggi:

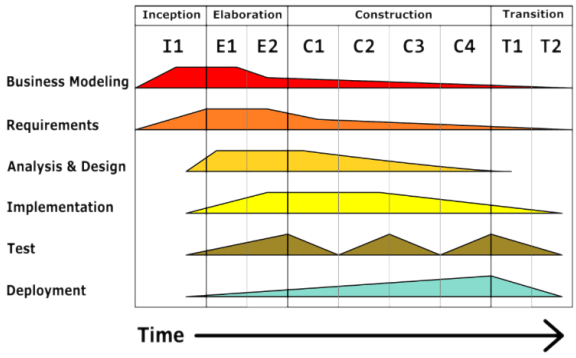
- analisi dei rischi non è insignificante
- modello complicato, le priorità dei rischi potrebbero portare a una consegna tarda
- alto sforzo di manutenzione

Lo sviluppo iterativo incrementale richiede programmare e testare un sistema parziale molto presto, e in genere ipotizza che lo sviluppo inizi prima che tutti i requisiti siano definiti in dettaglio. Il feedback degli stakeholder è usato per migliorare la specifiche già esistenti.

5.6. *Unified Process* (UP)

Il processo unificato è un framework iterativo e incrementale. UP combina buone pratiche già esistenti (ciclo di vita iterativo e sviluppo basato su cicli) in una descrizione di processo coesa e ben documentata, guidata dai casi d'uso. Essere architettura centrico significa che le scelte architetturali vanno fatte il prima possibile. Il processo unificato divide il progetto in quattro fasi:

- inizio: architetture candidate, identificano i rischi e termina con il Lifecycle Objective Milestone.
- elaborazione: valutazione rischi, validazione dell'architettura. Si implementa un architettura eseguibile. Termina con un piano per la fase di costruzione
- costruzione: implementazione feature di sistema
- transizione: rilasciare il sistema, raccogliere feedback

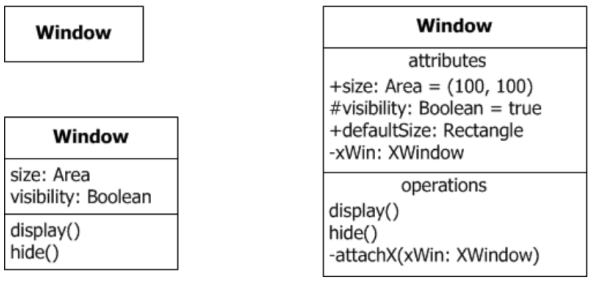


Il tempo investito in queste attività non è uguale per tutte. Ogni fase è composta da iterazioni, ognuna delle quali termina con una release da far vedere al committente. Le iterazioni che terminano una fase devono aver realizzato degli artefatti predefiniti (milestone). Se non si raggiunge il milestone bisogna aggiungere un'altra iterazione a quella fase.

6. Classi UML

6.1. Classe

La classe è un tipo, gli oggetti rappresentati sono riconducibili a un tipo, e ogni tipo ha una classe. Sono rappresentati con un rettangolo e può essere diviso in compartimenti da linee orizzontali (ad esempio attributi e operazioni/funzioni).



Proprietà: modificatori di visibilità, seguiti da un nome, il tipo e una molteplicità *Operazioni* (funzioni/metodi): modificatori di visibilità seguiti da un nome, dei parametri e un tipo opzionale.
Modificatori di visibilità:

- `+`: public
- `-`: private
- `#`: protected
- `~`: package

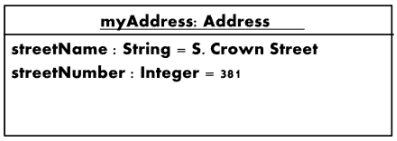
Molteplicità: indica se l'attributo deve esserci per forza, quante volte deve esserci (cardinalità). È Il range ammesso dei valori, definito da un range di valori.

- `1` indica che deve esserci esattamente un elemento
- `1..n` indica che devono esserci da `1` a `n` elementi
- `0..1` l'elemento è opzionale
- `1..*` deve esserci almeno una volta
- `*` ripetuto opzionale

6.2. Istanze delle classi

Rappresentano degli stati che devono essere conformi alle classi.

Graficamente anche gli oggetti si rappresentano con rettangoli, il cui nome è sottolineato, seguito da : e la classe di cui è un istanza.



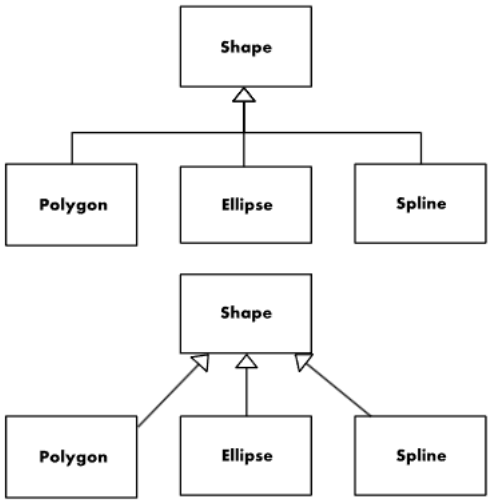
Le due istanze sono correlate, indicate tramite un segmento, dove agli endpoint sono presenti i due classificatori.

6.3. Relazioni

6.3.1. Generalizzazione

Si riconduce al concetto di ereditarietà, si legge come "A (elemento specifico) è un tipo di B (elemento generale)". C'è una gerarchia di ereditarietà. In UML è prevista l'ereditarietà multipla.

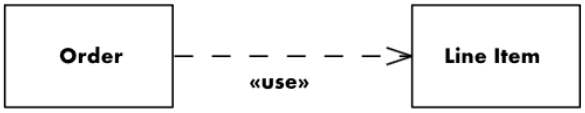
La generalizzazione è rappresentata graficamente con una freccia vuota.



6.3.2. Dipendenza

La dipendenza indica una relazione di fornitore-utilizzatore, e l'utilizzatore non è in grado di realizzare la sua semantica senza il fornitore. Se uno cambia, l'altro potrebbe essere costretto anch'esso a cambiare. Si dice "A dipende da B". Il cambiamento del fornitore può riflettersi anche nell'utilizzatore.

La dipendenza è rappresentata graficamente da una linea tratteggiata, che può essere stereotipata.

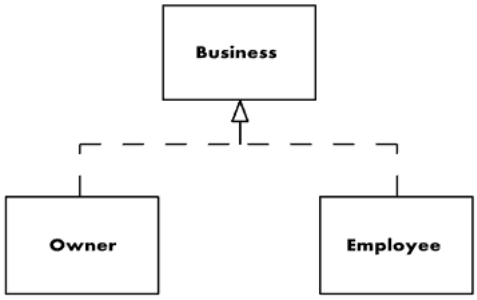


6.3.3. Realizzazioni

Una realizzazione è un tipo di dipendenza.

La realizzazione è rappresentata graficamente da una linea tratteggiata che termina con una freccia vuota.

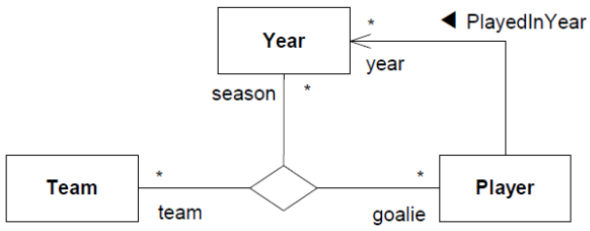
L'elemento che sta realizzando (realizing element) deve fornire un'implementazione concreta del comportamento o delle specifiche definite dall'elemento che viene realizzato (realized element).



6.3.4. Associazione

Un'associazione dichiara che possono esserci collegamenti tra più istanze dei tipi associati. Un link è una coppia dove con valore per ogni termine dell'associazione. Il legame non deve essere solo logico, ma c'è anche un meccanismo operativo per risalire agli altri termini.

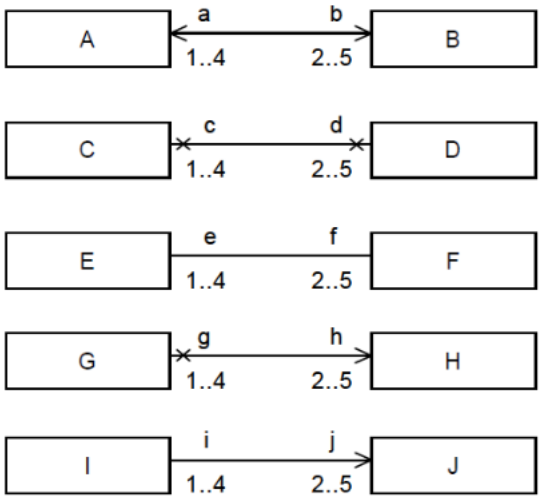
> Le associazioni sono rappresentate con frecce che hanno label.



A un giocatore non corrisponde un solo anno (molteplicità/numerosità).

6.3.4.1. Navigazione

Un padre può avere * figli, un figlio ha 1 padre (può essere omesso).

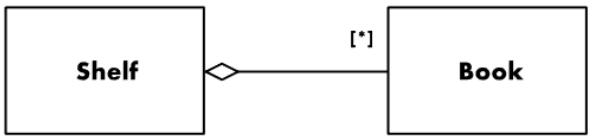


Se si mostrano tutte le frecce e croci: navigazione e assenza di navigazione sono esplicite. Se non ci sono frecce e croci la navigazione è oscurata. Se non ci sono croci non si può distinguere tra navigazione in entrambi i versi e situazioni dove non c'è navigazione. La croce si usa quando la relazione si può inferire ma non è immediata.

6.3.5. Aggregazione

> L'aggregazione è rappresentata graficamente con una freccia a rombo.

La classe al termine della freccia indica è dipendente da un numero di classi alla sorgente della relazione. La sorgente ha molteplicità.



6.3.6. Composizione

Un file esiste solo se c'è una cartella in cui metterlo. Il file non esiste senza folder. Il libro può esistere anche senza libreria. L'oggetto composto ha responsabilità per l'esistenza e contenimento di oggetti composti.

> La composizione è rappresentata graficamente con una freccia a rombo piena.

6.4. Classi Astratte

Le classi astratte non possono essere istanziate. Le sue istanze sono istanze di specializzazioni.

> Le classi astratte si rappresentano graficamente in corsivo o con l'annotazione `{abstract}`.

6.5. Interfacce

Le interfacce si modellano come le classi, ma si usa lo stereotipo `interface`. Le interfacce definiscono un obbligo da definire nella fase di implementazione. Dichiarano servizi coerenti che sono implementati dalle classi che li implementano.

7. Modello di analisi e modello di dominio

Nell'*Object-Oriented Analysis* (OOA), il modello di dominio (o modello a oggetti) rappresenta i concetti nel dominio del problema, le loro caratteristiche e le loro correlazioni. È un dizionario visivo per il dominio del problema.

