1 Lezione del 10-10-25

1.0.1 Ereditarietà multipla

Torniamo al discorso dell'ereditarietà. In UML è permessa l'**ereditarietà multipla**, in quanto questa esiste in linguaggi come il C++. Spesso, però, questo rappresenta un pattern da evitare ed è considerato un errore di progetto.

In prospettiva che si verificheranno errori, infatti, è meglio mantenere lineari le catene di ereditarietà in modo da minimizzare problemi.

1.1 Realizzazione casi d'uso

Abbiamo discusso finora la *modellazione delle classi di analisi*, che deve essere il primo artefatto generato dalla fase di analisi. Veniamo adesso al secondo, cioè la **realizzazione dei casi d'uso**.

Se le classi rappresentavano l'aspetto **statico** del sistema, le realizzazioni dei casi d'uso ne rappresentano l'aspetto **dinamico**.

Gli obiettivi che ci prefissiamo sono:

- Trovare quali classi interagiscono per realizzare il comportamento specificato da un caso d'uso;
- Trovare quali **messaggi** queste classi devono inviarsi per realizzare tale comportamento. In questo modo possiamo determinare quali *operazioni*, *attributi* e *relazioni* le classi dovranno avere.

Per dettagliare questo comportamento sfruttiamo 4 diagrammi, che sono:

- Diagrammi di **sequenza**: mostrano la sequenza di messaggi, dall'alto verso il basso, che due classi si scambiano per realizzare un comportamento;
- Diagrammi di **comunicazione**: mostrano le relazioni strutturali degli oggetti;
- Diagrammi di **visione generale dell'interazione**: mostrano come più comportamenti semplici implementano comportamenti complessi;
- Diagrammi di **temporizzazione**: mostrano gli aspetti in tempo reale di un'operazione.

1.1.1 Linee di vita

Una *linea di vita* (**lifeline**) rappresenta un partecipante in un'interazione, e in particolare il *ruolo* che tale partecipante intepreta in tale interazione.

Ogni linea di vita ha un nome (opzionale), un tipo e un selettore (opzionale). Il selettore è una condizione booleana che può essere usata per selezionare quali istanze partecipano all'interazione.

1.1.2 Messaggi

Un **messaggio** specifica una comunicazione fra due linee di vita in un'interazione. La comunicazione può essere:

- L'invocazione di un'operazione;

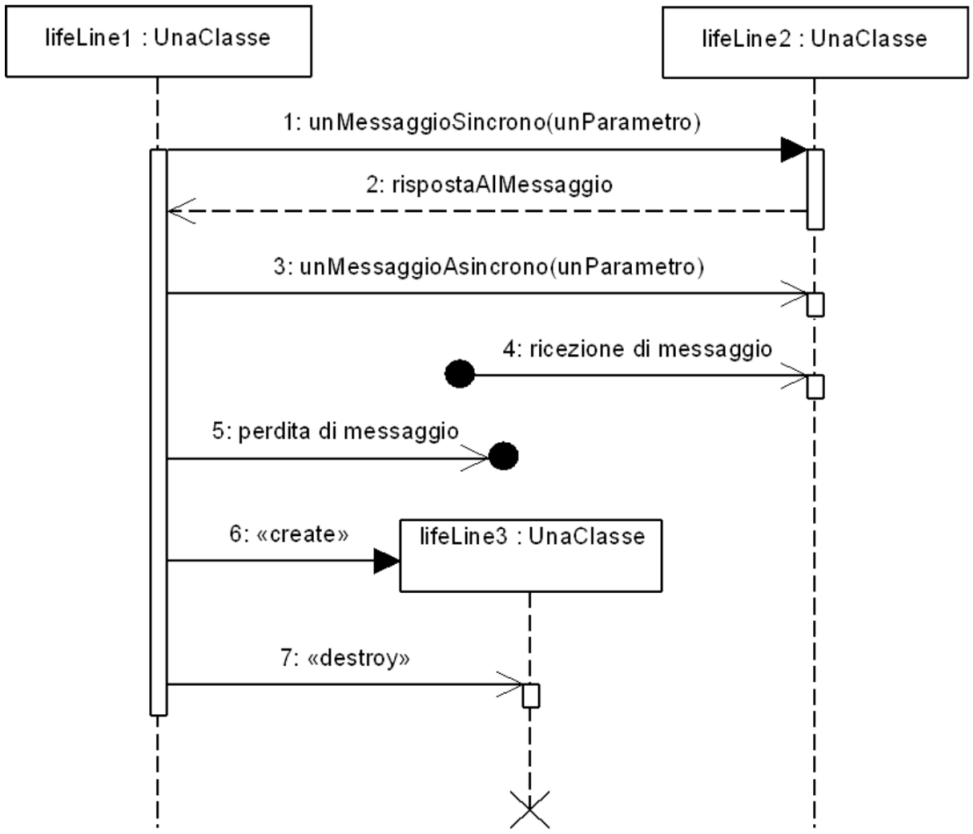
- La creazione o distruzione di un'istanza;
- L'invio di un segnale.

