Preparato da: Matteo Menichetti
Data: 30/12/2019

Progetto

Descrizione delle funzionalità del sistema

L'idea con la quale è stato creato questo progetto è quella di fornire un sistema software in grado di gestire alcuni mezzi di trasporto, con relativi hub, offerti dallo Stato italiano. I mezzi di trasporto pubblici considerati sono gli aerei, i bus ed i treni. I bus ed i treni presi in considerazione sono utilizzati per trasportare le persone verso un'aeroporto, quindi viene assunto che un aeroporto sia il punto di arrivo di bus o di treni. Le macchine e le motociclette rappresentano i veicoli delle persone che utilizzeranno i mezzi pubblici. Questi saranno parcheggiati nei garage.

I mezzi pubblici sono rappresentati dalle classi AirPlaneComposite, BusComposite e TrainComposite. I mezzi delle persone sono rappresentati dalle classi CarLeaf e MotorBikeLeaf.

Gli Hub sono rappresentati dalle classi AirPortComposite, StationComposite e GarageComposite che a loro volta, come verrà descritto in seguito, sono composizioni di oggetti.

Il sistema prevede di avere all'apice un aeroporto, il quale può contenere più gate, stazioni, piste di atterraggio e garage.
Aeroporti e stazioni possono avere, rispettivamente, più gate, stazioni, piste di atterraggio, garage o binari.
Binari, piste di atterraggio, gates possono essere occupati al più un veicolo.
Gli schermi, contenuti all'interno di un Hub, sono replicati all'interno della struttura che contiene l'Hub. Uno schermo ha la funzione di far visualizzare il veicolo che staziona nell'Hub in cui è posizionato.

Motivazioni della scelta dei Pattern utilizzati e dell'implementazione del sistema

I pattern utilizzati in questo progetto sono i seguenti: Adapter, Composite, Iterator, Observer e Visitor.

Adapter

Il pattern Adapter è stato utilizzato in quanto, nell'implementazione del pattern Visitor, alcuni dei metodi ereditati non avrebbero avuto un utilizzo pratico.
La classe PersonVisitorAdapter fornisce un implementazione "vuota" dei metodi del Visitor che presentano come argomento un oggetto (Composto o Foglia) che non contiene direttamente o indirettamente persone. I veicoli che contengono persone sono gli aeroplani, i bus ed i treni e indirettamente tutti gli Hub dove i questi veicoli stazionano.

