# Esempio ragionato di analisi-sviluppo

(Agenzia turistica)
Prima scrittura: ottobre 2006
Ultima Revisione 6 novembre 2007

## 1 Scopo del documento

Lo scopo di questo documento è illustrare l'intero processo di analisi-progettazione-sviluppo in riferimento ad un caso applicativo. Il caso preso in esame è di dimensioni ridotte, ma presenta qualche difficoltà concettuale.

La dimensione del problema è quella tipica di un esercizio di esame. Per questo motivo il documento serve anche come banco di prova per verificare la preparazione. [1]

## 2 Il caso di studio

Un'agenzia turistica avente sede in un luogo vacanziero organizza escursioni. Le escursioni hanno durata giornaliera e (per semplicità) sono di due tipi: gite in barca e gite a cavallo.

Per ogni escursione sono previsti alcuni optional acquistabili dall'eventuale partecipante. I tipi di optional possibili sono tre: pranzo, merenda, visita a un sito. I tipi di optional associati a una data escursione possono differire da caso a caso. Per esempio: la gita in barca del giorno x può prevedere il pranzo e la merenda, la gita in barca di due giorni dopo può prevedere il solo pranzo, l'escursione del giorno y può non prevedere optional.

Ogni singola escursione ha un suo prezzo (che può variare da giorno a giorno e in base al tipo). Il prezzo degli optional è fisso ed è determinato solo dal tipo. Ogni escursione prevede un numero massimo di partecipanti. I prezzi delle escursioni, degli optional, i limite ai partecipanti e tutto ciò che corrisponde alla definizione delle caratteristiche delle escursioni vengono fissati all'atto della definizione del programma stagionale.

Si deve realizzare un sistema in grado di gestire le prenotazioni. A tal fine il sistema deve:

0. Consentire di definire (costruire) il programma prima dell'inizio della stagione (e quindi prima dell'uso del sistema come strumento di gestione delle prenotazioni).
1. Permettere di registrare un partecipante ad una data escursione, consentendo, nel caso siano previsti, la scelta di eventuali optional; calcolare il costo dell'escursione comprensivo degli optional.
2. Cancellare un partecipante già iscritto (si trascuri l'aspetto del calcolo della quota di danaro da restituire).
3. Aggiungere o eliminare un optional relativamente a un dato partecipante e una data escursione; calcolare il nuovo prezzo dell'escursione.

## 3 Una premessa

Non rientra nello scopo di questo documento trattare il problema della *persistenza*.

È più che evidente che un sistema come quello che stiamo per sviluppare ha una qualche utilità solo se i dati introdotti sono resi *persistenti*. Il modo standard per dare persistenza consiste nell'avere una base di dati (relazionale) nella quale (dalla quale) i dati vengono salvati (recuperati) all'occorrenza. Ciò imporrebbe la presenza di un meccanismo di mappatura tra i gli oggetti e le tabelle della base di dati. Si parla di ORM (*Object-Relational Mapping*).

Dunque, il sistema verrà realizzato solo in riferimento al mondo a oggetti. Verrà cioè realizzato un "programma" che richiede che i dati vengano inseriti ex-novo ogni volta che esso viene messo in esecuzione. In sostanza viene costruito il modello *run-time* (o di esecuzione) dell'applicazione a oggetti.

[1] Naturalmente, in sede di esame, non si pretende l'analisi dettagliata di tutti gli aspetti del problema.

# 4 Analisi dei requisiti

## 4.1 Considerazioni preliminari

Il primo aspetto da evidenziare è lo *scopo* del sistema. La corretta identificazione dello scopo è importante in fase di analisi, quando si deve decidere quali sono le entità da modellare e come vanno modellate. Evidentemente la scelta deve cadere su entità che hanno senso rispetto agli obiettivi che il sistema si prefigge.
Nel caso specifico esso è più che manifesto: la gestione delle prenotazioni.

Il primo passo dell'analisi consiste nel fare l'elenco dei requisiti (seguendo lo stile IEEE 830). L'elencazione è la tecnica tradizionale per la specifica dei requisiti. Quando si usa UML la tecnica preferita è quella dei casi d'uso, tuttavia l'elencazione resta sempre un modo del tutto naturale per raccogliere i requisiti. Peraltro, i requisiti così raccolti possono essere incrociati con i casi d'uso, in modo di poter verificare se niente è stato tralasciato.
Nel nostro caso, i requisiti sono pochi e già ben identificati nella specifica del paragrafo 2, per cui non andremo oltre, facendo solo riferimento ai casi d'uso.

## 4.2 Casi d'uso

Prima di tutto si deve identificare gli attori.

Un attore è certamente l'impiegato dell'agenzia (l'Agente) che ha comunque accesso al sistema. Il cliente dell'agenzia (possibile partecipante a una o più escursioni) non è un attore. Lo sarebbe se egli stesso avesse accesso al sistema (per esempio via web), ma nel caso specifico ciò non è previsto e dunque il cliente dell'agenzia non ha nessuna interazione diretta col sistema. Per l'unico attore (l'agente) il cliente è un "dato" da inserire nel sistema.

La specifica del paragrafo 2 identifica già i casi d'uso anche in modo assai dettagliato.[2] Il corrispondente diagramma è in Figura 1.

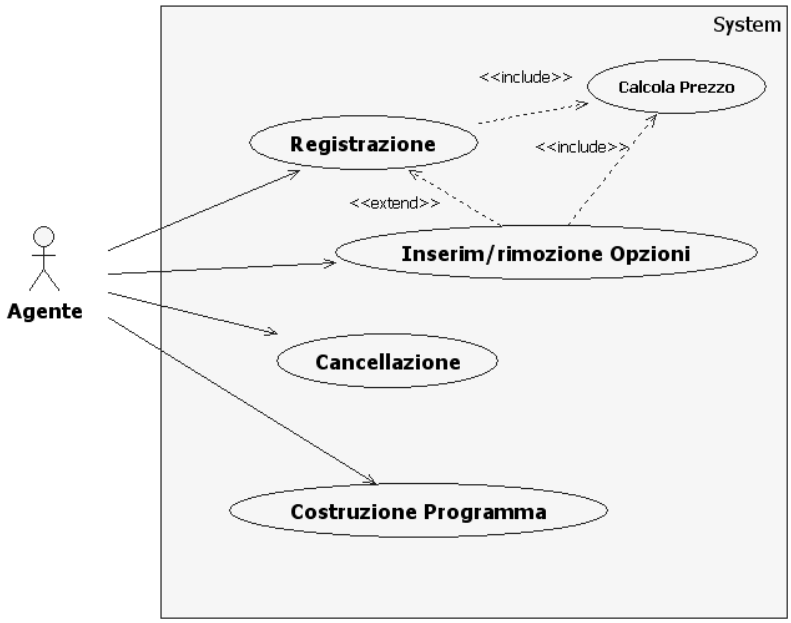


Figura 1: Il diagramma dei casi d'uso.

Si noti che nel diagramma di Figura 1 il caso d'uso "Inserim/rimozione Opzioni" costituisce un'estensione del caso d'uso "Registrazione" (un partecipante può decidere di non volere alcuna opzione). Il caso d'uso "Inserim/rimozione Opzioni" è stato collegato direttamente con l'attore perché è espressamente previsto dalla specifica che l'attore possa inserire o escludere un optional a registrazione già avvenuta.

[2]Non è bisogno di incrociare i requisiti testuali con i casi d'uso, in quanto c'è una corrispondenza 1 a 1.

| CU1: Registrazione |
|---|
| Attore: Agente |
| Precondizione:<br>Il sistema è idle e sullo schermo viene presentato il menu iniziale |
| Sequenza eventi:<br>1. il caso d'uso inizia quando l'attore seleziona "Registrazione"<br>2. il sistema richiede in quale giorno si vuole effettuare l'escursione; l'attore introduce la data<br>3. il sistema mostra quali sono le escursioni prevista alla data indicata e verifica che ci siano posti disponibili, presentando una schermata in cui si elencano le escursioni previste (disabilitando la selezione di quelle già complete), mostrando anche gli optional selezionabili per ciascuna di esse. Sempre sulla stessa schermata vengono previsti i campi relativi al potenziale partecipante.<br>4. l'attore seleziona una escursione tra quelle abilitate, e gli eventuali optional e inserisce i dati relativi a che viene registrato (nome, ecc.)<br>5. il sistema effettua la registrazione, comprensiva degli optional scelti. Il sistema verifica e tiene conto del fatto che il cliente è o no già presente (già registrato per altra escursione)<br>6. il sistema calcola il prezzo |
| Postcondizione:<br>La registrazione è stata effettuata, sullo schermo viene ripresentato il menu iniziale |
| sequenza alternativa:<br>3' nel caso in cui alla data indicata non siano previste escursioni o quelle previste siano tutte occupate il sistema mostra un messaggio conveniente; sulla stessa schermata l'attore può selezionare se ripartire dal passo 2. (richiesta della data dell'escursione) o tornare al menu iniziale. |

Tabella 1: Specifica del caso d'uso "CU1: Registrazione"

Poiché come discuteremo in seguito il diagramma dei casi d'uso poteva prendere anche una forma leggermente diversa, conviene approfondire il diagramma di Figura 1 riportando i casi d'uso in forma testuale. Il passaggio alla forma testuale richiede di entrare nel dettagli su come si svolgono le varie funzionalità. Consideriamo per primo i caso d'uso (CU1): "Registrazione".

