Il pattern Observer

a cura di **Angelo Furfaro**
da “Design Patterns”, Gamma et al.

Dipartimento di
Ingegneria Informatica, Elettronica, Modellistica e Sistemistica
Università della Calabria, 87036 Rende(CS) - Italy
Email: a.furfaro@unical.it
Web: http://angelo.furfaro.dimes.unical.it

Observer

Classificazione

- Scopo: comportamentale
- Raggio d'azione: oggetti

Altri nomi

Dependents, Publish-Subscribe

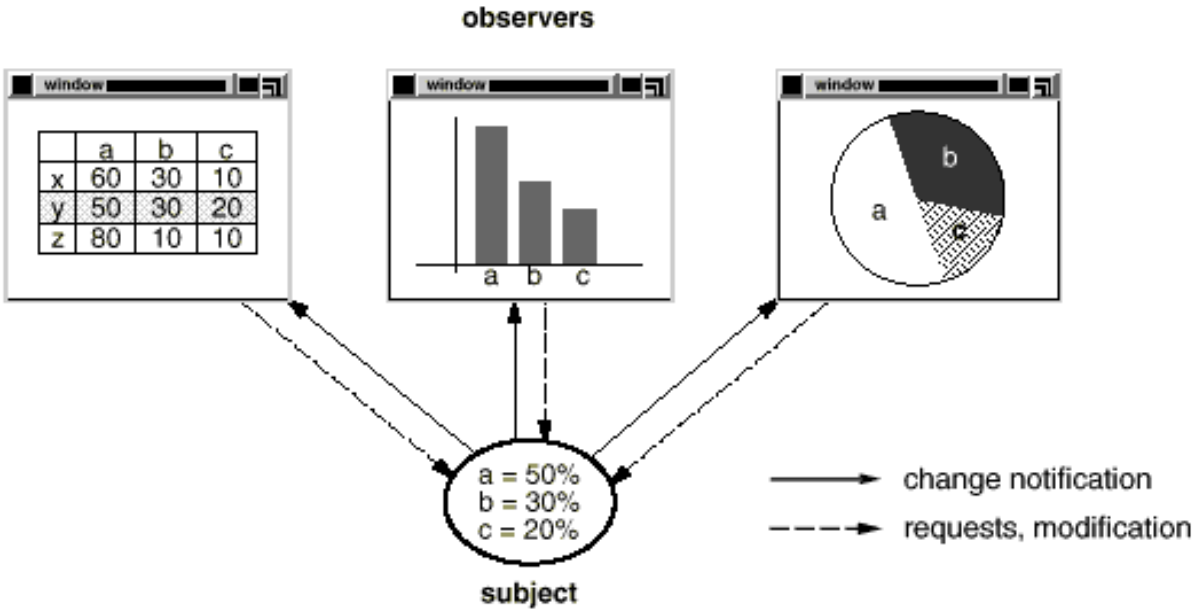
Scopo

- Definire una dipendenza uno a molti tra oggetti, in modo tale che se un oggetto cambia il suo stato, tutti gli oggetti dipendenti da questo siano notificati e aggiornati automaticamente.

Problema

- Un effetto collaterale comune al partizionare un sistema in insiemi di classi collaboranti è il bisogno di mantenere un alto livello di consistenza tra classi correlate.
- La consistenza non deve essere ottenuta a discapito del livello di accoppiamento, che deve rimanere basso, altrimenti si avrebbe un impatto negativo sulla riusabilità delle classi che sarebbe ridotta.

Esempio



- Molti toolkit grafici per la realizzazione di GUI separano gli aspetti relativi alla rappresentazione grafica rispetto alla gestione dei dati dell'applicazione
- Le classi che definiscono i dati dell'applicazione e quelle che curano gli aspetti relativi alla presentazione possono essere riutilizzati indipendentemente
- Gli stessi dati possono essere rappresentati tramite differenti oggetti grafici (ad es. un foglio di calcolo, un istogramma, un diagramma a torta), anche contemporaneamente
- Gli oggetti grafici non si conoscono reciprocamente ma se l'utente cambia, tramite uno di essi, i dati, tale cambiamento si riflette su ciascuno di essi.