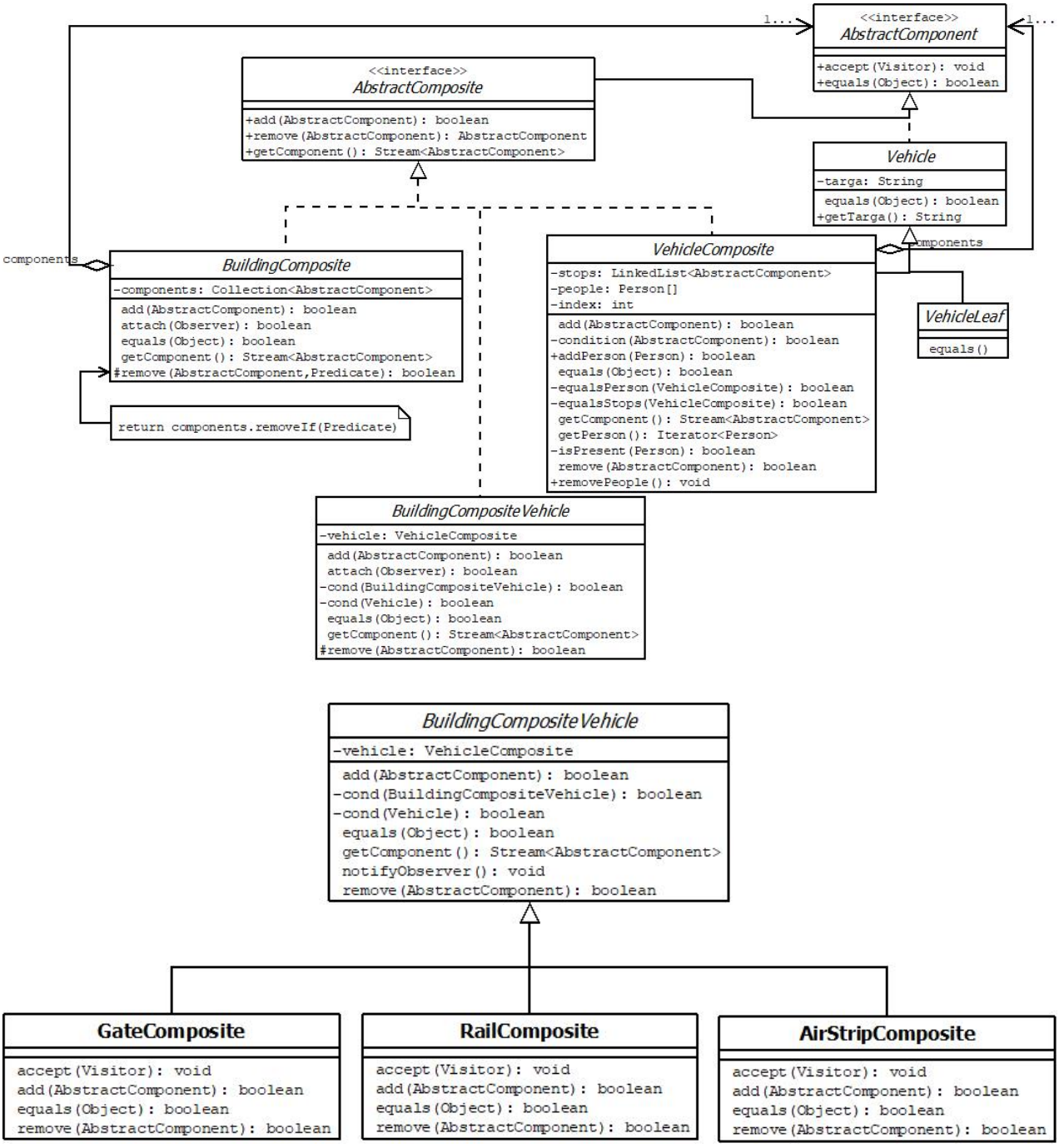
La classe PersonVisitorAdapter è implementata all'interno del package mp.progetto.pattern.adapter e viene estesa da PersonVisitor.

