object oriented design

attività di progetto

- l'obiettivo del progetto è
  - raggiungere la piena comprensione dei dettagli importanti legati alla realizzazione di un sistema
  - il sistema deve soddisfare requisiti funzionali ma anche di flessibilità, manutenibilità, estendibilità, ecc.
- lo strumento è
  - la descrizione di un modello "di progetto" mediante un opportuno linguaggio
  - il modello deve evidenziare gli aspetti rilevanti e ignorare tutti gli altri
    - può non essere semplice trovare il giusto compromesso
    - l'esperienza aiuta

design class diagram

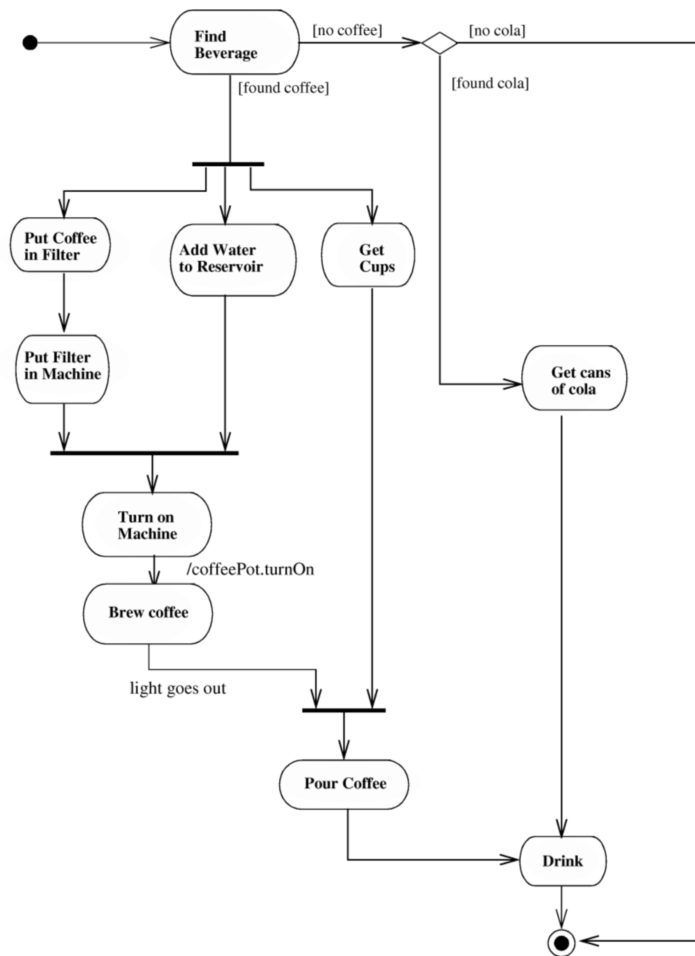
- esprimiamo il modello di progetto in notazione UML
- utilizziamo tutta la notazione necessaria
    - navigabilità
    - aggregazione
    - visibilità
    - ecc.
- mostra
  - un insieme di interfacce e classi software
    - descritte in maniera indipendente dal linguaggio
    - senza dettagli implementativi irrilevanti
  - un insieme di associazioni tra interfacce e classi software
    - ciascuna istanza dei una associazione è una relazione tra oggetti software
      - realizzata mediante puntatori o altro