Nella specifica di CU1 di Tabella 1, si fa implicita assunzione che il sistema disponga di una interfaccia grafica interattiva. Si tratta di una assunzione ovvia. La specifica di CU1 impone alcune precise modalità di funzionamento. Per esempio, si fa uso di videate che espongono informazioni e campi da riempire. Con riferimento al punto 3, si dice che il sistema presenta le escursioni previste per una certa data e indica quali sono gli optional disponibili, per i quali fornisce (presumibilmente attraverso bottoni) la possibilità di selezionarli. Ovviamente questo modo di operare comporta, ad esempio, che se viene selezionato un dato optional, non c'è necessità di verificare la sua disponibilità, in quanto è stato il sistema a proporlo poiché disponibile. La situazione sarebbe totalmente diversa se l'interfaccia non prevedesse la possibilità di selezionare da una lista proposta, ma si trattasse di inserire un optional senza saper se questo è o no disponibile per l'escursione scelta. In altri termini, l'interfaccia (che di norma è la parte più pesante di un sistema) influisce molto sulla stessa applicazione.

In Tabella 2 viene riportato il caso d'uso"Inserim/rimozione Opzioni".

In Figura 1, i casi d'uso "Registrazione" e "Inserimento/rimozione opzioni" includono "Calcola Prezzo".

| CU2: Inserimento/rimozione opzioni |
| --- |
| Attore: Agente |
| Precondizione:<br>Il sistema è idle e sullo schermo viene presentato il menu iniziale |
| Sequenza eventi:<br>1. il caso d'uso inizia quando l'attore seleziona "Aggiungi opzione a registrazione" (oppure "Cancella opzione a registrazione")<br>2. il sistema richiede il nome del partecipante<br>3. l'attore inserisce il nome<br>4. il sistema cerca tutte le escursioni a cui il partecipante è iscritto, mostrandole assieme agli optional previsti per essa e indicando quali sono già stati selezionati dal partecipante registrato. Il sistema permette di selezionare gli optional da aggiungere/cancellare<br>5. l'attore effettua la selezione<br>6. il sistema aggiorna i sui dati<br>7. il sistema calcola il nuovo prezzo dell'escursione nella nuova configurazione degli optional |
| Postcondizione:<br>La variazione è stata effettuata, sullo schermo viene ripresentato il menu iniziale |
| sequenza alternativa:<br>4' nel caso in cui il sistema non riconosca il nome fornito il sistema presenta un messaggio conveniente; a questo punto l'interfaccia consente queste alternative<br>• il nome è stato battuto erroneamente: l'attore inserisce il nome correttamente e si riparte dal punto 3<br>• il nome è stato battuto correttamente (e il cliente ha una ricevuta dell'avvenuta precedente registrazione): l'attore ha la possibilità di selezionare l'esecuzione di una procedura di emergenza, mirante a verificare se il sistema è integro ed eventualmente a reintegrare i dati del cliente in questione. Si tratta evidentemente di una funzionalità molto complessa che non dovrebbe mai essere richiesta. Infatti la necessità della sua esecuzione può essere dovuta a cattivo uso del sistema da parte dell'attore (p.es. non ha fatto la registrazione) oppure a un malfunzionamento dovuto a un errore nel programma. In questo secondo caso si apre un mondo di problemi... |

Tabella 2: Specifica del caso d'uso "CU2: Inserimento/rimozione opzioni"

Si tratta di un eccesso di pignoleria: è evidente che la "Registrazione" deve prevedere il calcolo del prezzo. Questo fatto viene meglio esplicitato dalla descrizione testuale della "Registrazione" e poteva essere omesso dal diagramma.

In Figura 1 è stato riportato senza altri dettagli il caso d'uso "Cancellazione". Infine è stato previsto anche il caso d'uso "Costruzione Programma" relativo all'inserimento a programma delle escursioni. Di questi casi d'uso non viene data descrizione testuale.

### 4.3 Commento

Si noti che sarebbe stato del tutto corretto immaginare che il caso d'uso "Inserim/rimozione Opzioni" fosse del tutto indipendente da "Registrazione", nel senso che l'attore effettua prima una registrazione e quindi un inserimento/rimozione e che i due casi d'uso sono totalmente a sé stanti.
Come pure non era sbagliato prevedere che il caso d'uso "Inserim/rimozione Opzioni" avvenisse solo come estensione della registrazione (escludendo il suo collegamento con Agente).

La scelta del paragrafo precedente è giustificata col fatto che essa è abbastanza naturale in riferimento all'uso di una moderna interfaccia grafica, ma di per sé non è l'unica possibile. Scelte come queste diventano chiare in fase di progetto-implementazione. Si tratta di aspetti fortemente influenzati dall'ambiente e dalle soluzioni realizzative.

In ogni caso, poiché seguiamo un metodo di sviluppo iterativo (tipo UP), non conviene sforzarsi di definire sin dall'inizio tutti i dettagli di esecuzione di casi d'uso. Questi potranno essere definiti con maggior precisione in un ciclo di iterazione successivo. (Evitare di cadere nel *big design upfront*.)

## 5 Il diagramma delle classi

Si deve ora passare alla costruzione del modello di dominio.

Se si esclude il legame dei partecipanti con gli optional, costruire il diagramma delle classi è cosa agevole. Occorre prevedere la classe "Programma", che aggrega gli oggetti della classe "Escursione", alla quale saranno legati gli oggetti "Partecipante". Da ciò si ottiene il diagramma di Figura 2. `MaxNPartec` è il numero massimo di partecipanti previsti da un'escursione, `costo` è il costo dell'escursione (senza optional), `data` è la Data in cui si tiene l'escursione. Si noti che la classe Escursione è astratta e che da essa derivano le due classi ipotizzate nella specifica del problema.

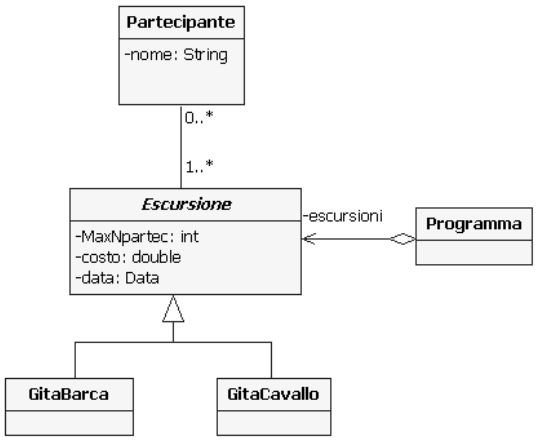


Figura 2: Costruzione del modello. Tutti gli attributi di Escursione sono ereditati dalle sottoclassi.

Il modello di Figura 2 segue i canoni dell'analisi OO. In particolare viene usata l'ereditarietà nel definire i due tipi di escursione. Il modello di Figura 2 impone che quando viene istanziata una escursione questa sia o una "GitaBarca" o una "GitaCavallo".[3]

[3]Ovviamente le istanze (gli oggetti) di queste due classi contengono in sé il tipo di Escursione. In ogni caso, è sempre possibile sincerarsi se una data istanza di Escursione è una GitaBarca o una GitaCavallo attraverso il costrutto `instanceof`.

La domanda che ci si deve porre rispetto al modello di Figura 2 è se sia effettivamente necessario rappresentare esplicitamente, attraverso specifiche classi, i due tipi di escursione o se non sia più appropriato un modello come quello di Figura 3, dove l'escursione è rappresentata da una classe concreta che presenta l'attributo `tipo`, usato allo scopo di distinguere il tipo di escursione.[4]

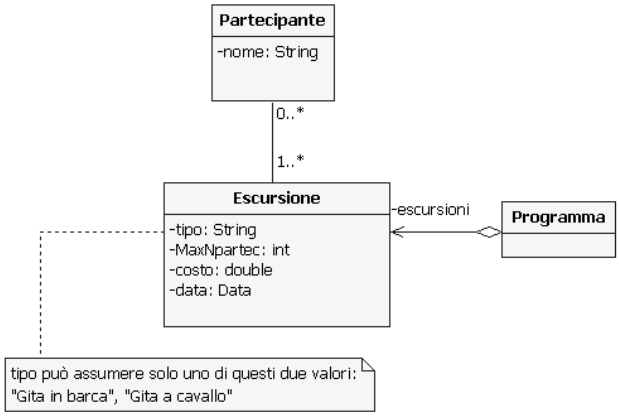


Figura 3: Modello alternativo a quello di Figura 2.

La scelta tra i modelli di Figura 2 e 3 dipende esclusivamente dal ruolo che gli oggetti devono svolgere in base alla specifica.

