PROGETTI

36 **Come ogni applicazione software anche il sistema ATC (*Advanced Trip Computer*) non si costruisce dal nulla:**

- **per la gestione della porta seriale è possibile riutilizzare le funzioni in linguaggio C presentate nell'ultimo capitolo del primo volume di cui si riportano i prototipi validi sia per il sistema operativo Linux sia per l'ambiente Windows:**

```
int COM_open    (unsigned char port, unsigned char mode,
                 unsigned short speed, char parity,
                 unsigned char bits, unsigned char stop,
                 unsigned char flow);
int COM_write   (int com, char buffer[], int N);
int COM_read    (int com, char *buffer, int N);
void COM_close  (int com);
```

- **per l'interpretazione delle stringhe NMEA[1] generate dal dispositivo GPS è possibile adattare il codice già utilizzato nell'ultimo capitolo del primo volume;**
- **per una corretta gestione della *password* (che dovrà essere salvata in un file potenzialmente leggibile da chiunque) è necessario ricorrere alle tecniche di crittografia simmetrica presentate nel capitolo relativo alla sicurezza del primo volume. A questo scopo sarà necessario individuare una libreria di funzioni crittografiche per il linguaggio C/C++;**
- **le coordinate geografiche fornite dai dispositivi GPS (latitudine e longitudine) non sono adatte per il calcolo delle distanze tra punti sulla superficie terrestre. Una rappresentazione cartesiana delle posizioni geografiche come le coordinate UTM[2] è indispensabile, così come è necessario il codice di conversione da coordinate geografiche a coordinate cartesiane.**

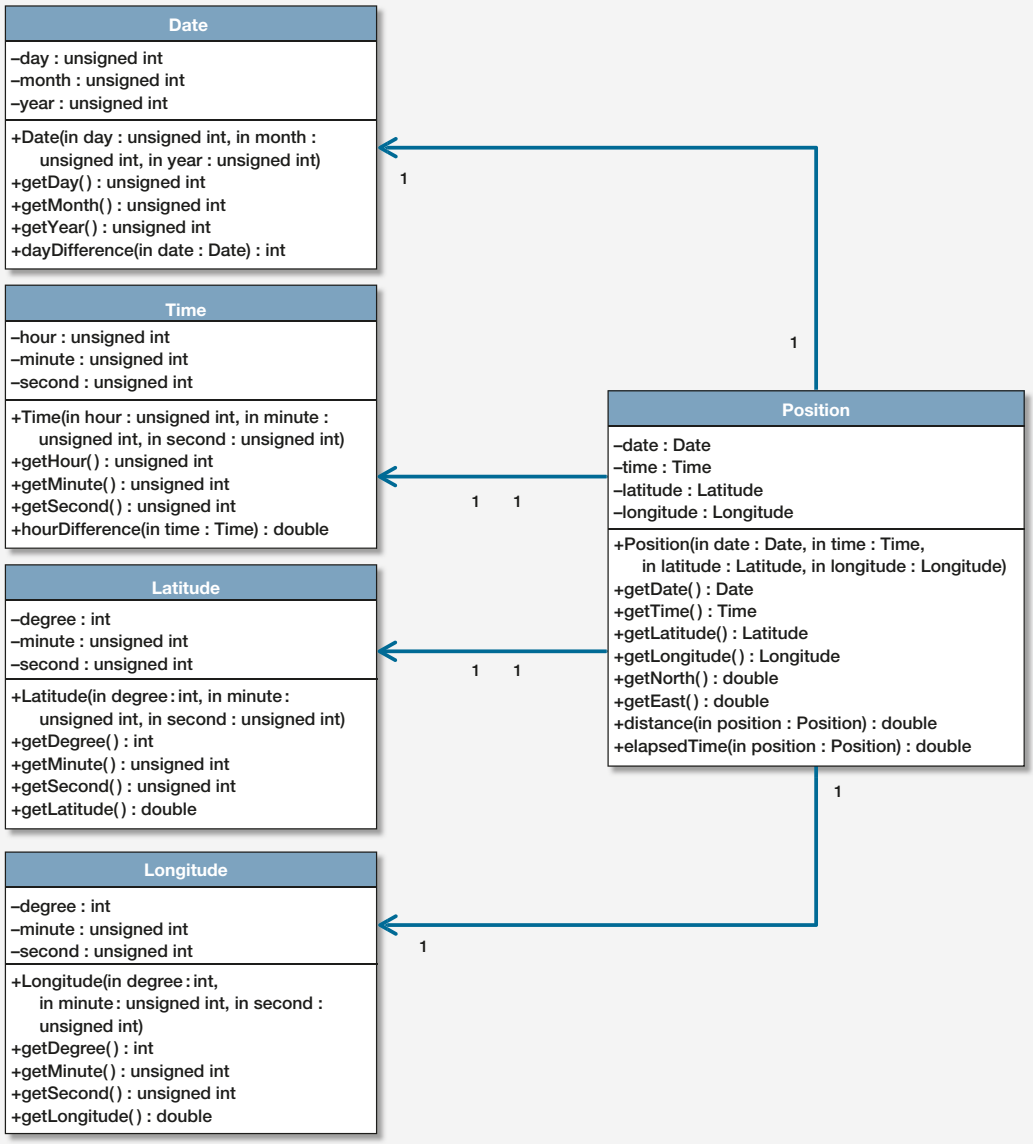
Una ricerca in rete consente di trovare la seguente funzione C per la conversione da coordinate geografiche (latitudine e longitudine) espresse in gradi decimali in coordinate UTM:

```
#define PI                          3.141592653589793

#define WGS84_E2                    0.006694379990197
#define WGS84_E4                    WGS84_E2*WGS84_E2
#define WGS84_E6                    WGS84_E4*WGS84_E2
#define WGS84_SEMI_MAJOR_AXIS       6378137.0
#define WGS84_SEMI_MINOR_AXIS       6356752.314245
#define UTM_LONGITUDE_OF_ORIGIN     3.0/180.0*PI
#define UTM_LATITUDE_OF_ORIGIN      0.0
#define UTM_FALSE_EASTING           500000.0
#define UTM_FALSE_NORTHING_N        0.0
#define UTM_FALSE_NORTHING_S        10000000.0
#define UTM_SCALE_FACTOR            0.9996

double m_calc(double latitude)
{
```