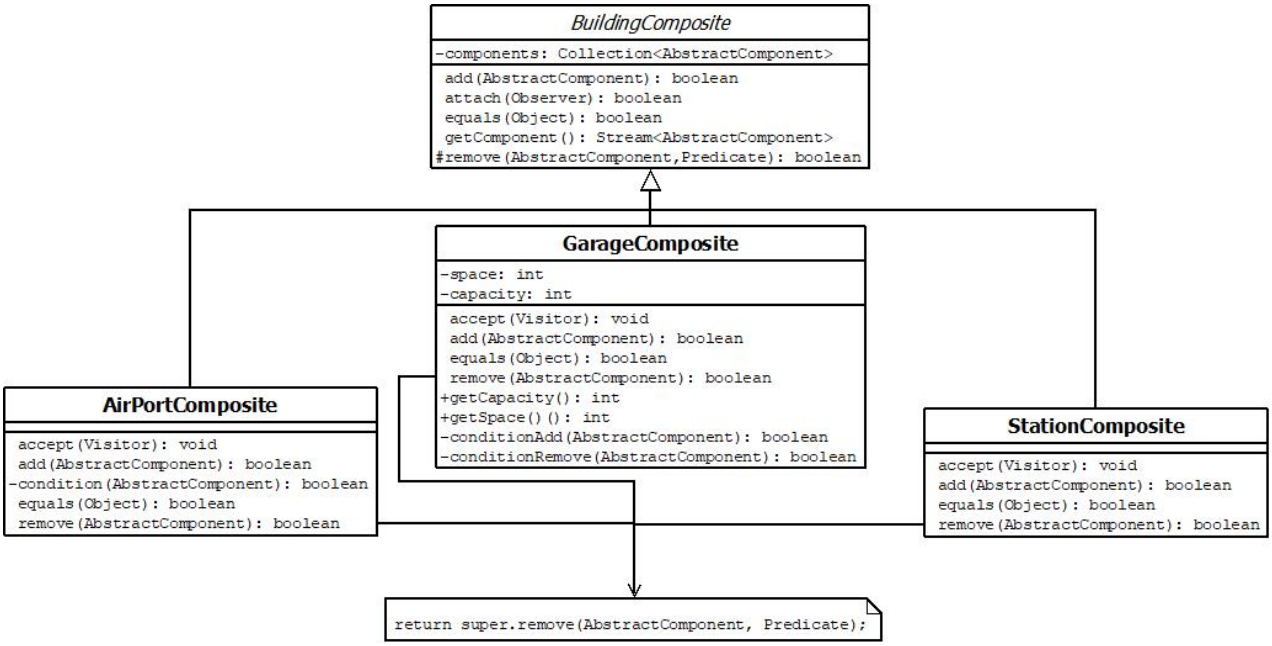
Composite

Il pattern Composite è stato utilizzato in quanto il suo intento permette di trattare strutture ad albero come Hub che contengono veicoli, Hub che contengono Hub e veicoli che contengono persone.

Assumiamo che le macchine e le motociclette non siano composti, che non contengono persone, perché non è possibile sapere con quali mezzi privati le persone arriveranno ai garage.

Segue il diagramma UML delle classi astratte e interfacce che compongono il pattern composite (contenute all'interno del pacchetto mp.progetto.pattern.composite) ed i diagrammi UML delle classi concrete (contenute all'interno del pacchetto mp.progetto.builndings.concrete), degli Hub, che implementano le classi astratte e le interfacce del pattern.





AbstractComponent rappresenta l'interfaccia comune a tutti gli oggetti della composizione e definisce i metodi (accept ed equals) che verranno implementati da questi.

AbstractComposite rappresenta l'interfaccia comune a tutti gli oggetti composti e dichiara i metodi add, remove e getComponent. AbstractComposite implementa AbstractComponent
Il metodo add viene dichiarato avendo come argomento un oggetto di tipo AbstractComponent e restituisce un valore di tipo boolean utile per sapere l'esito dell'operazione. Il principio con cui viene implementato il metodo add è quello di aggiungere un oggetto se questo non esiste già o il posto che vuole occupare non è già utilizzato. Considerazioni analoghe vengono fatte per il metodo remove e viene gestita l'evenienza che l'oggetto da eliminare non esista.
Il metodo getComponent resitituisce un oggetto di tipo Stream<AbstractComponent> utile per le operazioni che verranno definite quando l'interfaccia verrà implementata.

BuildingComposite rappresenta la classe, astratta, comune a tutti gli oggetti che, come le stazioni e gli aeroporti, sono Hub composti da altri Hub. Gli Hub sono contenuti in una struttura dati che implementa l'interfaccia Collection. BuildingComposite implementa AbstractComposite.
Il metodo add verifica che l'oggetto "passato" non esista già all'interno della Collection components.
Il metodo remove non implementa il metodo dell'interfaccia. Tale metodo verrà implementato dalle classi concrete. Questo metodo rimuove dalla Collection components il componente che rispetta le condizioni specificate dal Predicate (tali condizioni sono esplicate in seguito).

BuildingCompositeVehicle rappresenta la classe, astratta, comune a tutti gli Hub che, come la pista di atterraggio o il binario, sono composizioni di un veicolo e degli schermi, che daranno la rappresentazione del veicolo che staziona all'hub. BuildingCompositeVehicle implementa AsbtractComposite.
Il metodo add verifica, prima di aggiungere l'argomento ricevuto che questo non sia già presente e che sia un istanza della classe VehicleComposite. Se tali condizioni si verificano il metodo restituisce true, altrimenti false.
Il metodo attach riceve un Observer e setta il subject di tale Observer invocando il setter con argomento il riferimento soggetto da osservare.
I metodi cond sono metodi di "appoggio". Il metodo con parametro di tipo BuildingCompositeVehicle è invocato dal metodo equals. Questo metodo ha lo scopo di verificare che i veicoli dell'oggetto sul quale viene invocato abbia gli stessi veicoli dell'oggetto che è stato "passato" al metodo equals.
Il metodo cond con argomento di tipo Vehicle viene invocato dal metodo remove e verifica che il veicolo da eliminare esista effettivamente all'interno dell'Hub.
Il metodo remove è dichiarato come protected e non implementa il metodo dell'interfaccia ma elimina il veicolo, se le condizioni precedentemente esplicate sono verificate, e invoca il metodo notifyObserver.

Vehicle rappresenta la classe, astratta, comune a tutti i veicoli. La classe definisce getTarga e ridefinisce il metodo equals. Vehicle implementa AbstractComponent.

