**USE CASE (+ diagram)**

Testo strutturato (form/template) con vocabolario il dominio dell’applicazione. Si hanno descrizioni chiare e indipendenti dalle tecnologie.

Template: Nome (goal in frase verbale attiva e UpperCamelCase), ID, descrizione, attori primari, attori secondari, precondizioni (vincoli sullo stato corrente), scenario principale, postcondizioni (condizioni che devono essere vere quando lo use case termina con successo) e scenari alternativi

Elementi: Attori (ruoli delle entità (non le entità stesse) che interagiscono con il sistema), Use Case (quello che gli attori possono fare), Relazioni (partecipazione di un attore a uno use case), Confini del sistema (rettangolo attorno agli use cases)

Categorie di attori: primari (chi ha mire o guadagna sul/dal sistema) o secondari (quelli su cui il sistema ha mire o che producono qualcosa per il sistema).

Scenario: sequenza ordinata di interazioni tra sistema e attori (istanza → singolo cammino dello use case). Uno Use Case è un insieme di scenari che hanno in comune lo scopo finale dell’utente. Il principale è quello “perfetto” senza eventi indesiderati, sennò è secondario.

Uno scenario è costituito da un elenco di passi del tipo <#>Il<Sistema/Attore><azione>. Si ha una deviazione quando ci si allontana dallo scenario principale (semplice (se, altrimenti) o complessa (seq. di eventi alternativa), mentre si ha una ripetizione con per e fintantochè

Le specifiche degli scenari alternativi dello Use Case X hanno ID X.1, X.2, X.(...), e occorre specificare la relazione con lo use case principale (da che passo del principale inizia, o alternativamente in qualunque momento l’utente debba compiere l’azione alternativa).

Gerarchie di attori: i figli ereditano tutte le relazioni del padre

Inclusione tra use case: per decomporre uno use case complicato. Si usa per comportamento comune a più Use Cases → concetto di funzione, in cui lo Use Case principale (incompleto da solo) esegue i passi fino al punto di inclusione e poi lascia il controllo allo Use Case incluso (che può essere completo a sua volta o un frammento).

Si rappresenta con include(B) messo in A (grafico: freccia da A a B)

Estensione: mostra un comportamento opzionale (lo Use Case base è completo anche senza, semplicemente definisce un punto di estensione senza conoscere il comportamento dell’estensione). Si rappresenta con un extension point nello Use Case di base A (grafico: freccia da B ad A) e poi specificando nello use case di estensione che lo è.

Generalizzazione (per specializzare uno Use Case): i figli possono ereditare i passi del genitore (parentesi attorno al numero del passo per ereditare “as is”), aggiungere nuovi passi(a cui segue una rinumerazione) e ridefinire i passi ereditati (“1.” → “(o1.)”)

**UML - Class Diagram**

Definisce la parte statica di quello che voglia modellare, ossia: classi, feature (attributi, operazioni, metodi) e relazioni tra classi (associazioni, aggregazione/composizione, specializzazione/generalizzazione, dipendenze). Fa tutto questo su due possibili prosepttive: la concettuale (analista) in cui si descrive il mondo e la sw (sviluppatore) in cui si descrive il design di quella che sarà l’implementazione.

Classe: rettangolo con nome all’interno in cui poter aggiungere attributi e operazioni (se non si separa il rettangolo si sopprimono info, se si lasciano delle parti vuote s’intende che non ci sono attributi e/o operazioni (il nome è obbligatorio)

Attributi: “visibilità nome: tipo [molteplicità] = default {proprietà}”, con visibilità + (pubblica) o - (privata), tipo primitivo o di classe definita nello stesso modulo (spesso in questo caso si usano associazioni) e proprietà come ad es. readOnly

Molteplicità: 1 (default), 0..1, \* = 0..\*, 1..\*, n .. m, m = m >= 1 (gli elementi di un attributo con molteplicità > 1 sono considerati un set, se voglio dotarli di ordine aggiungo {ordered}).