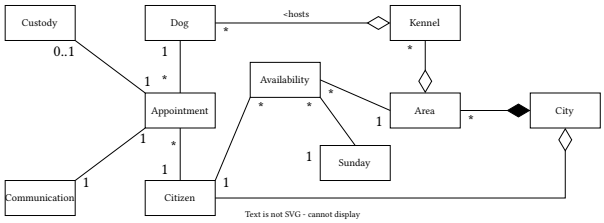
7.1. Disegnare il modello del dominio

- L'artefatto risultante è in genere un diagramma di classi UML
- Per identificare le classi si fa un analisi dei nomi e frasi soggetto che sono parte della descrizione del problema
- Si analizzano i verbi per identificare responsabilità e collaborazioni
- Si ripetono questi passi per raffinare il modello

7.2. Esercizio

> The City of Duckburg activates an initiative that allows dogs hosted in the kennels of the district to enjoy a walk in the Sunday afternoons. Citizens interested in taking custody of the animals register their availability specifying the Sunday and the area, among the many that make up the Municipality, in which they are willing to collect a dog (there are several kennels located in various areas).
> Given this availability the kennels assign dogs to citizens creating appointments which are then communicated to the volunteers.

- Sunday
- Kennel
- Dog
- Citizen
- Area
- City
- Custody
- Appointment
- Communication



8. Diagrammi di attività UML

I diagrammi di attività sono diagrammi comportamentali usati per rappresentare i comportamenti di classi, casi d'uso, interfacce, componenti e comportamenti di una classe (operazioni, algoritmi). I diagrammi di attività sono formalizzati per essere simili a semantiche della rete di Petri. Le semantiche del sistema sono descritte in termini di transazioni tra segni.

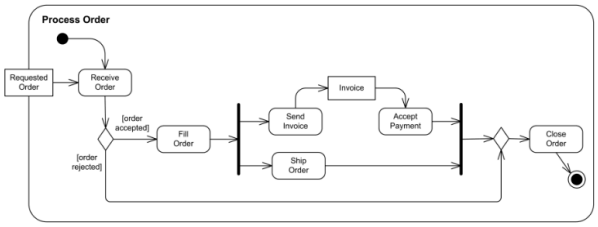
8.1. Elementi di un *activity diagram*

- Attività
- Nodi dell'attività
  - Azioni
  - Oggetti
  - Controllo (per mostrare scelte)
- frecce dell'attività

> Graficamente, c'è un pallino nero che indica il punto di partenza. Dei rombi rappresentano le scelte possibili (flussi). I

rettangoli con angoli arrotondati rappresentano il punto in cui il processo è in esecuzione (flusso di esecuzione).

A differenza dei diagrammi di flusso, i diagrammi di attività possono avere flussi di esecuzione concorrenti.



Elementi dei nodi di attività sono

8.1.1. Azioni

Elementi di comportamento atomico: non si sa cosa sta succedendo durante l'azione, non viene catturato il loro comportamento interno. Possono essere di vari tipi:

- funzioni primitive
- chiamate ad operazioni
- invio o ricezione di segnali
- manipolazione di oggetti
- invocazione di comportamenti

Sono rappresentate graficamente come rettangoli arrotondati. Un azione può avere insiemi di attività entranti o uscenti.

Un token attiva un azione, dopodiché l'azione può avere luogo. Si possono specificare anche altre condizioni

8.1.2. Archi

Definiscono le dipendenze causali con un meccanismo di attraversamento. Si possono specificare nomi e vincoli per rendere gli archi traversabili. Le condizioni sono specificate tra parentesi quadre.

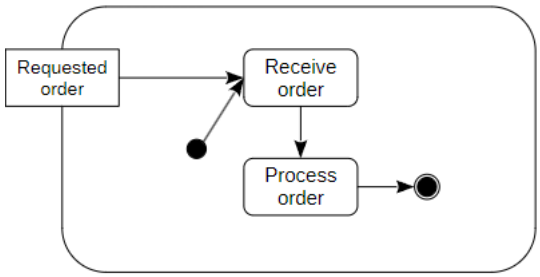
8.1.3. Oggetti

Un nodo oggetto è un istanza di un classificatore.

Si rappresentano come rettangoli oppure si usano pin per indicare che oggetti vengono dati in output e presi in input da un'altra azione.

8.1.4. Nodo parametrico

Indicano che un processo opera in funzione di un parametro di attivazione. Quando il parametro è disponibile, il processo avrà luogo.

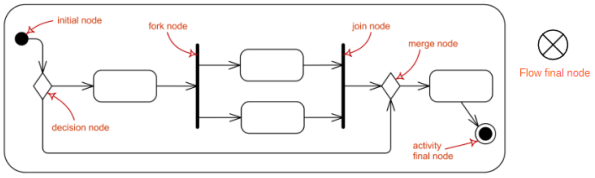


Si tratta di una forma speciale di oggetto.

8.1.5. Connettori

8.1.6. Controllo

Un nodo di controllo è usato per coordinare i flussi tra altri nodi. Include il nodo iniziale, finale, di decisione, di unione, di divisione, di unione.



Il nodo decisionale può fare uso di predicati/ guardie negli archi di uscita per mandare il flusso verso archi attraversabili. Il *flow final* termina tutte le attività in esecuzione.

8.1.6.1. Competizione dei token

I token non vengono spinti verso delle azioni, ma sono le azioni ad accettare i token. Quando un azione si completa, il token è rilasciato ed è offerto da altre azioni. È il token a scegliere quale azione vuole compiere in un dato momento del flusso di esecuzione.

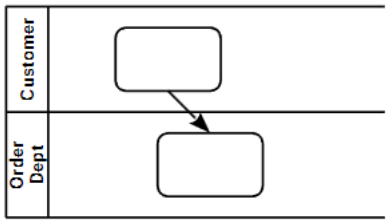
I nodi di decisione non fanno azioni, altrimenti sarebbero nodi di decisione. Sono passivi, non attivi. Le azioni sono solo nei nodi di azione.

8.1.6.2. Regione di espansione

È un nodo che prende in input collezioni, opera su ogni elemento delle collezione e produce elementi da inviare alle collezioni. Vengono processati dei dati fino a quando una condizione non è più vera. Se si usa lo stereotipo «`parallel`» vuol dire che i token saranno tutti inviati allo stesso momento e processati assieme.

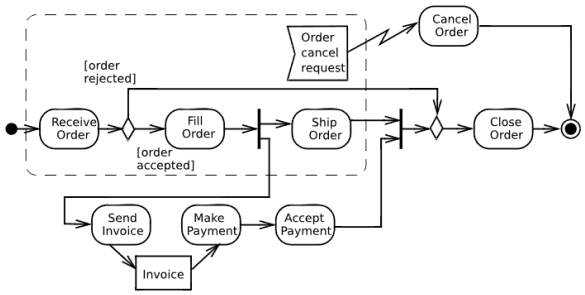
8.1.7. Partizione di attività

Un *activity partition* è un gruppo di attività per azioni che hanno una caratteristica comune. Le partizioni forniscono una visualizzazione ristretta dei comportamenti chiamati in attività e spesso corrispondono a unità organizzazionali. Ogni corsia è chiamata *swim lane*.



8.1.8. Regioni interrompibili e archi di interruzione

Una regione interrompibili è un tipo di gruppo di attività che ha un meccanismo per distruggere tutti i token e terminare tutti i processi in una sezione dell'attività delimitata dalla regione. Quando il token è accettato da un tipo speciale di arco viene lascia la regione e tutti gli altri token dentro ad essa vengono terminati.



8.1.9. Azioni correlate ad eventi

Le *event actions* sono usate per modellare interazioni che avvengono al di fuori dell'attività corrente. Produrre eventi è asincrono, consumare eventi è sincrono (bloccante). Le azioni ad evento sono:

- invio segnale
- accetta segnare
- ripetizione periodica di eventi

9. Diagrammi di interazione

9.1. Diagrammi di sequenza

I diagrammi di sequenza sono il tipo più comune di diagrammi di interazione. Si concentrano sullo scambio di messaggi tra lifeline. I diagrammi di sequenza descrivono le interazioni concentrandosi sulla sequenza dei messaggi scambiati e sulle corrispondenti specifiche di occorrenza sulle lifeline.

9.2. Lifeline

Una lifeline è un elemento nominato che rappresenta un singolo partecipante nell'interazione. Le parti e caratteristiche strutturali possono avere molteplicità maggiore di 1, le lifeline rappresentano solo una delle entità che sta interagendo.

9.3. Messaggio

Un messaggio rappresenta o una chiamata ad un operazione e l'inizio dell'esecuzione o l'invio e ricezione di un segnale. Può essere

- chiamata sincrona/asincrona
- segnale asincrono
- risposta
- creazione

- cancellazione

9.3.1. Chiamata sincrona

Rappresentano operazioni di chiamata-invio messaggi e sospendono l'esecuzione quando aspettano una risposta.

> Chiamate sincrone sono rappresentate con una freccia piena.

9.3.2. Chiamata asincrona

Invia il messaggio e procede subito senza aspettare il valore di ritorno.

> Chiamate asincrone sono rappresentate da una freccia aperta.

9.3.3. Risposta

> La risposta a una chiamata si rappresenta con una linea tratteggiata con la punta aperta.

9.3.4. Creazione

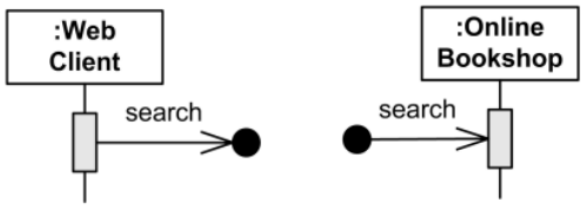
Un messaggio di creazione è inviato a una lifeline per creare stesso. In genere si invia un messaggio di creazione a un oggetto non-esistente per creare se stesso.

> Si rappresenta con una freccia tratteggiata aperta. Stereotipato con *destroy*.

9.3.5. Cancellazione

Un messaggio di cancellazione è inviato per terminare un'altra lifeline. La lifeline in genere termina con una croce a forma di X sotto.

9.4. Lost & Founds



9.5. *Gate*

Un gate è la fine di un messaggio, un punto di connessione per trasmettere un messaggio dentro o fuori (in/out) da un frammento di interazione.

9.6. *Interaction Fragment*

Un frammento di interazione è un elemento nominato che rappresenta l'unità di interazione più generica. Ogni frammento di interazione è concettualmente come un interazione indipendente. Non c'è una notazione generale per un frammento di interazione. Le sue sottoclassi definiscono le loro notazioni. Esempi di frammenti di interazione sono:

- Occorrenza
- Esecuzione
- Invariante di stato
- Frammento combinato
- Uso di interazione

9.6.0.1. Occorrenza

Un'occorrenza è un frammento di interazione che rappresenta un momento nel tempo (evento) all'inizio o alla fine di un messaggio o esecuzione.

9.6.0.2. Esecuzione

Un'esecuzione è un frammento di interazione che rappresenta un periodo nel ciclo di vita del partecipante dove sta

- eseguendo un'unità di comportamento o azione dentro la lifeline
- inviando un segnale a un altro partecipante
- aspetta una risposta da un altro partecipante

9.6.0.3. Frammento Combinato

Un frammento combinato è un frammento di interazione che definisce una combinazione dei frammenti di interazione. Un frammento combinato è definito da un operatore dell'interazione e operandi dell'interazione corrispondenti. Con i frammenti combinati l'utente può descrivere un numero di tracce in maniera compatta (simile a un ciclo `for`/`while`).

Rappresentati o con `loop(5,10)` o `[size<0]`.