1. In particolare la stringa in formato RMC comprende tutte le informazioni di cui necessita il programma (latitudine, longitudine, data, ora, direzione rispetto al Nord, velocità).
2. Le coordinate UTM (acronimo del sistema *Universal Transverse of Mercator*) rappresentano ciascuna posizione sulla superficie terrestre (escluse le aree polari) mediante valori relativi a due assi cartesiani (l'asse Sud-Nord e l'asse Ovest-Est) espressi in metri. Le coordinate sono in realtà riferite a varie zone in cui è suddivisa la superficie terrestre.

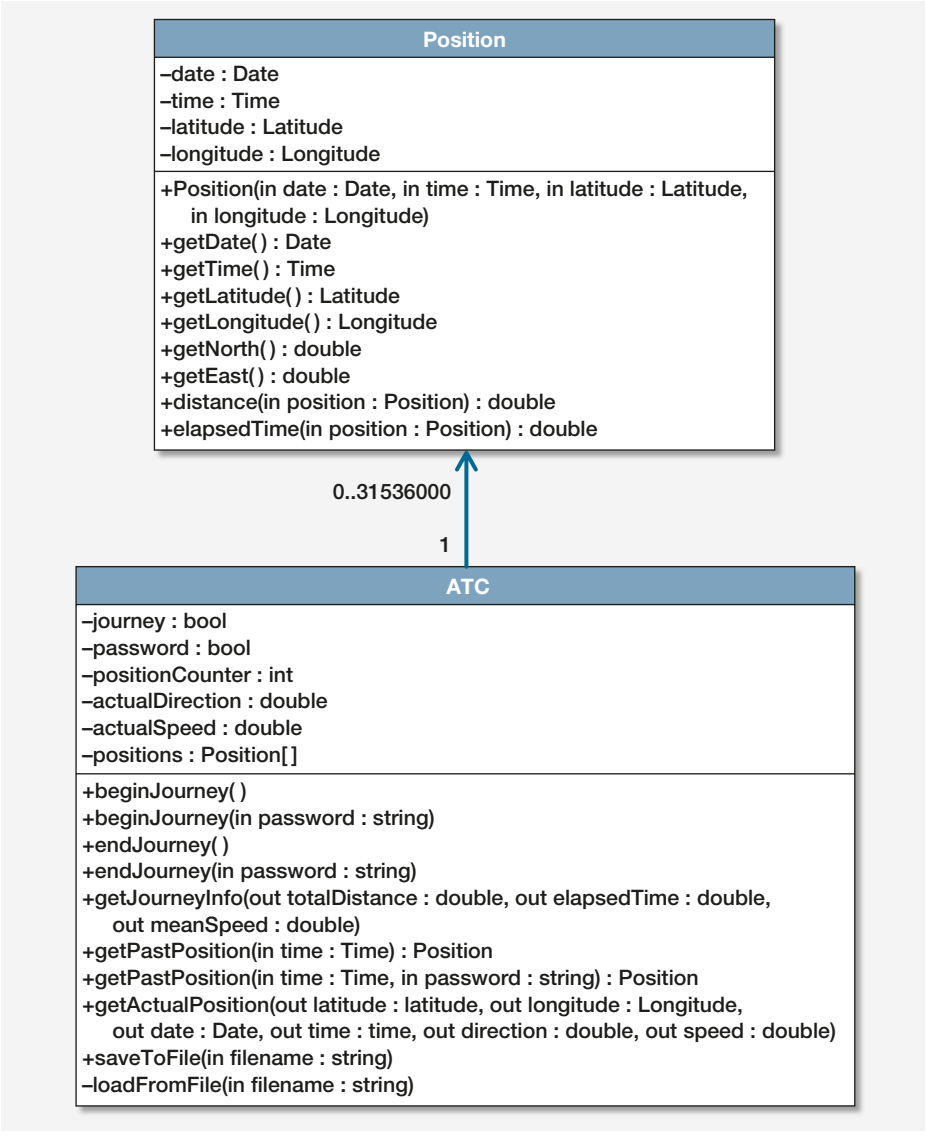
```
  return (1.0 - WGS84_E2/4.0 -
          3.0*WGS84_E4/64.0 -
          5.0*WGS84_E6/256.0) * latitude -
          (3.0*WGS84_E2/8.0 +
          3.0*WGS84_E4/32.0 +
          45.0*WGS84_E6/1024.0) * sin(2.0*latitude) +
          (15.0*WGS84_E4/256.0 +
          45.0*WGS84_E6/1024.0) * sin(4.0*latitude) -
          (35.0*WGS84_E6/3072.0) * sin(6.0*latitude);
}

// INPUT: posizione in latitudine/longitudine (WGS84)
// OUTPUT: posizione in UTM Est/Nord (metri)
void GPS2UTM(double latitude, double longitude,
             double* easting, double* northing)
