PATTERN MEMENTO

IMPLEMENTAZIONE IN ASPECTJ

JACOPO CARRAVIERI
CORSO DI TECNICHE SPECIALI DI PROGRAMMAZIONE A.A. 2017-2018
UNIVERSITÀ DEGLI STUDI DI MILANO

Indice

Introduzione 2
Il Pattern 2
Implementazione 3
Salvataggio dei Memento 3
Trigger 3
Originator multipli 4
Caretaker multipli 4
Struttura finale 5
Classi e Interfacce 5
Aspetti 5
Come usare il pattern 6
Compilazione 6
Classe Memento 6
Interfaccia Originator 6
Interfaccia Caretaker 6
Salvataggio dei Memento 6
Esempio 1: Editor di testo multitab 7
Diagramma delle classi 8
Diagrammi di sequenza 8
Esempio 2: Disegno di diagrammi (dal libro) 10
Modifiche 10
Diagramma delle classi 12
Esempio 3: History ad albero 13
Conclusioni 14

Introduzione

Nel contesto del corso di Tecniche Speciali di Programmazione, tenuto dal prof. Walter Cazzola presso l'Università degli Studi di Milano (a.a. 2017-2018) è stato richiesto agli studenti di produrre un'implementazione il più possibile generica e versatile di un pattern, nel mio caso il Memento. Tale progetto era da svolgere per mezzo del framework per l'AOP AspectJ, promuovendo in questo modo il pattern a concetto trasversale.

La presente documentazione illustra le scelte implementative, le difficoltà tecniche riscontrate e le soluzioni adottate e mostra tre esempi creati per verificare funzionamento e utilità del pattern così implementato.

Il Pattern

Al fine di decidere concretamente come implementare il pattern si è inizialmente esaminata la sua struttura di riferimento, illustrata dalla figura seguente.

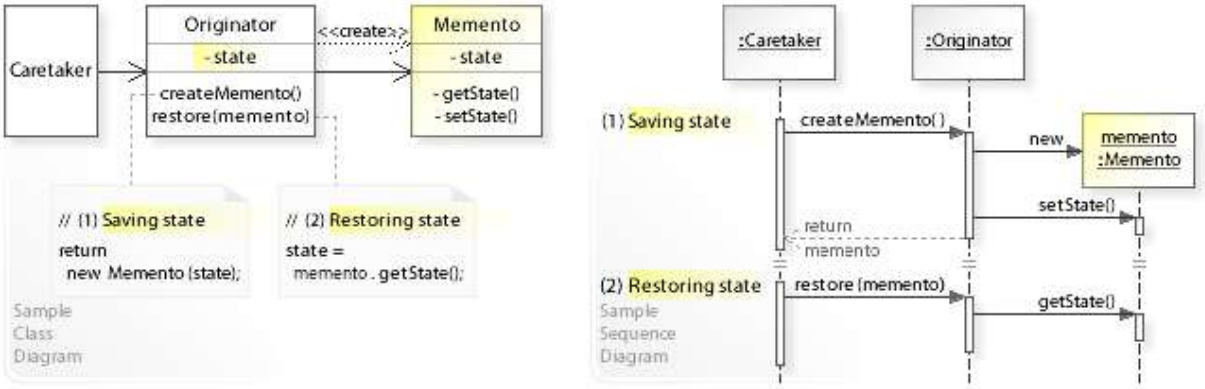


Figura 1 Diagramma delle classi e di sequenza del pattern Memento

Nella sua forma di base il pattern è organizzato nei seguenti elementi:

- **Originator**: si tratta dell'oggetto dotato di uno stato interno che deve essere salvato ed eventualmente ripristinato per mezzo degli oggetti Memento. Si può notare in particolare dal diagramma di sequenza come si occupi egli stesso, dopo l'invocazione di `createMemento`, di esternalizzare il suo stato. In questo modo il suo funzionamento interno non deve essere conosciuto dalle altre classi (se non appunto da Memento) e dunque lo stato è opaco e la sua implementazione può cambiare nel tempo.

- **Memento**: è la classe le cui istanze rappresentano lo stato dell'Originator in un determinato istante di tempo. È conservata ed eventualmente utilizzata in forma pubblica dal Caretaker, ha accesso ai membri privati dell'Originator. Quest'ultima relazione può essere vera anche in senso inverso.

- **Caretaker**: è l'entità che si occupa di manipolare l'Originator e produrre quando necessario gli oggetti Memento. La sua definizione è molto generica in quanto dipende fortemente dalla specifica applicazione: ad esempio in una GUI può essere identificato come l'applicazione che riceve gli input dall'utente ed in risposta modifica i componenti, salvandone al tempo stesso lo stato.

Implementazione

Al fine di usare concretamente il pattern (in particolare nelle applicazioni di esempio facenti parte della consegna) si è reso necessario ampliare e definire nel dettaglio la sua struttura. Arrivare alla versione finale è stato un processo graduale, frutto di tentativi e miglioramenti, ed è andato di pari passo con lo sviluppo dei programmi. Verranno di seguito illustrate le aggiunte che si sono rese necessarie.

Salvataggio dei Memento

Il pattern non si occupa di definire come gli oggetti Memento debbano essere memorizzati. Dopo aver analizzato i casi di studio si è deciso di introdurre il concetto di History. Essa altro non è che una lista di stati consecutivi che può essere attraversata dal Caretaker in maniera simile a quanto avviene nel pattern Iterator. Qui però il cursore giace sull'elemento corrente anziché tra gli elementi, come invece avviene nel caso di un iteratore: questo perché un componente è sempre in un determinato stato e non ha senso dire che lo stesso si trovi a cavallo tra due stati.