altri strumenti

- interaction diagrams
  - sequence diagrams
  - collaboration diagrams

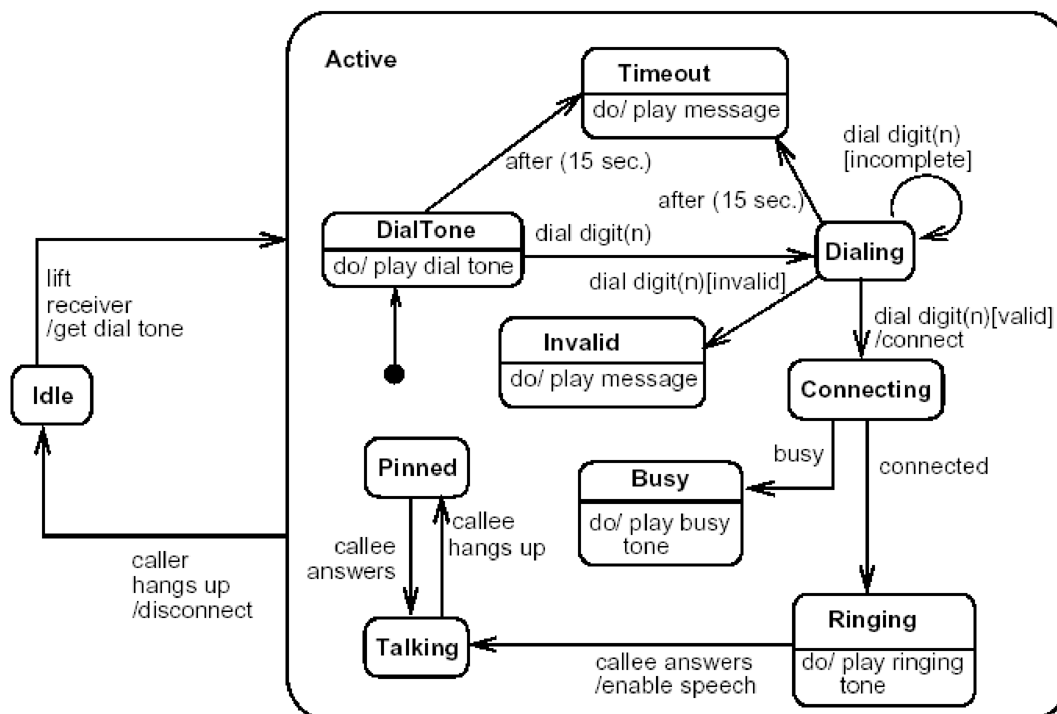
altri strumenti

- activity diagrams



altri strumenti

- state diagrams



modello di progetto: responsabilità

- il modello di progetto esprime la *responsabilità* di oggetti software
  - di conoscere
    - attributi
    - associazioni
    - attributi o associazioni derivate
  - di fare
    - attività interne all'oggetto
      - coinvolgono lo stato, i parametri e valori di ritorno
    - attività interattive con altri oggetti
      - attivare attività di altri oggetti
      - controllare attività di altri oggetti

relazione con il dominio

- minimizzare il gap semantico con il dominio
  - classi nell'analisi danno luogo spesso a interfacce o classi nel progetto
  - usare nomi coerenti
- il livello di dettaglio dipende dal progettista
  - in genere...
    - progetto interfacce molto dettagliato
    - progetto implementazioni meno dettagliato

arginare i cambiamenti

- un buon progetto può essere facilmente adattato ai cambiamenti
- ciascun cambiamento tende a propagarsi all'interno del progetto
  - *impatto* del cambiamento
- una interfaccia ben progettata è una barriera alla propagazione dei cambiamenti
  - accoppiamento basso tra implementazione e clients
- un buon progetto usa interfacce stabili per bloccare la propagazione dei cambiamenti

il progetto orientato alle interfacce

- almeno in una prima fase progettare solo interfacce
  - metodi pubblici
    - eventualmente annotate con pre/post condizioni quando non chiare
  - stato esterno
- e associazioni
  - quando le associazioni sono desumibili dalle interfacce
    - fanno parte dello stato esterno
  - pre/post condizioni possono coinvolgere altri oggetti
    - espressi tramite metodi accessor delle loro interfacce

progettare buone interfacce

hiding

- i dettagli implementativi possono (leggi devono!) essere nascosti dall'interfaccia
  - il client conosce solo l'interfaccia
    - semplifica l'utilizzo da parte del server
    - diminuisce il numero di dettagli da cui dipende il client
  - scegliere cosa esporre può essere un problema
  - interfacce che permettono l'accesso agli attributi sono possibili ma molto pericolose!
    - usa metodi getter and setter (x=getX(), setX(x))
  - un oggetto può avere interfacce diverse verso oggetti diversi
    - es. l'utente vede un sito di commercio elettronico in un certo modo, gli operatori di backoffice lo vedranno in un modo diverso, il system administrator in un altro ancora ecc.

