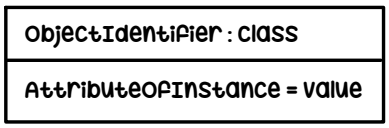
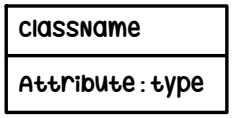
Lo schema concettuale è costituito da:
1- Diagramma delle classi e degli oggetti: descrive le classi dell'applicazione e le loro proprietà, e gli oggetti particolarmente significativi
2- Diagramma delle attività: descrive le funzionalità fondamentali del sistema, in termini di processi modellati nel sistema
3- Diagramma degli stati e delle transizioni: descrive, per le classi significative, il tipico ciclo di vita delle sue istanze
4- Documenti di specifica: uno per ogni classe e uno per ogni use case

Diagrammi UML (unified modelling language) studiati:
1- diagramma delle classi (dati)
2- diagramma delle attività (come manipolare i dati)
3- diagrammi degli stati e delle transizioni (come costruire oggetti reattivi, che cambiano stato quando reagiscono ad eventi ricevuti)

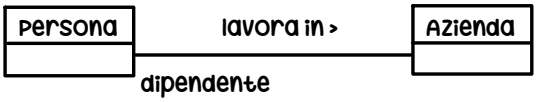
OGGETTO (livello estensionale)
Modella un'istanza della classe (dominio di riferimento). La classe indicata é la più specifica possibile.



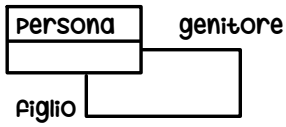
CLASSE (livello intensionale)
Modella un insieme di istanze omogenee alle quali sono associate proprietà statiche e dinamiche. Nota che type è un'estrazione di valore, non di entità, non riguarda più le estrazioni di entità (si "esce" dal diagramma delle classi) infatti un tipo NON può essere una classe.



ASSOCIAZIONE
Modella una relazione tra 2 o più classi (sottoinsieme del prodotto cartesiano tra i domini delle classi). Si può specificarne un nome e un verso (il verso è riferito al nome non alla relazione). Si può associare un ruolo alla classe che partecipa alla relazione. Le istanze di una associazione sono detti link.

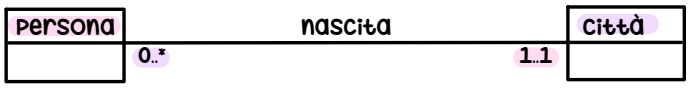


La specifica del ruolo é obbligatoria solo per associazioni che insistono più volte sulla stessa classe:



Anche le associazioni possono avere attributi. Formalmente, un attributo di un'associazione è una funzione che associa ogni link che è istanza dell'associazione un valore di determinato tipo.

MOLTEPLICITÀ DI ASSOCIAZIONI BINARIE
Il vincolo di molteplicità sulla partecipazione delle istanze di una classe all'associazione è scritto nel lato opposto rispetto all'associazione stessa.



La molteplicità specifica il numero minimo e massimo di partecipazioni all'associazione. Quando non specificata, la molteplicità di default è 0..* . Le molteplicità più rilevanti sono:
0..* nessun vincolo, si può omettere
0..1 è una funzione parziale, cioè con partecipazione opzionale
1..* indica una partecipazione obbligatoria, senza limiti sul numero di partecipazioni
1..1 la relazione è una funzione
0..y indica solo il limite massimo, la partecipazione è opzionale
x..* indica solo il limite minimo, non quello massimo
x..y indica un range

La molteplicità nelle associazioni n-arie è espressa tramite commento nei casi in cui vada specificata esplicitamente.

MOLTEPLICITÀ DI ATTRIBUTI
La molteplicità di default per gli attributi è 1..1 . Possono esserci attributi detti multivalore con molteplicità diverse.

ASSOCIAZIONI ORDINATE
Un modo per definire un ordine su un'associazione è quello di aggiungere un attributo alle istanze di tale associazione (es. posizione:int). Questo metodo necessita però di un commento che specifichi che esiste una sola istanza dell'associazione per ciascun valore di posizione, da 1 ad n.

Un altro modo è quello di aggiungere l'asserzione {ordered}, dal lato della collezione, che indica che le tuple dell'associazione sono ordinate e formano quindi una lista (che quindi è ordinata) senza ripetizioni.

GENERALIZZAZIONE
È possibile che tra due classi sussista la relazione IS-A, per cui ogni istanza di una classe sia istanza anche dell'altra. In UML questa relazione si modella tramite generalizzazione, che coinvolge una superclasse e una o più sottoclassi/classi derivate.
L'ereditarietà tra classi coinvolte in generalizzazioni fa si che le sottoclassi ereditino gli attributi della superclasse (possono poi avere attributi aggiuntivi), le associazioni (ogni istanza della sottoclasse può essere coinvolta nelle istanze delle associazioni in cui è coinvolta la superclasse) e le rispettive molteplicità. Una superclasse può avere più generalizzazioni che derivano da essa.