Il modello di Figura 2 sarebbe il più appropriato se i differenti tipi di escursioni presentassero comportamenti differenti. In altre parole se i due tipi di escursione differissero o per il numero di attributi o per il comportamento (ovvero per il numero di metodi o, a parità di metodi, nelle funzioni da essi svolte).
Nel nostro caso, per come è specificato il problema, le gite in barca e quelle a cavallo sono perfettamente equivalenti, nel senso che il termini "Gita in barca" e "Gita a cavallo" hanno solo una funzione classificatoria. Questo ci dice che non è necessario istanziare gite in barca o gite a cavallo (aumentando così il numero di classi e introducendo il polimorfismo), in quanto è sufficiente istanziare Escursioni, indicandone il tipo. Diverso sarebbe stato il caso se la gita in barca avesse un comportamento diverso dalla gita a cavallo, perché, per esempio, la prima espone il metodo `virare()` mentre la seconda espone `galoppare()` e ambedue le operazioni vengono invocate dal programma. In ultima analisi, si tratta di un problema tipico della programmazione a oggetti: si istanziano gli oggetti che hanno un loro comportamento e/o incapsulano dati. Al contrario le funzioni di classificazione o di identificazione possono essere svolte attraverso un qualche attributo.

Non essendoci differenziazione tra i due tipi di escursione, il modello di Figura 3 è perfettamente equivalente a quello di Figura 2 ed ha il pregio di essere più sintetico e semplice da trattare. Per questo motivo nel seguito faremo riferimento esso.[5]

Una seconda considerazione sul modello riguarda il fatto che nello stesso giorno si possono avere più escursioni dello stesso tipo. Ovviamente esse vengono istanziate come oggetti differenti, ma non hanno niente che le identifichi esplicitamente. In altre parole due gite in barca dello stesso giorno si distinguerebbero, ad esempio, dal numero dei partecipanti o dal costo. Nella pratica è conveniente assegnare un identificatore (un numero) all'escursione. Questo numero può essere assoluto (l'escursione 35), ovvero relativo (l'escursione 2 di tipo Gita in Barca del giorno x). Di solito è più conveniente un identificatore assoluto (anche tenuto conto del fatto che il sistema reale di prenotazioni si appoggerà su una base di dati relazionale). Tenuto conto di ciò modello diventa quello di Figura 4, dove `id` è un numero univoco per ogni escursione.
Lo stesso concetto andrebbe applicato anche ai partecipanti. Per semplicità assumiamo che l'attributo `nome` sia sufficiente a identificare in modo univoco il partecipante.

[4] Col modello di Figura 3 il programma istanzia oggetti della classe Escursione e si sincera del loro tipo leggendo il valore dell'attributo `tipo`. Ovviamente ci sono due soli possibili valori: "Gita in Barca" e "Gita a Cavallo".

[5] Almeno in linea di principio esiste anche una terza via: prevedere l'attributo `tipo` anche nel modello di Figura 2. Questa scelta può essere giustificata col desiderio di: (a) semplificare il trattamento del tipo di escursione (attraverso `tipo`); (b) mantenere aperta la strada all'introduzione di differenti comportamenti tra Gita in barca e Gita a Cavallo.

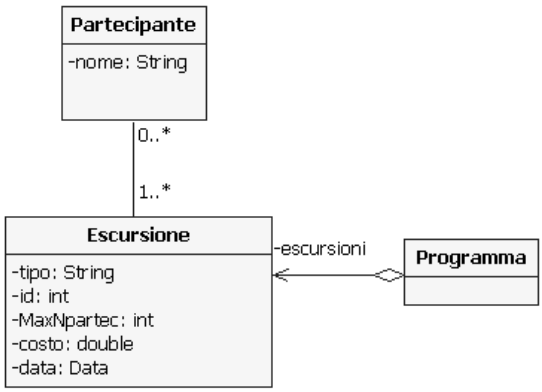


Figura 4: Aggiunta di un identificativo dell'Escursione.

Passiamo ora alla rappresentazione degli optional. Ci sono due aspetti: (a) la classificazione di tipi di optional previsti per una escursione; (b) il legame tra i partecipanti e gli optional scelti.

Per quanto si riferisce al primo aspetto la specifica dice che un'escursione può prevedere fino a un massimo di tre tipi di optional. Anche qui vale un ragionamento analogo –ma non identico– a quello fatto per i tipi di escursione: stiamo parlando di *tipi* di optional non dell'optional in quanto tale. Infatti, lo scopo del sistema è quello di badare alle prenotazioni e quindi ci basta che il sistema riesca a stabilire quali sono i tipi di optional previsti per una datata escursione. Se lo scopo del sistema fosse stato più ampio e, per esempio, prevedesse di pianificare l'acquisto di carne e pesce per preparare i pranzi di una data gita (assumendo che possano esserci svariati menu con ingredienti che appartengono all'uno e all'altro menu), allora sarebbe stato necessario istanziare i singoli pranzi scelti dai partecipanti a una data gita, in modo da poter fare il conteggio delle quantità richieste. Ogni pranzo istanziato terrebbe traccia, in base all'eventuale menù scelto, della quantità di carne richiesta per confezionarlo.

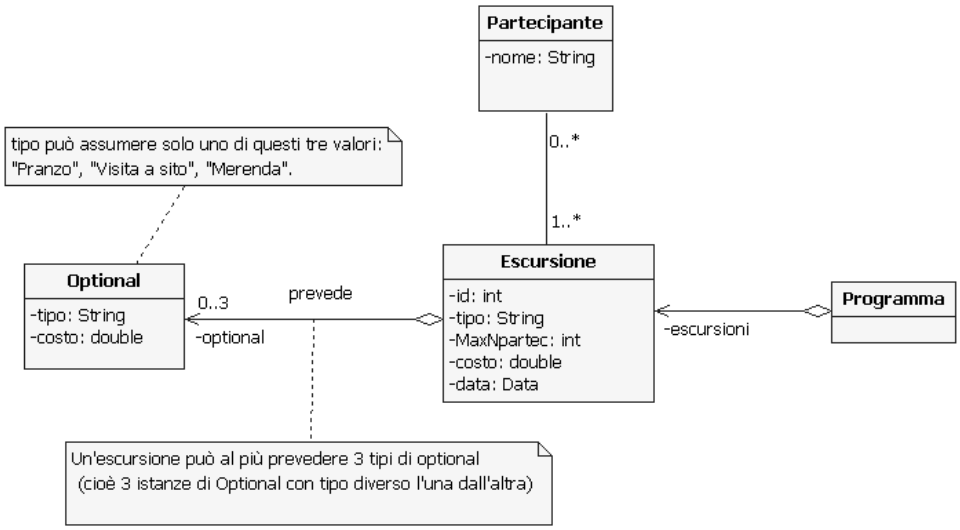


Figura 5: Aggiunta degli optional previsti dall'escursione.

Nel nostro caso basta solo sapere quali tipi di optional una gita prevede. Conseguentemente, nel sistema basteranno solo tre oggetti (tre singleton), aventi la sola funzione di identificare i tre tipi di optional. [6] Per

[6]La scelta di realizzare i tipi di optional come singleton è la più naturale. Ovviamente è sempre possibile istanziare specifici

rimarcare questo fatto denominiamo la classe in questione "`TipoDiOptional`" e stabiliamo che l'attributo `tipo` discrimini i tra i possibili tipi di optional.[7] Il modello diventa quindi quello di Figura 5.

Vale la pena di osservare che se la specifica del sistema avesse imposto l'istanziazione di tanti oggetti optional quanti quanti ne vengono scelti (si veda quanto detto poco sopra), sarebbe stato più appropriato dare alla classe il nome "`Optional`".

Resta ora da modellare le associazioni tra i partecipanti e gli optional. Si tratta di tener traccia di quali tipi di optional abbia scelto un partecipante in riferimento a una data gita. Si faccia caso che non si può stabilire una associazione diretta tra `Partecipante` e `Optional`, infatti il partecipante x può avere scelto l'optional Pranzo per l'escursione e1 e gli optional Merenda e Visita a sito per l'escursione e2. In altre parole si tratta di una relazione ternaria che richiede una classe di associazione. Il problema è che questa classe non può essere `Escursione`, in quanto essa svolge una funzione differente dal rappresentare il legame tra `Partecipante` e `Optional`. Occorre introdurre una nuova classe, con funzione di associazione tra `Partecipante` e `Optional`. Chiameremo `Scelta` questa classe. Essa è relativa (dipende) da una (singola) `Escursione`. Si ha così il diagramma di Figura 6. Nel diagramma non si è usato il simbolo di dipendenza tra `Scelta` e `Escursione`, per il semplice fatto che la dipendenza consiste nel riferimento a `Escursione`.

