

Relazione Progetto

Giannuzzi Riccardo

13 ottobre 2024

Indice

1 Descrizione caso di studio	3
1.1 Generatori	3
1.2 Sistemi	3
1.3 Operazioni sui componenti	3
1.4 Rifornimento Carburante	4
1.5 Monitoraggio Carburante	4
2 Schema UML	5
3 Design patterns	6
4 Scelte di Design e di Implementazione	6
4.1 Generatori e Sistemi	6
4.1.1 Implementazione del composite pattern	6
4.1.2 Testing di generatori e sistemi	7
4.2 Operazioni sulla gerarchia	7
4.2.1 Implementazione del visitor pattern	8
4.2.2 Implementazione default del visitor	8
4.2.3 Implementazione dei visitor concreti	8
4.2.4 Testing dei visitor	9
4.2.5 L'operazione <i>produceEnergy()</i> non è implementata tramite visitor	9
4.2.6 Conseguenze del visitor pattern	9
4.3 Rifornimento e monitoraggio carburante	9
4.3.1 Iplementazione Eventi	9
4.3.2 Implementazione del Subject	10
4.3.3 Testing della generazione di eventi	10
4.3.4 Implementazione degli Observer	11
4.3.5 Tessting degli observer	11

1 Descrizione caso di studio

Il programma si occupa delle gestione di una rete distribuita che produce energia elettrica.

1.1 Generatori

La produzione di energia elettrica è affidata a dei generatori, che consumano carburante e restituiscono energia. Per semplificare la gestione di energia e carburante, vengono rappresentati entrambi come semplici quantità intere, rispettivamente fuelUnits ed energyUnits. Ogni unità di fuelUnits, quando consumata, produce un'unità di energyUnits.

I generatori mettono a disposizione un metodo (*produceEnergy()*) che su richiesta produce, se il carburante lo permette, un'energyUnit e la restituisce.

1.2 Sistemi

I generatori possono essere raggruppati in un sistema di produttori di energia. Anche il sistema può produrre energia, semplicemente propaga la chiamata di produrre energia a tutti i suoi contenuti. Un sistema, oltre ai generatori, può contenere a sua volta altri sistemi in modo da distribuire i generatori in una struttura di sistemi e sottosistemi.

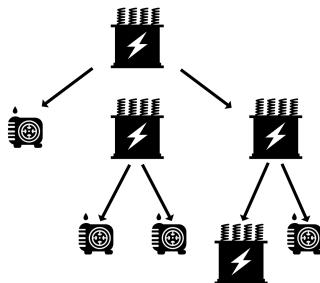


Figura 1: Struttura di un Sistema

1.3 Operazioni sui componenti

Oltre alla produzione di energia, dovranno essere fornire operazioni di utilità, che forniscono informazioni sui contenuti di un sistema:

- Numero di fuelUnits totali contenute nei generatori di un sistema.
- Numero di energyUnit previste per la prossima produzione.
- Rappresentazione con parentesi della struttura di un sistema in una stringa.

1.4 Rifornimento Carburante

Ogni generatore contiene la sua quantità di carburante e può essere rifornito singolarmente tramite il metodo `refuel()`. Dato che non si può sapere a priori quali generatori saranno usati più spesso, e rifornire un intero sistema uniformemente potrebbe essere uno spreco, si implementa un sistema di rifornimento automatico.

Tale sistema mantiene a sua volta una quantità di carburante, che sarà usata per mantenere i generatori riforniti. In particolare, appena un generatore tenta di produrre, deve essere rifornito con le fuelUnits mancanti rispetto alla quantità minima che dovrebbe avere. Se invece il generatore ha già la quantità minima di carburante, oppure il rifornitore non ha abbastanza carburante per rifornire il generatore, il rifornitore non deve fare niente. Quindi se il rifornitore ha una fuelUnit e il generatore ha bisogno di due fuelUnits, il generatore non viene rifornito e il rifornitore mantiene la sua fuelUnit.

Il rifornitore può essere responsabile di un generatore o un intero sistema. Nel secondo caso dove essere responsabile anche di tutti gli elementi contenuti dal sistema ed eventuali sotto-sistemi. Inoltre deve essere responsabile di eventuali elementi aggiunti e cessare di essere responsabile di eventuali elementi rimossi.