9.6.0.4. Uso di interazione

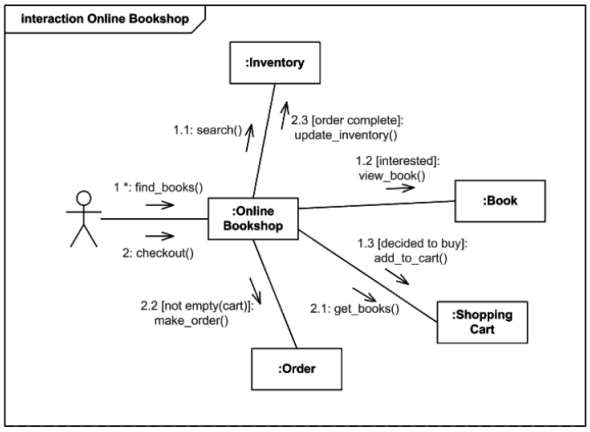
Un'uso di interazione è un frammento di interazione che permette di usare/chiamare un'altra interazione. Sequenze grandi e complesse possono essere semplificate con usi di interazione. In genere si riusa un'interazione tra le altre.

9.6.1. Consigli sui *sequence diagrams*

- Non generalizzare troppo le sequenze
- Quando si usano i diagrammi di sequenza per l'analisi
  - un caso d'uso può essere descritto da più di un diagramma di sequenza
  - il tipo di azione del messaggio può essere deciso al tempo di azione
  - nessun messaggio tra lifeline appartiene ad elementi del sistema

9.7. Diagrammi di comunicazione

Un diagramma di comunicazione mostra le interazione tra lifeline con regole generiche. Il diagramma di comunicazione è meno espressivo del diagramma di sequenza. Si assume che i messaggi sono ricevuti nello stesso ordine che sono generati.



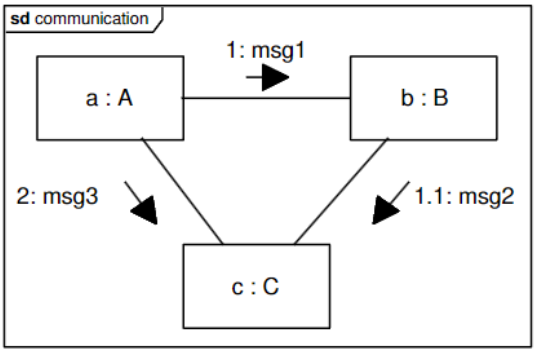
9.7.1. Messaggi

La notazione per i messaggi nei diagrammi di comunicazione seguono le stesse regole usate nei diagrammi di sequenza. Ogni messaggio ha una notazione di sequenza: *sequence expression* (numero che indica l'ordine)

```
Sequence-expression ::= sequence-term
'.' . . . ':' message-name

Sequence-term ::= [integer[name]]
[recurrence]
```

I termini di sequenza sono usati per rappresentare il nesting di messaggi in un interazione.



`1.1` Vuol dire che il messaggio avviene dopo 1, ma prima che abbia generato risposta. 2 vuol dire che il messaggio ha già ricevuto la risposta

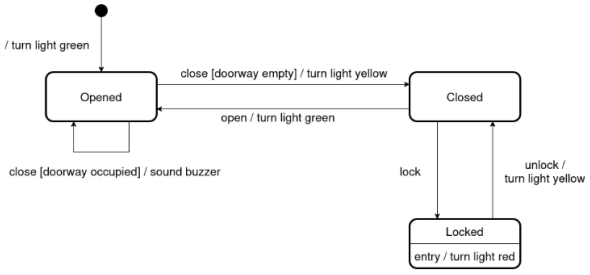
9.7.2. Concorrenza e ricorrenza

`Sequence-term ::= [integer[name]] [recurrence]` Messaggi che variano solo per nome sono considerati concorrenti. `recurrence ::= branch | loop branch ::= '[' guard ']'` Le guardie specificano le condizioni affinché il messaggio possa avvenire.

10. *State machine*

10.1. State machine comportamentale

Una state machine *comportamentale* descrive un comportamento definito da un evento di un sistema o una sua parte, tramite un traversamento di un grafo di vertici (stati) connessi da transizioni. Uno *stato* è la risposta alla domanda "in che stato sta?". Gli stati influenzano il comportamento o le interazioni di un soggetto.



Dalle transizioni si riesce a capire il comportamento dell'soggetto osservato. In ogni istante analizzato il soggetto sarà in uno degli stati. Le transizioni sono atomiche, istantanee. Non ci sono stati intermedi. Le *etichette* indicano la ragione per cui una transizione ha luogo. Un *evento* è un accadimento che influenza il comportamento del soggetto. Tutte le transizioni sono determinate da eventi. Per passare da uno stato ad un altro deve succedere qualcosa che influenza il comportamento del soggetto.

> I rettangoli arrotondati sono stati. Lo stato non è un'azione. Si chiamano stati perché il soggetto ci sta.

`<evento[<guardia>]/<azione>` su una freccia che va dallo stato vecchio verso quello nuovo. Gli eventi non sono azioni. Le *azioni vengono fatte dal soggetto*, gli *eventi sono esterni*. Non ci sono azioni fatte da qualcosa che non sia il soggetto. Si usano le guardie `[<guardia>]` per indicare se una transizione è attraversabile.

10.1.1. Stato

Lo stato definisce una condizione invariante di un sistema. Due stati sono lo stesso stato se definiscono lo stesso comportamento: dagli stessi stimoli si hanno le stesse risposte. Il comportamento è una cosa aggiunta allo stato. Uno stato può essere:

- semplice
- composito: contiene una o più regioni, stati in queste regioni sono dette sottostati.
- submachine

Gli stati possono essere associati a comportamenti:

- entrata (nello stato)
- uscita (dallo stato) `exit/<behavior name>`
- `doActivity`: il suo comportamento viene eseguito in contemporanea con altri comportamenti associati a stati.

> Uno stato è rappresentato da un rettangolo arrotondato, e può essere diviso in comportamenti divisi da una linea orizzontale.

I compartimenti possono essere:

- nome
- comportamenti interni
- transizioni interne

10.1.1.1. Pseudostati

Nodi che non sono stati e non possono catturare il comportamento del sistema nel tempo (ad esempio lo pseudostato iniziale *start*).

> Se il token non può fermarsi in quel punto allora è uno pseudostato. Il token può fermarsi solo negli stati.

Il token non può fermarsi in questi punti (e sono dunque dei pseudostati):

- *initial*: rappresenta il punto di partenza di una regione. È la sorgente di al massimo una transizione non associata a un trigger o una guardia.
- *join*: è il target di due o più transizione che si originano da vertici in diverse regioni ortogonali. Eseguono una funzione di sincronizzazione dove tutte le transizioni in arrivo devono essere completate prima che l'esecuzione possa continuare.
- *fork*: divide la transizione in arrivo in una o più transizioni che terminano in vertici che appartengono a regioni ortogonali diversi.
- *junction*: unisce o separa transizioni.
- *choice*: tipo di junction usato per realizzare branching condizionale in modo dinamico. Una guardia di tipo *else* può essere usata su una transizione predefinita che viene selezionata quando tutte le altre transizioni uscenti sono valutate false.
- *entry point*: il punto di entrata per uno stato composito o submachine.
- *exit point*: il punto di uscita per uno stato composito o di submachine.
- *terminate*: quando si entra questo pesudostato l'esecuzione termina subito.
- ***deep history***: reimpostano tutta la configurazione dello stato.
- *shallow history*: reimpostano solo il sottostato principale.

10.1.1.2. Storia degli stati

La *state history* è usata per tenere traccia dell'avanzamento dello stato di una regione. La regione può essere resettata da una transizione locale che si connette a uno pseudostato di tipo *history*.

11. Transizioni

Le transizioni sono passaggi atomici da uno stato all'altro. Le transizioni sono sempre attivati da eventi, *anche quando non sono espliciti*. Le transizioni possono avere guardie. Se le guardie sono false, l'evento è scartato e la transizione non avviene. `{<trigger>}* ['['<guard>']'] [/ <behavior-expr>]` Nel corso della sua esecuzione una transizione può essere

- raggiunta
- traversata
- completata

Transizioni composte: un trigger può causare l'attraversamento di una parte aciclica della macchina a stati senza che venga elaborato alcun altro evento. Ciò significa che l'elaborazione di un singolo evento può attivare più transizioni, attraversando diversi (pseudo)stati.

11.0.1. Run-to-completion

Una state machine viene creata, e poi inizializzata, eseguendo la transizione composta iniziale. Successivamente entra in un *wait point*. Quando gli eventi sono inviati, i trigger vengono valutati, e se almeno una transizione può essere eseguita, una nuova transizione composta viene eseguita, e si raggiunge un nuovo punto di attesa. Questo ciclo si ripete finché la state machine completa il suo comportamento o finché è asincronamente terminato da un agente esterno. Questo modello è chiamato *run to completion* (RTC). Le occorrenze degli eventi sono rilevate, inviate e processate dall'esecuzione della state machine una alla volta. I *completion event* sono prioritari, mentre gli altri sono inviati in ordine arbitrario. Un singolo evento può attivare più transizioni. Se queste transizioni hanno un'intersezione non vuota tra i loro stati di uscita, allora sono *in conflitto*. Le priorità delle transizioni in conflitto sono determinate dalla loro posizione relativa nella gerarchia degli stati. Il modello RTC semplifica la gestione della concorrenza nelle state machine. Quando la macchina è in uno stato non ben definito, questa non è reattiva. Eventi che dovrebbero occorrere non sono processati subito vengono messi nella coda degli eventi.

11.0.2. Eventi

Un evento è un occorrenza osservabile nell'ambiente del soggetto.

> Un evento implicito si genera quando termina il comportamento interno di un soggetto.

evento → guardia vera → transizione stato → azione

Gli eventi comandano le transizioni. Un evento non ha durata e può avere parametri. Lo stato finale determina uno stato da cui non si esce, ha solo archi in ingresso e nessuno in uscita. Quando si transita nello stato finale termina il ciclo di vita.

11.0.3. Regioni

Uno stato o transazione può essere organizzato in regioni. Le regioni *ortogonali distinte* indicano che i processi interni sono concorrenti. Le transizioni sono solitamente indicate da trigger. Quando questi vengono soddisfatti termina l'esecuzione dei processi concorrenti.

11.1. State machine di protocollo

Una state machine *di protocollo* descrive il ciclo di vita o le sequenze di interazioni valide (protocolli) per un sistema (classificatore).

12. Design *Object Oriented*

Ora che sappiamo cosa dovrebbe fare il sistema, dobbiamo pensare a come progettare un sistema che lo fa. Nella parte di progettazione potremmo lavorare su diversi tipi di prodotti dalla fase di analisi che possono essere diversi tra loro.