{
  int int_zone;
  double M, M_origin, A, A2, e2_prim, C, T, v;

  int_zone = (int)(longitude/6.0);
  if (longitude < 0)
    int_zone--;
  longitude -= (double)(int_zone)*6.0;
  longitude *= PI/180.0;
  latitude *= PI/180.0;
  M = WGS84_SEMI_MAJOR_AXIS*m_calc(latitude);
  M_origin = WGS84_SEMI_MAJOR_AXIS *
             m_calc(UTM_LATITUDE_OF_ORIGIN);
  A = (longitude - UTM_LONGITUDE_OF_ORIGIN) * cos(latitude);
  A2 = A*A;
  e2_prim = WGS84_E2/(1.0 - WGS84_E2);
  C = e2_prim*pow(cos(latitude),2.0);
  T = tan(latitude);
  T *= T;
  v = WGS84_SEMI_MAJOR_AXIS /
      sqrt(1.0 - WGS84_E2 * pow(sin(latitude),2.0));
  *northing = UTM_SCALE_FACTOR*(M - M_origin + v*tan(latitude) *
              (A2/2.0 + (5.0 - T + 9.0*C + 4.0*C*C)*
              A2*A2/24.0 + (61.0 - 58.0*T + T*T + 600.0*C -
              330.0*e2_prim)*A2*A2*A2/720.0));
  if (latitude < 0)
    *northing += UTM_FALSE_NORTHING_S;
    *easting = UTM_FALSE_EASTING + UTM_SCALE_FACTOR*v *
               (A + (1.0 - T + C)*A2*A/6.0 +
               (5.0 - 18.0*T + T*T + 72.0*C - 58.0*e2_prim) *
               A2*A2*A/120.0);
  return;
}
```