Nel corso dello sviluppo del terzo ed ultimo esempio infine la History è stata evoluta in modo da poter tenere traccia delle differenti diramazioni o linee temporali. In questo modo è diventato possibile in ogni momento esplorare tutte le modifiche avvenute nel documento da quando esso è stato creato, senza che nessuna di esse fosse scartata.

Trigger

Non è sempre possibile ripristinare lo stato semplicemente invocando `Originator.restore(Memento)` come vorrebbe la versione base del pattern; questa operazione è stata dunque resa facoltativa. Si consideri il programma per il disegno dei diagrammi: per tornare allo stato precedente `s0` dallo stato corrente `s1` bisogna annullare l'ultimo comando eseguito invocando `s1.unexecute()`, con una chiamata del tipo `restore(s0)` ciò non sarebbe possibile.

Si è deciso dunque di impiegare il concetto di trigger: scorrendo la History i metodo appropriati vengono di volta in volta invocati sugli oggetti Memento. Le callback utilizzabili dalle sottoclassi di Memento sono le seguenti:

- `onAddToHistory` L'oggetto è stato aggiunto alla history come nuovo stato.
- `onEnter` Ingresso nello stato in seguito ad un movimento nella history.
- `onEnterFromNext` Ingresso nello stato in seguito ad una operazione di undo.
- `onEnterFromPrevious` Ingresso nello stato in seguito ad una operazione di redo.
- `onExit` Uscita dallo stato in seguito ad un movimento nella history.
- `onExitToPrevious` Uscita dallo stato in seguito ad una operazione di undo.
- `onExitToNext` Uscita dallo stato in seguito ad una operazione di redo.

Originator multipli

Il supporto a tab multiple nel primo programma di esempio impone che il Caretaker (l'applicazione Swing) sia in grado di gestire più di un Originator (le tab) e che possieda inoltre una nozione di Originator corrente o attivo. Come conseguenza l'applicazione si trova a dover gestire tanti oggetti History quanti sono gli Originator, per questo motivo ogni History è stata associata al suo Originator.

In un primo momento questo avveniva esplicitamente in `CaretakerAspect` per mezzo di un'istanza di `Map<Origintor,History>`. Purtroppo questa soluzione imponeva, per evitare memory leak, di rimuovere esplicitamente il mapping ogni volta che un oggetto Originator non fosse più necessario (ovvero nel caso molto comune in cui una tab viene chiusa dall'utente). Java è provvisto di un garbage collector, e trattandosi questa di una soluzione 100% Java (dove non sono in gioco librerie native) ciò è stato giudicabile non accettabile.

Non è stato possibile risolvere il problema nemmeno con una `WeakHashMap`: la History contiene i Memento, e gli oggetti Memento conservano spesso una strong reference al loro Originator. Indirettamente quindi History, il valore mappato, ha una strong reference verso Originator, la chiave. Come si evince dalla documentazione ufficiale della classe `WeakHashMap` ciò ne impedisce il normale funzionamento.

History è stato dunque spostato in un nuovo aspetto `OriginatorAspect` che, agganciato all'istanza di Originator per mezzo del modificatore `perthis`, ne condivide il ciclo di vita: quanto una tab viene rimossa dalla GUI ed il conteggio delle referenze scende a zero, è eliminata e così anche la History.

Caretaker multipli

Si è ipotizzato che determinate applicazioni potessero aver bisogno di più istanze attive di Caretaker: un esempio può essere un server che esegue dei comandi per conto dei client. Ogni Thread associato ad un client è un Caretaker, questo perché ogni client deve avere la sua History dei comandi (o le sue History, nel caso la specifica applicazione preveda anche più Originator) tramite la quale poter eventualmente effettuare operazioni di rollback e commit in maniera totalmente indipendente.

Un esempio di questo tipo potrebbe essere foglio di calcolo HTML/AJAX con supporto a sheet multipli ognuno dotato della propria History. Ogni cambiamento è istantaneamente salvato sul server in maniera simile a quando avviene lato client in un normale editor. Il Thread sul server associato al client (o più nello specifico alla tab nel browser del client che contiene l'istanza del programma) è un Caretaker mentre ogni sheet nell'editor è mappato sul server ad un Originator.

Questo caso di esempio non è stato implementato, in quanto la dimensione del progetto, anche solo per dimostrarne il funzionamento basilare, sarebbe stata eccessiva. Tuttavia, il caso di Caretaker multipli è stato simulato con successo istanziando tutti e tre gli esempi descritti nella presente documentazione nello stesso `main`. Questo funziona grazie al fatto che, in maniera simile al caso di Originator multipli, `CaretakerAspect` è istanziato e legato all'istanza di Caretaker.

