12 Πρότυπα σχεδίασης συμπεριφοράς

Τεχνολογία Λογισμικού

Σχολή Ηλεκτρολόγων Μηχανικών & Μηχανικών Υπολογιστών
Εθνικό Μετσόβιο Πολυτεχνείο

Χειμερινό εξάμηνο 2017-18

Δρ. Κώστας Σαΐδης (saiko@di.uoa.gr)

Πρότυπα σχεδίασης συμπεριφοράς

Πρότυπα σχετικά με τη συμπεριφορά / επικοινωνία των αντικειμένων

Περιεχόμενα

Επίσης

8. Iterator / Iterable
9. Observer / Observable
10. Callback / Future / Promise

Visitor

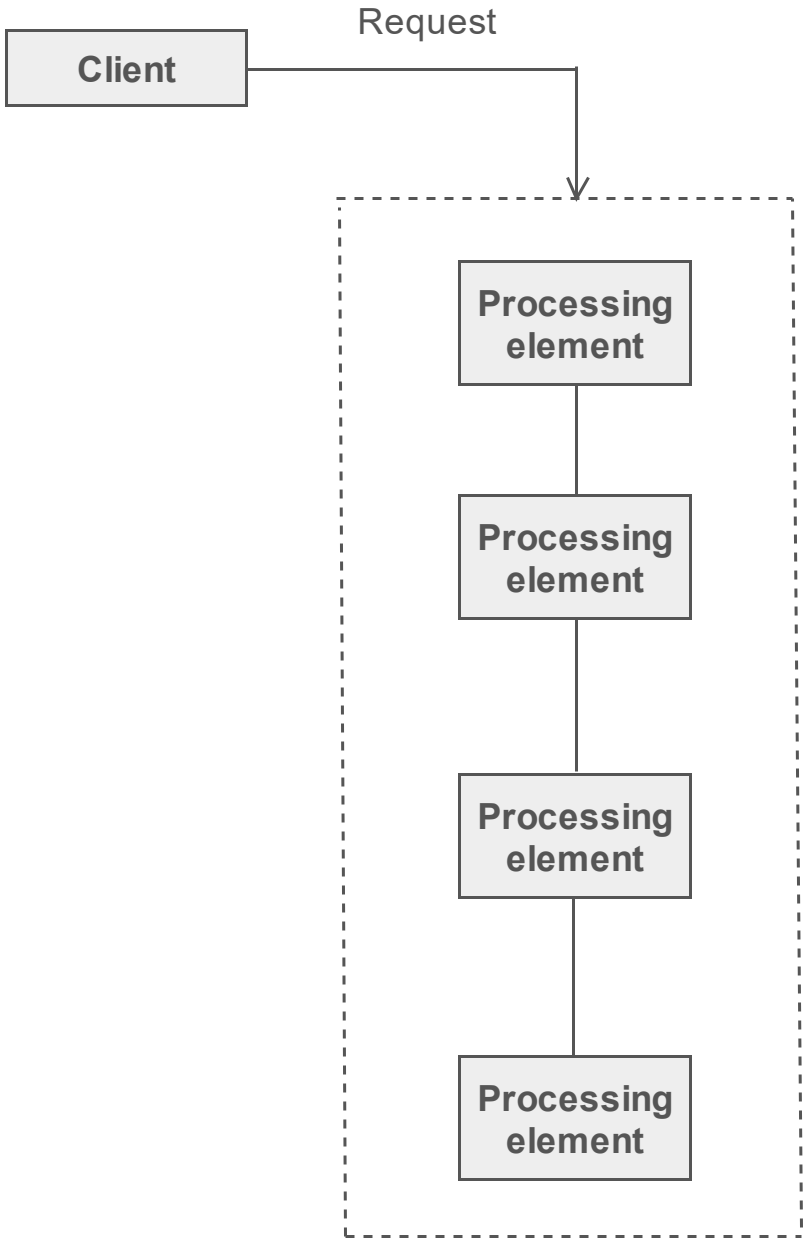
Σκοπός: Να διαχωρίσει τον αλγόριθμο από τη δομή δεδομένων.