Operazioni: “visibilità nome (lista param): returnType {proprietà}” con lista parametri nella forma “direzione nome: tipo = default” (con direzione in, out o inout). Si omettono getter/setter e si ricordano i due tipi di operazioni: query (no side-effect) e modificatori

Datatype: stereotipo <<datatype>>: diverso da oggetto perchè al suo contrario non ha un'identità (due oggetti con stessi valori sono diversi perchè hanno ID diverso, mentre le istanze dei datatype sono valori).

Associazioni: relazione fisica o concettuale tra classi. Il “>” accanto al nome dell’associazione indica la direzione di lettura (ad esempio Persona-possiede>-Auto). In alternativa si può indicare il ruolo (ad esempio Persona(Proprietario)-Auto). Si può indicare il verso di navigazione, ossia la direzione verso la quale si possono reprire info (ad esempio Persona-Possiede→Auto indica che nota una Persona si può sapere che auto possiede).

Molteplicità delle Associazioni:numero di link (istanza dell’assoc.) tra gli oggetti. Si legge al contrario dell’ER, ad esempio [Persona]1–Possiede–1..\*[Auto] indica che una Persona possiede almeno un'Auto (e una Auto è posseduta da una e una sola Persona).

Static: attributi (oggetti di una stessa classe condividono stesso valore per un attributo) e operazioni (operano sulla classe stessa e non su una x istanza (es: costruttori).

Associazioni riflessive: associazione a se stesso (es: “padre” con Cartella)

Associazioni bidirezionali: linea senza arrow o entrambe le arrow (navigabili in due direzioni)  
Associazioni bidirezionali 1 a 1 → sincronizzazione (es. marito e moglie)

Implementazione: non esiste un unico modo (di solito associazione N=1 → attributo, N=0..\* → HashSet, N={ordered}0..\* → Linked/ArrayList

Commenti: come i lingProgr a livello classe, per il behaviour dell’operazione a livello feature

Aggregazione: relazione “tutto-parti” (Regione è parte di una Nazione: Nazione<>--Regione)

Composizione: forma forte di aggreg. (“è composto di”: se il composto viene distrutto le parti anche + una parte può appartenere ad un solo oggetto intero alla volta (non condivisione)).

Generalizzazione (relazione concettuale “è un”, in cui una classe è anche istanza della superclasse) VS Ereditarietà (meccanismo implementativo in cui elementi specializzati incorporano struttura/comportamento dei generali) → aggiunta di nuovi attributi e operazioni, overriding, ecc..  
Metodo: diverso da operazione (dichiarazione procedura astratta) → implementazione

(ad es. posso avere un operazione generica isIn() e tre metodi diversi che la implementano)

Dipendenze: Cliente - - -> Fornitore indica che una modifica in Fornitore può apportare un cambiamento al Cliente (perchè Cliente “dipende da” Fornitore, ad esempio perchè ne chiama le operazioni, o crea oggetti di quel tipo, o usa quel tipo come var locale o param).

Ad esempio LineStorage- ->Line (NB: servono solo le dipendenze più importanti + nello sviluppo si vogliono minimizzare le dipendenze ed evitare le cicliche (es: tangle in package)

Interfaccia: o l’insieme delle op. visibili all’esterno degli oggetti che sono istanze di quelle classe oppure un’entità simile ad una classe ma priva di implementazione. Si usa <<interface>> nell’interfaccia e si collega con freccie tratteggiate ma con la punta intera.

→ si usano per separare l’implementazione dall’interfaccia, cioè per diminuire l’accoppiamento tra classi (+ avere ereditarieà multipla dove non supportata)

Lollipop (notazione per interfacce): il lollipop rappresenta l’interfaccia esposta da una classe e il semicerchio (socket) rappresenta l’interfaccia richiesta (è una dipendenza). Ad esempio:

SearchEngine – O) “Cerca” – Visualizzatore indica che un SearchEng fornisce la possibilità di accedere al proprio servizio “Cerca” tramite interfaccia, e che la classe Visualizzatore utilizza tale servizio.