GENERALIZZAZIONE DISGIUNTA
Una generalizzazione in cui vale la asserzione {disjoint} rappresenta una generalizzazione in cui gli insiemi che rappresentano le sottoclassi sono disgiunti.

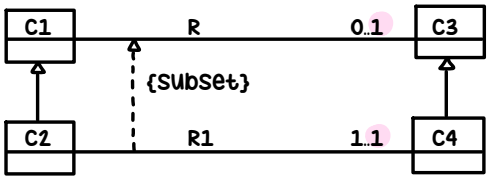
GENERALIZZAZIONE COMPLETA
Una generalizzazione in cui vale la asserzione {complete} rappresenta una generalizzazione in cui l'intersezione tra insiemi che rappresentano le sottoclassi forma l'insieme che rappresenta interamente la superclasse.

Le due asserzioni possono essere specificate singolarmente o insieme {disjoint, complete}.

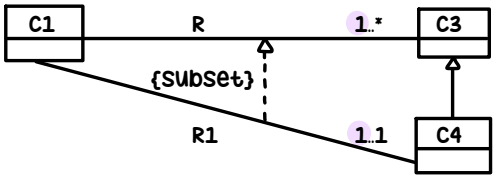
EREDITARIETÀ MULTIPLA
In UML, a contrario di Java, è possibile avere l'ereditarietà multipla, per cui una sottoclasse può essere in relazione di generalizzazione con più superclassi (es. studenteLavoratore in generalizzazione con due superclassi, studente e Lavoratore, entrambe in relazione di generalizzazione con la superclasse Persona).

SPECIALIZZAZIONE DI ATTRIBUTO
Una sottoclasse può avere, per lo stesso attributo presente nella superclasse, un dominio più ristretto che permette la specializzazione.

SPECIALIZZAZIONE DI ASSOCIAZIONE
L'asserzione del vincolo {subset} specifica una dipendenza tra due relazioni, una delle quali è sottoinsieme dell'altra



Nota che in questo caso la sottoclasse C2 eredita la molteplicità massima con cui partecipa all'associazione R1.



In questo caso invece (l'associazione R1 insiste sulla stessa classe C1) la partecipazione delle istanze di C1 alla relazione R eredita la molteplicità minima della partecipazione alla relazione R1.

OPERAZIONI
Le classi hanno anche proprietà dinamiche che in UML si definiscono operazioni che seguono questa sintassi:
NomeOperazione (nomeAttributo : tipoAttributo) : tipoRisultatoDellaOperazione
Il tipo degli attributi e del risultato può essere un tipo valore o un identificatore di una classe del diagramma.
Il risultato dell'operazione viene assegnato all'oggetto di invocazione, cioè all'istanza su cui viene invocata l'operazione, che è un'istanza della classe su cui tale operazione è definita.

FASI DI CICLO DI VITA DEL SOFTWARE
1- Raccolta dei requisiti (non ce ne occupiamo, li riceviamo già stilati)
2- Analisi: produce lo schema concettuale, formato da diagramma delle classi e degli oggetti, diagramma delle attività, diagramma degli stati e delle transizioni.
3- Progetto e realizzazione: è la fase in cui si sceglie la tecnologia da utilizzare (per noi Java) per realizzare il progetto.
   L'output della fase di progetto è l'input della fase di realizzazione, ed è costituito da:
   1- scelta delle classi UML che hanno responsabilità sulle associazioni
   2- scelta e progettazione delle strutture dati
   3- scelta della corrispondenza tra tipi UML e Java
   4- scelta della gestione delle proprietà di una classe UML
   5- progettazione delle API delle principali classi Java
   ...
4- Verifica e manutenzione

RESPONSABILITÀ SU UN'ASSOCIAZIONE
Una classe ha responsabilità su un'associazione se, per ogni oggetto istanza di tale classe, vogliamo poter eseguire operazioni sulle istanze dell'associazione a cui l'oggetto partecipa, con lo scopo di: conoscere le istanze dell'associazione a cui l'oggetto partecipa, aggiungerne, cancellarne, o aggiornare il valore di un attributo di una di esse.
Esistono tre criteri per attribuire la responsabilità:
1- Criterio della molteplicità: se una classe partecipa ad un'associazione con molteplicità diversa da 0..* ha responsabilità su tale associazione.
2- Criterio delle operazioni: se su una classe è definita un'operazione che sfrutta una relazione per elaborare dei dati, la classe ha responsabilità su tale associazione.
3- Criterio dei requisiti: nel caso in cui non si applichino i due criteri precedenti l'associazione non può rimanere senza una classe che ne ha responsabilità. Allora tale responsabilità va individuata tra i requisiti.