Esempio

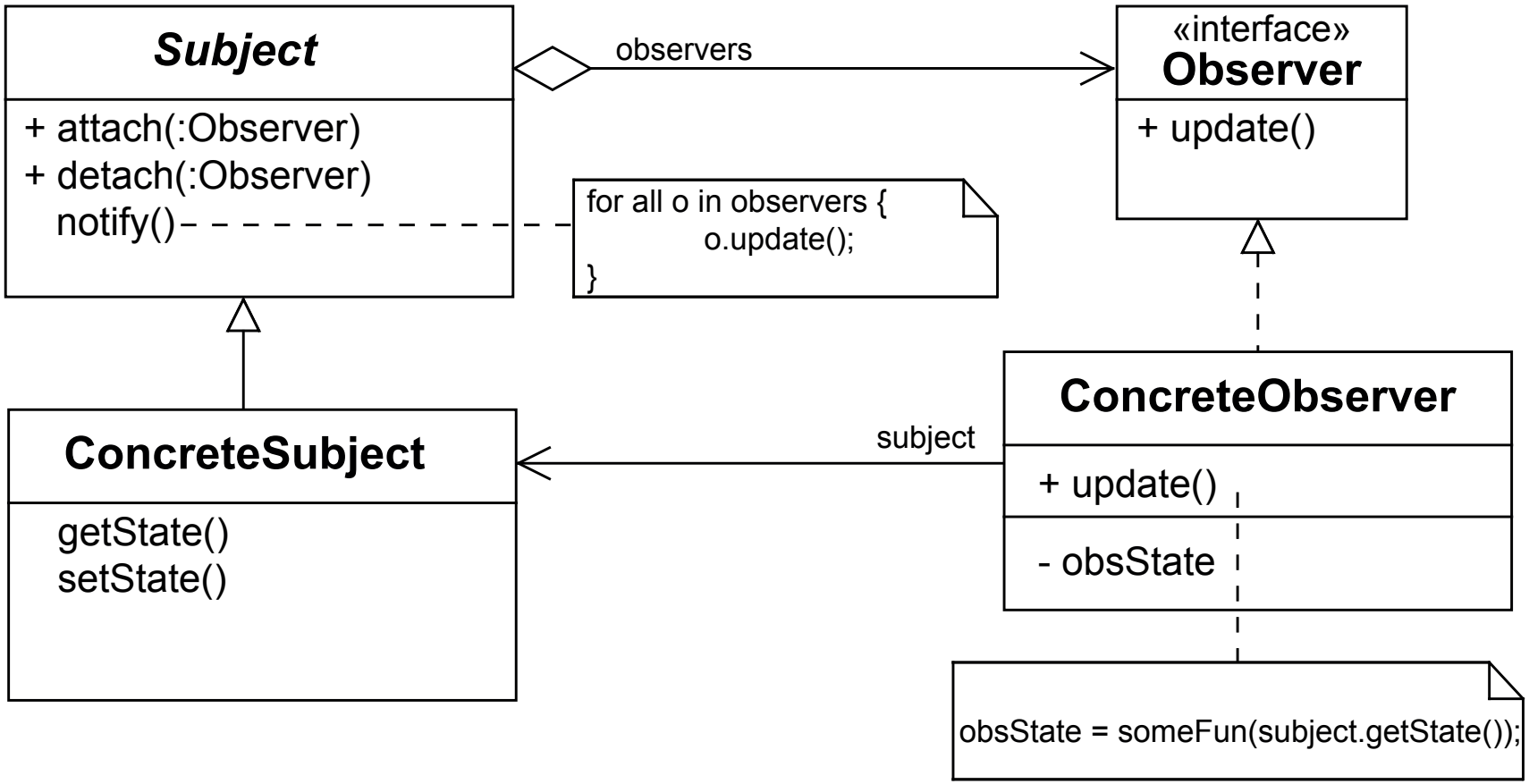
- Questo comportamento implica che gli oggetti grafici dipendono dagli oggetti che corrispondono ai dati e di conseguenza devono essere notificati quando questi ultimi cambiano il loro stato.
- Non c'è motivo di limitare il numero di oggetti grafici che dipendono dagli stessi dati.
- Il pattern `Observer` dettaglia come realizzare queste relazioni di dipendenza: i ruoli chiave sono il soggetto (*subject*) e l'osservatore (*observer*).
- Per un dato soggetto possono esistere un numero imprecisato di osservatori che dipendono da esso. Tutti gli osservatori sono notificati quando il soggetto cambia il suo stato. In risposta, ogni osservatore richiederà le informazioni necessarie per sincronizzarsi con il nuovo stato del soggetto.
- Questo tipo di interazione è anche noto come *publish-subscribe*. Il soggetto è chi *pubblica* le notifiche senza sapere chi sono gli osservatori. Un numero qualsiasi di osservatori può iscriversi per ricevere le notifiche.