La freccia che rappresenta il messaggio può codificare diverse informazioni:

- **Messaggi sincroni**, rappresentati da freccia piena, significano che il mittente si ferma ad aspettare che il ricevitore esegua il messaggio;
- **Messaggi asincroni**, rappresentati da freccia vuota, significano che il mittente non si ferma ad aspettare il ricevitore ma prosegue;

Esistono poi altri modelli che rappresentano informazioni come il *messaggio di ritorno*, la *creazione o distruzione di oggetti*, ecc...

Vediamo quindi un esempio di due linee vita associate a *classi* che si scambiano messaggi in UML:

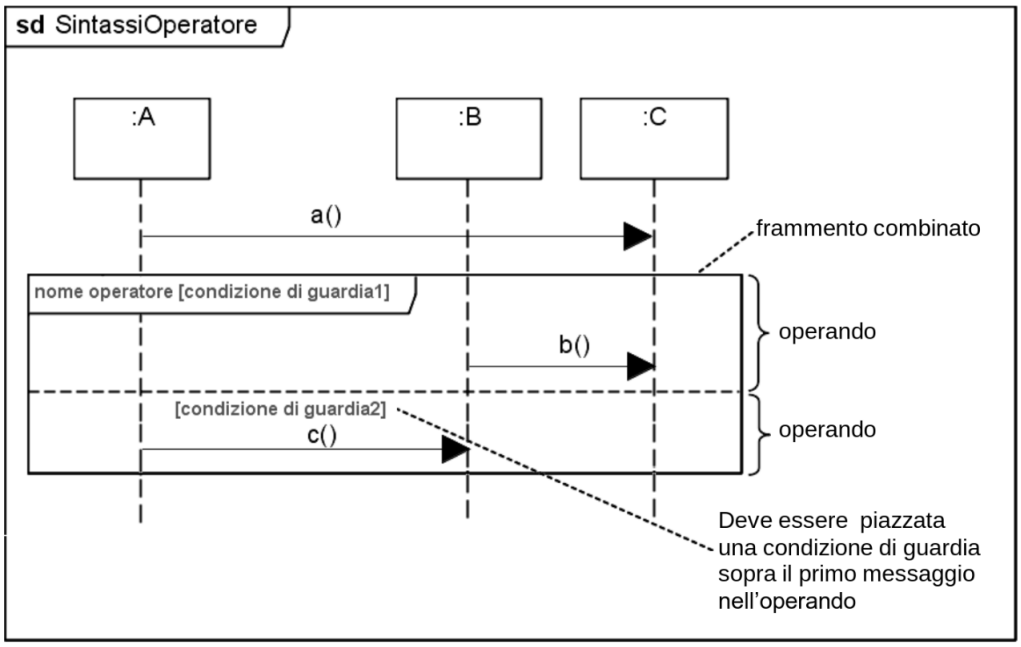


1.1.3 Frammenti combinati

I diagrammi di sequenza possono essere suddivisi in "aree" dette **frammenti combinati**. Ogni frammento combinato ha un operatore, uno o più operandi e zero o più condizioni di guardia.

