Ingegneria DEL Software: introduzione a UML

Ingegneria del sw (Università di Bologna)

INGEGNERIA DEL SOFTWARE(2)
MODULO 2

ISTRUTTORE: GIANCARLO SUCCI

giancarlo.succi2@unibo.it

Programma del corso:

Programma del corso: 0
Premessa: 3
Sub-system and module: 3
Sub-system (Sottosistema): 3
Module (Modulo): 4
Information hiding : 4
Low coupling: (da rivedere!!) 5
Cohesion 6
Simplicity 7
Introduction to the Unified Modeling Language (UML) 8
Introduction to OO 8
Perché l'Approccio Orientato agli Oggetti (OO) è così Popolare: 8
Cos'è lo Sviluppo Software Orientato agli Oggetti: 9
Sviluppo di Sistemi Orientato agli Oggetti: 9
Metodologie Orientate agli Oggetti (OO): 10
Classi 14
Operations (a.k.a. Services) 15
Ereditarietà 16
Polymorphism 18
Various forms of Polymorphism 18
UML in Some Details 19
UML has many entities 19
UML diagrammi 20
Object Oriented Concept Modeling 20
Goals of OO Concept Modeling 20
Use Cases for OO Concept Modeling 21
What is a Use Case? 21
I fatti chiave sui casi d'uso includono: 22

La cattura del contesto..... 22
Actors..... 24
Extends relationship..... 24
Includes relationship..... 25
Comparing extends/includes..... 26
Textual description..... 26
Object Oriented Analysis..... 27
OOA- A Generic View..... 27
Extraction of Classes..... 28
Class diagram..... 28
We Have 3 Perspectives..... 28
Classi..... 29
Classi vs tipi..... 29
Associazioni..... 30
Diagrammi e classi..... 31
Naming associations..... 32
Roles..... 32
Multiplicity..... 33
Responsabilità..... 34
Navigability - Indicated by Arrow..... 35
Naming conventions..... 36
Association classes..... 36
Le classi e gli oggetti..... 37
Attributi..... 38
Difference between attributes and associations..... 39
Operations..... 39
Types of operations..... 40
Aggregation..... 40
Aggregation and composition..... 41
Generalization vs. Extension..... 42
Instantiation and generalization..... 42
Concept of generalization..... 43
Class Diagram with Inheritance..... 44
How to define classes (revised)?..... 44
To which class does an object belong?..... 44
Changing classes..... 45
Generalization: extension & restriction..... 45
Perspectives..... 45
Multiple inheritance..... 46
When to use class diagrams..... 48
Creating a class diagram..... 49
Introduction to the Unified Modeling Language (UML)Part 2..... 50
Object Oriented Design in UML..... 50
Object-Oriented Design & Design Issues..... 50

Ingredients for OOD..... 51
What distinguishes OOD from OOA?..... 51
Structural Diagrams for OOD in UML..... 52
Behavioral Diagrams for OOD in UML..... 53
Statechart diagrams vs Interaction diagrams..... 53
Object states..... 54
State changes..... 54
Example of Statechart Diagrams: States of a hockey game..... 55
Example of Statechart Diagrams (2): Order Management..... 57
Activity diagrams..... 60
Example of activity diagrams (the coffee pot)..... 61
Structure of activity diagrams..... 61
Interaction diagrams..... 62
Sequence diagrams..... 63
Asynchronous messages..... 64
Complexity and sequence diagrams..... 65
Where are the boundaries?..... 65
Collaboration diagrams..... 66
Comparing sequence & collaboration diagrams..... 67
Content of collaboration diagrams..... 67
Collaboration diagram example..... 68
A Comprehensive Example:The Elevator..... 69
Elevator -- Use Case..... 69
Elevator - First Class Diagram..... 69
Elevator -State Diagram..... 70
Elevator - Sequence Diagram..... 70
Elevator - Collaboration Diagram..... 70
System Design..... 71
Systems and Sub-Systems (Sistemi e Sotto-Sistemi):..... 71
Come suddividere un sistema in sottosistemi più piccoli?..... 72
Packages (Pacchetti):..... 72
Nested Packages (Pacchetti Nidificati):..... 74
Come può essere limitata la complessità di un'interfaccia di pacchetto?..... 74
Regole Pratiche:..... 75
Pacchetti vs. Diagrammi:..... 75
Uno Sguardo ai Design Patterns:..... 75
Components in UML..... 76
Stereotipi per i Componenti:..... 78
Nodi:..... 79
Components and Nodes..... 79
Real Time System Analysis and Design in UML..... 80
Costrutti Real-Time in UML:..... 81
Eventi Temporali e di Modifica:..... 81
Vincoli Temporali:..... 82

ROOM: Real Time Object Oriented Modeling..... 83
The Object Constraint Language (OCL).....83
Principali Caratteristiche di OCL :..... 84
Introduction to Refactoring.....85

Premessa:

Nel vasto panorama dell'Ingegneria del Software, ci sono alcuni concetti fondamentali che svolgono un ruolo cruciale nella progettazione e nello sviluppo di sistemi software robusti ed efficaci. Questi concetti ci guidano nella creazione di sistemi comprensibili, manutenibili e performanti. Nei seguenti appunti, esploreremo in dettaglio quattro di questi concetti chiave:

- **Sotto-sistema e Modulo.**
- **Nascondere le Informazioni (Information Hiding)**
- **Accoppiamento (Coupling).**
- **Coesione (Cohesion)**
- **Semplicità**

Sub-system and module:

Sub-system (Sottosistema):

Un sottosistema è una parte distintiva di un sistema più grande che svolge una specifica funzione o un sottoinsieme di responsabilità all'interno del sistema complessivo. Può essere affrontato da due approcci principali:

- **Approccio Procedurale:** In questo contesto, un sottosistema è spesso considerato come un insieme di procedure o funzioni che lavorano insieme per svolgere un compito specifico. Ad esempio, un sistema di gestione di database (DBMS) può essere considerato un sottosistema all'interno di un sistema di gestione aziendale più ampio.
- **Approccio Orientato agli Oggetti (OO):** In un contesto orientato agli oggetti, un sottosistema può essere rappresentato come un insieme di classi o oggetti che collaborano per eseguire determinate funzioni all'interno del sistema. Ogni classe all'interno del sottosistema ha responsabilità specifiche e interagisce con le altre classi quando necessario.

Un esempio pratico di sottosistema potrebbe essere un sistema di elaborazione degli errori all'interno di un'applicazione software. Questo sottosistema è responsabile della gestione degli errori e delle eccezioni che si verificano durante l'esecuzione del software.

Module (Modulo):

Un modulo è una suddivisione più specifica di un programma software, spesso associato a un linguaggio di programmazione particolare. Esso contiene un insieme di funzioni, procedure o classi che svolgono compiti specifici all'interno del software. Ecco come può essere affrontato in base a due approcci principali:

- **Approccio Procedurale**: In questo caso, un modulo è considerato come un insieme di funzioni o procedure che eseguono operazioni specifiche. Questo aiuta a organizzare il codice in unità funzionali. Ad esempio, un modulo in un linguaggio come C può contenere una serie di funzioni correlate che gestiscono l'input e l'output dei dati.
- **Approccio Orientato agli Oggetti (OO)**: In un contesto OO, un modulo può essere visto come un insieme di classi correlate che rappresentano oggetti e comportamenti specifici all'interno del software. Ad esempio, in un'applicazione di grafico, potrebbe esserci un modulo che contiene classi per disegnare forme geometriche.

I moduli sono spesso utilizzati per organizzare il codice in unità logiche e riutilizzabili, semplificando la manutenzione e la gestione del software. Sono una componente chiave della struttura di un programma e possono essere organizzati in una gerarchia o una struttura modulare per rendere il codice più chiaro ed efficiente.

Information hiding :

I moduli devono nascondere la loro implementazione interna: Un modulo dovrebbe celare i dettagli della sua implementazione interna agli altri moduli o componenti del sistema. Ciò significa che le informazioni specifiche su come il modulo svolge il suo compito dovrebbero essere nascoste, e solo un'interfaccia pubblica dovrebbe essere accessibile agli altri componenti del sistema.

Il modulo deve essere accessibile solo tramite un'interfaccia pubblica: L'accesso al modulo dovrebbe avvenire solo attraverso un'interfaccia pubblica definita. Gli altri moduli non dovrebbero avere accesso diretto ai dettagli interni del modulo, ma dovrebbero interagire con esso solo attraverso i metodi e le funzioni pubbliche fornite dall'interfaccia.

Nessun accesso diretto ai dati interni e ai metodi privati: I dettagli interni del modulo, compresi i dati e i metodi privati, non dovrebbero essere accessibili o modificabili direttamente da altri moduli. Questi dettagli dovrebbero essere protetti e resi inaccessibili dall'esterno.

I dati sono accessibili attraverso un insieme ben definito di metodi accessor: Se il modulo deve fornire accesso ai dati interni, dovrebbe farlo attraverso un insieme limitato e ben definito di metodi accessor. Questi metodi permettono agli altri moduli di accedere ai dati in modo controllato e sicuro.

Utilizzare l'astrazione per definire moduli e interfacce: L'astrazione è un concetto chiave nella progettazione dei moduli. Gli oggetti e i metodi dovrebbero essere progettati in modo

astratto in modo che siano indipendenti dai dettagli implementativi. Ciò consente una maggiore flessibilità e facilità di manutenzione.

Cambiare l'implementazione (a condizione che non vi sia alcuna modifica all'interfaccia) non dovrebbe avere alcun effetto sul resto del sistema: Una volta definita un'interfaccia pubblica stabile per il modulo, le modifiche all'implementazione interna del modulo dovrebbero essere possibili senza influenzare gli altri moduli o il sistema nel suo complesso. Questo principio di progettazione promuove la modularità e la manutenibilità del codice.

Low coupling: (da rivedere!!)

Nell'ingegneria del software, l'accoppiamento si riferisce al grado di dipendenza o interdipendenza tra i moduli o le componenti di un sistema software. Questo concetto è fondamentale per la progettazione e lo sviluppo software, poiché un'accoppiamento eccessivo può rendere un sistema complesso, difficile da mantenere e soggetto a errori. D'altra parte, un basso accoppiamento promuove la modularità, la manutenibilità e la flessibilità del sistema.

Debolmente Accoppiato vs. Fortemente Accoppiato: Un sistema con moduli debolmente accoppiati ha interconnessioni e dipendenze deboli, il che è auspicabile. Al contrario, un sistema fortemente accoppiato implica una forte interdipendenza tra i moduli, rendendo il sistema più complesso.

Aumento dell'Ordine di Accoppiamento: Aumentare l'ordine di accoppiamento significa aumentare il grado di dipendenza tra moduli, il che può portare a una maggiore complessità.

Chiamate di Metodi tra Moduli: I moduli possono comunicare tra loro tramite chiamate di metodi o funzioni, creando un certo livello di accoppiamento.

Accoppiamento dei Dati vs. Accoppiamento di Controllo: L'accoppiamento dei dati si verifica quando un modulo passa dati a un altro modulo, mentre l'accoppiamento di controllo si verifica quando un modulo controlla il comportamento di un altro modulo.

Ereditarietà tra Moduli: La gerarchia delle classi o degli oggetti all'interno dei moduli può influenzare l'accoppiamento.

Dipendenza da Funzionalità Specifiche del Compilatore o del Sistema Operativo: La dipendenza da funzionalità specifiche del compilatore o dal sistema operativo può rendere il modulo meno portabile.

Accoppiamento I/O, Accoppiamento Comune, Accoppiamento del Contenuto: Questi sono tipi specifici di accoppiamento che descrivono come i moduli possono dipendere l'uno dall'altro.

In conclusione, l'obiettivo principale nella progettazione del software è ridurre al minimo l'accoppiamento tra i moduli per migliorare la modularità e la manutenibilità del sistema. La comprensione di questi concetti è essenziale per sviluppare software di alta qualità.

Cohesion

La coesione è un concetto chiave nella progettazione dei moduli e si riferisce al grado in cui gli elementi all'interno di un modulo sono strettamente correlati e diretti verso l'esecuzione di una singola funzione o compito. Un modulo altamente coeso è uno in cui tutti i suoi elementi lavorano insieme per svolgere una specifica attività o funzione. Una coesione elevata è auspicabile poiché rende i moduli più chiari, facili da mantenere e riusabili.

Tipi di Coesione dei Moduli:

- **Coesione Coincidentale (Coincidental Cohesion):** In questo caso, gli elementi del modulo sono raggruppati insieme senza una vera relazione o motivo. Questa è la forma più bassa di coesione ed è da evitare poiché rende difficile la comprensione e la manutenzione del codice.
- **Coesione Logica (Logical Cohesion):** Gli elementi del modulo sono logicamente correlati e condividono uno stesso contesto o scopo. Non ci sono altre interazioni tra di loro. Questo è un livello di coesione migliore rispetto a quello coincidentale, ma può essere ulteriormente migliorato.
- **Coesione Temporale (Temporal Cohesion):** Gli elementi del modulo vengono eseguiti entro lo stesso limite di tempo o in sequenza cronologica. Anche se possono avere scopi diversi, sono eseguiti insieme. Ad esempio, un modulo che gestisce gli eventi di avvio e spegnimento di un sistema.
- **Coesione Procedurale (Procedural Cohesion):** Gli elementi del modulo sono organizzati in modo che il controllo del flusso passi da uno all'altro. Questo è un livello di coesione più elevato poiché le parti lavorano insieme per raggiungere un obiettivo comune.
- **Coesione Comunicazionale (Communicational Cohesion):** Gli elementi del modulo sono coesi perché condividono lo stesso input o output, ad esempio, elementi che condividono le stesse operazioni di input/output.
- **Coesione Sequenziale (Sequential Cohesion):** L'output di un elemento del modulo è l'input del successivo. Questo indica che le parti sono collegate in sequenza, ciascuna elabora i dati provenienti dalla precedente.
- **Coesione Informativa (Informational Cohesion):** Gli elementi del modulo accedono alla stessa struttura dati o ai dati condivisi. Sono coesi perché lavorano con le stesse informazioni.
- **Coesione Funzionale (Functional Cohesion):** Questo è il più alto livello di coesione. Gli elementi del modulo sono diretti verso l'esecuzione di una singola funzione o compito ben definito. Tutti gli elementi contribuiscono al raggiungimento di questo obiettivo.

In sintesi, l'obiettivo nella progettazione dei moduli è massimizzare la coesione, preferendo la coesione funzionale e informativa , e minimizzare l'accoppiamento tra i moduli per garantire che il software sia chiaro, manutenibile ed efficiente.

Simplicity

Nell'ambito dello sviluppo software, la costruzione e la semplicità del codice sono fondamentali per la creazione di sistemi software efficaci e sostenibili. Questi concetti si concentrano sulla realizzazione di codice pulito, manutenibile e facile da comprendere, riducendo al minimo la complessità e l'inefficienza.

Costruzione del Codice:

Una delle prime linee guida è quella di costruire solo il codice necessario per risolvere i problemi attuali, evitando di prevedere eccessivamente i bisogni futuri. Questo approccio promuove l'efficienza e la chiarezza del codice, evitando la sovra-complessità.

Inoltre, il refactoring è un processo cruciale. Questo coinvolge la ristrutturazione di un sistema software funzionante per renderlo più semplice e manutenibile. Il refactoring aiuta a migliorare la qualità del codice senza modificarne il comportamento.

Semplicità a Diversi Livelli:

La semplicità deve essere considerata a diversi livelli di progettazione del software:

- A livello del metodo, si mira a creare metodi brevi con firme di metodo compatte, migliorando la comprensibilità e la manutenibilità.
- A livello di modulo, si cerca di mantenere un'interfaccia pubblica semplice, composta da pochi metodi ben definiti. La riduzione del numero di metodi pubblici semplifica l'utilizzo del modulo.
- A livello di sistema, la semplicità implica l'evitare l'uso di moduli intermedi superflui e variabili globali. È importante ridurre al minimo i percorsi informativi e di controllo all'interno del sistema, mantenendo le gerarchie di ereditarietà piccole e prevenendo la duplicazione del codice.
- A tutti i livelli, evitare la duplicazione dei codice.

In conclusione, la costruzione mirata del codice e la ricerca della semplicità sono pratiche chiave nell'ingegneria del software. Questi approcci aiutano a creare software di alta qualità, facilitando la manutenzione e la comprensione del codice, oltre a migliorare la sua efficienza e robustezza.

Introduction to the Unified Modeling Language (UML)

These lectures are NOT a comprehensive review of UML (it would take 6 months full time … :)

Introduction to OO

Perché l'Approccio Orientato agli Oggetti (OO) è così Popolare:

L'approccio Orientato agli Oggetti (OO) è uno dei paradigmi di programmazione più diffusi e popolari nell'ingegneria del software. Ci sono diverse ragioni che spiegano la sua popolarità:

- **Incremento della Produttività:** Molte organizzazioni e sviluppatori credono che l'OO possa aumentare la produttività nello sviluppo del software. Questo perché l'OO favorisce la modularità e la riutilizzabilità del codice, consentendo ai team di lavorare in modo più efficiente e di gestire complessità crescenti.
- **Strutturazione Naturale del Mondo:** L'OO è ispirato alla struttura del mondo reale, dove oggetti interagiscono tra loro e con il loro ambiente. Questo approccio offre un modo intuitivo e naturale di rappresentare il mondo all'interno del software. Ad esempio, gli oggetti in un sistema di gestione di una biblioteca possono essere libri, utenti, librerie, e le interazioni tra di essi rispecchiano le interazioni reali.
- **Oggetti:** Nel paradigma OO, il codice è organizzato attorno agli oggetti, che sono rappresentazioni dei concetti o degli elementi del mondo reale. Questo rende il software più comprensibile e manutenibile, in quanto il codice si allinea più strettamente con il dominio del problema.
- **Messaggi:** Nel mondo OO, gli oggetti comunicano tra loro attraverso messaggi. Questo paradigma di comunicazione è concettualmente simile alla comunicazione tra oggetti reali, rendendo più naturale l'interazione tra le componenti del software.
- Responsabilità: L'OO promuove la definizione chiara delle responsabilità degli oggetti. Ogni oggetto ha un ruolo ben definito all'interno del sistema e si occupa solo delle operazioni pertinenti. Questo contribuisce a ridurre la complessità e a facilitare la manutenzione.

In sintesi, l'OO è popolare perché si basa su principi intuitivi e naturali, struttura il software in modo modulare e favorisce la riutilizzabilità del codice. Questi aspetti contribuiscono all'efficienza nello sviluppo e alla creazione di sistemi software che rispecchiano meglio il mondo reale.

Cos'è lo Sviluppo Software Orientato agli Oggetti:

Lo sviluppo software orientato agli oggetti (Object-Oriented Software Development) è un approccio alla progettazione e allo sviluppo del software che si basa su concetti chiave legati agli oggetti. Ecco una spiegazione dei principali aspetti dell'OO nello sviluppo software:

- **Una Visione del Mondo dell'Applicazione:** Nell'OO, il software è concepito come una rappresentazione astratta del mondo reale. Gli oggetti nel software rappresentano entità, concetti o elementi del mondo reale, e le interazioni tra questi oggetti riflettono le relazioni e le dinamiche presenti nel dominio dell'applicazione.
- **Una Descrizione di un Modello dell'Applicazione:** L'OO fornisce un modo per descrivere il modello dell'applicazione mediante la definizione di classi, oggetti e relazioni tra di essi. Questo modello consente agli sviluppatori di comprendere e rappresentare in modo chiaro la struttura e il comportamento del sistema software.
- **Una Metodologia Completa:** L'OO è una metodologia completa che guida l'intero processo di sviluppo del software. Questa metodologia comprende fasi come l'analisi dei requisiti, la progettazione, l'implementazione e la manutenzione. Ciò significa che gli stessi concetti e principi orientati agli oggetti sono applicati in tutte le fasi del ciclo di vita del software. Permette **di sviluppare sistemi software complessi**. Gli oggetti vengono creati per rappresentare le parti del sistema e le interazioni tra di essi sono gestite in modo da implementare la logica dell'applicazione.Inoltre **i concetti simili vengono utilizzati in tutto il processo di sviluppo**. Ad esempio, la gerarchia di ereditarietà utilizzata nella progettazione delle classi è applicata sia nella definizione dei requisiti che nell'implementazione effettiva del software.

In sintesi, lo sviluppo software orientato agli oggetti è un approccio completo che si basa sulla rappresentazione del mondo reale attraverso oggetti e offre una metodologia coerente per progettare, sviluppare e mantenere sistemi software. Questo paradigma favorisce la comprensibilità, la manutenibilità e la riutilizzabilità del codice, rendendolo uno dei paradigmi più diffusi nell'ingegneria del software.

Sviluppo di Sistemi Orientato agli Oggetti:

Lo sviluppo di sistemi orientato agli oggetti è una strategia che mira a raggiungere l'alta qualità del software attraverso l'applicazione dei seguenti concetti chiave:

- **Nascondere le Informazioni (Information Hiding):** Questo concetto consiste nell'isolare i dettagli interni di un modulo o di un oggetto, consentendo solo un accesso controllato attraverso un'interfaccia pubblica. Ciò promuove la sicurezza e la stabilità del sistema, in quanto gli utenti esterni non sono influenzati dai cambiamenti interni.
- **Astrazione (Abstraction):** L'astrazione implica la creazione di rappresentazioni concettuali di entità del mondo reale, trascurando i dettagli non rilevanti per un

determinato contesto. Questo concetto aiuta a semplificare la comprensione e la gestione dei sistemi complessi.

- **Modularizzazione:** La modularizzazione consiste nell'organizzare il software in moduli o componenti autonomi, ognuno dei quali svolge un ruolo specifico all'interno del sistema. Questa suddivisione favorisce la manutenibilità, la facilità di test e la riutilizzabilità del codice.
- **Riuso (Reuse):** L'approccio orientato agli oggetti promuove il riuso del codice. Gli oggetti e le classi possono essere progettati in modo da essere riutilizzabili in diversi contesti, riducendo così lo sforzo di sviluppo e migliorando la coerenza del sistema.

Un approccio orientato agli oggetti, in modo più o meno esplicito, incoraggia gli sviluppatori a utilizzare questi concetti per ottenere un software di alta qualità. Questo paradigma favorisce la creazione di sistemi comprensibili, manutenibili ed efficienti, spingendo gli sviluppatori a pensare in modo astratto, modulare e a considerare il riuso del codice come una pratica fondamentale.

Metodologie Orientate agli Oggetti (OO):

Nella fine degli anni '80 e l'inizio degli anni '90, emersero diverse metodologie orientate agli oggetti per la progettazione e lo sviluppo di software. Queste metodologie differivano nelle notazioni e nei processi, ma avevano in comune l'adozione dei principi dell'orientamento agli oggetti. Le principali metodologie che emersero in quel periodo includevano:

- **Metodologia Booch:** Michael Booch è stato uno dei pionieri nell'adozione dell'orientamento agli oggetti nella progettazione del software. La sua metodologia Booch si basava su un approccio strutturato alla progettazione del software, enfatizzando la chiarezza e la comprensibilità del modello. Booch ha sviluppato notazioni grafiche per rappresentare oggetti, classi e relazioni.
- **Metodologia Rumbaugh:** Grady Rumbaugh ha contribuito in modo significativo allo sviluppo delle metodologie orientate agli oggetti. La sua metodologia era nota per l'approccio basato su modelli e la notazione grafica chiara. Rumbaugh ha creato il metodo OMT (Object Modeling Technique), che è stato uno dei precursori dell'Unified Modeling Language (UML).
- **Metodologia Jacobson:** Ivar Jacobson ha sviluppato una metodologia conosciuta come OOSE (Object-Oriented Software Engineering). Questo metodo si basava sull'identificazione dei casi d'uso (use case) per modellare il comportamento del sistema. Jacobson ha contribuito in modo significativo all'evoluzione delle pratiche di progettazione orientata agli oggetti.
- **Unified Modeling Language (UML):** L'UML è diventato uno standard de facto per la modellazione orientata agli oggetti. È stato sviluppato da un gruppo di esperti, tra cui Booch, Rumbaugh e Jacobson, con l'obiettivo di unificare le diverse metodologie e notazioni in un unico linguaggio di modellazione. L'UML fornisce una serie di diagrammi (come diagrammi delle classi, diagrammi dei casi d'uso e diagrammi delle sequenze) per rappresentare vari aspetti del sistema software.

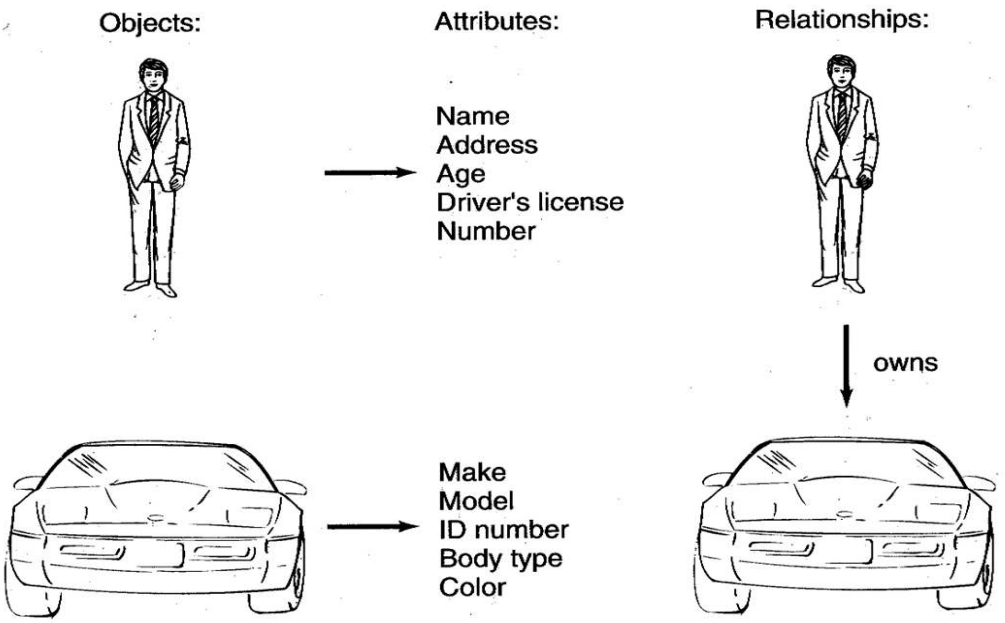
Queste metodologie hanno contribuito in modo significativo a promuovere l'adozione dell'orientamento agli oggetti nell'ingegneria del software. Successivamente, l'UML è emerso

come uno standard ampiamente accettato per la modellazione orientata agli oggetti, fornendo un linguaggio comune per i professionisti del software.

La "Key Idea" dietro alle OO methodology si basa sulla rappresentazione del mondo reale in termini di **oggetti interagenti**, e questa rappresentazione è utilizzata in tutte le fasi del ciclo di sviluppo del software, che comprendono:

- **Modellazione dei Concetti OO (OO Concept Modeling):** In questa fase iniziale del ciclo di sviluppo, si cerca di rappresentare concetti e oggetti del mondo reale che sono rilevanti per il problema da risolvere. Questo processo coinvolge l'identificazione delle classi di oggetti, delle loro proprietà (attributi) e dei loro comportamenti (metodi). Ad esempio, se stai sviluppando un'applicazione per una libreria, le classi potrebbero includere "Libro," "Cliente," "Prestito," e così via.
- **Analisi Orientata agli Oggetti (OO Analysis):** In questa fase, si analizza come questi oggetti interagiscono tra loro e si identificano le relazioni tra di essi. Ad esempio, si definiscono le associazioni tra classi, le gerarchie di ereditarietà, e si catturano i requisiti del sistema in termini di oggetti e comportamenti. L'obiettivo è creare un modello concettuale del sistema che rifletta accuratamente la realtà.
- **Progettazione Orientata agli Oggetti (OO Design)**:Questa fase si concentra sulla creazione di una struttura di sistema dettagliata basata sul modello concettuale sviluppato nell'analisi. Gli oggetti vengono suddivisi in classi, vengono progettati i dettagli delle interfacce e dei metodi, e vengono prese decisioni sulla gestione dei dati e sulla logica del sistema. Il risultato è un progetto tecnico che guida l'implementazione pratica del software.
- **Programmazione Orientata agli Oggetti (OO Programming):**Nella fase di programmazione, si traduce il modello di progettazione in codice sorgente effettivo. Le classi definite durante l'analisi e la progettazione vengono implementate e gli oggetti interagiscono nel codice per fornire la funzionalità desiderata.

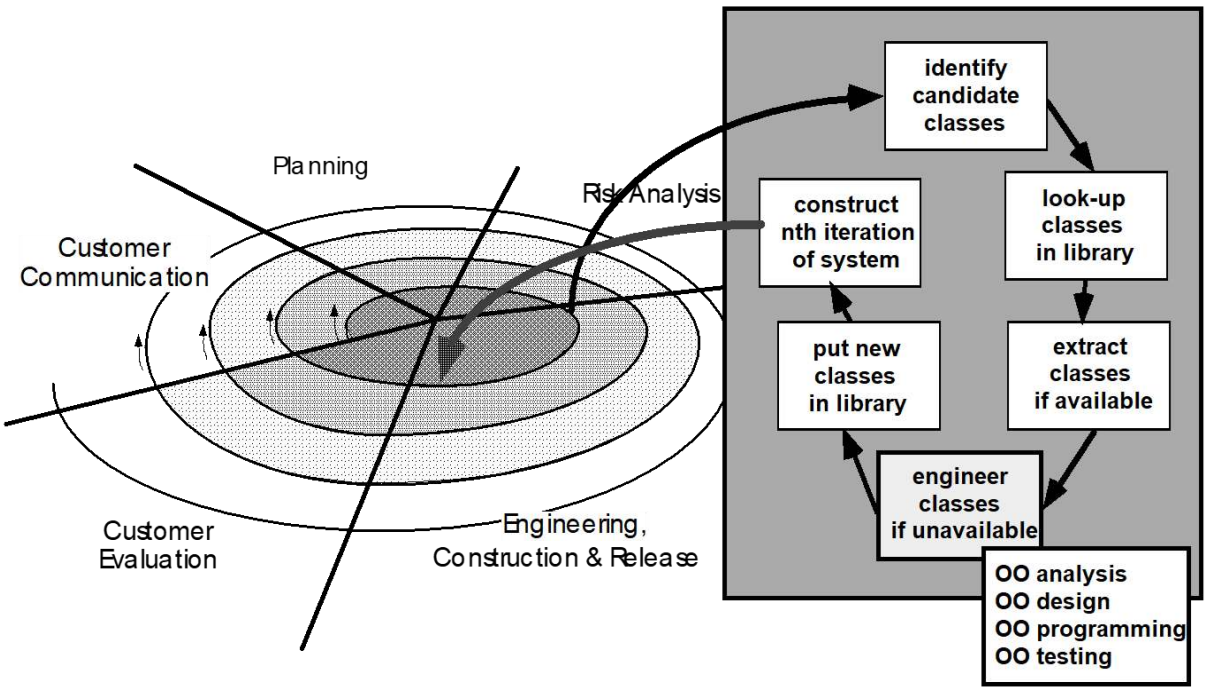
A Simple OO Model



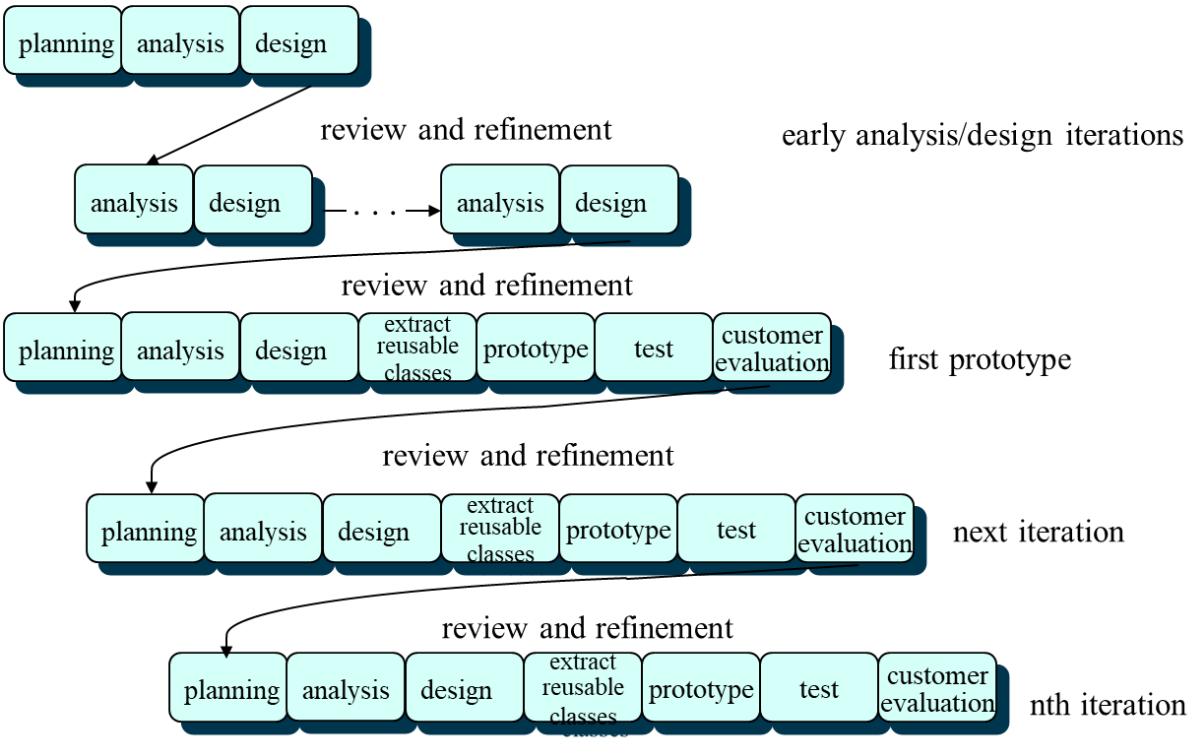
L'approccio orientato agli oggetti (OO) è basato su una serie di concetti chiave che costituiscono la struttura fondamentale per la progettazione e lo sviluppo dei sistemi software. Questi concetti comprendono:

- **classes and class hierarchies**
    - **Classi**:Una classe è un costrutto fondamentale nell'OO ed è la base per la creazione di oggetti. Una classe definisce le caratteristiche e i comportamenti comuni a un gruppo di oggetti correlati. Ad esempio, se stiamo sviluppando un'applicazione per una biblioteca, una classe potrebbe essere "Libro," che definisce tutti gli attributi (titolo, autore, ISBN) e i metodi (prestito, restituzione) associati ai libri.
    - **Gerarchie di Classi:**Le gerarchie di classi consentono di organizzare le classi in una struttura gerarchica, in cui le classi più specifiche ereditano attributi e metodi dalle classi più generiche. Questo concetto è noto come ereditarietà ed è essenziale per la creazione di classi specializzate basate su classi più generiche. Ad esempio, una classe "Romanzo" potrebbe ereditare da "Libro," acquisendo così tutti gli attributi e i metodi di "Libro" oltre a quelli specifici dei romanzi.
    - **Attributi:**Gli attributi rappresentano le caratteristiche degli oggetti e definiscono lo stato dell'oggetto stesso. Ad esempio, un attributo di un oggetto "Libro" potrebbe essere "titolo" o "autore." Gli attributi vengono utilizzati per memorizzare dati specifici dell'oggetto
    - **Metodi:**I metodi rappresentano i comportamenti degli oggetti e definiscono le azioni che un oggetto può eseguire. Ad esempio, un metodo di un oggetto "Libro" potrebbe essere "prestito()" o "restituzione()." I metodi definiscono come gli oggetti rispondono alle richieste o alle azioni.