Nel caso in cui l'associazione insista più volte sulla stessa classe, la responsabilità si attribuisce al ruolo piuttosto che alla classe.

STRUTTURE DATI
È importante distinguere, quando si parla di rappresentazioni di collezioni omogenee di oggetti, l'interfaccia (es. Set<Elem>) dalle implementazioni (es. HashSet<Elem>). Le strutture dati utilizzate saranno quelle del Collection Framework di Java.

CORRISPONDENZE TIPI UML E JAVA

TIPO UML	RAPPRESENTAZIONE IN JAVA
intero	int
interoPositivo	int
range x..y	int
reale	double
stringa	String
insieme	Set

GESTIONE DELLE PROPRIETÀ DELLE CLASSI IN JAVA
Final definisce una variabile che può essere inizializzata ma non assegnata.

DIAGRAMMA DELLE CLASSI

TRADUZIONE IN JAVA DEL DIAGRAMMA DELLE CLASSI
1- realizzazione delle singole classi
2- realizzazione delle associazioni
3- realizzazione delle generalizzazioni

1- REALIZZAZIONE DELLE SINGOLE CLASSI
La classe Java ha lo stesso nome della classe UML. Il costruttore va sempre ridefinito poiché un oggetto deve essere significativo fin dal momento in cui viene istanziato e questo non può essere possibile con il costruttore di default. Equals, hashcode e clone sono già definiti, e questo poiché le classi che definiamo in Java sono astrazioni di entità, per cui si possono usare le funzioni già definite. Vanno definite le funzioni di set (se mutabili) e get per gli attributi definiti per la classe. Tostring può essere omesso essendo principalmente per debug.

Le limitazioni sugli attributi di una classe si possono gestire tramite delle eccezioni (es. nel costruttore o nelle funzioni di set, se il valore passato è fuori dal range richiesto si segnala l'errore con una exception).

Per gli attributi multivalore si utilizzerà una interfaccia del Collection Framework (es. Set) e una struttura dati (es. HashSet) del tipo specificato dall'attributo. La costruzione della struttura dati, dichiarata tra i campi della classe, va gestita nel costruttore. Per le strutture dati, dal momento che sono astrazioni di valore e quindi rappresentano strutture non manipolabili, non si usa la funzione di set ma quelle di add e remove. Inoltre la funzione di get restituisce una copia (clone) della struttura dati contenente i dati di interesse, per evitare conflitti sulla struttura dati originaria.

Le operazioni definite per le classi vengono riportate nelle classi Java con la stessa segnatura e vengono semplicemente definite col codice necessario per il loro funzionamento.

2- REALIZZAZIONE DELLE ASSOCIAZIONI

Si distinguono i seguenti casi:

- associazioni binarie
  - a responsabilità singola
    - senza attributi
      - con molteplicità 0..1
      - con molteplicità 0..*
    - con attributi
      - con molteplicità 0..1
      - con molteplicità 0..*
  - a responsabilità doppia
  - con molteplicità diversa da 0..1 e 0..*
- associazioni n-aria
- associazioni ordinate

ASSOCIAZIONI BINARIE A RESPONSABILITÀ SINGOLA SENZA ATTRIBUTI CON MOLTEPLICITÀ 0..1

Nella classe che ha la responsabilità sull'associazione si aggiungono un campo corrispondente ad un oggetto (è un object identifier) dell'altra classe coinvolta e i metodi get e set della relazione.

ASSOCIAZIONI BINARIE A RESPONSABILITÀ SINGOLA SENZA ATTRIBUTI CON MOLTEPLICITÀ 0..*

Nella classe che ha la responsabilità sull'associazione si aggiungono un campo corrispondente ad una struttura dati HashSet di oggetti dell'altra classe coinvolta (inizializzata vuota nel costruttore) e i metodi inserisci (inserisce un nuovo link nell'hashset), rimuovi (rimuove un link) e get (restituisce l'hashset di oggetti legati all'oggetto di invocazione tramite l'associazione).