Una delle funzionalità fondamentali del software ATC consiste nella memorizzazione di una sequenza di posizioni geografiche associate all'istante temporale in cui sono state rilevate. Il seguente diagramma UML rappresenta le classi che consentono di rappresentare adeguatamente una singola posizione geografica nel tempo.



L'applicazione ATC non deve in nessun caso modificare le posizioni geografiche registrate dal dispositivo GPS; di conseguenza le classi che rappresentano la data, l'ora, la latitudine e la longitudine consentono di impostare i valori degli attributi esclusivamente una sola volta mediante il costruttore, ma non espongono metodi *set* per la loro successiva modifica.
Latitudine e longitudine sono valori angolari «con segno»: convenzionalmente la latitudine Nord è positiva, mentre la latitudine Sud è negativa; la longitudine Est è positiva e la longitudine Ovest è negativa. Per questo motivo i soli attributi *degree* sono valori interi con segno a differenza degli attributi *minute* e *second*. Si noti che le classi *Latitude* e *Longitude* hanno metodi specifici per recuperare il valore angolare in forma decimale.

La classe *Position* che rappresenta una posizione geografica espone i metodi per ottenere le coordinate cartesiane UTM (*getNorth* e *getEast*) e per calcolare la distanza in metri tra due posizioni (*distance*), oltre che la differenza in ore tra i tempi di registrazione tra due diverse posizioni.
La classe principale dell'applicazione ATC deve esporre i metodi necessari per consentire al codice del programma di implementare i requisiti funzionali specificati nel diagramma UML dei casi d'uso. Il seguente diagramma UML delle classi definisce la classe *ATC* e la sua relazione con la classe *Position*.



La classe *ATC* ha come attributo un vettore di oggetti di classe *Position*: la dimensione del vettore deve essere tale da consentire di memorizzare una posizione geografica al secondo per al massimo un anno, cioè 3600 secondi × 24 ore × 365 giorni = 31 536 000 posizioni. L'attributo *positionCounter* ha il solo scopo di mantenere il conteggio del numero di posizioni geografiche effettivamente registrate nel vettore *positions* e coincide con l'indice dell'elemento del vettore in cui memorizzare la prossima posizione.
Gli attributi *journey* e *password* di tipo `bool` hanno lo scopo di memorizzare lo stato globale interno di un oggetto di classe *ATC*: *journey* – che il costruttore inizializzerà al valore `false` – assume valore `true` se è già stato invocato il metodo *beginJourney* e non è stato ancora invocato il metodo *endJourney*; *password* assume il valore

`true` solo se la versione del metodo *beginJourney* invocata è quella che consente di specificare una *password*. In questo caso solo le versioni con *password* dei metodi *getPastPosition* ed *endJourney* avranno effetto e solo se la *password* è corretta. Si noti che la *password* fornita dall'utente dovrà essere salvata in un file una volta cifrata.
Nel corso della progettazione software è opportuno aggiornare una tabella che associ i requisiti definiti in precedenza con le classi già definite in UML e i relativi metodi e attributi.