Applicabilità

È opportuno impiegare il pattern `Observer` nei seguenti casi:

- Quando un'astrazione presenta due aspetti, di cui uno dipendente dall'altro. Incapsulando questi aspetti in due oggetti separati è possibile riusarli indipendentemente.
- Quando una modifica a un oggetto implica delle modifiche in altri oggetti che dipendono da questo, ma in generale non è noto il numero degli oggetti dipendenti.
- Quando un oggetto deve essere in grado di notificare altri oggetti senza conoscerne l'identità. In altre parole si vuole mantenere un alto livello di disaccoppiamento tra questi oggetti.

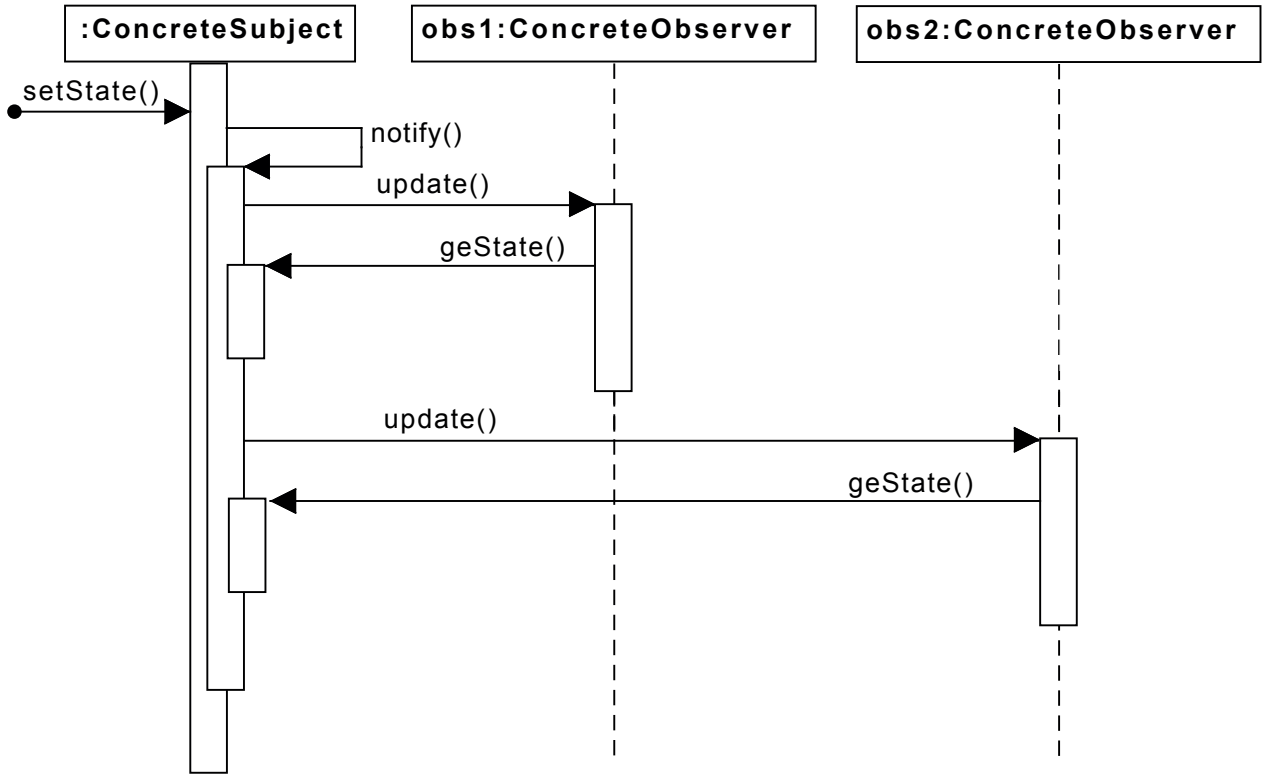
Struttura


Subject
+ attach(:Observer)
+ detach(:Observer)
notify()
observers
«interface»
Observer
+ update()
for all o in observers {
o.update();
}
ConcreteSubject
getState()
setState()
subject
ConcreteObserver
+ update()
- obsState
obsState = someFun(subject.getState());

Partecipanti

- **Subject**:
  - Conosce i propri osservatori; un numero qualunque di oggetti `Observer` può osservare un soggetto.
  - Fornisce un'interfaccia per registrare e cancellare le registrazioni degli oggetti `Observer`.
- **Observer**:
  - Fornisce un'interfaccia di notifica per gli oggetti a cui devono essere notificati i cambiamenti nel `Subject`.
- **ConcreteSubject**:
  - Contiene lo stato a cui gli oggetti `ConcreteObserver` sono interessati.
  - Inoltra una notifica ai suoi `Observer` quando il proprio stato si modifica.
- **ConcreteObserver**:
  - Memorizza un riferimento a un oggetto `ConcreteSubject` oppure ottiene dinamicamente il riferimento all'oggetto `Subject` da cui ha origine la notifica in caso in esso cui osservi più `Subject`.
  - Contiene informazioni che devono essere sincronizzate con lo(gli) stato(i) del (dei) `Subject`.
  - Implementa l'interfaccia `Observer` per ricevere le notifiche del `Subject`.