VehicleComposite rappresenta la classe, astratta, comune a tutti i veicoli che, come il treno, gli aerei ed i bus sono composti da persone e fermate. VehicleComposite estende Vehicle.
Il metodo add è invocato per aggiungere un Hub come fermata al veicolo ed implementa il metodo dell'interfaccia AbstractComposite. Questo metodo invoca il metodo condition, il quale verifica che il component che deve essere aggiunto alle fermate sia un edificio.
Il metodo addPerson ha lo scopo di aggiungere una persona al veicolo fin quando non vengono esauriti i posti.
Il metodo equals invocando i due metodi equalsStop ed equalPerson verifica se due oggetti sono uguali. equalsStop verifica che le fermate siano uguali confrontando le due linkedlist se le dimensioni sono uguali e sono vuote (sennò verifica che non siano entrambe vuote). equalsPerson sfrutta il pattern Iterator per confrontare, elemento per elemento, i vettori people.
Il metodo getPerson restituisce un Iterator<Person> derivato dalla classe PersonIterator per rappresentare il vettore people senza esporre la struttura data utilizzata.
Il metodo isPresent è invocato dal addPerson e verifica se una persona è presente tra le persone del vettore people.
Il metodo removePeople ha lo scopo di rimuovere le persone dal veicolo in quanto quest'ultimo a fine corsa.

VehicleLeaf rappresenta la classe, astratta, comune a tutti i veicoli, come macchine e motociclette, che vengono utilizzate dalle persone che usufruiscono dei mezzi di trasporto pubblico. Questi veicoli non possono accedere a nessun Hub se non ad un Garage preposto. VehicleLeaf estende Vehicle.

GarageComposite rappresenta un garage dove sono contenuti i veicoli delle persone. space e capacity rappresentano lo spazio, che varia con l'aggiunta e la rimozione dei veicoli, e la quantità di veicoli che il garage può ospitare. GarageComposite estende BuildingCompositeVehicle.
Il metodo add verifica che l'oggetto ricevuto sia un veicolo e che lo spazio sia sufficiente prima di invocare il metodo add della super classe (aggiungendolo se non esiste già). Se le condizioni si sono verificate diminuisce lo spazio disponibile e restituisce true, altrimenti false.
Il metodo remove invoca il metodo remove della super classe.

AirPortComposite rappresenta un aeroporto. Questa classe è un Hub di Hub ed estende BuildingComposite.
Il metodo add verifica che l'oggetto ricevuto sia un'edificio. Se lo è questo viene aggiunto agli Hub che compongono l'aeroporto insieme agli schermi.
Il metodo remove è l'operazione inversa. Elimina gli schermi contenuti nell'hub eliminato ed invoca il metodo remove della super classe BuildingComposite sul componente ricevuto. Tale componente viene eliminato se il predicato è verificato.

La classe **StationComposite** implementa il medesimo comportamento ed anch'essa estende BuildingComposite.

Le classi **AirStripComposite, GateComposite, RailComposite** implementano la classe astratta **BuildingCompositeVehicle** eseguendo type checking sugli argomenti che i metodi add, remove ed equals ricevono.

Iterator
Il pattern Iterator è utilizzato per esporre la struttura dati Person[] people della classe astratta VehicleComposite. Il pattern è stato implementato per questa struttura dati. In strutture dati più complesse, utilizzate nel progetto,(ArrayList, LinkedList, ecc.) è disponibile il metodo iterator, da invocare su tali strutture e per questo il pattern non viene applicato a tali strutture

All'interno del pacchetto mp.progetto.pattern.iterator viene dichiarata l'interfaccia Iterator e la classe astratta PersonIterator.

PersonIterator implementa l'interfaccia iterator e ridefinisce il metodo equals
Il metodo equals confronta ogni elemento di entrambi i PersonIterator e restituisce true se questi sono uguali.