ASSOCIAZIONI BINARIE A RESPONSABILITÀ SINGOLA CON ATTRIBUTI CON MOLTEPLICITÀ 0..1

Si ricorre alla implementazione del tipo Link (TipoLinkAssociazione), che è astrazione di valore, che ha come campi gli oggetti coinvolti nella relazione e gli attributi di essa. Il costruttore controlla che gli oggetti che gli vengono passati non siano nulli ed assegna sia quelli sia il valore agli attributi. Si ha un get per ogni classe coinvolta nell'associazione ma non si hanno i set poiché si opta per un'implementazione funzionale. Si ha un get per ogni attributo. Si ha la ridefinizione di equals e hashcode nelle quali non si usano mai gli attributi.
Nella classe che ha responsabilità dell'associazione si avrà un campo TipoLinkAssociazione e le funzioni per l'inserimento (che controlla che il link e il secondo oggetto dell'associazione siano diversi da null e che il primo oggetto dell'associazione sia quello corrente this), la cancellazione, e la restituzione di link.

ASSOCIAZIONI BINARIE A RESPONSABILITÀ SINGOLA CON ATTRIBUTI CON MOLTEPLICITÀ 1..*

L'implementazione del tipo link rimane identica. Nella classe che ha responsabilità dell'associazione si avrà un hashset di tipo link, le funzioni di inserimento e rimozione di un tipo link nell'hashset (stessi controlli sugli elementi dell'associazione: il link deve essere diverso da null e il primo oggetto della relazione deve essere l'oggetto corrente this), la funzione di restituzione dell'insieme di tutti i tipi link relativi all'oggetto corrente.

ASSOCIAZIONI BINARIE A RESPONSABILITÀ DOPPIA [CON ATTRIBUTI]

Questa tipologia si implementa direttamente tramite il tipo link, in modo da averlo già predisposto nel caso di associazioni con attributi. Si avrà quindi un campo TipoLink, le funzioni inserisciLink, eliminaLink, getLink. La particolarità sta nell'implementazione delle due funzioni di inserimento ed eliminazione: in entrambe le classi che hanno responsabilità sull'associazione queste due funzioni chiameranno le rispettive funzioni di inserimento ed eliminazione di un'altra classe, la classe Manager.
La classe Manager conterrà quindi le due funzioni per l'inserimento e l'eliminazione, che prendono come parametro un link che viene inserito, dopo i controlli, in entrambe le classi che hanno responsabilità sull'associazione tramite un'altra funzione, inserisciperManager, questa stavolta definita in ciascuna due classi originarie e che prende come parametro non un link ma un oggetto Manager. La funzionalità del manager è quella di garantire che l'inserimento avvenga in entrambe le classi: per fare questo sfrutta una funzione delle classi originarie (quindi pubblica) che è sicuro di poter essere l'unico a sfruttare poiché essa richiede l'istanziazione di un oggetto Manager, che ha il costruttore private.
se una delle due classi partecipa con molteplicità 0..* invece di 0..1 il manager chiamerà la funzione inserisciperManager che nel caso della classe con molteplicità 0..* inserirà il link nell'insieme di link di tale classe (nel caso di molteplicità 0..1 il caso è quello precedente).

ASSOCIAZIONI BINARIE CON MOLTEPLICITÀ DIVERSA DA 0..1 E 0..*

Associazioni con tipi diversi di molteplicità, cioè 1..1 e n..m, non prevedono una soluzione "giusta". Si procede con l'implementazione che meglio soddisfa la struttura del progetto, senza alcuna limitazione sulle funzionalità di creazione, inserimento ed eliminazione, mentre nella funzionalità di lettura, get, si procede al controllo delle condizioni di molteplicità prima di restituire il risultato. Per fare questo avremo bisogno di un metodo aggiuntivo a supporto del get, che controlli che il numero totale di link (chiama .size() sull'hashset di link) sia adeguato rispetto alle condizioni, e che sia pubblico in modo che il client possa utilizzarlo per verificarlo autonomamente.
Nell'implementazione delle classi che partecipano a tali associazioni le molteplicità minima e massima sono dichiarate come public static final int MIN e MAX per facilitarne la fruizione e la manutenzione.

ASSOCIAZIONI N-ARIE

Nel caso di associazioni n-arie si ricorre sempre alla classe TipoLink (anche per associazioni senza attributi) che modellerà un link a n elementi invece che a due. La logica sia del TipoLink che dell'eventuale Manager rimane la stessa.

ASSOCIAZIONI ORDINATE

Le associazioni ordinate si realizzano nello stesso modo di quelle non ordinate, con l'unica differenza che per mantenere l'ordinamento si fa uso delle classi List e LinkedList invece di Set e HashSet. La funzione di inserimento di link dovrà controllare che il link non sia già presente per mantenere l'unicità richiesta dall'associazione ordered.

3- REALIZZAZIONE DELLE GENERALIZZAZIONI
La classe figlia deve estendere (extends) la classe genitore, e nel costruttore la prima istruzione deve essere la chiamata a super().