L'**operatore** determina come gli operandi vengono eseguiti: solitamente gli operatori ricalcano i costrutti dei linguaggi strutturati o altre caratteristiche dell'operazioni che vengono svolte (ad esempio esecuzione *parallela*, *atomica*, ecc...).

La sintassi in UML sarà quindi la seguente:



Vediamo una tabella che riassume gli operatori principali:

Operatore	Nome	Significato
`opt`	opzione	Se la condizione è vera, viene eseguito un singolo operando
`alt`	alternative	L'operando la cui condizione è vera viene eseguito
`loop`	loop	Cicla max-min volte mentre la condizione è vera
`break`	break	Se la condizione di guardia è vera, l'operando è eseguito, ma non il resto dell'interazione
`ref`	riferimento	Il frammento combinato si riferisce ad un'altra interazione
`par`	parallelo	Tutti gli operandi vengono eseguiti in parallelo
`critical`	critico	L'operando viene eseguito atomicamente
`seq`	sequenza debole	Tutti gli operandi vengono eseguiti in parallelo con il vincolo che gli eventi che arrivano sulla stessa linea di vita da operandi differenti avvengano nella stessa sequenza degli operandi
`strict`	sequenza stretta	Gli operandi vengono eseguiti in stretta sequenza

L'operatore `loop` in particolare prevede più sintassi che rappresentano diversi tipi di iterazioni:

- Loop for, che possono essere numerati;
- Loop while, che possono essere infiniti o fino a condizione non soddisfatta;
- Loop for-each, che possono essere ripetuti su collezioni o classi.

1.1.4 Occorrenze di interazione

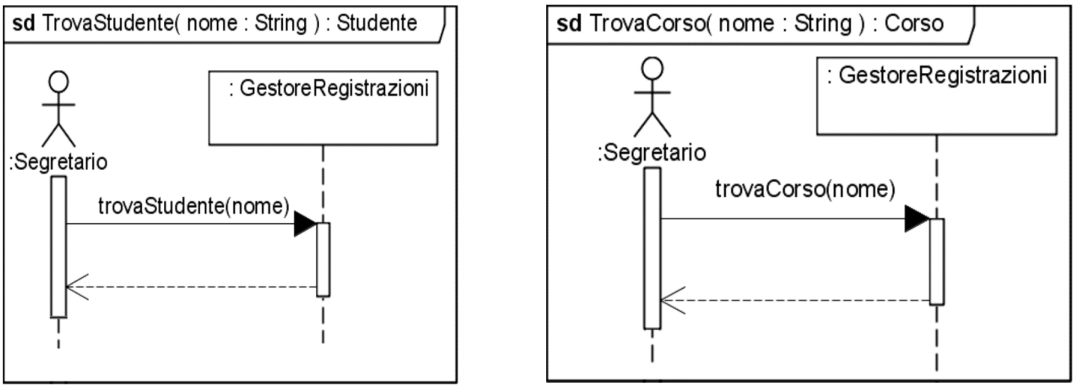
Per evitare di ripetere più volte gli stessi frammenti di interazione, l'UML mette a disposizione il riferimento ad interazione, detto **occorrenza di interazione** (che abbiamo visto come operatore `ref`).

Questo può essere usato per riferirsi a frammenti di interazione in altri diagrammi di interazione.

1.1.5 Parametri

Le interazioni possono essere **parametrizzate**. Questo permette di fornire valori diversi all'interazione ad ognuna delle sue occorrenze (meccanismo simile al passaggio di argomenti).

A scopo di esempio vediamo le interazioni parametrizzate *TrovaStudente* e *TrovaCorso* di un sistema di gestione univeristario:

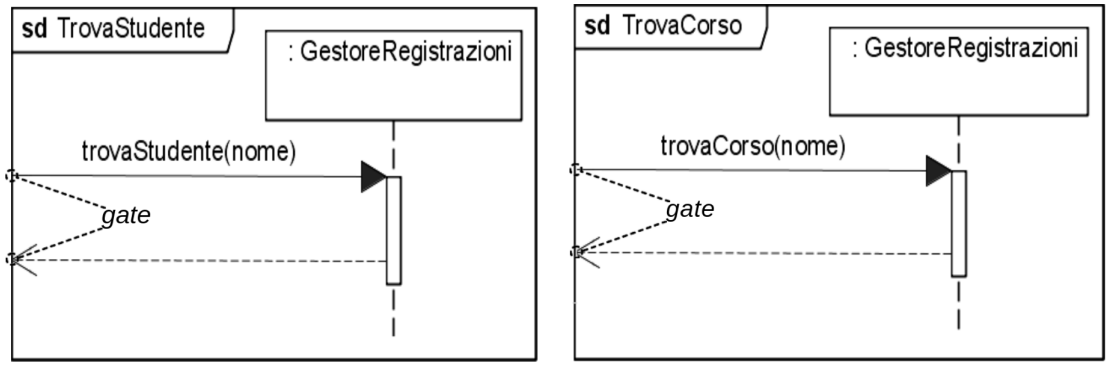


1.1.6 Gate

I *cancelli* (**gate**) rappresentano vie di accesso alle interazioni dall'esterno. Vengono quindi usati quando un'interazione è attivata da una linea di vita che non è parte dell'interazione stessa.

Vengono rappresentati graficamente come entrate ed uscite sulla finestra dell'interazione: questi punti connettono messaggi fuori dall'interazione a messaggi dentro l'interazione.

Sempre nel contesto dell'esempio sompra, vediamo come le interazioni viste possono essere realizzate anche coi gate:



Il meccanismo dei gate potrebbe sembrare simile a quello dei parametri. In sostanza si può dire che:

- I parametri rappresentano bene informazioni note all'interno dell'interazione;
- I gate rappresentano invece informazioni che provengono dall'esterno dell'interazione.

1.2 Workflow progetto

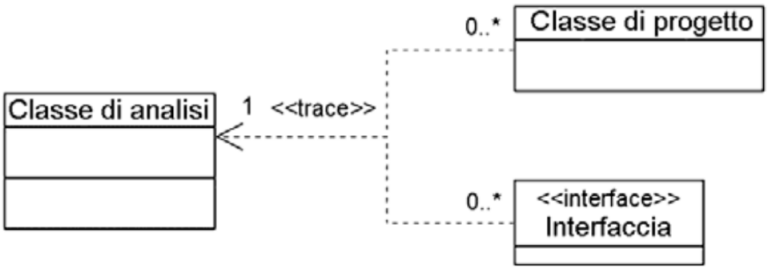
L'obiettivo del workflow **progetto** è approfondire la fase di *design* del sistema: qui si devono definire completamente tutte le funzionalità implementate per soddisfare i requisiti dell'utente.

Abbiamo studiando nel workflow *requisiti* come estrapolare i requisiti dal sistema dal dominio di sua applicazione; nel workflow *analisi* abbiamo esplorato il dominio del problema in modo da modellare strutture di classe e interazioni; adesso vogliamo tradurre questi modelli in soluzioni tecniche vere e proprie.

Abbiamo che questo workflow è tipico della fase di *elaborazione* e *costruzione* del software vero e proprio.

1.2.1 Classi di progetto

Di base, per ogni *classe di analisi* ci sono una o più **classi di progetto**, e opportunamente delle *interfacce*:



Queste non hanno più il solo scopo di modellare il dominio di problema, ma anche di essere realizzabili da programmatori veri e propri.

Abbiamo quindi che sono influenzate da:

- Il **dominio di problema**: come abbiamo visto raffinando le classi di analisi attraverso dettagli implementativi;
- Il **dominio della soluzione**: composto dall'insieme di *librerie*, *database*, *interfacce* e altre tecnologie che verranno adottate per l'implementazione vera e propria.

Campi e metodi delle classi di progetto devono essere completamente definiti (e implementabili). Per i campi, si può iniziare a specificare i tipi di variabile (se applicabile), per i metodi tipi di argomenti e ritorno. Opportunamente, è meglio dividere classi in più sottoclassi che *"ingigantire"* una classe fino a renderla inutilizzabile.

