Concetti relativi al riutilizzo:

1. Oggetti applicazione e oggetti soluzione.
2. Ereditarietà delle specifiche ed ereditarietà dell'implementazione.
3. Delegazione.
4. Delega ed ereditarietà in Design Patterns.

Oggetti applicazione e oggetti soluzione

Le attività più difficili nello sviluppo di un sistema sono l'identificazione degli oggetti e la decomposizioni del sistema in oggetti. L'analisi dei requisiti è concentrata sul dominio applicativo per ottenere l'identificazione degli oggetti. Il sistem design, invece, è indirizzato sia al dominio applicativo che implementativo per ottenere l'identificazione dei sottosistemi. Infine l'Object Design focalizza sul dominio implementativo: identificazioni di più oggetti.
Durante l'analisi dei requisiti, le tecniche utilizzate per trovare gli oggetti si basano sui seguenti passi:

1. Si parte con i casi d'uso
2. Si identificano gli oggetti partecipanti
3. Si effettua un'analisi testuale del flusso di eventi(alla ricerca di nomi, verbi, etc..)
4. Si estraggono gli oggetti del dominio applicativo intervistando i clienti
5. Si trovano gli oggetti usando conoscenze generali.

Durante il System Design, le tecniche per trovare gli oggetti si basano:

1. Sulla decomposizione in sottosistemi.
2. Sul tentativo di identificare gli stati e le partizioni.

Durante l'Object Design, le tecniche per trovare gli oggetti si basano:

1. Sulla ricerca degli oggetti addizionali, applicando conoscenze del dominio implementativo.
2. Perfezioniamo e dettagliamo sia gli oggetti di applicazione che quelli di soluzione.

Oggetti di applicazione, detti anche "oggetti dominio", rappresentano concetti del dominio rilevanti per il sistema. Gli oggetti soluzione rappresentano componenti che non hanno una controparte nel dominio dell'applicazione, come archivi di dati persistenti, oggetti dell'interfaccia utente o middleware.

Ereditarietà delle specifiche ed ereditarietà dell'implementazione

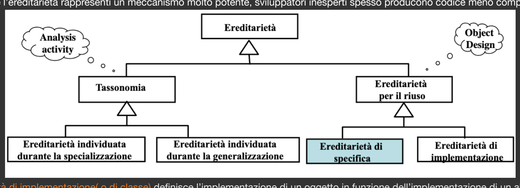
Durante l'analisi, usiamo l'ereditarietà per organizzare gli oggetti in una gerarchia comprensibile (descrizione di tassonomie). Con tale uso, ci consente di differenziare il comportamento comune del caso generale(superclasse) dal comportamento specifico degli oggetti specifici (sottoclassi). Quindi partendo dagli oggetti astratti, i lettori del modello di analisi comprendono le funzionalità di base del sistema e scendendo acquisiscono informazioni sui oggetti concreti e comportamenti specifici.
L'utilizzo dell'ereditarietà durante l'Object Design permette di ridurre le ridondanze, migliorare l'estensibilità, la modificabilità e il riuso del codice.

- Esempio: non dobbiamo modificare un metodo difetto ad ogni classe specializzata ma soltanto alla superclasse.

Inoltre fornendo classi e interfacce astratte utilizzate dall'applicazione possiamo scrivere nuovi comportamenti specializzati scrivendo nuove sottoclassi conformi alla interfacce astratte.

- Esempio: un applicazione che manipola le immagini in termini di una classe di immagini astratta, che definisce tutte le operazioni che tutte le immagini dovrebbero superare e una serie di classi specializzati per ogni formato di immagine.

Nonostante l'ereditarietà rappresenti un meccanismo molto potente, sviluppatori inesperti spesso producono codice meno comprensibile e più fragile di quello che otterrebbero senza farne uso.



L'ereditarietà di implementazione(o di classe) definisce l'implementazione di un oggetto in funzione dell'implementazione di un altro oggetto. Quindi gli sviluppatori riutilizzano il codice usando la tecnica "override" una classe esistente con lo scopo di perfezionare il comportamento della classe relativa.

Esempio:
Abbiamo una struttura dati stack:

Stack
push(Object elementi) pop() Object top()

Abbiamo la classe list che svolge operazioni simili a quelle necessarie per realizzare uno stack:

LIst
add() remove(int i) get(int i)

Usando l'ereditarietà di implementazione:

```
Implementazione usando l'ereditarietà di implementazione
Class Stack extends List { //Constructor omitted
    Object top() {
        return get(this.size()-1);
    }

    void push (Object elem){
        add(elem);
    }

    void pop() {
        remove(this.size()-1);
    }
}
```

Le operazione della superclasse sono esposte all'utilizzatore della sottoclasse quindi possono essere utilizzate in modo inaspettato. Ma in tutto ciò, non viene segnalato nessun errore, né in fase di compilazione né in fase di esecuzione. Si tratta di errore logico, ovvero che si utilizza la struttura LIFO in modo scorretto.