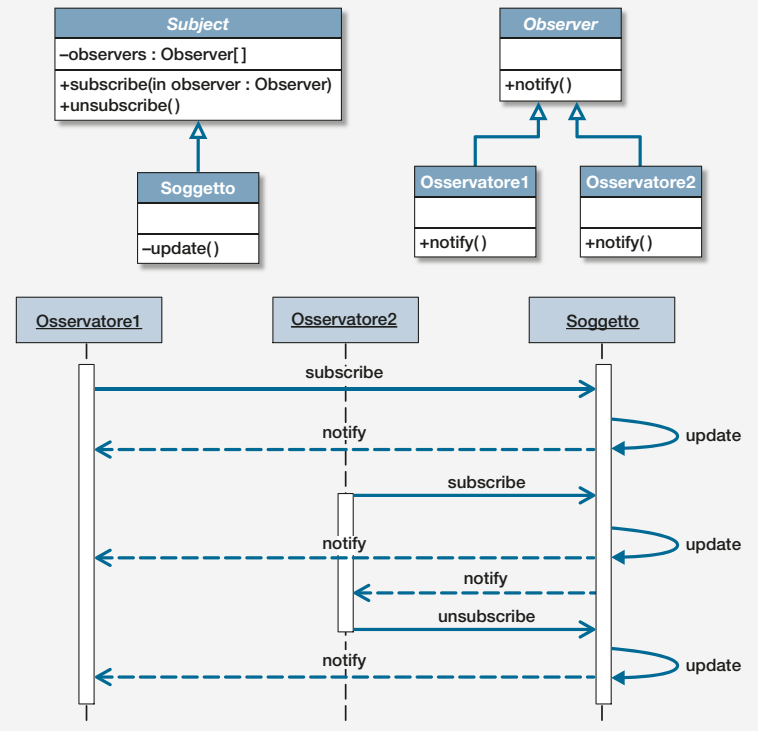
Requisito	Tipologia	Priorità	Definizione	Classe attributo e/o metodo
1	Funzionale	MUST	L'utente inizia il viaggio.	*ATC beginJourney*
2	Funzionale	MUST	L'utente finisce il viaggio.	*ATC endJourney*
3	Funzionale	MUST	L'utente richiede al sistema la visualizzazione della distanza totale percorsa, del tempo complessivo trascorso dall'inizio del viaggio e della velocità media mantenuta.	*ATC positions getJourneyInfo*
4	Funzionale	MUST	Il sistema aggiorna ogni secondo e visualizza automaticamente all'utente la posizione corrente, la data/ora, la velocità e la direzione rispetto al Nord attuali.	*ATC positions actualDirection actualSpeed getActualPosition*
5	Funzionale	SHOULD	L'utente richiede al sistema la visualizzazione della posizione a una data/ora precedente l'attuale e successiva all'inizio del viaggio.	*ATC positions getPastPosition*
6	Funzionale	SHOULD	L'utente richiede al sistema il salvataggio su file dei dati relativi all'intero viaggio.	*ATC positions saveToFile*
7	Non funzionale	SHOULD	Il sistema salva automaticamente e periodicamente su un file i dati relativi all'intero viaggio.	*ATC positions saveToFile*
8	Tecnologico	MAY	Il formato del file di salvataggio dei dati di viaggio è di immediato utilizzo per la visualizzazione delle posizioni in *Google Maps* e *GoogleEarth*.	
9	Tecnologico	MUST	Il sistema interfaccia un dispositivo GPS seriale che fornisce le stringhe previste dallo standard NMEA 0183.	*GPS*
10	Non funzionale	SHOULD	Il sistema consente viaggi lunghi fino a un anno (365 giorni, corrispondenti a 365×24×3600 posizioni memorizzate).	*ATC positions*
11	Non funzionale	MUST	Il sistema non interrompe il suo funzionamento in assenza di informazioni valide fornite dal dispositivo GPS riprendendo automaticamente la memorizzazione quando disponibili.	*GPS*
12	Sicurezza	MAY	Il sistema non consente di selezionare la fine del viaggio a un utente diverso da quello che lo ha iniziato.	*ATC password endJourney*
13	Sicurezza	MAY	Il sistema non consente a un utente diverso da quello che ha iniziato il viaggio di visualizzare le posizioni precedenti.	*ATC password getPastPosition*
14	Non funzionale	MAY	Il sistema è in grado di riprendere il proprio funzionamento dopo un'interruzione.	*ATC loadFromFile* [3]
15	Tecnologico	SHOULD	Il sistema è implementato in linguaggio Java in modo da risultare fruibile su computer di tipo diverso, in particolare su dispositivi palmari e mobili.	