    - **Ereditarietà**:La ereditarietà è un concetto chiave che consente alle classi di ereditare attributi e metodi da altre classi. Questo permette di creare classi specializzate che condividono le caratteristiche di classi più generiche. Le classi specializzate possono quindi aggiungere o modificare attributi e metodi in base alle loro esigenze specifiche.
    - **Relazioni con Altre Classi:**Le classi possono avere relazioni con altre classi, come associazioni, aggregazioni o composizioni. Queste relazioni consentono di modellare come le classi interagiscono tra loro all'interno del sistema.
- **Oggetti: Istanze di Classi:**Gli oggetti sono le istanze delle classi. Quando viene creata un'istanza di una classe, si ottiene un oggetto con i suoi attributi e i metodi associati. Ad esempio, se hai una classe "Libro," un oggetto "Libro" potrebbe rappresentare un libro specifico nella biblioteca.
    - **Attributi con Valori Assegnati:**Gli attributi degli oggetti possono avere valori assegnati che riflettono lo stato dell'oggetto. Ad esempio, un attributo "titolo" di un oggetto "Libro" avrà un valore specifico, come "Il Signore degli Anelli."
    - **Relazioni Istanziabili:**Le relazioni tra oggetti sono basate sulle relazioni tra le classi. Quando gli oggetti interagiscono, scambiano messaggi tra loro per invocare metodi e ottenere risposte alle azioni richieste.
- **Messaggi e Metodi per Rispondere a un Messaggio:**Nel mondo orientato agli oggetti, gli oggetti comunicano tra loro scambiando messaggi. Questi messaggi sono interpretati dagli oggetti come richieste per eseguire determinati metodi. Ad esempio, un oggetto "Libro" potrebbe ricevere un messaggio "prestito()" e risponderà eseguendo il metodo associato per registrare il prestito del libro.



Processo Tipico per un Progetto Orientato agli Oggetti



Classi

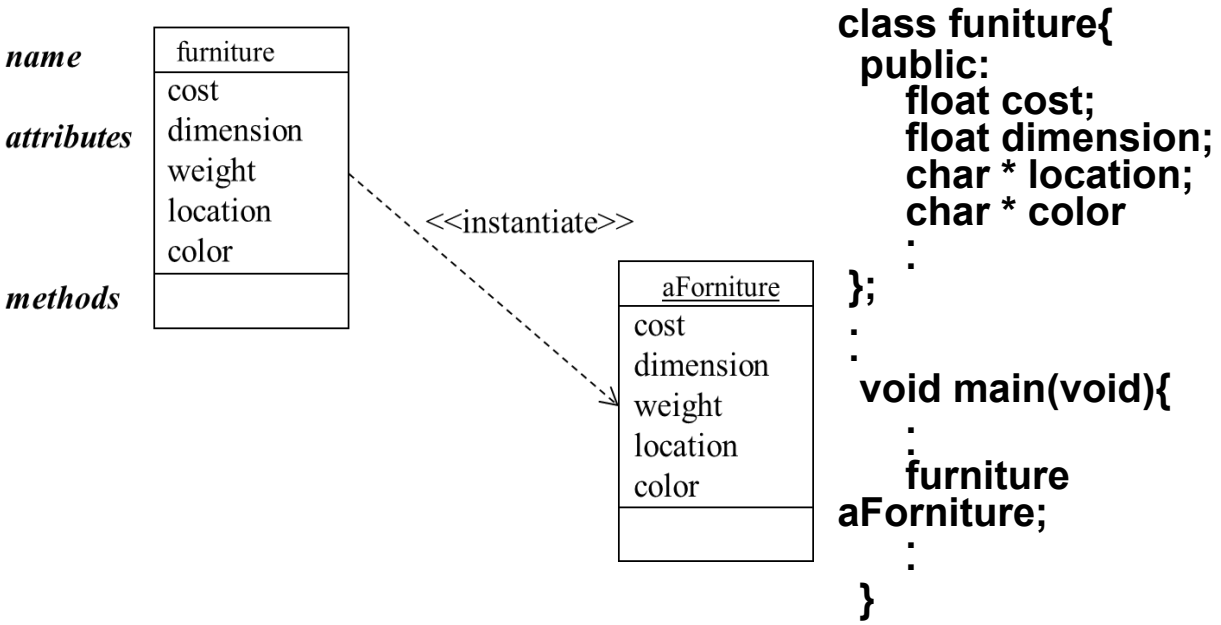
Una classe è una rappresentazione astratta e concettuale di un insieme di oggetti simili. Può essere definita come un:

- Modello
- Descrizione generalizzata
- Pattern
- “blueprint" che descrive una collezione di oggetti simili

Una classe definisce le caratteristiche comuni a tutti gli oggetti che vi appartengono, noti come **attributi**. Questi attributi rappresentano le proprietà o le caratteristiche degli oggetti e definiscono lo stato degli oggetti stessi. Inoltre, una classe definisce i comportamenti che tutti gli oggetti della classe possono eseguire, noti come **operazioni**. Queste operazioni rappresentano le azioni o i metodi che gli oggetti possono compiere.

Una volta che una classe è stata definita con i suoi attributi e le sue operazioni, è possibile creare un'istanza specifica o un oggetto della classe stessa. Questo processo consiste nell'applicare il modello astratto definito dalla classe per creare un'entità con attributi e comportamenti specifici. Ad esempio, se abbiamo una classe "Automobile," possiamo creare un'istanza specifica di un'automobile, come una "Fiat Punto," che eredita le caratteristiche generali della classe ma ha dettagli unici come il numero di targa o il chilometraggio.

C++ CODE



Operations (a.k.a. Services)

Un'operazione, all'interno del contesto orientato agli oggetti (OO), è una procedura eseguibile che è incapsulata all'interno di una classe e progettata per agire su uno o più attributi di dati definiti come parte della stessa classe. Questa operazione è strettamente associata alla classe e opera sugli oggetti di quella classe. Vediamo i dettagli di questa definizione:

Un'operazione rappresenta una sequenza di istruzioni o un blocco di codice che può essere eseguito quando richiamato. Questo codice è incapsulato all'interno della classe e può svolgere un'ampia gamma di compiti, dalla manipolazione dei dati alla gestione di comportamenti specifici.

L'operazione è parte integrante di una classe, il che significa che è definita all'interno del contesto di quella classe. Questa caratteristica di incapsulazione consente di organizzare il codice in modo ordinato e di raggruppare operazioni correlate con gli attributi della classe.

L'operazione è progettata per lavorare su uno o più attributi di dati che sono specifici per la classe. Gli attributi sono le variabili che memorizzano le informazioni o lo stato dell'oggetto e l'operazione può modificarli o accedere a essi per svolgere il suo compito.

Nell'ambito dell'OO, le operazioni vengono invocate attraverso il passaggio di messaggi tra oggetti. Un oggetto invia un messaggio a un altro oggetto per richiedere l'esecuzione di una determinata operazione. Questo approccio di "messaggio-passing" permette agli oggetti di collaborare e comunicare tra loro.

Il termine "operazione" ha vari sinonimi a seconda del linguaggio di programmazione o del contesto. Alcuni di questi includono "servizio," "funzione entry" (nell'ambito di Concurrent Pascal), "member function" (in C++), e "metodo" (in linguaggi orientati agli oggetti come Java o Python). Indipendentemente dal nome specifico utilizzato, queste entità svolgono la stessa funzione di definire e incapsulare comportamenti specifici all'interno delle classi OO.

Ereditarietà

L'ereditarietà è una caratteristica fondamentale nell'orientamento agli oggetti (OO) che consente di definire classi che estendono altre classi esistenti, introducendo nuovi attributi e/o metodi specializzati. Questo concetto di ereditarietà è essenziale per la creazione di gerarchie di classi e per la definizione di relazioni tra classi. Vediamo in dettaglio questa definizione:

L'ereditarietà permette di creare nuove classi (classi figlie o derivate) che estendono o ereditano caratteristiche da altre classi esistenti (classi genitori o base). Ad esempio, una classe "Cane" può ereditare da una classe "Animale," il che significa che la classe "Cane" acquisisce tutti gli attributi e i metodi definiti nella classe "Animale."

Le classi figlie hanno la flessibilità di ridefinire (sovrascrivere) attributi e metodi ereditati dalla classe genitore per adattarli alle proprie esigenze specifiche. Inoltre, possono anche aggiungere nuovi attributi e metodi per specializzarsi ulteriormente. Questa caratteristica consente di adattare le classi alle situazioni specifiche senza dover ricreare tutto da zero.

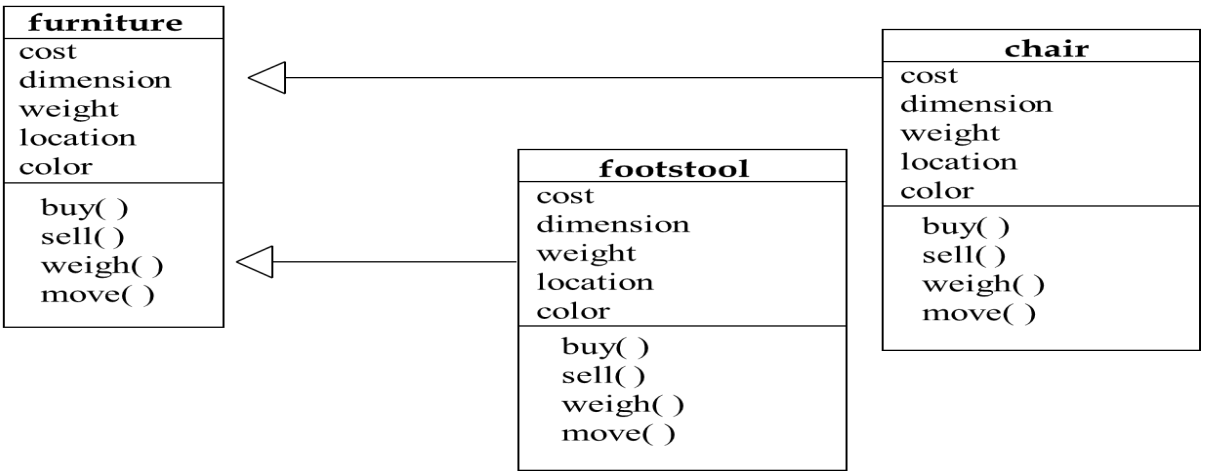
Spesso, l'ereditarietà viene descritta utilizzando espressioni comuni come "è-un" o "estende." Ad esempio, si dice che un "Cane è un Animale," il che riflette il fatto che la classe "Cane" è una specializzazione della classe "Animale." Questo tipo di espressione aiuta a chiarire le relazioni tra classi in una gerarchia.

L'ereditarietà contribuisce alla creazione di gerarchie di classi in cui le classi figlie ereditano da classi genitore. Questa struttura gerarchica aiuta a organizzare le classi in base a similitudini e differenze e consente una gestione più efficiente del codice.

L'ereditarietà rappresenta un processo di specializzazione, in cui le classi figlie specializzano le caratteristiche delle classi genitore per adattarle a un contesto specifico. Inoltre, le classi genitore generalizzano le caratteristiche comuni condivise da classi figlie.

In sintesi, l'ereditarietà nell'OO permette la creazione di nuove classi che estendono classi esistenti, consentendo una maggiore riusabilità del codice e la modellazione di relazioni tra classi basate su similitudini e differenze. Questo concetto è fondamentale per la

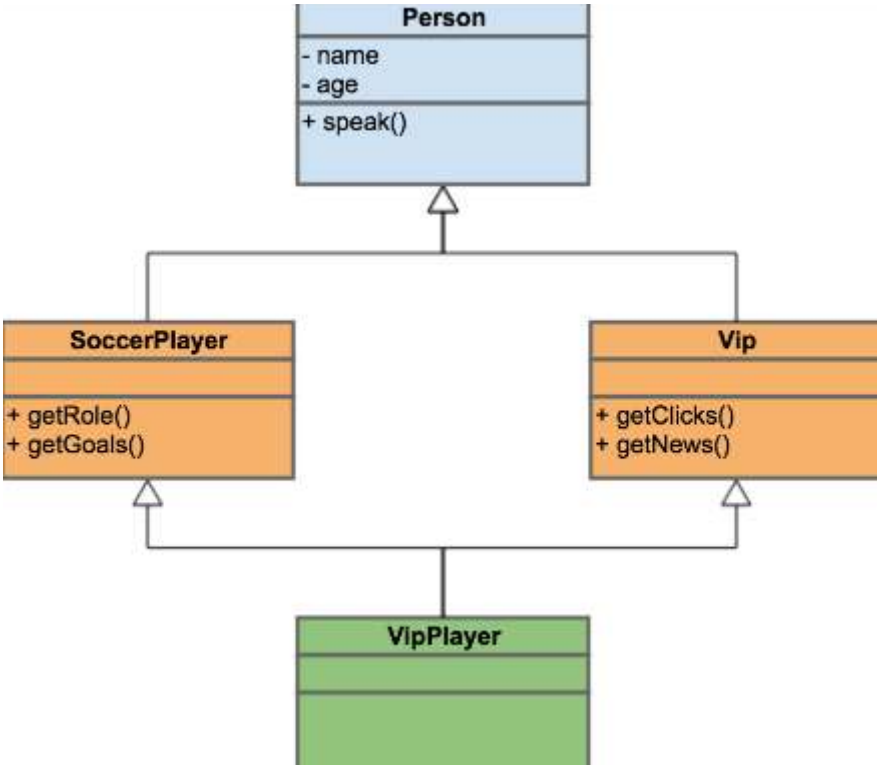
strutturazione e l'organizzazione del codice in progetti OO complessi.



È fondamentale comprendere che **l'ereditarietà** e **l'istanziamento** sono due concetti distinti nell'orientamento agli oggetti (OO), e spesso vi è confusione tra di essi.

- **Ereditarietà:** Riguarda l'estensione delle classi esistenti per creare nuove classi specializzate che ereditano attributi e metodi. Definisce le relazioni tra classi.
- **Istanziamento:** È il processo di creazione di oggetti basati su una classe. Ogni oggetto ha il proprio stato e può eseguire azioni specifiche. Riguarda la creazione di istanze di classi.

Con il **termine ereditarietà** multipla si intende invece, nei linguaggi ad oggetti, la capacità di una classe di ereditare funzionalità da più di una superclasse.
L'ereditarietà multipla rappresenta da sempre una delle principali armi a doppio taglio del mondo dello sviluppo software e tutti i principali linguaggi di programmazione hanno dovuto farci i conti e trovare una soluzione per mantenere stabilità e consistenza.



In generale, c'è da dire che nonostante l'ereditarietà multipla rappresenti una notevole potenzialità nel mondo Object Oriented, favorendo notevolmente la flessibilità e il riutilizzo del codice, **viene solitamente considerata un approccio da evitare** a causa della complessità che può derivare da una siffatta architettura. In particolare, i due problemi principali che possono sorgere quando si utilizza l'eredità multipla sono i seguenti:

- Ambiguità dei nomi
- Poca efficienza nella ricerca dei metodi definiti nelle classi

Il primo problema (**Ambiguità dei nomi**) può verificarsi se una proprietà o un metodo ereditato è definito con lo stesso nome in tutte le classi padre di una data classe.
Il secondo problema (**Poca Efficienza nella ricerca dei metodi definiti nelle classi**) è causato dalla nuova struttura che assume l'architettura in questo tipo di approccio che, come detto, è adesso rappresentata da un grafo. Infatti, con l'utilizzo di una struttura a grafo non è più possibile utilizzare la ricerca lineare (ideale sulle strutture ad albero) per l'individuazione dei metodi ma è necessario effettuare una sorta di "backtracking" sul grafo stesso.

Polymorphism

Il **polimorfismo** è un altro concetto fondamentale della programmazione ad oggetti (OOP – Object-Oriented Programming) che consente di scrivere codice che può gestire oggetti di diverse classi in modo uniforme, senza la necessità di conoscere il tipo specifico di ogni oggetto.
In pratica, il polimorfismo consente di creare metodi che possono accettare oggetti di classi diverse e di eseguire operazioni specifiche su di essi a seconda del loro tipo effettivo.

In matematica ci sono tantissimi esempi di polimorfismo basti pensare all'operazione "+" che può essere applicata a qualsiasi tipo di numero (naturale,intero,reale,complesso, …)

Il polimorfismo ci aiuta a gestire un enorme set di classi con operazioni simili fra loro senza dover utilizzare e di conseguenza ricordare nomi bizzarri (e.s., printf, fprintf, sprintf, …)

Various forms of Polymorphism

Sebbene il concetto di polimorfismo nei tipi abbia una interpretazione generalmente condivisa, il modo in cui i linguaggi di programmazione lo interpretano e lo implementano varia. In generale, esistono tre tipi di polimorfismo:

- **ad-hoc (sovraccarico/overloading**), in cui si sovraccarica la definizione di una data operazione su diversi tipi specifici;
- **di sottotipo (subtyping)**, in cui si stabiliscono relazioni da-astratto-a-specifico tra i tipi e si ottiene un polimorfismo con operazioni su tipi astratti;
- **parametrico (universale),** in cui abbiamo simboli astratti che rappresentano parametri di tipo.

UML in Some Details

L'**Unified Modeling Language** (UML) è un linguaggio di modellazione unificato che cerca di integrare approcci più vecchi nel campo della progettazione e sviluppo del software. È stato sviluppato da Rational, un'azienda produttrice di strumenti CASE (Computer-Aided Software Engineering), che ha assunto importanti figure come Booch, Rumbaugh e Jacobsen per contribuire allo sviluppo del linguaggio.

UML è stato standardizzato dall'OMG (Object Management Group) ed è ampiamente supportato da quasi tutti gli strumenti CASE orientati agli oggetti. Tuttavia, potrebbero esserci alcune limitazioni o variazioni nell'implementazione da parte di diversi strumenti.

Attualmente, UML si trova alla versione 1.3, anche se è possibile che siano state rilasciate versioni successive dal momento in cui questa informazione è stata resa disponibile. UML fornisce una notazione visuale per la modellazione dei sistemi software e dei processi, facilitando la comunicazione e la comprensione tra gli sviluppatori e gli stakeholder del progetto.

UML has many entities

UML (Unified Modeling Language) è ricco di entità o elementi che vengono utilizzati per rappresentare diversi aspetti di un sistema software. Queste entità includono:

- **Class (Classe):** Una classe rappresenta un tipo di oggetto o entità all'interno del sistema. Definisce la struttura e il comportamento degli oggetti appartenenti a quella classe.
- **Interface (Interfaccia):** Un'interfaccia definisce un contratto che specifica i metodi che una classe deve implementare. È utilizzata per definire le interfacce tra le classi senza specificare l'implementazione effettiva.
- **Datatype (Tipo di Dati):** Questo elemento rappresenta i tipi di dati utilizzati nel sistema, come int, string, o tipi personalizzati definiti dall'utente.
- **Component (Componente):** Un componente rappresenta una parte logica o fisica del sistema software. È spesso utilizzato per suddividere il sistema in unità più gestibili.
- **Node (Nodo):** Rappresenta un nodo fisico o una risorsa hardware all'interno dell'infrastruttura di rete o del sistema.
- **Use Case (Caso d'Uso):** Un caso d'uso rappresenta un singolo scenario di interazione tra un attore (utente esterno o sistema) e il sistema software. È utilizzato per catturare i requisiti e i comportamenti del sistema.
- **Subsystem (Sottosistema):** Un sottosistema è una suddivisione logica di un sistema software più grande. Rappresenta una raccolta di classi, oggetti o componenti correlati che collaborano per fornire una funzionalità specifica all'interno del sistema.

Queste entità forniscono un insieme completo di strumenti per modellare e documentare sistemi software in modo comprensibile e preciso, facilitando la comunicazione e la comprensione tra gli sviluppatori, gli stakeholder e gli altri membri del team del progetto.

UML diagrammi

UML (Unified Modeling Language) include nove tipi di diagrammi, che possono essere suddivisi in due categorie principali: diagrammi strutturali e diagrammi comportamentali.

Diagrammi Strutturali:

- **Diagramma delle Classi:** Rappresenta le classi del sistema, i loro attributi, le relazioni e i metodi.
- **Diagramma degli Oggetti:** Mostra istanze specifiche delle classi e le relazioni tra di loro in uno specifico momento.
- **Diagramma dei Componenti:** Descrive i componenti del sistema e le dipendenze tra di essi.
- **Diagramma di Distribuzione:** Rappresenta la disposizione fisica dei componenti di sistema e le connessioni tra i nodi.

Diagrammi Comportamentali:

- **Diagramma dei Casi d'Uso:** Identifica i casi d'uso, gli attori e le relazioni tra di essi per catturare i requisiti funzionali del sistema.
- **Diagramma delle Sequenze:** Illustra l'interazione tra gli oggetti nel tempo, mostrando la sequenza di messaggi scambiati tra di essi.
- **Diagramma di Collaborazione:** È simile al diagramma delle sequenze ma mette l'accento sull'organizzazione degli oggetti e delle loro interazioni.
- **Diagramma degli Stati (Statechart):** Rappresenta lo stato e il comportamento di un oggetto o di una classe in risposta agli eventi.
- **Diagramma delle Attività:** Mostra il flusso di attività, i passaggi e le decisioni in un processo o operazione del sistema.

Questi diagrammi consentono agli sviluppatori e agli stakeholder di rappresentare in modo chiaro e dettagliato vari aspetti di un sistema software, compresi i suoi componenti strutturali e i comportamenti dinamici. Ogni tipo di diagramma è utilizzato per scopi specifici nella fase di progettazione e sviluppo del software.

Object Oriented Concept Modeling

Goals of OO Concept Modeling

Gli obiettivi della modellazione concettuale orientata agli oggetti includono:

1. Comprendere il contesto operativo del sistema attraverso l'analisi orientata agli oggetti del contesto (OO Context Analysis). Questo significa identificare e comprendere come il sistema

interagirà con il suo ambiente, quali saranno i suoi attori principali e quali saranno i casi d'uso o le funzionalità chiave coinvolte.

2. Comprendere i requisiti effettivi del sistema tramite l'analisi orientata agli oggetti dei requisiti (OO Requirement Analysis). Questa fase implica la raccolta e la definizione dettagliata dei requisiti del sistema, tra cui le sue funzionalità, le regole di business e le relazioni tra gli oggetti coinvolti.

Spesso, la fase di analisi dei requisiti viene chiamata semplicemente "analisi dei requisiti", ma in realtà comprende entrambe queste attività. La modellazione concettuale orientata agli oggetti è fondamentale per comprendere in modo completo e accurato cosa il sistema deve fare e come si relaziona con il mondo circostante.

Use Cases for OO Concept Modeling

Gli Use Cases (casi d'uso) nella modellazione concettuale orientata agli oggetti svolgono un ruolo fondamentale nel comprendere e rappresentare il comportamento del sistema. Una descrizione di un caso d'uso è uno scenario che illustra un "filo di utilizzo" per il sistema e include i seguenti elementi:

1. Un diagramma, in cui sono rappresentati gli attori che svolgono ruoli specifici, come persone o dispositivi, nel contesto del sistema. Gli attori sono collegati ai casi d'uso che rappresentano i vari modi in cui il sistema può essere utilizzato da questi attori.

2. Una descrizione testuale che sequenza le attività coinvolte nel caso d'uso. Questa descrizione spiega in dettaglio come gli attori e il sistema interagiscono per raggiungere un obiettivo specifico. Include passi specifici, input e output, decisioni e condizioni di successo o errore.

I casi d'uso sono uno strumento prezioso per catturare e comunicare i requisiti del sistema in modo comprensibile sia per i professionisti tecnici che per i non tecnici. Forniscono una visione chiara di come il sistema verrà utilizzato e come le diverse parti interagiranno per soddisfare le esigenze degli utenti e dei clienti.

What is a Use Case?

Un caso d'uso è una rappresentazione di una tipica interazione tra attori (come utenti o altri sistemi) e il sistema in esame. Serve a descrivere un processo specifico che soddisfa le necessità di un utente o un gruppo di utenti. Un caso d'uso fornisce una visione dettagliata di uno scenario particolare, cioè come il sistema viene utilizzato in una determinata situazione.

Ad esempio, nel contesto di un elaboratore di testo, alcuni casi d'uso potrebbero includere:

- "Rendere il testo in grassetto": Questo caso d'uso descriverebbe come un utente può selezionare del testo e applicare la formattazione in grassetto.
- "Creare un indice": Questo caso d'uso spiegherebbe come un utente può generare un indice all'interno di un documento, inclusa la selezione delle parole chiave e la creazione della struttura dell'indice.

- "Cancellare una parola": Questo caso d'uso illustrerebbe il processo per eliminare una parola o una sezione di testo all'interno del documento.

In sostanza, i casi d'uso sono uno strumento essenziale per catturare e documentare in modo dettagliato come un sistema interagisce con gli utenti e gli altri attori coinvolti, aiutando a garantire che i requisiti siano chiari e compresi da tutti gli interessati.

I fatti chiave sui casi d'uso includono:

- Granularità: I casi d'uso possono variare in termini di dimensioni, da piccoli e specifici a grandi e complessi. Possono rappresentare interazioni utente-sistema di varie complessità.
- Funzione visibile all'utente: Spesso, i casi d'uso catturano funzionalità visibili all'utente, cioè azioni o processi che un utente può osservare o eseguire direttamente all'interno del sistema. Questi casi d'uso sono orientati alle esigenze degli utenti.
- Obiettivo discreto: I casi d'uso sono definiti in modo che raggiungano un obiettivo specifico o uno scopo ben definito. Ogni caso d'uso è progettato per ottenere una specifica operazione o risultato.
- Funzionalità esternamente richiesta: I casi d'uso descrivono la funzionalità richiesta esternamente dal sistema. Cioè, si concentrano su come il sistema deve interagire con attori esterni, come utenti o altri sistemi, al fine di soddisfare le esigenze degli utenti o degli stakeholder.

In sintesi, i casi d'uso sono uno strumento importante per rappresentare in modo chiaro e dettagliato le interazioni tra un sistema e gli utenti, enfatizzando il raggiungimento di obiettivi specifici e la funzionalità visibile all'utente. Possono variare in complessità e sono fondamentali per la definizione dei requisiti del sistema.

La cattura del contesto

La cattura del contesto è una delle prime attività da svolgere quando si affronta la modellazione dei casi d'uso. Questo processo permette di ottenere una visione generale dell'ambiente in cui il sistema opererà e dei vari attori (utenti, altri sistemi, dispositivi, ecc.) che interagiranno con esso. Inizialmente, non è necessario definire dettagli specifici dei casi d'uso, ma è importante avere una comprensione chiara del contesto in cui il sistema funzionerà.

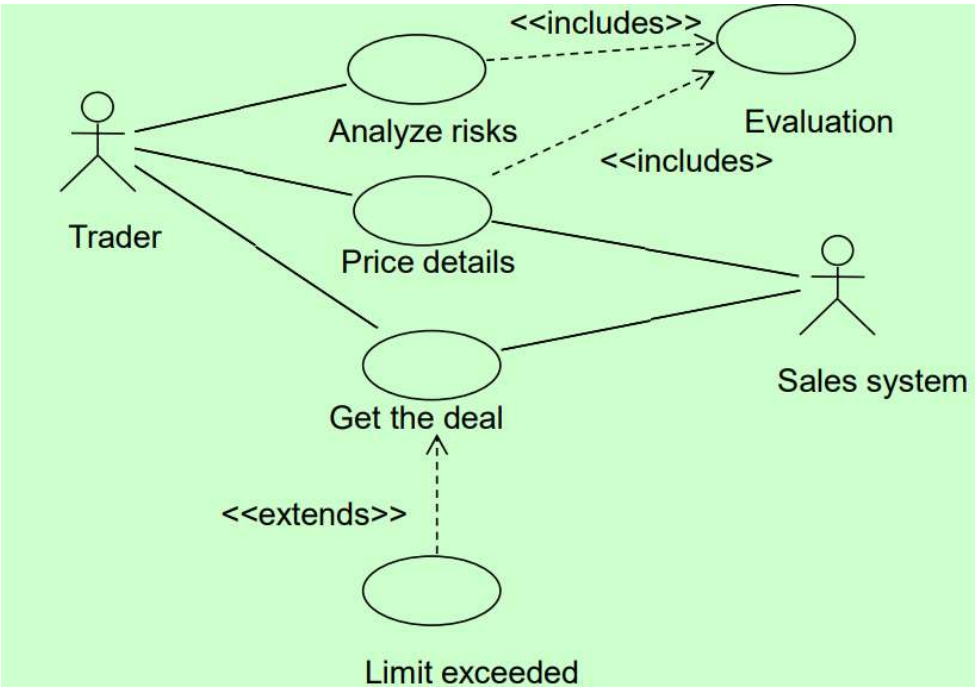
Per definire un caso d'uso, il passo successivo consiste nel identificare ogni azione o operazione discreta che il cliente o l'utente desidera eseguire con il sistema. Ogni di queste operazioni rappresenta un caso d'uso e dovrebbe essere denominata in modo chiaro e conciso. Si dovrebbe anche fornire una breve descrizione testuale del caso d'uso, in poche frasi o paragrafi, per spiegare in modo generale cosa fa il caso d'uso.

È importante notare che i dettagli specifici di ogni caso d'uso verranno aggiunti in fasi successive del processo di sviluppo del software. Inizialmente, l'obiettivo è identificare in modo esaustivo tutti i casi d'uso desiderati, dando loro nomi significativi e descrizioni di alto

livello. Successivamente, verranno aggiunti dettagli più specifici, inclusi scenari dettagliati, input e output, e altre informazioni necessarie per una comprensione completa del caso d'uso.
Nello sviluppo di un caso d'uso, è importante comprendere il ruolo e le interazioni degli attori nel sistema. Ecco alcune considerazioni chiave relative agli attori:

- Principali Compiti o Funzioni:
  Identificare i principali compiti o funzioni che l'attore svolgerà all'interno del sistema. Questi compiti dovrebbero riflettere il ruolo e le responsabilità dell'attore. Ad esempio, se l'attore è un "Cliente", i suoi principali compiti potrebbero includere "Effettuare un Ordine," "Visualizzare la Storia degli Ordini" e "Aggiornare le Informazioni dell'Account."
- Informazioni di Sistema:
  Determinare quali informazioni di sistema l'attore acquisirà, produrrà o modificherà durante le sue interazioni con il sistema. Ciò potrebbe coinvolgere l'input di dati, il recupero di dati o la modifica di dati. Ad esempio, un attore "Venditore" potrebbe acquisire informazioni sui prodotti, produrre registrazioni di vendita e modificare dati di inventario.
- Informare il Sistema sulle Modifiche Esterne:
  Valutare se l'attore deve informare il sistema su modifiche nell'ambiente esterno. Alcuni attori potrebbero dover segnalare eventi esterni al sistema. Ad esempio, un "Fornitore" potrebbe dover informare il sistema delle modifiche allo stock di prodotti.
- Informazioni Desiderate dall'Attore:
  Capire quali informazioni l'attore desidera ottenere dal sistema. Questo può includere rapporti, dati aggiornati o informazioni specifiche relative alle attività dell'attore.
- Notifiche su Cambiamenti Inattesi:
  Considerare se l'attore desidera essere informato da parte del sistema riguardo a cambiamenti inattesi o eventi importanti. Ad esempio, un "Manager" potrebbe voler essere avvertito quando si verificano situazioni critiche nel sistema.

Queste considerazioni aiutano a definire chiaramente il ruolo e le interazioni degli attori nel contesto del sistema e a stabilire le basi per la progettazione di casi d'uso efficaci.



Actors

Gli attori, nell'ambito della modellazione dei casi d'uso, rappresentano i ruoli che gli utenti o altri elementi giocano in relazione al sistema. Gli attori sono fondamentali poiché sono coloro che eseguono i casi d'uso. È importante notare che gli attori non devono necessariamente essere esseri umani; possono essere sia persone fisiche che altri sistemi, dispositivi o entità.

La chiave per identificare gli attori è cercare i ruoli chiave o le entità che interagiscono con il sistema in un modo o nell'altro. Ogni attore è associato a uno o più casi d'uso che riflettono le attività o le operazioni che svolgono all'interno del sistema. Gli attori possono trarre vantaggio dai casi d'uso o partecipare attivamente a essi, a seconda delle loro funzioni e responsabilità.

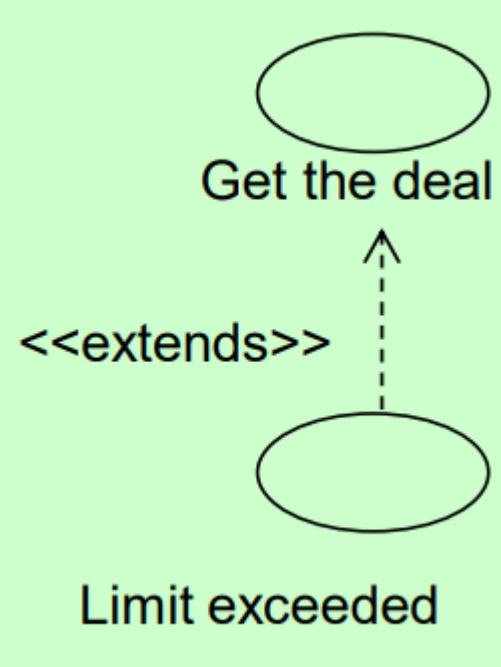
In sintesi, gli attori rappresentano i partecipanti al sistema, che possono essere persone, sistemi, dispositivi o qualsiasi entità coinvolta nell'interazione con il sistema. Identificare chi sono gli attori e comprenderne i ruoli è un passo importante nella modellazione dei casi d'uso e nella definizione dei requisiti del sistema.

Extends relationship

Il rapporto "estende" (extends) è un concetto chiave nella modellazione dei casi d'uso e viene utilizzato per rappresentare situazioni in cui un caso d'uso più complesso incorpora un caso d'uso base o principale. Ecco alcune considerazioni importanti sull'utilizzo di questa relazione:

- **Cattura del caso d'uso principale:** Inizialmente, si dovrebbe catturare il caso d'uso principale, che rappresenta la sequenza di azioni normali o comuni che gli attori eseguiranno. Questo caso d'uso principale riflette il flusso standard del processo.
- **Identificazione delle estensioni:** Per ogni passo o punto chiave all'interno del caso d'uso principale, è necessario chiedersi cosa potrebbe andare storto o come

potrebbe svilupparsi diversamente la situazione. Questo aiuta a identificare le situazioni in cui potrebbero essere necessarie estensioni.



