RELAZIONE PROGETTO ALBERTO BILIOTTI

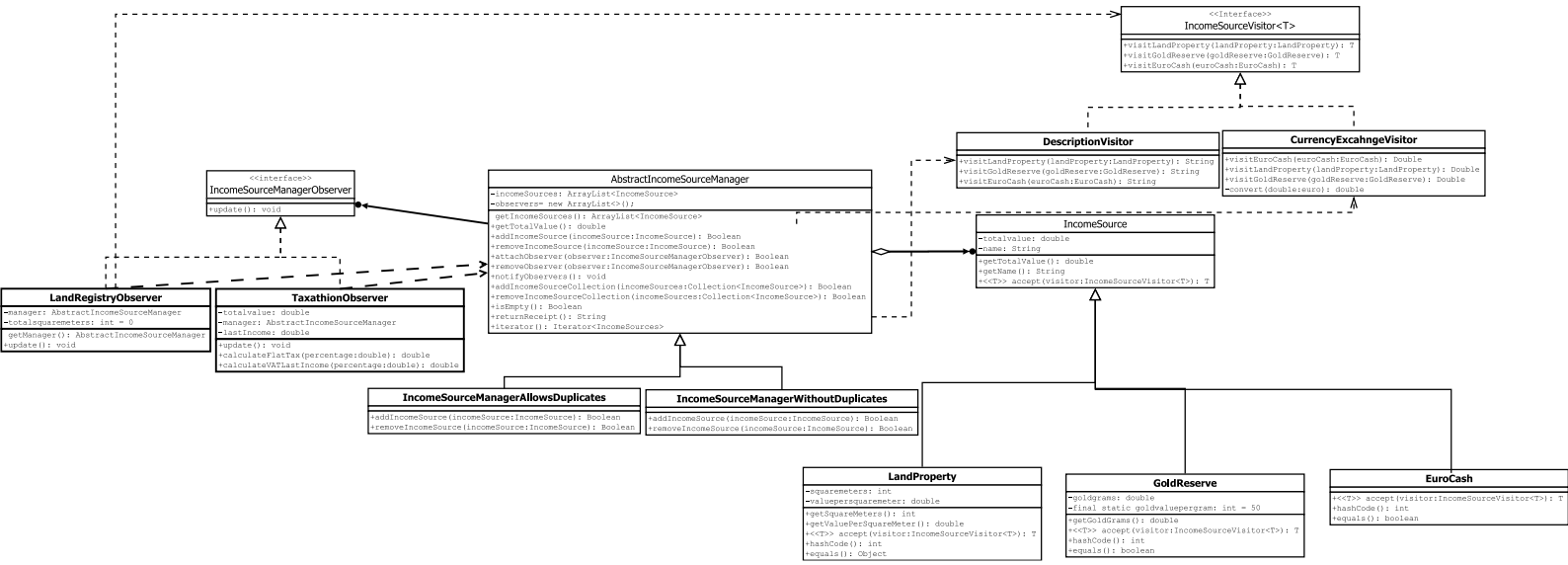
Numero di matricola: 7026939

Email: alberto.biliotti@stud.unifi.it

Descrizione funzionalità

Lo "scopo" di questo progetto è implementare un gestore per tre diversi tipi di fonti di profitto: Immobili, riserve d'oro e riserve di contanti: esso è in grado di gestire una collezione potenzialmente infinita di fonti di profitto (o collezioni di esse) aggiungendone o rimuovendone quando richiesto: per fare ciò imposterò i metodi di aggiunta e rimozione di collezioni come Template Method che avranno come "hooks", (ovvero come metodi astratti che andranno implementati dalle sottoclassi) i metodi di aggiunta e di rimozione di un singolo elemento in quanto voglio lasciare la libertà al Client di aggiungere o rimuovere fonti profitto come risulta più comodo(nel progetto come esempio ho implementato due diverse sottoclassi di cui una permette di aggiungere fonti di profitto ripetute mentre l'altra lo impedisce).
Inoltre il gestore è in grado di: controllare se è vuoto o meno e nel caso restituire il booleano 'true', restituire un iteratore contenente tutte le fonti di profitto contenute nel gestore, restituire il valore totale di tutte le fonti di profitto contenute nel gestore e restituire una "ricevuta" in forma di stringa di tutte le fonti di profitto indicandone le caratteristiche principali di ogni fonte di profitto.
Il gestore potrà inoltre sia avvisare eventuali enti esterni(qui nel progetto ho simulato un eventuale ente catastale ed uno fiscale), sia eventualmente in futuro avvalersi di un sistema esterno per la rappresentazione dei dati ottenuti dal gestore grazie al pattern "Observer" che ho implementato attraverso l'interfaccia "IncomeSourceManagerObserver", Inoltre un motivo per cui ho scelto di lasciare i metodi di aggiunta delle fonti di profitto astratti è proprio per permettere al Client di poter decidere come avvisare i propri osservatori in totale libertà.
Infine è bene soffermarsi anche sulle entità semplici del mio progetto: le fonti di profitto, come già detto possono essere di tre tipi diversi: Immobili, riserve d'oro e riserve di contanti.
Essendo entità semplici non hanno molta logica ma posso comunque verificare l'uguaglianza tra loro e volendo il Client può anche aggiungere operazioni attraverso il pattern Visitor di cui, come esempio, ho realizzato due implementazioni nelle quali i metodi di una permettono di restituire una "descrizione" in cui sono scritte tutte le caratteristiche della fonte di profitto mentre quelli dell'altra permettono di convertire il valore della fonte di profitto in una valuta estera passando come parametro il tasso di cambio. Le riserve d'oro hanno inoltre una costante (una variabile d'istanza final) statica che contiene il tasso di cambio dell'oro che ho messo come costante in quanto il valore dell'oro è considerato essere praticamente costante e ho voluto mantenere questa caratteristica.