3. Per soddisfare il requisito è sufficiente che il costruttore della classe *ATC* inizializzi il vettore *positions* con le posizioni in precedenza salvate automaticamente in un file di nome predefinito.

Trascurando per il momento il requisito 15 che richiede, anche se in modo non obbligatorio, la scelta di uno specifico linguaggio di programmazione, la compilazione della tabella evidenzia che la nostra progettazione ha soddisfatto tutti i requisiti funzionali, ma non è evidentemente ancora completa in quanto non sono stati soddisfatti vari requisiti non funzionali e tecnologici di cui alcuni classificati come «MUST» (9 e 11). L'aspetto dell'applicazione che abbiamo finora trascurato è sicuramente il più complesso: si tratta dell'interfacciamento con il dispositivo GPS seriale. Al di là degli aspetti implementativi – che abbiamo già preso in esame nell'ultimo capitolo del primo volume – il problema è come fare in modo che un oggetto di classe *ATC* riceva, ogni volta che essa è disponibile, la nuova posizione rilevata automaticamente da un oggetto di un'ipotetica classe *GPS*. Questo è un problema di programmazione ricorrente ed è «risolto» da un *pattern* comportamentale diffusamente implementato dagli sviluppatori di applicazioni OO:

PATTERN

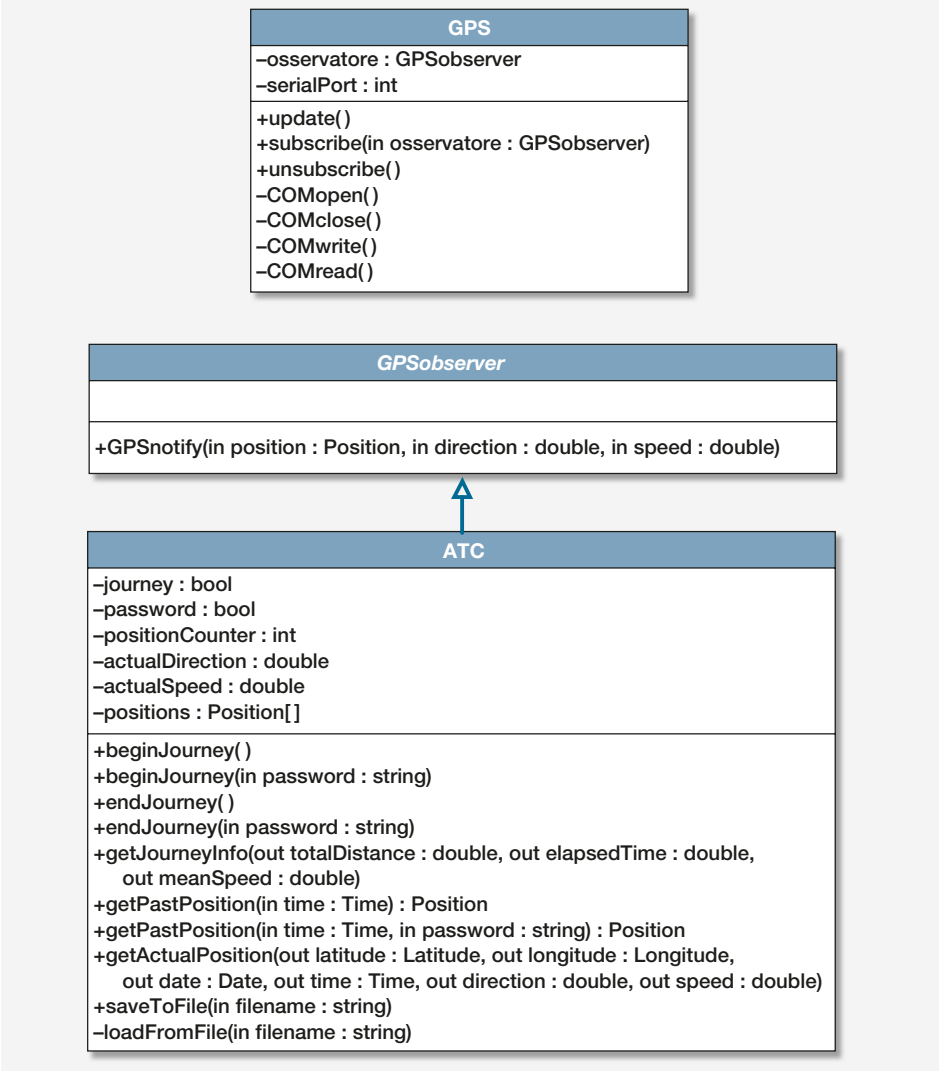
Nome	***Observer***
Problema	Definire una dipendenza tra un «soggetto» e molti «osservatori» in modo che un cambiamento di stato dell'oggetto soggetto sia notificato a tutti gli oggetti osservatori.
Contesto e forze	La progettazione OO comporta l'articolazione del software in classi distinte che necessitano di cooperare senza dipendere troppo strettamente l'una dall'altra per non creare un accoppiamento tale da impedirne il riuso in contesti diversi.
Soluzione	Dovendo notificare i cambiamenti di stato del soggetto agli osservatori, a questi viene richiesto di sottoscriversi presso il soggetto che provvederà a notificare a ciascuno di essi i propri cambiamenti di stato invocando un metodo specifico. Allo scopo viene definita una classe astratta, che definisce il prototipo del metodo di notifica, da cui la classe degli osservatori deriva: in questo modo la sottoscrizione di un'istanza di un oggetto osservatore presso il soggetto rende disponibile un metodo specifico di notifica da invocare a ogni aggiornamento dello stato. I seguenti diagrammi UML illustrano la soluzione prospettata.