Struttura finale

Il pattern così come è stato implementato è descritto dal diagramma seguente:

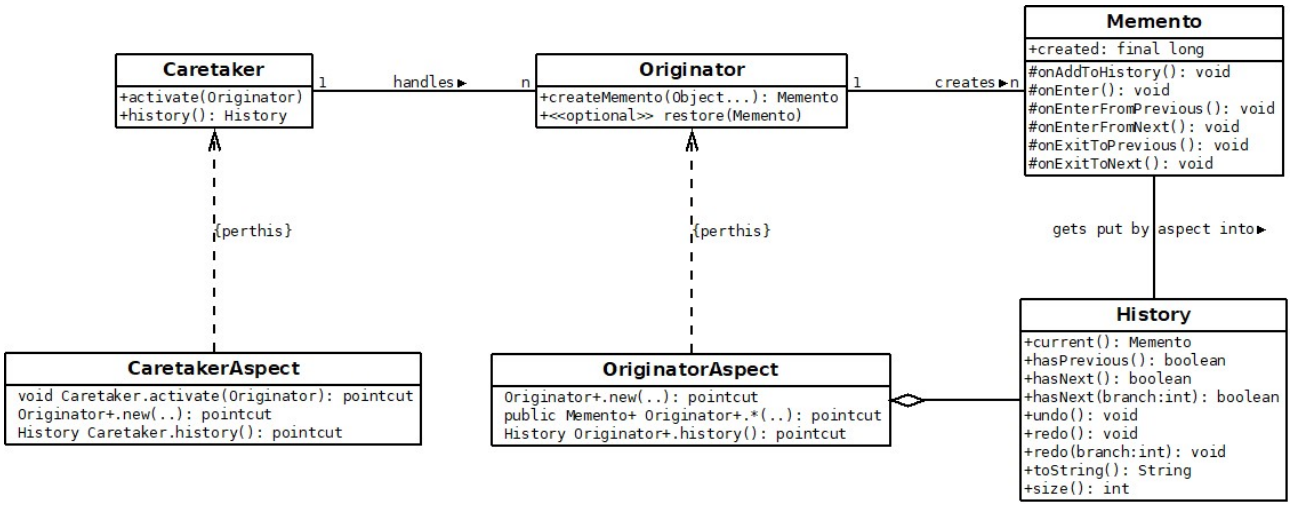


Figura 2 Diagramma delle classi e degli aspetti dell'implementazione

Classi e Interfacce

Si invita a fare riferimento alla Javadoc.

Aspetti

Il primo aspetto CaretakerAspect si lega alle istanze delle classi che implementano l'interfaccia Caretaker, intercettando le chiamate ai metodi di quest'ultima e fornendone la logica. Mantiene il concetto di Originator corrente, nel caso il programma ne preveda molteplici. Possiede tre pointcut:

- Esecuzione del costruttore di una classe che implementa Caretaker
- Attivazione di un Originator specifico
- Richiesta dell'oggetto History

L'aspetto OriginatorAspect si lega invece alle istanze delle classi che implementano Originator e serve per associarvi un oggetto History e fornire anche qui la logica per i metodi dell'interfaccia. Possiede anch'esso tre pointcut:

- Esecuzione del costruttore di una classe che implementa Originator
- Chiamata ad un metodo pubblico della stessa che ritorna un oggetto Memento
- Richiesta della History (da parte di CaretakerAspect)

Come usare il pattern

Per utilizzare il pattern all'interno della propria applicazione è necessario copiarvi il package di sorgenti `ch.jacopoc.memento`. Occorre poi identificare nell'applicazione le tre entità coinvolte ed associarvi le relative interfacce (Caretaker e Originator) e classi (Memento). Non è necessario estendere alcun aspetto.

Compilazione

Si può compilare la propria applicazione con `ajc -1.8` e la consueta lista di file. In alternativa è possibile far uso (su Linux) del sistema di build sviluppato ad-hoc per lo svolgimento del progetto:

1. Copiare tutti i file/package dell'applicazione nella directory `src/`
2. Invocare `./compile.sh <outpath/outfile.jar> <[package.]MainClass>`

Nella directory `outpath` saranno prodotti: il file `outfile.jar` dell'applicazione, due script di avvio (per Windows e Linux) e copia di tutte le librerie di runtime che sono state trovate nella cartella `src/` stessa. L'esempio è compilabile con `./compile.sh ../bin/out.jar main.Main`.

Classe Memento

Estendere questa classe astratta definendo lo stato da salvare nonché implementando i trigger necessari per il ripristino dello stesso. I Trigger sono eseguiti dalla History quando è attraversata tramite le chiamate ad `undo` o `redo`. Spesso conveniente che sia una nested class dell'Originator, così da poter accedere direttamente allo stato privato di quest'ultimo e viceversa.

Interfaccia Originator

Implementare l'interfaccia usando come template argument la classe del punto precedente. È necessario fornire una implementazione valida del metodo `createMemento(Object…)`, mentre il metodo `restore(T)` è facoltativo e può essere utilizzato, se necessario, in combinazione ai trigger. Il metodo `history()` è invece privato ed usato internamente dal pattern.

Quando viene invocato un costruttore di una sottoclasse di Originator, viene anche istanziata la History corrispondente in `OriginatorAspect`. Quest'ultima richiede uno stato iniziale, perciò in questa fase il metodo `createMemento(Object…)` viene invocato una prima volta. Gli argomenti che sono stati passati al costruttore vengono riutilizzati in questa invocazione, così da poter essere eventualmente utilizzati per la creazione del primo stato (come nell'editor di testo di esempio).

Interfaccia Caretaker

Occorre infine implementare l'interfaccia Caretaker. La classe che la implementa è quella che crea e manipola gli oggetti Originator. Non è necessario implementare alcun metodo in quando il loro funzionamento è garantito da `CaretakerAspect`. L'interfaccia espone due metodi: `activate(Originator)`, che può essere utilizzato in un contesto multi-Originator per selezionare quello attivo (altrimenti il pattern provvedere ad attivare automaticamente ogni oggetto Originator appena costruito) ed `history()` che recupera e ritorna l'oggetto History associato all'Originator attivo.