12.1. Metodo super semplificato ispirato a UP (Unified Process)

Viene studiato perché è la struttura più usata e si aspetta molti parametri che abbiamo in output dalla fase di analisi. Come UP dipende dagli use case. Come prima cosa bisogna trovare o creare elementi all'interno della logica del business o spazio del dominio responsabili del supporto a tutte le interfacce all'interno del modello. Gradualmente si aggiungono nuovi elementi e operazioni all diagramma di design della classe

13. Qualità del software e principi object oriented

13.1. Obiettivo del design

L'obiettivo delle attività associate al design è produrre sistemi software di alta qualità

13.1.1. Qualità del sistema software

- Esterne: caratteristiche visibili e apprezzabili dall'utente
  - correttezza
  - usabilità
  - efficienza
  - affidabilità
  - integrità: non vengono alterati i dati
  - adattabilità: riesce a girare su sistemi operativi diversi
  - accuratezza
  - robustezza: produce valore anche di fronte a fallimenti totali o parziali
- Interne: qualità che sono relative a come è organizzato internamente il software, non visibili all'utente e non apprezzabili da esso
  - manutenibilità: quando posso farci manutenzione sopra. Maggiore il livello, più è semplice fare manutenzione.
  - flessibilità: disponibilità ad aggiunta di nuove caratteristiche
  - portabilità: quanto è portabile su sistemi diversi
  - riusabilità: quanto è semplice prendere elementi di una soluzione passata in una nuova
  - testabilità: quanto è facile testare in maniera autonoma il codice
  - comprensibilità

13.2. Il software non è *write-once*

I costi associati all'evoluzione del software sono molto alti e possono coprire il 50-90% dei costi di produzione del software. Una bassa qualità

equivale a costi maggiori in caso di aggiornamenti. Ci sono 3 modelli per capire il livello di qualità del codice:

13.2.1. Software product quality (ISO 25010)

Definisce le caratteristiche misurabili internamente o esternamente

- adeguatezza funzionale
- affidabilità
- efficienza delle performance
- operabilità
- sicurezza
- compatibilità
- manutenibilità
- portabilità

13.2.2. Data quality model

A volte si usa un framework per valutare la qualità, ma in genere si vuole un software riutilizzabile e progettato per cambiare.

13.2.3. Quality in use model

Sono qualità non funzionali legate alla percezione dell'utente

- efficienza
- soddisfazione
- sicurezza
- usabilità

13.2.4. Principi *Object Oriented*

Si progettano sistemi object oriented affinché possiamo correttamente identificare gli oggetti, e applicavi i principi della programmazione object oriented per produrre software di alta qualità.
Concetti di base:

- Astrazione: focus sulle caratteristiche essenziali
- Incapsulamento: nascondere i dettagli nelle classi
- Eredità: comportamento e stato possono essere specializzati
- Polimorfismo: il comportamento dipende su cosa bisogna essere

13.3. Design *smells*

Sono indicatori che ci potrebbe potenzialmente essere un errore o che qualcosa non è stato progettato nel modo giusto.

- Rigidità: tendenza del software a resistere al cambiamento, anche semplice. Un software è rigido se un singolo cambiamento provoca cambiamenti a cascata. Più moduli devono essere cambiati, più il software è rigido.
- Fragilità: tendenza di un programma a rompersi in più punti quando viene fatto un singolo cambiamento.
- Immobilità: un design è immobile quando contiene parti che sarebbero state utili in altri sistemi, ma lo sforzo e il rischio coinvolti nel separarle è troppo grande.
- Viscosità: la facilità o difficoltà di apportare modifiche a un sistema software in modo da preservarne il design originale. Un ambiente è viscoso quando è lento e inefficiente.
- Complessità inutile: un eccessiva flessibilità del codice può portare a una struttura complessa. Questo capita spesso quando si cerca di anticipare i cambiamenti al codice.
- Ripetizione inutile: uso del copia-incolla invece di richiamare il metodo direttamente.
- Opacità: la tendenza di un modulo ad essere di difficile comprensione. Codice che cambia col tempo diventa sempre più difficile da leggere: bisogna fare sforzi per renderlo continuamente comprensibile e minimizzare l'opacità.

La sorgente della maggior parte degli smell è riconducibile al management delle dipendenze. Le dipendenze sono tutti i percorsi attraverso i quali possono diffondersi cambiamenti e sono alla base di tutti i problemi nello sviluppo software a qualsiasi scala. Meno dipendenze ci sono più è probabile che le modifiche restino isolate.

13.4. SOLID

Le scelte fatte aiutano a migliorare la qualità del software solo se seguono certi principi. Alcuni principi che possono aiutare a ridurre gli smell:

13.4.1. Single responsibility principle (SRP)

I metodi all'interno di una classe dovrebbero occuparsi di cose correlate tra loro. Deve esserci un unico asse di cambiamento all'interno di una classe. Una classe deve avere solo una responsabilità, e solo un motivo per dover essere cambiata.

13.4.2. Open-closed principle (OCP)

Una classe deve essere aperta per le estensioni, ma chiusa per le modifiche. Se non si rispetta OCP, si rischia di ottenere un sistema rigido. Non si può modificare il funzionamento di un metodo esistente. Si definisce un nuovo comportamento come override del metodo vecchio, lasciandolo così invariato. *Refactoring*: tecnica disciplinata per ristrutturare una soluzione modificandone la struttura interna, lasciando il suo comportamento esterno invariato. *Si può violare OCP solo per fare refactoring.*

Se funziona, non metterci le mani. Tutti gli utilizzatori di quella classe stanno facendo affidamento su quel metodo, quindi se lo cambi si rischiano di causare problemi. Quando il progetto è grande, diventa difficile risalire alla modifica che ha causato i problemi.

13.4.3. Liskov substitution principle (LSP)

LSP è il principio su cui si basa il polimorfismo. Afferma che quello che cerchiamo è un meccanismo di sostituibilità.

> Se per ogni oggetto o1 di tipo S c'è un oggetto o2 di tipo T, tale che per tutti i programmi P definiti in termini di T, il comportamento di P è invariato quando o1 è sostituito con o2, allora S è un sottotipo di T.

```
class Rectangle{
  private double shortSide, longside;
  public double calculateArea(){
    shortSide*longSide;
  }
  public void setLongSide(double newValue)
{
    this.longSide = newValue;
  }
  public void setShortSide(double
newValue){
    this.shortSide = newValue;
  }
}

class Square extends Rectangle{
  // garantisce che la lunghezza dei lati
del quadrato sia sempre la stessa
  @Override
  public void setLongSide(double newValue)
{
    setSides(newValue);
  }
  @Override
  public void setShortSide(double
newValue){
    setSides(newValue);
  }

  private void setSides(double newValue){
    this.shortSide = newValue;
    this.longSide = newValue;
  }

}
void myMethod(Rectangle rectangle){
  r.setLongSide(7);
  r.setShortSide(6);
  r.calculateArea();
}
```

Se si passa un rettangolo ci si aspetta che l'area sia 7×6. Per rispettare LSP:

- struttura: deve esserci covarianza nel tipo di ritorno dei metodi, controvarianza dei tipi dei parametri del metodo e non ci sono nuove eccezioni.
- comportamento:
  - le precondizioni nono possono essere rinforzate nel sottotipo
  - le postcondizioni e invarianti non possono essere indebolite nel sottotipo
  - Il vincolo di storia deve essere rispettato: i sottotipi possono modificare gli elementi di stato ereditati solo in conformità con i meccanismi consentiti presenti nel supertipo.

Senza `@override`, Liskov non succede.

13.4.4. Interface segregation principle (ISP)

La dipendenza da una classe a un'altra deve dipendere sull'interfaccia il più piccola possibile.

> Non si dovrebbe dipendere da metodi non utilizzati.

Se non si rispetta questo principio si rischia di avere dipendenze non necessarie e implementazioni di interfacce che creano complessità inutile e potenzialmente violano LSP.

13.4.5. Dependency inversion principle (DIP)

I moduli di alto livello non dovrebbero dipendere da moduli di basso livello.

> Le classi di alto livello sono quelle che implementano le feature utilizzate direttamente dal cliente finale. Le funzionalità di basso livello sono quelle non utilizzate o viste dall'utente (memorizzazione, trasmissione su rete,...). Le classi di basso livello sono *a disposizione* di quelle di alto livello, non sono attivate dalle interazione diretta con l'utente. Le classi di alto livello sfruttano le funzionalità delle classi di basso livello (*layer architecture*).
> Si ha una dipendenza tra la classe di alto livello e quella di basso livello.

La dipendenza è una strada di diffusione del cambiamento. Il problema non è nella dipendenza in sé, il fatto che certe dipendenze siano più o meno critiche dipende da come le classi sono messe in relazione tra di loro. Una dipendenza è di qualità migliore dell'altra perché è causa più spesso di cambiamenti (viene modificato più spesso). Non verranno mai diffusi i cambiamenti di dipendenze mai modificate. Il basso livello è per sua natura più volatile, più soggetto a modifiche dell'alto livello. Entrambi dovrebbero dipendere da astrazioni. I dettagli non dovrebbero dipendere dalle astrazioni, le astrazioni devono dipendere dai dettagli. La *layered architecture* classica crea una serie di dipendenze critiche.

Deve esserci un livello intermedio di astrazione, più i moduli sono astratti più sono vicini alla logica del dominio e meno volatili sono.

> I moduli di alto e basso livello dovrebbero dipendere da delle interfacce.

In genere è meglio usare le interfacce, non i tipi concreti. `double sum(ArrayList<double> list) {...}` funziona solo con le `ArrayList`. Se si usa l'interfaccia List `double sum(List<double> list){...}`, si possono usare sia `ArrayList` che `LinkedList`.

14. General Responsibility Assignment Software Patterns (GRASP)

Tutte le scelte di applicare i principi SOLID influenzano la qualità del prodotto finale. GRASP si concentra sulla progettazione object oriented dove si arriva alla soluzione guidati dall'idea di responsabilità. Con responsabilità si intende un contratto o obbligazione di un classificatore. Si hanno due responsabilità:

Fare	**Conoscere**
Creare un oggetto o fare un calcolo	Conoscere i dati incapsulati
Iniziare interazioni con altri oggetti dividendo la responsabilità iniziale con altre più piccole	Conoscere cose che può derivare o calcolare
Controllare e coordinare le attività in altri oggetti	Conoscere cose che può derivare o calcolare

14.1. Responsibility Driven Design (RDD)

Le responsabilità sono assegnate alle classi degli oggetti durante il design degli oggetti.

> Assegnare le responsabilità in modo meccanico o troppo rigido ("slavish"), senza una vera comprensione o riflessione critica, può portare a un'architettura software disordinata, comunemente definita "big ball of mud".

GRASP può essere usato per fare RDD garantendo che i principi SOLID sono usati. GRASP è un aiuto per imparare OO design con le responsabilità. Aiuta a capire il design essenziale degli oggetti e applicare ragionamenti metodici, razionali ed esplicativi.

14.2. Pattern

> Ogni pattern descrive un problema ricorrente in un ambiente, e poi descrive il nucleo della soluzione a quel problema in modo che la soluzione possa essere riutilizzata, senza mai farla nello stesso modo due volte. I pattern forniscono soluzioni pratiche a tutti i problemi che sorgono durante il design.

Un design pattern è una soluzione riutilizzabile per un problema ricorrente in un certo contesto. I GRASP pattern sono:

14.2.1. Creator

Problema: chi è responsabile di creare un'istanza dell'oggetto? *Soluzione*: si assegna alla classe B la responsabilità di creare un'istanza della classe A se una delle seguenti condizioni è vera

- B contiene o aggrega compositamente A
- B registra A
- B richiama spesso metodi di A
- B ha i dati per l'inizializzazione di A che saranno passati ad A quando viene creato.