Collaborazioni

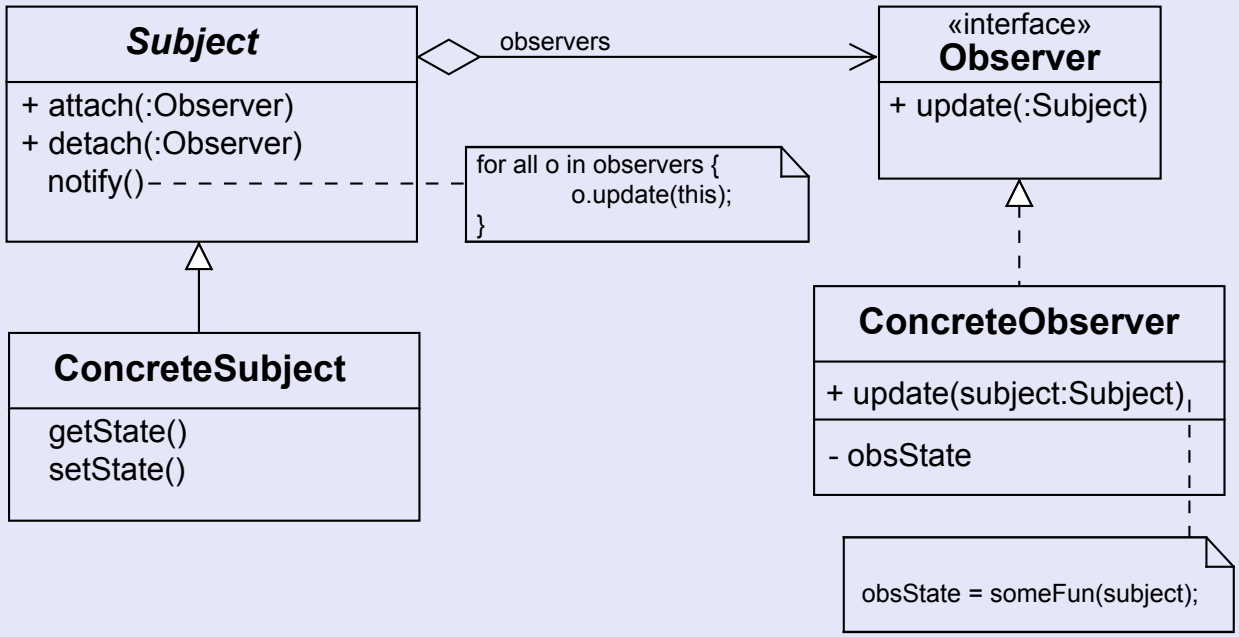


- `ConcreteSubject` notifica i propri `Observer` quando avviene un cambiamento che potrebbe rendere il loro stato non inconsistente rispetto al proprio.
- Dopo essere stato informato di un cambiamento nel `ConcreteSubject`, un osservatore concreto può chiedere al subject informazioni sul suo stato. `ConcreteObserver` usa questa informazione per riconciliare il suo stato con quello del subject.

Conseguenze

- ☺ Il pattern `Observer` consente di modificare i subject e gli observer indipendentemente uni dagli altri.
- ☺ Si possono riusare i subject senza riusare i suoi observer, e vice versa.
- ☺ E' possibile aggiungere observer senza modificare il subject o gli altri observer.
- ☺ Il subject conosce la lista degli osservatori (conformi all'intefaccia `Observer`) ma ne ignora il tipo concreto.
- ☺ Visto il basso accoppiamento tra il subject e i suoi observer, essi possono appartenere a strati diversi di un'applicazione. Tipicamente il subject appartiene a un livello più basso e notifica gli observer che appartengono a un livello più alto.
- ☺ Il pattern `Observer` supporta la comunicazione broadcast: per inviare la notifica, il subject non deve specificare i destinatari. Gli osservatori interessati sono notificati in broadcast. E' responsabilità di un observer decidere se ignorare o meno la notifica ricevuta.
- ☹ Poiché gli osbserver non si conoscono reciprocamente questo può nascondere il costo implicato da un cambiamento nel subject. Il subject non conosce il numero di cambiamenti potenzialmente causati in oggetti dipendenti dagli observer. Questo è aggravato dal fatto che nello schema di base il subject non fornisce dettagli su *cosa* è cambiato.