L'ereditarietà di specifica(o di interfaccia, o subtyping) definisce la possibilità di utilizzare un oggetto al posto di un altro. Quindi si eredita da una classe astratta che possiede operazioni già specificate ma non implementate.

In questo contesto, è stato formulato il principio di sostituzione di Liskov nella quale afferma:

"Se un oggetto di tipo S può essere sostituito ovunque ci si aspetta un oggetto di tipo T, allora S è un sottotipo di T"

Afferma che un metodo scritto in termini di una superclasse T deve essere in grado di usare istanze di qualunque sottoclasse di T senza sapere se utilizza la superclasse o una sua sottoclasse. Gli oggetti appartenenti a una sottoclasse devono essere in grado di esibire tutti i comportamenti e e le proprietà esibiti da quelli appartenenti alla superclasse in modo tale da poter essere "sostituiti" senza intaccare la funzionalità del programma.
Una relazione di ereditarietà che soddisfa tale principio è chiamata Ereditarietà stretta.

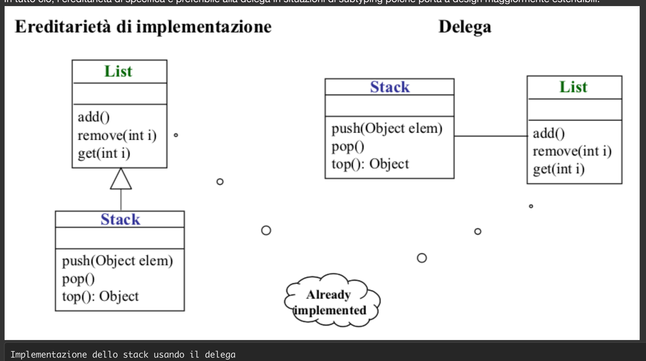
L'ereditarietà permette di diminuire il grado di disaccoppiamento tra le classi. Ma ciò introduce un forte accoppiamento tra la superclasse e le sottoclassi. Questo è accettabile quando la gerarchia dell'ereditarietà rappresenta una tassonomia (due oggetti sono simili). Come List e Stack che sono strettamente accoppiati che introducono molti problemi qualora List subisse delle modifiche. Anche se usare tale metodo ci farebbe risparmiare tempo in quanto una superclasse dispone dei metodi che vorremmo riutilizzare dovremo cercare di rispettare la tassonomia in modo da rispettare l'accoppiamento.

Composizione

Un'altra soluzione valida rispetto all'uso dell'ereditarietà di implementazione sarebbe la delega.
Due oggetti collaborano nel gestire una richiesta. Quindi un oggetto Client che richiede l'esecuzione di un'operazione della classe delegata, invia un messaggio alla classe delegata. In questo modo si ha due vantaggi:

1. L'estensibilità: la classe , che nell'esempio dell'ereditarieta conteneva i metodi della superclasse, non può utilizzare i metodi della List quindi possiamo cambiare la rappresentazione interna della classe in un'altra classe(un elenco) senza dover influenzare i client della classe corrispondente.
2. Subtyping: la classe corrispondente non eredita una superclasse quindi non può essere sostituito dalla superclasse in qualsiasi pezzo di codice che utilizzava tale superclasse. Quindi la superclasse si comporterà allo stesso modo.

In tutto ciò, l'ereditarietà di specifica è preferibile alla delega in situazioni di subtyping poiché porta a design maggiormente estendibili.




```
Implementazione dello stack usando il delega
class Stack{
    private List aList;

    Stack(){
        aList=new List();
    }

    Object top(){
        return aList.get(aList.size()-1);
    }

    void push(Object elem){
        aList.add(elem);
    }

    void pop(){
        aList.remove(aList.size()-1);
    }
```

Delega:

- Pro:
  - L'incapsulamento non è violato: si accede agli oggetti solo attraverso la loro interfaccia.
  - Flessibilta: consente di comporre facilmente comportamenti in fase di esecuzione e di cambiare il modo in cui questi comportamenti sono composti.
- Contro:
  - E' definita dinamicamente durante l'esecuzione attraverso oggetti che acquisiscono riferimenti ad altri oggetti: più inefficiente in esecuzione rispetto a software statico.

Eridatarietà:

- Pro:
  - Definita staticamente al momento della compilazione.
  - Immediata da utilizzare.
- Contro:
  - E' impossibile cambiare l'implementazione ereditata durante l'esecuzione.
  - L'implementaccone di una sottoclasse diventa strettamente dipendente dalla classe padre infatti qualsiasi modifica nell'implementazione della classe padre induce modifiche nella sottoclasse.