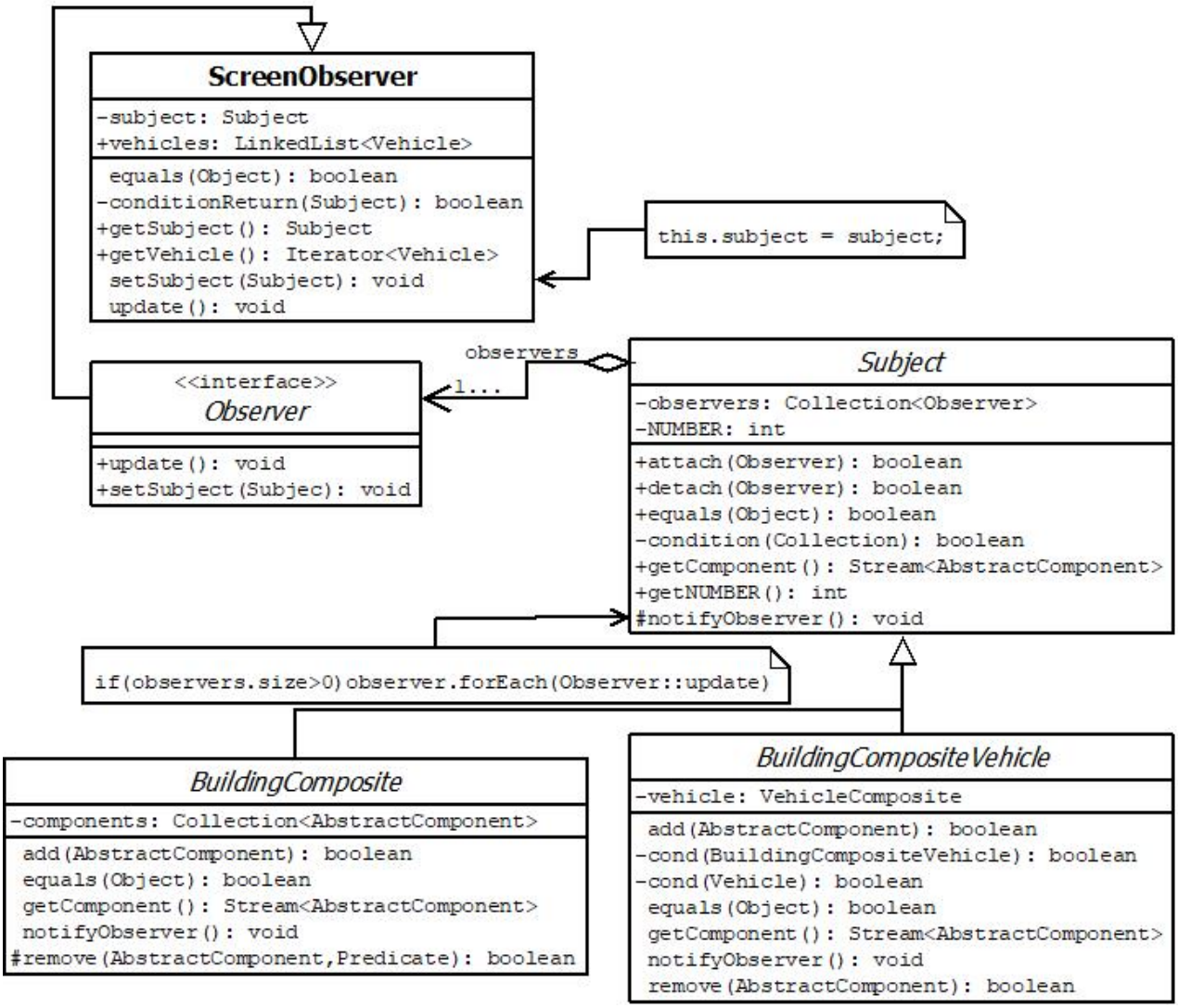
Observer
L'idea con la quale è stato applicato questo pattern è quella di rappresentare i veicoli (treni ed aerei) che stazionano negli Hub preposti attraverso degli schermi. Gli schermi degli Hub composti da altri Hub sono delle repliche e si assume che

dopo che un Hub (ad esempio un Binario) è stato aggiunto ad un Hub, di maggiore rilevanza, non gli vengano aggiunti ulteriori schermi. Questo pattern permette, con una situazione di dipendenza uno-molti, di mantenere la consistenza tra i dati sugli schermi dopo che lo stato dei soggetti osservati è stato modificato. L'interfaccia Observer viene implementata dalla classe ScreenObserver, che rappresenta gli schermi. La classe astratta Subject, che rappresenta gli oggetti che sono osservati, viene estesa da BuildingsComposite e BuildingCompositeVehicle.

Observer definisce i metodi update e setSubect i quali sono implementati da ScreenObserver e quando vengono invocati hanno questo comportamento:

- setSubject imposta il soggetto;
- update verifica se si sono verificati cambiamenti nello stato di subject. Se risulta che sia stato aggiunto/eliminato un veicolo tramite verifica sullo stato di subject viene aggiornata la lista di elementi dello schermo.