hiding: esercizio

- progettare l'interfaccia di un oggetto software r: RubricaTelefonica nei seguenti casi
  - l'utente della rubrica che può solo consultare
  - l'utente della rubrica che può anche modificare la rubrica
  - il system administrator dell'oggetto r che ha bisogno di dati sull'efficienza di r
  - lo sviluppatore di RubricaTelefonica che deve verificare che il layout in memoria delle singole voci della rubrica sia corretto.

interfacce: precondizioni

- a ciascun metodo è associata una precondizione
  - condizione che deve essere soddisfatta perché il metodo possa essere legalmente chiamato
  - la condizione dipende da
    - parametri P
    - stato esterno dell'oggetto al momento della chiamata S_o
    - stato esterno di altri oggetti S_x
  - è un predicato su P, S_o, S_x che può essere vero o falso
    - per un metodo M indichiamo la precondizione così: pre M(P, S_o, S_x)

interfacce: postcondizioni

- a ciascun metodo M è associata una postcondizione
  - condizione che è sempre soddisfatta all'uscita di M
  - la condizione è su
    - parametri P
    - stato esterno dell'oggetto al momento della chiamata S_o
    - stato esterno di altri oggetti al momento della chiamata S_x
    - valore di ritorno R
    - stato esterno dell'oggetto all'uscita S'_o
    - stato esterno di altri oggetti all'uscita S'_x
  - è un predicato su P, S_o, S_x, R, S'_o, S'_x che può essere vero o falso
    - per un metodo M indichiamo la precondizione così:
      post M(P, S_o, S_x, R, S'_o, S'_x)

semplifichiamo

- il caso in cui pre M e post M non coinvolgono S_x è frequente
  - pre M(P, S_o)
  - post M(P, S_o, R, S'_o)
- questo caso è frequente per il principio del "basso accoppiamento"
  - cioè i progettisti tendono ad evitare dipendenze da S_x

accessors (o queries)

- un metodo M è un *accessor* (o *query*) se
  - $\forall$ P, S_o: post M (P, S_o, R, S'_o) $\rightarrow$ $S'_o = S_o$ e R è funzione di P e S_o
  - in altre parole
    - per qualsiasi parametro attuale lo stato non cambia e
    - il valore di ritorno dipende dai parametri e dallo stato corrente
- tali metodi sono “non invasivi”, li possiamo inserire in una sequenza di messaggi senza fare danni!
- esercizio
  - mostra un esempio di interfaccia con un metodo accessor senza parametri (fai un esempio concreto)
  - modifica il precedente esempio aggiungendo un accessor con un parametro
  - e con due parametri

modifiers

- un metodo M è un modifier se
  - $\exists$ P, S_o: post M (P, S_o, R, S'_o) $\rightarrow$ $S'_o \neq S_o$
  - in altre parole
    - esistono dei valori dei parametri attuali e dello stato per cui lo stato cambia
  - nota che il cambiamento di stato può influenzare il risultato degli accessor chiamati dopo
  - tali metodi *sono invasivi*
    - non possiamo inserire una chiamata ad un modifier in una sequenza di messaggi a cuor leggero, la sequenza potrebbe non funzionare più.
- esercizio
  - dai un esempio concreto di un oggetto con una interfaccia dotato di un metodo modifier
  - dai un esempio di stato dell’oggetto e di chiamata al modifier che *non* modifica tale stato

reazione tra interfaccia e stato

- il controllo sullo stato dipende dall'interfaccia
  - riusciamo a portare un oggetto in uno stato qualsiasi?
  - riusciamo sempre a capire qual'è lo stato dell'oggetto?
  - possiamo capirlo senza modificare lo stato?
- queste domande hanno senso sia per lo stato interno che per quello esterno