Una classe di progetto è ben definita quando risulta:

- **Completa** e **sufficiente**: le operazioni pubbliche della classe definiscono un *contratto* ben chiaro e definito con i clienti della stessa. Una classe è completa quando

offre ai suoi clienti tutto quello che i clienti si aspettano, senza operazioni in più (che non rientrano nel dominio di competenza della classe) e senza operazioni in meno (cosa che non onererebbe il contratto);

- **Primitiva**: le classi non dovrebbero implementare lo stesso comportamento più volte: dovrebbero fornire servizi singoli e atomici;
- **Alta coesione**: le classi dovrebbero modellare un singolo concetto astratto e contenere campi e metodi direttamente riconducibili a quel concetto astratto;
- **Basso accoppiamento**: le classi dovrebbero essere associate in interazioni con le sole classi con cui dividono *responsabilità*. Di base, non conviene usare metodi di altre classi solo perché il codice verrebbe altrimenti ripetuto: meglio non sacrificare la solidità strutturale per scrivere meno righe.

1.2.2 Ereditarietà

In analisi, si usa la relazione di erediterietà se si vuole modellizzare una relazione "is a" chiara e definita. In fase di progetto, l'ereditarietà scoperta in fase di analisi può essere usata per rafforzare la struttura delle classi e risparmiare codice per funzioni a queste comuni.

1.2.3 Classi annidate

Le classi annidate sono specifiche ad alcuni linguaggi come il Java. Solitamente sono sconsiliate in quanto non ci permettono di distinguere facilmente chi sta dando problemi fra la classe madre e la classe annidata.

1.2.4 Relazioni fra classi di progetto

Veniamo al problema di rendere le **relazioni** fra classi (cioè le *associazioni*) in un'implementazione in fase di progetto.

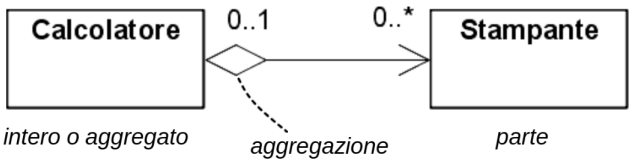
Partiamo dall'idea che nei linguaggi di progammazione che consideriamo (OOP standard, come Java o C++), non c'é nessun modo di rendere relazioni bidirezionali, o relazioni molti a molti.

Dobbiamo quindi seguire delle procedure per trasformare le associazioni di questo tipo in associazioni implementabili nel nostro dominio di soluzione.

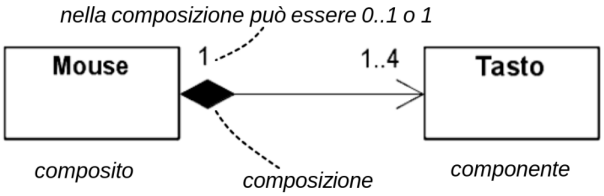
Queste procedure possono essere:

- Trasformare associazioni in relazioni di **aggregazione** o **composizione** dove appropriato:
  - La relazione di **aggregazione** è una relazione lieve fra oggetti, che lega una parte ad un'intero dove l'intero è composto da molte parti. Le parti dell'intero sono in qualche modo autosufficienti, ed esistono a priori di questo.

    Un esempio di aggregazione è la relazione fra un computer e le sue parti:

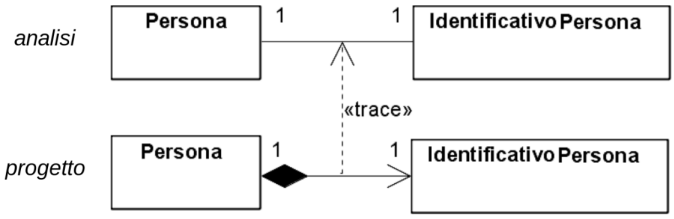


– La relazione di **composizione** è di contro una relazione molto forte, dove la parte esiste in funzione dell'intero. Ad esempio, in questo caso se muore la parte muore anche l'intero.
Un esempio di composizione è la relazione fra la parte del computer e i suoi componenti:

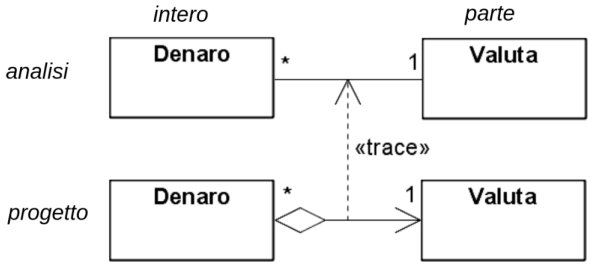


Chiaramente, bisogna distinguere prima chi nella relazione è la parte e chi l'intero, e poi risolvere problemi come capire le molteplicità (lato intero, da 0 a 1 si può comporre, oltre bisogna aggregare).

- Implementare associazioni uno-a-uno: questo si può fare sia per composizione che per aggregazione. Ad esempio:

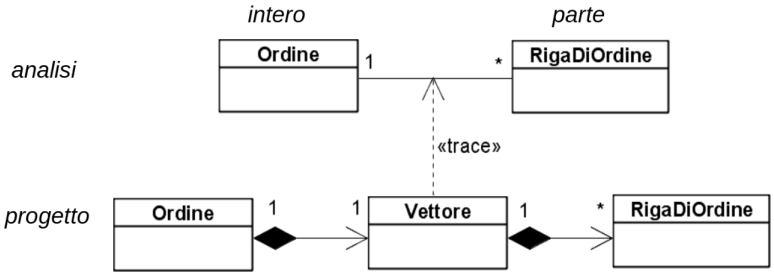


- Implementare associazioni molti-a-uno: questo si fa solitamente per aggregazione, prendendo le *molte* parti come aggregate all'*uno*. Ad esempio:



- Implementare associazioni uno-a-molti: qui bisogna usare una classe *collezione* esplicita, come ad esempio un vettore.

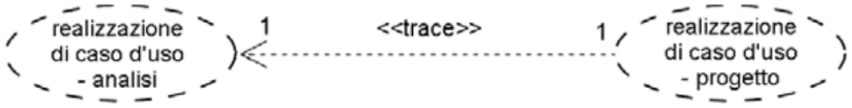
  Vediamo infatti il confronto fra la relazione molte a molti in fase di analisi alla sua implementazione in fase di progetto:



- Implementare associazioni molti-a-molti;
- Implementare associazioni bidirezionali;
- Implementare classi di associazioni.

1.3 Realizzazione casi d'uso di progetto

Come per le classi di analisi, anche i casi d'uso vanno rielaborati in **casi d'uso di progetto**:



Rispetto all'attività corrisondente in fase di analisi, abbiamo in fase di progetto alcune differenze:

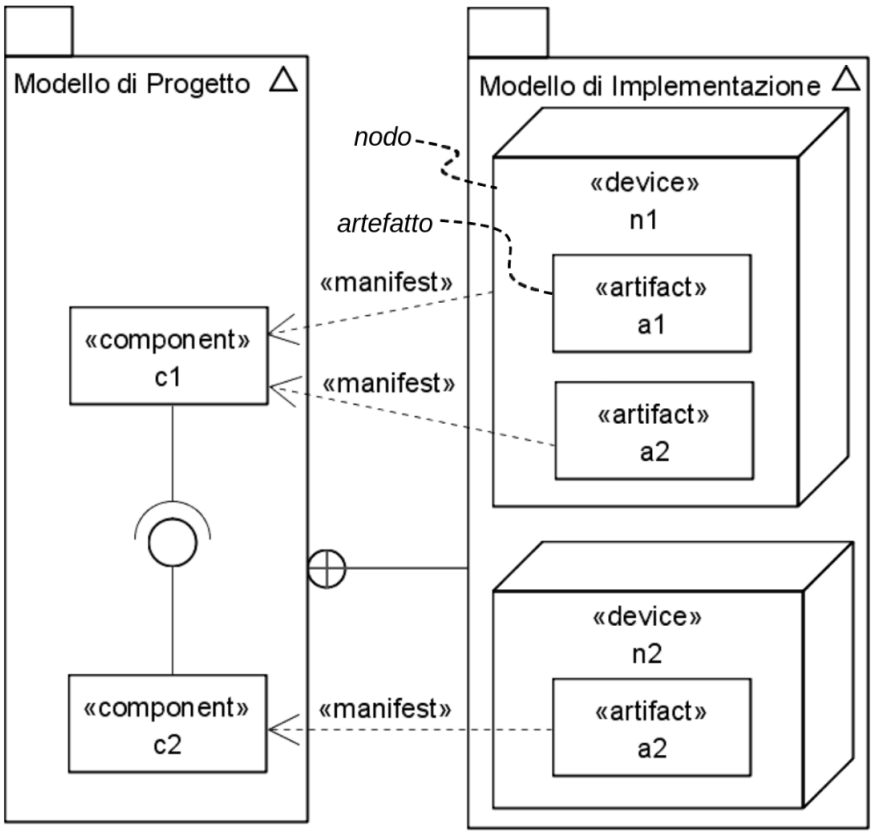
- La realizzazione dei casi d'uso coinvolge l'uso di di classi progetto, interfacce e componenti piuttosto che classi d'analisi;
- Il processo di creazione di realizzazione di casi d'uso di progetto scoprirà nuovi requisiti non funzionali e classi di progetto (atti ad interagire con il dominio della soluzione);
- Le realizzazioni dei casi d'uso aiutano a trovare i **meccanismi centrali**, cioè modi standard di risolvere un particolare problema di progetto che sono applicati consistemente nello sviluppo del sistema.

Una realizzazione di un caso d'uso nel progetto consiste in **diagrammi di interazione** e **diagrammi delle classi**.

1.4 Workflow implementazione

Veniamo quindi all'implementazione effettiva del sistema progettato in fase di progetto. Chiave di questo workflow è la scrittura di **codice**, **testing** estensivo di ogni singola funzionalità che si vuole produrre, e **dislocazione** su un'architettura reale del sistema sviluppato.

Vediamo un diagramma che spiega come vogliamo associare al modello realizzato in fase di progetto un *implementaione* vera e propria:



Nell'esempio, la relazione `<<manifest>>` significa che i componenti d'implementazione, appunto, *implementano* i relativi componenti della fase di progetto.

Solitamente nella fase di implementazione questa fase non è più necessario modellare ulteriori componenti. Eccezioni potrebbero essere le situazioni:

- Si vuole generare il codice direttamente dal modello;
- Se si sta organizzando uno sviluppo basato sui componenti (*Component Based Development*): in questo caso scegliere come disporre i componenti diventa un problema strategico.

Gli artefatti che vogliamo generare sono:

- **Diagrammi di componenti** che mostrano come gli artefatti rendono manifesti i componenti implementati;
- **Diagrammi di dislocazione** che rappresentano come gli artefatti vengono fisicamente dislocati su architetture reali per implementare il servizio richiesto.

1.5 Diagrammi di dislocazione

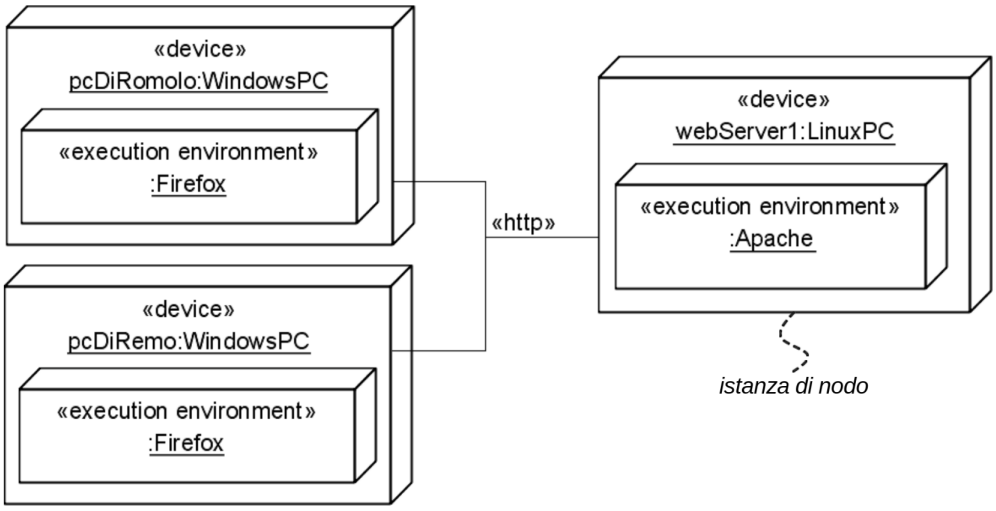
I diagrammi di dislocazione rappresentano come gli artefatti prodotto vengono, appunto, *dislocati* su istanze di nodi (macchine, ecc...). Questo processo è estremamente importante ad esempio nello sviluppo di architetture distribuite.