Vincolo: per ridurre il #oggetti ammissibili: si possono aggiungere tramite note tra parentesi graffe, ad es. in linguaggio OCL: basato su logica del prim’ordine → ecco alcuni esempi:  
“Il proprietario di un veicolo deve avere almeno 18 anni”

Context Veicolo, Inv: self. proprietario.età >= 18

“nessuna Persona ha più di tre veicoli”

Context Persona, Inv: self.parcoVeicoli → size <= 3 (equivalente a 0..3 nelle molt.)

“Tutti i veicoli di una persona sono neri”

Context Persona, Inv: self.parcoVeicoli → forAll(v|v.colore=nero)

“compleanno() incrementa di 1 l’età di una Persona”

Context Persona::compleanno(), post: self.età = self.età@pre + 1

**UML - Sequence Diagram**

Messaggi sincroni: il mittente si pone in attesa del risultato (freccia piena)  
Messaggi asinconi (x processi concorrenti): il mittente continua (freccia vuota)

Partecipanti: “nome:Tipo” o “:Tipo” (anonimo) → ognuno con la sua linea di vita

specifica di esecuzione: intervallo di tempo in cui il partecipante ha il controllo (linea di vita piena invece che tratteggiata)

messaggio trovato: messaggio con mittente omesso

sintassi dei messaggi: returnVal = message(params):returnType

es: da “getProduct ” (omissioni) fino a “d=getProduct(id:ItemID):Product”  
→ fare d1=getDate all’andata oppure una freccia di ritorno con sopra “aDate” è uguale

self (chiamata metodo della classe stessa): freccia verso una box sulla linea di vita, poi exit

creazione oggetto: freccia new e oggetto creato alla stessa altezza

distruzione: freccia con x alla fine, sulla linea di vita dell’oggetto cancellato

frame: cornici per la logica di controllo (diversi operatori + eventuale guardia)

operatori: loop (while con guardia), opt (if), alt (if-else), ref (a un altro Sd), par (//)

Per cosa? Per visualizzare interazione tra oggetti, NON logica di controllo / algoritmi

Livelli: SSD (di sistema) a livello requisiti, oppure SD a livello design

SSD: eventi I/O del sistema relativi a uno o più casi d’uso (utile per chiarire I/O,  
 testing, per passaggio da req a design (punto di partenza per design))  
*UML - Communication Diagram*  
Rispetto ai Sequence (che danno enfasi sulla sequenza temporale dei msg), danno enfasi sulla comunicazione tra gli oggetti. Vedi le slide per i pochi dettagli sulla sintassi

**UML - State Machine (comportamento di un obj per il suo ciclo di vita)**

Diagramma di comportamento: descrive come comportarsi quando riceve un input, in modo diverso a seconda dello stato = situazione della vita di un'entità durante la quale delle condizioni sono soddisfatte e alcune azioni possono essere eseguite (per UML lo stato è più astratto rispetto alla def OOP che si basa sul set di valori di attributi).

Graficamente: grafi con nodi=stati e archi=transizioni(evento+guardia+attività)

Semantica: associata a un entità, che descrive - eventi si salvano su una coda ed estratti man mano - la semantica degli eventi è run-to-completion (e2 estratto solo se e1 processato) - se due guardie g1, g2 sono vere insieme si esegue ei random (non det)

Nodo init (1-1) e Nodo end (0-N)

Transizione: stati origine-dest + (e/g/a) → tutte opzionali (se NO e: eseguo subito)

NB: e è subìto, a è eseguita / a è espressa in modo informale o pseudocode o ocl,...

Prospettiva: concettuale → in prospettiva sw corrisponde a una classe

Eventi: msg, when(qualcosa), temporali, …

Attività interne (senza transizione di stato): mostrate nel secondo slot di un evento (se ci sono). Due attività interne speciali sono entry ed exit (es: exit / update field). È simile ad una self transition ma NON identica semanticamente (perché con quella a ogni giro riattivo entry & exit). C'è anche l'attività do-activity, che non è istantanea (può durare nel tempo ed essere interrotta): se c'è una transizione senza evento da un evento con una do-activity, essa non viene eseguita finché la do-activity non ha finito.