Viene riportato il diagramma UML del pattern Observer applicato al progetto. Tali interfacce e classi, astratte e concrete, appartengono al pacchetto mp.progetto.pattern.observer (eccezione fatta per le classi BuildingsComposite e BuildingsCompositeVehicle viste in precedenza ma che implementano Subject).



Observer è l'interfaccia che definisce i metodi che dovranno essere implementati dagli osservatori.
ScreenObserver implementa l'interfaccia Observer e rappresenta gli schermi. Viene mantenuto il riferimento all'osservato per effettuare le dovute modifiche sulla LinkedList di veicoli. È stata scelta una LinkedList perché su uno schermo viene ordinata la lista di veicoli aggiungendo all'ultimo posto il veicolo che arriva per ultimo e viceversa per la rimozione di un veicolo.
Il metodo setSubject deve essere invocato quando viene eseguita l'operazione di attach (Classe Subject) di un'oggetto di tipo Observer, in questo caso di tipo ScreenObserver, per la corretta esecuzione del metodo update.
Il metodo update verifica che tutti i veicoli di Subject siano contenuti in vehicles, dove sono contenuti i veicoli dello schermo, e se sono stati rimossi veicoli dal soggetto osservato.

Visitor
L'idea con la quale è stato applicato questo pattern è quello di aggiungere funzionalità che altrimenti sarebbero state complesse da implementare dato la conformazione delle classi dopo aver applicato il pattern Composite. Sono stati implementati due visitor: PersonVisitor e VehicleVisitor. I due Visitor hanno lo scopo di visitare un Hub, l'oggetto sul quale viene invocato il metodo accept, e restituire le persone o i veicoli che stazionano in un preciso momento nell'Hub. Dato che sono previste operazioni su oggetti di tipo Person e Vehicle l'interfaccia Visitor sarà generica e saranno i Visitor concreti a definire il tipo. Per la finalità delle operazioni da eseguire è stato scelto di applicare il Visitor void e di utilizzare un "accumulatore".

I Visitor concreti e l'interfaccia Visitor sono dichiarati all'interno del package mp.progetto.pattern.visitor.

PersonVisitor è la classe che permette di stabilire chi è all'interno di un'edificio. La classe non implementa tutti i metodi visit che sono definiti nella interfaccia Visitor ed estende la classe PersonVisitorAdapter in quanto non vengono prese in considerazione le persone all'interno dei propri veicoli. I metodi visit che "visitano" edifici che sono sottotipi di BuildingComposite sono composizioni di oggetti e contengono al loro interno Hub e per questo invocano il metodo visitBuildingComposite.
I metodi visit che "visitano" edifici che sono sottotipi di BuildingCompositeVehicle sono composizioni di oggetti e contengono al loro interno veicoli e per questo invocano il metodo visitBuildingCompositeVehicle.
Il metodo visitBuildingComposite invoca per ogni Hub, contenuto all'interno di un edificio, il metodo accept con argomento un riferimento al Visitor stesso.
Il metodo visitBuildingCompositeVehicle invoca sul veicolo, contenuto all'interno dell'Hub, se esiste, il metodo accept con argomento un riferimento al Visitor stesso.
I metodi visit che "visitano" veicoli invocano il metodo addPeople, con argomento il riferimento al veicolo da visitare, e permettono di aggiungere alla collezione tutte le person contenute all'interno del veicolo contenuto all'interno di un edificio.

VehicleVisitor è la classe che permette di stabilire quanti veicoli siano all'interno di un edificio.
I metodi visit che "visitano" edifici che sono sottotipi di BuildingComposite invocano il metodo visitBuildingComposite. Tale metodo invoca il metodo accept su ogni hub contenuto all'interno dell'edificio da visitare.
I metodi visit che "visitano" edifici che sono sottotipi di BuildingCompositeVehicle invocano il metodo visitBuildingCompositeVehicle. Tale metodo invoca accept, con argomento il riferimento alla classe VehicleVisitor, sul veicolo, se presente, contenuto all'interno dell'hub.
I metodi visit che "visitano" veicoli invocano il metodo add ed aggiungono il veicolo ricevuto come parametro alla collezione vehicles.

Entrambe le classi si basano sul metodo getAccumulator per restituire i veicoli e le persone all'interno degli edifici.