Salvataggio dei Memento

I Memento vengono automaticamente intercettati e salvati nella History ogni volta che un metodo pubblico dell'Originator, che ritorna Memento o una sua sottoclasse, è invocato dal Caretaker. Pertanto nel o nei punti dell'applicazione in cui è necessario salvare lo stato è sufficiente far uso di un'istruzione del tipo:

```
originator.createMemento(…);
```

Esempio 1: Editor di testo multitab

Il primo esempio di applicazione del pattern è un editor di testo con supporto per le tab. Gli editor di testo sono un tipo di programma estremamente diffuso, nonché l'esempio forse più comune di utilizzo del pattern memento: questo perché deve essere sempre possibile annullare e ripristinare le modifiche effettuate al documento, fino a tornare eventualmente allo stato iniziatale.

Ogni tab rappresenta un documento e ha una propria storia dei cambiamenti. Questo ha richiesto l'ampliamento della struttura base del pattern, in modo che fosse possibile associare ad un Caretaker (l'applicazione principale scritta in Java Swing) più Originator (le JTextArea contenute nelle tab) nonché selezionarne uno come attivo per salvare gli oggetti Memento nella History corretta.

La prima implementazione degli oggetti Memento salvava l'intero contenuto dell'editor ad ogni modifica, in seguito però per rendere il caso di studio più realistico gli oggetti Memento sono stati modificati in modo da contenere solo il delta rispetto allo stato precedente. Infine si è voluto considerare anche il caso in cui lo stato iniziale fosse diverso dall'editor vuoto, come per esempio quando si apre un file esistente.

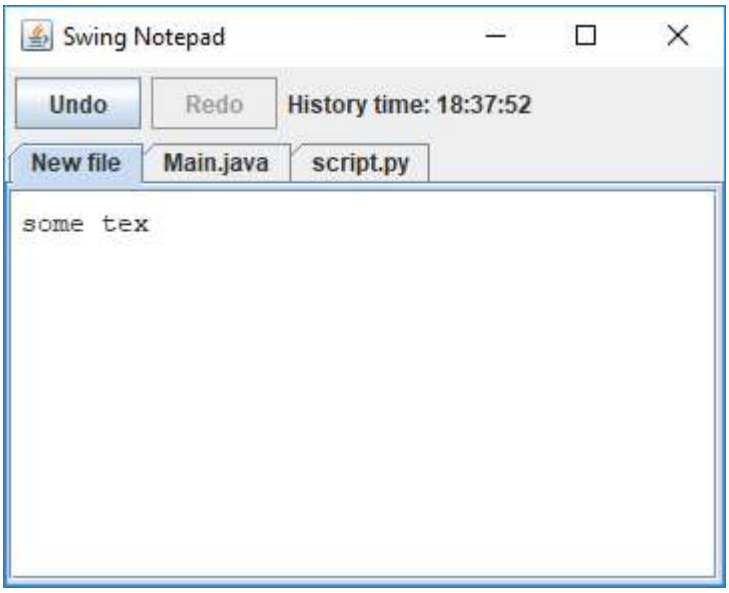
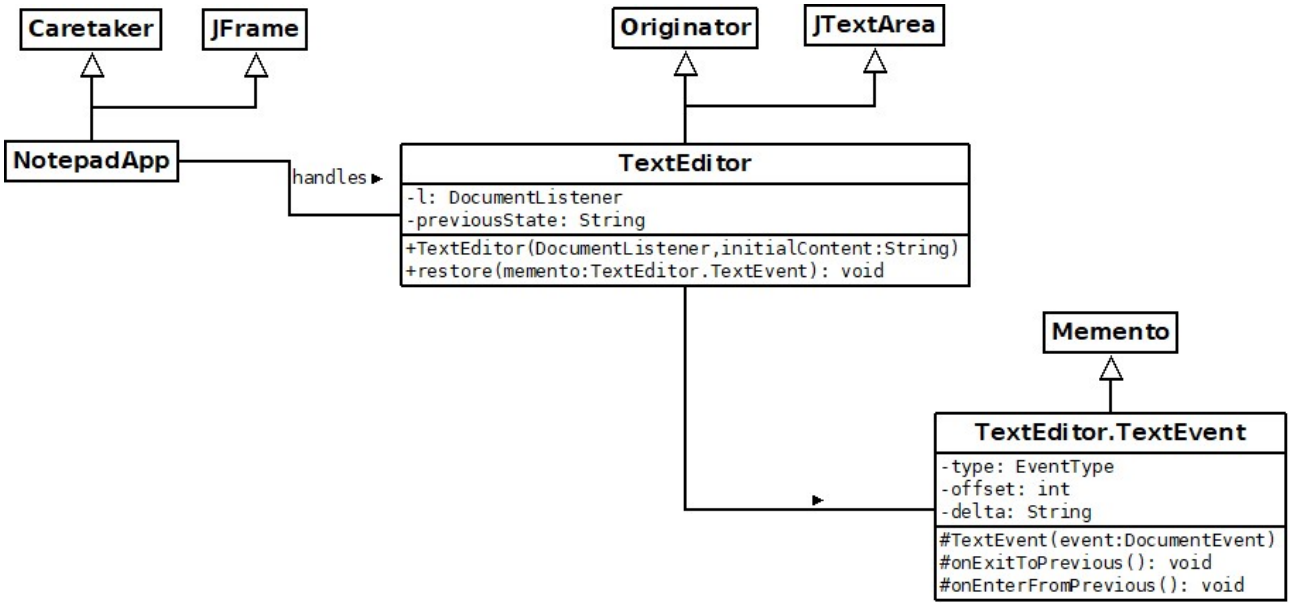