Stati compositi: macro-stato con altri dentro (posso mostrarlo o nasconderlo con i due pallini)

→ transizioni in uscita dal macro vengono ereditate da tutti quelli micro

**UML - Activity Diagram**

Descrivono come viene svolta un'attività (flusso di azioni) di una o più entità (cioè qual'è il flusso di azioni e in che ordine). Sono usati in vari contesti diversi, ad es. similarmente si flowchart (con la differenza che abbiamo i fork e i join per il parallelismo).

Nodi: azione: unità di comportamento/ oggetto: obj come IO di azioni / controllo: flusso

→ Nodi di controllo: decisione(if), fork, inizio, fine attività (intera), fine flusso (fine ramo)

→ Nodi di azione: o informale, o invocando un'altra attività (con simbolo rake = fork), o invocando op di altra classe, o (raro) con pseudocodice → la scelta dipende dal livello

Token: "pedine che viaggiano lungo il diagramma" (un nodo azione viene eseguito quando ci sono token su tutti i nodi in entrata, poi genero i token per i nodi in uscita).

Comportamento nodi: quello che ci si aspetta da fork e join

Nodo init (1-1) e Nodo end (0-N)

→ se raggiungo un nodo finale di attività finisce tutto indipendentemente dal resto

Stato: si può aggiungere lo stato di un obj per mostrarne l'evoluzione, Partizioni/swimlanes: per far capire chi esegue cosa (partecipanti), Parametri e valori di ritorno: se ci sono, sono rappresentati come nodi oggetti al bordo delle attività, Regioni interrompibili: interruz. forzate (es: eccezioni) espresse con rettangolo tratteggiato da cui poter uscire se c'è un problema

**UML - Component Diagram**

Descrivono entità logiche realizzate da artefatti (entità fisiche) → componenti di un sistema e dipendenze tra esse → architettura sw di un sistema (high level design), perché le classi sono a grana troppo fine (non buona panoramica)

Facade: entità che espone l'interfaccia all'esterno

Componente: o con parola chiave <<component>> o con apposita icona

Interfacce: lollipop (fornite) o socket (richieste), oppure con le frecce

Dipendenza: Manager →<<use>> Office (Manager ha bisogno di Office)

Es Type: PC (su tutti i PC diversi). Oppure generalizz. verso PC (v. Esercizio su slide)

**UML - Deployment Diagram**

Mette insieme le componenti dal Component (logiche) con gli artefatti (fisici) che le implementano, dislocate su diverse macchine.

Si possono unire Component e Deployment

Nodi: rappresentano qualcosa che può ospitare del sw (cioè una risorsa computazionale, come una periferica fisica (device) o un ambiente sw di esecuzione. Una connessione tra nodi rappresenta un canale di comunicazione (es: specificando il tipo TCP/IP)

Artefatti: entità concrete del mondo reale (source code, script, tabelle db, pagine html). Possono essere dislocati (deployati) sui nodi.

Dipendenza: relazione tra artefatti (cambiando x potrei cambiare y)

Manifest: rappresentazioni/manifest. fisiche dei componenti

<<device>>: dispositivo hw / <<executionEnviroment>>: ambiente esecuzione

→ si possono specificare ulteriori info con i tagged value, es: {OS = "Linux"}

**UML - Package Diagram**

Costrutto che permette di mettere insieme più elementi UML. Ogni package definisce un namespace (regione dove tutti i nomi devono essere univoci) → si evita confusione su entità con nomi simili, usando poi il nome completamente qualificato (pack::name). Si ha il simbolo del package e vari modi per includere cose dentro un package.

Visibilità: + pubblica (visibile ai clienti del package), - privata

Dipendenze: es: package Fish dipende da Water (le dip dai package riassumono le interne)

→ nel package Fish esiste almeno una classe che dipende da una classe che è nel package Water

Per suddividere classi in package bisogna seguire i soliti principi di buona progettazione:

High cohesion, low coupling, common reuse principle (classi in package riusate insieme), acyclic dependency principle (no cicli (tangle)), … → fusione e separazione per risolvere cicli