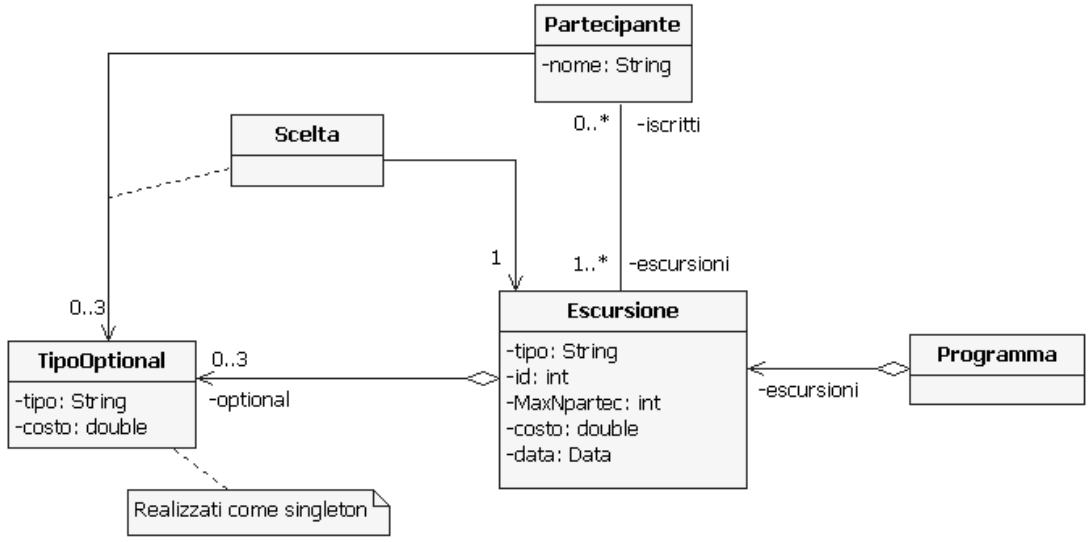


Figura 6: Il modello di dominio nella versione finale.

# 6 Realizzazione dei casi d'uso

Stabiliamo anzitutto che le classi del modello vengono raccolte nel package `modello`. Stabiliamo inoltre di realizzare la logica applicativa attraverso classi distinte da quelle del modello che raggrupperemo nel package `applicazione`. Il diagramma di implementazione è in Figura 7. La figura (impropriamente) mostra anche l'attore. Il colloquio tra l'attore e il resto del sistema sarà mediato presumibilmente da una interfaccia grafica (realizzata attraverso un package che denomineremo GUI). Per il momento non consideriamo l'aspetto dell'interfaccia, assumendo che l'attore sia direttamente collegato alle classi facenti parte di `applicazione`.[8]

La realizzazione dei casi d'uso consiste nel vedere attraverso diagrammi di sequenza (o di comunicazione) come si esplicano i differenti casi d'uso. In particolare, lo scopo dei diagrammi è quello di verificare se il modello regge i casi d'uso previsti. La parte che esegue esamina alcuni casi.

(tipi di) optional per ogni escursione. Ma ciò ha solo l'effetto di aumentare il numero di oggetti istanziati, senza nessuna ragione.

[7] In altre parole `tipo` potrà assumere solo questi tre valori "Pranzo", "Merenda" e "Visita".

[8] Schematicamente, quello di Figura 7 è il pattern MVC a cui sia stata soppressa la vista.

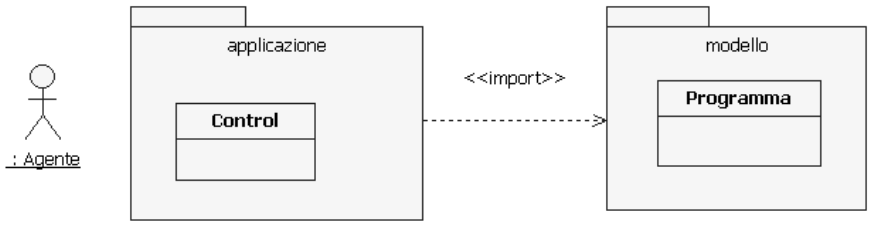


Figura 7: Suddivisione in packages. La classe Programma del modello svolge la funzione di una sorta di interfaccia tra l'applicazione e il resto delle altre classi del modello.

## 6.1 Realizzazione del CU Costruzione del programma

Per comodità cominciamo a esaminare questo caso d'uso, che corrisponde all'espletamento di una funzionalità che deve necessariamente precedere l'uso normale del sistema.[9]

In Figura 8 viene mostrato il diagramma di sequenza corrispondente al caso d'uso in questione. Come

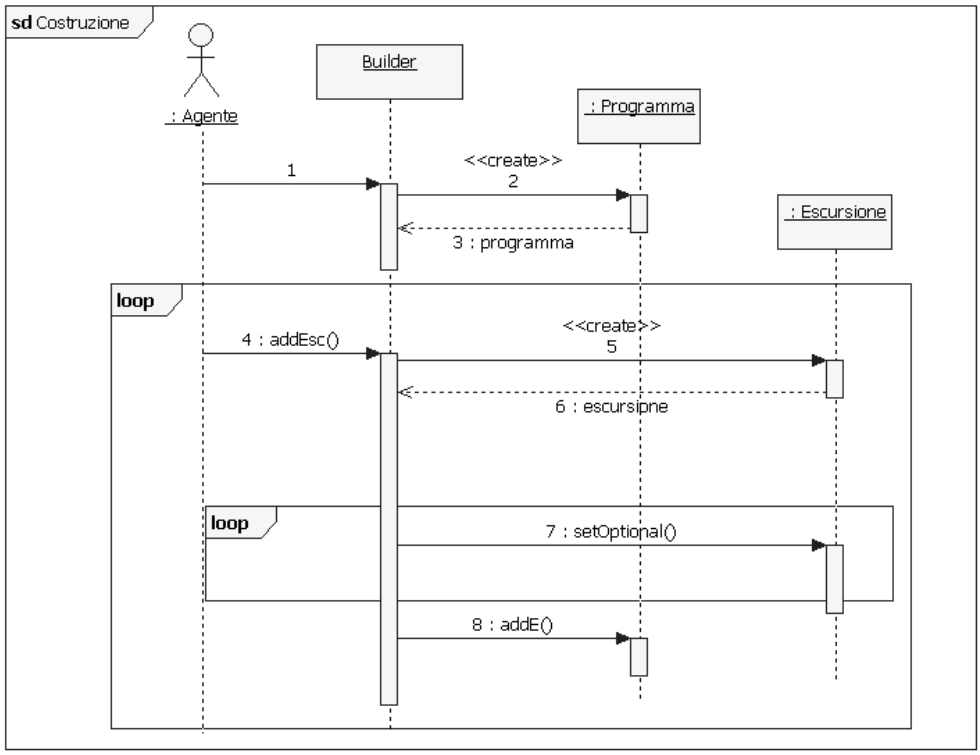


Figura 8: Diagramma di sequenza per la costruzione del Programma. Nello schema non è riportata la creazione degli optional (necessaria prima di aggiungerli alle escursioni).

indicato nella didascalia della figura, lo schema non riporta la creazione degli optional.

Si noti che è stata prevista nella logica applicativa la classe `Builder`, deputata a costruire il programma delle escursioni della nostra agenzia.[10]

[9] Non è buona regola considerare per primo il caso "anomalo" o un caso d'uso di contorno, ma qui stiamo seguendo un percorso didattico e questo caso d'uso, oltre ad essere del tutto a sé stante, è alquanto semplice da realizzare.

[10] Questa classe sta naturalmente nella logica applicativa, ma poteva a buon diritto essere impacchettata nel modello. Ciò

## 6.2 Realizzazione del CU Registrazione

Stabiliamo di prevedere la classe `EscManager` nel package `applicazione`, avente la responsabilità di gestire le escursioni. In Figura 9 viene data la prima parte del diagramma di sequenza per la registrazione di un cliente (il Partecipante).

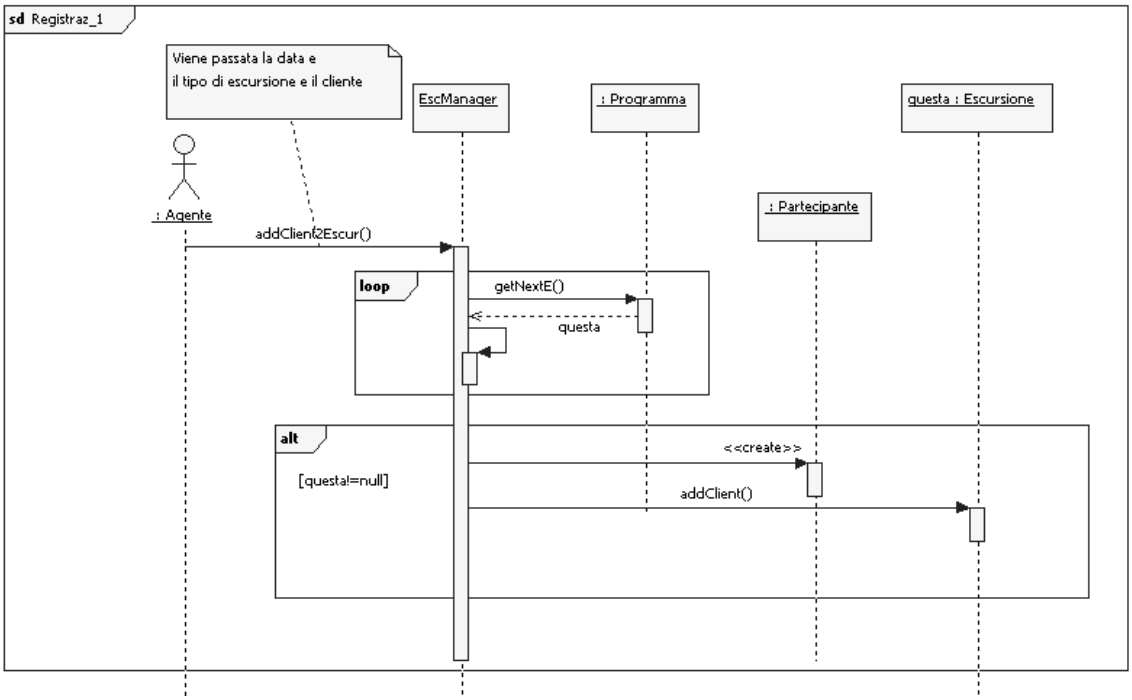


Figura 9: Registrazione di un partecipante. Manca la parte della scelta degli optional (vedere Figura 10). Si ipotizza che il partecipante non sia rappresentato nel sistema; sarà bene effettuare la verifica dell'esistenza del partecipante prima di procedere alla sua creazione!