Figura 3 Editor di testi in esecuzione

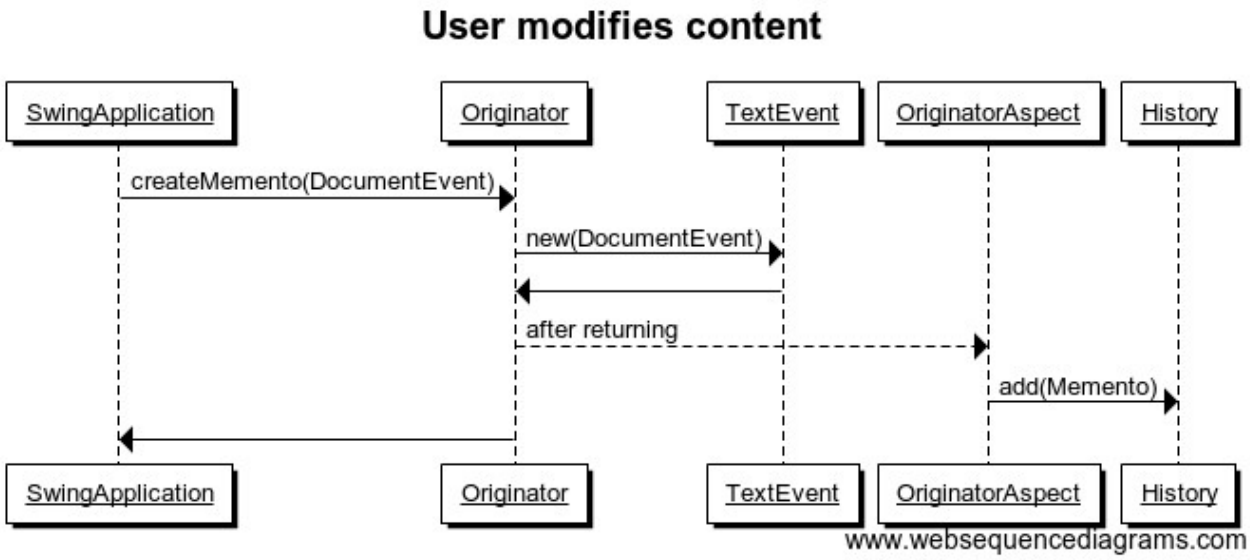
```
[+'', {+'s'}]
[+'', +'s', {+'o'}]
[+'', +'s', +'o', {+'m'}]
[+'', +'s', +'o', +'m', {+'e'}]
[+'', +'s', +'o', +'m', +'e', {+' '}]
[+'', +'s', +'o', +'m', +'e', +' ', {+'t'}]
[+'', +'s', +'o', +'m', +'e', +' ', +'t', {+'e'}]
[+'', +'s', +'o', +'m', +'e', +' ', +'t', +'e', {+'x'}]
[+'', +'s', +'o', +'m', +'e', +' ', +'t', +'e', +'x', {+'t'}]
[+'', +'s', +'o', +'m', +'e', +' ', +'t', +'e', +'x', +'t', {+'<br/>'}]
[+'', +'s', +'o', +'m', +'e', +' ', +'t', +'e', +'x', {+'t'}, +'<br/>']
[+'', +'s', +'o', +'m', +'e', +' ', +'t', +'e', {+'x'}, +'t', +'<br/>']
```

Figura 4 Stati della History

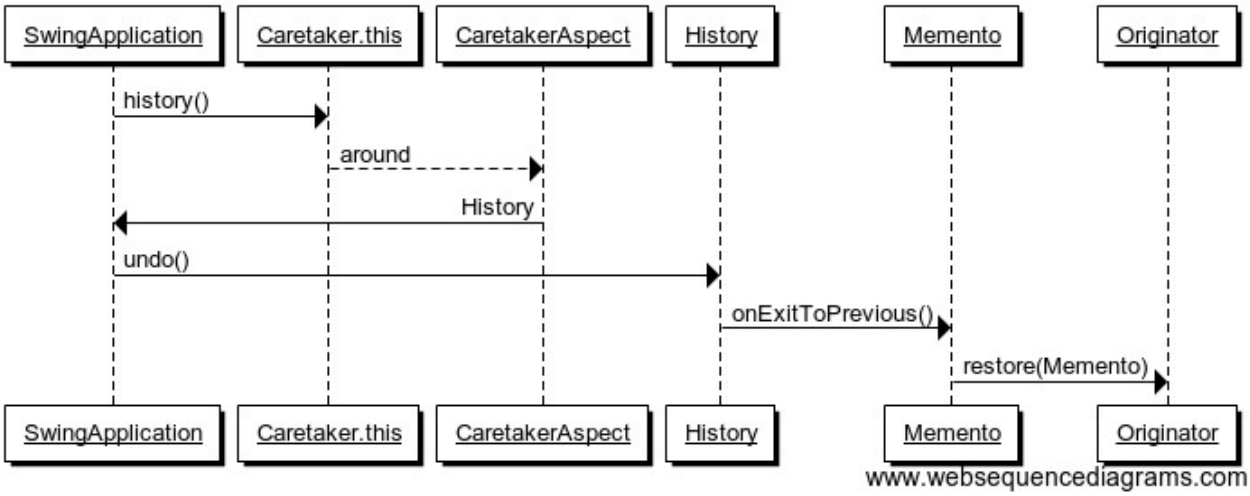
Diagramma delle classi



Diagrammi di sequenza



User clicks undo



Esempio 2: Disegno di diagrammi (dal libro)