Η "λογική" της επίσκεψης στα στοιχεία της δομής (traversal) διαχωρίζεται από τη δομή αυτή καθ' αυτή.

Το έχουμε δει ξανά στην εισαγωγή για τα πρότυπα σχεδίασης

Chain of responsibility

Σκοπός: Να περάσει ένα αίτημα μέσω μιας αλυσίδας αντικειμένων


Request
Client
Processing element
Processing element
Processing element
Processing element

Παράδειγμα

Το παράδειγμα του Composite προτύπου από το προηγούμενο μάθημα χρησιμοποιεί το Chain of responsibility

Πραγματικό παράδειγμα

Servlet filters

Command

Σκοπός: Να "ενθυλακώσει" ένα αίτημα ως αντικείμενο, διαχωρίζοντας τον αποστολέα από τον παραλήπτη

Παράδειγμα

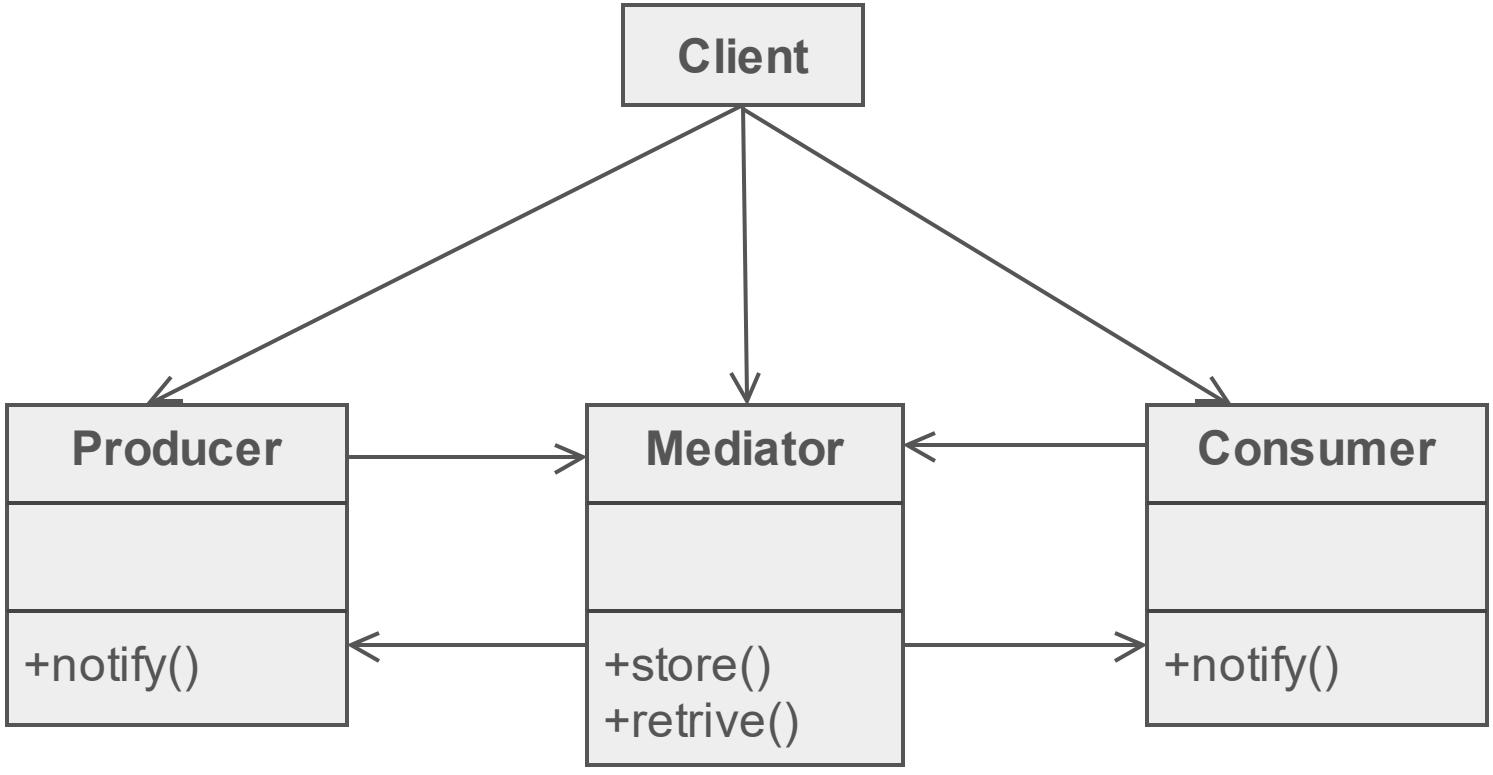
java.lang.Runnable

java.util.concurrent.Callable

Mediator

Σκοπός: Να δημιουργήσουμε έναν "ενδιάμεσο" που "γνωρίζει" πώς αλληλεπιδρούν διάφορα αντικείμενα

Χρήσιμο για σχέσεις N-N


Client
Producer
+notify()
Mediator
+store()
+retrive()
Consumer
+notify()

Memento

Σκοπός: Να "εξωτερικεύσουμε" με ασφαλή τρόπο την εσωτερική κατάσταση ενός αντικειμένου ώστε να μπορούμε να την επαναφέρουμε αργότερα.

Για την υλοποίηση λειτουργιών undo ή rollback

Παράδειγμα

```
class Memento {
  private final String state;

  public Memento(String state) {
      this.state = state;
  }

  public String getState() {
      return state;
  }
}
```

```
class Originator {
  private String state;
  ...//more fields that "depend" on the state

  private void setState(String state) {
    this.state = state;
    //change fields depending on the state
  }

  public Memento save() {
    return new Memento(state);
  }

  public void restore(Memento m) {
    setState(m.getState());
  }
}
```

Strategy

Σκοπός: Να αναπαραστήσουμε μια οικογένεια αλγορίθμων με ομοιόμορφο τρόπο ώστε να τους εναλλάσσουμε

Παράδειγμα

```
class Client {
  String id