  Allora B è un esperto nella creazione di A (B ha tutte le informazioni per creare A).

14.2.2. Information Expert

Problema: come si assegnano responsabilità a oggetti affinché i nostri sistemi sono più facili da capire, mantenere ed estendere, e le nostre scelte ci permettono di riutilizzare più componenti in applicazioni future. *Soluzione*: si assegna la responsabilità all'*information expert*, la classe che ha l'informazione necessaria per soddisfare la responsabilità.

Può anche essere che non ci sia un esperto, se nessuna classe contiene una certa informazione.

14.2.3. Controller

System Operation: invocazione di un metodo che corrisponde all'attivazione di qualcosa nell'interfaccia dell'utente. *Problema*: quale oggetto riceve e coordina una *system operation*? *Soluzione*: si assegna la responsabilità a una classe che rappresenta

- tutto il sistema
- un oggetto radice
- un dispositivo su cui sta venendo eseguito il software
- un sottosistema grande

Rappresenta un scenario di caso d'uso del sistema dove avviene il system event. Si usa a stessa classe controller per tutti gli eventi nello stesso use case.

14.2.4. Low Coupling

Problema: come possiamo progettare un sistema che:

- abbia poche dipendenze,
- sia facile da modificare,
- e favorisca il riuso del codice?

Soluzione: si assegnano le responsabilità affinché l'accoppiamento tra le parti rimanga basso.

14.2.5. High Cohesion

Problema: come si mantengono gli oggetti concentrati, comprensibili e gestibili e supportati per il *low coupling*? *Soluzione*: si assegna la

responsabilità affinché la coesione rimanga alta. Non si aggiunge il metodo a classi esistenti, ma si crea una nuova classe.

14.2.6. Pure Fabrication

Problema: cosa si fa quando non si vuole violare high cohesion e low coupling, ma soluzioni offerte dal Expert non sono corrette? *Soluzione*: si assegna un insieme di responsabilità altamente coeso a una classe artificiale di convenienza che non rappresenta un concetto del dominio del problema.

14.2.7. Indirection

Problema: dove si assegna responsabilità per evitare coupling diretto tra due o più oggetti? *Soluzione*: si assegna la responsabilità a un oggetto intermedio per mediare tra altri componenti affinché non vengano accoppiati direttamente.

14.2.8. Polimorfismo

Problema: la variazione condizionale causata da statement del control-flow produce codice difficile da leggere. *Soluzione*: si usano alternative basate sul tipo. Quando alternative correlate sono funzione del tipo di oggetto su cui si sta operando, allora queste variazioni comportamentali non dovrebbero essere fatte attraverso l'`if`. Un `if` sbagliato è quello dove la variazione comportamentale dipende dall'oggetto su cui si sta operando `if(A instanceof B) {...}` . Il polimorfismo rende subito evidente se ci sono errori nella gerarchia di classi.

14.2.9. Protected Variations (PV)

Problema: come limitiamo la portata dei cambiamenti? Come si evitano cambiamenti a catena? *Soluzione*: si identificano punti dove si ipotizza ci saranno cambiamenti, si assegnano le responsabilità per creare un'interfaccia stabile attorno a loro.

> OCP è simile a Protected Variation. LSP formalizza il principio di protezione contro variazioni in diverse implementazioni di un interfaccia, o di sottoclassi che estendono una superclasse.

Si rischia di creare sempre sottoclassi che non servono (*defensive programming*). Avere elementi inutilizzati è una spesa. Se si ha un oggetto A e si deve aggiungere un oggetto B, si fa solo refactoring aggiungendo una classe base. Poi si aggiunge la nuova sottoclasse B.

14.3. Dipendenze

Una dipendenza è una relazione fornitore-utilizzatore tra due elementi, dove la modifica del fornitore influenza il cliente. Una dipendenza implica che le semantiche del cliente non sono complete senza i fornitori. Tipi di dipendenze:

- dipendenza d'uso: cambiamenti nel nome o parametri dei metodi possono influenzare il fornitore.
- istanziazione: come i metodi, ma con il costruttore.
- parametro di una classe: il cliente riceve un parametro e lo passa ad un altra.

Se A dipende da B, e B cambia, è anche necessario cambiare A (dipendenza sintattica). Dipendenze semantiche: il nome e i parametri di un metodo rimangono gli stessi, ma cambia l'oggetto restituito. Se si usano correttamente i principi *open-close*, questi problemi non si hanno. Non tutte le dipendenze sono uguali. Se si è dipendenti da una classe che non cambia mai, il costo della dipendenza sarà 0.

> Bisogna avere dipendenze di buona qualità affinché il cambiamento sia minimo. Bisogna avere le dipendenze negli elementi non volatili.

Si hanno prove empiriche che gli elementi astratti tendono ad essere più stabili di quelli concreti. È meglio avere più dipendenze buone che poche dipendenze cattive.

15. Agile Software Development

15.1. Il problema

Spesso ci si trova a spendere molto per overhead (attività non correlate alla realizzazione del prodotto, ma lo facilitano). Si rischia che queste attività di overhead non sempre siano utili quanto ci si aspettava. Investire troppo in overhead non è l'idea migliore. Non si cerca di migliorare il processo per migliorare il prodotto. Il tempo speso per scrivere documentazione estensiva, negoziare contatti, fare controllo rischi, potrebbe essere perso quando il piano non è chiaro fin dall'inizio. Vogliamo che non solo il software sia flessibile, ma anche il modo per realizzarlo.

15.2. Manifesto dell'Agile Software Development

Più valorizzati	Meno valorizzati
gli individui e le interazioni	processi e strumenti
software funzionante	documentazione comprensiva
collaborazione con i clienti	negoziare i contratti
rispondere ai cambiamenti	seguire un piano

Scritto da individui del mondo della consulenza che si sono poi spostati alla progettazione software.

15.3. Principi

Agile non è un processo o un metodo di design, è una collezione di buone prassi guidate da una serie di principi ispirati da valori. I principi:

1. soddisfare il committente con consegna del software continua e di valore
2. accettare i cambiamenti nei requisiti
3. consegnare software frequentemente
4. collaborazione giornaliera tra sviluppatori e business
5. costruire i progetti attorno a individui motivati
6. fornire loro un ambiente e l'aiuto di cui hanno bisogno, fidandosi di loro per fare il lavoro
7. tramettere informazioni con conversazioni faccia a faccia
8. software funzionante è la misura principale di progresso
9. processi agile promuovono un ritmo di sviluppo sostenibile nel tempo
10. gli sponsor e sviluppatori devono tenere un passo costante a tempo indefinito
11. mirare all'eccellenza tecnica e buon design migliora l'agilità
12. semplicità: massimizzare la quantità di lavoro non fatto
13. le migliori architetture, requisiti e design emergono da gruppi che si organizzano da soli
14. a intervalli regolari, il team riflette su come essere più efficienti, e di seguito sistema il suo comportamento

Nella pratica non tutti i meccanismi funzionano.

15.4. Metodi

Sono stati proposti dei metodi ispirati ad Agile:

- Modelling agile
- Agile Unified Process
- Crystal clear
- Extreme programming
- Scrum

15.5. Pratiche

- Refactoring
- Small release cycles
- Continuous integration
- Coding standard
- Collective ownership
- Planning game
- Whole team
- Daily meetings
- Test-Driven Design

- Code and design reviews
- Pair programming
- Document late
- Use of design patterns

15.6. Code review

Quando viene fatta una modifica al codice di un prodotto già sul mercato, questo deve passare dei test prima di fare il commit. Si fanno delle revisioni prima di aggiungere modifiche al codebase principale. La code review viene fatta da tutti i membri del team.

> Le modifiche fatte vengono controllate da chi ha scritto il codice, e da qualcuno che ne fa la review.

Uno dei vantaggi della code review è anche l'aumento della visione di ciò che avviene all'interno del processo e la condivisione della conoscenza.

15.6.1. Pair programming

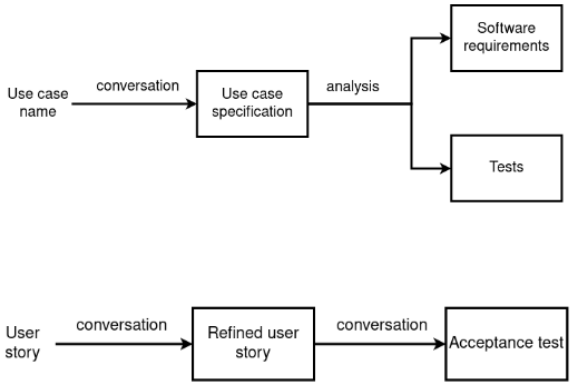
Alternativa a code review, il codice viene scritto a coppie. Ci si alterna tra pilota e navigatore: il pilota è il programma che sta alla tastiera, mentre il navigatore è quello che sta facendo la code review.

15.6.2. Test-driven design

15.6.3. User stories

Si usano le user stories per rappresentare i requisiti. Le user stories sono delle descrizioni di sistema funzionali, che vengono processate e guidano l'implementazione e vengono raffinate con test di accettazione che validano l'implementazione. Si usano dei template: `as a <role>, I want <goal> so that <benefit>`

- ruolo: classe di utente, attore
- obiettivo



15.6.3.1. Acceptance test

Se il software passa il test, è automaticamente accettato. Seguono il template *given-when-then.*

> *Given* my bank account is in credit, and I made no withdrawals recently, When I *attempt* to withdraw an amount less than my card's limit, *then* the withdrawal should complete without errors or warnings.

15.6.3.2. Le storie sono volatili

Le storie non devono sopravvivere al loro processing. Persistono sono gli acceptance test associati ad esse. Gli test dicono cosa realizzare e servono anche da validazione.

15.6.3.3. Storie, epiche, temi

- Le storie che descrivono le feature di alto livello sono raccolte presto ma non sono molto specifiche e sono raffinate quando il progetto prosegue.
- Le epiche sono storie grandi: in genere serve più di un'iterazione per svilupparle. Poi sono divise in storie più piccole quando ci si avvicina alla fase di sviluppo.
- I temi sono collezioni di storie correlate.

15.6.3.4. INVEST

Un insieme di criteri per valutare la qualità di una storia

- Independent: le storie non devono dipendere l'una dall'altra

- Negotiable: le storie non sono contratti, sono il risultato da una negoziazione e possono essere ri-negoziate in qualsiasi momento.
- Valuable: le storie devono fornire valore
- Estimable: il team deve essere in grado di stimare il livello di complessità e la quantità di lavoro richiesta per l'analisi della storia. Un alto grado di incertezza è un buon indicatore che la storia non è abbastanza precisa.
- Small: una storia deve essere realizzata in una iterazione. Alla fine dell'iterazione devono essere considerate finite.
- Testable: una storia è finita solo quando le feature corrispondenti passano i test di accettazione.

15.6.4. Extreme programming

Extreme programming è un metodo di sviluppo software basato su AGILE, le sue caratteristiche sono:

15.6.4.1. Test driven development

Si inizia a scrivere il codice scrivendo i test, per prevenire i bug.

15.6.4.2. Whole team

15.6.4.3. Continuous process

- Integrazione continua
- Miglioramento del design
- Aggiornamenti piccoli

L'indicatore dello stato del progetto è la funzionalità del software. Va periodicamente rivisitato il debito tecnico all'interno del progetto.