Parte dei requisiti era di produrre ed applicare il pattern ad un esempio contenuto nel libro "Design Patterns", nel capitolo relativo allo specifico pattern. Per quanto riguarda il pattern memento l'esempio proposto è un software per il disegno di diagrammi ed in particolare le componenti che si occupano di risolvere i vincoli matematici per collegare i blocchi tra loro. La struttura così come proposta dal libro è illustrata dal seguente diagramma delle classi:

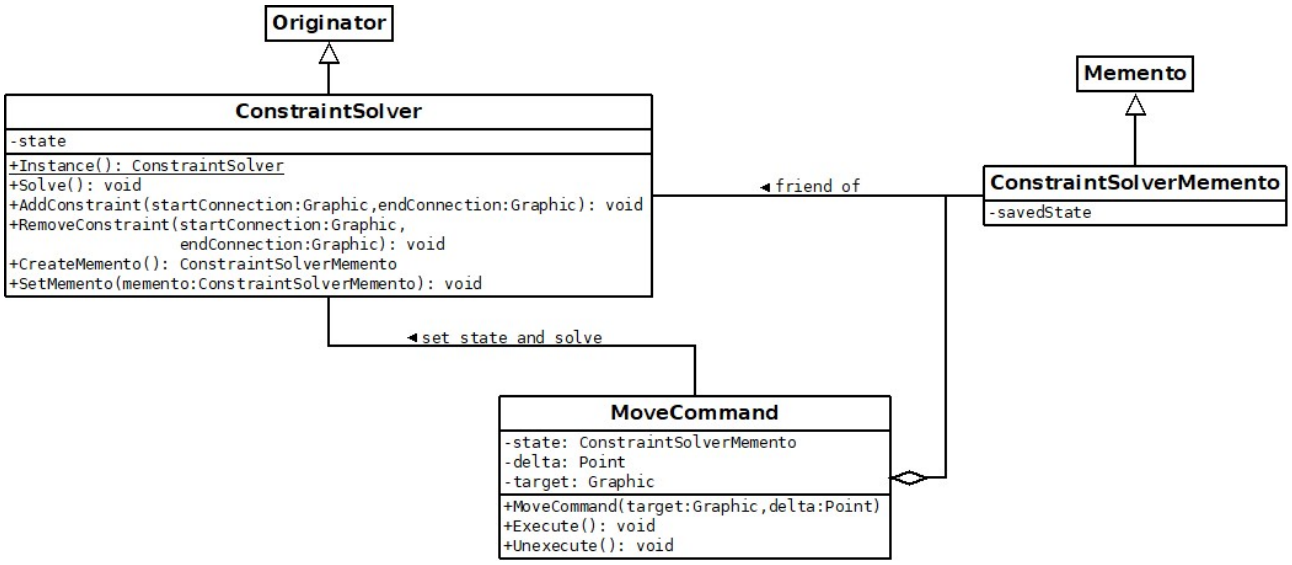


Figura 5 Esempio dal libro

Modifiche

Prima di poter procedere ad implementare il pattern si è resa necessaria un'operazione di refactoring. Il libro propone il `ConstraintSolver` come unico Originator e dunque come unica entità il cui stato deve essere salvato. Tuttavia, come si può evincere dallo pseudocodice fornito dal libro stesso:

```
void MoveCommand::Execute () {

        ConstraintSolver* solver = ConstraintSolver::Instance();
        _state = solver->CreateMemento();
        // create a memento
        _target->Move(_delta);
        solver->Solve();

}
```

Non tutto lo stato interno è conosciuto al `ConstraintSolver`: si pensi ad esempio all'esecuzione di `MoveCommand` su un oggetto privo di constraint . Tuttavia anche questo è un comando di cui è necessario tenere traccia. Inoltre, lo stato del `ConstraintSolver` in un determinato instante di tempo non è il semplice stato interno (che in realtà è lo stato precedente) bensì la somma di quest'ultimo e gli effetti dell'invocazione `solver->Solve()`.

Al fine di risolvere le questioni sollevate nel paragrafo precedente si è deciso di introdurre una classe `DiagramEditor` che conosca tutto lo stato (ovvero `ConstraintSolver` ed esistenza e posizione di tutte le istanze di `Graphic`) e rendere la classe `Command` sottoclasse di Memento.

Infine si è implementato il tutto sotto forma di prototipo funzionante. Esso esegue in fase di inizializzazione alcuni comandi, simulando le azioni di un utente. Lo scopo di questo prototipo è dimostrare che è possibile navigare liberamente nella History mantenendo i collegamenti consistenti con la posizione dei blocchi.

I collegamenti vengono ricalcolati ad ogni spostamento, ma solo quelli che collegano il blocco effettivamente spostato che vengono dunque all'occorrenza invalidati. L'aggiunta stessa di blocchi o collegamenti è a sua volta un comando.