Observing more Subjects

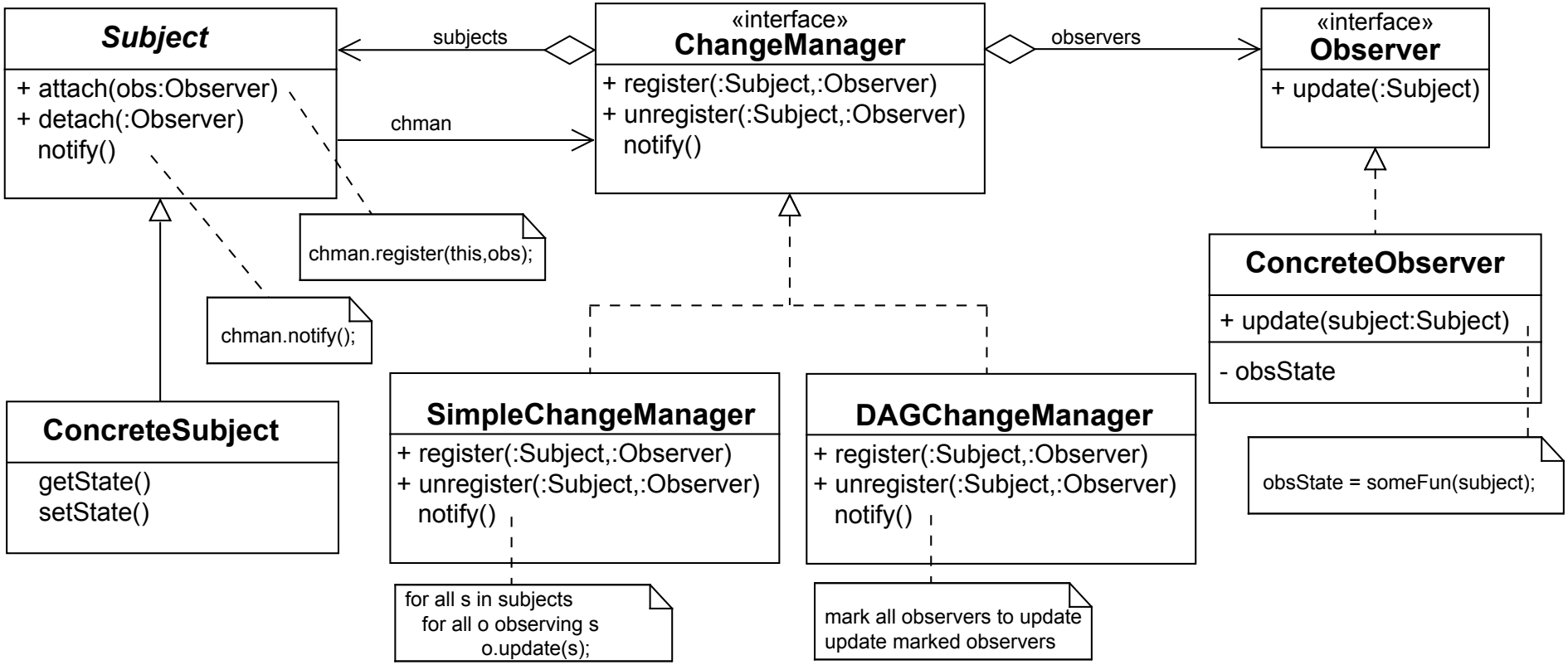


- In alcune situazioni ha senso che un observer dipenda da più di un subject.
- Per esempio un foglio di calcolo può dipendere da più sorgenti di dati.
- Per fare ciò è necessario estendere il metodo `update` in modo da informare l'observer circa l'identità del subject che ha inviato la notifica.

Push and pull

- Le implementazioni del pattern `Observer` spesso fanno in modo che il subject invii informazioni addizionali sul cambiamento.
- Il subject fornisce queste informazioni come argomento del metodo `update`. La quantità di informazione può variare molto.
- Un estremo è il **push model**: il subject invia informazioni dettagliate sul cambiamento.
- L'altro è il **pull model**: il subject invia solo la notifica e gli osservatori hanno la responsabilità di richiedere le esplicitamente i dettagli necessari.
- Il pull model enfatizza che il subject ignori i suoi observer, mentre il push model assume che i subject conoscano le necessità dei propri observer.
- Il push model può rendere gli observer meno riusabili, poiché le classi `Subject` fanno delle assunzioni sulle classi `Observer` che possono non essere sempre vere.
- Il pull model potrebbe essere inefficiente, poiché le classi `Observer` devono capire cosa è cambiato senza aiuto da parte del `Subject`.

Encapsulating complex update semantics