15.6.4.4. Shared Understanding

- Standard di programmazione
- Codice di proprietà comune
- Design semplice
- Metafora del sistema

Non deve esserci il codice di qualcuno.

15.6.4.5. Planning Game

È il processo di pianificazione in *extreme programming*, basato sulle storie. Si fa prima di ogni iterazione, ed è composto da due parti:

- pianificazione delle release (include i clienti)
- pianificazione dell'iterazione (solo tra sviluppatori)

I clienti ordinano le storie in base al loro valore, gli sviluppatori in base al rischio. Si scelgono le storie che saranno finite nella prossima release. Alla fine del tempo prefissato devono essere passati tutti i test.

15.7. Evoluzione

Sviluppo ed evoluzione fanno parte del ciclo di vita del software. Approcci agile, la conoscenza è condivisa implicitamente e si produce poca documentazione

16. Software Testing

16.1. Validazione e verifica

Il testing è l'attività più rilevante tra quelle di validazione e verifica, usate per controllare se il software creato è corretto rispetto alle specifiche. Si può avere software validato ma non verificato (piace all'utente, ma fa le cose sbagliate, o viceversa). La validazione consiste nel catturare i comportamenti attesi. La verifica viene fatta attraverso altri test. Il testing dice quando il sistema è sbagliato, non quando è giusto. Più test si fanno, più è probabile rilevare la presenza di un qualche tipo di errore logico. Maggiore è il numero di test, maggiore sarà il livello di confidenza nella correttezza del sistema. Il livello di confidenza è pagato con la scrittura dei test. I test non sono gratuiti, devono essere mantenuti e fatti cambiare con le classi.

> Il testing nel software è diverso da quello dell'ingegneria.

16.2. Costi

I meccanismi di testing devono essere introdotti il prima e il più spesso possibile. Più un problema persiste nel codice, più sarà difficile correggerlo.

16.3. Lessico

- Difetto (bug): il risultato di un errore logico di qualche genere.
- Fallimento: il malfunzionamento è visible
- Problema (issue): descrizione del fallimento
- Test case: il risultato atteso dell'esecuzione di un programma

16.4. Livelli di testing

- Unit → classe/metodo: maggiori dei test di integrazione, poco costosi sia da scrivere che da eseguire
- Integrazione → gruppo di moduli
- End to end → l'intero sistema: bisogna mandare in esecuzione l'intera applicazione. Non sempre sono automatici

Salendo la piramide diminuiscono le cose da testare, ma fare il testing diventa più complicato e costoso.

16.5. Test statico/dinamico

16.5.1. Analisi Statica

Si analizza il codice per trovare bug. Si basa su metodi formali

- model checking
- data-flow analysis
- abstract interpretation
- symbolic execution

Si trasformano i progetti in modelli matematici e si controlla se soddisfano certe proprietà. Le revisioni da parte di persone sono una forma di analisi statica.

> Il codice non viene eseguito.

Si usano pattern di bug per valutare la qualità del codice. In genere sono regole per controllare se si stanno rispettando stili di programmazione sicuri.

16.5.2. Analisi Dinamica

Si esegue il codice per trovare bug. Il test è progettato con un approccio *blackbox* o *whitebox*.

- Blackbox: ci interessa solo il risultato senza guardare il codice che c'è dentro.
  - *Boundary value analysis*: Si trovano i punti di discontinuità nei valori di input e si testano.
  - *equivalence partitioning*: Si testa un valore casuale appartenente a ciascuno degli intervalli attorno ai punti di discontinuità e agli intervalli della validità del dominio.
  - decision table testing
  - all-pairs testing
  - state transition tables
- Whitebox: si usa la struttura del codice per definire i test. I metodi di test includono:
  - Flusso dei dati
  - Flusso di controllo: si identifica un test per controllare
    - copertura del codice
    - copertura del branch
    - copertura del path

Pro e contro di black e white

	Contro	**Pro**
white	complesso	conoscenza del codice è acquisita quando si creano i test case, copertura maggiore
black	copertura sconosciuta	i tester non devono essere sviluppatori, si avvicina di più ai requisiti

16.6. Testare i test

Per valutare la qualità del insieme dei test si testano mutazioni del codice.

- si creano mutazioni del codice
- si eseguono test sulle mutazioni
- se un test passa c'è un problema.

16.6.1. Struttura di un test

Tutti i test sono strutturati nella stessa maniera

1. Si mette il sistema da testare (system under testing - SUT) nello stato desiderato
2. Si interagisce con il SUT
3. Si verifica che i risultati della interazioni siano quelli aspettati

AAA: arrange-act-assert

16.7. Unit testing

Si testa una singola funzione: il subject è molto piccolo. Si controlla che il codice rispetti le aspettative quando è scritto per la prima volta, e anche dopo averlo modificato (*regression testing*). I SUT devono essere isolati. Il test set di ogni unit deve avere casi indipendenti. I test eseguiti devono essere indipendenti l'uno dall'altro. Test interdipendenti non possono essere eseguiti in parallelo. Si posso fare test su

- valore di ritorno
- stato (di un oggetto), post-condizione di un'operazione
- collaborazione con altri oggetti (test del comportamento)

Per isolare il codice dalla dipendenze d'uso si usano delle copie. Forniscono la stessa interfaccia ma con codice diverso. *Dependency injection* facilita l'isolamento.

17. Design Patterns

Si parte dal problema e vengono offerte una o più soluzioni. Un pattern è tale se la soluzione è valida. Un pattern si identifica in base alle soluzioni trovate in passato. I pattern sono in genere raggruppati in un sistema dove possono essere richiamati l'uno con l'altro (catalogo in informatica). I cataloghi hanno pattern che presentano soluzioni a gruppi di problemi. C'era un momento in cui tutte le cose rappresentate dovevano essere pattern. Al giorno d'oggi rappresentare tutto sotto forma di pattern non è efficace. Trattiamo pattern presi da *Design Patterns: Elements of Reusable Object-Oriented Software*.

17.1. Documentazione

Ogni pattern ha una struttura uniforme

- Nome e classificazione: nome unico e descrittivo
- Obiettivo: ragione per cui usare il pattern
- Alias
- Motivazioni: problema e contesto in cui usare questo pattern
- Applicabilità: situazioni in cui il pattern è applicabile
- Struttura: rappresentazione in forma grafica che precede UML
- Partecipanti: classi e oggetti utilizzati e il loro ruolo nel design
- Collaborazione: descrizione di come classi e oggetti interagiscono gli uni con gli altri
- Conseguenze: risultati causati dall'utilizzo del pattern
- Implementazione: descrizione di un implementazione di un pattern
- Codice di esempio (in c++)
- Utilizzi conosciuti: esempi di utilizzi concreti del pattern
- Pattern correlati

17.2. Fini

- Creazionale: processo di istanziazione (Abstract Factory, Builder, Factory Method, Prototype, Singleton)
- Comportamentale: algoritmi e responsabilità (Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy)
- Strutturale: come le classi e gli oggetti sono composti per fare strutture più grandi (Chain of responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template method, Visitor)

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Facade Flyw eight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

17.2.1. Template Method

Problema: come posso condividere un comportamento parzialmente definito in una gerarchia di ereditarietà? Con il template method si definisce lo scheletro di un algoritmo in un operazione, delegando alcuni passi a sottoclassi. Si lascia cambiare certi passi dell'algoritmo alle sottoclassi, senza cambiare la sua struttura. Si spostano i comportamenti più comuni in cima all'albero di ereditarietà anche quando parzialmente specificato. Nuove implementazioni della superclasse astratte non creano problemi con i clienti: le dipendenze sono dirette ad elementi più stabili. Inoltre, viene favorita l'aderenza a OCP.

17.2.2. Strategy

Problema: come posso separare un oggetto da parte del suo comportamento e cambiarlo a runtime, più comportamenti diversi attivabili? Con strategy si definisce una famiglia di algoritmi, si incapsula ognuno di essi e li si rende intercambiabili. Strategy consente all'algoritmo di variare indipendentemente dai clienti che la usano. Strategy favorisce l'implementazione di OCP e obbedisce PV. Favorisce la composizione rispetto all'ereditarietà.

17.2.3. State/strategy

Problema: come posso cambiare il comportamento di un oggetto in base allo stato interno dell'oggetto stesso? L'oggetto deve comportarsi come se cambiasse la classe. La soluzione è strategy. La differenza rispetto a strategy è che la chiamata di cambiamento di comportamento viene da fuori. In state è l'oggetto stesso che decide come comportarsi rispetto al suo stato.

17.3. Privilegiare la composizione rispetto all'ereditarietà

Le due tecniche principali per condividere e riutilizzare codice sono ereditarietà e composizione. Si favorisce la composizione degli oggetti rispetto all'ereditarietà tra classi.

17.3.1. Problemi dell'ereditarietà

L'ereditarietà risolve due problemi

- polimorfismo tramite sottotipaggio
- condivisione del comportamento

Nella maggior parte dei linguaggi di programmazione, queste due soluzioni in genere si ostacolano a vicenda.

> Non si rispetta OCP e PV. La soluzione è non usare superclassi, ma la composizione.

```
// Interface for movement behavior
interface GoBehavior {
 void goTo(String destination);
}

// Interface for control behavior
interface ControlBehavior {
 void notifyDriver(String message);
}

// Concrete movement behavior for driving
class DriveBehavior implements GoBehavior
{
 public void goTo(String destination) {
 System.out.println("Driving to " +
destination);
 }
}

// Concrete movement behavior for flying
class FlyingBehavior implements GoBehavior
{
 public void goTo(String destination) {
 System.out.println("Flying to " +
destination);
 }
}
```

```
// Concrete control behavior for remote
control
class RemoteBehavior implements
ControlBehavior {
 public void notifyDriver(String message)
{
 System.out.println("Sending remote
message: " + message);
 }
}

// Concrete control behavior for human
driver
class HumanBehavior implements
ControlBehavior {
 public void notifyDriver(String message)
{
 System.out.println("Notifying human
driver: " + message);
 }
}

// Superclass for all vehicles
abstract class Vehicle {
 protected GoBehavior goBehavior;
 protected ControlBehavior
controlBehavior;

 public Vehicle(GoBehavior goBehavior,
ControlBehavior controlBehavior) {
 this.goBehavior = goBehavior;
 this.controlBehavior = controlBehavior;
 }

 public void goTo(String destination) {
 goBehavior.goTo(destination); // Uses
delegation
 }

 public void notifyDriver(String message)
{
 controlBehavior.notifyDriver(message); //
Uses delegation
 }

 public void setGoBehavior(GoBehavior
goBehavior) {
 this.goBehavior = goBehavior;
 }

 public void
setControlBehavior(ControlBehavior
controlBehavior) {
 this.controlBehavior = controlBehavior;
 }
}

// Concrete vehicle types
class Car extends Vehicle {
 public Car() {
 super(new DriveBehavior(), new
HumanBehavior());
 }
}

class Aircraft extends Vehicle {
 public Aircraft() {
 super(new FlyingBehavior(), new
HumanBehavior());
 }
}

class Drone extends Vehicle {
 public Drone() {
 super(new FlyingBehavior(), new
RemoteBehavior());
 }
}
```

> La qualità di una soluzione software si vede di fronte ai cambiamenti.

