Design pattern – IV

Stefano Mizzaro

Dipartimento di matematica e informatica
Università di Udine
http://www.dimi.uniud.it/~mizzaro
mizzaro@dimi.uniud.it

PAOO, Lezione 12
30/4/2004

Riassunto

- I 7 pattern strutturali: Adapter, Façade, Composite, Decorator, Bridge, Proxy, Flyweight
- I 5 pattern creazionali: Factory Method, Abstract Factory, Singleton, Prototype, Builder

Scaletta

- Seminario:
  - Junit, di Mauro Lorenzutti
- Analisi dei pattern creazionali
- Altri pattern: i pattern comportamentali
  - I primi 4: Template Method (Metodo sagoma), Strategy (Strategia), State (Stato), Command (Comando)
  - Somiglianze, abbastanza semplici…
  - Poi vedremo gli altri

Analisi dei pattern creazionali

- Sono piuttosto simili
- Sono spesso in alternativa
- Factory Method: ereditarietà
- Abstract Factory, Builder, Prototype: composizione
- Ci sono legami:
  - Singleton usato per avere un'unica Factory
  - Factory Method usato in Abstract Factory
  - …
- Abstract Factory, Prototype, Builder portano a progetti più flessibili e complessi
- Factory Method è più semplice (spesso si parte da qui)

I pattern comportamentali

- Algoritmi
- Assegnamento/suddivisione responsabilità fra oggetti collaboranti
- Gestione comunicazioni fra oggetti collaboranti
- Flussi di controllo difficili da seguire durante l'esecuzione

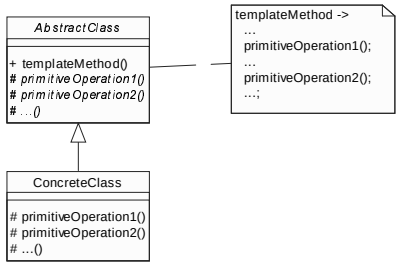
Gli 11 pattern comportamentali

13. Template Method (Metodo sagoma)
14. Strategy (Strategia)
15. State (Stato)
16. Command (Comando)
17. Observer (Osservatore)
18. Mediator (Mediatore)
19. Iterator (Iteratore)
20. Visitor (Visitatore)
21. Chain of Responsibility (Catena di responsabilità)
22. Memento (Memento, Ricordo)
23. Interpreter (Interprete)

13. Template Method (Metodo sagoma)

- Scopo
  - Definire la struttura di un algoritmo in un metodo…
  - … lasciandone alcune parti non specificate.
  - L'implementazione delle parti non specificate è in altri metodi la cui implementazione è delegata alle sottoclassi
- Le sottoclassi ridefiniscono solo alcuni passi dell'algoritmo, non la struttura generale

Diagramma Template Method



Commenti

- Utile nei Framework OO
- Anche più sottoclassi concrete
- **`primitiveOperation`**$_i$
  - Detti "metodi gancio" (hook)
  - Eventualmente **`public`**
  - Eventualmente stub (non **`abstract`** e **`{}`**)
- Evitare di imporre la definizione di tanti metodi alla/e sottoclasse/i

Template Method vs. Factory Method

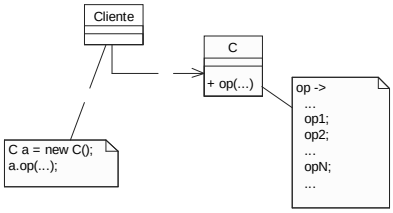
- L'idea di base è simile:
  - In una classe, metodi astratti invocati da un altro metodo e
  - specificati nelle sottoclassi
- Ma sono ≠
  - Il Template Method è il metodo che invoca i metodi astratti
  - Il Factory Method è un metodo astratto…
  - …e deve creare e restituire l'istanza

14. Strategy (Strategia)

- Scopo
  - Definire una famiglia di algoritmi…
  - …incapsularli…
  - … e renderli intercambiabili
  - Permettere agli algoritmi di variare
- Mentre Template Method usa l'ereditarietà, Strategy usa la composizione

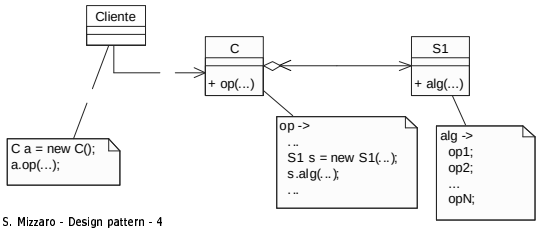
Capiamo lo Strategy (1/6)