REGOLE DI VISIBILITÀ
Un attributo public è visibile a tutti, ovunque. Un attributo protected è visibile alla classe stessa, alle classi nello stesso package e alle classi derivate (sia dentro che fuori il package). Un attributo di cui non si specifica la visibilità è visibile alla classe stessa e alle classi nello stesso package ma non alle classi, anche quelle derivate, al di fuori di tale package. Un attributo private è visibile solo all'interno della classe stessa.

Nota che protected all'interno del package si comporta come public, mentre all'esterno del package si comporta come public per le classi derivate e come private per le altre classi. Per questo motivo in Java servirà dedicare un package ad una sola classe, facendo attenzione a non mettere mai classe genitore e classe figlia nello stesso package.

RIDEFINIZIONE
Nel caso di associazioni legate da generalizzazione è possibile eseguire ridefinizione di metodi con @override

GENERALIZZAZIONI DISGIUNTE E COMPLETE
Poichè Java non supporta l'ereditarietà multipla, si assume che ogni generalizzazione sia disgiunta. se invece una generalizzazione é completa, cioè le istanze delle classi figlie compongono tutte le istanze della classe genitore che quindi non può avere istanze proprie, si segna la classe genitore come abstract.

Nota che se si segna come abstract una funzione nella classe genitore abstract, le classi figlie saranno forzate a ridefinire tale funzione nella loro implementazione.

DIAGRAMMA DELLE ATTIVITÀ

Il diagramma modella delle attività, delle funzionalità atomiche sequenziate opportunamente e sulle quali possono essere definite condizioni o cicli, con la possibilità che più attività vengano eseguite concorrentemente. Più funzionalità rappresentano poi la funzionalità complessiva (la funzione). Una singola attività serve a descrivere una manipolazione di dati, ed è trasversale rispetto alle classi cioè può coinvolgerne più di una.

Il diagramma è composto da task (attività atomiche), dal flusso di controllo, e da segnali di I/O (servono a inserire o portare fuori dal diagramma le informazioni).

ATTIVITÀ
Esistono attività atomiche, che hanno un nome, un punto di ingresso e uno di uscita, e attività composte, che sono composizioni di attività atomiche.
Esistono simboli di inizio e di fine, di terminazione, di attesa, di eccezione, di invio di segnale output (come write) , di ricezione di segnale input (come read).

FLUSSO DI CONTROLLO
È modellato tramite alcuni elementi: la transizione, il punto di decisione (XOR SPLIT) e il punto di merge (XOR JOIN). Attraverso questi elementi è possibile modellare l'if-else, il while e il do-while. Altri costrutti sono: la fork (AND SPLIT) e la join (AND JOIN).

SPECIFICA DELLE OPERAZIONI
sono inserite all'interno della specifica di una classe:
IniziospecificaclasseC
    operazione (variabile : tipo, .. , variabile : tipo) : tiporisultato
        pre : condizione
        post : condizione
    ... (altre operazioni)
Finespecifica

La segnatura si prende dal diagramma delle classi. Le due condizioni sono quelle che devono valere prima di poter fare l'operazione e dopo che l'operazione è stata fatta.
Nella definizione della precondizione si usa this per indicare l'oggetto di invocazione dall'operazione, nella postcondizione si usano this, result, pre(alfa) (che indica il valore della espressione alfa nello stato corrispondente alla precondizione).

SPECIFICA DELLE ATTIVITÀ
La specifica di attività atomiche è simile alla specifica delle operazioni. La specifica di attività complesse è invece più complessa, e si sceglie di scriverla tramite pseudocodice riscrivendo il diagramma con l'aiuto di variabili ausiliarie.

SPECIFICA DI ATTIVITÀ ATOMICHE
IniziospecificaAttivitaAtomica alfa
alfa (parametro : tipo, .. , parametro : tipo) : (risultato1 : tipo, .. , risultaton : tipo)
    pre : condizione
    post: condizione
Finespecifica

Nella definizione di precondizioni e postcondizioni NON si può usare this, non essendoci un oggetto di invocazione, mentre si possono usare risultato1, .. ,risultaton per riferirsi ai risultati e pre(alfa) per riferirsi al valore dell'espressione alfa nello stato corrispondente alla precondizione.

SPECIFICA DI ATTIVITÀ COMPLESSE
La specifica è formata dalla segnatura (che è analoga a quella delle attività atomiche), da variabili di processo, e dal flusso di controllo.
Le variabili di processo sono variabili ausiliarie che si aggiungono ai parametri di ingresso e uscita per formare una sorta di memoria di lavoro locale delle attività. Possono essere scritte e lette più volte e servono a memorizzare i risultati intermedi per passarli come parametri a sottoattività successive. Queste sottoattività non hanno accesso a queste variabili se non gli vengono passate come parametri.