  //...
  BillingStrategy billingPlan
}

interface BillingStrategy {
  //Fundamental billing functions
}

class EnterprisePlan implements BillingStrategy {
  //Billing details of the enterprise plan
}

class SimplePlan implements BillingStrategy {
  //Billing details of the simple plan
}
```

Null object

Σκοπός: Να υλοποιήσουμε με ομοιόμορφο τρόπο το NO-OP (do nothing)

Αποφεύγουμε τους ελέγχους για null διατηρώντας τη λογική του αλγορίθμου απλή και καθαρή

Παράδειγμα

Ένα ενδεχόμενο FreePlan στην περίπτωση του BillingStrategy παραδείγματος μπορεί να υλοποιηθεί ως Null object (μια υλοποίηση του BillingStrategy με μεθόδους που δεν κάνουν τίποτα)!

Iterator / Iterable

Σκοπός: Να μοντελοποιήσουμε την επανάληψη ή τη δυνατότητα αυτής

Iterator

java.util.Iterator

```
interface Iterator<E> {
  boolean hasNext();
  E next();
  void remove(); //we don't care about this in the class
}
```

Χρήση

```
while(iterator.hasNext()) {
  Element e = iterator.next();
  //do something with e
}
```

Iterable

java.lang.Iterable

```
interface Iterable<E> {
  Iterator<E> iterator();
}
```

Pull paradigm

- Ο χρήστης/client του Iterable κάνει pull για την επόμενη τιμή (καλεί τις hasNext/next)

Observer / Observable

Σκοπός: Να ενημερωνόμαστε για τις αλλαγές στην κατάσταση ενός αντικειμένου

Σχέση 1-N (1 observable, πολλοί observers)

Observable (Subject)

java.util.Observable

```
class Observable {
  void addObserver(Observer o);
  void deleteObserver(Observer o);
  boolean hasChanged();
  void notifyObservers();
  void notifyObservers(Object arg);
}
```

Observer

java.util.Observer

```
interface Observer {
  void update(Observable o, Object arg);
}
```

Push paradigm

- Ο χρήστης/client του Observable ενημερώνεται για την επόμενη τιμή (το Observable κάνει push)
- Publish/Subscribe