- **Pianificazione delle variazioni:** Ogni variante o situazione in cui il caso d'uso principale potrebbe essere esteso viene rappresentata come un caso d'uso separato. Questi casi d'uso di estensione catturano i dettagli delle situazioni speciali o dei percorsi alternativi che possono verificarsi.
- **Collegamento con il caso d'uso principale:** Le estensioni vengono collegate al caso d'uso principale utilizzando la relazione "estende". Questo indica che il caso d'uso di estensione viene attivato solo in determinate condizioni o situazioni specifiche all'interno del caso d'uso principale.

In sintesi, il rapporto di estensione è utile per gestire situazioni eccezionali o varianti all'interno dei casi d'uso, consentendo di mantenere il caso d'uso principale più semplice e concentrato sul flusso standard. Le estensioni aiutano a catturare dettagli importanti sul comportamento del sistema in scenari specifici.

Includes relationship

La relazione "include" è utilizzata nei casi d'uso quando c'è un pezzo di comportamento che è comune o condiviso tra più casi d'uso. Questo comportamento comune viene isolato e definito come un caso d'uso separato, che può essere "incluso" in uno o più casi d'uso principali.

L'uso della relazione "include" permette di evitare la duplicazione di descrizioni e passi simili all'interno dei casi d'uso, migliorando la chiarezza e la gestibilità della modellazione. In altre parole, quando si identifica un comportamento che è riutilizzabile in diversi casi d'uso, anziché copiarlo all'interno di ognuno di essi, si crea un caso d'uso di inclusione che racchiude quel comportamento comune e lo collega ai casi d'uso principali.

Ad esempio, supponiamo di avere due casi d'uso diversi, uno per "Creare un nuovo utente" e un altro per "Modificare i dati dell'utente". Entrambi i casi d'uso richiedono la validazione dei dati inseriti dall'utente. Invece di ripetere la stessa sequenza di passi di validazione in entrambi i casi d'uso principali, è possibile creare un caso d'uso di inclusione chiamato "Validazione dei dati utente" e collegarlo ai casi d'uso principali. In questo modo, il comportamento di validazione viene gestito in modo centralizzato e può essere riutilizzato in altri casi d'uso se necessario.

L'utilizzo della relazione "include" consente una modellazione più efficiente e consente di gestire meglio il comportamento comune tra i casi d'uso senza doverlo ripetere in ciascuno di essi.

Comparing extends/includes

La differenza principale tra le relazioni "extends" (estende) e "includes" (include) nei casi d'uso risiede nell'intento e nelle situazioni in cui vengono utilizzate:

- Extends" (Estende): La relazione "estende" viene utilizzata quando si desidera rappresentare una situazione in cui un caso d'uso aggiunge comportamenti extra o eccezionali a un caso d'uso principale. Gli attori coinvolti sono generalmente gli stessi sia nel caso d'uso principale che nelle estensioni. Un attore è collegato al caso d'uso principale, e le estensioni vengono attivate solo in determinate condizioni.
- "Includes" (Include): La relazione "include" viene utilizzata quando si vuole rappresentare un comportamento comune condiviso tra più casi d'uso. Spesso, il caso d'uso incluso rappresenta un insieme di azioni o comportamenti che sono riutilizzati in più casi d'uso principali. Non è necessario che ci sia un attore specifico associato al caso d'uso incluso, e diverse estensioni possono coinvolgere attori diversi nei casi chiamanti.

In breve, "extends" viene utilizzato per gestire situazioni in cui un caso d'uso principale viene esteso con comportamenti aggiuntivi in casi eccezionali, mentre "includes" viene utilizzato per isolare comportamenti comuni che possono essere riutilizzati tra vari casi d'uso principali, spesso senza un attore specifico associato al caso d'uso incluso. Entrambe le relazioni sono utili per migliorare l'organizzazione e la chiarezza della modellazione dei casi d'uso.

Textual description

Una descrizione testuale è una spiegazione dettagliata e passo dopo passo delle interazioni tra gli attori (cioè le persone o i sistemi coinvolti) e un caso d'uso specifico. Questa descrizione dovrebbe essere chiara, precisa e concisa, in modo da fornire una guida dettagliata su come l'attore(s) interagisce con il caso d'uso.

Descrizione di un caso d'uso di esempio: Ottenere l'offerta

1. L'utente inserisce il nome utente e il numero di conto bancario.
2. Viene effettuata una verifica per assicurarsi che siano validi.
3. L'utente inserisce il numero di azioni da acquistare e l'identificativo dell'azione.
4. Si determina il prezzo delle azioni.
5. Viene effettuato un controllo per verificare che il limite di acquisto non sia superato.
6. L'ordine viene inviato alla Borsa di New York (NYSE).
7. Viene memorizzato il numero di conferma dell'ordine.

Si noti che:

1. Non sono stati elencati tutti i casi d'uso (ogni diagramma fornisce una vista parziale del sistema).

2. La relazione di inclusione (include) supporta la suddivisione delle specifiche comuni del sistema, consentendo la riutilizzazione delle funzionalità in più casi d'uso.

3. Non c'è una corrispondenza 1:1 tra le schermate dell'interfaccia utente e gli ovali dei casi d'uso. Un caso d'uso può coinvolgere più schermate o viceversa, a seconda delle interazioni necessarie.

4. Attenzione, poiché queste sono considerazioni importanti da tenere a mente quando si modella i casi d'uso e le interazioni all'interno di un sistema.

Textual Description (only for the frequent flyer use case)
L'utente accede al sottosistema per ottenere ulteriori informazioni sul suo stato di viaggiatore frequente. All'interno del sottosistema, l'utente può accedere a (a) informazioni generali sul programma del viaggiatore frequente, lo schema delle ricompense, come iscriversi, come accumulare miglia con le aziende partner, e (b) informazioni specifiche sul suo stato, come le miglia accumulate e il livello di status. L'utente può anche aggiornare il suo indirizzo.
NOTA: Questo è un formato molto diverso!

Object Oriented Analysis

OOA- A Generic View

Analizziamo brevemente il processo di Object-Oriented Analysis (OOA) con una visione generica:

1. **Estrazione delle Classi Candidato:**Identifica le principali entità coinvolte nel sistema e le considera come possibili classi. Ad esempio, in un sistema di prenotazione per taxi, le classi candidate potrebbero includere Prenotazione, Conducente di Taxi e Veicolo.
2. **Stabilire Relazioni di Base tra le Classi:**Determina le connessioni iniziali tra le classi candidate. Ad esempio, una Prenotazione può essere associata a un Conducente di Taxi.
3. **Definire una Gerarchia di Classi:**Se alcune classi sono specializzazioni o generalizzazioni di altre, crea una gerarchia di classi. Ad esempio, una classe generale "Veicolo" potrebbe avere sottoclassi come "Auto" e "Furgone".
4. **Identificare Attributi per ogni Classe:**Determina le proprietà fondamentali di ogni classe. Ad esempio, la classe Conducente di Taxi potrebbe avere attributi come Nome, ID e Numero di Telefono.
5. **Specificare Metodi per i Servizi degli Attributi:**Per ogni classe, identifica i metodi necessari per manipolare o fornire accesso agli attributi. Ad esempio, la classe Prenotazione potrebbe avere metodi come "ConfermaPrenotazione" o "AnnullaPrenotazione".
6. **Indicare come Classi/Oggetti sono Relazionati:**Utilizza diagrammi e notazioni appropriate per mostrare le relazioni tra classi e oggetti. Ad esempio, un diagramma delle classi UML può illustrare le associazioni tra Prenotazione, Conducente di Taxi e Veicolo.
7. **Costruire un Modello Comportamentale:** Considera come le classi interagiscono tra loro nel contesto del sistema. Definisci i comportamenti o i flussi di lavoro che coinvolgono più classi.

8. **Iterare sui Primi Cinque Passi:**Rivedi e rifina il modello in base alle nuove informazioni o alle modifiche necessarie. L'iterazione è essenziale per sviluppare un modello accurato e completo.

Extraction of Classes

L'estrazione delle classi è un passo cruciale nel processo di analisi orientato agli oggetti. Basandoci sulla sequenza normale "Ottieni l'affare", possiamo identificare le seguenti classi principali:

1. **Enter the user name & bank account**
2. **Check that they are valid**
3. **Enter number of shares to buy & share ID**
4. **Determine price**
5. **Check limit**
6. **Send order to NYSE**
7. **Store confirmation number**

Class diagram

Il diagramma delle classi è centrale nella modellazione orientata agli oggetti, poiché visualizza la struttura statica di un sistema. Esso rappresenta i tipi di oggetti presenti e le relazioni tra di essi. Le principali componenti includono:

- **Associazione:** Indica le relazioni tra le classi, evidenziando come gli oggetti di una classe possano essere collegati a quelli di un'altra. Ad esempio, un'associazione può mostrare come un Oggetto di tipo "Studente" è associato a un Oggetto di tipo "Corso".
- **Sottotipi (Subtypes):**Rappresenta la gerarchia di ereditarietà tra le classi, indicando le superclassi e le sottoclassi. Questo è utile quando alcune classi condividono attributi e comportamenti comuni. Ad esempio, "Animale" potrebbe essere una superclasse, e "Cane" e "Gatto" potrebbero essere sottoclassi.
- **Dipendenza:**Indica la relazione in cui una classe è influenzata da un'altra, ma senza un forte vincolo come nell'ereditarietà. Ad esempio, una classe "Aula" può dipendere dalla classe "Insegnante" per visualizzare informazioni correlate, ma senza una relazione diretta di sottotipo o associazione.

In sintesi, il diagramma delle classi offre una panoramica visiva della struttura statica del sistema, mostrando come le classi sono collegate attraverso associazioni, eredità e dipendenze. Questo strumento è essenziale per la progettazione e la comprensione degli aspetti statici di un sistema orientato agli oggetti.

We Have 3 Perspectives

Le tre prospettive menzionate si riferiscono all'approccio Object-Oriented Analysis (OOA), Object-Oriented Design (OOD) e Object-Oriented Programming (OOP).

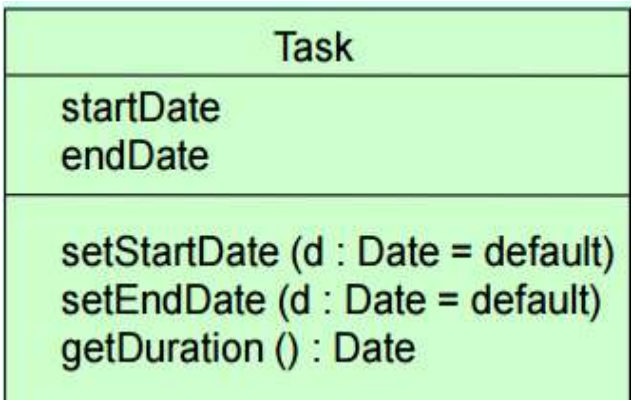
- **Concettuale (OOA):**
    - Mostra i concetti del dominio

    - Indipendente dall'implementazione.
    - Si concentra sulla comprensione e sulla rappresentazione dei concetti chiave relativi al dominio di interesse senza preoccuparsi degli aspetti tecnici di come tali concetti verranno implementati.
- **Specificazione (OOD):**
    - Definisce la struttura generale del sistema in esecuzione.
    - Specifica le interfacce del software, inclusi i tipi di dati e le relazioni tra di essi.
    - Si spinge oltre la fase concettuale, dettagliando come i concetti identificati verranno implementati nel sistema.
- **Implementazione (OOP):**
    - Fornisce i dettagli concreti dell'implementazione.
    - Spesso è l'unica prospettiva utilizzata nella programmazione pratica.
    - Traduce i concetti e le specifiche definite nelle fasi precedenti in codice effettivo, occupandosi degli aspetti tecnici e pratici dell'esecuzione del sistema.

In sintesi, queste prospettive costituiscono un approccio graduale e progressivo per sviluppare software orientato agli oggetti, passando dalla comprensione astratta del dominio (OOA) alla specifica strutturale (OOD) e infine all'implementazione pratica (OOP).

Classi

Una classe rappresenta un concetto fondamentale nella programmazione orientata agli oggetti. Essa agisce come uno schema o un modello da cui vengono creati gli oggetti. Una classe definisce un insieme di oggetti che condividono caratteristiche simili. Inoltre, la classe specifica il nome dell'oggetto, gli attributi che ne caratterizzano lo stato e le operazioni che possono essere eseguite su di esso.



Gli oggetti creati da una classe ereditano le sue caratteristiche, ma ciascun oggetto può avere valori unici per i propri attributi. La classe funge da struttura di base, consentendo di organizzare e raggruppare concetti simili all'interno di un programma. Inoltre, fornisce un modo per definire e gestire il comportamento degli oggetti associati ad essa, mediante la specifica delle operazioni che possono essere eseguite su di essi. In questo modo, la programmazione orientata agli oggetti offre un approccio modulare e organizzato per la progettazione del software.

Classi vs tipi

La distinzione tra "Class" e "Type" è una parte chiave della programmazione orientata agli oggetti.

Type:

- Rappresenta un protocollo compreso da un oggetto, definendo un insieme di operazioni che l'oggetto può eseguire.
- È un concetto astratto che specifica ciò che un oggetto può fare senza preoccuparsi di come effettivamente lo fa.
- Nei linguaggi di programmazione, un tipo può essere implementato da diverse classi, purché soddisfino le specifiche del tipo.

Class:

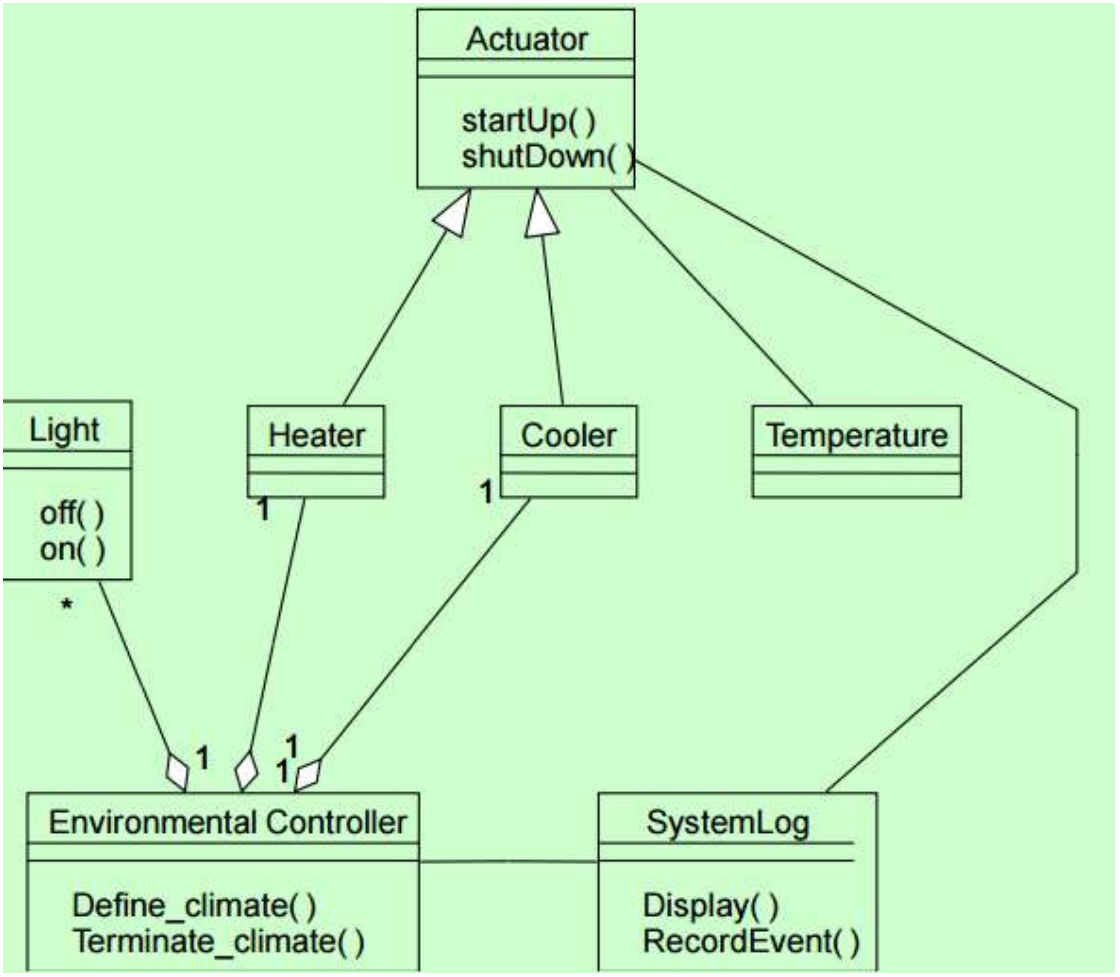
- È un costrutto orientato all'implementazione, che definisce la struttura e il comportamento di un oggetto.
- Può implementare uno o più tipi, garantendo che gli oggetti creati da questa classe soddisfino i protocolli specificati dai tipi.
- Nei linguaggi di programmazione come Java, un tipo può essere rappresentato da un'interfaccia, mentre in C++ può essere un'astratta classe.
- Nelle rappresentazioni visive come UML (Unified Modeling Language), il concetto di "type" può essere evidenziato utilizzando il stereotipo "<<type>>".

In sintesi, mentre un "Type" definisce cosa può fare un oggetto, una "Class" offre l'implementazione pratica di tali comportamenti, potendo implementare uno o più tipi. La separazione tra type e class contribuisce alla flessibilità e all'astrazione dei concetti nella programmazione orientata agli oggetti.

Associazioni

Le relazioni tra istanze di classi rappresentano connessioni concettuali tra oggetti appartenenti a diverse classi. Nel contesto accademico, se diciamo che uno studente è registrato per un corso, questo suggerisce un'associazione tra le istanze delle classi "Studente" e "Corso". Allo stesso modo, se un professore sta insegnando un corso, si crea una relazione tra le istanze delle classi "Professore" e "Corso". Queste connessioni possono essere modellate attraverso attributi, metodi o strutture dati nelle classi coinvolte. Ad esempio, una classe "Corso" potrebbe mantenere un elenco di studenti registrati e indicare il professore assegnato. La gestione di tali relazioni consente di riflettere accuratamente le interazioni reali tra gli oggetti nel sistema, contribuendo a una progettazione e un'implementazione del software più robuste.

Diagrammi e classi



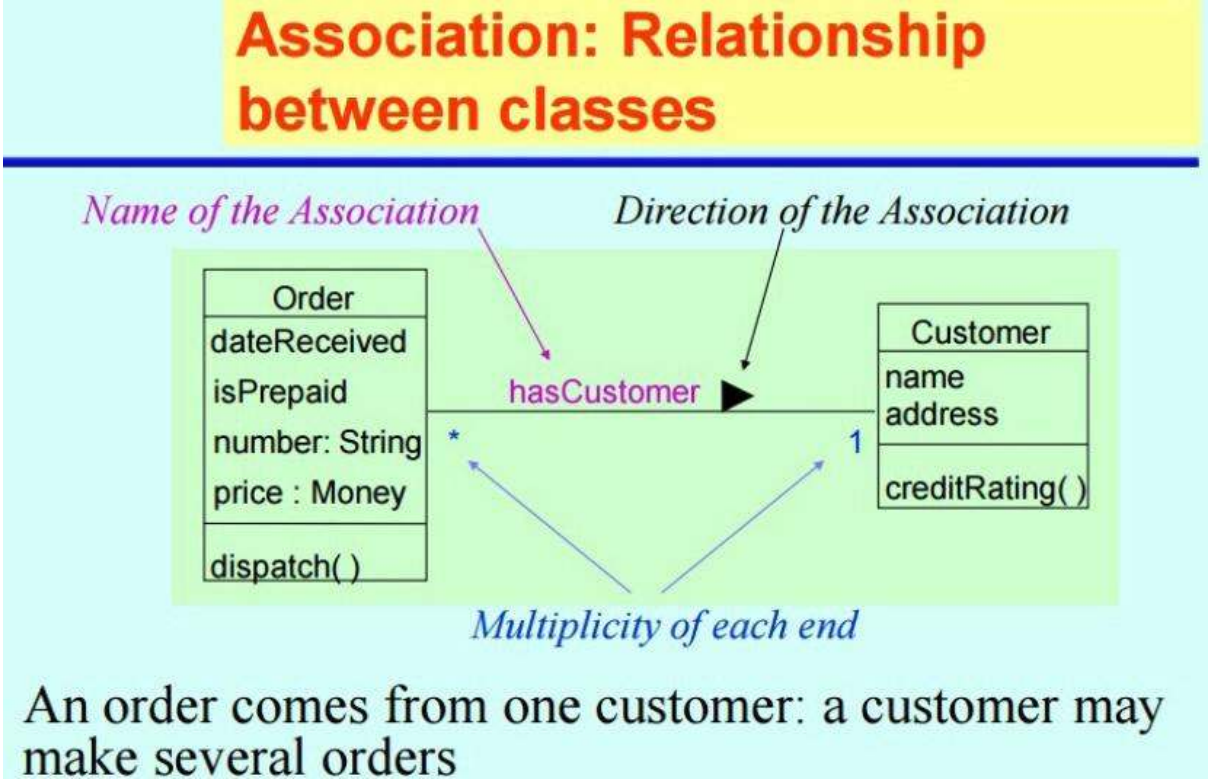
Le classi e i diagrammi sono strumenti cruciali nella progettazione orientata agli oggetti, ma è essenziale utilizzarli con attenzione per garantire chiarezza ed efficacia.

Una classe può essere inclusa in diversi diagrammi, riflettendo il fatto che un'entità può essere coinvolta in diverse parti di un sistema. Tuttavia, è importante seguire alcune linee guida per garantire che i diagrammi siano comprensibili ed esprimano chiaramente gli aspetti desiderati del sistema.

I diagrammi dovrebbero illustrare aspetti specifici del sistema e, pertanto, dovrebbero evitare di includere troppe classi, associazioni o dettagli non rilevanti. Limitare il numero di classi e associazioni contribuisce a mantenere la chiarezza e facilita la comprensione del diagramma. Inoltre, è consigliabile nascondere attributi o operazioni non rilevanti per concentrarsi sugli aspetti chiave della progettazione.

La creazione di un diagramma efficace spesso richiede più iterazioni, con l'opportunità di rifinire e migliorare la rappresentazione visuale del sistema. Questo processo iterativo aiuta a sviluppare un diagramma "corretto" che rispecchia in modo accurato la struttura e le relazioni del sistema oggetto di progettazione.

In sintesi, l'uso di classi e diagrammi richiede un approccio attento e iterativo per garantire che siano strumenti utili e comprensibili nella progettazione del software.

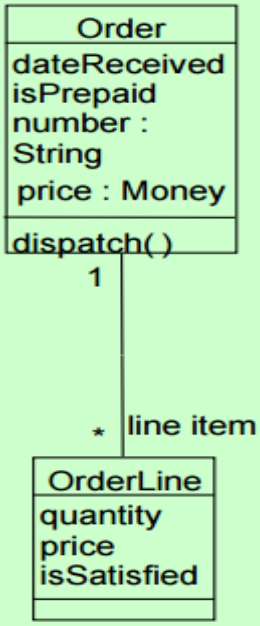


Naming associations

Nell'assegnare nomi alle associazioni tra le classi, è consigliabile evitare nomi privi di significato come "associated_with", "has", o "is_related_to". Al contrario, si dovrebbe cercare di utilizzare nomi che esprimano chiaramente la natura della relazione.

I nomi delle associazioni spesso prendono la forma di frasi verbali per catturare l'azione o la natura della connessione tra le classi. Ad esempio, invece di "has", si potrebbe utilizzare "has_part" per indicare una relazione più specifica, o al posto di "is_related_to", si potrebbe optare per un nome che descriva meglio la natura della connessione.

Assegnare nomi significativi alle associazioni contribuisce a rendere il modello più chiaro e comprensibile, fornendo un'indicazione diretta sulla natura delle relazioni tra le classi coinvolte.



Roles

Nel contesto delle associazioni tra classi, è importante considerare il concetto di "Roles". Un'associazione coinvolge due ruoli, che indicano la direzione o il significato specifico della connessione tra le classi.
Un "**Ruolo**" in un'associazione può essere:

- **Etichettato esplicitamente:** Un nome specifico assegnato manualmente al ruolo per indicare chiaramente la sua funzione o il suo significato nella relazione.
- **Implicitamente denominato dopo la classe di destinazione:** In alternativa, il ruolo può assumere il nome della classe di destinazione, riflettendo automaticamente il suo significato.

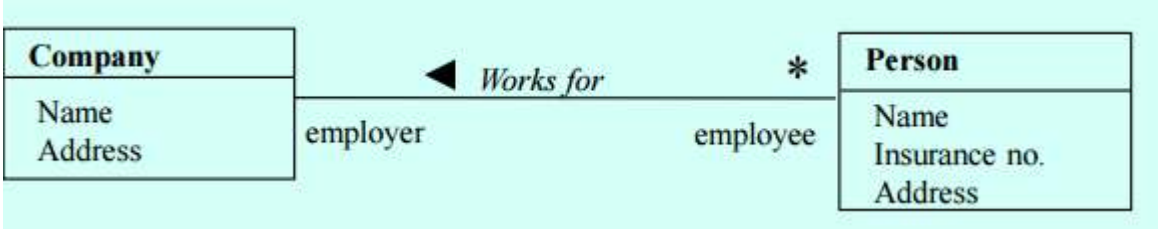
L'utilizzo di ruoli contribuisce a definire chiaramente il contesto dell'associazione, fornendo informazioni sulla natura delle relazioni tra le classi coinvolte. Questo può migliorare la comprensione del modello e facilitare la comunicazione tra gli sviluppatori e gli stakeholder del progetto.

Nel contesto delle associazioni tra oggetti, il concetto di "Role names" è cruciale per identificare chiaramente un'estremità dell'associazione. Ogni ruolo ha un nome che specifica la natura della connessione da quella prospettiva particolare.

Ad esempio, consideriamo l'associazione "Works for" tra le classi "Person" e "Company". In questo caso, il ruolo del dipendente potrebbe essere etichettato come "employee", mentre il ruolo del datore di lavoro potrebbe essere denominato "employer". Questi ruoli forniscono un contesto chiave per capire la direzione e la natura della relazione.

È importante notare che i nomi dei ruoli sono obbligatori nelle associazioni tra oggetti della stessa classe. Ad esempio, nella relazione "Supervises" tra due istanze della classe "Person" (forse un manager e un venditore), i ruoli specificano chiaramente la natura della supervisione (Manager e Salesperson).

L'uso appropriato dei nomi dei ruoli rende il modello più chiaro e comprensibile, contribuendo a una migliore documentazione e comunicazione del design del sistema.



Multiplicity

La molteplicità è un concetto chiave nelle relazioni tra classi nell'ambito della programmazione orientata agli oggetti. Essa indica quanti oggetti possono partecipare a una relazione specifica tra due classi.

Ad esempio, se abbiamo un'associazione tra le classi "Studente" e "Corso" con una molteplicità di "1..n" dal lato dello studente, ciò indica che uno studente può essere associato a uno o più corsi, mentre dal lato del corso, la molteplicità potrebbe essere "0..1" o "1", indicando che un corso può essere associato a nessuno o a un solo studente.

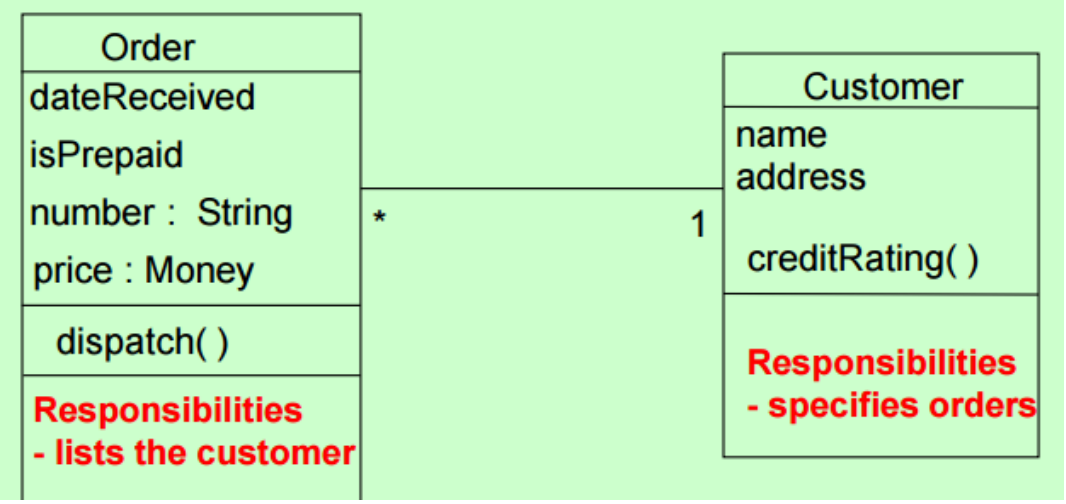
La molteplicità può essere rappresentata in un diagramma UML utilizzando numeri o intervalli per indicare la quantità di partecipanti consentiti da ciascun lato dell'associazione. Questo aiuta a definire chiaramente le regole di partecipazione e le relazioni quantitative tra le istanze delle classi coinvolte. La comprensione della molteplicità è essenziale per modellare accuratamente le relazioni e garantire la coerenza nella progettazione del sistema.

Le notazioni indicate rappresentano la molteplicità nelle relazioni tra classi in un diagramma UML:

- ***:0..infinity**: Questo indica che un oggetto può partecipare alla relazione da nessuno a un numero infinito di volte. In altre parole, non ci sono restrizioni superiori sul numero di partecipazioni.
- **1: 1..1:** Questo indica che un oggetto deve partecipare alla relazione esattamente una volta.
- **0..1:** Questo indica che un oggetto può partecipare alla relazione zero o una volta.
- **1..100** Questo indica che un oggetto può partecipare alla relazione da una a cento volte.
- **2,4,5**: Questa rappresentazione specifica i casi esatti in cui un oggetto può partecipare alla relazione, cioè due volte, quattro volte o cinque volte.

Queste notazioni aiutano a definire le regole di partecipazione e la quantità di oggetti che possono essere coinvolti in una relazione specifica tra classi. La comprensione di queste indicazioni è essenziale per modellare correttamente le associazioni nel contesto della programmazione orientata agli oggetti.

Responsabilità



Nel contesto della progettazione orientata agli oggetti, le responsabilità possono essere definite come compiti specifici affidati a una classe o a un oggetto all'interno di un sistema software.

Ad esempio, se consideriamo un sistema in cui "Il cliente specifica gli ordini", questo indica che la classe o l'oggetto associato al cliente avrà la responsabilità di fornire un meccanismo per la specifica degli ordini. Dall'altra parte, "Gli ordini elencano il cliente" suggerisce che la classe o l'oggetto associato agli ordini avrà la responsabilità di tenere traccia e registrare l'associazione con il cliente corrispondente.

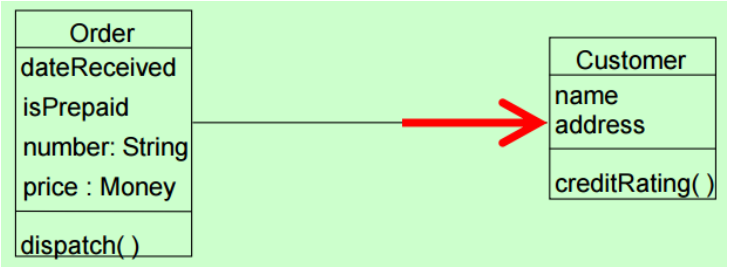
Queste responsabilità contribuiscono a definire il comportamento e le interazioni delle classi nel sistema, promuovendo una progettazione chiara e ben strutturata.

Navigability - Indicated by Arrow

La navigabilità in un diagramma delle classi indica la capacità di un'istanza di una classe di accedere alle istanze associate di un'altra classe. Questo può essere rappresentato da frecce direzionali in un diagramma UML, che indicano la direzione in cui è possibile navigare attraverso l'associazione.
Ad esempio:

- - "Order has to be able to determine the Customer": Questo suggerisce che un'istanza della classe "Order" può accedere all'istanza della classe "Customer" associata. La freccia direzionale può essere tracciata dal lato dell'"Order" al lato del "Customer" per indicare questa navigabilità.
- "Customer does not know all Orders": In questo caso, indica che un'istanza della classe "Customer" potrebbe non essere in grado di accedere a tutte le istanze della classe "Order" associate ad essa. La freccia potrebbe non essere presente o essere bidirezionale solo in modo limitato.
- "Bi-directional association: Navigability in both directions": Se l'associazione è bidirezionale, significa che le istanze di entrambe le classi possono navigare l'una all'altra. Tuttavia, può richiedere l'uso di ruoli specifici per garantire l'identificazione corretta durante la navigazione.

La navigabilità è un aspetto importante nella progettazione delle associazioni tra classi, poiché influisce sulla struttura e sul comportamento del sistema. La sua corretta definizione contribuisce a una modellazione più precisa delle interazioni tra le classi.



Naming conventions

Le convenzioni di denominazione sono regole stabilite per attribuire nomi coerenti e significativi agli elementi del codice, facilitando così la comprensione e la manutenzione del software. Nel contesto del codice fornito:

java

```
class Order {
   public Enumeration orderLines();
   public Customer customer();
}
```

Le convenzioni di denominazione adottate sembrano seguire uno stile orientato agli oggetti comune. Alcune osservazioni:

- **Nomi di Classe:** "Order" rappresenta una classe, e il nome è scritto con la convenzione di inizio maiuscola. Questo è conforme a uno degli standard di denominazione comuni nelle lingue di programmazione orientate agli oggetti.
- **Metodi:**
    - orderLines():Il nome del metodo segue la convenzione camelCase, in cui la prima parola inizia con una lettera minuscola e le successive parole sono scritte con la prima lettera maiuscola. Questo metodo restituisce un'enumerazione di linee d'ordine.
    - customer():Similarmente, il nome del secondo metodo è in stile camelCase e suggerisce che il metodo restituisce un oggetto di tipo "Customer".

L'uso di queste convenzioni facilita la lettura del codice e aiuta a capire il ruolo delle classi e dei metodi all'interno del sistema. Inoltre, la scelta di nomi descrittivi come "orderLines" e "customer" contribuisce a una chiara espressione dell'intento delle funzionalità offerte dalla classe.

Association classes

Le classi di associazione (association classes) sono utili quando gli attributi non appartengono a nessuna delle classi coinvolte direttamente, ma piuttosto alla relazione stessa tra le classi. Questa situazione si verifica quando ci sono informazioni specifiche e rilevanti solo per la relazione tra le istanze delle classi.