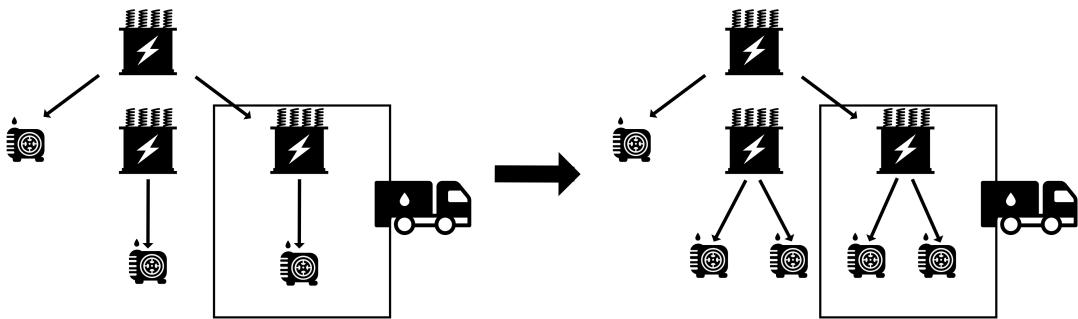


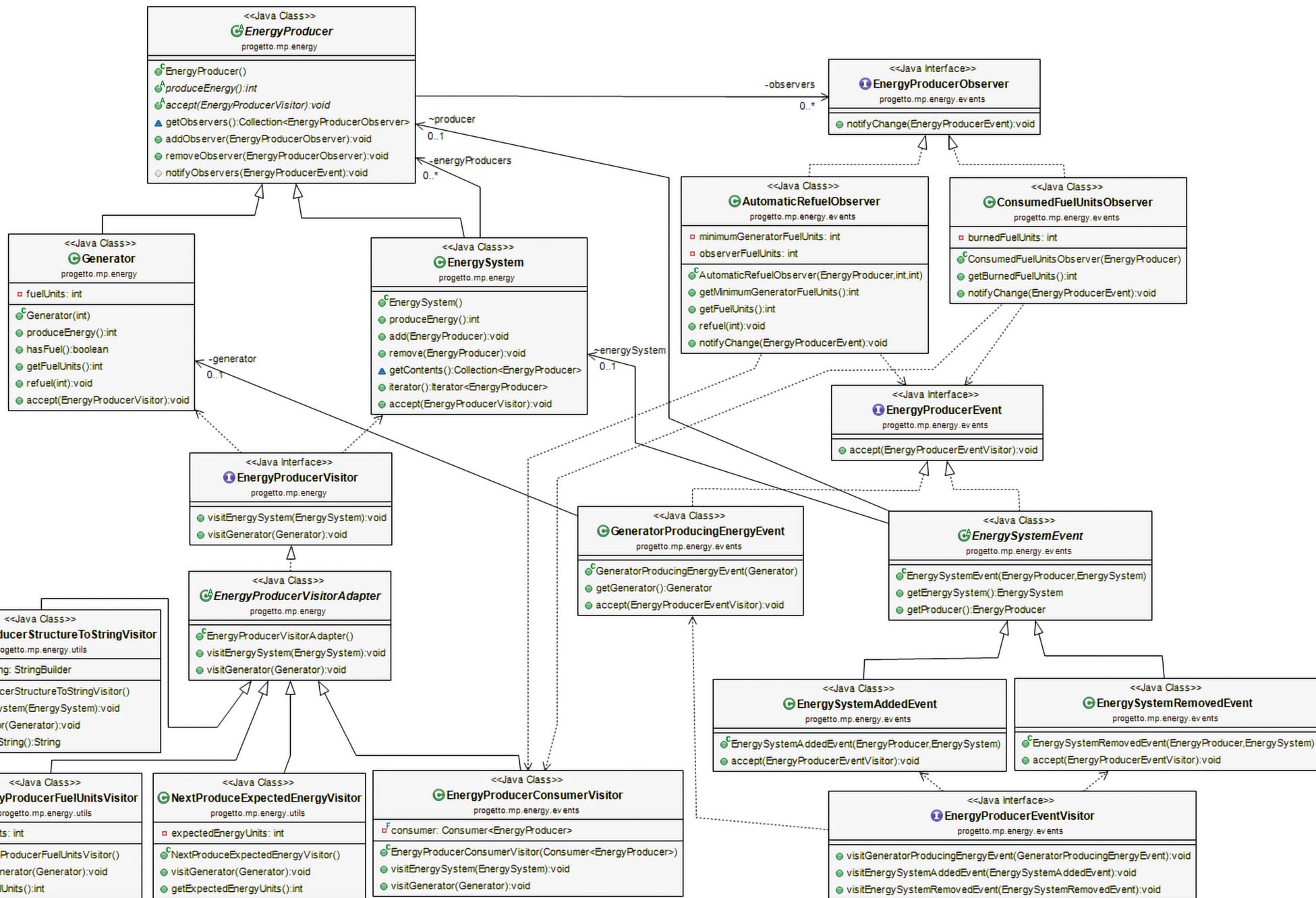
Figura 2: Rifornitore

1.5 Monitoraggio Carburante

Dato che i vari generatori possono essere distribuiti tra più sistemi, può essere utile tenere traccia di quanto un generatore, o un sistema, viene usato. Questo significa monitorare il consumo totale di carburante di un sistema o un generatore. Come per il rifornitore, il monitoraggio può essere responsabile di un generatore o di un intero sistema. Quindi, nel secondo, caso dovrà tenersi aggiornato rispetto a eventuali elementi aggiunti o rimossi.

Una volta istanziato, il sistema di monitoraggio, deve mantenere aggiornato il numero di fuelUnits consumate dalle operazioni `produceEnergy()` andate a buon fine. Quindi, se un generatore tenta di produrre energia, ma non ha carburante, non deve essere incrementato il numero di fuelUnits totali consumate. Stessa cosa nel caso in cui viene eseguita una produzione da parte di un elemento di cui non è responsabile.

2 Schema UML



3 Design patterns

Sono stati applicati quattro design patterns visti a lezione:

- Composite
- Vistor
- Adapter
- Observer

4 Scelte di Design e di Implementazione

Approfondimenti sulle scelte che hanno portato all'applicazione dei Design Pattern e di come sono stati implementati, tenendo in considerazione anche i principi di design SOLID.

4.1 Generatori e Sistemi

La gerarchia di generatori e sistemi è una composizione ricorsiva di oggetti in una struttura ad albero. I generatori sono le foglie e i sistemi sono i nodi intermedi, i cui figli non sono necessariamente foglie. Inoltre è presente l'operazione *produceEnergy()* comune a tutte le componenti della gerarchia: i generatori devono produrre energia usando il proprio carburante, mentre i sistemi devono inoltrare l'operazione ai propri figli.