In Figura 10 viene data la continuazione della sequenza di registrazione. Qui è stata introdotta una ulteriore classe applicativa `ClientManager` avente il compito di gestire i Partecipanti.

Si noti che la separazione in due diagrammi non corrisponde esattamente al modello di Figura 1, dove l'inserimento/rimozione di optional è stata considerata come un'estensione. Nei diagrammi invece i due casi d'uso sono stati trattati come del tutto indipendenti. Il risultato è che se non viene eseguita la fase di inserzione degli optional, tutto avviene come se il partecipante non ne avesse scelto alcuno.

## 6.3 Considerazione

Stendere i diagrammi è cosa piuttosto noiosa. Qualunque sia il tool usato. A mano si fa prima!.

Prima di passare alla progettazione conviene comunque stendere dei diagrammi che mostrino gli aspetti essenziali. I diagrammi sono utili per capire non solo la dinamica del sistema ma anche per determinare le operazioni che si rendono necessarie. Inoltre forniscono anche indicazioni strutturali. Per esempio, il diagramma di Figura 9 indica che il manager delle escursioni (`EscManager`) deve avere un riferimento al `Programma`. Ovviamente sia `EscManager` sia `Programma` (come pure `ClientManager`) saranno istanziati come singleton.

I diagrammi precedenti sono fin troppo dettagliati, in quanto riportano alternative, loop, ecc. Conviene semplificare e seguire il criterio di tracciare il diagramma relativo allo scenario principale (quello che va a buon fine). I casi anomali si trattano a livello di dettaglio. Per esempio il loop del diagramma di Figura 9 poteva essere trasformato in una autochiamata che indichi che si fa la ricerca dell'escursione, mentre l'alternativa che segue può essere eliminata se si fa riferimento al caso normale. Avremmo avuto il diagramma di Figura 11.

avrebbe avuto qualche vantaggio in termini di coesione del codice. Per esempio: la creazione di escursioni o di optional richiede che le corrispondenti classi dichiarino pubblico il loro costruttore se questo viene chiamato da `Builder` fuori del package `modello`; se invece `Builder` fa parte di `modello` i costruttori in questione possono essere dichiarati di package (default), in modo da dare minor visibilità all'esterno.

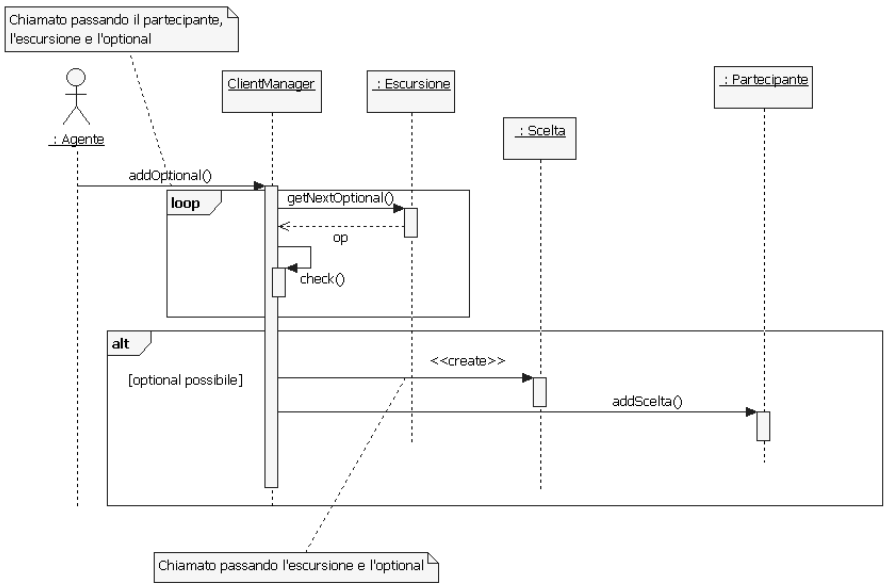


Figura 10: Registrazione di un partecipante. Scelta degli optional (continuazione di Figura 9). Nel tracciare il diagramma si è assunto che la `Scelta` sia realizzata secondo lo schema di Figura 14 (si veda il paragrafo 7.1.

# 7 La progettazione e il passaggio alla realizzazione

L'introduzione delle classi applicative come esterne al modello è una scelta di carattere progettuale, a valle dell'analisi, come pure alcune delle scelte conseguenti discusse qui di seguito. Come detto nella didascalia di Figura 7 la classe `Programma` (o, meglio, l'oggetto che la istanzia) viene a rappresentare l'interfaccia tra il modello e l'esterno, nel senso che dall'esterno si accede agli altri oggetti del dominio applicativo attraverso `Programma`. In sintesi:

- Poiché la nostra agenzia avrà un solo programma per le gite da essa organizzate, è naturale che `Programma` sia un singleton.
- `Programma` diventa un *contenitore* degli oggetti del modello. Questa, assieme alla precedente è una situazione che si presenta frequentemente: un singolo oggetto che contiene e fa da porta di accesso agli altri oggetti del modello.[11]

La stesura dei diagrammi di sequenza è essa pure una attività che può essere riguardata come attività di progettazione, in quanto consente di definire meglio le interfacce (in pratica i ruoli) delle classi di analisi.[12]

I diagrammi di sequenza ci hanno permesso di identificare alcuni metodi delle classi del modello. In Figura 12 viene presentato di nuovo il digramma delle classi con i metodi identificati (più altri metodi di cui è facile prevedere la presenza anche se per ora non si è affrontato l'aspetto relativo). Al diagramma sono state aggiunte le classi applicative e metodi per ora identificati. Il modello viene usato per la generazione automatica del codice (gli stub delle classi); ovviamente in fase di sviluppo ci sarà necessità di aggiungere altri metodi (del resto, come abbiamo più volte rimarcato, si è evitato di entrare nei dettagli minimi difficilmente precisabili a questo punto).

Prima di passare alla realizzazione occorre verificare se il modello e le stesse classi applicative (peraltro ancora abbastanza generiche) non debbano essere meglio precisate e/o riorganizzate. Nel nostro caso c'è da definire come si realizza la classe di associazione `Scelta`. Di questo si parla al paragrafo 7.1.

[11] Nel caso concreto, un'agenzia avrà presumibilmente un diverso programma ogni anno. Pertanto `Programma` deve essere istanziato solo una volta per ciascun anno (non una volta in assoluto). Il costruttore del singleton può essere congegnato in modo conseguente.

[12] Il non aver descritto completamente tutti i casi d'uso e non averne tracciata la realizzazione è conseguenza del fatto che avanziamo per gradi, evitando di cadere nella paralisi da analisi. Potremo, in un secondo, tempo realizzare i casi d'uso mancanti. È questo il modo di procedere nella vera pratica – non solo in un esercizio didattico– anche se ciò può comportare qualche rifattorizzazione del lavoro già svolto.