17.4. Pattern creazionali

17.5. new è pericoloso

Ogni volta che si crea una classe concreta con new si crea una dipendenza (ad esempio se cambia il costruttore). Queste dipendenze sono di cattiva qualità e si ha una violazione del DIP.

```
interface PersistanceManager {
  void save(...);
}

class DBPersistanceManager implements
PersistanceManager {
  @Override
  void save(...) {
    ...
  }
}
```

```
void main(){
  PersistanceManager pMgr = new
DBPersistanceManager();
  pMgr.save(...)
}
```

`pMgr` dipende sia dalla classe concreta che dall'interfaccia. Non vogliamo che quando cambia la classe concreta bisogna cambiare `new DBPersistanceManager`.

Si delega la creazione dell'oggetto a una classe a parte, detta factory. Le factory separano il cliente dal processo di istanziazione e delega la creazione dell'oggetto ad un interfaccia comune. Non si ha più dipendenza dalla classe concreta, ma si ha dipendenza dalla factory.

```
void main(){
  PersistanceManager pMgr =
Factory.create();
  pMgr.save(...)
}
```

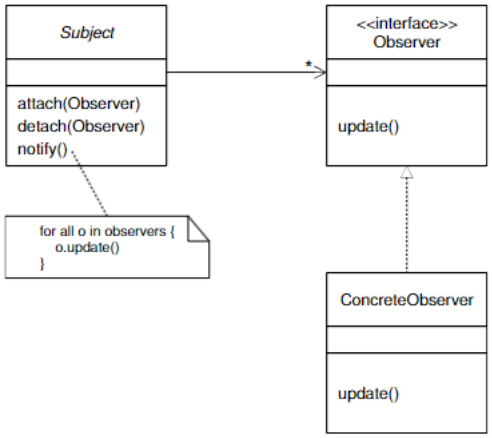
Vogliamo avere dipendenza di buona qualità, non molte dipendenze di basse qualità.

- Le interfacce solo elementi stabili, quindi sono dipendenze buone.
- Le factory sono elementi stabili, perché vi si possono solo aggiungere metodi, quindi dipendenze di alta qualità.

Con le factory si hanno una sola dipendenza critica, indipendentemente dal numero degli utilizzatori. Tutte le factory sono implementate con la stessa interfaccia.

17.5.1. Problema della notificazione/polling

Vogliamo che le modifiche di una classe si propaghino su una serie di oggetti. Si fa che gli oggetti interessati ricevono la notifica quando avviene il cambiamento, non viceversa. Si definisce una dipendenza uno a molti tra oggetti cosicché quando uno cambia stato, tutti i suoi dipendenti sono notificati e automaticamente cambiati.



Problema: come posso isolare un client dalla complessità interna di un sottosistema? Le Façade forniscono un'interfaccia unica a una serie di interfacce di un sottosistema. Le Façade definiscono un interfaccia di alto livello che rende un sottosistema più facile da usare.

17.6. Singleton

Problema: come posso garantire che una classe abbia una sola istanza e fornisca un punto di accesso globale a essa? Il Singleton assicura che una classe abbia una sola istanza e fornisce un punto di accesso globale a essa. I singleton sono un tipo di code smell, ma garantiscono che una sola istanza di una classe venga creata. In genere sono singleton

- Factories
- Loggers
- Classi di configurazione
- Accesso alle risorse

Non si vuole che siano singleton

- Classi dove una singola istanza è parte della specifica del sistema, ma non intrinseca al dominio del problema
- Oggetti che dovrebbero essere globalmente accessibili.

17.7. Proxy

Problema: come posso intercettare l'accesso ad un oggetto per indirizzare problemi ortogonali? I problemi ortogonali ono problemi che non sono parte del dominio del problema, ma che sono comunque importanti. Ad esempio, la sicurezza, il

logging, la gestione della memoria, ecc. Il proxy fornisce un sostituto o un segnaposto per un altro oggetto per controllare l'accesso a esso. Il proxy può aggiungere comportamento senza aggiungere responsabilità. Il proxy si usa per

- controllo dell'accesso
- contare gli accessi
- loggare gli accessi
- accedere ad oggetti RemoteBehavior

17.8. Decorator

Con i decorator si aggiungono responsabilità a un oggetto in modo dinamico. I decorator forniscono una flessibilità maggiore rispetto all'ereditarietà, perché si possono combinare più decorator.

17.9. Adapter

Problema: come posso accedere a una classe i cui metodo non sono quelli che il client si aspetta (parametri, tipi…)? L'adapter converte l'interfaccia di una classe in un'altra interfaccia che il cliente si aspetta. L'adapter permette a classi che non possono lavorare insieme a causa di interfacce incompatibili di lavorare insieme.

17.10. Bridge

Problema: come posso terminare la prevalenza di astrazioni del cliente? Bridge: separa un'astrazione dalla sua implementazione affinché le due possano variare indipendentemente. Si usa quando si ha un'astrazione e più implementazioni di essa. Le implementazioni possono essere cambiate a runtime. La pattern del bridge mette le astrazioni e le implementazioni in due classi separate, e si usa la delegazione per legarle.

17.11. Memento

Senza violare l'incapsulamento, si cattura e si esternalizza lo stato interno di un oggetto affinché possa essere ripristinato in un momento successivo. Un caretaker chiede a un originator di salvare il suo stato, e poi lo ripristina quando necessario. Il caretaker non sa come è fatto lo stato, ma sa che può essere ripristinato. Il memento è l'oggetto che contiene lo stato dell'originator. Il memento non ha metodi pubblici, quindi non può essere modificato dall'esterno.

17.12. Iterator

Fornisce un modo per accedere agli elementi di un oggetto aggregato senza esporre la sua rappresentazione interna. L'iteratore ha metodi per accedere agli elementi e per controllare se ci sono più elementi. Il client non sa come è fatto l'oggetto aggregato, ma sa che può essere iterato. Il client non sa nemmeno se l'iteratore è interno o esterno all'oggetto aggregato.

17.13. Mediator

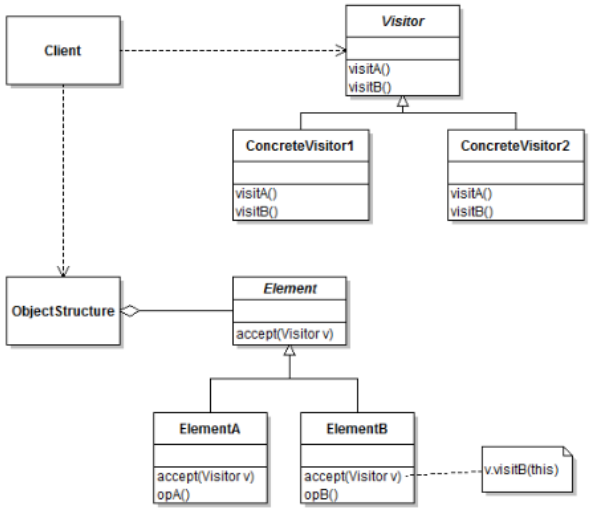
Si definisce un oggetto che incapsula come un insieme di oggetti interagiscono. Il mediatore promuove il loose coupling evitando che gli oggetti si riferiscano l'uno all'altro esplicitamente, e permette di variare le interazioni tra gli oggetti. Il mediatore ha metodi per registrare gli oggetti e per invocare i metodi degli oggetti registrati.

17.14. Composite

Si compongono oggetti in strutture ad albero per rappresentare gerarchie parte-tutto. Il composite permette di trattare oggetti singoli e composizioni di oggetti in modo uniforme. Il composite ha metodi per aggiungere e rimuovere oggetti figli, e per invocare i metodi degli oggetti figli.

17.15. Visitor

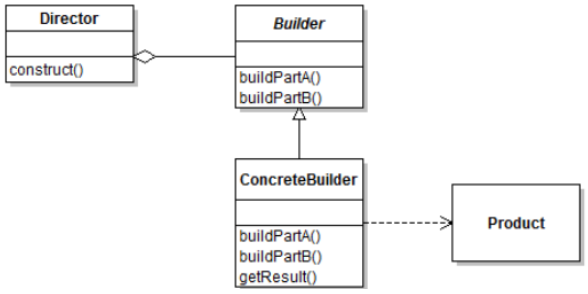
Simile all'Iterator, ma usa il principio di *inversion of control*. Al posto di usare `iterator.next()` si richiama un metodo per ogni elemento. Si prende ognuno degli elementi che compongono il visitor e si invoca un metodo su di esso. Il visitor ha metodi per ogni tipo di elemento, e il client non sa come è fatto l'elemento, ma sa che può essere visitato. Il visitor permette di aggiungere nuove operazioni senza cambiare le classi degli elementi.



17.16. Builder

Pattern creazionale che si usa per separare la costruzione di un oggetto complesso dalla sua rappresentazione in modo che lo stesso processo di costruzione possa creare rappresentazioni diverse. Il builder ha metodi per costruire le parti dell'oggetto e per restituire l'oggetto completo. Il client non sa come è fatto l'oggetto, ma sa che può essere costruito.

```
Foo foo =
  Foo.builder().
  setWidth(a).setHeight(b).
  setDepth(c).setColor(d).build()
```



Il director restituisce l'oggetto aggregato, costruito dal builder.

17.17. Command

Si incapsula una richiesta come un oggetto, permettendo di parametrizzare i client con code e operazioni. Il command permette di ritardare l'esecuzione di una richiesta, di mettere in coda le richieste e di supportare le operazioni annullabili. Il command ha metodi per eseguire e annullare la richiesta. Il client non sa come è fatto il comando, ma sa che può essere eseguito.

17.18. Abstract factory

Fornisce un'interfaccia per creare famiglie di oggetti correlati o dipendenti senza specificare le loro classi concrete. L'abstract factory ha metodi per creare gli oggetti e il client non sa come è fatto l'oggetto, ma sa che può essere creato. Il client non sa nemmeno se l'oggetto è interno o esterno all'abstract factory.

17.19. Prototype

Specifica i tipi di oggetti da creare usando un'istanza esistente come prototipo. Il prototype consente di copiare oggetti esistenti senza rendere il codice dipendente dalle loro classi. Il prototype ha metodi per clonare l'oggetto e il client non sa come è fatto l'oggetto, ma sa che può essere clonato.

17.20. Flyweight

Si usa la condivisione per supportare grandi quantità di oggetti granulari in modo efficiente. Il flyweight è un oggetto che condivide il suo stato con altri oggetti. Ha metodi per accedere allo stato condiviso e il client non sa come è fatto l'oggetto, ma sa che può essere condiviso. Il flywheight è istanziato una sola volta in sistemi poveri di memoria.

17.21. Chain of responsibility

Si evita di accoppiare il mittente di una richiesta al suo ricevente, dando la possibilità di passare la richiesta lungo una catena di gestori. La catena di responsabilità consente di invocare più oggetti senza sapere chi gestirà la richiesta. Il client non sa come è fatto l'oggetto, ma sa che può essere gestito. Il client non sa nemmeno se l'oggetto è interno o esterno alla catena di responsabilità.

17.22. Interpreter

Data una lingua, si definisce una rappresentazione per la sua grammatica tramite

un interpreter che usa la rappresentazione per interpretare le frasi della lingua. L'interpreter ha metodi per interpretare le frasi e il client non sa come è fatto l'oggetto, ma sa che può essere interpretato.