Quindi si può applicare il pattern strutturale **Composite**. In particolare, si preferisce la versione type safe, che prevede la gestione dei figli gestita solo dal sistema, in modo da evitare eventuali errori di tipo a run-time.

4.1.1 Implementazione del composite pattern

Il ruolo di Component viene implementato tramite una classe astratta (non un'interfaccia per motivi trattati nelle sezioni successive) *EnergyProducer* che dichiara l'operazione *produceEnergy()*. Mentre i ruoli di Leaf e il Composite sono implementati dalle classi concrete:

- *Generator*: implementa la produzione di energia, consumando, se disponibile, una fuelUnit e restituendo una EnergyUnit. Se non sono presenti fuelUnits semplicemente la quantità di energia prodotta dal generatore è 0. Ogni generatore memorizza le fuelUnits in un campo intero privato inizializzato nel costruttore. Per permettere il rifornimento, viene messo a disposizione un metodo pubblico, che aggiunge le fuelUnits passate come parametro a quelle già presenti nel generatore. Sia per l'inizializzazione, che per il rifornimento, viene lanciata una *IllegalArgumentException* nel caso venga passata una quantità di fuelUnits negativa.

- *EnergySystem*: gestisce le operazioni sui figli, che vengono memorizzati in una *Collection* di *EnergyProducer*. Tale *Collection* viene implementata concretamente con un *ArrayList*. Quindi le operazioni di aggiunta e rimozione dei figli sono implementate tramite *add()* e *remove()* di *ArrayList*. L'implementazione concreta della *Collection* non è decisiva, perché non viene esposta al client e può essere cambiata se i requisiti lo richiedessero in futuro.