Ad esempio, consideriamo una relazione "Studente" e "Corso", in cui oltre alle informazioni sulla singola istanza di "Studente" o "Corso", vogliamo conservare dettagli specifici per l'associazione tra uno studente e un corso. In questo caso, potremmo utilizzare una classe di associazione per rappresentare questa relazione e memorizzare gli attributi ad essa associati.

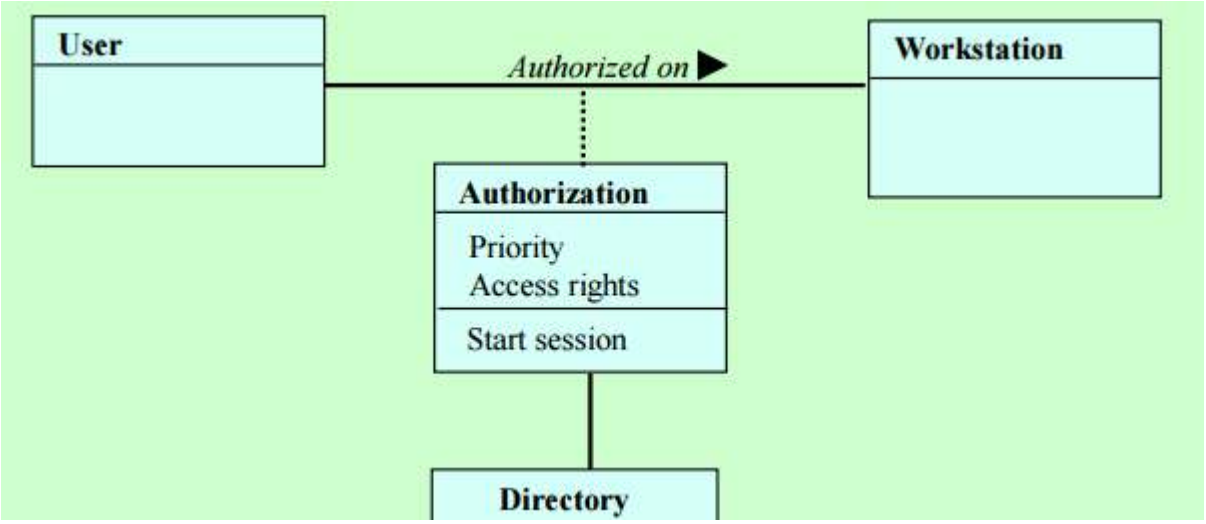
```
```java
class Student {
   // attributi specifici dello studente
}
```
```

```
class Course {
    // attributi specifici del corso
}

class Enrollment {
    // attributi specifici dell'associazione tra Studente e Corso
    private Date enrollmentDate;
    private int grade;

    // metodi e logica specifica dell'associazione
}
```

Nell'esempio sopra, "Enrollment" è una classe di associazione che conserva gli attributi specifici dell'associazione tra uno studente e un corso, come la data di iscrizione e il voto. Utilizzare classi di associazione può contribuire a mantenere il modello più pulito e a riflettere più accuratamente la complessità del dominio del problema che si sta affrontando.



Le classi e gli oggetti

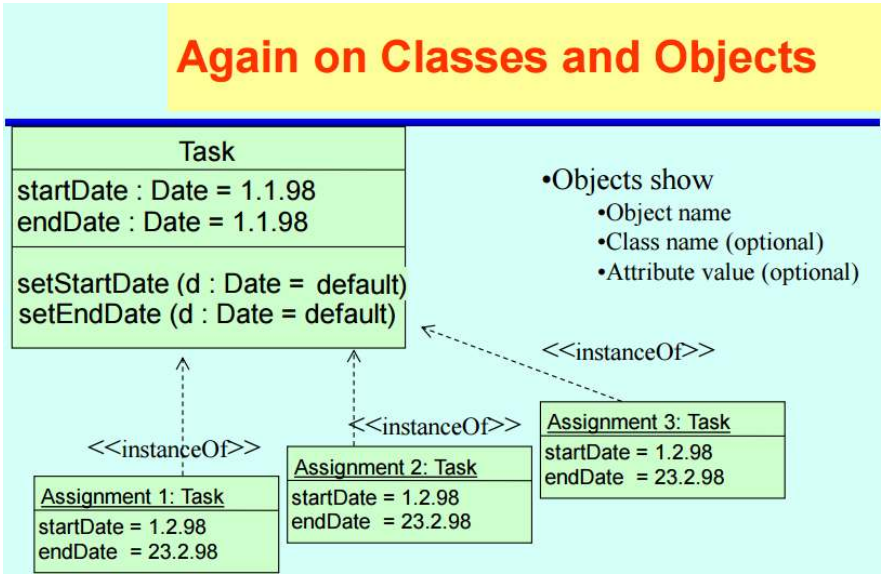
Le classi e gli oggetti sono concetti fondamentali nella programmazione orientata agli oggetti.

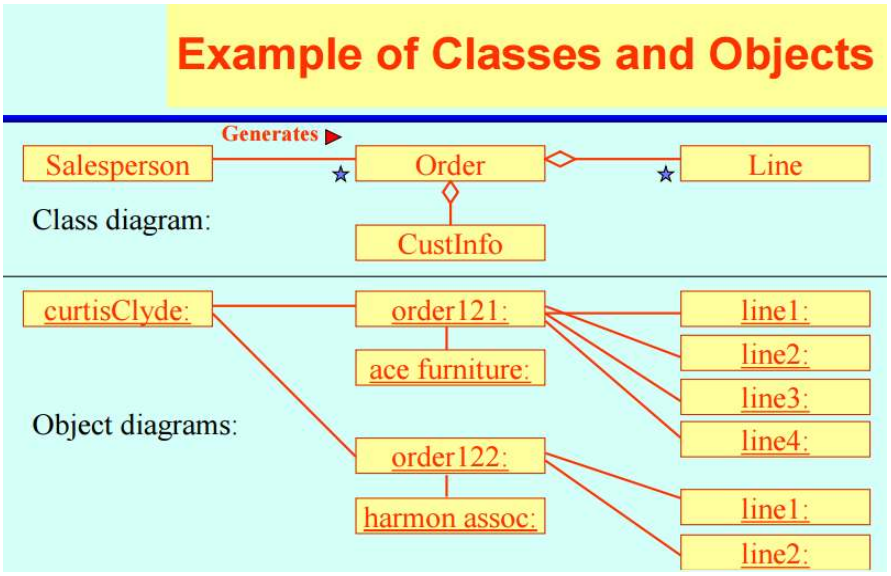
Una classe è una struttura che definisce il modello o il tipo di un gruppo di oggetti. Essa specifica il nome della classe, gli attributi (che rappresentano lo stato degli oggetti) e le operazioni (che definiscono il comportamento degli oggetti). La classe funge come uno schema o un prototipo dalla quale gli oggetti specifici vengono creati.

Gli oggetti, d'altra parte, sono le istanze di una classe. Quando si crea un oggetto, si crea un'istanza di quella classe specifica, ereditando le sue caratteristiche (attributi e operazioni).

Gli oggetti rappresentano entità concrete del mondo reale e interagiscono tra loro attraverso le operazioni definite nella classe.

In sintesi, le classi definiscono la struttura e il comportamento di gruppi di oggetti, mentre gli oggetti sono le istanze specifiche di tali classi che esistono durante l'esecuzione del programma. Questo paradigma fornisce un modo organizzato e modulare per progettare e implementare software.





Attributi

Gli attributi sono le caratteristiche associate a un'istanza di una classe in programmazione orientata agli oggetti.

A livello concettuale, indicano concetti astratti, come ad esempio il fatto che un cliente può avere un nome. Nella fase di specifica, si definisce che un cliente può comunicare il proprio nome e impostarlo, iniziando a delineare le interazioni degli oggetti con il proprio stato. Infine, nell'implementazione, si specifica che esiste una variabile di istanza disponibile per

conservare il nome effettivo del cliente, traducendo così i concetti astratti in strutture di dati e comportamenti concreti nel codice. Questo percorso da concetti astratti a implementazioni concrete contribuisce alla costruzione di un software strutturato e funzionale.

Difference between attributes and associations

Dal punto di vista concettuale, le differenze tra attributi e associazioni possono essere minime. Entrambi rappresentano informazioni associate a un'istanza di una classe.

Tuttavia, da una prospettiva di specifica/implementazione, emergono distinzioni significative. Le associazioni possono consentire una navigabilità da un tipo a un attributo e possono memorizzare riferimenti a oggetti, consentendo la condivisione di valori tra le istanze. Al contrario, gli attributi sono generalmente singolarmente valutati (0..1) e memorizzano valori piuttosto che riferimenti.

In pratica, gli attributi spesso contengono oggetti semplici come numeri, stringhe, date o oggetti di tipo "Money". La chiarezza nella comprensione di queste differenze contribuisce a una progettazione più precisa e a una migliore gestione delle informazioni all'interno di un sistema orientato agli oggetti.

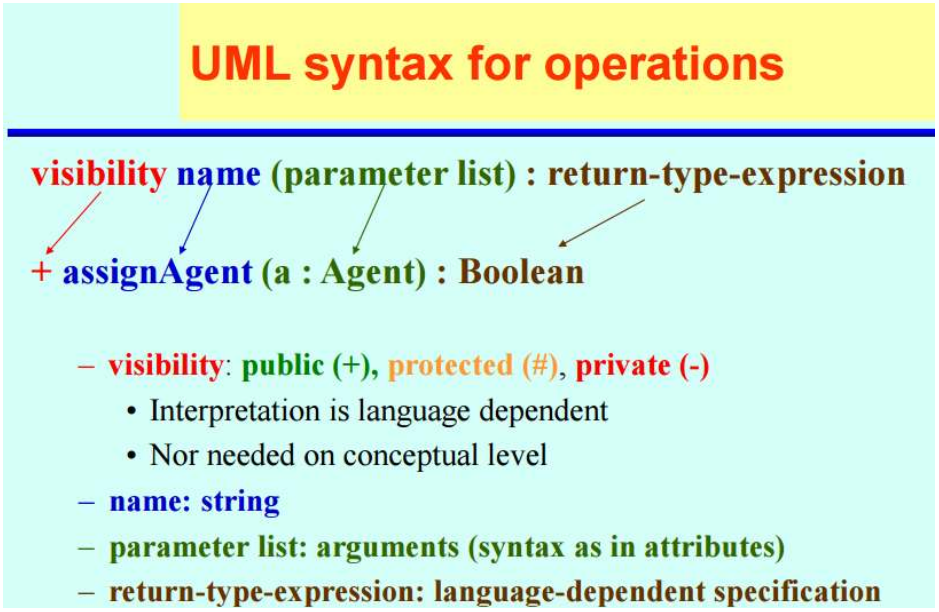
Operations

Le operazioni, o metodi, sono processi che una classe sa eseguire in un contesto di programmazione orientata agli oggetti. Corrispondono ai messaggi che una classe può ricevere o inviare. Le operazioni di una classe rappresentano le azioni o i comportamenti che un oggetto di quella classe può eseguire.

Dal punto di vista concettuale, le operazioni riflettono le responsabilità principali della classe. A livello di specifica, le operazioni diventano i messaggi pubblici che compongono l'interfaccia della classe. Questi messaggi rappresentano il modo in cui altri oggetti possono interagire con l'istanza della classe.

Tuttavia, di solito non si mostrano operazioni che manipolano direttamente gli attributi di una classe a livello di specifica. Questa pratica è in linea con il principio di incapsulamento, che promuove la gestione degli attributi all'interno della classe stessa, limitando l'accesso diretto da parte di altre classi.

In sintesi, le operazioni di una classe rappresentano le azioni che possono essere eseguite su un oggetto di quella classe. Queste operazioni definiscono il comportamento dell'oggetto e costituiscono l'interfaccia attraverso la quale altre classi possono interagire con essa.



Types of operations

Le operazioni in programmazione orientata agli oggetti possono essere categorizzate in due tipi principali: query e modifier.

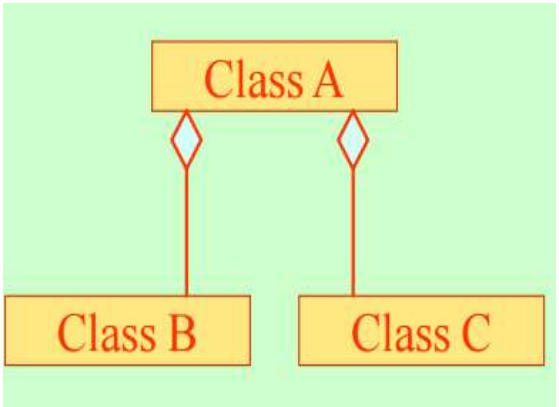
Le operazioni di tipo "query" restituiscono un valore senza modificare lo stato interno della classe. Esse forniscono informazioni sullo stato corrente dell'oggetto senza alterarlo. Al contrario, le operazioni di tipo "modifier" modificano lo stato interno della classe, aggiornando attributi o eseguendo azioni che influiscono sul comportamento futuro dell'oggetto.

Nel contesto di ottenere e impostare valori, l'operazione di "ottenimento" rappresenta una query, restituendo informazioni, mentre l'operazione di "impostazione" è un modifier, poiché modifica lo stato dell'oggetto.

Queste categorie di operazioni aiutano a organizzare il comportamento delle classi, garantendo una gestione chiara e coerente delle informazioni all'interno del sistema.

Aggregation

L'aggregazione è una forma speciale di associazione in programmazione orientata agli oggetti.

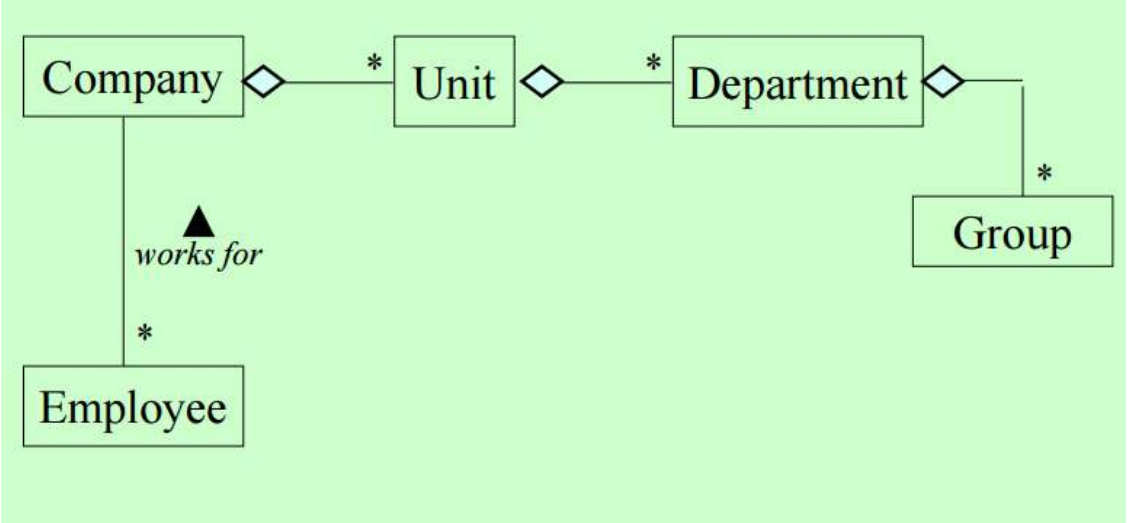


In un'aggregazione, i componenti sono parti dell'oggetto aggregato. Ad esempio, una "Auto" può essere aggregata da un "Motore" e "Ruote", che sono parti costituenti dell'auto. L'aggregazione è transitiva, il che significa che se un oggetto è parte di un altro oggetto, e quest'ultimo è parte di un terzo oggetto, allora il primo oggetto è indirettamente parte del terzo.

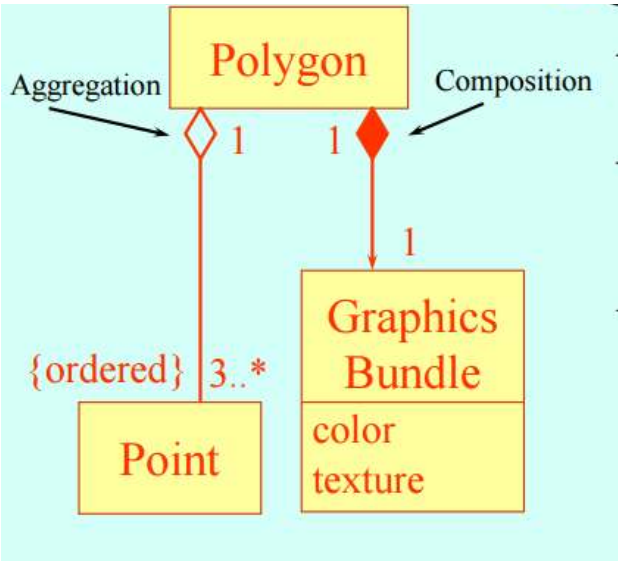
Un esempio comune di aggregazione è la "parti dell'auto". Quando si esegue un'aggregazione di parti, si può immaginare una "esplosione delle parti", in cui le parti possono essere analizzate e dettagliate ulteriormente.

Un altro esempio può essere l'organigramma di una società, dove i dipendenti sono parti di dipartimenti, che a loro volta sono parti della struttura organizzativa dell'azienda.

L'aggregazione aiuta a modellare relazioni strutturali tra gli oggetti, sottolineando la composizione e la gerarchia delle parti all'interno di un sistema.



Aggregation and composition

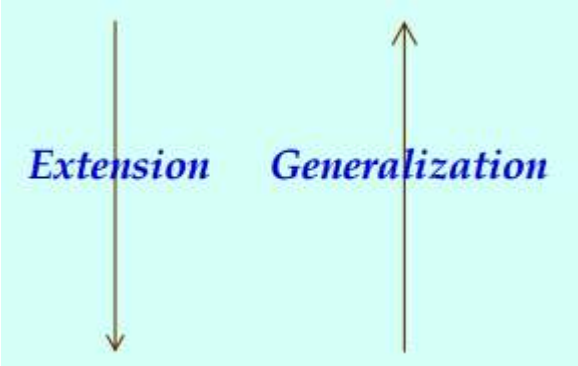


La composizione è un tipo di associazione in programmazione orientata agli oggetti in cui i componenti appartengono esclusivamente a un tutto. Le parti vengono create quando viene creato l'oggetto principale e vengono eliminate quando l'oggetto principale viene eliminato, seguendo un principio di "cascading delete". Ad esempio, la relazione tra un "Computer" e i suoi "Componenti" è un esempio di composizione, dove i componenti sono strettamente legati al computer e vengono eliminati con esso. La composizione è spesso utilizzata nelle associazioni "1..1" e implica una stretta dipendenza tra le parti e l'intero.

Per riconoscere l'aggregazione in programmazione orientata agli oggetti, valuta se la relazione può essere descritta come "parte di". Se le operazioni eseguite sull'oggetto principale influenzano anche gli oggetti considerati parti e se le parti vengono create quando viene creato l'oggetto principale e vengono eliminate quando l'oggetto principale viene eliminato, allora è probabile che sia un'aggregazione. Considera il concetto di "parte di" e l'impatto delle operazioni sull'intera struttura per determinare se si tratta di un'associazione o di un'aggregazione.

Generalization vs. Extension

Nella programmazione orientata agli oggetti, la generalizzazione si riferisce alla creazione di una classe più generale che rappresenta le caratteristiche comuni di diverse classi più specifiche. Ad esempio, una classe "Veicolo" potrebbe rappresentare le caratteristiche condivise da "Auto", "Camion" e "Autobus".



L'estensione, invece, riguarda la creazione di nuove classi che ampliano o specializzano una classe esistente. Ad esempio, la classe "Auto" potrebbe essere estesa per creare classi più specifiche come "Berlina" e "Station Wagon", che ereditano le caratteristiche dell'"Auto" ma possono avere differenze specifiche.

In termini pratici, immagina che "Veicolo" sia la categoria generale che include concetti comuni, mentre "Auto", "Camion" e "Autobus" sono sottocategorie più specifiche con caratteristiche aggiuntive.

L'uso combinato di generalizzazione ed estensione consente una struttura gerarchica flessibile e modulare, migliorando l'organizzazione e la riusabilità del codice.

L'istanziazione e la generalizzazione sono concetti fondamentali nella gerarchia delle classi in programmazione orientata agli oggetti.

L'istanziazione si verifica quando si crea un'istanza di una classe specifica. Ad esempio, "Shep" è un'istanza specifica di un "Border Collie", che a sua volta è un'istanza di un "Cane" e così via.

Instantiation and generalization

La generalizzazione, d'altra parte, rappresenta la relazione di "è un tipo di". Ad esempio, "Border Collie" è un tipo di "Cane", che a sua volta è un tipo di "Animale". La generalizzazione è transitiva, il che significa che se A è un tipo di B e B è un tipo di C, allora A è anche un tipo di C.

Quando combiniamo questi concetti negli esempi forniti:

1. "Shep è un Border Collie": Istanziamento di "Shep" come un'istanza specifica di "Border Collie".
2. "Un Border Collie è un Cane": Generalizzazione che indica la relazione tra "Border Collie" e "Cane".
3. "I Cani sono Animali": Ulteriore generalizzazione che estende la gerarchia a livello di "Animali".
4. "Un Border Collie è una Razza": Un aspetto specifico della generalizzazione che indica la classificazione di "Border Collie" come una razza di cane.
5. "Il Cane è una Specie": Un altro aspetto di generalizzazione che collega "Cane" al concetto più generale di "Specie".

Tuttavia, è importante notare che la generalizzazione è transitive, mentre l'istanziazione non lo è. Quindi, affermazioni come "Shep è una Razza" o "Border Collie è una Specie" potrebbero risultare incoerenti nell'ambito della gerarchia degli oggetti e delle classi.

Concept of generalization

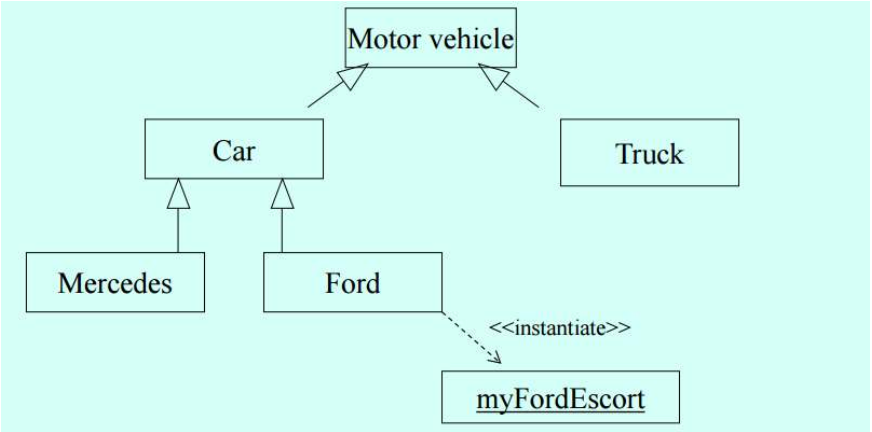
Il concetto di generalizzazione nella programmazione orientata agli oggetti si basa sulla relazione di sottoinsieme tra le classi. Una classe, che definisce implicitamente un insieme di oggetti, può essere considerata come il rappresentante di un insieme specifico.

Ad esempio, considera la classe "Car". Implicitamente, questa classe definisce un insieme di oggetti che sono automobili. La notazione "aCar $\in$ Car" indica che un'istanza specifica di automobile, rappresentata da "aCar", appartiene all'insieme di tutte le automobili definite dalla classe "Car".

La generalizzazione, in questo contesto, è una relazione di sottoinsieme. Se consideriamo la classe "Truck", possiamo dire che "Truck" è un sottoinsieme di "Car", indicato come "Truck $\subseteq$ Car". Questo significa che ogni camion è un tipo specifico di automobile, sottolineando la gerarchia tra le classi.

In sintesi, la generalizzazione si basa sulla creazione di gerarchie di classi, in cui una classe più generale definisce implicitamente un insieme di oggetti, e una classe più specifica rappresenta un sottoinsieme di quella classe più generale.

Class Diagram with Inheritance



How to define classes (revised)?

Per definire classi in un contesto di progettazione del software, è possibile seguire una serie di passaggi:

- **Analisi dei casi d'uso:** Esamina i casi d'uso per identificare i sostantivi. I sostantivi spesso corrispondono a concetti che possono essere modellati come classi.
- **Definizione di classi:**Assegna una classe per ogni sostantivo identificato nei casi d'uso. Aggiungi ulteriori classi, se necessario, per rappresentare concetti rilevanti nel tuo dominio.
- **Documentazione delle regole di appartenenza:**Documenta le regole che determinano quali oggetti appartengono a ciascuna classe. Queste regole possono definire le caratteristiche che gli oggetti devono possedere per appartenere a una classe specifica.
- **Aggiunta di associazioni:**Identifica le relazioni tra le classi e aggiungi associazioni per modellare tali relazioni. Le associazioni definiscono come le istanze di una classe interagiscono con le istanze di un'altra classe.
- **Considerazione delle relazioni di sottoinsieme:**Rifletti sulle relazioni di sottoinsieme tra le classi per costruire generalizzazioni. Se esiste una gerarchia tra le classi, considera la creazione di classi più generiche e specializzate.

Seguendo questi passaggi, è possibile creare un modello di classi che rappresenta accuratamente il dominio del problema e fornisce una base solida per la progettazione del software. L'approccio focalizzato sui casi d'uso aiuta a garantire che le classi riflettano in modo significativo le necessità del sistema e le interazioni tra gli oggetti.

To which class does an object belong?

Per determinare a quale classe appartiene un oggetto, si utilizza la definizione di appartenenza a classe, che può essere implicita o esplicita.

- **Definizione implicita dell'appartenenza:**L'appartenenza a classe può essere definita implicitamente attraverso regole. Queste regole stabiliscono le condizioni che un oggetto deve soddisfare per appartenere a una classe specifica. Le informazioni sui valori degli attributi disponibili consentono di determinare l'appartenenza a una

classe. In termini di logica terminologica nell'ambito dell'intelligenza artificiale, questo processo può coinvolgere concetti come sottosomma e classificatore.

- **Definizione esplicita dell'appartenenza:**L'appartenenza a classe può anche essere definita esplicitamente attraverso un'enumerazione. L'istanziamento di un oggetto specifico definisce la sua appartenenza a una classe. Tuttavia, questo approccio può presentare sfide, poiché è importante vietare operazioni che violano le restrizioni della classe.

In sostanza, stabilire a quale classe appartiene un oggetto coinvolge la valutazione delle regole implicite o l'enumerazione esplicita, garantendo che l'oggetto soddisfi le condizioni specificate dalla classe. Questo processo è essenziale per costruire modelli di classi significativi e garantire che gli oggetti siano adeguatamente rappresentati all'interno del sistema.

Changing classes

La dinamica di cambiare le classi è supportata in UML (Unified Modeling Language), che offre la possibilità a un oggetto di cambiare dinamicamente la sua classe utilizzando lo stereotipo di tipo. In linguaggi di programmazione come C++ e Java, questo concetto può essere implementato utilizzando una classe di base comune e successivamente modificando gli oggetti puntati o referenziati con costruttori appropriati. Questo approccio consente la flessibilità nell'adattare dinamicamente gli oggetti alle esigenze del sistema durante l'esecuzione del programma, offrendo una gestione dinamica e polimorfica delle classi.

Generalization: extension & restriction

Nel contesto della generalizzazione in programmazione orientata agli oggetti, l'estensione e la restrizione sono concetti chiave.

La generalizzazione implica che le classi figlie (sottoclassi) ereditino gli attributi e le operazioni dalla classe genitore (superclasse). L'estensione si verifica quando vengono aggiunti nuovi attributi o operazioni nella sottoclasse, ampliando così le funzionalità ereditate. D'altro canto, la restrizione si manifesta quando vengono imposti vincoli aggiuntivi sugli attributi ereditati dalla superclasse.

Ad esempio, considera la relazione tra "Ellisse" e "Cerchio". Un cerchio può essere visto come un'estensione di un'ellisse, ereditando gli attributi come lunghezza degli assi. Tuttavia, la restrizione si applica quando si specifica che un cerchio è un'ellisse con assi uguali. È importante notare che, nella pratica, una modifica arbitraria delle dimensioni degli assi di un cerchio può violare questa restrizione, sottolineando la necessità di cautela nell'applicare restrizioni nelle gerarchie di classi.

Perspectives

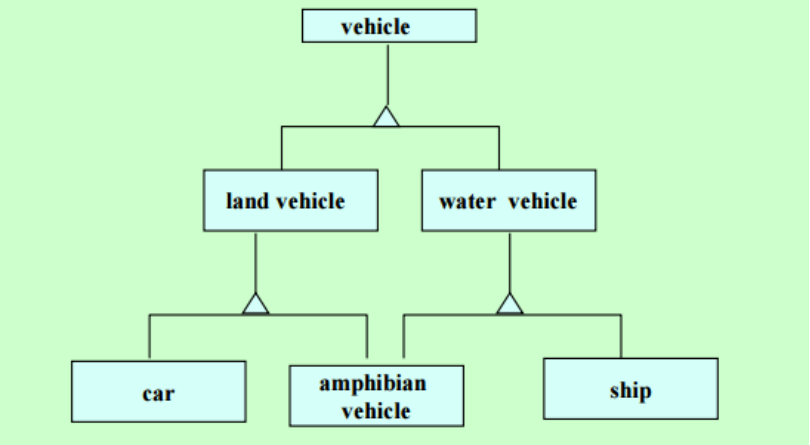
Nel contesto della progettazione orientata agli oggetti, le prospettive di conceptual (concettuale), specification (specificazione), e implementation (implementazione) forniscono diversi approcci alla generalizzazione:

- **Conceptual (concettuale):** In questa prospettiva, la relazione è basata sulla relazione di sottoinsieme, dove una classe più specifica è considerata un sottoinsieme della classe più generale. Questo riflette la gerarchia concettuale di come i concetti sono organizzati.
- **Specification (specificazione):**Qui, la sottoclasse deve conformarsi all'interfaccia della superclasse. Questa prospettiva è più orientata alle specifiche dell'interfaccia e garantisce che le sottoclassi possano essere utilizzate ovunque venga utilizzata la superclasse.
- **Implementation (implementazione):**Questa prospettiva coinvolge l'ereditarietà dell'implementazione e la creazione di sottoclassi che estendono o specializzano la superclasse. Questo è spesso associato alla programmazione basata sulla classe e coinvolge l'ereditarietà di attributi e metodi.

Tuttavia, è importante notare l'avvertimento: l'ereditarietà dovrebbe essere utilizzata solo quando è supportata a livello concettuale. Se la relazione concettuale non lo giustifica, è preferibile utilizzare l'aggregazione. Ad esempio, una pila (stack) non è una lista con alcune sostituzioni, e se la relazione concettuale non è adeguata, l'ereditarietà può portare a modelli non intuitivi o incoerenti.

Multiple inheritance

L'ereditarietà multipla è un concetto che consente a una classe di ereditare attributi e comportamenti da più di una superclasse. In altri termini, una classe può avere più di una classe genitore da cui ereditare proprietà.



Questo approccio può portare a una maggiore flessibilità nella progettazione delle classi, consentendo di comporre funzionalità diverse provenienti da più fonti. Tuttavia, l'ereditarietà multipla può anche portare a complessità nel sistema, come ad esempio ambiguità nei casi in cui due superclassi forniscono metodi con lo stesso nome.

In linguaggi di programmazione che supportano l'ereditarietà multipla, come C++ o Python, è possibile dichiarare una classe che eredita da più classi. Questa caratteristica può essere utilizzata con attenzione per migliorare la riusabilità del codice e la strutturazione della gerarchia delle classi, ma deve essere gestita con cautela per evitare potenziali problemi di ambiguità o complessità.

La discussione sull'ereditarietà multipla comporta vantaggi e svantaggi significativi.

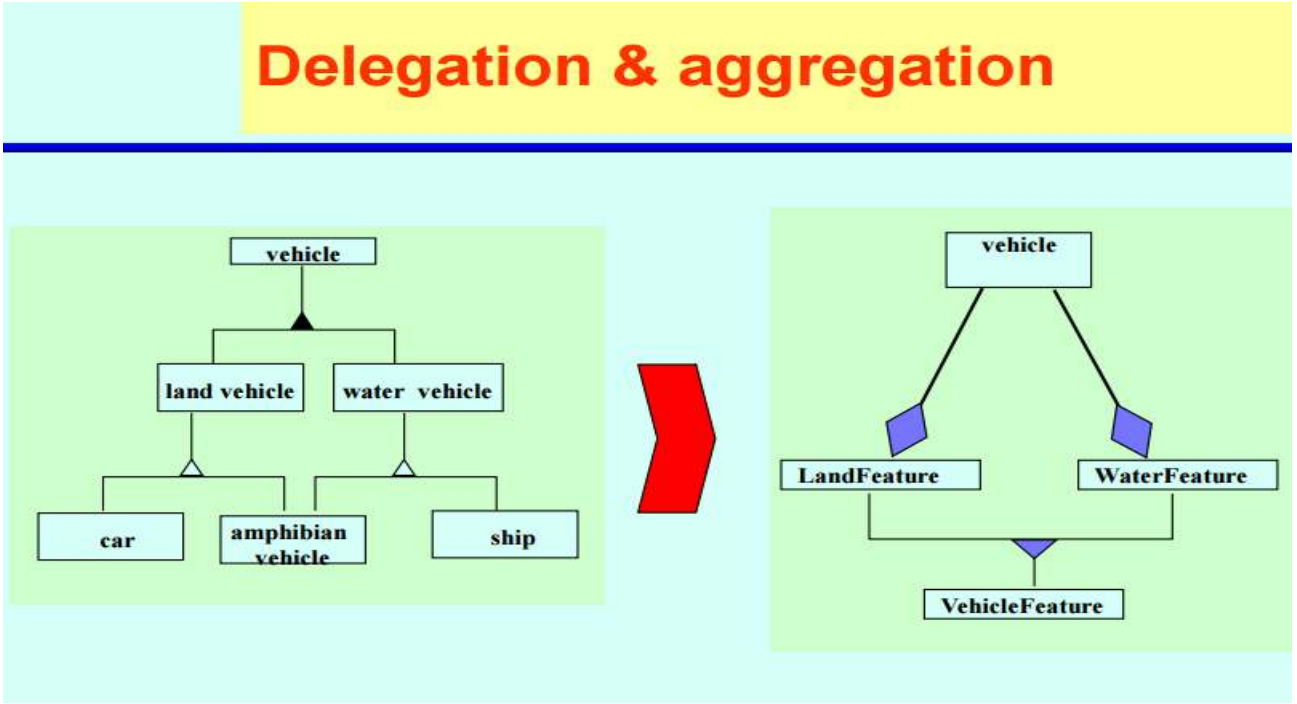
Vantaggi:

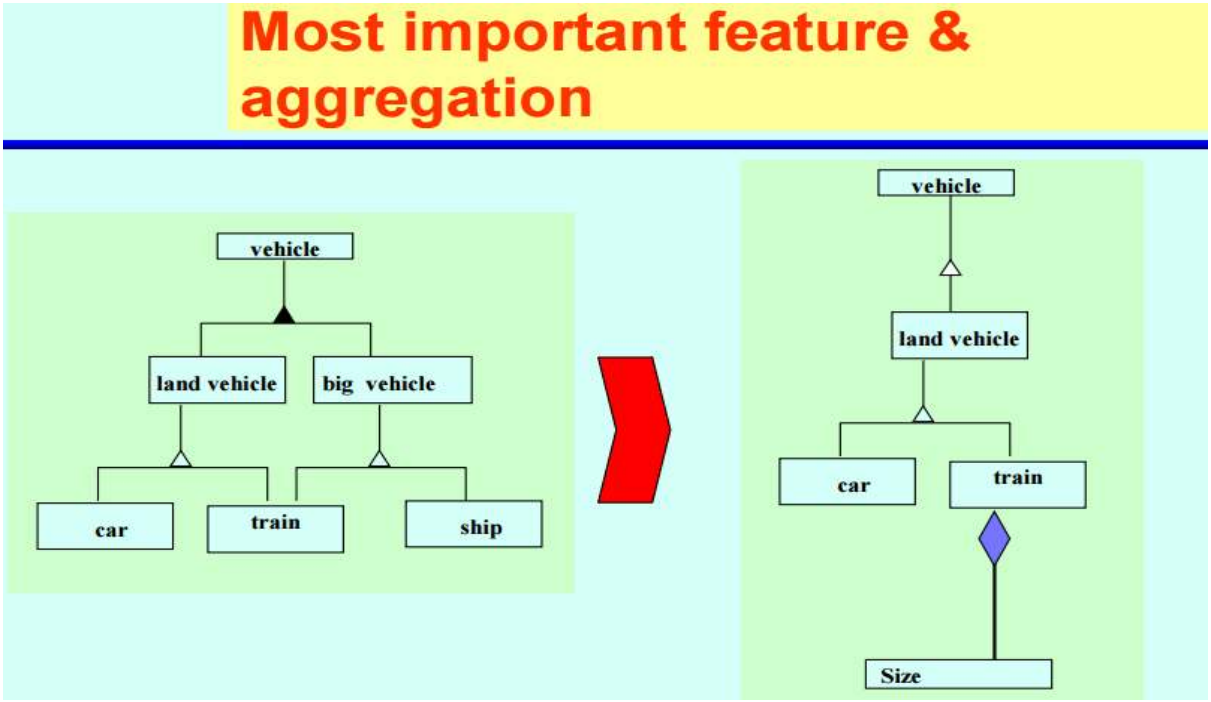
- **Prossimità al pensiero umano:**L'ereditarietà multipla può riflettere più fedelmente la complessità delle relazioni concettuali nel mondo reale, consentendo di modellare oggetti che sono associati a molteplici concetti.
- **Flessibilità elevata per la specifica delle classi:**L'abilità di ereditare da più fonti offre una maggiore flessibilità nella progettazione delle classi, consentendo di comporre funzionalità diverse provenienti da diverse classi genitore.
- **Maggiori opportunità per il riutilizzo:** L'ereditarietà multipla può aumentare le opportunità di riutilizzo del codice, poiché una classe può ereditare funzionalità da più classi, riducendo la necessità di scrivere codice simile più volte.