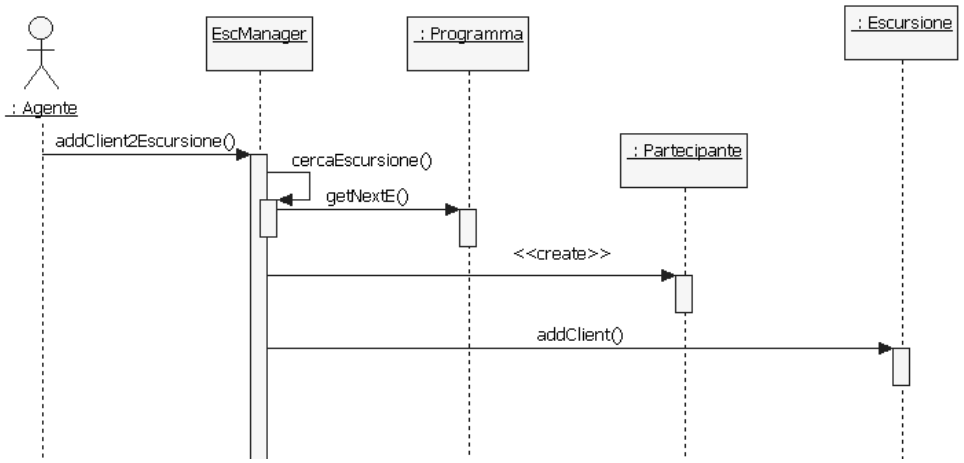


Figura 11: Versione semplificata del diagramma di Figura 9

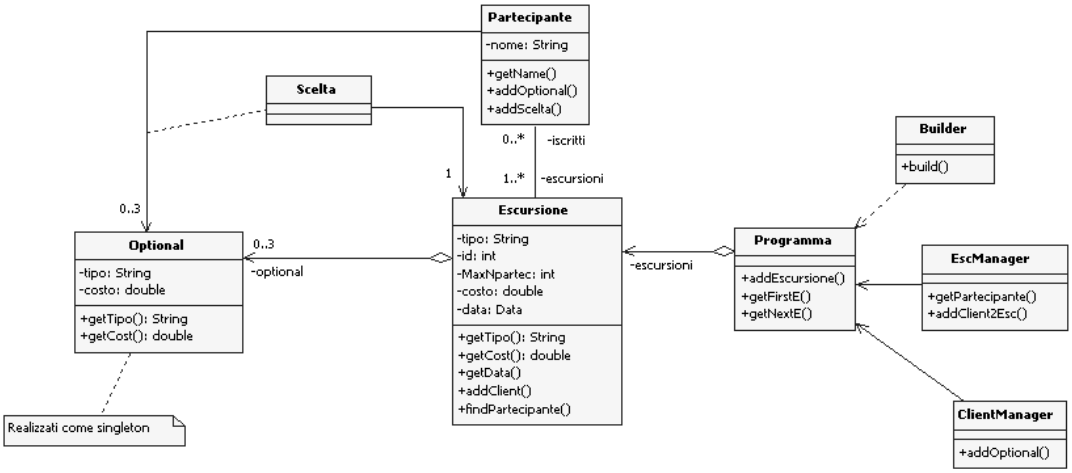


Figura 12: Modello finale con l'aggiunta dei metodi identificati attraverso i diagrammi di sequenza mostrati in precedenza.

E' in questa fase che, si possono applicare i *Design Patterns*: soluzioni predefinite, anche se diverse da caso a caso, a categorie di problemi che si presentano in fase di progettazione. Abbiamo già parlato del pattern singleton.

Conviene che la prima realizzazione venga sviluppata senza l'interfaccia grafica e cioè facendo riferimento al modello di Figura 7. Questo modello prevede che l'attore comunichi direttamente con le classi applicative. Ovviamente in mancanza dell'interfaccia grafica, se il prototipo deve lavorare secondo lo schema di Figura 7 occorre pur sempre sviluppare alcune classi che facciano da interfaccia tramite la console del calcolatore. Si tratterebbe di lavoro che andrebbe completamente sprecato. Niente vieta di sostituire il nostro attore Agente con un programma: il programma Main, avente il compito di simulare l'Agente. Ciò significa che nel Main di

si dovrà prevedere un insieme di chiamate ai metodi delle classi di applicazione di modo da rappresentare un sottoinsieme rappresentativo delle azioni eseguibili dall'Agente.

Con gli ultimi ragionamenti conviene che Main stia in un package a parte (`simulatoreAttore`) fuori da `applicazione`. Il diagramma dei packages diventa quello di Figura 13, dove si noterà la per presenza del package `common`, un package di supporto dove si potranno mettere classi di utilità (come la classe che implementa una `Data`) visibili da tutto il resto.

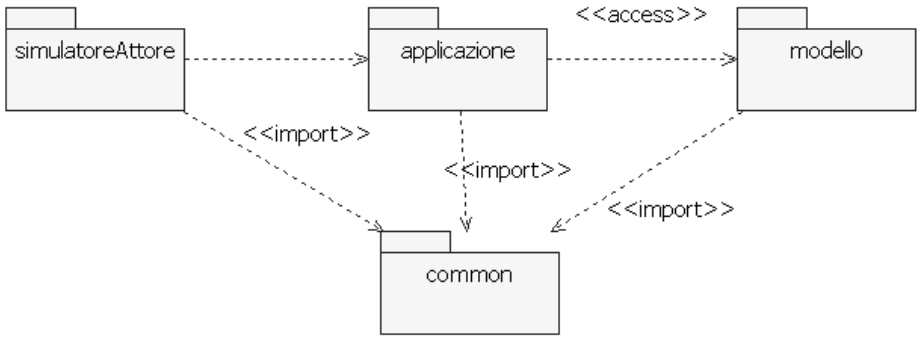


Figura 13: Organizzazione dei package

## 7.1 Come si realizza la classe di associazione

Con riferimento al modello di Figura 6, c'è il problema della reificazione della classe di Scelta. Le Figure 14 e 15 mostrano due soluzioni alternative e i legami che si instaurano tra gli oggetti nei due diversi casi. Nel primo una scelta tiene traccia un singolo optional scelto dal partecipante per una data escursione; nel secondo una scelta tiene traccia di tutti gli optional selezionati dal partecipante per una data escursione.

La seconda soluzione è da preferire in quanto consente in modo naturale di verificare che un partecipante possa scegliere più del numero max di optional. Con la prima soluzione il controllo è più complicato in quanto richiede che si ricerchino tutte le scelte che si riferiscono a una certa escursione, mentre nella seconda soluzione tutte le scelte relative a una data escursione sono raggruppate. In ambedue i casi resta il problema di garantire che per una data escursione un partecipante scelga, per esempio, due volte il pranzo.