Contesto risultante	L'interdipendenza tra le classi del soggetto e degli osservatori è minima ed esse possono essere utilizzate indipendentemente l'una dall'altra; il soggetto «conosce» tutti i propri osservatori in un dato momento ed essi possono essere aggiunti o eliminati in modo dinamico e indipendente dal soggetto stesso.

L'aggiornamento automatico dello stato interno del soggetto mediante invocazione del metodo privato *update* richiede che esso sia un oggetto «attivo» che evolve

autonomamente dall'invocazione dei propri metodi da parte di altri oggetti. Sviluppando applicazioni per i moderni sistemi operativi è possibile costruire oggetti di questo tipo attivando uno specifico *thread* nel costruttore dell'oggetto.
Nello specifico dello studio di un caso, è possibile evitare di definire la classe astratta *Subject* definendo direttamente i metodi *subscribe* e *unsubscribe* nella classe GPS che rappresenta funzionalmente il dispositivo e che incorpora le funzioni per la gestione della comunicazione seriale; inoltre, dato che l'osservatore è unico, non è necessario definire un vettore ma è sufficiente un solo attributo di tipo *Observer*.
Per non rendere necessaria l'implementazione della classe *GPS* in modo che gli oggetti istanziati risultino «attivi», il metodo *update* è stato classificato come pubblico consentendone l'invocazione continua e ripetuta da parte del programma principale dell'applicazione ATC.
Il seguente diagramma UML rappresenta questa particolare realizzazione del *pattern Observer*:



La classe *GPS* e il metodo *notify* che la classe *ATC* eredita dalla classe astratta *GPSobserver* consentono di soddisfare i requisiti 9 e 11 (quest'ultimo è diretta conseguenza del fatto che in caso di non funzionamento del dispositivo GPS non viene invocato il metodo *notify*) completando così la nostra progettazione software. Oltre al requisito 15 relativo alla scelta del linguaggio di programmazione, rimane infatti non soddisfatto il solo requisito 8 relativo al formato del file di salvataggio delle posizioni che sarà definito in fase di sviluppo.