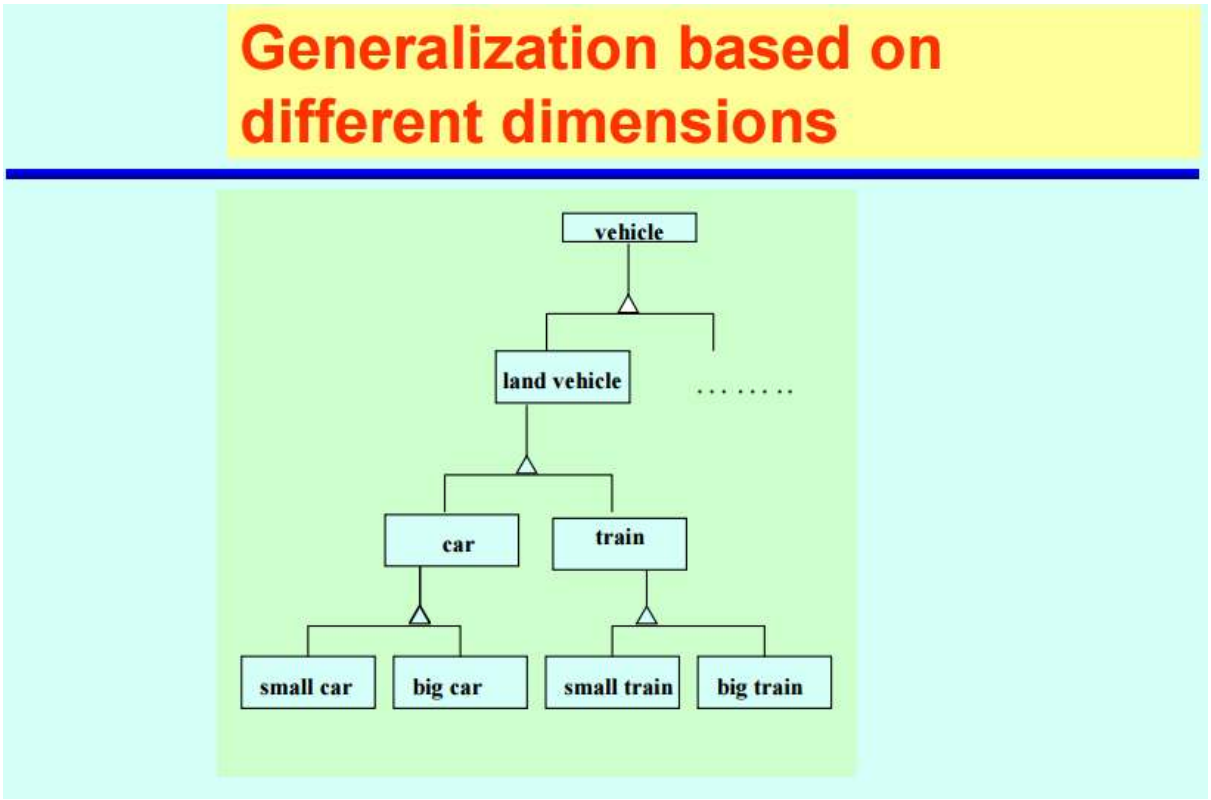
Svantaggi:

- **Perdita di chiarezza su quale metodo viene eseguito:**Quando una classe eredita da più superclassi, può sorgere l'ambiguità su quale metodo specifico viene chiamato, portando a una perdita di chiarezza nella comprensione del codice.
- **Complessità di implementazione elevata:** L'implementazione di ereditarietà multipla può diventare più complessa, specialmente quando si gestiscono molteplici fonti di funzionalità e comportamenti.
- **Necessità di risolvere conflitti:**La risoluzione di conflitti tra le funzionalità ereditate da diverse classi può richiedere attenzione speciale, aggiungendo un livello di complessità al processo di sviluppo.

Evitare l'ereditarietà multipla è, fondamentalmente, una questione di implementazione. Spesso, il modo più semplice è ristrutturare il modello. Tecniche comuni di ristrutturazione includono l'utilizzo di delega e aggregazione, l'ereditarietà basata sulla caratteristica più importante con delega del resto, e la generalizzazione basata su dimensioni diverse. La scelta tra queste tecniche dipende dalle specifiche esigenze del progetto e dalla struttura desiderata per evitare complicazioni e ambiguità associate all'ereditarietà multipla.







When to use class diagrams

I diagrammi delle classi sono fondamentali nello sviluppo orientato agli oggetti. Tuttavia, è importante utilizzare queste rappresentazioni con saggezza. Evita di sovraccaricare i diagrammi con troppe notazioni e inizia con concetti semplici. Adatta il livello di dettaglio al contesto: limita i dettagli nell'analisi, concentrandoti sulla specifica piuttosto che sull'implementazione. Concentrati su aree chiave del sistema e preferisci pochi diagrammi

aggiornati rispetto a molti modelli obsoleti per mantenere la chiarezza e la rilevanza del tuo lavoro.

Creating a class diagram

1. **Inizia con le Classi Principali:**
    - Identifica le classi principali coinvolte nel sistema. Queste sono le entità fondamentali che costituiranno la base del tuo diagramma.
2. **Aggiungi Associazioni Ovvie:**
    - Identifica le associazioni dirette e ovvie tra le classi principali. Questo può essere il punto di partenza per visualizzare le relazioni fondamentali tra le entità.
3. **Aggiungi Attributi:**
    - Per ogni classe, aggiungi gli attributi che ne definiscono le caratteristiche. Questi possono essere le proprietà specifiche dell'oggetto rappresentato dalla classe.
4. **Specifica la Moltiplicità:**
    - Definisci la moltiplicità delle associazioni per indicare quante istanze di una classe sono associate a un'altra. Ad esempio, uno-a-uno, uno-a-molti, molti-a-molti.
5. **Aggiungi Operazioni:**
    - Indica le operazioni o i metodi che le classi possono eseguire. Queste sono le azioni o i comportamenti associati a ciascuna classe.
6. **Raffina e Ottimizza:**
    - Rivedi il diagramma, rimuovendo eventuali dettagli superflui o associazioni non necessarie. Mantieni il diagramma chiaro e focalizzato sui concetti chiave.

Quando crei diagrammi delle classi, considera le seguenti linee guida:
Considera che una classe può apparire in diversi diagrammi, favorirendo il riuso e la chiarezza. I diagrammi dovrebbero evidenziare aspetti specifici del sistema, evitando sovraccarichi di informazioni. Limita il numero di classi e associazioni per mantenere la focalizzazione. Nascondi attributi o operazioni non rilevanti per l'obiettivo specifico del diagramma, mantenendo la concisione. Prevedi più iterazioni durante la creazione del diagramma per perfezionare gradualmente il modello.
Evita di creare classi "pesanti":

- Evita che il controller svolga troppe responsabilità, cercando di far fare tutto a una singola classe.
- Limita le altre classi a encapsulare solo dati, senza aggiungere complessità con funzionalità eccessive.

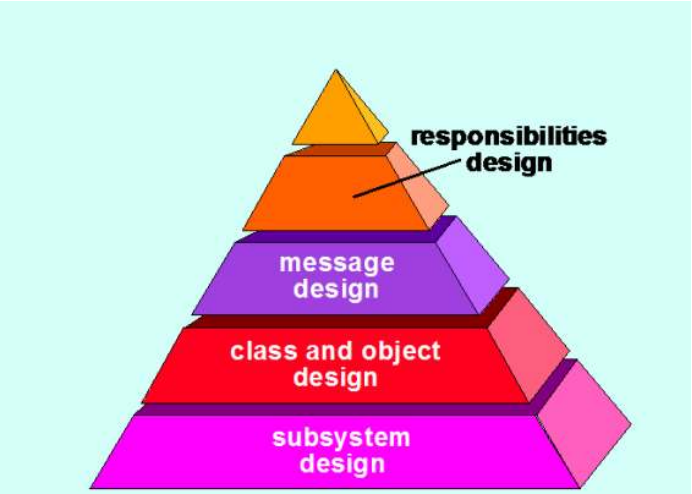
Questo approccio favorisce una struttura più snella e chiara, facilitando la comprensione e manutenzione del codice.

Introduction to the Unified Modeling Language (UML)Part 2

Object Oriented Design in UML

Object-Oriented Design & Design Issues

Nel contesto del Design Orientato agli Oggetti (Object-Oriented Design), le problematiche di progettazione includono aspetti cruciali come decomposability, composability, understandability, continuity e protection.



- **Decomposability (Scomponibilità):**Rappresenta la capacità di scomporre un sistema complesso in componenti più gestibili. In OOD, la decomposizione in classi e oggetti consente di affrontare parti specifiche del problema in modo modulare.
- **Composability (Componibilità)**: Riflette la capacità di progettare componenti (classi, oggetti) che possono essere riutilizzati in diversi contesti, promuovendo la costruzione di sistemi più flessibili e modulari.
- **Understandability (Comprensibilità)**:Indica quanto sia facile comprendere un componente del programma senza dover fare riferimento a informazioni esterne. La chiarezza nella struttura delle classi e la relazione tra gli oggetti contribuiscono a una comprensione più rapida.
- **Continuity (Continuità)**: Si riferisce alla facilità con cui è possibile apportare piccole modifiche in un'area specifica del sistema senza generare impatti diffusi. La progettazione orientata agli oggetti mira a isolare le modifiche locali.
- **Protection (Protezione):**Rappresenta la capacità di ridurre la propagazione degli effetti collaterali in caso di errori in un componente specifico. L'incapsulamento e la gestione degli errori locali contribuiscono a mitigare gli impatti negativi sull'intero sistema.

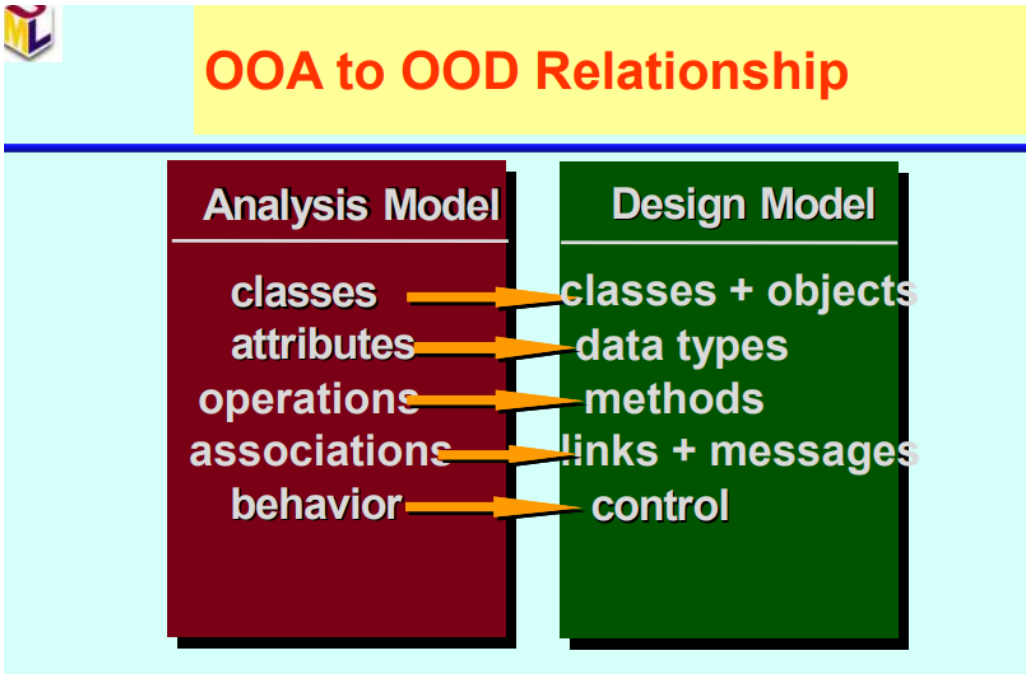
L'Object-Oriented Design promuove la creazione di sistemi modulari, riutilizzabili e comprensibili attraverso la strutturazione dei concetti del dominio in classi e oggetti, facilitando la gestione delle problematiche di progettazione.

Ingredients for OOD

Nel contesto del Design Orientato agli Oggetti (OOD), gli ingredienti fondamentali includono:

1. **Problem Domain Component (Componente del Dominio del Problema):** Questo costituisce i sottosistemi responsabili dell'implementazione diretta dei requisiti del cliente. Le classi e gli oggetti in questa componente modellano concetti specifici del dominio del problema.
2. **Human Interaction Component (Componente di Interazione Umana):**Questo riguarda i sottosistemi che implementano l'interfaccia utente, inclusi sottosistemi GUI riutilizzabili. Questa componente si occupa della presentazione delle informazioni e dell'interazione con gli utenti.
3. **Task Management Component (Componente di Gestione delle Attività):**Questa componente gestisce e coordina attività concorrenti, sia all'interno di un sottosistema che tra sottosistemi differenti. Si occupa di garantire l'ordine e la sincronizzazione delle operazioni.
4. **Data Management Component (Componente di Gestione dei Dati):**Questa componente è responsabile della memorizzazione e del recupero degli oggetti. Gestisce l'accesso ai dati, garantendo la coerenza e l'integrità delle informazioni memorizzate.

Questi componenti contribuiscono alla struttura modulare e alla chiarezza del design orientato agli oggetti, suddividendo le responsabilità in aree specifiche e facilitando la gestione delle complessità del sistema.



What distinguishes OOD from OOA?

Il Design Orientato agli Oggetti (OOD) si differenzia dall'Analisi Orientata agli Oggetti (OOA) per diversi motivi:

Nel dettaglio, OOD si occupa di implementazioni più concrete delle classi e degli oggetti, fissando i nomi in modo più definito. La firma dei messaggi è più precisa, delineando chiaramente i metodi e i loro parametri. La molteplicità e la sua realizzazione sono trattate in modo più specifico, e la visibilità di attributi e metodi è analizzata dettagliatamente, determinando chi può accedere e modificare i componenti del sistema. OOD include anche algoritmi per i metodi, specificando in modo più dettagliato come le operazioni sono effettuate. Inoltre, i diagrammi di sequenza e collaborazione sono più dettagliati, evidenziando l'interazione tra gli oggetti in modo più specifico. Infine, OOD può introdurre notazioni aggiuntive nei diagrammi, offrendo dettagli specifici dell'implementazione che potrebbero essere trascurati in OOA. Nel complesso, OOD si concentra su una visione più concreta e specifica, traducendo le idee concettuali in soluzioni implementabili.

Pertanto, nel Design Orientato agli Oggetti (OOD), i diagrammi delle classi rimangono presenti ma vengono perfezionati per adattarsi alla progettazione del sistema. Oltre ai diagrammi delle classi, si introducono diverse altre rappresentazioni grafiche:

Diagrammi Strutturali:

- Diagrammi degli Oggetti: Mostrano istanze specifiche di classi e le loro relazioni.
- Diagrammi di Deployment:Rappresentano la disposizione fisica dei componenti del sistema.

Diagrammi Comportamentali:

- Diagrammi di Sequenza:Illustrano l'interazione sequenziale tra gli oggetti nel sistema.
- Diagrammi di Collaborazione:Forniscono una visione più globale delle interazioni tra gli oggetti.
- Diagrammi di Statechart: Rappresentano gli stati e le transizioni di uno o più oggetti durante il loro ciclo di vita.
- Diagrammi di Attività:Mostrano il flusso delle attività all'interno del sistema.

Questi diagrammi aggiuntivi forniscono una panoramica completa del design, affrontando aspetti strutturali e comportamentali. Ogni tipo di diagramma si concentra su un aspetto specifico del sistema, consentendo ai progettisti di comunicare in modo efficace e comprendere meglio il funzionamento complessivo del sistema.

Structural Diagrams for OOD in UML

Nei Diagrammi Strutturali per il Design Orientato agli Oggetti (OOD) in UML, i Diagrammi delle Classi mantengono la stessa struttura utilizzata nell'Analisi Orientata agli Oggetti (OOA). Tuttavia, si affinano per adattarsi alla progettazione del sistema.

Parallelamente, compaiono i Diagrammi degli Oggetti, i quali si concentrano sugli oggetti, ovvero le istanze specifiche delle classi. Essi risultano essere essenzialmente equivalenti ai diagrammi delle classi, differenziandosi nella rappresentazione degli oggetti anziché delle classi. Questa equivalenza semplifica l'analisi dei diagrammi degli oggetti in quanto non richiede una trattazione più approfondita rispetto ai diagrammi delle classi.

Complessivamente, questa struttura di diagrammi strutturali in OOD consente una transizione fluida dall'analisi alla progettazione, mantenendo coerenza nella rappresentazione delle classi e degli oggetti nel sistema.

Behavioral Diagrams for OOD in UML

Nei Diagrammi Comportamentali per il Design Orientato agli Oggetti (OOD) in UML, troviamo diverse rappresentazioni:
Statechart Diagrams:
Descrivono l'evoluzione degli stati di qualsiasi classificatore nel sistema, comunemente utilizzati per modellare gli oggetti e come essi attraversano vari stati durante il loro ciclo di vita.
Activity Diagrams:
Descrivono l'evoluzione delle attività nel sistema, mostrando il flusso delle operazioni e delle azioni tra gli oggetti.
Sequence Diagrams:
Descrivono le interazioni temporali tra gli oggetti, evidenziando l'ordine cronologico delle operazioni.
Collaboration Diagrams:
Descrivono le interazioni tra gli oggetti organizzate in base alle collaborazioni tra di loro.

In breve, questi diagrammi consentono di visualizzare in dettaglio il comportamento dinamico del sistema, concentrando l'attenzione su come gli oggetti interagiscono e come si evolvono nel tempo durante l'esecuzione delle attività.

Statechart diagrams vs Interaction diagrams

I diagrammi di Statechart e quelli di Interazione sono strumenti distinti nel contesto del Design Orientato agli Oggetti (OOD) in UML.

I **diagrammi di Interazione** mettono in luce le dinamiche di come gli oggetti interagiscono tra di loro. Si concentrano sulla visualizzazione delle comunicazioni e delle sequenze temporali delle operazioni, offrendo una prospettiva chiara sulle relazioni e gli scambi di informazioni tra gli elementi del sistema.

D'altra parte, i diagrammi di **Statechart** si concentrano sul comportamento di un singolo oggetto. Essi analizzano in profondità come quell'oggetto specifico cambia il suo stato in risposta ai messaggi che riceve. Questa rappresentazione è più focalizzata e dettagliata, offrendo una visione più specifica e fine-grained delle transizioni di stato di un oggetto durante il suo ciclo di vita.

In sintesi, mentre i diagrammi di Interazione forniscono una panoramica delle dinamiche tra gli oggetti nel sistema, i diagrammi di Statechart offrono un'analisi più approfondita e specifica del comportamento di un singolo oggetto, concentrandosi sulle sue transizioni di stato. Entrambi sono strumenti essenziali, ma con obiettivi distinti all'interno del processo di progettazione orientata agli oggetti.

Object states

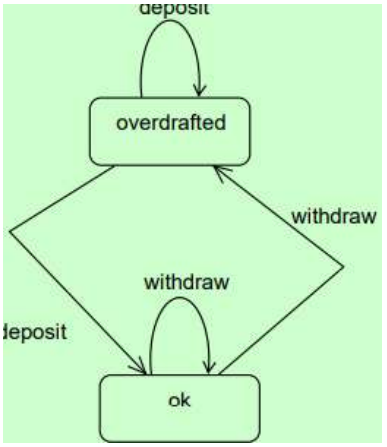
Gli stati degli oggetti rappresentano i diversi insiemi di valori che descrivono un oggetto in un dato momento. In altre parole, lo stato di un oggetto è il risultato dell'insieme specifico di valori che i suoi attributi assumono in un determinato istante. Questa concezione di stato è cruciale per comprendere e modellare il comportamento dinamico degli oggetti in un sistema orientato agli oggetti.

Ogni oggetto attraversa una serie di stati durante il suo ciclo di vita, e lo stato attuale è determinato dalle informazioni contenute nei suoi attributi. Ogni stato riflette un'istanza unica dell'oggetto, catturando le sue caratteristiche e le sue condizioni in un preciso momento temporale.

Quindi, quando parliamo di "stato" in un contesto di progettazione orientata agli oggetti, ci riferiamo a una rappresentazione specifica e istantanea dell'oggetto, definita in base ai valori attuali dei suoi attributi. Questo concetto di stato gioca un ruolo fondamentale nella modellazione delle dinamiche degli oggetti e nella comprensione di come essi rispondono alle varie interazioni e ai cambiamenti nell'ambiente del sistema.

State changes

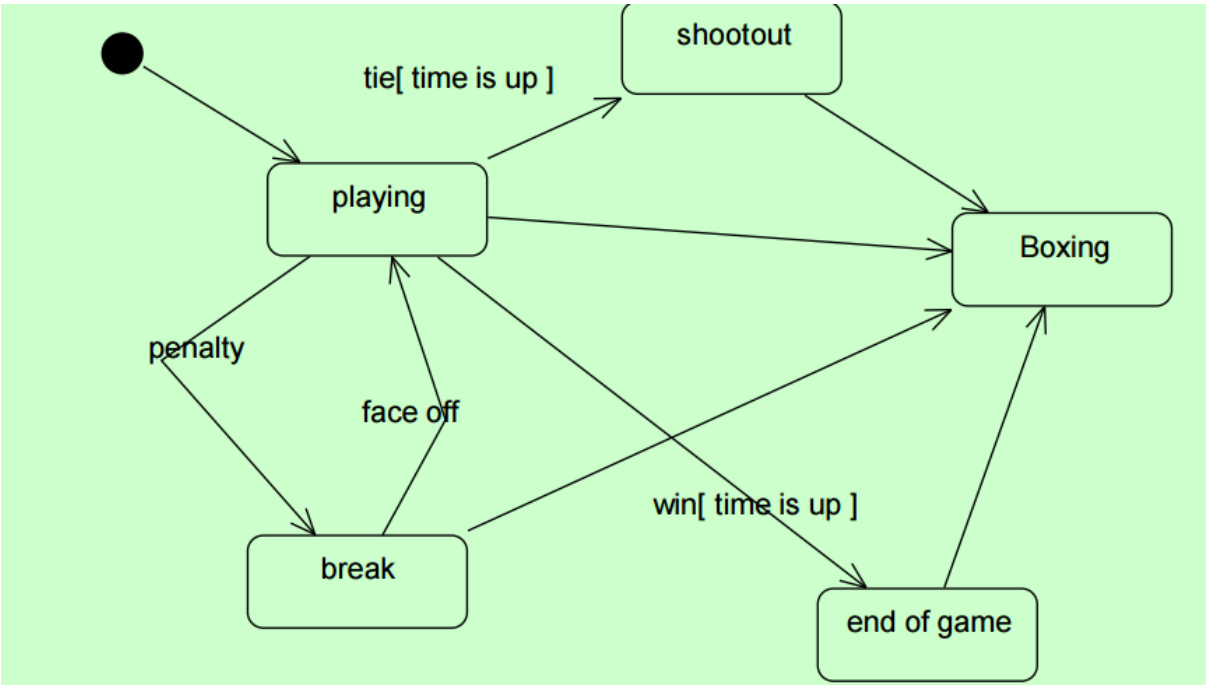
Le transizioni di stato rappresentano il passaggio da uno stato all'altro all'interno del ciclo di vita di un oggetto. Queste transizioni avvengono in risposta a eventi specifici, che possono essere interpretati come messaggi ricevuti da un oggetto. Gli eventi, in questo contesto, sono stimoli o richieste che possono originare da altri oggetti o dal sistema stesso.



Quando si parla di "eventi" in relazione alle transizioni di stato, si fa riferimento a messaggi che vengono inviati o ricevuti dagli oggetti. Questi eventi possono o non possono causare un cambiamento nello stato dell'oggetto. In alcuni casi, un evento può essere irrilevante rispetto allo stato attuale dell'oggetto, mentre in altri può innescare una transizione di stato significativa.

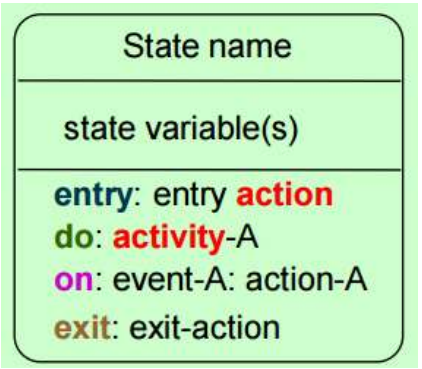
Quindi, in sintesi, le transizioni di stato si verificano quando gli oggetti ricevono eventi, ovvero messaggi, e la loro risposta a tali eventi può comportare un cambiamento nel loro stato. Questo meccanismo di transizione di stato è fondamentale per modellare il comportamento dinamico degli oggetti nel contesto di un sistema orientato agli oggetti.

Example of Statechart Diagrams: States of a hockey game



La notazione dei diagrammi di stato è una rappresentazione grafica utilizzata per descrivere il comportamento di un sistema in termini di stati, transizioni tra stati e le azioni associate a ciascuno stato. Nella notazione dei diagrammi di stato, i concetti principali includono:

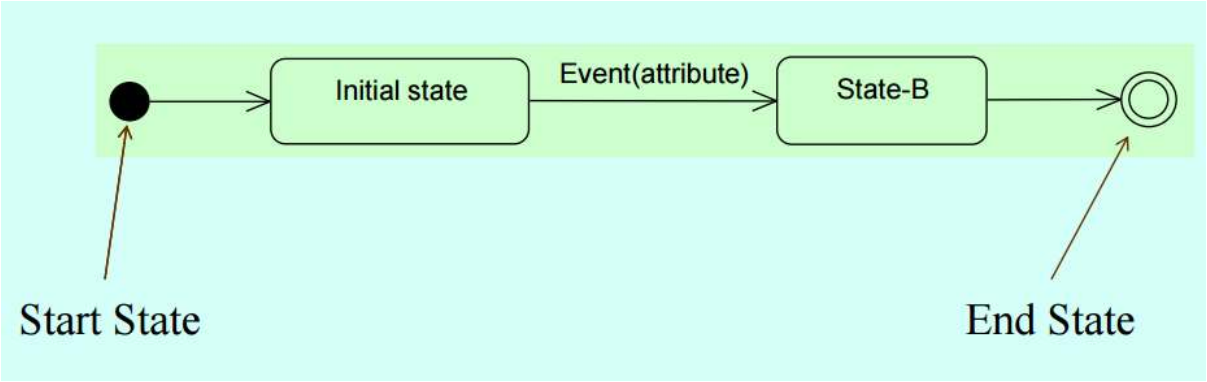
- **Attività (Activity)**:Rappresenta un'azione che può richiedere più tempo e può essere interrotta. Ad esempio, potrebbe rappresentare un processo complesso o una procedura che richiede un certo periodo di tempo per essere completata.
- **Azione (Action):** Rappresenta un'azione che si verifica rapidamente. Questo termine è utilizzato per indicare un'attività che ha una durata molto breve.
- **Occur quickly(Avviene rapidamente):** In questo contesto, "rapidamente" significa che l'azione si verifica in un breve periodo di tempo, senza richiedere una durata prolungata.
- **Entry (ingresso):**Un'azione eseguita quando si entra in uno stato. Ad esempio, può rappresentare l'inizializzazione di variabili o la preparazione di risorse quando si entra in uno stato specifico.
- **Do (eseguire):**Un'attività continua eseguita mentre si è nello stato. Ad esempio, potrebbe rappresentare l'aggiornamento costante di un display durante lo stato.
- **On (su):**Un'azione eseguita in risposta a un evento specifico mentre si è nello stato. Ad esempio, potrebbe rappresentare l'attivazione di una funzione quando si verifica un determinato evento.
- **Exit (uscita):**Un'azione eseguita quando si esce da uno stato. Ad esempio, può rappresentare la pulizia delle risorse o l'archiviazione dei risultati quando si esce da uno stato specifico.



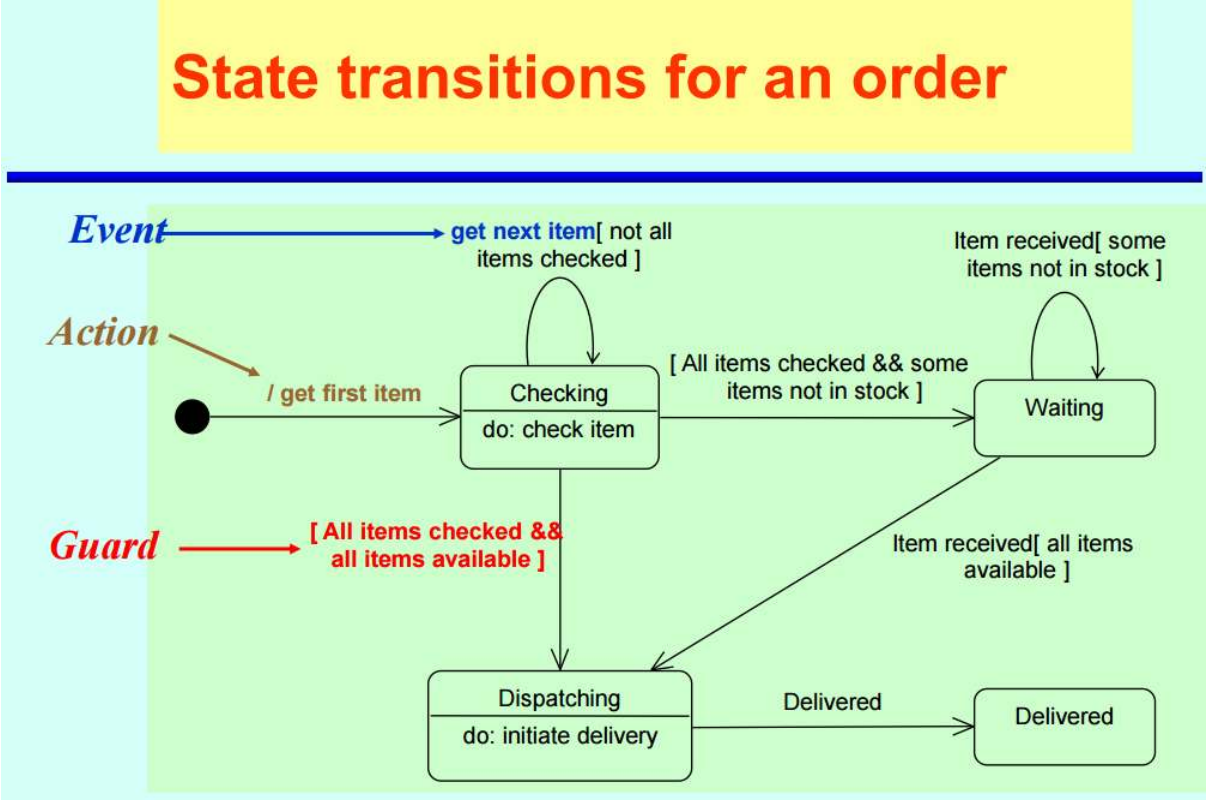
Questa specifica si riferisce a elementi comuni nei diagrammi di stato, aggiungendo dettagli specifici alla rappresentazione di transizioni tra gli stati:

- **Event (Evento)**:Rappresenta un'azione o un segnale che può innescare una transizione da uno stato all'altro. Nel tuo caso, l'evento specifico è "message send" (invio di un messaggio).
- **Guard Condition (Condizione di Guardia):**Una condizione logica associata a una transizione. La transizione avviene solo se la condizione di guardia è valutata come vera. Può essere omessa se la transizione deve verificarsi senza alcuna condizione aggiuntiva.Le condizioni di guardia della transizione che esce da uno stato sono mutualmente esclusive.Se ci sono più transizioni che escono da uno stato, le loro condizioni di guardia dovrebbero essere tali da essere mutualmente esclusive. In altre parole, solo una di esse dovrebbe poter essere soddisfatta in un dato momento.
- **Action (Azione):**Rappresenta un'azione che si verifica rapidamente e che non può essere interrotta. Questo concetto è in linea con quanto già menzionato, dove "action" rappresenta un'azione di breve durata e non interrompibile.