PATTERN FUNTORE
È una classe che rappresenta una funzione, cioè le cui istanze rappresentano attivazioni (computazioni) di una funzione. La funzione diventa la classe (implements Runnable) i cui dati (campi) sono i parametri (sia di ingresso che di uscita) della funzione più un campo per il risultato restituito, il corpo della funzione diventa il corpo di un metodo della classe (che può essere runnato una volta sola, controllo tramite Boolean) che assegna il risultato al campo dati che usiamo appunto come campo per il risultato. Il metodo che restituirà il risultato controlla prima che la funzione sia stata davvero eseguita (controllo su campo dati boolean impostato a true nel metodo che rappresenta il corpo della funzione) e poi restituisce il valore del risultato.
Nel main si istanzia il funtore e si passa ad un thread che poi si fa partire.
Nota che le varie funzioni nella classe funtore sono synchronized.

serve a separare l'eseguibile dall'esecutore (viene utilizzato per i thread Java).

PATTERN SINGLETON
serve a realizzare una classe che ha un solo oggetto. Per fare ciò serve impedire l'uso dell'operatore new, cioè serve rendere il costruttore private, per poi usare un metodo statico per farci restituire il singolo oggetto ogni volta che serve.

REALIZZAZIONE CON LAZY CONSTRUCTION
nella classe il campo che definisce l'unica istanza sarà private, static e conterrà l'unica istanza di tale classe. Il costruttore sarà private. Il metodo per restituire l'istanza sarà synchronized per essere thread-safe e ritornerà l'unica istanza della classe (se l'unica istanza risulta uguale a null la istanzierà prima di restituirla).

REALIZZAZIONE CON EAGER CONSTRUCTION
nella classe il campo private static che definisce l'unica istanza chiamerà direttamente il costruttore con new. Il costruttore sarà private, inizializzerà i campi significativi. Il metodo che restituisce l'unica istanza sarà synchronized per essere thread-safe.

REALIZZAZIONE
Per le attività atomiche implementiamo ogni task come una classe funtore (implements Task, l'eseguibile deve essere un runnable). All'interno di tale classe sarà definito un metodo public void esegui() che contiene il codice del task. Questo metodo sarà utilizzato solo dal task executor. Implementiamo il task executor come un singleton che contiene un metodo public synchronized void perform(Task t) di esecuzione del task, che al suo interno chiama t.esegui(). L'implementazione di un singolo task executor fa si che non vi sia possibilità di deadlock.

Per i segnali di ingresso e uscita si implementa una classe segnaliIO con un metodo statico per ogni input/output.

Per le attività complesse si implementa la attività principale tramite una classe funtore, senza campi dati né costruttore (non avendo campi dati il costruttore di default va bene). Si definiscono un campo Boolean eseguita per il controllo sull'avvenuta esecuzione e si istanzia un private TaskExecutor exc = TaskExecutor. getInstance(); che corrisponde all'unico task executor presente. All'interno della run ci saranno tutte le variabili di processo (saranno variabili locali), le chiamate a segnaliIO, le varie operazioni che compongono l'attività complessa.
È possibile implementare sotto attività dell'attività principale (ad esempio due thread concorrenti che partono da un'attività complessa principale) allo stesso modo di attività complesse (quindi tramite pattern funtore che implementa runnable, variabili locali interne al run ecc..)

DIAGRAMMA DEGLI STATI E TRANSIZIONI

È ciò che modella degli oggetti che sono reattivi rispetto agli eventi. La parte di analisi ancora una volta si fa a partire dal diagramma di stati e transizioni fornito da UML, definito per una classe, che descrive l'evoluzione di un oggetto dovuta al fatto che questo oggetto riceva eventi che triggerano side effect sul diagramma delle classi e cambiano di stato tale oggetto. Il diagramma descrive la sequenza di stati, le risposte e le azioni che un oggetto attraversa nell'arco della sua vita. Uno stato rappresenta una situazione in cui un oggetto ha un insieme di proprietà stabili. Una transizione modella un cambiamento di stato ed è denotata da: evento[condizione] / azione . L'evento è l'evento che viene ricevuto, che può avere condizioni aggiuntive specificate, l'azione è l'azione intrapresa una volta che la transizione viene attivata e lo stato viene cambiato (l'azione può essere di side effect, sul diagramma delle classi o sull'oggetto stesso, o di invio di eventi). Il diagramma deve essere deterministico (transizioni dallo stesso stato hanno eventi diversi, o combinazioni dello stesso evento e di due condizioni di cui una è la negazione dell'altra). In alcuni casi si vogliono mostrare dei processi che l'oggetto esegue senza cambiare stato. Questi processi si chiamano attività e si specificano negli stati con la notazione: do/attività.