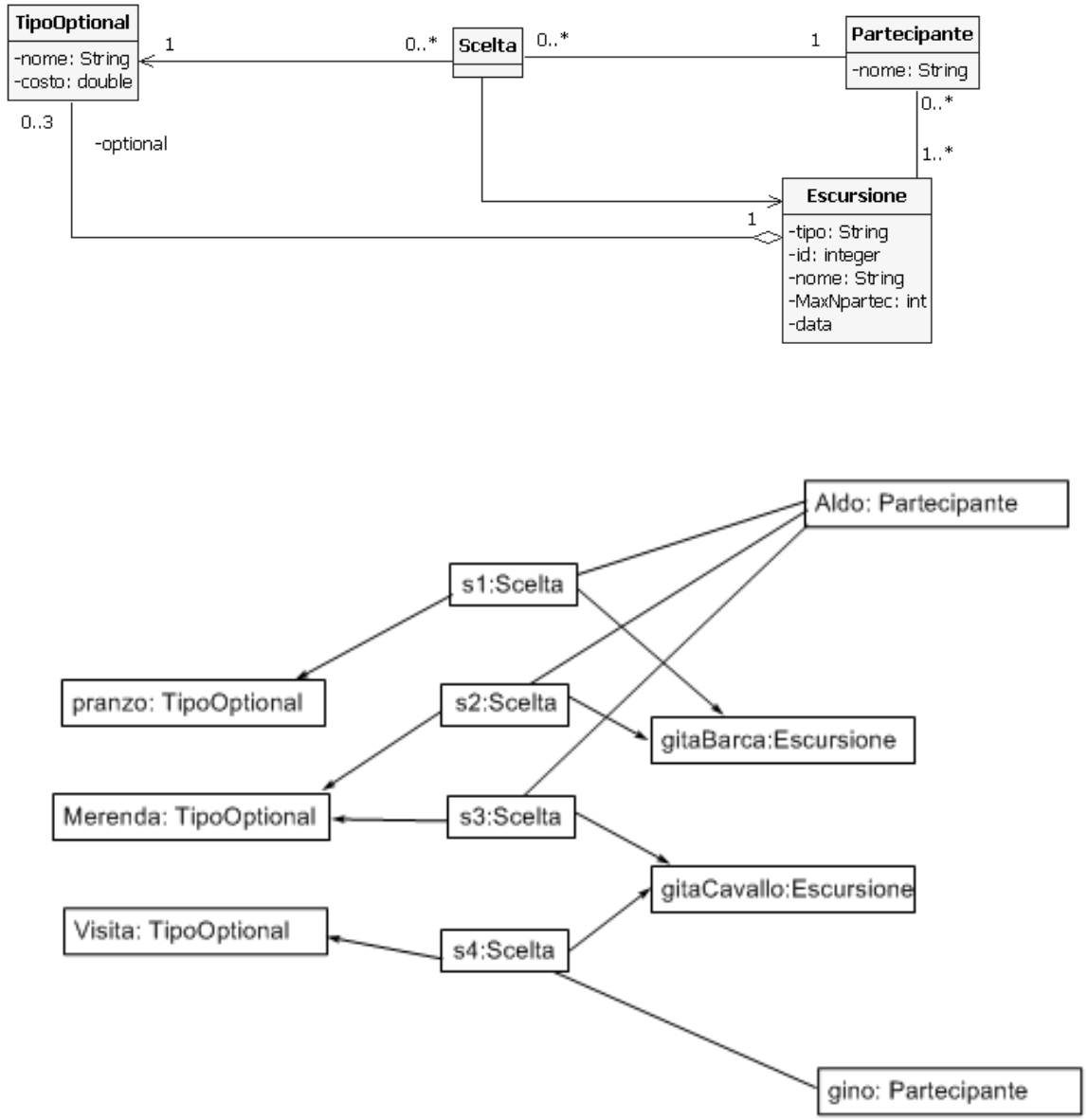


Figura 14: Primo modo di realizzare la classe di associazione Scelta

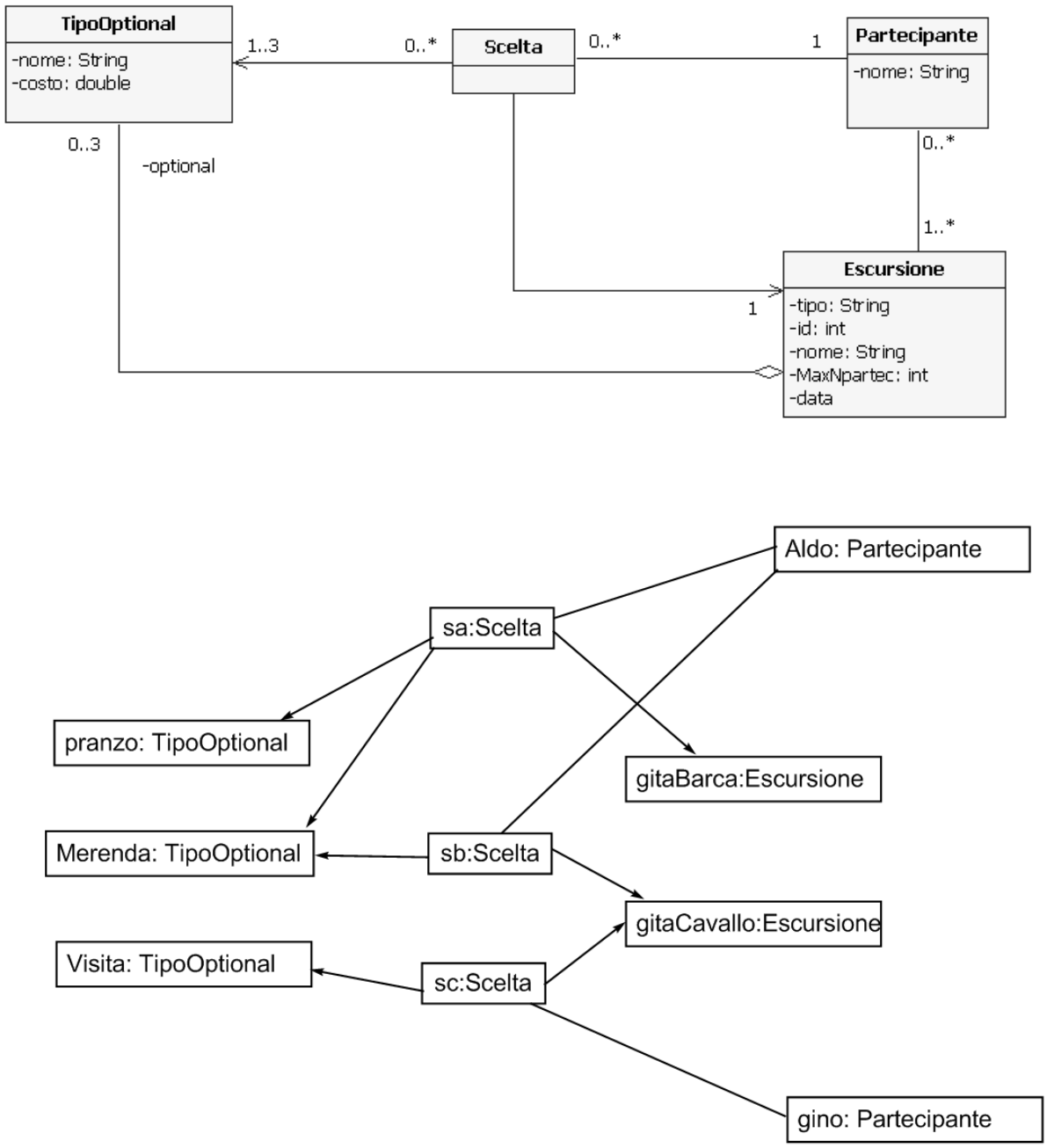


Figura 15: Modo alternativo di realizzare la classe di associazione Scelta

# 8 (Prima) Realizzazione

A titolo di esempio riportiamo alcune classi.

## 8.1 Il Package common

### 8.1.1 La classe `Data`

```
package common;

public class Data {  // non si tiene conto dell'anno
    public int mese;
    public int giorno;

    public Data(int giorno, int mese){
        this.giorno = giorno;
        this.mese = mese;
    }

    public boolean uguale(Data d){
        if ((this.giorno == d.giorno ) & (this.mese == d.mese ) ) return true;
        else return false;
    }

    public int getGiorno(){
        return giorno;
    }

    public int getMese(){
        return mese;
    }
}
```

## 8.2 Il package modello

### 8.2.1 La classe `Programma`

Essa è praticamente tratta in modo diretto dal modello di Figura 12.

```
package modello;
import java.util.*;
//  Generated by StarUML(tm) Java Add-In  (ma non in modo così completo!!!!)

public class Programma {                //  realizzata come Singleton
    private static Programma instance = null;
    private ArrayList<Escursione> escursioni;
    private int ix;                     //posizione del cursore (indice di lettura in escursioni)
                                        //ix è SEMPRE posizionato all'ultima escursione estratta

    private Programma(){                //costruttore (privato per fare il singleton)
        escursioni = new ArrayList<Escursione>();
    }

    public static Programma getInstance(){    //restituisce sempre lo stesso oggetto
        if (instance == null)
            instance = new Programma();
        return instance;
    }

    public void addE(Escursione e){
        escursioni.add(e);
    }

    public Escursione getFirstE() {
        if (escursioni.isEmpty()) return null;
        else{
            ix= 0;
            return escursioni.get(ix);
        }
    }

    public Escursione getNextE(){
        ix++;
        if (ix< escursioni.size())
            return escursioni.get(ix);
        else return null;
    }
}
```

### 8.2.2 La classe `Scelta`

La versione riportata è quella relativa alla soluzione di Figura 14.

```
package modello;

public class Scelta {
    private TipoOptional o;
    private Escursione e;

    public Scelta(TipoOptional o, Escursione e){
        this.o = o;
        this.e = e;
    }

    public TipoOptional getOptional(){      //Ipotesi: un oggetto Scelta per
        return o;                           //ciascuna scelta
    }
    public Escursione getEscursione(){
        return e;
    }

}
```

## 8.3 La classe `Escursione`

Attenzione: non viene dato un id ad ogni gita costruita. Il lettore è invitato ad apportare le dovute modifiche in modo che ogni Escursione abbia un suo identificatore univoco.

```
package model;
import java.util.*;
import common.Data;

public  class Escursione {
    protected String tipo;                      //GitaBarca, GitaCavallo, ..
    protected int id;
    protected double costo;
    protected int MaxNpartec;                    //Numero massimo di partecipanti consentito
    protected Data data;
    protected ArrayList<TipoOptional> optional;     //Optional possibili
    protected int ix;                                       //Indice nella lista di optional
    protected Partecipante[] iscritti;          //partecipanti iscritti
    protected int nIscritti = 0;                //numero partecipanti correntemente iscritti
                                                //si poteva derivare esaminando Partecipante[]
    public Escursione(Data d, String t){
        data = d;
        tipo = t;
        if (tipo == "Gita in barca"){
            MaxNpartec = 6;
            costo = 10.;
        }
        else if (tipo == "Gita a cavallo"){
            MaxNpartec = 4;
            costo = 40.;
        }
        else {
            System.out.println("\nTipo Escursione incognito." + tipo);
```

```
        }
        optional = new ArrayList<TipoOptional>();
        iscritti = new Partecipante[MaxNpartec];
    }


    public double getCost(){
        return costo;
    }


    public String getTipo(){
        return tipo;
    }

    public Data getData(){
        return data;
    }

    public int getIscritti(){
        return nIscritti;
    }

    public void setOptional(TipoOptional  t){
        optional.add(t);
    }


    public TipoOptional getFirstOptional() {
        if (optional.isEmpty()) return null;
        else{
            ix= 0;                          //resta posizionato al primo optional
            return optional.get(ix);
        }
    }

    public TipoOptional getNextOptional(){
        ix++;                               // resta posizionato all'optional estratto
        if (ix< optional.size())
            return optional.get(ix);
        else return null;
    }

    public void addCliente(Partecipante c){
        if (nIscritti < MaxNpartec){
            iscritti[nIscritti] = c;
            nIscritti++;
        }
        else {System.out.println("Escursione" + tipo + " del " +
                data.giorno + "/" + data.mese + " completa. " + c.getName()+ " non è stato iscritto")
        }
    }
/*
 * Ci sono anche questi metodi
 *   public Partecipante findPartecipante(String s);    //per cercare il partecipante di nome s
 *   public void stampaIscr();                           //se si vuole fare la stampa di tutti gli
 *                                                      // iscritti a una data escursione
```

```
 */
}
```

### 8.3.1 la classe `Partecipante`

La versione riportata è relativa alla soluzione di Figura 14.

```
package model; import java.util.*;

public class Partecipante {
    private String nome;
    private ArrayList<Escursione> escursioni;
    private ArrayList<Scelta> scelte;
    private int ix;                  //indice in scelte

    public Partecipante(String nome){
        this.nome = nome;
        scelte = new ArrayList<Scelta>();
        escursioni = new ArrayList<Escursione>();
    }

    public void addEscursione(Escursione e){
        escursioni.add(e);
    }
    public void addScelta(Scelta s){    //ATTENZIONE: non viene fatto il controllo sul
        scelte.add(s);                  // numero massimo di scelte  !!!!!
    }
    public String getName(){
        return nome;
    }

    public Scelta getFirstScelta(){        //restituisce la prima scelta in lista
        if (scelte.isEmpty()) return null;
        else {
            ix = 0;
            return scelte.get(ix);
        }
    }
    public Scelta getNextScelta(){
        ix++;
        if (ix < scelte.size())
            return scelte.get(ix);
        else return null;
    }

    public Escursione getFirstEscursione(){        //restituisce la prima escursione in lista
        if (escursioni.isEmpty()) return null;
        else {
            ix = 0;
            return escursioni.get(ix);
        }
    }

    public Escursione getNextEscursione(){
        ix++;
        if (ix < escursioni.size())
            return escursioni.get(ix);
        else return null;
```

```
    }
}
```