Diagramma delle classi UML



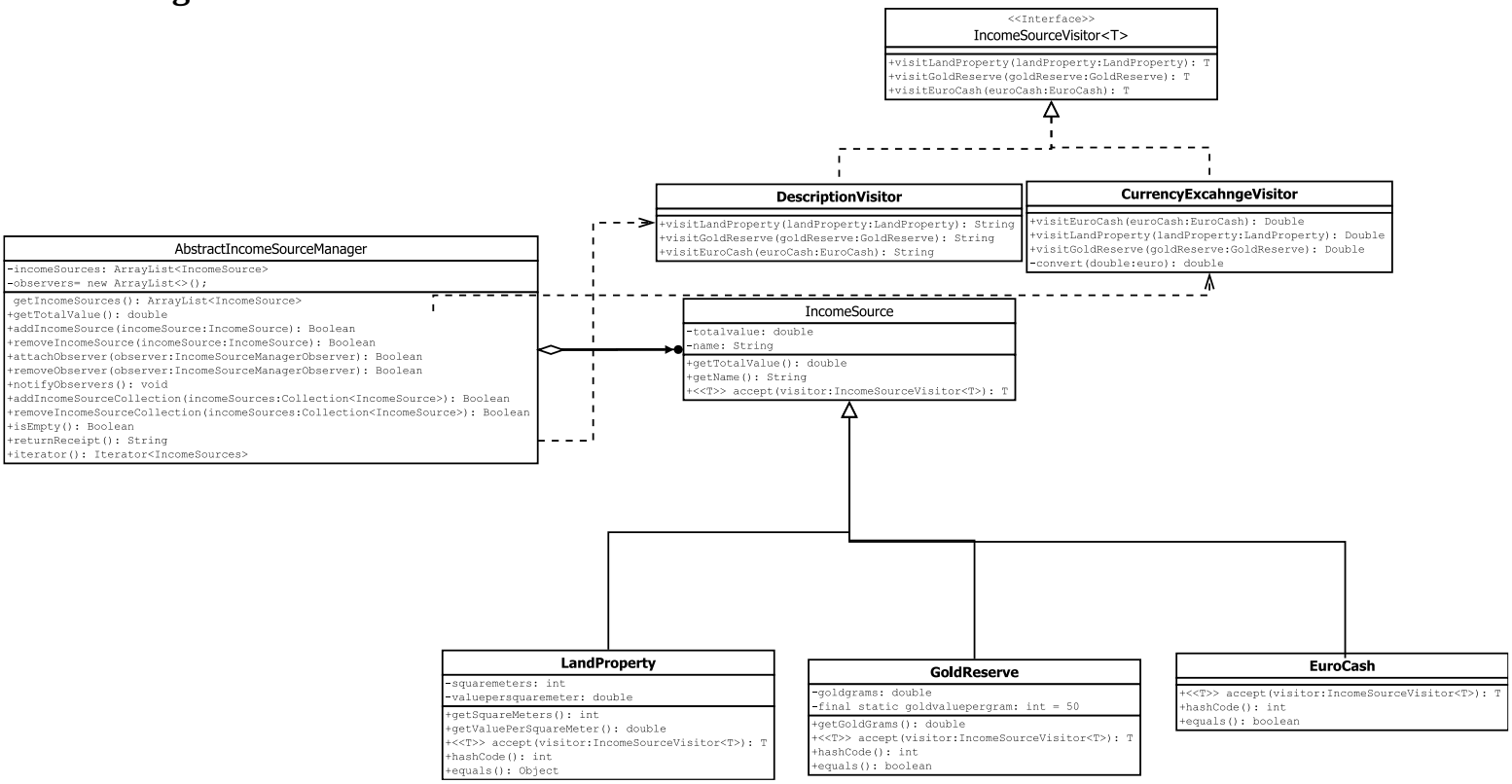
Realizzato con il Software DIA

Scelte di Design e strategie per testare

Come accennato in precedenza i pattern che ho utilizzato nel progetto sono:

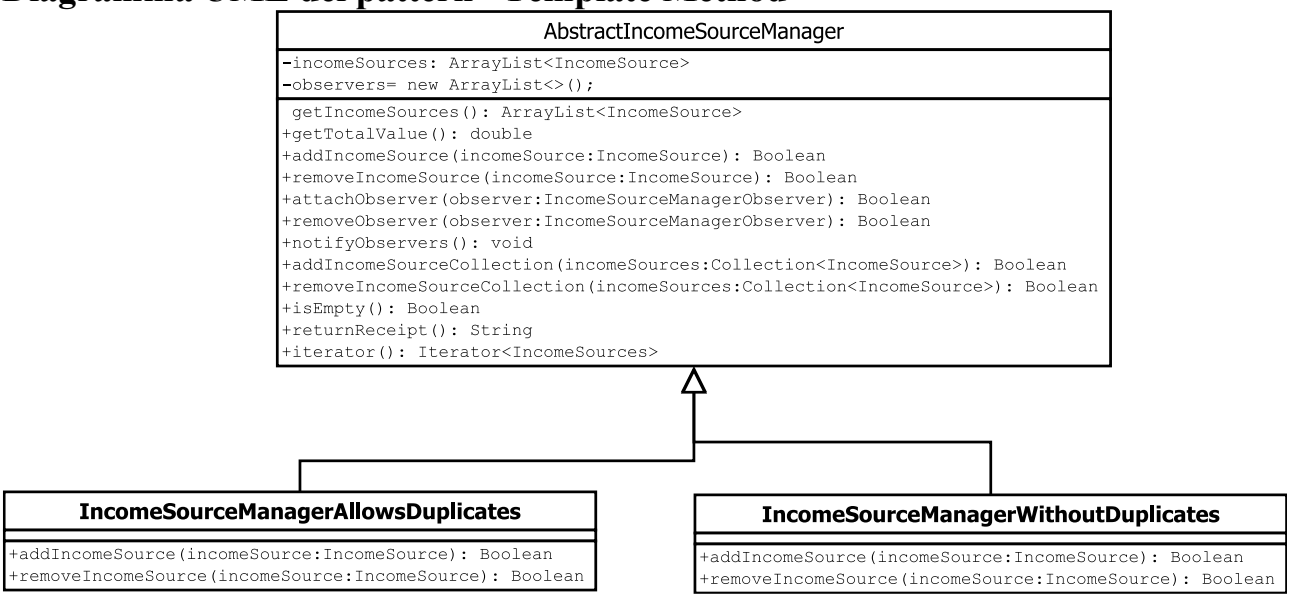
- Il **Visitor** per le fonti di profitto che mi permette di aggiungere in futuro altre operazioni sulle fonti di profitto senza dover modificare il codice ma solo estenderlo(rispettando così l' "Open Close Principle" che raccomanda di preferire l'estensione del codice alla sua modifica diretta), in particolare utilizzo la versione "generica" del Visitor, ovvero non "impongo" un tipo di ritorno specifico ai metodi della classe Visitor ma uso i tipi generici per lasciare libertà di scelta del tipo di ritorno al Client: questo permetterà di avere un'unica interfaccia per tutti i tipi di ritorno possibili invece di dover creare una nuova interfaccia (e di conseguenza anche un nuovo metodo accept)per ogni tipo di ritorno necessario. Inoltre oltre alle due già citate classi Visitor c'è n'è anche un'altra anche se è nidificata e privata, dentro la classe Observer per il catasto di cui parlerò in seguito, che si chiama "LandPropertySquareMeterVisitor" e che non appare nel diagramma UML delle classi in quanto classe interna privata e quindi "visibile" solo all'interno della classe Observer stessa.
  Purtroppo l'utilizzo del pattern Visitor rompe obbligatoriamente l'incapsulamento delle variabili di istanza delle mie entità semplici (le fonti di profitto) ma ho ritenuto che i vantaggi offerti dal Visitor fossero superiori agli svantaggi generati dalla rottura dell'incapsulamento.