L'operazione *produceEnergy()* viene implementata tramite uno Stream generato dalla collezione. Prima viene applicato il metodo di trasformazione *map()* per chiamare *produceEnergy()* su tutti gli elementi della collezione. Poi vengono raccolte le *energyUnits* prodotte, con il metodo di riduzione *reduce()* a cui viene passata una lambda per sommarle.

4.1.2 Testing di generatori e sistemi

Per quanto riguarda i generatori, viene testata tutta la logica della produzione di energia considerando i due casi possibili: generatore con carburante e generatore senza carburante. Inoltre viene testata la logica del refuel, sia nel caso sia passato un argomento positivo, sia nel caso sia passato un argomento negativo in cui deve essere lanciata un'*IllegalArgumentException*. Dato che è presente della logica anche nel costruttore viene testato sia il caso in cui deve essere lanciata l'*IllegalArgumentException*, sia il caso base in cui viene eseguito solo l'assegnamento.

Per i sistemi invece, vengono testate operazioni per aggiungere e rimuovere i figli, usando un metodo package-private che restituisce il riferimento alla collezione. Tale metodo viene usato, nel test dell'operazione per rimuovere i figli, anche per aggiungerli direttamente alla collezione. Questo perché i due metodi per gestire i figli devono essere testati in isolamento. Infine viene testata la logica del metodo *produceEnergy()* su un sistema vuoto, un sistema non vuoto e un sistema innestato.

4.2 Operazioni sulla gerarchia

Si ha una gerarchia di oggetti, sui quali si deve poter aggiungere nuove operazioni senza modificare le classi esistenti, isolando nelle classi principali solo le operazioni di base che necessitano di dettagli implementativi interni e delegando all'esterno le altre operazioni. Nello specifico si vuole processare una struttura di oggetti, di classi differenti ma con supertipo comune, in base al tipo che avranno quegli oggetti a run-time.

Dato che Java non supporta l'overloading dinamico si sopprime a questa mancanza con un pattern comportamentale, il **Visitor**. In particolare, per fornire più libertà ad eventuali implementazioni future, si preferisce la variante con i metodi void, che mantiene uno stato interno e fornisce un metodo per restituire il risultato finale della visita. Quindi, una volta usato, un visitor concreto non dovrà essere utilizzato di nuovo.

4.2.1 Implementazione del visitor pattern

Il visitor potrebbe aver bisogno di accedere sequenzialmente ai figli di un Sistema, quindi, per evitare di esporre l'implementazione concreta della collezione che contiene figli, conviene usare il pattern **Iterator**. Dato che i figli sono memorizzati in una collezione Java non è necessario implementarlo: è sufficiente aggiungere un metodo che restituisce quello già implementato delle collezioni.

Il pattern visitor, viene implementato tramite l'interfaccia *EnergyProducerVisitor*, con un metodo void di visita per i generatori e un'altro per i Sistemi. Viene lasciata ai visitor concreti la responsabilità di implementare lo stato e il metodo per restituirlo. Infine viene aggiunto, come astratto, il metodo *accept()*, nella classe astratta *EnergyProducer*, che dovrà essere implementato dalle due classi concrete: *Generator* ed *EnergySystem*.

4.2.2 Implementazione default del visitor

Visto che si visita una struttura composta annidata, molte implementazioni di visitor probabilmente la visiteranno in profondità usando l'iterator. Dunque conviene fornire un'implementazione di default tramite il pattern strutturale **Adapter**.

Si crea la classe astratta *EnergyProducerVisitorAdapter*, in modo che non sia istanziabile, e le si fa implementare l'interfaccia del visitor. Viene definita un'implementazione vuota, nel caso della visita a un generatore e l'implementazione che va in profondità con l'iterator, nel caso della visita di un sistema.

4.2.3 Implementazione dei visitor concreti

Per implementare le operazioni sui componenti tramite i visitor concreti, dato che devono visitare in profondità, conviene estendere l'adapter:

- Numero fuelUnits totali di un sistema: è sufficiente ridefinire tramite Overriding la visita del generatore. All'interno della visita, si acquisisce il numero di fuelUnits del generatore e si aggiungono a al campo che contiene il risultato.
- Numero energyUnit previste per la prossima produzione: come nel caso precedente si ridefinisce la visita del generatore, ma in questo caso invece di sommare, si controlla se sono presenti fuelUnits. Se sono presenti si incrementa di 1 il campo del risultato.
- Rappresentazione con parentesi della struttura di un sistema in una stringa: in questo caso è necessario ridefinire anche la visita del sistema. Si deve aggiungere una parentesi quadra aperta, prima della chiamata in profondità e una chiusa dopo. La chiamata ai figli viene fatta chiamando il metodo della superclasse, senza usare di nuovo l'iterator. Per la visita dei generatori invece, dato che l'unico elemento di stato sono fuelUnits, vengono aggiunte due parentesi tonde che contengono le fuelUnits del generatore. Per costruire la stringa viene usato un campo privato di tipo *StringBuilder* sul quale viene chiamato metodo *append()*.

Possibili risultati di una visita di un sistema: "[[4)(3)(0)]", "[[2][(8)][(0)]]".

4.2.4 Testing dei visitor

Viene testato, che i vari visitor producano il risultato corretto in tutti casi possibili di gerarchia: solo generatore, sistema vuoto, sistema non vuoto e sistema innestato complesso. Dato che i visitor hanno uno stato, devono essere istanziati prima di ogni test, quindi conviene aggiungere il metodo di setup con annotazione `@Before` nella classe dei JUnit test.

4.2.5 L'operazione `produceEnergy()` non è implementata tramite visitor

L'operazione `produceEnergy()` è sensato rimanga all'interno della gerarchia, in quanto, richiede dettagli implementativi interni. Infatti per estrarre l'operazione di produzione con un visitor si dovrebbero rendere pubblici i dettagli della gestione del carburante.

Se questo non fosse il caso, allora il pattern Composite non avrebbe più un'operazione comune, e senza di essa non si potrebbe più dire di star utilizzando quel pattern.

4.2.6 Conseguenze del visitor pattern

Applicare il pattern visitor comporta che, se in futuro si volesse aggiungere un nuovo tipo alla gerarchia, si deve aggiungere un nuovo metodo all'interfaccia del visitor. Questo romperebbe i visitor concreti già esistenti. Tuttavia secondo le specifiche di questo caso di studio, le classi della struttura cambieranno raramente. D'altro canto si dovranno definire spesso nuove operazioni sulla struttura. Quindi questo compromesso è accettabile.

4.3 Rifornimento e monitoraggio carburante

Il sistema di rifornimento e quello di monitoraggio, dipendono dalle modifiche dello stato dei generatori e dei sistemi che osservano. Quindi ogni volta che cambia lo stato dei generatori, devono essere notificati e aggiornarsi di conseguenza. Ciò implica che i generatori e i sistemi devono notificare, potenzialmente molteplici oggetti, senza sapere chi sono. Inoltre queste classi dovrebbero rimanere disaccoppiate in modo da essere riusabili, mantenibili e facilmente testabili.

Per risolvere questo problema si può applicare il pattern comportamentale **Observer**.

4.3.1 Implementazione Eventi

La notifica notifica viene inviata a tutti gli osservatori, indipendentemente da se sono interessati o meno, quindi è necessario stabilire un protocollo per specificare cosa è cambiato nell'oggetto osservato. Questo viene implementato con una gerarchia di eventi. L'osservato quando notifica gli osservatori specifica, in base a cosa sta cambiando, l'evento che ha causato tale notifica. In questo modo l'osservatore capisce cosa è successo, si aggiorna ed eventualmente, può accedere anche all'oggetto che ha generato l'evento per ottenere ulteriori informazioni. L'evento quindi deve contenere come campi i riferimenti agli oggetti che lo hanno scaturito.

Gli eventi necessari per il rifornimento e il monitoraggio carburante sono tre:

- La produzione di energia del generatore: dovrà contenere il riferimento al generatore
- L'aggiunta di un componente a un sistema: dovrà contenere il riferimento al sistema e il riferimento al componente aggiunto.
- La rimozione di un componente a un sistema: gli stessi riferimenti del caso precedente.

Visto che l'obiettivo finale è comportarsi in modo opportuno in base al tipo di evento a run-time, si rende la gerarchia di eventi visitabile applicando di nuovo il pattern **Visitor**. Ogni classe di evento estenderà una interfaccia con il metodo *accept()* e si definisce l'interfaccia del nuovo visitor, che avrà un metodo di visita per ogni tipo di evento.