Non è necessario includere tutti questi dettagli in ogni transizione. Possiamo scegliere di omettere parti che non sono rilevanti o necessarie per la specifica rappresentazione del sistema. Ad esempio, si potrebbe avere una transizione semplice senza condizioni di guardia o azioni associate.



Example of Statechart Diagrams (2): Order Management



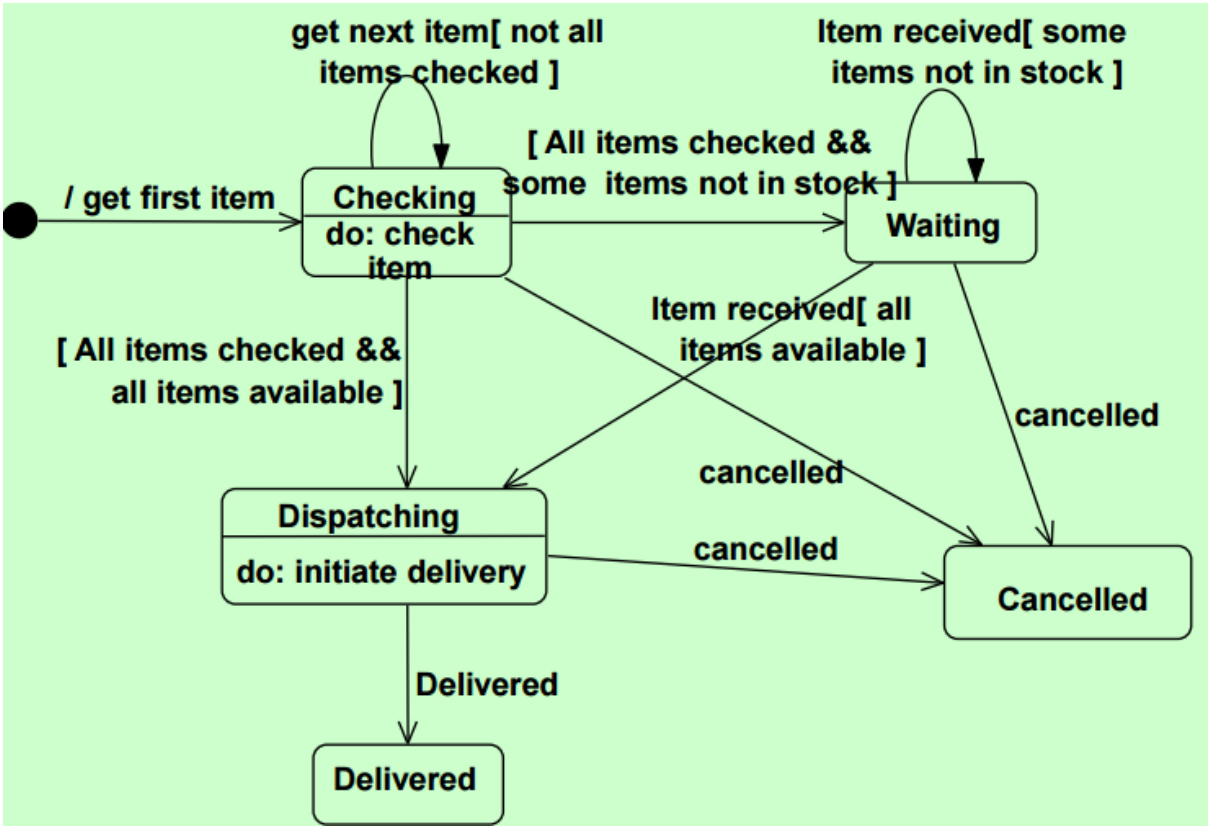
Nel corso del processo di gestione degli ordini, **può verificarsi una situazione in cui gli utenti desiderano o necessitano di annullare un ordine precedentemente effettuato**. Questa esigenza potrebbe derivare da vari fattori, come un cambio improvviso di decisione da parte del cliente, problemi di disponibilità dei prodotti o eventuali imprevisti che rendono necessario l'annullamento dell'ordine.

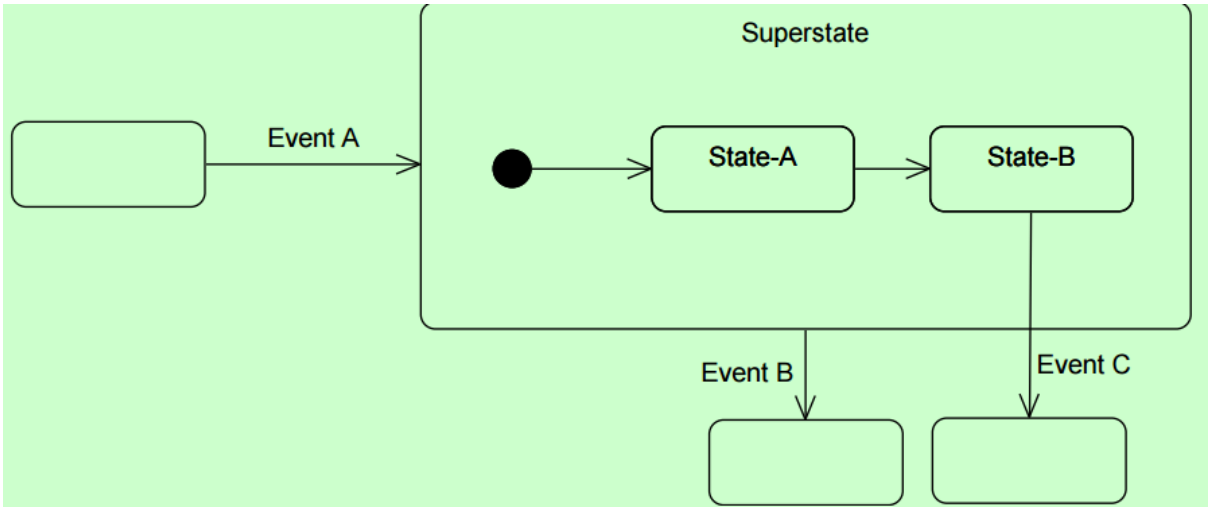
Per risolvere questa problematica e garantire un'esperienza utente fluida e flessibile, sono state ideate soluzioni specifiche. L'obiettivo è fornire agli utenti la libertà di annullare un ordine in qualsiasi fase del processo, rispondendo così alle loro esigenze dinamiche.

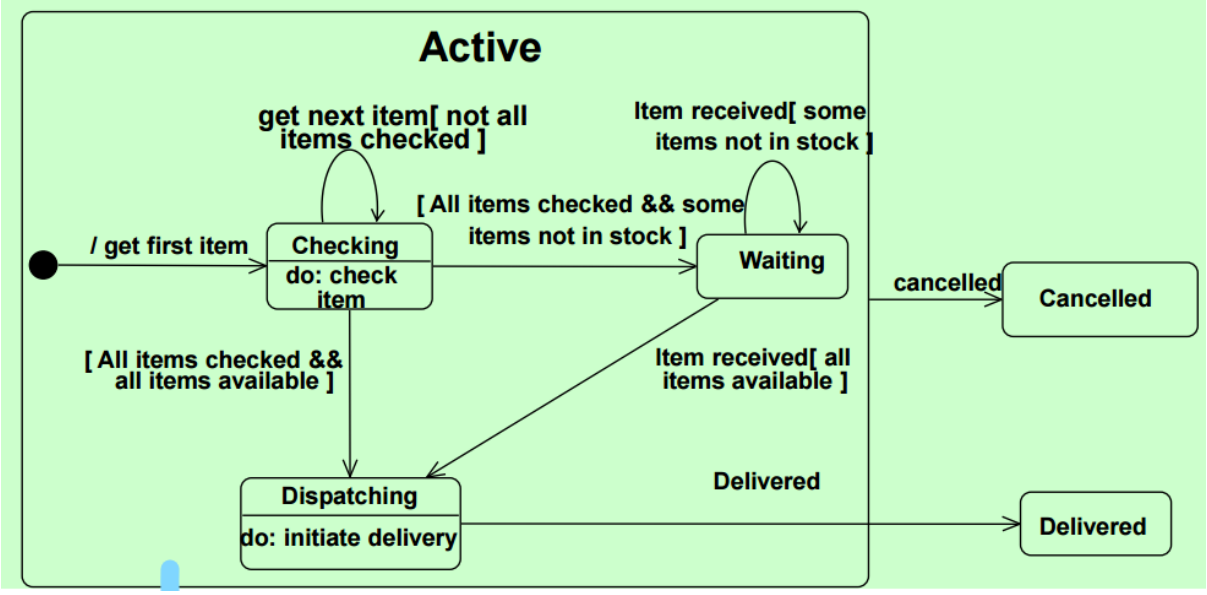
Soluzioni proposte:

- **Transizioni da ogni stato allo stato "annullato":** Questa soluzione prevede la creazione di transizioni dirette da ciascuno stato possibile al nuovo stato "annullato". In questo modo, qualsiasi ordine, indipendentemente dalla fase in cui si trova, può essere immediatamente annullato senza complessi percorsi intermedi.
- **Utilizzo di uno stato superiore (superstate) con una singola transizione:**Alternativamente, si può introdurre uno stato superiore che rappresenta il concetto generale di "annullamento". Da questo stato superiore, una singola transizione può condurre direttamente allo stato "annullato". Questo approccio semplifica la gestione delle transizioni e fornisce una rappresentazione più chiara del processo di annullamento nell'ambito complessivo del diagramma di stato.

Entrambe le soluzioni sono progettate per rispondere alla necessità improvvisa di annullare un ordine e consentono una gestione flessibile degli ordini nell'intero sistema.





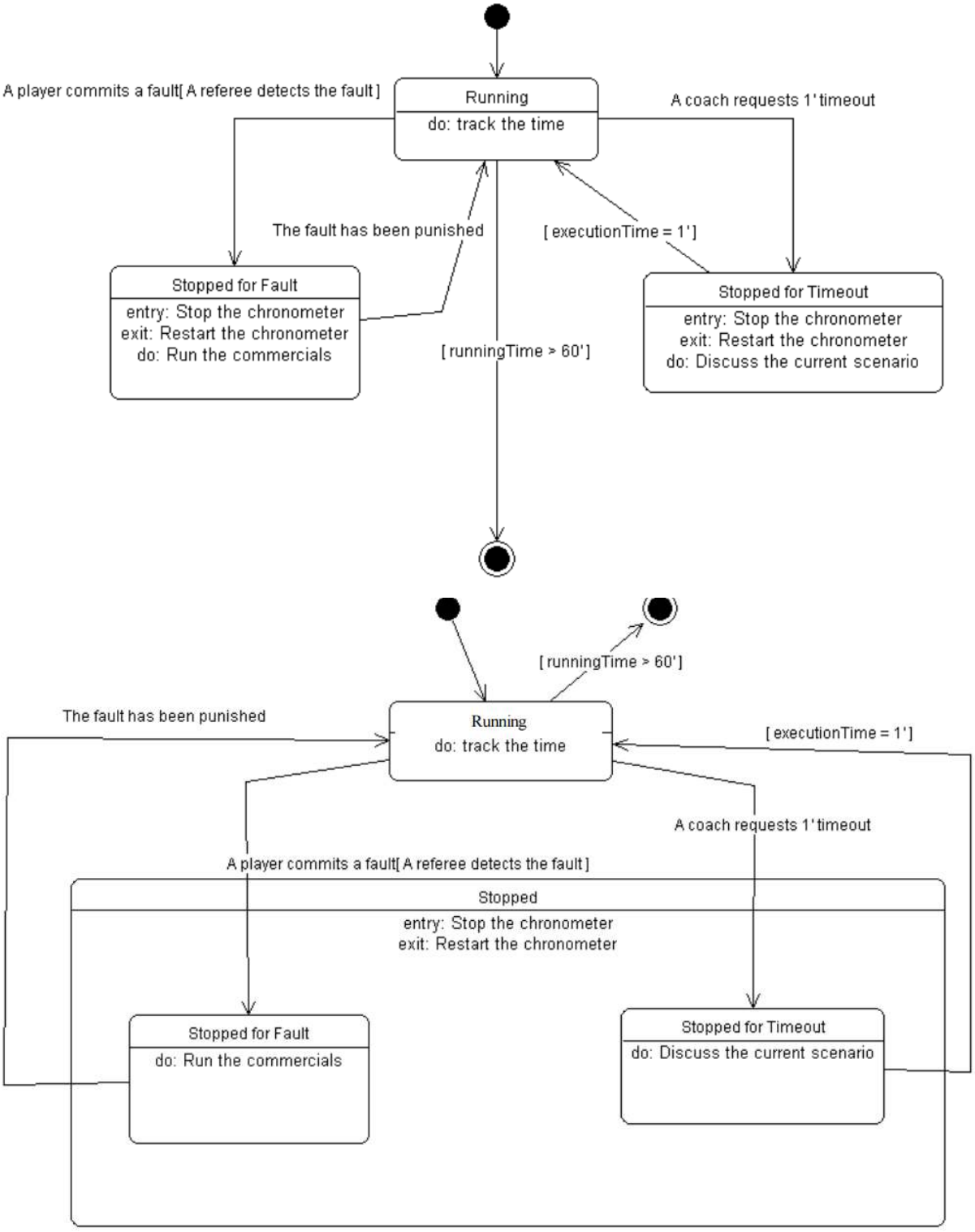


I diagrammi di stato NON DEVONO NECESSARIAMENTE fare riferimento a classi o oggetti, ma possono altresì fare riferimento a sottosistemi, ecc...
Tuttavia, spesso questa è l'applicazione più diffusa dei diagrammi di stato.

Nel contesto delle rappresentazioni visive attraverso i diagrammi di stato, è importante sottolineare che questi non sono limitati a descrivere solo le relazioni tra classi o oggetti. In realtà, la loro versatilità consente di modellare anche altri aspetti del sistema, come sottosistemi o componenti di alto livello.

Quando si progetta un sistema complesso, l'uso di diagrammi di stato può estendersi oltre la rappresentazione degli stati e delle transizioni specifiche di un oggetto. Possono essere impiegati in modo più ampio per delineare lo stato di un intero sottosistema o di un componente del sistema, offrendo una panoramica completa del comportamento del sistema.

Tuttavia, va notato che, nonostante la loro flessibilità, l'uso più comune e ampiamente diffuso dei diagrammi di stato rimane la rappresentazione delle transizioni di stato di oggetti specifici. Questa applicazione più tradizionale fornisce una comprensione dettagliata del comportamento di singoli elementi nel contesto di un sistema più ampio.
Define the statechart diagram of a basketball game



Activity diagrams

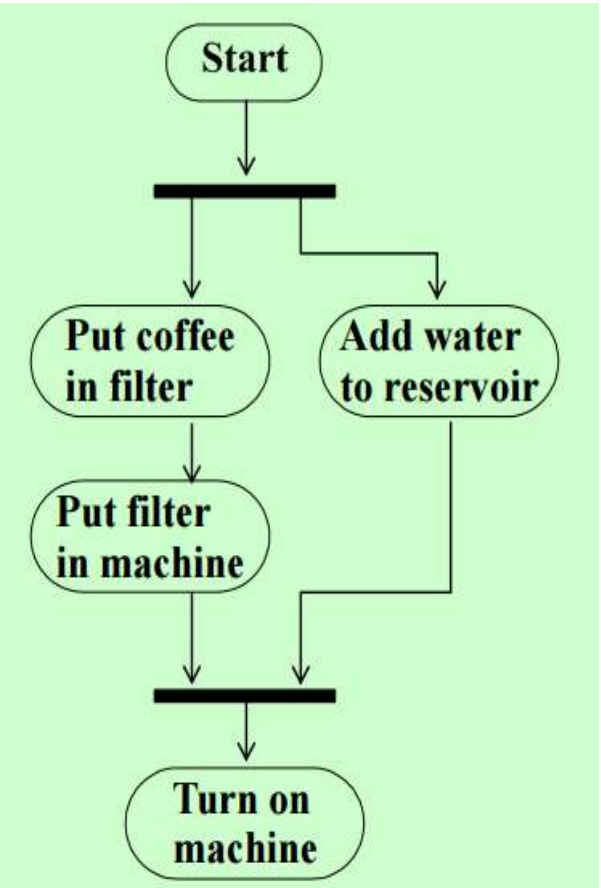
I diagrammi di attività sono utilizzati per descrivere il flusso di lavoro e l'elaborazione parallela in un sistema. Essi forniscono una rappresentazione visiva delle attività e delle azioni che avvengono all'interno di un processo o di un sistema.
Attività:

- **Concettuali**: Rappresentano compiti o attività da eseguire nel contesto di un processo. Queste attività sono di natura concettuale e non sono ancora associate a metodi specifici o implementazioni tecniche.
- **Specifiche/Implementative:** Rappresentano l'implementazione concreta di un'attività, spesso associata a un metodo all'interno di una classe. In questa fase, le attività sono collegate più strettamente all'implementazione tecnica.

Questi diagrammi sono simili ai Petri nets, che sono strumenti matematici utilizzati per la descrizione formale di sistemi distribuiti. Entrambi forniscono una rappresentazione grafica di processi e flussi di lavoro, consentendo di visualizzare in modo chiaro le attività, le decisioni e le dipendenze tra di esse.

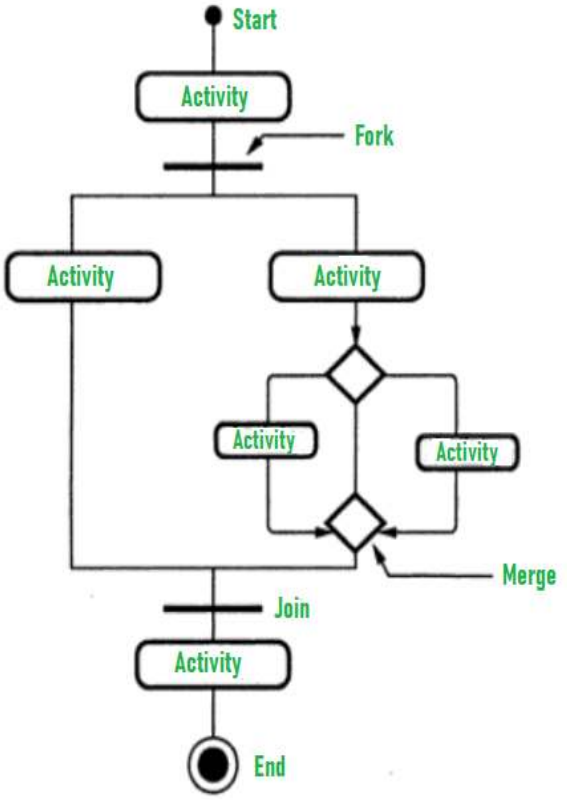
In breve, i diagrammi di attività sono uno strumento potente per modellare e comunicare i flussi di lavoro e le attività all'interno di un sistema, fornendo una comprensione visiva che facilita la progettazione e l'analisi dei processi.

Example of activity diagrams (the coffee pot)



Structure of activity diagrams

Un diagramma delle attività è una rappresentazione visiva di un processo o di un flusso di lavoro. Nella sua struttura, ci sono elementi chiave come "**Fork**" (ramificazione), "**Join**" (unione) e la **"Synchronization Bar"** (barra di sincronizzazione).



Structure of Activity Diagram

Il "Fork" indica una divisione del flusso in più percorsi paralleli, consentendo che le attività vengano eseguite in modo indipendente l'una dall'altra e in qualsiasi ordine. Il "Join" è l'opposto del "Fork" e indica il punto in cui i percorsi paralleli si riuniscono nuovamente, richiedendo che tutte le attività precedenti siano completate prima che il flusso possa continuare.

La "Synchronization Bar" rappresenta la sincronizzazione delle attività. Le attività possono essere svolte in parallelo, il che significa che possono avvenire in qualsiasi ordine. Tuttavia, la barra di sincronizzazione sottolinea che tutte le attività precedenti devono essere completate prima che il flusso prosegua.

Complessivamente, il diagramma delle attività mostra un ordine parziale delle attività, evidenziando le relazioni temporali e le dipendenze tra di esse, mentre i concetti di "Fork", "Join" e "Synchronization Bar" contribuiscono a gestire la parallelizzazione e la sincronizzazione nel flusso di lavoro.

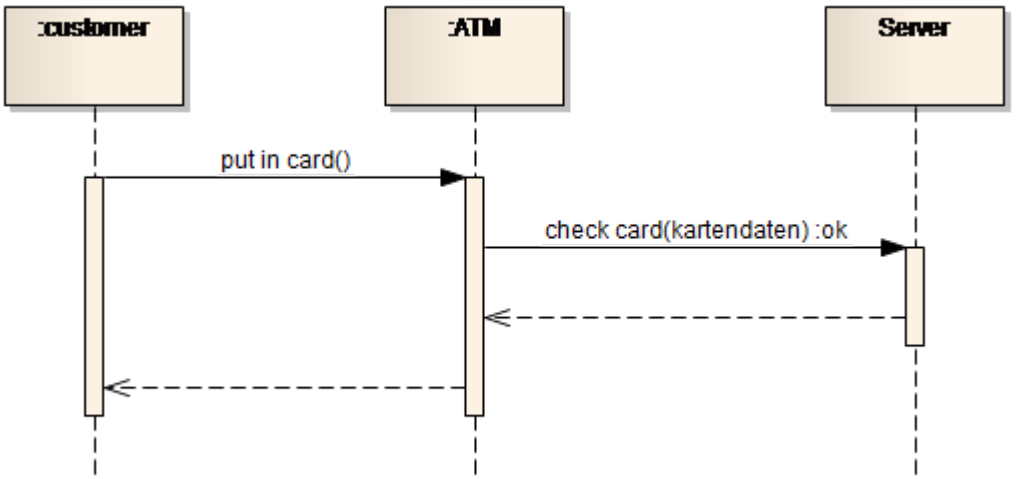
Nei diagrammi delle attività, i "Branch" rappresentano separazioni nel flusso, spesso associati a decisioni. Ad esempio, un ramo può indicare un percorso positivo e un altro uno negativo in base a una "Condition" (condizione). Le condizioni sono espressioni logiche che determinano quale ramo seguire. Questi elementi consentono di modellare la logica decisionale nei processi, rendendo i diagrammi delle attività efficaci per rappresentare flussi di lavoro complessi.

Interaction diagrams

I diagrammi di interazione si concentrano su entità "reali", ovvero gli oggetti, e dettagliano come gli oggetti comunicano tra loro. Questi diagrammi offrono due visualizzazioni principali:

1. **Vista basata sul tempo (Time-based view):** Questa visualizzazione evidenzia la sequenza temporale delle interazioni tra gli oggetti. Mostra chiaramente l'ordine in cui gli oggetti comunicano nel corso del tempo, rivelando come ciascuna interazione influenza il sistema nel suo complesso.
2. **Vista basata sull'organizzazione (Organization-based view):**Questa visualizzazione si concentra sulla struttura organizzativa degli oggetti e sulle relazioni tra di essi. Mette in evidenza come gli oggetti sono organizzati e come si influenzano reciprocamente, offrendo una prospettiva sulla struttura interna del sistema.

In sintesi, i diagrammi di interazione forniscono una comprensione dettagliata di come gli oggetti si scambiano messaggi e collaborano all'interno di un sistema. Le due visualizzazioni, basate sul tempo e sull'organizzazione, offrono prospettive complementari per analizzare e progettare l'interazione tra gli oggetti in modo efficace.

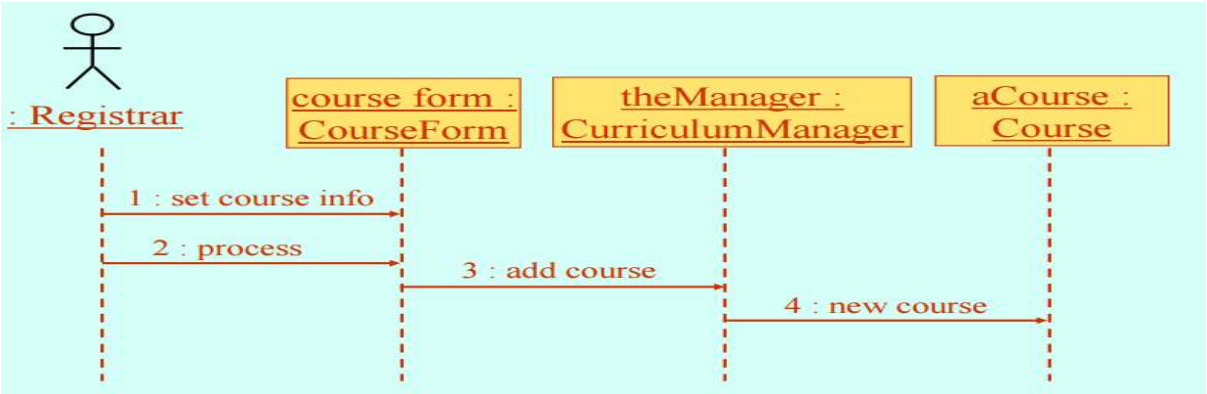


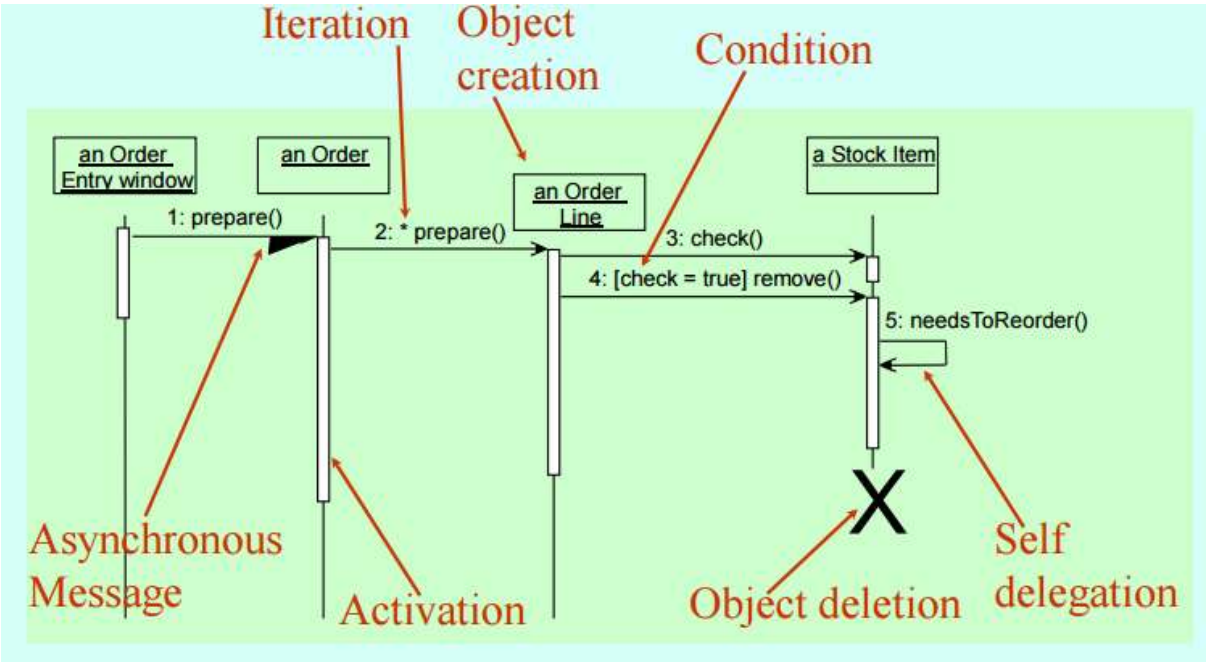
Sequence diagrams

I diagrammi di sequenza mostrano le interazioni degli oggetti organizzate in sequenza temporale. Questi diagrammi si concentrano su:

1. **Oggetti (e classi)**:Rappresentano le entità coinvolte nelle interazioni. Gli oggetti sono disposti lungo una linea orizzontale per indicare la loro partecipazione e il flusso delle interazioni.
2. **Scambio di messaggi:**Si focalizzano sugli scambi di messaggi tra gli oggetti per eseguire le funzionalità degli scenari. Le frecce verticali indicano la direzione e il flusso dei messaggi tra gli oggetti durante l'interazione.

Il diagramma organizza gli oggetti su una linea orizzontale e gli eventi su una linea temporale verticale, creando una rappresentazione chiara delle sequenze temporali delle interazioni. In breve, i diagrammi di sequenza offrono una visualizzazione intuitiva e dettagliata delle dinamiche di interazione tra gli oggetti in un sistema.





I diagrammi di sequenza contengono i seguenti elementi principali:

- **Oggetti:**Rappresentano le entità coinvolte nelle interazioni e scambiano messaggi tra di loro. Gli oggetti sono posizionati lungo una linea orizzontale per mostrare la sequenza temporale delle loro partecipazioni.
- **Messaggi:** Le interazioni tra gli oggetti sono descritte attraverso messaggi. I messaggi possono essere di diversi tipi, tra cui:
    - **Sincroni:** Indicati da "call events" e rappresentati da frecce piene. Questi messaggi denotano chiamate di metodo o funzioni in cui l'oggetto chiamante attende una risposta prima di procedere.
    - **Asincroni:** Indicati da "signals" e rappresentati da frecce a metà. Questi messaggi indicano comunicazioni che non richiedono un'attesa immediata per la risposta. L'oggetto chiamante può continuare senza attendere una risposta.
- **Messaggi «create» e «destroy»:** Indicano la creazione o la distruzione di un oggetto nell'interazione.

Questi elementi combinati consentono di visualizzare in modo chiaro e dettagliato come gli oggetti comunicano tra loro e scambiano informazioni durante l'esecuzione di uno scenario specifico.

Asynchronous messages

Nei diagrammi di sequenza, i messaggi asincroni costituiscono un elemento dinamico che aggiunge flessibilità alle interazioni tra oggetti. A differenza dei messaggi sincroni, quelli asincroni non bloccano il chiamante, consentendo al flusso principale di procedere senza dover attendere una risposta immediata.

Questi messaggi asincroni possono svolgere diverse azioni. In primo luogo, hanno la capacità di **generare un nuovo thread** di esecuzione, introducendo la possibilità di eseguire attività in modo parallelo, ideale per gestire operazioni concorrenti senza interferire con il flusso principale.

Inoltre, i messaggi asincroni possono essere utilizzati per **creare nuovi oggetti** durante l'esecuzione del programma. Questa caratteristica è particolarmente utile quando si desidera istanziare un oggetto senza interrompere il flusso principale dell'applicazione.

Infine, i messaggi asincroni offrono la possibilità di **comunicare con un thread già in esecuzione**. Questa forma di comunicazione non bloccante consente agli oggetti di scambiare informazioni o richieste con un'entità in esecuzione parallelamente, contribuendo a una gestione più efficiente delle attività nel sistema.

In conclusione, l'utilizzo dei messaggi asincroni nei diagrammi di sequenza fornisce un approccio dinamico e non bloccante per gestire la comunicazione tra oggetti, consentendo una maggiore flessibilità nella progettazione e nell'esecuzione dei sistemi software.

Complexity and sequence diagrams

Nel contesto dei diagrammi di sequenza, la gestione della complessità è cruciale per garantire chiarezza e comprensibilità. Alcuni principi chiave per affrontare la complessità includono:

- **KISS (Keep It Small and Simple)**:il principio "KISS" suggerisce di mantenere i diagrammi di sequenza il più piccoli e semplici possibile. L'obiettivo è evitare sovraccarichi di informazioni e rendere più agevole la comprensione del flusso di interazione tra gli oggetti.
- **Chiarezza nei Diagrammi:**I diagrammi sono strumenti progettati per rendere le cose chiare. Se la logica condizionale è semplice, è opportuno aggiungerla direttamente al diagramma. Tuttavia, se la logica condizionale diventa complessa, può essere più utile creare diagrammi separati per mantenere la chiarezza.
- **Logica Condizionale:**
    - Semplice: Quando la logica condizionale è semplice e non appesantisce eccessivamente il diagramma, può essere inclusa direttamente.
    - Complessa: Se la logica condizionale è complessa, considera la possibilità di creare un diagramma separato dedicato a quella parte del flusso. Questo aiuta a mantenere i diagrammi principali focalizzati e comprensibili.

Globalemente, l'obiettivo è mantenere la chiarezza e facilitare la comprensione del flusso di interazione. Se la complessità richiede una gestione più attenta, l'approccio di disegnare diagrammi separati può essere un modo efficace per affrontare dettagli specifici senza sovraccaricare il diagramma principale.

Where are the boundaries?

Nell'ambito dei sistemi software, i **confini** rivestono un ruolo cruciale nella definizione delle comunicazioni tra il sistema e il mondo esterno. Questi limiti possono assumere varie forme, come l'interfaccia utente o l'interazione con altri sistemi. Nella rappresentazione visiva delle interazioni, come i diagrammi di sequenza o di collaborazione, è spesso utile includere classi di confine.

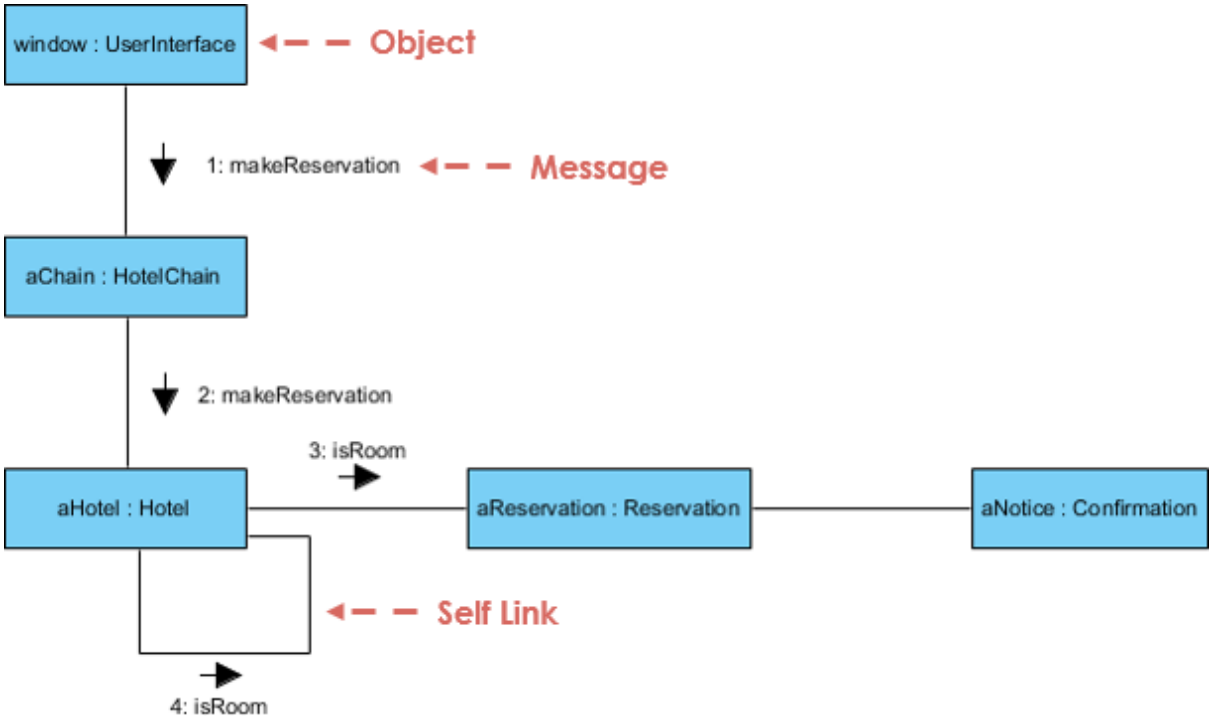
Le classi di confine catturano i requisiti dell'interfaccia, delineando come il sistema interagirà con l'ambiente esterno, senza dettagliare l'implementazione interna. Questo approccio consente di concentrarsi sugli aspetti comunicativi, catturando le esigenze dell'interfaccia senza entrare nei dettagli di come verrà effettivamente realizzata. In breve, i confini definiscono il perimetro delle interazioni, fornendo una chiara demarcazione tra il sistema e il mondo circostante.