Esistono 2 tipi di diagrammi di dislocazione:

- **Descrittivi**: contiene nodi ed artefatti, e le relazioni che vi avvengono. Un nodo è una macchina, mentre un'artefatto è software in esecuzione su tale macchina;
- **D'istanza**: contengono istanze di nodi, di artefatti, e di relazioni. Qui le istanze di nodi sono macchine specifiche, le istanze di artefatti sono processi specifici, e così via.

  Comunque, è possibile usare nodi e artefatti anonimi quando il loro comportamento è noto a prescindere dalla specificità.

Vediamo quindi l'esempio di un diagramma di dislocazione, *d'istanza*, relativo ad un'applicazione distribuita:



1.5.1 Nodi

In dettaglio, i **nodi** rappresentano risorse computazionali che possono essere usate per realizzare il servizio. Ne distinguiamo 2 tipi:

- `<<device>>`: nodi che rappresentano dispositivi fisici;
- `<<execution environment>>`: nodi che rappresentano ambienti di esecuzione per software.

Le **associazioni** fra nodi e artefatti sono a questo punto canali di comunicazione dove informazioni possono essere scambiate.

1.5.2 Artefatti

Gli artefatti rappresentano lato software tutto ciò che viene effettivamente implementato, fra cui:

- File sorgenti;
- File eseguibili;
- Script;
- Tabelle di basi di dati;

- Documenti;
- Risultati del processo di sviluppo.

1.6 Workflow test

Veniamo quindi a discutere l'attività di **testing** che è opportuno applicare durante e dopo lo sviluppo.

Sappiamo che è impensabile sviluppare software privo di errori. Per questo dobbiamo:

- Testare il software spesso;
- Scrivere software che sia facile da testare.

1.6.1 Operazioni di test

La fase di testing si effettua vari livelli e consiste in operazioni distinte:

- Test di **unità**;
- Test di **modulo**;
- Test di **sottosistema**;
- Test di **sistema** o **integrazione**;
- Test **alfa**: prima fase dove si fornisce il sistema al cliente;
- Test **beta**: seconda fase, in cui si fornisce il sistema a più utenti;
- Test **benchmark**: il sistema viene testato in maniera standardizzata per valutarne le prestazioni;
- Test di **stress**: il sistema viene messo al prova in termini di resilienza, sicurezza, ecc... sopponendolo ad attachi o comunque stress.

1.6.2 Testing e debugging

Testing e debugging sono processi diversi fra di loro ma legati:

- Il **testing** serve a valutare la presenza di un errore;
- Il **debugging** è l'atto di risolvere tale errore.

I test di **regressione** sono utili dopo il debugging, per confermare che l'attività di debugging non abbia prodotto ulteriori errori.

1.6.3 Distribuzione di errori

Gli errori, come abbiamo detto, si verificano a prescindere dalle nostre migliori intenzioni. Possiamo adottare alcune euristiche per capire come sono distribiti:

- Gli errori tendono a concentrarsi in moduli **specifici**, solitamente i *più complessi*;
- Un modulo che contiene un errore probabilmente ne conterrà altri;

- Moduli strettamente accoppiati condivideranno più errori;
- Moduli che vengono rielaborati spesso (*refactoring* o *manutenzione*) sono più facilmente soggetti ad errori, sopratutto man di mano che il processo di refactoring si prolunga nel tempo.