STATI COMPOSTI
Uno stato composto (o macro stato) è uno stato che ha un nome e contiene a sua volta un diagramma. Per il macro stato esiste uno stato iniziale. I sotto stati ereditano le transizioni in uscita del macro stato.

OGGETTI REATTIVI
Si scambiano eventi, che hanno mittente e destinatario (ammettiamo comunicazione punto-punto e broadcasting) e possono avere parametri con specifico contenuto informativo detto payload.
L'azione può lanciare eventi, per altri oggetti (punto-punto) o in broadcasting. L'azione è solitamente identificata con l'evento lanciato.
verrà data una specifica di ciò che avviene ad ogni transizione: quali eventi sono recepiti e lanciati (e a chi), come cambiano eventuali variabili di stato ausiliarie associate allo stato dell'oggetto, come cambia l'istanziazione del diagramma delle classi.
Le variabili di stati ausiliarie servono alla corretta realizzazione delle azioni associate alle varie transizioni. sono cose che vanno memorizzate in una transizione per poi essere usate in una transizione successiva.

PROGETTO DI CLASSI CON ASSOCIATO DIAGRAMMA DEGLI STATI E DELLE TRANSIZIONI

DECISIONI PRELIMINARI:
-gli eventi sono messaggi che oggetti reattivi si scambiano. Hanno mittente (può non essere dichiarato) e destinatario, ed è permessa la comunicazione point-to-point o broadcasting, mentre non lavoreremo a quella multicasting)
-viene definito un framework di gestione di eventi
-la gestione degli eventi viene attuata inizialmente tramite un unico flusso di controllo (ALL'ESAME NO). successivamente i flussi di controllo dei singoli oggetti vengono resi indipendenti e concorrenti (ogni oggetto reattivo vivrà sul un thread dedicato)

PATTERN OBSERVABLE OBSERVER (useremo Listener invece di Observer nelle implementazioni)
-Un observable rappresenta un oggetto osservabile da altri oggetti legati ad esso ("registrati") detti Observer
-ogni observable registra i suoi observer tramite la funzione addObservable(observer) e può notificare qualcosa attraverso la funzione fireAll(), che chiama su ciascun observer registrato il suo metodo fired()
-ogni observer implementa una funzione per reagire alle notifiche dell' observable, chiamata fired()

Nel nostro caso implementeremo un Environment, che sarà l'unico observable, a cui sono registrati tutti gli oggetti reattivi che sono i vari observer. Gli observer mandano eventi all'environment (anche quegli eventi destinati ad altri oggetti reattivi. L'environment si occuperà dello smistamento).

REALIZZAZIONE

REALIZZAZIONE DEGLI OGGETTI
Gli oggetti reattivi sono oggetti, quindi come tutti gli altri sono nel diagramma delle classi, con la struttura già vista. Ci si deve occupare poi dell'aspetto reattivo dell'oggetto, come modellato dal diagramma degli stati e delle transizioni. Per rendere l'oggetto reattivo viene dichiarato che esso implementa l'interfaccia Listener (ancora da definire), e che quindi è un osservatore dell'environment.

REALIZZAZIONE DEGLI EVENTI
Gli eventi sono rappresentati da oggetti di una classe Java evento (astrazione di valore. Le istanze sono oggetti immutabili i cui metodi pubblici non fanno side effect). La classe evento rappresenta eventi generici dotati di mittente e destinatario. Quando il destinatario è null allora l'evento è in broadcasting. Quando il mittente è null allora il mittente non si è dichiarato e rimane nascosto. Tutti gli eventi specifici per una data applicazione sono istanze di classi derivate da Evento. Gli oggetti la cui classe più specifica è Evento vengono usati direttamente solo per abilitare transizioni in casi particolari.
Viene definito un costruttore, i metodi get, equals, hashcode. Non viene ridefinito clone in quanto gli eventi sono oggetti immutabili e quindi non può esserci condivisione di memoria.

REALIZZAZIONE DEGLI STATI
Nella realizzazione della classe relativa all'oggetto dinamico si aggiungeranno, oltre alla parte già vista per gli oggetti in generale, alcuni elementi: una enumerazione (public static enum stato {STATO_0 , .. , STATO_n }) statica, cioè comune a tutta la classe di oggetti. un campo private stato statocorrente (inizializzato allo stato iniziale stato.STATO_0). variabili di stato ausiliarie private (che servono per conservare informazioni che vengono reperite in una transizione ed utilizzate in una successiva) e che quindi sono inaccessibili dal cliente, possono essere accedute solo dalle transizioni. Il metodo getstato che ritorna la variabile statocorrente.