L'evento di aggiunta e quello di rimozione condividono gli stessi campi, quindi conviene creare una classe astratta che gestisce i due campi e farla estendere ai due classi concrete. Per l'evento del generatore non è necessario, a meno che non si voglia aggiungere un altro evento con lo stesso campo.

4.3.2 Implementazione del Subject

Il motivo per cui *EnergyProducer*, il Component del Composite pattern, è una classe astratta e non un interfaccia è che rappresenta anche il Subject dell'Observer pattern. Quindi deve fornire e implementare l'interfaccia per la gestione degli osservatori.

Gli osservatori vengono memorizzati in una Collezione Java e vengono forniti i metodi per aggiungerli e rimuoverli. Inoltre viene implementato anche il metodo *notifyObservers()* che notifica tutti gli osservatori, passando come parametro il riferimento a un evento.

A questo punto *Generator* e *EnergySystem*, che già estendevano *EnergyProducer*, rappresentano i ConcreteSubject e dovranno invocare, quando modificano il loro stato, il metodo *notifyObserver()*. In particolare, i generatori lo invocano quando viene chiamato il meotod *produceEnergy()* e i sistemi quando vengono aggiunti o rimossi elementi.

4.3.3 Testing della generazione di eventi

Per testare l'aggiunta e la rimozione degli osservatori è sufficiente definire un observer vuoto con una lambda e controllare, con il metodo package-private che restituisce il riferimento alla collezione, che gli osservatori siano stati effettivamente rimossi o aggiunti.

Invece per testare la generazione di eventi in isolamento è necessario un MockObserver con un metodo che restituisce l'evento generato. In questo modo nei test, dopo aver chiamato il metodo che dovrebbe generare l'evento, si può ottenere il riferimento di tale evento con il MockObserver. Tramite un cast si controlla che l'evento sia quello aspettato, in questo modo, nel caso in cui il cast fallisse, anche il test fallirebbe e questo è giusto perché sarebbe stato generato un evento sbagliato. Inoltre viene testato che il contenuto degli eventi sia corretto.

4.3.4 Implementazione degli Observer

Per implementare gli osservatori concreti viene usato, all'interno del metodo `notifyChange()`, una classe anonima che implementa il visitor degli eventi. In questo modo le notifiche di cambiamento vengono gestite in base al tipo dell'evento a run-time:

- Monitoraggio carburante: Mantiene, in un campo intero, il numero di fuelUnits bruciate dai generatori che osserva. Quindi ogni volta che viene notificato un evento di produzione di energia, controlla se il generatore ha carburante e in caso positivo incrementa il contatore. Per permette ai client di accedere al numero di fuelUnits bruciate, mette a disposizione un getter pubblico.
- Rifornimento carburante: Acquisisce nel costruttore il numero minimo di fuelUnits da mantenere nei generatori e le fuelUnits con cui rifornire i generatori. Poi ogni volta che viene notificato un evento di produzione di energia, controlla quante fuelUnits mancano. Solo nel caso in cui, al generatore manchino più di 0 fuelUnits e l'osservatore abbia abbastanza fuelUnits, viene eseguito il rifornimento.

Entrambi gli osservatori devono gestire l'aggiunta e la rimozione di elementi degli *EnergySystem* che osservano. Questo viene implementato con un visitor che acquisisce nel costruttore un Consumer di EnergyProducer. Tale visitor applica, andando in profondità, il consumer agli elementi su cui è stato chiamato. Dato che è usato da entrambi gli osservatori, invece di dichiararlo come classe interna, il visitor viene definito in un file a parte.

4.3.5 Tessting degli observer

Per entrambi gli observer vengono testati gli eventi nei seguenti casi:

- Eventi su elementi già istanziati prima dell'aggiunta dell'observer.
- Eventi di oggetti non osservati dall'observer.
- Eventi di oggetti semplici aggiunti e poi rimossi dopo l'aggiunta dell'observer
- Eventi di oggetti innestati aggiunti e poi rimossi dopo l'aggiunta dell'observer

Inoltre viene testata la logica interna, in modo da coprire ogni possibile diramazione delle condizioni degli observer. Dell'observer per il rifornimento di carburante vengono testati anche il costruttore e i metodi refuel, poiché, come per il generatore, possono lanciare una `IllegalArgumentException` nel caso in cui viene passato una quantità negativa di fuelUnits.