### 8.3.2 La classe TipoOptional

Abbiamo dato il nome TipoOptional, anziché Optional, per evidenziare che ha funzione di identificazione di un tipo. La classe realizza tre versioni di singleton: il pranzo, la merenda e la visita. Si faccia caso a come si ottengono. Si fa l'ipotesi semplificativa, come nel caso dell'escursione, che il costo dipenda dal tipo.

```
package modello;

public class TipoOptional { //  realizzata il Singleton in tre versioni in base al tipo
    String tipo;
    double costo;

private static TipoOptional pranzo = null;
private static TipoOptional merenda = null;
private static TipoOptional visita= null;

    private TipoOptional(String t){
        tipo = t;
        if (t== "Pranzo") costo = 5.;
        else if (t== "Merenda") costo = 2.;
        else if (t== "Visita") costo = 1.;
        else System.out.println("Impossibile: le chiamate sono sotto!!");
    }

    public static TipoOptional getIstancePranzo(){
        if(pranzo == null)
            pranzo = new TipoOptional("Pranzo");
        return pranzo;
    }

    public static TipoOptional getIstanceMerenda(){
        if(merenda == null)
            merenda = new TipoOptional("Merenda");
        return merenda;
    }

    public static TipoOptional getIstanceVisita(){
        if(visita == null)
            visita = new TipoOptional("Visita");
        return visita;
    }

    public String getTipo(){
        return tipo;
    }

    public double getCost(){
        return costo;
    }
}
```

## 8.4 Il package simulatoreAttore

### 8.4.1 La classe `Main`

Riportiamo solo un piccolo tratto.

```
package simulatoreAttore;
import applicazione.ClientManager;
import applicazione.EscManager;
import applicazione.Builder;
import model.*;
import common.*;

public class Main {
// Main simula l'Agente e fornisce l'interfaccia tra Agente e
// classi di applicazione

    public static void main(String[] args) {

        Builder b = new Builder();
        Programma p = b.buildProgram();       //costruisce il Programma dell'agenzia
                                      //buildProgram() simula a sua volta l'inserimento a
                                      //programma di escursioni da parte dell'attore

        ClientManager cM = new ClientManager(p);
        EscManager eM = new EscManager(p);

        System.out.println("================================================" +
              "\nLista escursioni dopo inizializzazione");
        eM.stampaListaEscursioni();

        Escursione e = eM.getEscursione(new Data(17,11), "Gita a Cavallo");
        if (e== null) System.out.println("L'escursione Gita a Cavallo del 17/11 non esiste");

        e = eM.getEscursione(new Data(5,7), "Gita in Barca");
        Partecipante c = e.findPartecipante("Bruno" );
        if (c==null) System.out.println("Il partecipante Bruno non esiste per questa escursione" +
                "\n====================================");

        \\ altra roba ::::::::::::
}
```

Il lettore è invitato a esaminare il testo del `Main` allegato cercando di capire quali azioni svolge.

## 8.5 Passi successivi

Per completare il sistema occorrerebbe analizzare i rimanenti casi d'uso, tracciare i relativi diagrammi di sequenza (ma in un caso semplice come questo, se ne può fare a meno, essendo intuibili le interazioni tra gli oggetti) e passare alla scrittura del codice corrispondente.

Occorrerebbe anche progettare un'interfaccia grafica per le effettive interazioni con l'attore, eliminando il Main.

Qui di seguito mostriamo solo un aspetto relativo al calcolo dei costi. Possiamo stabilire che la classe `ClientManager` prevede il metodo `getTotalCost()` che restituisce il costo totale.

Se la soluzione data alla classe di associazione è quella di Figura 14 (primo modo), il metodo assume questa forma:

```
public double getTotalCost(Partecipante c){
        Scelta s;
        Escursione e;
        double costo = 0;

// calcolo del costo delle sole escursioni
        e = c.getFirstE();
        while (e != null){
            costo = costo+ e.getCost();
            e = c.getNextE();
        }
// aggiunta del costo delle scelte
        s = c.getFirstScelta();
        while (s != null){
            costo = costo + s.getOptional().getCost();
            s = c.getNextScelta();
        }
        return costo;
    }
```

E' facile convincersi che la soluzione di Figura 15 è preferibile. Infatti essa permette agevolmente di calcolare il costo di una data escursione per un dato partecipante, in quanto ad una scelta è associata una sola escursione (il loop sarebbe solo sugli optional). Il costo totale per partecipante sarebbe la somma dei costi in cui essi incorrono nelle singole escursioni.

## 8.6 Ulteriori commenti

Si possono fare altre osservazioni.

- Il modello di dominio 12 impone che la ricerca di un dato partecipante passi attraverso le escursioni (`Programma` → `Escursione` → `Partecipante`). Poiché è presumibile che buona parte delle ricerche si effettui in base al partecipante, il sistema sarebbe alquanto inefficiente. Per rimediare si può aggiungere l'associazione (monodirezionale) tra `Programma` e `Partecipante`.

  Meglio ancora se si introduce una nuova classe `Agenzia`, la quale fa da contenitore a tutto il modello e alla quale sono associati direttamente i partecipanti, come in Figura 16.

  Più in generale, questo ci porta a considerare il problema della ricerca di oggetti entro strutture dati (collezioni, vettori, ecc) e della loro organizzazione. Nello sviluppare l'esempio abbiamo aggiunto gli oggetti in fondo a quelli già presenti nelle strutture usate. Anche questo fatto può portare a situazioni di inefficienza. Una migliore organizzazione presume strutture ordinate e metodi ricerca un po' meno grezzi (p.e., hashing).

- Con riferimento allo schema di Figura 16, si osservi che `Agenzia` è l'interfaccia del modello per le applicazioni (e.g., `EscManager`, `Clientmanager`). E' bene che le applicazioni siano il più possibile disaccoppiate rispetto al modello, in modo che le prime possano essere svolte senza prevedere collegamenti diretti con gli oggetti del modello, con l'esclusione del loro contenitore. In tal modo, il modello può anche essere modificato all'occorrenza senza che ciò comporti variazioni nelle applicazioni, che non siano quelle legate alla navigazione (che peraltro è attuata dal modello stesso). Mancando nelle applicazioni i riferimenti diretti agli oggetti interni del modello, la ristrutturazione di quest'ultimo non ha effetto.

  Il basso accoppiamento favorisce la modularità, il riuso, lo sviluppo separato dei componenti, la separazione dei concetti, ecc. Il minimo accoppiamento tra componenti è un obiettivo da perseguire sempre.

  Il modello di Figura 16 è migliore di quello delle figure precedenti perché permette l'accesso immediato ai partecipanti, che sono stati ribattezzati clienti, in quanto un cliente può esistere nel sistema anche se non è iscritto a nessuna escursione (per esempio: una persona si è iscritta e poi cancellata, ma il sistema tiene il suo nome per ragioni commerciali - pubblicità, ecc.)

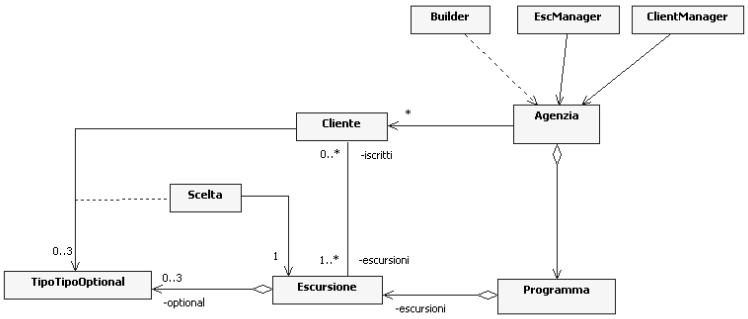


Figura 16: Estensione del modello con la classe `Agenzia` che fa da contenitore a tutti gli oggetti del sistema.

- Nel `Main` dopo che è stato costruito il programma stagionale, vengono istanziati il manager delle escursioni e quello dei clienti (i partecipanti), passando il riferimento al programma. I due manager si memorizzano il riferimento al programma.

  Abbiamo osservato che il sistema potrebbe richiedere anche la gestione dei programmi degli anni passati, ad esempio per effettuare ricerche a fini statistici o commerciali. In questo caso si anno queste alternative:

  - si passa ai metodi del manager il programma come parametro (equivale a istanziare un solo manager di un dato tipo di oggetti);
  - si istanziano tanti manager quanti sono gli anni (sconsigliabile);
  - si istanzia il manager dovuto (con lo stesso nome) ogni volta che serve (preferibile).