REALIZZAZIONE DELLE TRANSIZIONI
Nella realizzazione della classe relativa all'oggetto dinamico si aggiungerà, oltre alla parte già vista per gli oggetti in generale e quella per la gestione degli stati (vedi sopra), una funzione Fired così definita: public Evento fired(Evento e) , che definisce come vengono gestite le transizioni. Nota che la segnatura della funzione cambierà appena verrà affrontata la gestione concorrente. La funzione si occuperà di controllare che l'oggetto che la invoca sia il destinatario dell'evento (o uno dei destinatari del broadcast) e in caso dichiarerà un nuovo oggetto evento (da restituire come risultato della funzione). seguirà poi uno switch che gestirà gli eventi rilevanti in base allo stato corrente dell'oggetto. All'interno del singolo caso, corrispondente allo stato, si gestiscono tramite un if-else i vari casi di eventi che è possibile recepire a partire dallo stato corrente. Il caso in cui l'evento appartiene alla classe che realmente triggera il passaggio di stato (la transizione), si controllano eventuali condizioni, si inizializza il nuovo oggetto evento in modo che prenda i parametri necessari rispetto alla transizione specifica, si aggiorna lo stato dell'oggetto a quello nuovo, a seguito della transizione. Alla fine dello switch si restituisce l'evento.
Nota che nello switch ci deve essere un caso di default (default : throw new RuntimeException ("Stato corrente non riconosciuto");).

REALIZZAZIONE DELL'ENVIRONMENT SEQUENZIALE
Una Map (o Dizionario) con chiave il Listener e informazione la coda degli eventi inidirizzati al Listener.
HashMap <Listener, LinkedList<Evento>> codeEventiDeiListener;
Nota che il campo Listener è l'identificatore del Listener, non l'oggetto vero e proprio.

Nella public class Environment ci sarà quindi un campo private HashMap <Listener, LinkedList<Evento>> codeEventiDeiListener; Il costruttore, le funzioni addListener, aggiungiEvento, eseguiEnvironment (che manda in esecuzione il sistema e invoca l'esecuzione di fired() dei Listener, ed è quindi di fatto il fireAll()) .

REALIZZAZIONE DELL'ENVIRONMENT CONCORRENTE
Eventi e stati rimangono invariati. Le transizioni verranno adattate alla concorrenza (saranno thread safe, il fired sarà quindi concorrente nel senso che più oggetti possono chiamarlo concorrentemente. Il fired dovrà quindi essere un task, cioè un pattern funtore). La realizzazione dell'environment sarà completamente concorrente: ogni oggetto eseguirà il suo diagramma stati-transizioni su un thread separato (che si occupa di eseguire il suo fired()). L'environment stesso lavorerà su un thread separato, e sarà dotato di una struttura dati condivisa thread-safe per lo scambio di eventi, accessibile dai thread degli oggetti reattivi oltre che dal thread dell'environment stesso.

La cosa fondamentale è che il metodo fired() degli oggetti reattivi diventa un funtore, un eseguibile.

VEDI SLIDES PRIMA MEZZ'ORA DELL'ULTIMA LEZIONE

Su ogni oggetto reattivo gira un processo che continuamente (while (true)) va sulla struttura dati dell'environment, prende dati torna indietro ecc. questo é realizzato tramite una classe class EsecuzioneListener implements Runnable, quindi tramite il pattern funtore. All'interno verrà definito il metodo run che all'interno di un while(true) effettuerà le seguenti istruzioni: prende il prossimo evento dall'environment e ci chiama sopra il fired(). Prima di chiamare la fired sull'evento controlla che l'evento preso non sia quello di stop, che è un evento che definiremo in seguito e che serve ad arrestare il while(true).

La struttura dati per l'environment sarà un concurrent hash map, a cui si può accedere in maniera three safe. Le code sono code bloccanti, LinkedBlockingQueue (così che l'oggetto si addormenta se non ci sono eventi nella lista. Utile perc non dobbiamo occuparci del caso "magazzino vuoto").

In public final class Environment, oltre al campo private static concurrentHashMap <Listener, LinkedBlockingQueue<Evento>> codeEventiListener = new concurrentHashMap <Listener, LinkedBlockingQueue<Evento>>(); , sono definiti i metodi (tutti public static) addListener(), getInsiemeListener(), aggiungiEvento(Evento e), prossimoEvento(Listener lr). Nota che il costruttore è privato poiché non si possono costruire oggetti Environment.

Accanto a questa struttura c'è bisogno di un gestore dell'environment, e si definisce quindi una public final class EsecuzioneEnvironment, che si occupa di inizializzarla (aggiungendo i listener), di far partire tutti i thread di ogni listener, e di interrompere i thread di tutti i listener.


PIOVE
Ci stanno rubando i colori!
Woof
Cip ciop bastardi
Sono una Cane