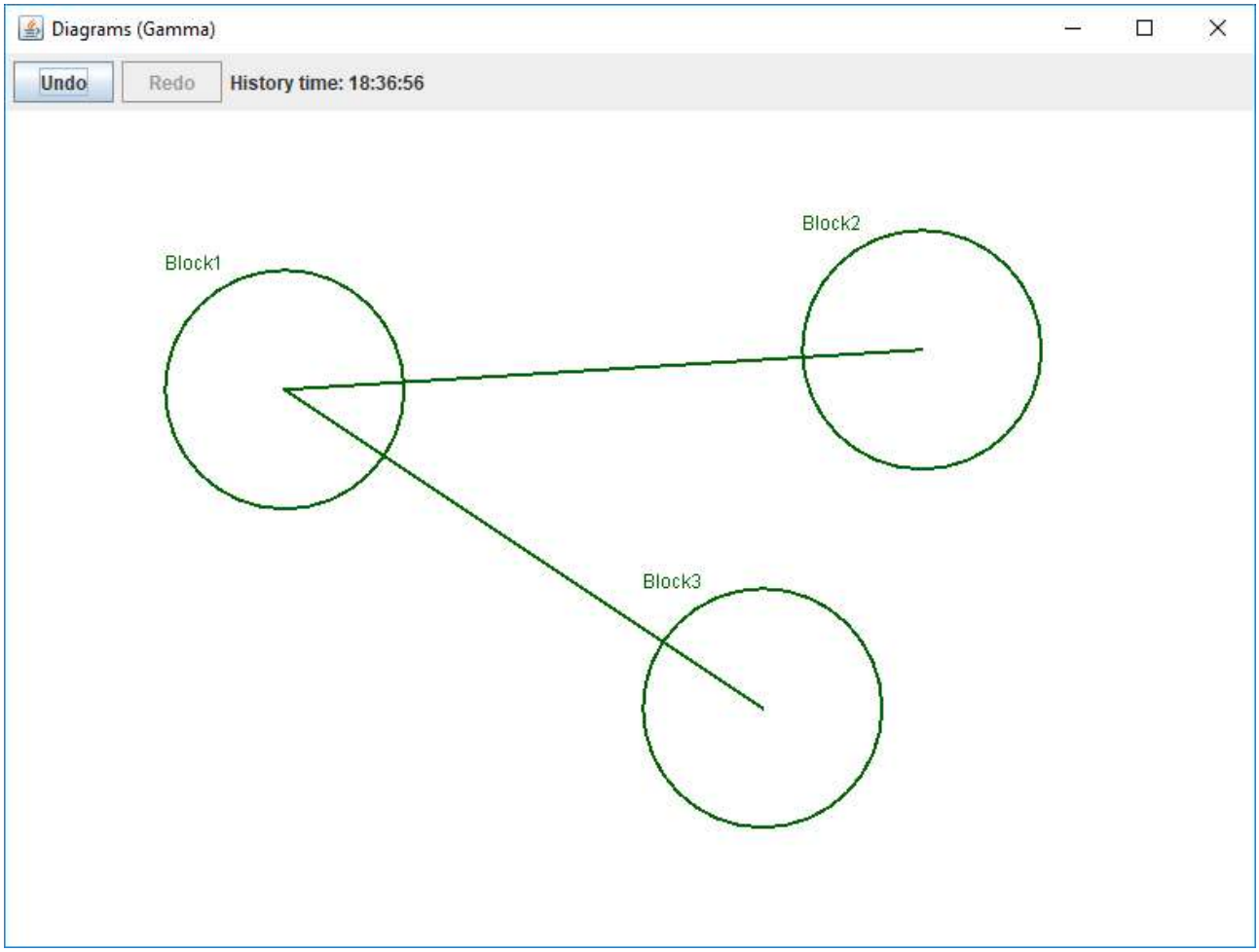


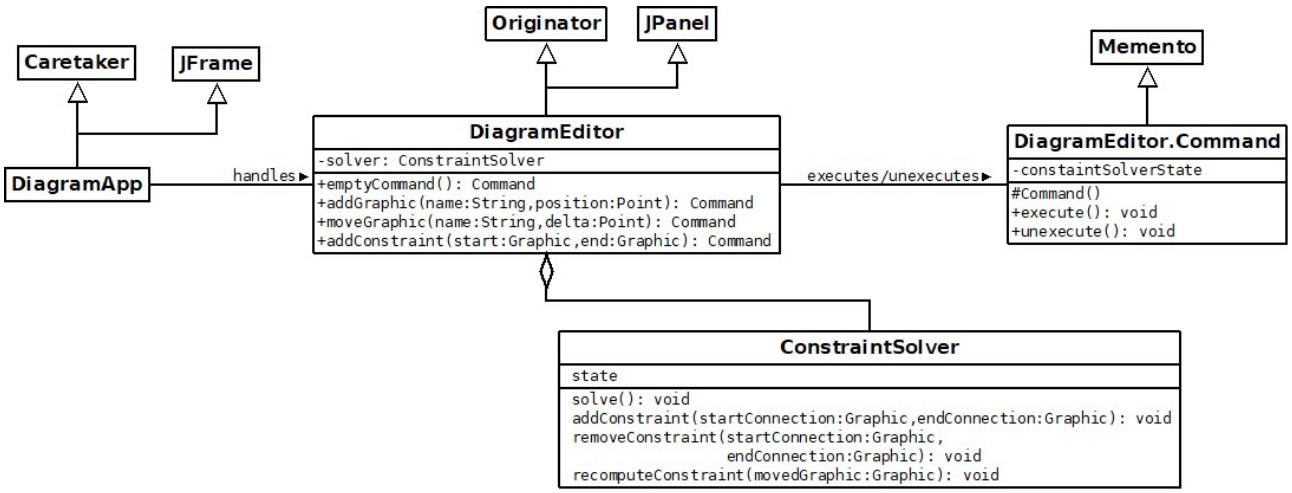
Figura 6 L'applicazione in esecuzione

```
[EmptyCommand, AddGraphic, AddGraphic, AddConstraint, AddGraphic, {AddConstraint}, MoveGraphic]
[EmptyCommand, AddGraphic, AddGraphic, AddConstraint, {AddGraphic}, AddConstraint, MoveGraphic]
[EmptyCommand, AddGraphic, AddGraphic, {AddConstraint}, AddGraphic, AddConstraint, MoveGraphic]
[EmptyCommand, AddGraphic, AddGraphic, AddConstraint, {AddGraphic}, AddConstraint, MoveGraphic]
[EmptyCommand, AddGraphic, AddGraphic, AddConstraint, AddGraphic, {AddConstraint}, MoveGraphic]
[EmptyCommand, AddGraphic, AddGraphic, AddConstraint, AddGraphic, AddConstraint, {MoveGraphic}]
```