Collaboration diagrams

I diagrammi di collaborazione sono strumenti visivi che mostrano come gli oggetti interagiscono tra loro, considerando anche le unità organizzative e i confini del sistema. Questi diagrammi forniscono una prospettiva chiara sulle relazioni e le interazioni tra gli oggetti coinvolti.

La sequenza delle messaggi in un diagramma di collaborazione è determinata da un sistema di numerazione. Questa numerazione segue un ordine che può essere semplice (1, 2, 3, 4, ...) o gerarchico (1, 1.1, 1.2, 1.3, 2, 2.1, 2.1.1, 2.2, 3). Tale sistema numerico indica la sequenza temporale degli scambi di messaggi, aiutando a comprendere l'ordine degli eventi nell'interazione degli oggetti.

Ad esempio, la numerazione può evidenziare chiaramente quale operazione chiama un'altra operazione, fornendo un'indicazione visiva della catena di chiamate di funzioni. In questo modo, i diagrammi di collaborazione offrono una rappresentazione visiva e sequenziale delle interazioni tra gli oggetti e le operazioni all'interno del sistema.



Comparing sequence & collaboration diagrams

I **diagrammi di sequenza** sono particolarmente efficaci nel visualizzare il flusso temporale delle interazioni. Essi forniscono una chiara sequenza di messaggi scambiati tra gli oggetti nel corso del tempo, facilitando la comprensione dell'ordine degli eventi.

Al contrario, la sequenza di messaggi può risultare più complessa da comprendere nei **diagrammi di collaborazione**. Questi ultimi, invece, sono ottimali per evidenziare il flusso di controllo attraverso le unità organizzative del sistema. La disposizione dei diagrammi di collaborazione può illustrare in modo statico le connessioni tra gli oggetti.

Tuttavia, esprimere in modo chiaro flussi di controllo complessi può risultare difficile in entrambi i tipi di diagrammi. In generale, la scelta tra diagrammi di sequenza e di collaborazione dipende dall'obiettivo specifico dell'analisi o della progettazione che si sta affrontando e dalla necessità di evidenziare il flusso temporale o il controllo organizzativo.

Content of collaboration diagrams

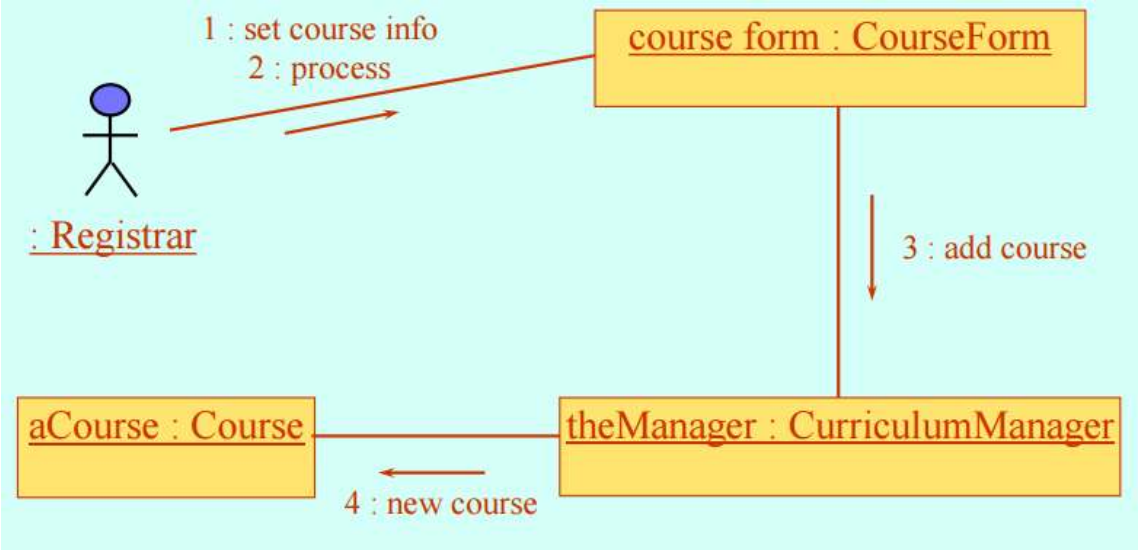
Gli **oggetti** in un sistema interagiscono scambiandosi **messaggi** tra loro. Questi messaggi possono essere di diversi tipi:

- **Sincroni**:Indicati da frecce complete, rappresentano eventi di chiamata in cui il mittente aspetta la risposta del destinatario prima di procedere.
- **Asincroni**: Rappresentati da frecce a metà, indicano segnali in cui il mittente non attende la risposta immediata del destinatario e può continuare le sue operazioni.

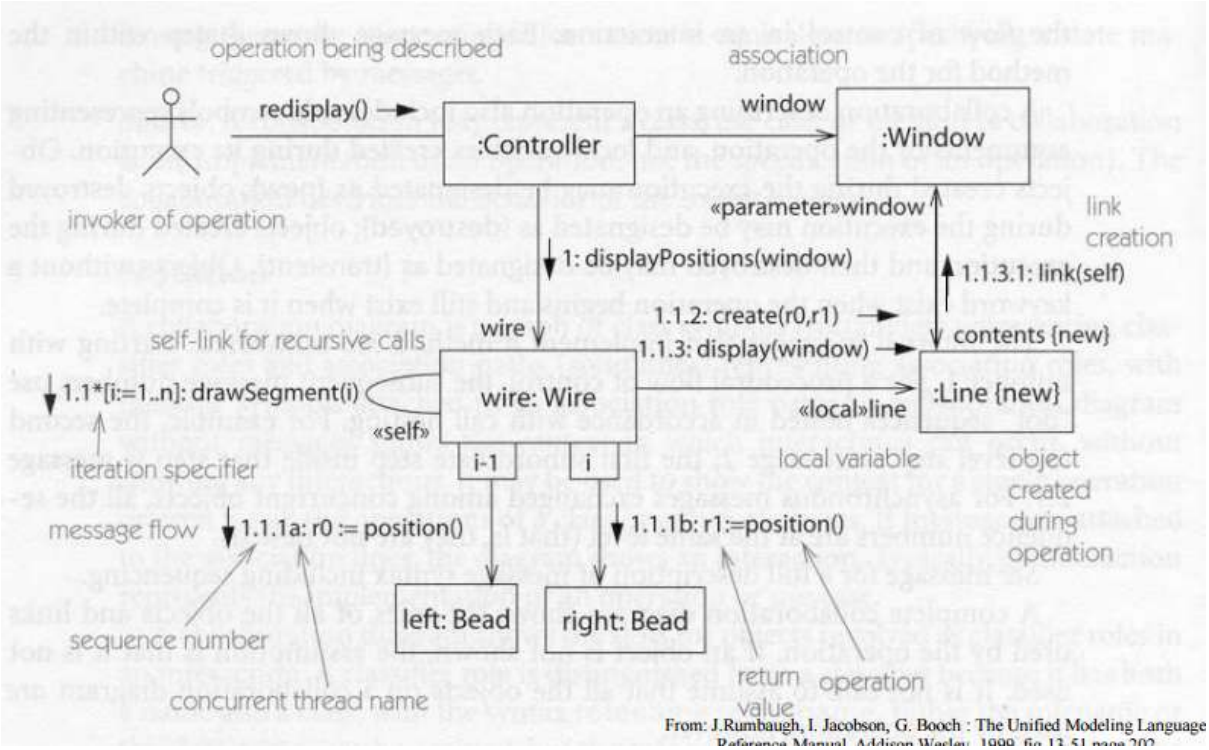
Inoltre, ci sono anche **messaggi di creazione** («create») **e distruzione** («destroy»), che denotano la creazione o la rimozione di un oggetto.I messaggi sono numerati e possono includere cicli, consentendo una rappresentazione più complessa delle interazioni.

Questa struttura è molto simile a quanto si riscontra nei diagrammi di sequenza, evidenziando la loro affinità nell'esprimere il flusso delle operazioni e delle comunicazioni tra gli oggetti.

Collaboration diagram example

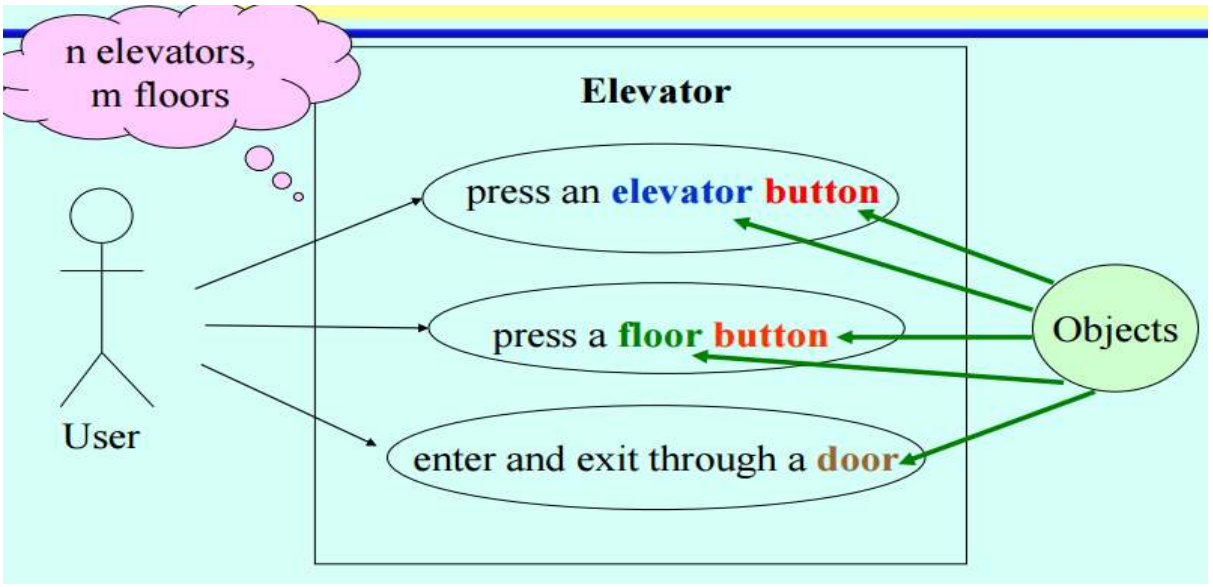


Collaboration diagrams can become VERY complex

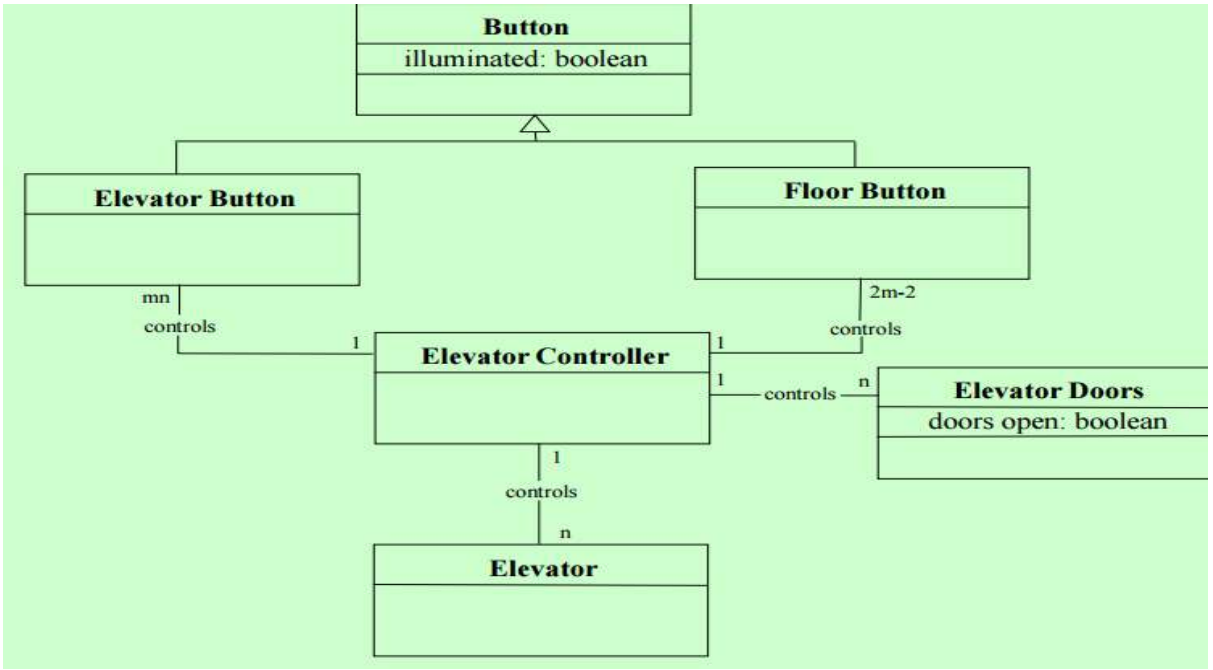


A Comprehensive Example:The Elevator

Elevator -- Use Case

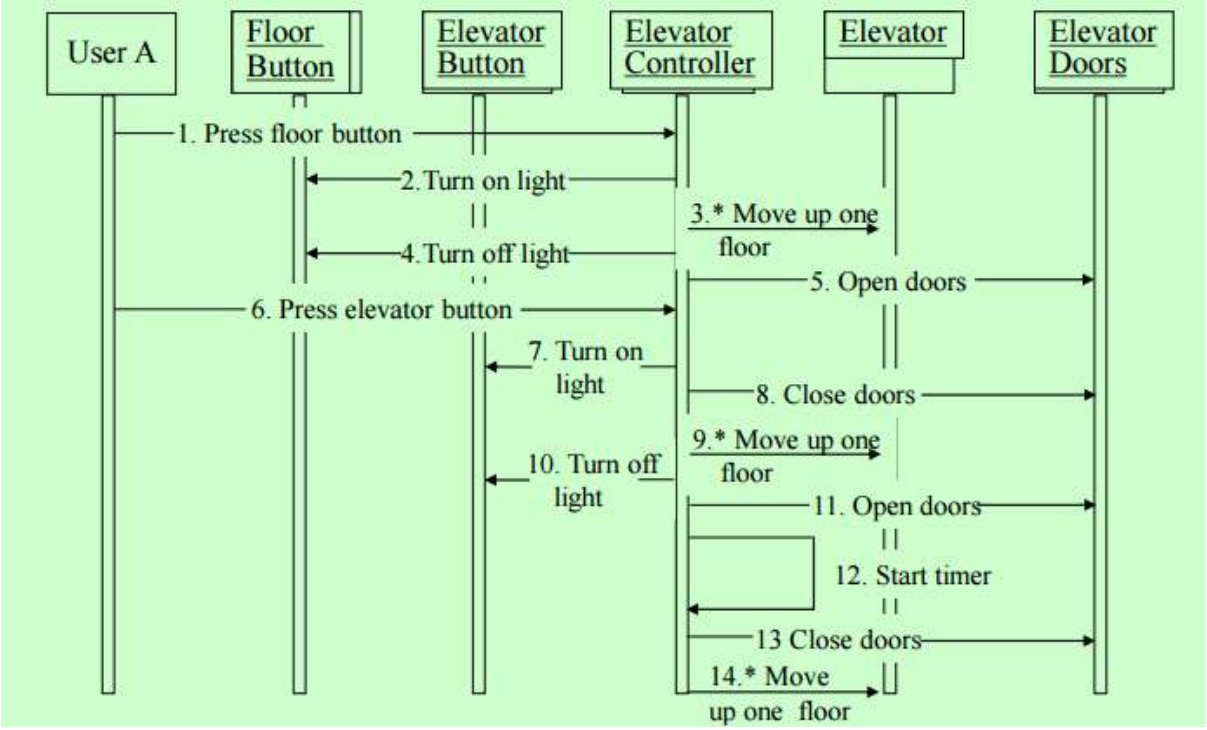


Elevator - First Class Diagram

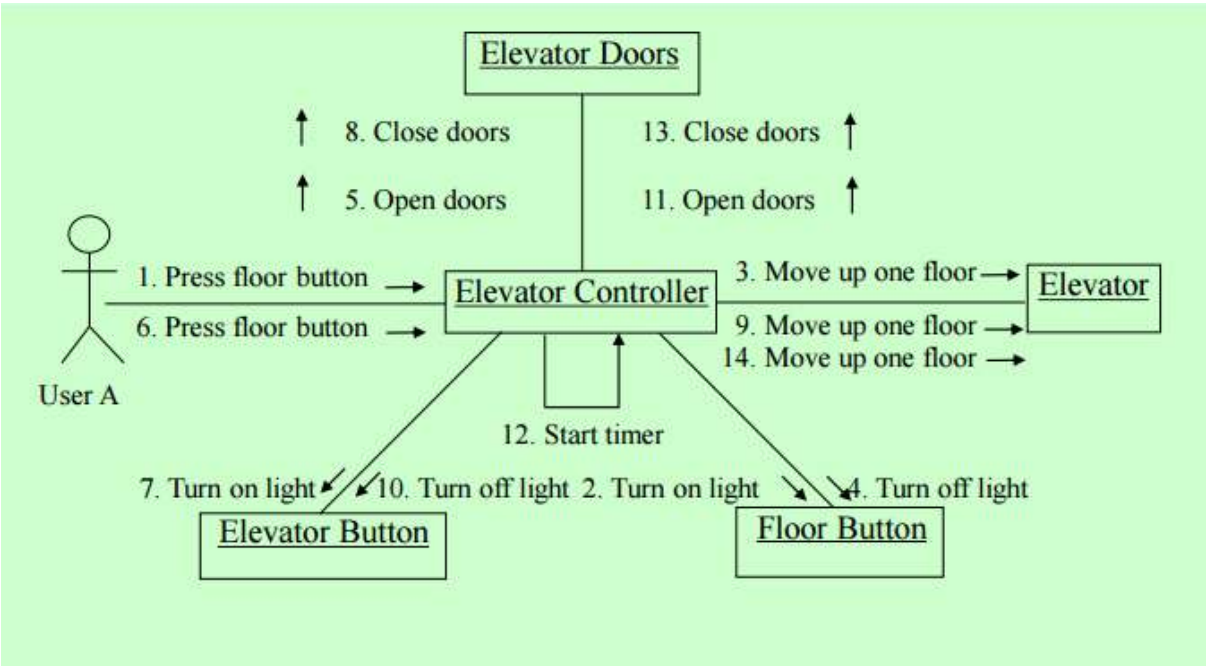


Elevator -State Diagram

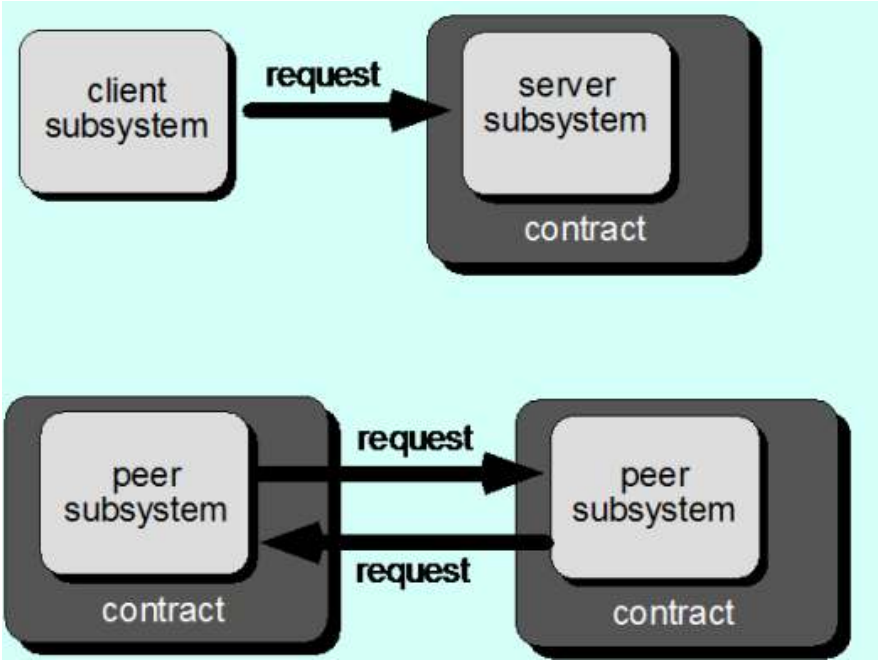
Elevator - Sequence Diagram



Elevator - Collaboration Diagram



System Design



Il System Design, o progettazione di sistema, è una fase chiave nell'ingegneria del software, che segue l'analisi dei requisiti. In questa fase, gli ingegneri del software traducono i requisiti identificati in una specifica progettazione di sistema. Questo coinvolge la creazione di una struttura di alto livello che definisce come il sistema sarà organizzato, compresi i moduli principali, le interazioni tra di essi e l'architettura generale. La progettazione di sistema fornisce una visione dettagliata di come il software soddisferà i requisiti funzionali e non funzionali, considerando aspetti come la modularità, l'efficienza, la manutenibilità e la scalabilità.

Systems and Sub-Systems (Sistemi e Sotto-Sistemi):

Il concetto di sistemi e sotto-sistemi si riferisce alla suddivisione di un sistema complesso in parti più gestibili e comprensibili. Un sistema può essere suddiviso in sotto-sistemi autonomi, ciascuno dei quali svolge funzioni specifiche. Questo approccio modulare semplifica lo sviluppo, la manutenzione e l'evoluzione del

sistema nel suo complesso. I sotto-sistemi possono a loro volta essere suddivisi in ulteriori componenti o moduli, creando una gerarchia di elementi che cooperano per raggiungere gli obiettivi del sistema complessivo. La definizione chiara delle relazioni e delle interfacce tra i sistemi e i sotto-sistemi è fondamentale per garantire la coerenza e l'efficacia del sistema globale.

Come suddividere un sistema in sottosistemi più piccoli?

- **Principio di Divide et Impera (Divide & Conquer):**Un approccio chiave è il principio "Divide et Impera", noto anche come "Divide & Conquer". Questo suggerisce di frammentare un sistema più grande in parti gestibili. Suddividendo il problema complesso in sottoproblemi più semplici, è possibile affrontare e risolvere ciascun componente separatamente, semplificando così l'intero processo.
- **Metodi Strutturati: Decomposizione Funzionale:**Nell'ambito dei metodi strutturati, si utilizza la decomposizione funzionale. Questa tecnica prevede la suddivisione delle funzioni del sistema in sottofunzioni più piccole. In questo modo, ogni sottofunzione può essere trattata come un modulo separato, semplificando la progettazione e l'implementazione.
- **Orientamento agli Oggetti (OO): Raggruppamento di Classi in Unità di Livello Superiore (Packages/Components):**In programmazione orientata agli oggetti, le classi correlate vengono raggruppate in unità di livello superiore come pacchetti o componenti. Questo approccio facilita la gestione delle relazioni e delle interfacce tra classi connesse, semplificando la comprensione e la manutenzione del sistema.

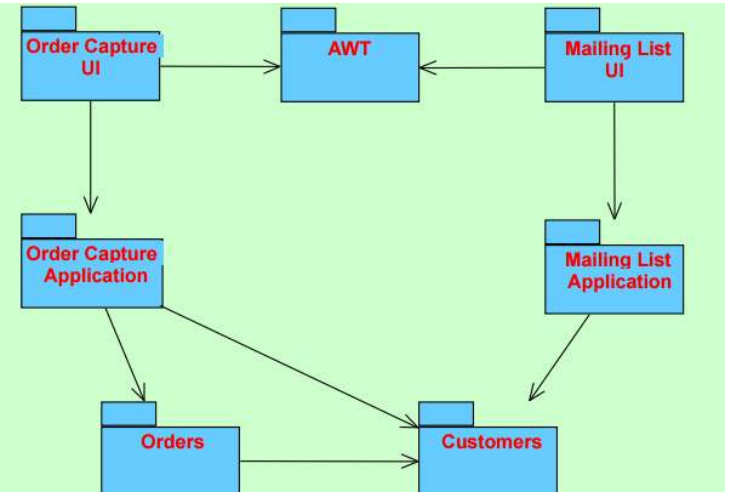
Packages (Pacchetti):

Visualizzazione dei Pacchetti:

I pacchetti sono strutture organizzative utilizzate per raggruppare elementi correlati all'interno di un sistema. La visualizzazione dei pacchetti fornisce una rappresentazione chiara e ordinata della suddivisione logica del sistema in unità funzionali.

Mostrare le Dipendenze tra Pacchetti:

Nei diagrammi di progettazione, è comune mostrare le dipendenze tra pacchetti. Una dipendenza tra due pacchetti indica che le modifiche alla definizione di uno di essi possono comportare modifiche nell'altro. Questo è un concetto cruciale per gestire le relazioni e garantire che eventuali modifiche a una parte del sistema non influiscano negativamente su altre parti.



Obiettivo e Arte della Progettazione su Larga Scala:

L'obiettivo principale, e al contempo un'arte, della progettazione su larga scala è quello di minimizzare le dipendenze tra pacchetti. Minimizzare le dipendenze è fondamentale per limitare gli effetti collaterali delle modifiche. Riducendo le dipendenze, si aumenta la coesione all'interno dei pacchetti, rendendo il sistema più robusto, manutenibile ed estensibile.

La gestione efficace delle dipendenze tra pacchetti è un aspetto chiave della progettazione del software, contribuendo a garantire la solidità e la flessibilità del sistema nell'affrontare cambiamenti futuri.

Nested Packages (Pacchetti Nidificati):

I pacchetti nidificati sono una struttura organizzativa in cui un pacchetto può contenere altri pacchetti. Ci sono due interpretazioni principali dei pacchetti nidificati:

- **Trasparente:** In questo caso, tutti i contenuti dei pacchetti contenuti sono visibili all'esterno. Tutte le classi e i componenti dei pacchetti nidificati sono accessibili direttamente dall'esterno del pacchetto contenitore.
- **Opaco**:Al contrario, nell'interpretazione opaca, solo le classi del pacchetto contenitore sono visibili all'esterno. Le classi e i componenti all'interno dei pacchetti nidificati non sono direttamente accessibili dall'esterno.

Esempio:Supponiamo di avere una struttura di pacchetti come "Orders" e "Customers", e all'interno di "Orders" ci sono "Order Capture Application" e "Mailing List Application". Se interpretiamo i pacchetti in modo trasparente, possiamo vedere direttamente tutte le classi all'interno di "Order Capture Application" e "Mailing List Application". Se interpretiamo in modo opaco, vedremo solo le classi di "Orders" senza dettagli sulle classi interne dei sottopacchetti.

Come può essere limitata la complessità di un'interfaccia di pacchetto?

Per limitare la complessità di un'interfaccia di pacchetto, si possono adottare alcune strategie pratiche. Prima di tutto, è utile assegnare **visibilità di pacchetto** esclusivamente alle classi all'interno del pacchetto, mantenendo nascosti i dettagli interni. Successivamente, **definire una classe pubblica** come interfaccia esterna, esponendo solo ciò che è necessario all'esterno del pacchetto. Infine, per gestire le operazioni pubbliche, è **consigliabile delegarle alle classi appropriate interne**, garantendo una chiara separazione tra l'interfaccia pubblica e l'implementazione interna. Queste pratiche mirano a semplificare l'uso del pacchetto, rendendo più agevole la comprensione e la gestione da parte degli sviluppatori esterni.

Regole Pratiche:

- **Evitare cicli nella struttura delle dipendenze:**Cerca di evitare cicli nella struttura delle dipendenze tra classi o componenti del sistema. I cicli possono portare a complessità e difficoltà nel gestire le relazioni.
- **Rifattorizzare il sistema con troppe dipendenze:**Se noti un eccesso di dipendenze nel sistema, considera la possibilità di rifattorizzare. Troppi legami possono rendere il sistema complicato e meno manutenibile nel tempo.
- **Utilizzare dipendenze quando il diagramma delle classi non è leggibile su un foglio di carta di dimensioni standard**: Introduce dipendenze solo quando il diagramma delle classi diventa così complesso da non essere più leggibile su un foglio di carta di dimensioni standard. Questo può essere un indicatore di complessità che richiede una rappresentazione più dettagliata o una suddivisione in moduli più gestibili.

Pacchetti vs. Diagrammi:

Pacchetti rappresentano divisioni fisiche dello sviluppo:

- **Obiettivo:**Semplificare la definizione di pacchetti di lavoro e lo sviluppo di sistemi software. I pacchetti sono utilizzati per organizzare fisicamente il lavoro e le risorse nello sviluppo del software.

Diagrammi rappresentano l'assemblaggio logico delle informazioni:

- **Obiettivo:** Agevolare la comprensione del dominio di destinazione. I diagrammi sono strumenti logici che aiutano a visualizzare in modo chiaro e comprensibile l'assemblaggio logico delle informazioni nel contesto del dominio di destinazione.

In breve, i pacchetti si concentrano sulla struttura fisica dello sviluppo e sulla divisione del lavoro, mentre i diagrammi mirano a rappresentare in modo chiaro e logico le informazioni relative al dominio di destinazione. Entrambi sono strumenti cruciali, ognuno con un obiettivo specifico nel processo di sviluppo del software.

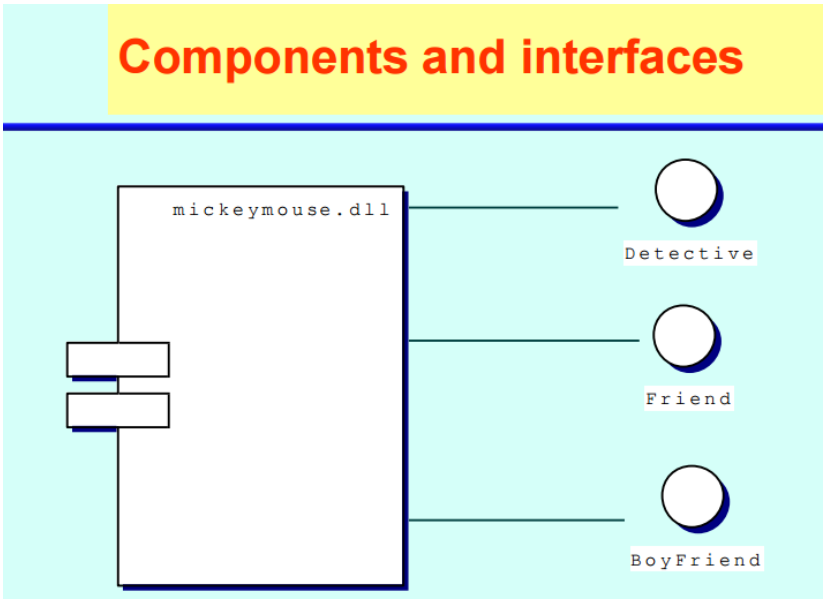
Uno Sguardo ai Design Patterns:

Nei sistemi orientati agli oggetti, è possibile individuare modelli ricorrenti di classi e oggetti che comunicano. Questi modelli, noti come "design patterns", risolvono problemi di progettazione specifici, conferendo maggiore flessibilità, eleganza e, alla fine, riusabilità al design orientato agli oggetti. I design pattern consentono ai progettisti di riutilizzare soluzioni di successo, basando i nuovi progetti sull'esperienza passata. Un progettista familiarizzato con questi pattern può applicarli immediatamente nella progettazione.

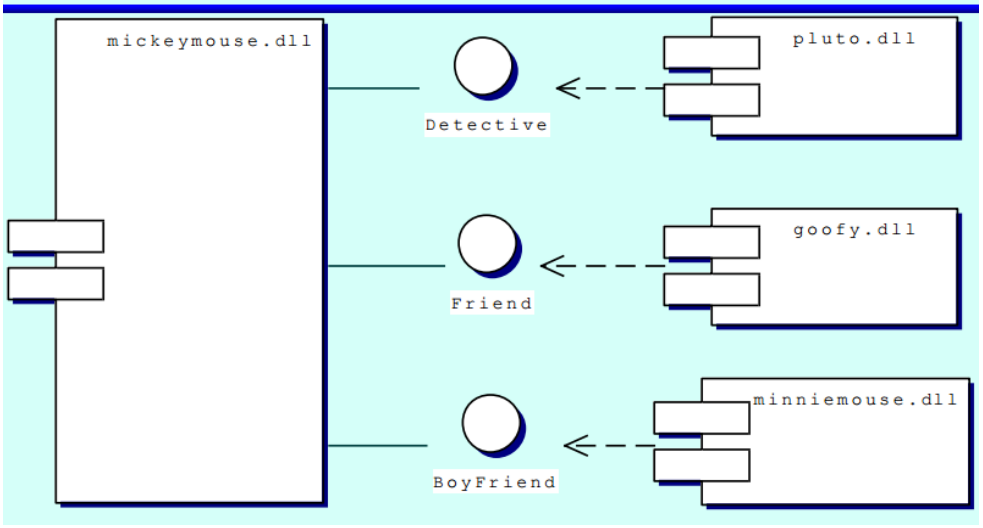
Il nome del design pattern è un'astrazione che trasmette significato sull'applicabilità e l'intento. La descrizione del problema indica l'ambiente e le condizioni necessarie per rendere il design pattern applicabile. Le caratteristiche del pattern indicano gli attributi del design che possono essere adattati per consentire al pattern di adattarsi a una varietà di problemi. Le conseguenze associate all'uso di un design pattern forniscono un'indicazione delle ramificazioni delle decisioni di progettazione.

Questi concetti sono stati introdotti da Gamma e i suoi colleghi, e rappresentano un modo potente per affrontare problemi di progettazione comuni in modo efficace ed efficiente.