17.23. Pattern Moderni

17.23.1. Principio holliwoodiano

> Non chiamarci, noi chiamiamo te.

Il principio holliwoodiano è un principio di design che incoraggia a scrivere codice che non dipende da implementazioni specifiche, ma piuttosto da interfacce o astrazioni. Questo principio è spesso associato a tecniche di programmazione come **l'inversione del controllo** e la programmazione orientata agli eventi.

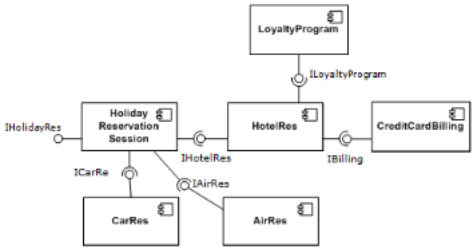
17.23.2. Dependency injection

Inversion of control non è un pattern, è così vago che è un idioma. Sempre basato su IoC è il *Dependency Injection* (DI), che è un pattern di design che implementa IoC. Il DI è un modo per fornire le dipendenze a un oggetto piuttosto che far sì che l'oggetto stesso le crei. In questo modo, si riduce il coupling tra gli oggetti e si facilita il testing e la manutenzione del codice.

Si elimina la dipendenza da `new` e Factory, semplificando la fase di testing e aumentando la modularità. Tipi di injection:

- Constructor injection: le dipendenze vengono passate al costruttore dell'oggetto.
- Setter injection: le dipendenze vengono passate tramite metodi setter.
- Interface injection: l'oggetto richiede le dipendenze tramite un'interfaccia.
- Method injection: le dipendenze vengono passate come argomenti a un metodo specifico.

Nelle componenti basate su componenti, i sistemi software sono costruiti incollando componenti in base alle interfacce fornite e requisite.

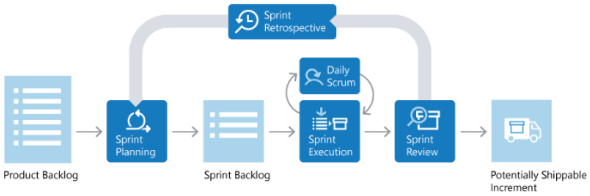


I componenti sono scambiabili purché implementano le stesse interfacce.

18. Scrum

Uno scrum è una metodologia nata nel contesto software, rientra nelle famiglie dei metodi *lean* per l'organizzazione aziendale.

18.1. Ciclo di vita



Scrum ha un ciclo di vita iterativo. L'esecuzione si chiama sprint.

18.1.1. Sprint

Uno sprint è un iterazione timeboxed, ha una durata prefissata e include tutti i cicli di design, programmazione e testing. Termina con un incremento potenzialmente vendibile.

18.1.2. Ruoli

- Core (pigs): non si possono fare scrums senza i ruoli core:
  - *product owner*: stakeholder, decide le priorità, scadenze e feature. Guarda il progetto con gli occhi del committente.
  - *scrum master*: persona senior all'interno dell'organizzazione, che è stata attiva nello sviluppo e conosce bene scrum, ed è al servizio dei colleghi. Infatti è responsabile della corretta applicazione delle pratiche di scrum all'interno del progetto.
  - team di sviluppo: auto organizzati, full time sul progetto, l'insieme delle competenze dei

membri deve coprire tutte le competenze necessarie, di 5-9 persone. Questa forma di scrum non è appropriata per progetti di grosse dimensioni. Non si vuole dipendere da risorse esterne perché potrebbero bloccare la fase di sviluppo. Sono i membri del team a decidere che lavoro vogliono fare.
- Additional (chickens):
  - clienti
  - management

18.1.3. Artefatti

- Product backlog: lista di tutte le cose che devono essere fatte nel progetto. Verrà completato in più iterazioni. In questa lista, curata dal *product owner*, ci sono
  - Requisiti funzionali (user stories)
  - Bugfix
  - Requisiti non funzionali
  - Requisiti tecnologici
  - Chore: cose che servono, ma non all'utente finale
- Sprint Backlog: contiene cose da fare all'interno dello sprint nel dato intervallo di tempo. A ogni task è associato un tempo per completarla.
- Potentially shippable product increment
- (Burn down chart)

18.1.4. Sprint planning

Si scelgono le task restanti da completare nello prossimo sprint e si stimano la loro durata. La fase di pianificazione dura una giornata. Il product owner indica le funzionalità, e il tempo per implementarle va deciso dal team.

18.1.5. Scrum Estimation

Le user stories usano dei punti in base alla complessità.

- Capacity driven planning: si prendono le storie dal backlog in base al tempo richiesto
- Velocity driven planning: si scelgono le storie dal backlog in base ai punti.

18.1.6. Scrum Giornaliero

Incontro di 15 minuti dove membri del team rispondono a

- cosa ho fatto ieri per contribuire allo sprint?
- cosa farò oggi per contribuire allo sprint?
- prevedo ostacoli al raggiungimento del goal dello sprint?

Non si fa analisi che è lasciata al product owner

18.1.7. Sprint review

Alla fine di uno sprint si fa la review con tutto il team e gli stakeholder, si presenta l'incremento con tutti i problemi e soluzioni. Si discute del product backlog. C'è poi la retrospezione con lo scrum master e il team di sviluppo, dove si discutono miglioramenti per il prossimo sprint.

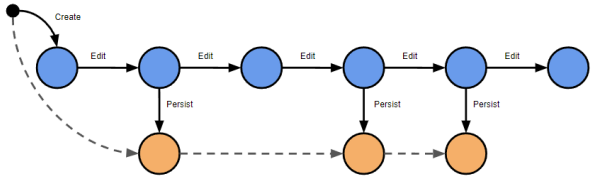
18.1.8. Scaling

Per progetti grandi, può essere possibile dividerlo in sottosistemi. Ci sono framework ispirati a scrum per funzionare a larga scala (leSS).

19. Collaborazione

19.1. Version Control

19.1.1. Ciclo di vita degli artefatti



Non si vuole che tutti i salvataggi fatti siano recuperati. Quando si fa un commit, un sistema prende la copia attuale del file e la mette da un'altra parte dove può poi essere recuperata.

19.1.2. Gestire le revisioni

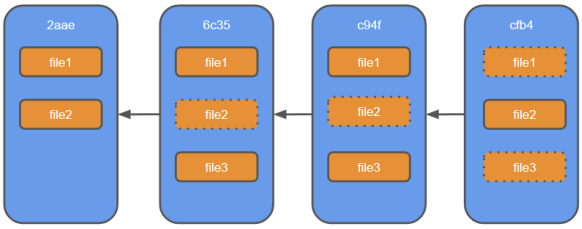
Implementare un operazione di commit che fa **snapshot** di file. Una **repository** contiene file versionati, i cui snapshot possono essere presi e ripristinati (operazione di **restore**). Ogni file può avere associato altri dati (metadati):

- data e timestamp dello snapshot
- chi ha modificato il file

- perché è stato fatto

19.1.3. Il Changeset

Quando si creano revisioni spesso si opera non su i singoli file, ma su un gruppo di file correlati che sono stati tutti modificati per uno stesso fine. Questo gruppo di file si chiama **changeset** Lo stream di snapshot:



Quello a sinistra è quello più vecchio. Non serve memorizzare un file negli snapshot successivi se non è stato modificato.

19.1.4. Collaborazione

Il sistema di revisione è un sistema distribuito

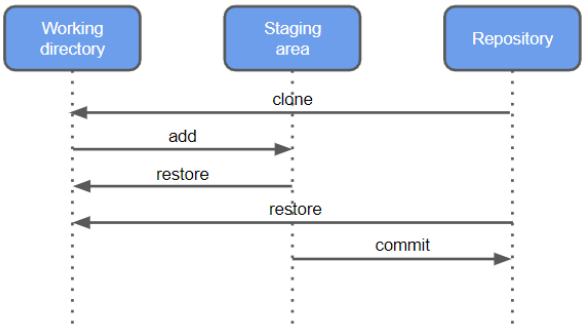
- Approccio client-server (centralizzato): il sistema di revisione è un servizio su un host remoto che espone un'API
- Approccio peer-to-peer (distribuito): tutte le workstation hanno repository locali che possono essere sincronizzate con quelle remote. In genere si ha un'unica repository remota e più repository locali che possono essere sincronizzate con quella remota.

19.1.4.1. Conflitti

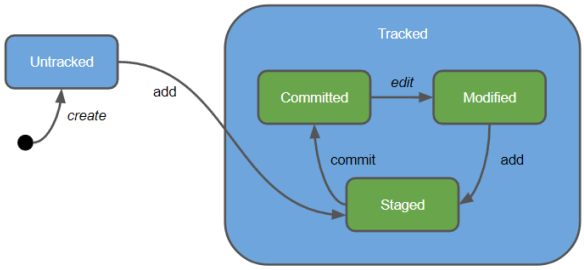
- Sistema centralizzato: si usa un lock: solo l'utente che ha il lock per un certo file può creare revisioni per quel file. Questo sistema ha dei problemi:
  - Ognuno può vedere le revisioni degli altri
  - È un problema se qualcuno si dimentica di rilasciare un lock.
  - cattive abitudini di gestione del lock influenzano tutti gli sviluppatori
  - Se il server è spento tutti sono bloccati.
  - Se non c'è la rete non si può lavorare.
- Sistema distribuito: si gestiscono: si definiscono workflows per minimizzare l'impatto dei conflitti. Quando ci sono cambiamenti contemporanei, si sceglie uno tra i cambiamenti o si fa un merge manuale.

19.2. Git

Git è un sistema di versionamento distribuito.

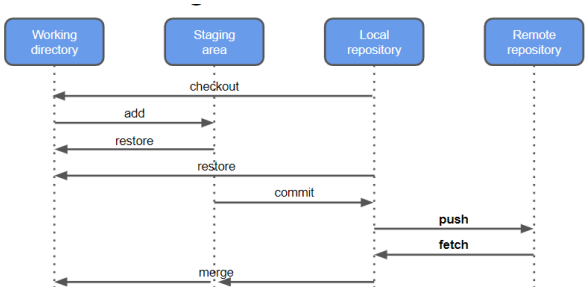


È l'utente a scegliere quali file modificati inserire nel prossimo snapshot. Nella staging area si sceglie quali file includere nello snapshot e quali no (i file esclusi non fanno parte del changeset).
Gli stati dei file



Una volta che il file è tracciato, il file si può trovare in 3 stati

- **committed**: l'ultima versione del file locale coincide con quella sulla repository remota
- **modified**: l'ultima versione del file locale non coincide con quella sulla repository remota
- **staged**: il file modificato sarà inserito nel prossimo changeset.



19.2.1. Branching

Un branch è una linea temporale che diverge da quella principale. I branches devono essere esplicitamente creati e nominati. I branch non divergono per sempre, a un certo punto si vuole fare riconciliare i branches.

19.2.2. HEAD

Lo stato della directory è elaborato in relazione a una specifica commit. HEAD è il riferimento che punta a quel commit. Quando si cambia branch corrente, si fa puntare HEAD a quel branch, il quale a sua volta farà riferimento alla commit in cima.

> In genere head punta al commit più recente del branch su cui si sta lavorando