- Quando la relazione di dipendenza tra i subject e gli observer è molto complessa è opportuno introdurre un oggetto a cui attribuire la responsabilità di gestire tali relazioni.
- Tale oggetto è indicato con il nome di `ChangeManager`: il suo scopo è quello di minimizzare il lavoro richiesto per far in modo che gli observer riflettano le modifiche nei soggetti osservati.

ChangeManager

Il `ChangeManager` ha tre responsabilità:

1. Mappa ciascun subject su i suoi osservatori e fornisce un'interfaccia per gestire questo mapping. In questo modo si evita che i subject memorizzino gli observer e vice versa.
2. Definisce una particolare strategia di aggiornamento.
3. Aggiorna tutti gli observer che dipendono da una richiesta di un subject.

- Nel diagramma precedente sono riportate due classi che implementano l'interfaccia `ChangeManager`.
- `SimpleChangeManager` banalmente aggiorna tutti gli observer di ciascun subject.
- `DAGChangeManager` gestisce dipendenze modellate come grafi orientati aciclici (Directed-Acyclic Graphs) tra i subject e i rispettivi osservatori assicurando che ciascun osservatore venga notificato una sola volta in conseguenza di un cambiamento che coinvolge più soggetti.

Pattern Correlati

- **Mediator**: Poiché incapsula una semantica complessa di aggiornamento, il `ChangeManager` si comporta come un mediatore tra i subject e gli observer.
- **Singleton:** Il `ChangeManager` può utilizzare il pattern Singleton per renderlo unico e accessibile globalmente.