lo stato

- i concetti di spazio degli stati, osservabilità e raggiungibilità usati in teoria dei sistemi (o controlli automatici) si applicano anche agli oggetti
- i nostri oggetti sono
  - sistemi NON lineari (e non linearizzabili!)
  - a tempo discreto
  - sono automi a stati finiti ma lo stato ha una struttura
- lo spazio degli stati è l'insieme di tutti i possibili valori che lo stato dell'oggetto può assumere indipendentemente dall'interfaccia
  - per lo stato esterno è un modello
  - per lo stato interno, nei casi più semplici, è il prodotto cartesiano degli stati esterni dei vari attributi che contiene

stati osservabili

- lo stato è *osservabile* se esiste una procedura (una sequenza di messaggi) per ottenere piena informazione su di esso
- l'osservabilità può essere
  - parziale
  - dipendente dal tempo
  - dipendente dallo stato stesso
- attenzione la procedura potrebbe prevedere la chiamata a modifiers!
  - osservabilità senza modifica
  - osservabilità con necessità di modifica

stati osservabili

- esercizio: considera la seguente interfaccia
  - Stack
    - int getTop() // accessor, con una precondizione, quale?
    - void push(int x) // modifier
    - void pop() // modifier, con una precondizione, quale?
  - dai una procedura per osservare lo stato di uno Stack ad un dato istante
    - con modifica o senza modifica?
- esercizio: dai una classe la cui osservabilità dello stato dipende dallo stato

stati raggiungibili

- uno stato S è *raggiungibile* se esiste una procedura (che prevede l’invio di messaggi modifiers) per portare lo stato interno di un oggetto a S
- esercizio: considera la seguente interfaccia
  - Stack2
    - pair<int,int>getTop() // accessor
    - void push(int x, int y) // modifier
    - void pop2() // modifier, toglie due elementi dalla cima
  - esiste una procedura per portare lo stack ad avere un numero dispari di elementi?
  - da cosa dipende la risposta?

stati consistenti

- quando modelliamo decidiamo i valori leciti dello stato, i cosiddetti *stati consistenti*

pieno controllo

- la osservabilità e la raggiungibilita dipendono
  - per lo stato esterno del modelo dello stato e dall'interfaccia
  - per lo stato interno dal implementazione e dall'interfaccia
- è bene avere pieno controllo su ciò che accade negli oggetti
- linee guida per il design delle interfacce
  - progettare lo stato e le interfacce in modo da assicurare la piena osservabilità dello stato
    - l'osservabilità dovrebbe prevedere procedure semplici
    - usa metodi accessor come getX(): TipoDiX
  - progettare lo stato e le interfacce in modo che tutti gli stati consistenti siano raggiungibili e tutti quelli inconsistenti siano irraggiungibili
    - la raggiungibilità dovrebbe prevedere procedure semplici
    - usa metodi modifier come set(x,y,z) con precondizioni stringenti sui parametri)
    - se l'insieme degli stati coerenti è un prodotto cartesiano puoi usare più metodi modifier come setX(x), setY(y), setZ(z), poiché sono più flessibili

minimalità

- è utile avere interfacce con pochi metodi
  - sono più facili da apprendere per l'utente
  - accoppiamento basso tra implementazione e clients
  - meno errori nell'utilizzo
  - implementazioni più semplici
    - debugging più semplice
  - flessibilità nell'utilizzo
    - tante piccole interfacce permettono anche miglior riuso
  - molti "design patterns" richiedono di realizzare interfacce
    - potremmo dover avere molte realizzazioni di una stessa interfaccia
    - realizzare interfacce semplici è più facile

minimalità

- qualsiasi interfaccia può avere un numero illimitato di metodi
  - è solo una questione di fantasia
  - la maggior parte dei metodi...
    - sono utili in casi molto rari
    - può essere realizzata in termini di altri metodi “primitivi”
- le interfacce migliori hanno i metodi strettamente indispensabili
  - realizza gli altri metodi esternamente (es. classi “utilities”)
  - funzioneranno con qualsiasi implementazione: meno lavoro!