Diagramma UML dei Visitor



- Il secondo pattern che ho usato è stato il **Template Method** che, come già anticipato, ho utilizzato per implementare sia il metodo per aggiungere che quello per rimuovere una collezione usando come metodi astratti quelli per aggiungere una singola fonte di profitto, questo genera due importanti benefici: il primo è il fatto che il Client può scegliere come inserire o rimuovere una fonte di profitto dal gestore (ad esempio se in futuro volessi un gestore che accetta solo fonti di profitto che hanno una determinata caratteristica oppure che lancia delle eccezioni in determinati casi sono libero di farlo semplicemente estendendo la classe del gestore astratto senza dover modificare il codice mentre la seconda è il fatto di poter avvisare i miei Observer come e quando preferisco, ad esempio poterei voler avvisare i miei osservatori solo al raggiungimento di una certa soglia di valore. Inoltre evito anche la duplicazione di codice e rispetto l'"Open Close Principle" in quanto favorisco l'estensione dei miei metodi rispetto alla loro modifica diretta.

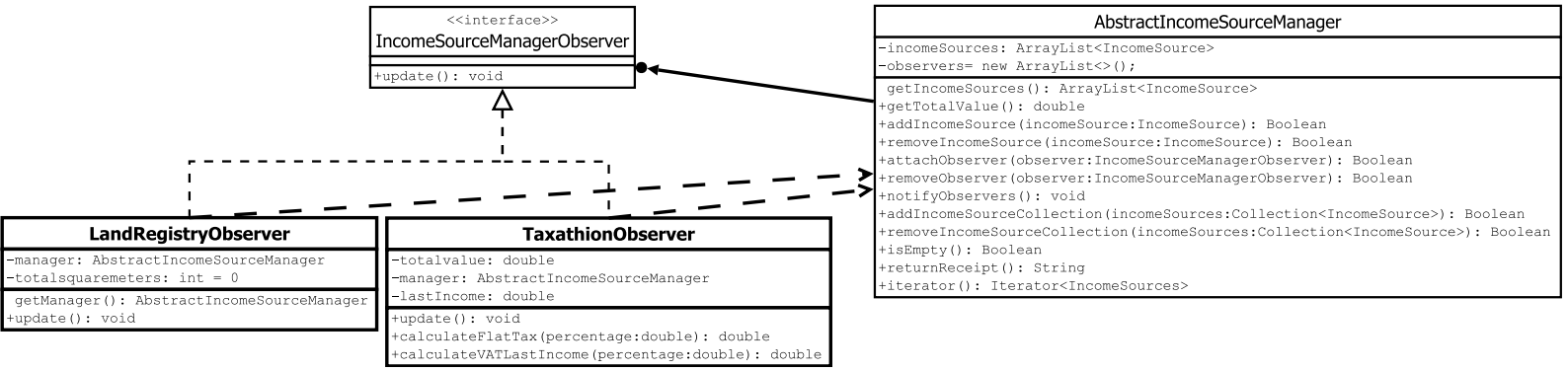
Diagramma UML del pattern “Template Method”