Παράδειγμα

Το knockout.js υποστηρίζει το Observer/Observable

Callback

Σκοπός: Να εκτελέσουμε ένα κομμάτι κώδικα αφού έχει ολοκληρωθεί ένας υπολογισμός (σύγχρονα ή ασύγχρονα)

Παράδειγμα

```
interface Callback {
  void call()
}
```

Θα μπορούσε να είναι και Runnable ή Callable

```
abstract class Task {
  abstract void execute()
  void executeAnd(Callback callback) {
    execute()
    callback.call()
  }
}
```

Πραγματικό παράδειγμα

Τα Javascript callbacks (π.χ. σε μια AJAX κλήση)

Callback hell

```
getData = function(param, callback){
  $.get('http://example.com/get/'+param,
    function(responseText){
      callback(responseText);
    });
}

getData(0, function(a){
  getData(a, function(b){
    getData(b, function(c){
      getData(c, function(d){
        getData(d, function(e){
          // ...
        });
      });
    });
  });
});
```

Future / Promise

Σκοπός: Να διαχειριστούμε το αποτέλεσμα ενός υπολογισμού (a value that will eventually become available) με ομοιόμορφο τρόπο, ανεξάρτητα του αν ο υπολογισμός γίνεται σύγχρονα ή ασύγχρονα

Τρεις καταστάστεις

- Pending (not yet available)
- Fulfilled (with an optional value)
- Rejected (due to an error or a timeout, with an optional value)

Εκτέλεση

Thread pool ή Event loop

Διαφορές ανά γλώσσα ή framework

Future, Promise, Deferred, Delay

Συνήθως

Future

Μια read-only αναφορά σε μια τιμή που δεν έχει ακόμα υπολογιστεί (ο χρήστης του Future δεν έχει έλεγχο στην τιμή που θα προκύψει).

Promise (CompletableFuture)

Μια single-assignment μεταβλητή για την τιμή του Future (ο χρήστης του Promise μπορεί να θέσει άπαξ την τιμή που θα προκύψει).

Promise = a future with a public set method

Future στη Java

Package java.util.concurrent

```
interface ExecutorService {
  Future<V> submit(Callable<V> callable)
}
```

Το `ExecutorService` είναι συνήθως κάποιο ThreadPool

```
interface Future<V> {
  //Attempts to cancel execution of this computation.
  boolean cancel(boolean mayInterruptIfRunning);

  //Waits if necessary for the computation to complete,
  //and then retrieves its result.
  V get();

  //Waits if necessary for at most the given time for
  //the computation to complete, and then retrieves
  //its result, if available.
  V get(long timeout, TimeUnit unit)

  //Returns true if this computation was cancelled before it
  //completed normally.
  boolean isCancelled()

  //Returns true if this computation completed.
  boolean isDone()
}
```

Παράδειγμα

```
class Bulk {
  private ExecutorService executor = //a thread pool
  private List<Future> futures = []

  void add(final Job job) {
    Future f = executor.submit(new Callable<Void>() {
      @Override
      Void call() throws Exception {
        job.execute();
      }
    });
    futures.add(f);
  }

  void execute() {
    for(Future f: futures) {
      f.get(); //wait for all futures to complete
    }
  }
}
```

Χρήση

```
Bulk bulk = new Bulk();
bulk.add(job1);
bulk.add(job2);
bulk.add(job3);
bulk.add(job4);
bulk.add(job5);
bulk.add(job6);
bulk.execute();
```

Promise στην Javascript (ES6)

```
var handlerFunction = function(resolve, reject) {
  //resolve is the function to call in case
  //of successful completion

  //reject is the function to call in case of
  //failure
};
var promise = new Promise(handlerFunction);
promise.
       then(someFunction).   //gets the resolved value
       catch(errorFunction); //gets the rejected value
```

Με το νέο συντακτικό

```
var promise = new Promise((resolve, reject) => {
  try {
    //perform a task (usually asynchronous)
    resolve(task.result);
  }
  catch(e) {
    reject(e);
  }
});
```

Πραγματικό παράδειγμα

JQuery.ajax(...) returns a promise