minimalità: esercizio

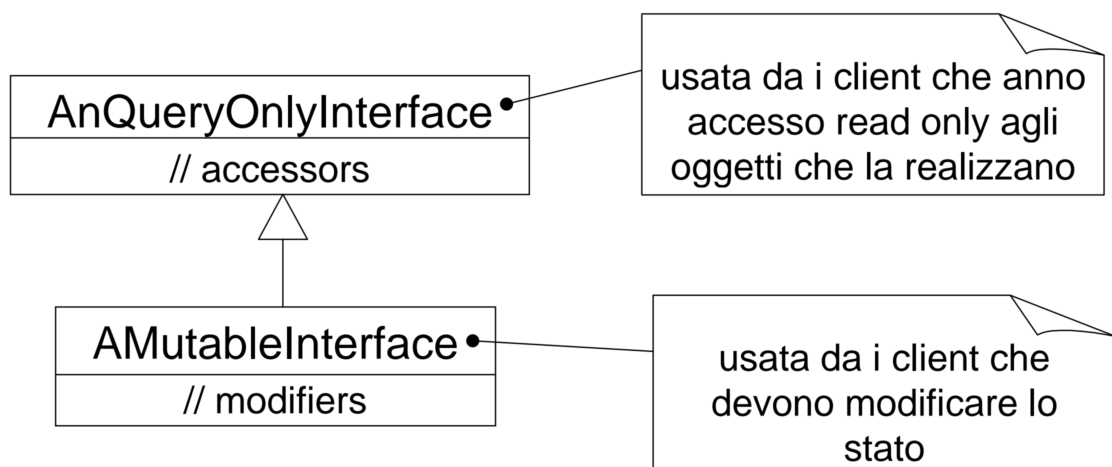
- si vuole progettare una architettura software per il calcolo del codice fiscale a partire dai dati sulle persone
- dare l’interfaccia della classe persona
- quale classe ha la responsbilità per il calcolo del codice fiscale?

minimalità: mutable vs. immutable

- se una classe deve essere acceduta in sola lettura rendere questo esplicito tramite una interfaccia immutable (solo di accessor)
  - per la maggior parte degli utilizzi di oggetti il modo “sola lettura” è sufficiente (regola dell’80/20)
  - previene utilizzi indiscriminati di modifiers
    - meno bugs
    - codice più chiaro
  - aumenta disaccoppiamento
    - cambiamenti sulla classe client non influenzano la classe server
  - diminuisce il numero di classi che sono modificatori potenziali di un oggetto aumentando la comprensibilità del progetto

minimalità: mutable vs. immutable

- spesso un oggetto viene creato e inizializzato e poi letto per il resto della sua vita
- separare la parte mutable dalla immutable



esercizio

- si vuole modellare il concetto di “Network” al fine di utilizzarlo all’interno di un editor grafico di reti.
  – una network contiene “Router” e “Subnet”
  – ciascun router può essere connesso a più subnet tramite il concetto di “Interface” (coppia router-subnet)
  – deve essere possibile osservare la topologia della rete
  – deve essere possibile modificare lo stato della network in termini di aggiunta o cancellazione di router subnet ed interface
  – deve essere possibile effettuare un test di connessione della rete
  – deve essere possibile verificare la presenza di cicli
- dare un progetto basato su interfacce per Network

Immutable e value semantic

- oggetti con uno stato molto piccolo sono progettati come valori immutabili
  – niente metodi modifier
  – identità basata su uguaglianza
  – es. la classe String di java
- candidati per overloading di operatori
  – se ammessi dal linguaggio di programmazione (es. C++)
  – evitare l’overloading di operatori negli altri casi

esercizio scacchi