- Il terzo pattern che ho implementato è il pattern **Observer** che ho sfruttato per creare due classi concrete: una serve per aggiornare un eventuale ente catastale esterno, andando quindi a modificarsi realmente solo quando avviene una modifica nelle proprietà immobiliari mentre l’altra classe Observer è intesa per essere utilizzata da un ente fiscale esterno per calcolare sia un’eventuale “Flat Tax”, ossia una tassa che si basa su una percentuale fissa del valore del patrimonio totale che è passata come parametro al metodo di calcolo, che l’IVA (nel progetto è chiamata VAT all’inglese) sull’ultima aggiunta di una singola fonte di profitto, ossia calcola una percentuale, passata come parametro al metodo di calcolo, del valore dell’ultima fonte di profitto aggiunta al gestore osservato. I Vantaggi dell’utilizzo di questo pattern sono molteplici: innanzitutto faccio in modo che eventuali enti esterni possano essere aggiornati in tempo quasi reale delle modifiche alla mia collezione di fonti profitto per calcolare imposizioni fiscali o semplicemente per censire delle proprietà controllate dal gestore; inoltre implementando il pattern Observer ho anche il vantaggio di poter implementare in futuro anche un eventuale sistema per la rappresentazione grafica del valore del patrimonio controllato dal gestore il cui comportamento ho però voluto testare lo stesso implementando nella cartella dei test una classe Observer “mock”, ovvero che imita quello che sarebbe il comportamento di

un’eventuale classe per la rappresentazione grafica dei dati, che approfondirò maggiormente in seguito.

Diagramma UML degli Observer



- Infine ho anche sfruttato anche il pattern Iterator grazie al metodo iterator della classe del gestore astratto nella quale implemento però l’interfaccia Iterator offerta da Java stesso in quanto già completa di suo senza stare a creare una nuova interfaccia che sarebbe solo ridondante, facendo così in modo che restituisca le fonti di profitto in ordine alfabetico per comodità, attraverso una classe anonima che faccio ritornare al metodo iterator.

Metodi per testare

Per testare il mio codice ho usato la libreria AssertJ in quando risulta sia molto più facile da usare per i test rispetto ai metodi standard di Junit sia molto più chiaro e comprensibile alla semplice lettura.

Non ho testato le fonti di profitto in quanto sono entità semplici senza praticamente alcuna logica se escludiamo i metodi equals, hashcode ed i getter che sono generati dall’IDE ed il metodo accept per i Visitor che verrà testato insieme alle classi dei Visitor.

Inoltre come già accennato in precedenza ho voluto includere nella cartella dei test anche la classe Observer per imitare un’eventuale classe per la rappresentazione dei dati provenienti dal gestore che ho poi testato in una classe a parte nella stessa cartella.

Infine ho anche aggiunto dei getter con visibilità di pacchetto per poter testare con più facilità il codice come ad esempio il getter per la lista delle fonti di profitto del gestore che ho usato al posto di usare l'iterator che è inteso per essere utilizzato dai Client.

Lista Design Pattern usati nel progetto

I Pattern che usato nel progetto sono:

- **Visitor**
- **Template Method**
- **Observer**
- In parte anche Iterator (si legga sopra)