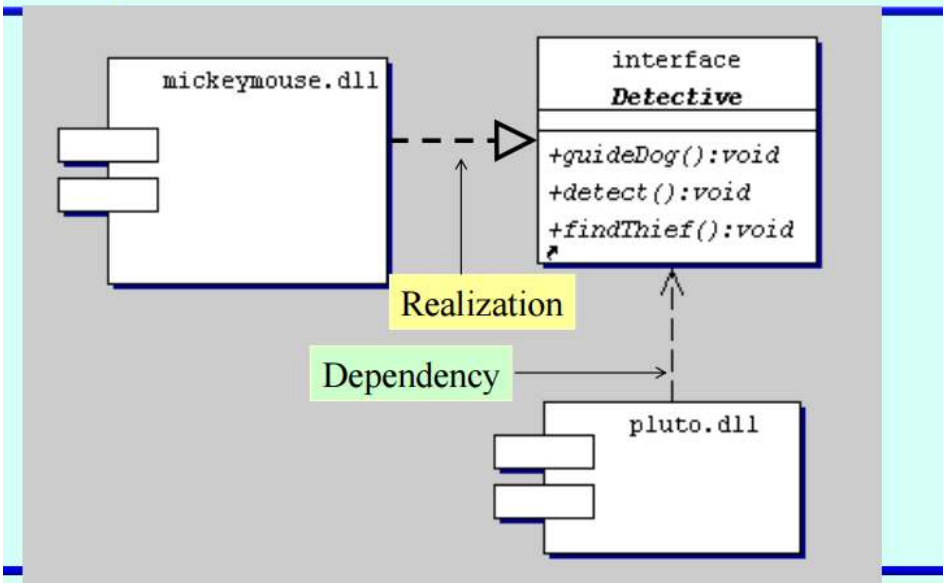
Components in UML

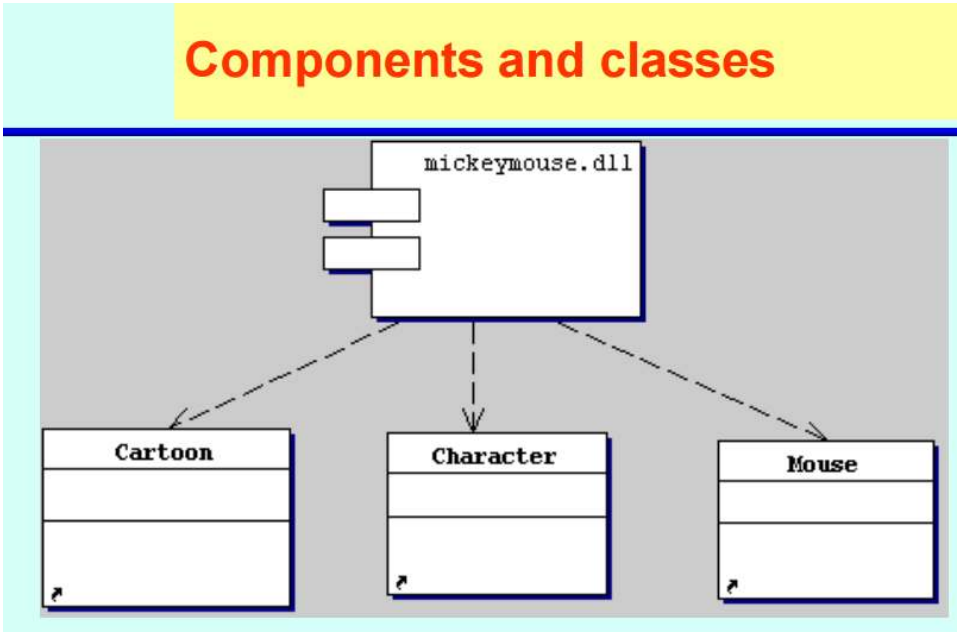


Components and Collaborations



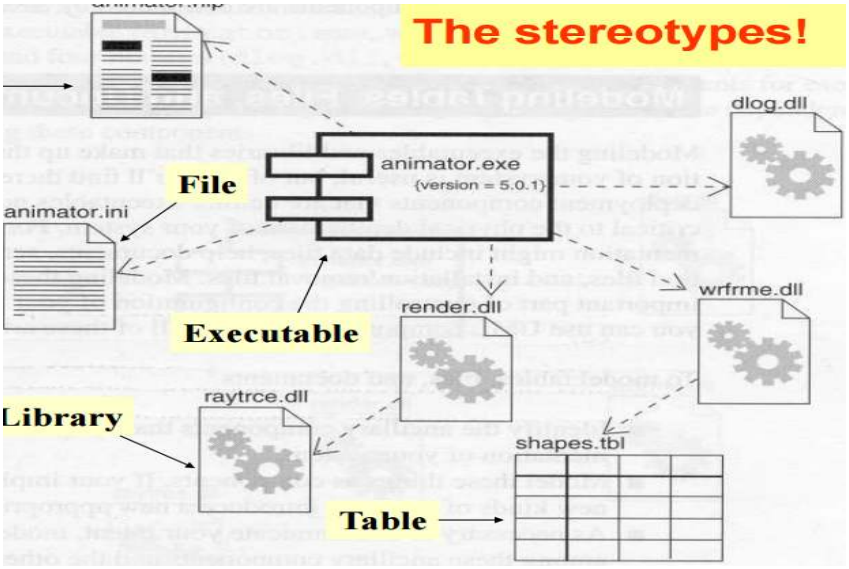
Detailed View of a Collaboration





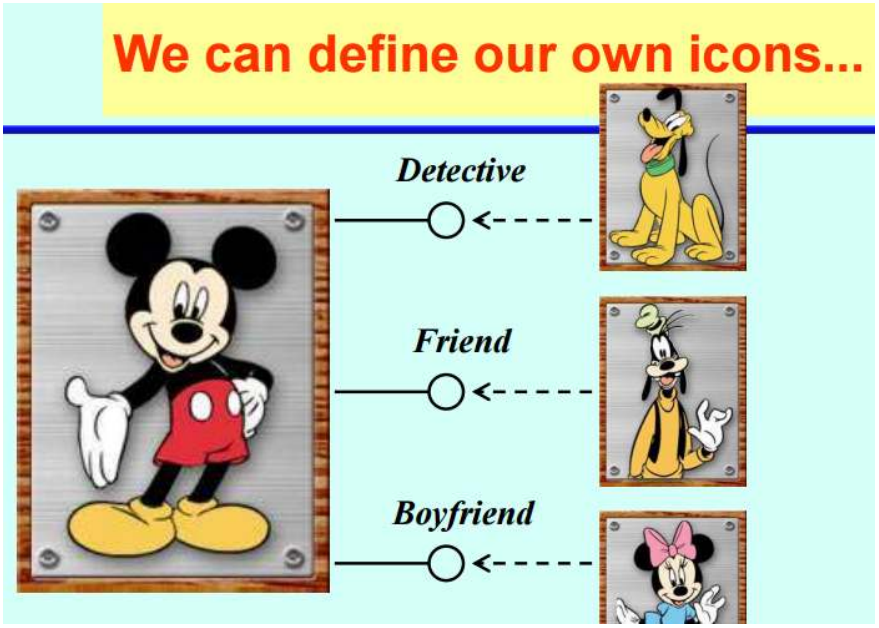
Stereotipi per i Componenti:

UML definisce cinque tipi di stereotipi che possono essere applicati ai componenti:



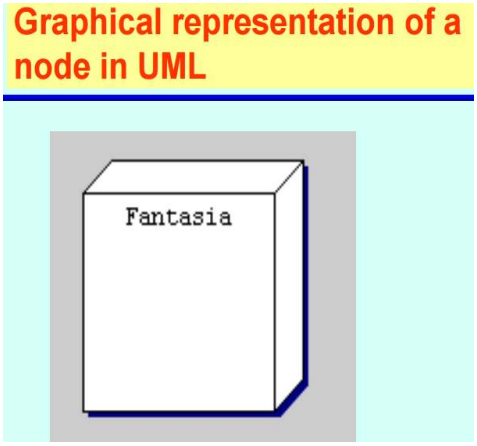
- **Eseguibile**
- **Libreria**
- **Tabella**
- **File**
- **Documento**

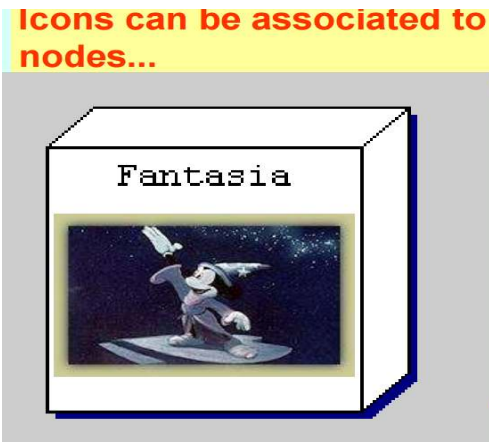
Questi stereotipi forniscono un modo per categorizzare e distinguere i diversi tipi di componenti all'interno di un sistema, consentendo una migliore comprensione delle loro funzionalità e scopi. Ad esempio, un componente contrassegnato come "eseguibile" potrebbe indicare che è responsabile dell'esecuzione di un'applicazione, mentre un componente contrassegnato come "libreria" potrebbe rappresentare una raccolta di funzioni o risorse riutilizzabili. L'applicazione di stereotipi contribuisce a rendere più chiara la struttura e il significato dei componenti all'interno di un modello UML.

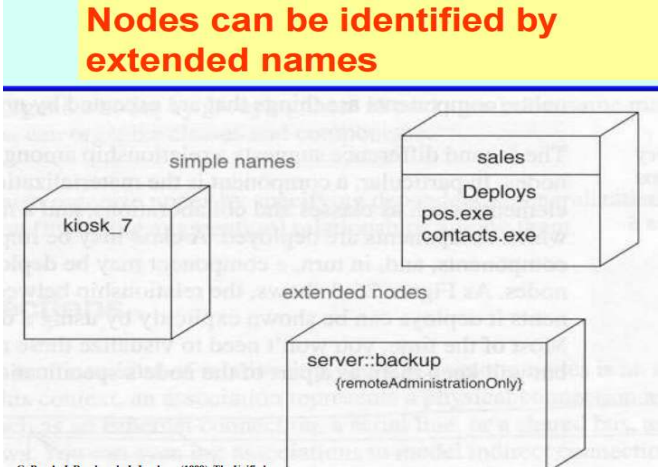


Nodi:

Un nodo è un elemento fisico che esiste durante l'esecuzione e rappresenta una risorsa computazionale, di solito dotata almeno di una certa quantità di memoria e, spesso, di capacità di elaborazione. I nodi nel contesto dell'UML (Unified Modeling Language) sono elementi chiave per modellare l'architettura di un sistema distribuito o di rete, identificando le risorse hardware in cui i componenti del sistema possono essere eseguiti. Ogni nodo rappresenta un punto fisico in cui il software può essere eseguito o i dati possono essere archiviati durante l'esecuzione del sistema.







Components and Nodes

Componenti:

- Sono simili ai nodi (hanno nomi; partecipano a relazioni di dipendenza, generalizzazione e associazione; possono essere nidificati; possono avere istanze; possono partecipare a interazioni; ...).

- Tuttavia, i componenti sono entità che partecipano all'esecuzione di un sistema.

Nodi:
- Sono simili ai componenti (hanno nomi; partecipano a relazioni di dipendenza, generalizzazione e associazione; possono essere nidificati; possono avere istanze; possono partecipare a interazioni; ...).
- Tuttavia, i nodi sono entità che eseguono i componenti.

Differenze:
- I componenti rappresentano l'incapsulamento fisico di elementi altrimenti logici.
- I nodi rappresentano la distribuzione fisica dei componenti.

In breve, mentre i componenti sono le entità che partecipano all'esecuzione di un sistema, i nodi sono le entità che effettivamente eseguono tali componenti, e la loro relazione è fondamentale per comprendere l'architettura e l'implementazione di un sistema software.

Real Time System Analysis and Design in UML

Affrontare l'utilizzo di UML in contesti real-time richiede un'approccio considerando due aspetti chiave.

In primo luogo, è importante esplorare le caratteristiche intrinseche di UML che possono essere impiegate per supportare applicazioni real-time. UML, come linguaggio di modellazione unificato, offre un repertorio di funzionalità che possono essere adattate per modellare e gestire le dinamiche proprie delle applicazioni real-time.

D'altra parte, ci sono estensioni specificamente proposte per UML che sono mirate a soddisfare le particolari esigenze delle applicazioni real-time. Queste estensioni, integrate attraverso le caratteristiche auto-estendenti di UML, consentono di personalizzare il linguaggio per affrontare in modo mirato le sfide e i requisiti unici che emergono in contesti real-time.

Quindi, in sostanza, UML diventa uno strumento versatile che può essere utilizzato sia con le sue funzionalità di base che con estensioni specifiche per modellare e supportare applicazioni real-time, offrendo una flessibilità cruciale per adattare il linguaggio alle esigenze dinamiche di tali scenari complessi.

Costrutti Real-Time in UML:

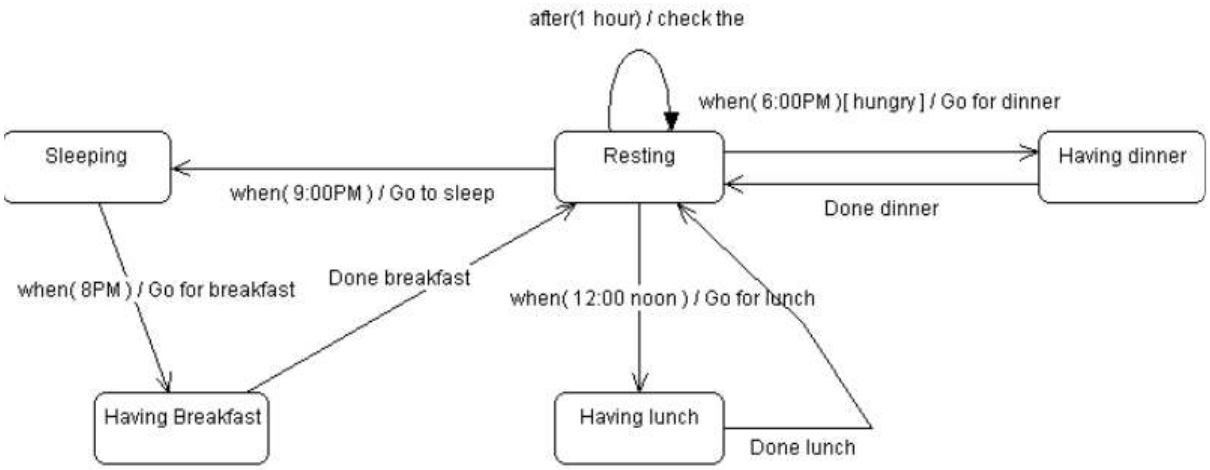
UML offre specifiche caratteristiche volte a supportare applicazioni real-time, offrendo un repertorio di elementi progettati per affrontare le sfide uniche di tali contesti dinamici.

- **Change Event (Evento di Modifica):**Questo costrutto consente di modellare gli eventi che si verificano a seguito di modifiche nello stato di un sistema. È particolarmente rilevante in scenari real-time in cui le transizioni di stato possono essere innescate da cambiamenti specifici e influenzano direttamente il flusso temporale dell'applicazione.
- **Time Event (Evento Temporale):**UML comprende un evento temporale che consente di catturare e rappresentare gli eventi che si verificano in risposta al trascorrere del tempo. Questo è fondamentale nelle applicazioni real-time in cui il controllo degli eventi è strettamente correlato al tempo.
- **Timing Constraints (Vincoli Temporali):** Questo aspetto di UML permette di definire vincoli temporali che specificano le relazioni temporali tra gli eventi o le attività. È un elemento cruciale quando si progetta per applicazioni real-time, dove la precisione temporale è essenziale per il corretto funzionamento del sistema.

Eventi Temporali e di Modifica:

- **Eventi Temporali e di Modifica specificano quando un'azione deve essere eseguita in un istante specifico:**Gli eventi temporali e di modifica in UML sono progettati per indicare quando un'azione deve essere eseguita in un determinato istante di tempo, aggiungendo una dimensione temporale critica alla modellazione del sistema.
- **"when" è un evento di modifica che specifica la data/ora in cui si verificherà l'evento**:Ad esempio, con "when (12:00AM) / goForLunch()", si stabilisce che l'evento "andare a pranzo" si verificherà alle 12:00AM.
- **"after" è un evento di modifica che specifica dopo quanto tempo verrà eseguito un dato evento:**Un esempio con "after(5 minutes) / put the egg in the pot" indica che l'azione "mettere l'uovo nella pentola" si svolgerà dopo 5 minuti.

In breve, questi eventi consentono di modellare con precisione l'aspetto temporale delle attività nel sistema, specificando sia il momento esatto in cui devono avvenire gli eventi che l'intervallo di tempo dopo il quale determinate azioni saranno eseguite. Ciò contribuisce a rendere il modello UML più aderente alla realtà delle dinamiche temporali del sistema.



Vincoli Temporali:

Vincoli Temporali possono essere aggiunti ai messaggi:

- In UML, è possibile applicare vincoli temporali ai messaggi per definire le restrizioni sul tempo di esecuzione delle operazioni. Tra le variabili integrate ci sono "startTime," "stopTime," e "executionTime," che consentono di specificare dettagli temporali cruciali.

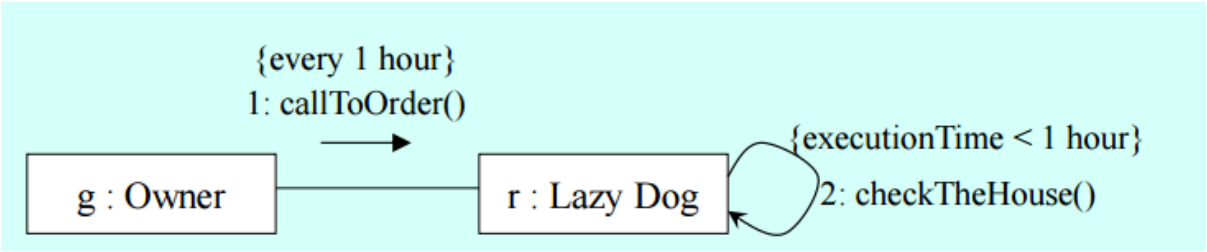
Variabili integrate:

- Le variabili "startTime," "stopTime," e "executionTime" offrono dettagli specifici sulla temporizzazione delle operazioni, permettendo di modellare con precisione quando un messaggio inizia, termina o quanto tempo richiede per essere eseguito.

Costrutto "every":

- L'utilizzo del costrutto "every" identifica un messaggio che viene ritrasmesso a intervalli temporali specifici. Questo è particolarmente utile quando si desidera modellare la periodicità di un'azione nel sistema.

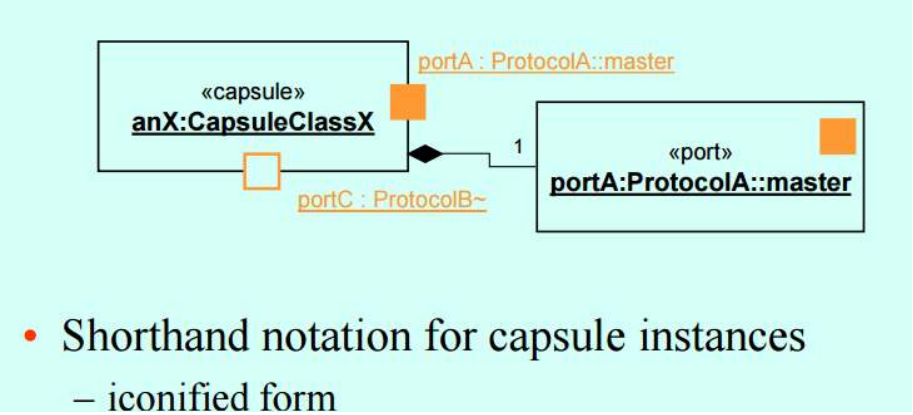
In sintesi, i vincoli temporali in UML consentono di dettagliare gli aspetti temporali delle interazioni tra gli oggetti, fornendo un livello aggiuntivo di precisione nella modellazione delle dinamiche temporali del sistema. Ciò è cruciale quando si desidera catturare e rappresentare in modo accurato il comportamento del sistema rispetto al tempo.



In altre parole, sebbene UML fornisca strumenti per definire vincoli temporali e dettagli temporali nel modello, è importante notare che molti strumenti o linguaggi di programmazione potrebbero non applicare automaticamente questi vincoli durante l'implementazione del sistema. Pertanto, la responsabilità di rispettare tali vincoli può

ricadere sul team di sviluppo, richiedendo una gestione attenta e una sincronizzazione manuale tra il modello UML e l'implementazione effettiva.

ROOM: Real Time Object Oriented Modeling



The Object Constraint Language (OCL)

OCL, acronimo di Object Constraint Language, è come la chiave segreta che svela il significato implicito all'interno del tuo modello UML. È un linguaggio espressivo progettato per definire vincoli e condizioni, arricchendo il modello con dettagli formali e specifiche rigorose.

Questo linguaggio è come un pittore che dipinge con parole, senza sporcarsi le mani con effetti collaterali. Le espressioni in OCL sono come pennellate che definiscono le regole del gioco senza interferire con il flusso naturale del sistema.

Nonostante la sua potenza, OCL è una guida dietro le quinte, non un attore principale. Non può essere eseguito autonomamente, ma la sua presenza nel modello garantisce una comprensione più chiara e una formalizzazione delle condizioni del sistema.

OCL è un linguaggio formale, un accordo tra il progettista e il modello. Le sue espressioni hanno una semantica ben definita, assicurando che ciascun dettaglio sia interpretato con chiarezza nel contesto del vasto panorama di UML.

In breve, OCL è il traduttore silenzioso che dà voce ai dettagli del tuo modello UML, trasformandolo da un'illustrazione in un'opera completa di regole e specifiche.

L'obiettivo di OCL è definire in modo chiaro e inequivocabile le dichiarazioni di guardia in UML. In altre parole, OCL si propone di formulare in maniera pulita e senza ambiguità le istruzioni di controllo che regolano il comportamento del sistema rappresentato nel linguaggio di modellazione UML.

Principali Caratteristiche di OCL :

OCL presenta alcune caratteristiche distintive che contribuiscono alla sua potenza e flessibilità nell'ambito di UML. Una di queste è la sintassi degli attributi, che segue lo stesso stile di Java, con **l'eccezione** di utilizzare "self" al posto di "this". Ad esempio, un'espressione come `self.weight < 100` è comune in OCL.

Un altro elemento chiave è la **tipizzazione forte delle espressioni, con la verifica della conformità del tipo. Questo significa che** OCL assicura che le espressioni siano compatibili dal punto di vista del tipo. Ad esempio, l'espressione `self.weight < "abc"` evidenzia questa caratteristica, dove il confronto tra un attributo di tipo numerico e una stringa sarebbe segnalato come un'incompatibilità di tipo.

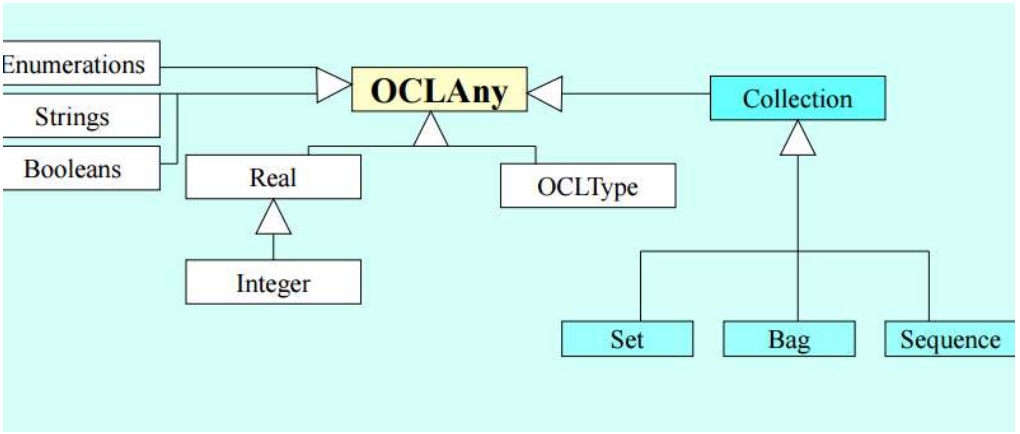
In UML, è possibile definire condizioni che devono essere soddisfatte **prima (pre) e dopo (post)** l'esecuzione di un'operazione, fornendo una guida chiara sullo stato del sistema prima e dopo l'esecuzione di specifiche azioni.

Esempi di assegnazione di condizioni pre e post a operazioni:

- Per l'operazione `Dog::bark()`:
- Condizione pre: "il quartiere è felice".
- Condizione post: "il quartiere è arrabbiato".
- Per l'operazione `Dog::eat(f: Real)`:
- Condizione post: "peso := peso@pre + f".

Le **associazioni** con cardinalità 1 sono trattate come attributi in OCL. Ad esempio, nell'espressione `Fido.owner = Joe`, l'associazione tra l'animale domestico Fido e il proprietario Joe viene trattata come un attributo.

D'altra parte, le associazioni con cardinalità diversa da 1 sono gestite come insiemi. Ad esempio, `Joe->pet` rappresenta un insieme di tutti gli animali domestici di Joe e dispone di operazioni ben definite come il "." e "->". Questi operatori possono essere utilizzati in modo polimorfico, adattandosi alle diverse situazioni all'interno del modello.



In OCL, le collezioni supportano vari metodi che consentono di manipolarle in modi diversi. Alcuni di questi metodi includono:

- `size`: restituisce la dimensione della collezione.
- `includes`: verifica se un elemento è presente nella collezione.
- `count`: conta il numero di occorrenze di un elemento nella collezione.
- `includesAll`: verifica se tutti gli elementi di un'altra collezione sono presenti.
- `isEmpty`: verifica se la collezione è vuota.
- `notEmpty`: verifica se la collezione non è vuota.
- `sum`: restituisce la somma degli elementi numerici nella collezione.
- `exists`: verifica se almeno un elemento soddisfa una condizione.
- `forAll`: verifica se tutti gli elementi soddisfano una condizione.
- `iterate`: esegue una serie di operazioni su ogni elemento della collezione.
- `select`: restituisce una nuova collezione contenente solo gli elementi che soddisfano una condizione.
- `reject`: restituisce una nuova collezione contenente solo gli elementi che non soddisfano una condizione.
- `collect`: restituisce una nuova collezione contenente i risultati di un'espressione applicata a ciascun elemento.

Introduction to Refactoring

Il concetto di refactoring nei diagrammi UML è una pratica che mira a migliorare la struttura e la chiarezza del modello senza cambiarne il comportamento. Si basa sul principio di apportare modifiche incrementali per rendere il modello più comprensibile, manutenibile e aderente alle best practice.

Un esempio semplice di refactoring potrebbe riguardare la semplificazione di una relazione complessa tra classi o la razionalizzazione di un diagramma per ridurre l'ingombro visivo. Ad esempio, potrebbe coinvolgere la combinazione di classi simili, la suddivisione di classi troppo complesse o la riorganizzazione delle relazioni per renderle più intuitive.

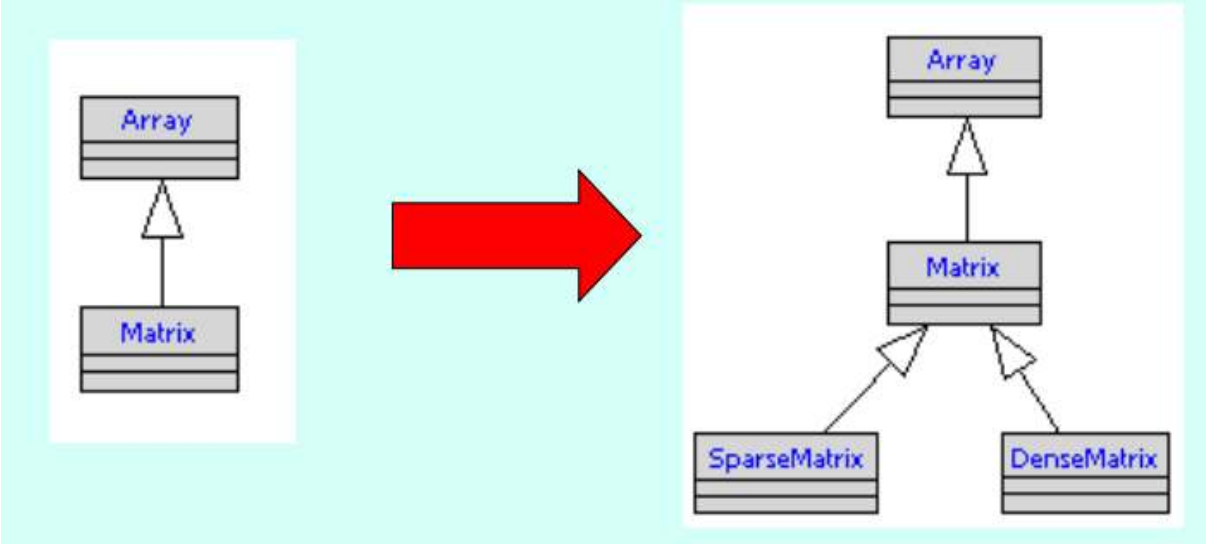
Le tecniche utilizzate per il **refactoring** nei diagrammi UML includono:

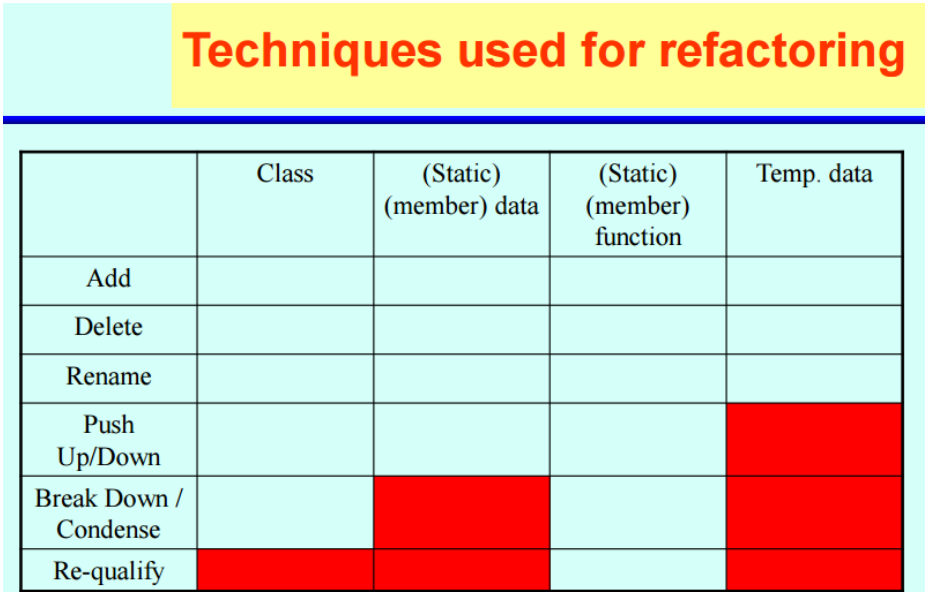
- **Raggruppamento di Classi:**Unire classi simili per semplificare la struttura del modello.
- **Suddivisione di Classi:**Se una classe è troppo complessa, suddividerla in classi più piccole e specializzate.
- **Semplificazione delle Relazioni**: Razionalizzare le relazioni tra classi per renderle più chiare e comprensibili.
- **Riorganizzazione dei Diagrammi:R**idisegnare i diagrammi per migliorare la disposizione e la leggibilità complessiva.

L'obiettivo principale del refactoring è migliorare la qualità del modello UML, facilitando la comprensione del sistema rappresentato e agevolando eventuali fasi di manutenzione o aggiornamento.

L'idea chiave è che i sistemi software sono entità in evoluzione che richiedono un costante aggiornamento e manutenzione. È essenziale definire approcci per rendere l'aggiornamento e la manutenzione "semplici", evitando complessità e difficoltà eccessive. La pratica chiave in questo contesto è lo sviluppo incrementale con cicli di vita brevi e refactoring costante. Questo approccio consente di affrontare i cambiamenti in modo graduale, mantenendo il sistema flessibile e adattabile nel tempo.

Refactoring dei diagrammi UML implica la revisione del sistema, la ristrutturazione del suo design e dei suoi componenti, cercando la semplicità e una maggiore aderenza alla struttura del dominio. L'obiettivo è mantenere invariato il comportamento del sistema, ma migliorarne la struttura interna per renderlo più comprensibile e adattabile.





I pattern di progettazione possono essere strumentali per il refactoring in quanto:

1. Suggeriscono la struttura interna del codice.
2. Guidano lo sviluppo di nuove strutture di oggetti e classi.
3. Aiutano a separare le strategie di implementazione dall'obiettivo di implementazione.

La guida per il refactoring con i Design Patterns offre orientamenti chiari e specifici per migliorare la struttura e la flessibilità del codice. Ecco alcune direttive principali:

- **Creazione coerente tra le classi (Abstract Factory):**Promuove la creazione di famiglie di oggetti correlati senza specificare le loro classi concrete, garantendo una coerenza tra le diverse istanze.
- **Creazione in base alle esigenze dell'ambiente (Factory Method):**Definisce un'interfaccia per la creazione di un oggetto, lasciando alle sottoclassi la decisione su quale classe istanziare.
- **Creazione basata su un'entità prototipica (Prototype):** Consente di creare nuovi oggetti duplicando un prototipo esistente, utile quando la creazione diretta è complessa o costosa.
- **Numero variabile di funzionalità (Decorator):**Aggiunge responsabilità a un oggetto dinamicamente, consentendo una gestione flessibile di funzionalità aggiuntive.
- **Struttura ad albero (Composite):**Compongono oggetti in strutture ad albero per rappresentare gerarchie parte-tutto, semplificando la gestione di singoli oggetti e composizioni complesse.
- **Double dispatching (Visitor):**Consente di definire una nuova operazione senza modificare le classi degli oggetti su cui opera, promuovendo l'estensibilità.
- **Implementazione flessibile di un algoritmo (Strategy):** Definisce una famiglia di algoritmi, incapsula ciascuno di essi e li rende intercambiabili, consentendo di variare l'algoritmo indipendentemente dai clienti che lo utilizzano.
- **Implementazione flessibile di una classe (Bridge)**: Separa un'astrazione dalla sua implementazione, permettendo loro di variare indipendentemente.

Questi pattern offrono un approccio strutturato e testato al refactoring, contribuendo a migliorare la qualità e la manutenibilità del codice.

Il refactoring dovrebbe essere avviato in risposta a inefficienze e carenze nel programma, piuttosto che per miglioramenti astratti. Un altro principio guida è scrivere il codice esattamente una volta, promuovendo la coerenza e riducendo la duplicazione. Infine, il refactoring dovrebbe includere l'eliminazione di entità di grandi dimensioni, come metodi, classi o pacchetti, per migliorare la chiarezza e la manutenibilità del sistema.

Quando ci si trova di fronte a un sistema che funziona, ma il tempo e le risorse finanziarie sono limitate, si affronta la sfida di lavorare con le astrazioni di progettazione di qualcun altro. In questo contesto, la priorità diventa mantenere il funzionamento del sistema, anche in assenza di tempo e risorse finanziarie, mentre si naviga tra le astrazioni di progettazione esistenti.