- Algoritmo complesso cablato nel codice di un metodo **`op`** di una classe **`C`** (usata da un **`Cliente`**)



Capiamo lo Strategy (2/6)

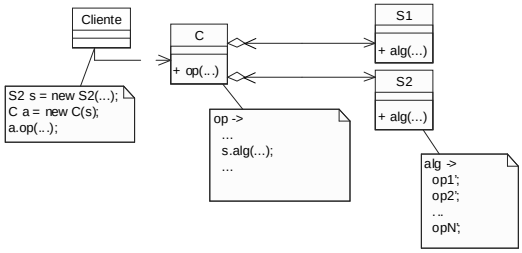
- Estraggo l'algoritmo in un metodo **`alg`**
- con interfaccia ben definita
- e metto **`alg`** in un'altra classe **`S1`**



Capiamo lo Strategy (3/6)

- Per non rimetterci:
  - Da **`alg`** potrei aver bisogno di accedere a attributi/metodi di **`C`** e a variabili locali di op
  - navigabilità da **`S1`** a **`C`**
  - Parametri
- Complicazioni? Non solo:
  - **`C`** potrebbe essere configurato da **`Cliente`** per usare **`S1`** (argomento nel costruttore)
  - Questo mi permette, un domani, di aggiungere **`S2`** con un'implementazione diversa di **`alg`** con minimo sforzo

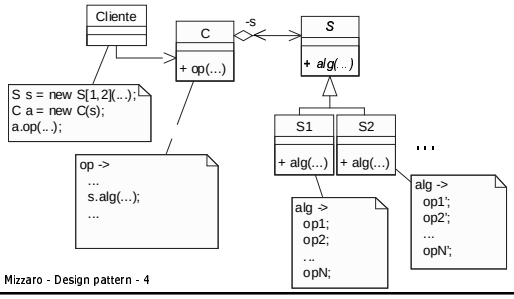
Capiamo lo Strategy (4/6)



- Ma ora mi rendo conto che **`S1`** e **`S2`** hanno parecchio in comune…

Capiamo lo Strategy (5/6)

- Aggiungo una sopraclasse astratta con **`alg`** astratto



Capiamo lo Strategy (6/6)

- Così **`C`** non conosce le sottoclassi, ma parla alla classe base
  - (ricordate che **`C`** può essere configurato dal **`Cliente`**)
- Ora (e un domani!) posso aggiungere altri **`alg`** in altre sottoclassi
- Con una minima modifica il **`Cliente`** ha un altro algoritmo

Commenti

- Il **`Cliente`** può usare algoritmi con caratteristiche diverse…
  - (occupazione memoria, velocità, comprensibilità,…)
- …in modo trasparente
- Si chiama Strategy e non Algorithm…
  - "Strategia" è un concetto più generale
- La strategia può essere scelta al run-time
  - … e non sarebbe così se ereditassi da **`C`**

Pattern correlati

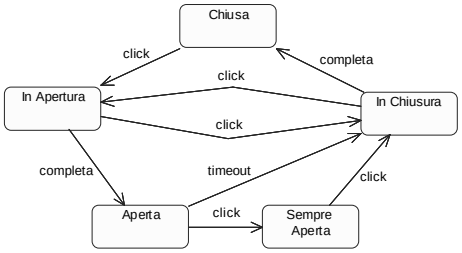
- Flyweight, Singleton
  - Gli oggetti Strategy sono spesso ottimi candidati per i pattern Flyweight e Singleton
- Factory
  - Posso usarlo per avere un cliente completamente indipendente dall'oggetto Strategy concreto
- Template Method (composizione e eredità)

15. State (Stato)

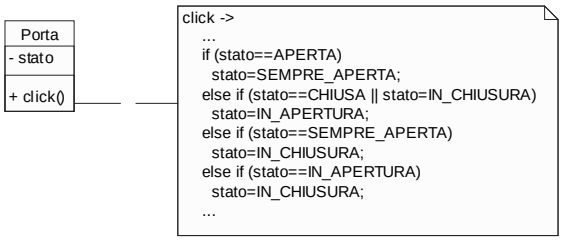
- Scopo
  - Permettere a un oggetto di cambiare il suo comportamento al variare del suo stato interno
- Come funziona
  - Si estrae la rappresentazione dello stato in classi esterne…
  - … organizzate in una gerarchia…
  - … e si sfrutta il polimorfismo per variare il comportamento

Esempio

- Porta che si apre/chiude
  - Click su pulsante (e tempi di attesa…)