Figura 7 Stati della History in seguito ad undo/redo

Diagramma delle classi

La struttura finale del programma diventa:



Esempio 3: History ad albero

Il terzo ed ultimo esempio è servito per testare che la logica di History fosse in grado di gestire correttamente il diramarsi della stessa. Questo succede quando si torna indietro ad uno stato precedente e da lì si procede a produrre nuovi stati: una diramazione più recente dev'essere creata. Si è deciso di organizzare questo terzo esempio come test percorrendo le diverse diramazioni e controllando la correttezza dello stato corrente.

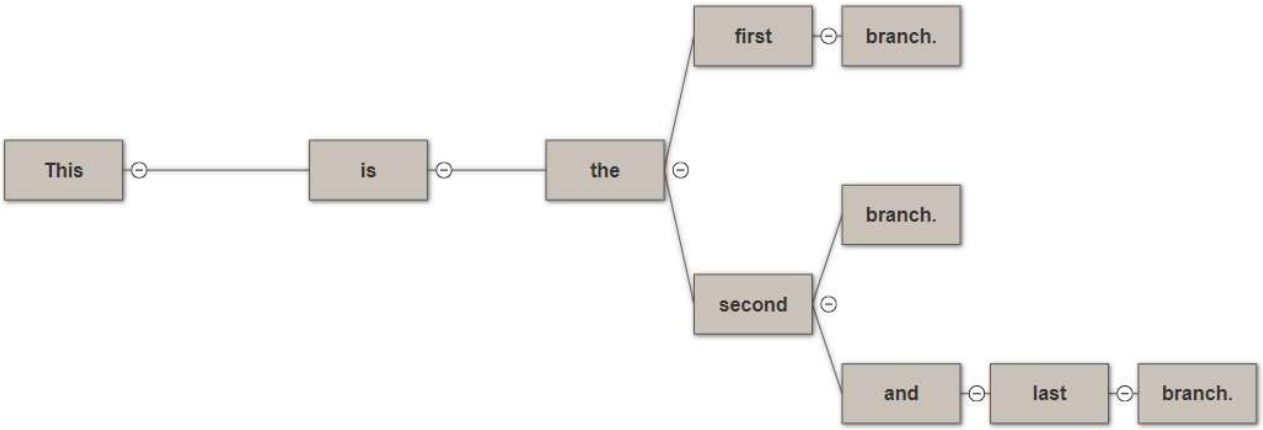


Figura 8 History di test

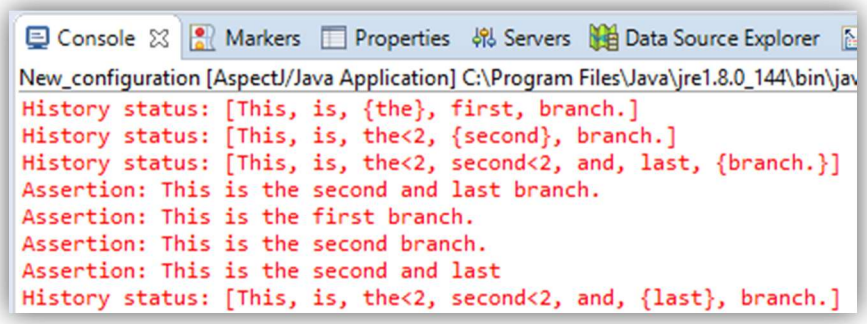


Figura 9 Output del programma

Conclusioni

Il pattern memento si è rivelato particolarmente adatto ad essere implementato in AspectJ. In particolare i primi due esempi mostrano come il suo utilizzo sotto forma di concetto trasversale sia efficace ed efficiente anche per applicazioni tra loro diverse.

Poter impiegare AspectJ su dei casi concreti è stato particolarmente istruttivo: spesso infatti lo stesso viene trattato solo in teoria e questo non permette di comprenderlo appieno. Seppur le applicazioni siano solo degli esempi, sono prototipi completamente funzionanti e potenzialmente estendibili.

Il dover lavorare su un esempio esistente è stato ugualmente interessante: ha richiesto di fare correttamente refactoring di codice esistente da un lato e di ideare l'implementazione del pattern in modo che fosse sufficientemente flessibile per adattarsi dall'altro.