Implementazione ingenua



Commenti

- Comportamento "cablato"
- Tanti if… (campanello d'allarme)
- Difficile da comprendere…
- Alternativa
  - Usiamo una gerarchia di classi per rappresentare gli stati della porta
  - Ogni (sotto)classe uno stato
  - Ogni classe ha il suo `click`

Implementazione con State

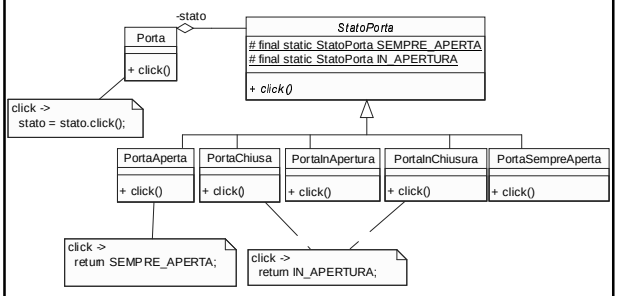
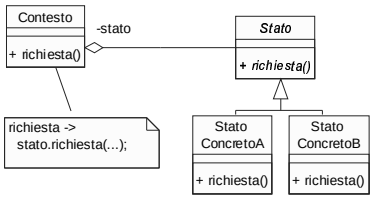


Diagramma State



Commenti

- Soluzione ingenua difficile da manutenere/estendere
- "Distribuzione" del comportamento su sottoclassi
  - Più classi
  - Applicazione meno compatta
  - Ogni sottoclasse deve "conoscere" altre sottoclassi
- Da usare quando ci sono molti stati con molte scelte condizionali

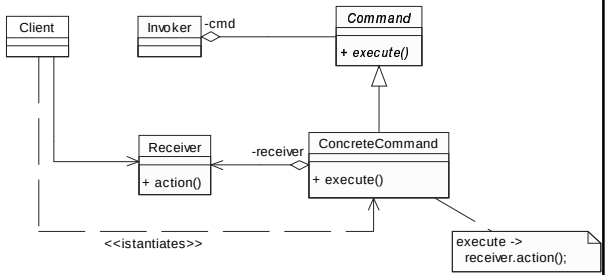
State vs. Strategy

- Similitudine con Strategy
  - Si mette qualcosa in una classe esterna…
  - …o meglio in una gerarchia esterna, e si usa il polimorfismo
  - Gli stati possono essere Flyweight o Singleton
- Differenze
  - Nel mondo reale, strategia e stato sono cose ≠
  - In Strategy c'era il cliente
- Mah…

16. Command (Comando)

- Scopo
  - Incapsula una richiesta in un oggetto…
  - … consentendo di parametrizzare i client con richieste diverse, di mantenere uno storico delle richieste, di gestire l'undo
- Se una richiesta, un comando, è un oggetto,
  - "vive di vita propria": può essere memorizzato, passato come argomento, ecc. ecc.

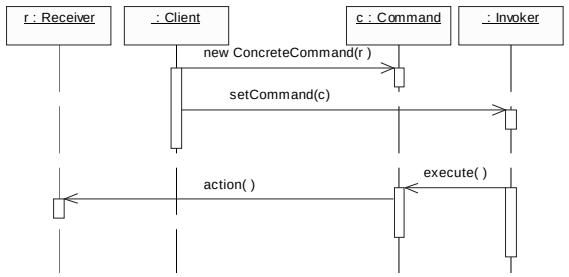
Diagramma Command



Funzionamento

- Un'istanza di **`Client`** crea un'istanza di **`ConcreteCommand`** e gli passa il **`Receiver`**
  - Un'istanza di **`Command`** è un comando: a qs. punto il comando è un oggetto…
- Poi, un'istanza di **`Invoker`** (che memorizza un riferimento al comando) ne invoca polimorficamente **`execute()`**…
- … e l'istanza di **`ConcreteCommand`** invoca action di **`Receiver`**

Diagramma di sequenza



Commenti

- **`Command`** disaccoppia **`Invoker`** (che invoca la richiesta) e **`Receiver`** (che esegue)
- I comandi sono oggetti ⇒
  - È possibile comporre più comandi in un comando composto (Composite)
  - È facile aggiungere nuovi comandi
  - È possibile memorizzare uno storico dei comandi per l'undo

Riassunto

- Analisi dei pattern creazionali
- Altri pattern: i pattern comportamentali
  - I primi 4: Template Method (Metodo sagoma), Strategy (Strategia), State (Stato), Command (Comando)
  - Somiglianze, abbastanza semplici…
  - Poi vedremo gli altri