Μεθοδολογία Προγραμματισμού

Μοτίβα σχεδίασης (Design Patterns)

Νικόλαος Πεταλίδης

Τμήμα Μηχανικών Πληροφορικής και Επικοινωνιών
Διεθνές Πανεπιστήμιο της Ελλάδος

Εισαγωγή
Εαρινό Εξάμηνο

Μοτίβα σχεδίασης

- Οι σχεδιαστές έχουν διαπιστώσει ότι συγκεκριμένα μοτίβα (patterns) εμφανίζονται διαρκώς κατά τη σχεδίαση προγραμμάτων
- Για αυτό έχουν αναπτύξει συγκεκριμένες λύσεις οι οποίες μπορούν να εφαρμοστούν όποτε αναγνωρίζεται ότι το πρόβλημα εντάσσεται σε ένα από τα γνωστά μοτίβα ανάπτυξης

Τι είναι ένα μοτίβο σχεδίασης

Δίνουν έμφαση στη δομή του συστήματος ανεξάρτητα από το πως μεταβάλλεται στο χρόνο

- Ένα προγραμματιστικό μοτίβο που παρουσιάζεται συχνά
- Γενικό: μπορεί να εφαρμοσθεί σε πολλές περιπτώσεις
- Περιγράφει τη μορφή του κώδικα και όχι τις λεπτομέρειες

Πόσα είναι τα μοτίβα σχεδίασης

- Υπάρχουν εκατοντάδες μοτίβα σχεδίασης αλλά αρχικά ορίσθηκαν 23 βασικά
- Κάθε μοτίβο έχει τουλάχιστον
  - Ένα όνομα
  - Ένα σκοπό για τον οποίο δημιουργήθηκε
  - Σύζευξη δεδομένων
  - Μια περιγραφή του προβλήματος το οποίο προσπαθεί να λύσει

Κατηγορίες μοτίβων σχεδίασης

Δημιουργικά Ποιος είναι ο καλύτερος τρόπος να φτιάχνεις αντικείμενα.

Συμπεριφοράς Επικοινωνία μεταξύ αντικειμένων

Δομικά Συνδυασμός ομάδων αντικειμένων

Παραδείγματα μοτίβων σχεδίασης

Δημιουργικά Abstract Factory, Factory, Prototype, Singleton

Συμπεριφοράς Iterator, Visitor

Δομικά Composite, Adaptor

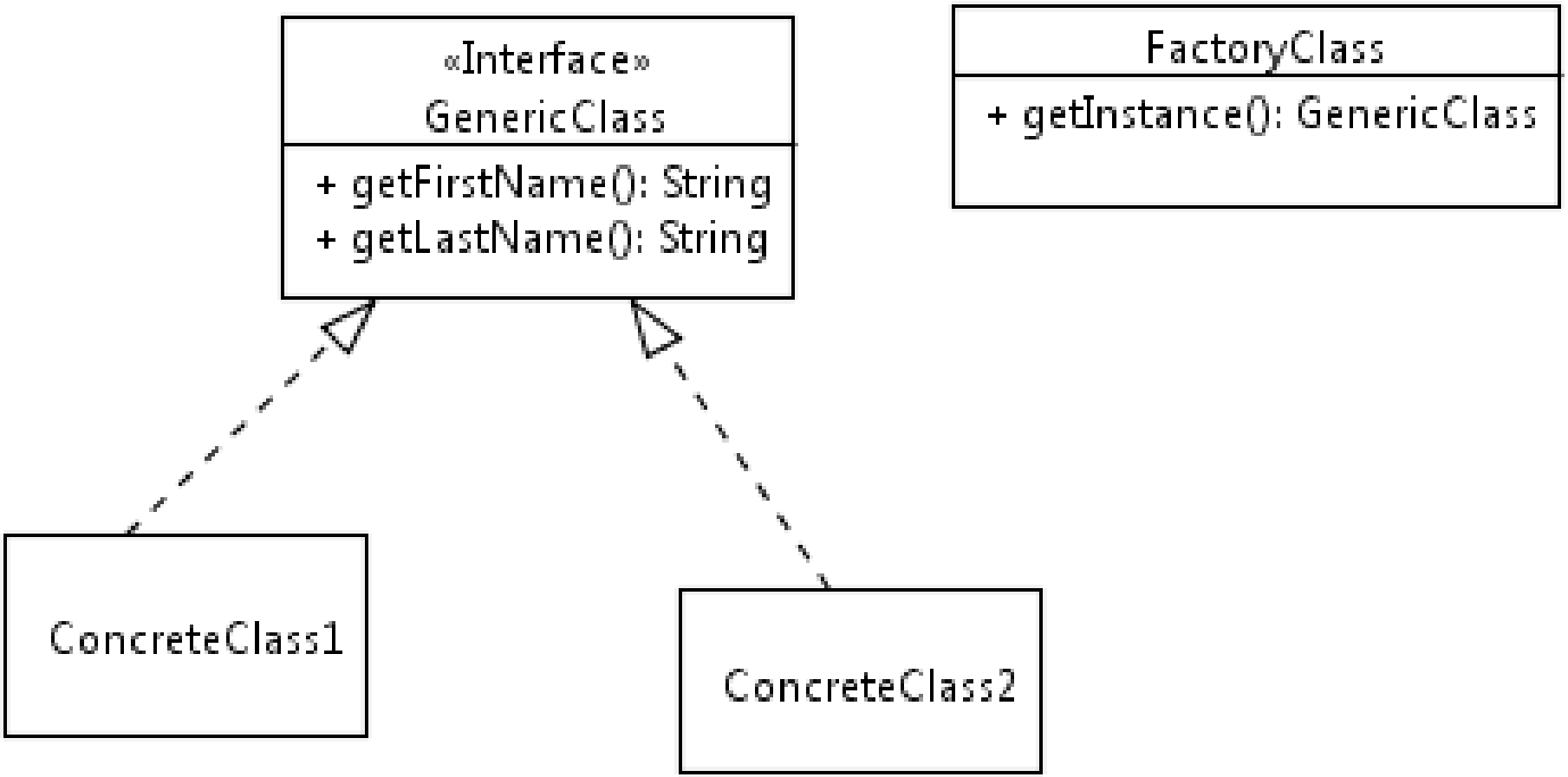
Factory Pattern

- Έχετε ένα αντικείμενο το οποίο πρέπει να επεξεργαστεί δεδομένα που μπορεί να αναπαριστώνται με διαφορετικό τρόπο.
- Για παράδειγμα σε μια φόρμα ο χρήστης μπορεί να δώσει τα στοιχεία του είτε ως *Επίθετο, Όνομα* είτε ως *Όνομα Επίθετο*
- Θέλετε πρόσβαση στο επίθετο και το όνομα μέσα από τις μεθόδους μιας κλάσης και δε θέλετε να γεμίσετε το πρόγραμμα σας με if statements

Factory Pattern

- Φτιάχνετε μια βασική κλάση Χ και δύο κλάσεις ΧΨ, ΧΖ που κληρονομούν τη Χ
- Λόγω κληρονομικότητας όμως εσείς μπορείτε να χρησιμοποιήσετε είτε τη ΧΨ είτε τη ΧΖ σα να ήταν η Χ
- Φτιάχνετε μια κλάση Factory η οποία επιστρέφει πάντα την κατάλληλη κλάση ΧΨ ή ΧΖ
- Αποτέλεσμα: Στον κώδικά σας απλά έχετε μια κλήση στη Factory και χειρίζεστε το αποτέλεσμα σα να ήταν η κλάση Χ ανεξάρτητα από τα δεδομένα που περάσατε.

Σχέδιο

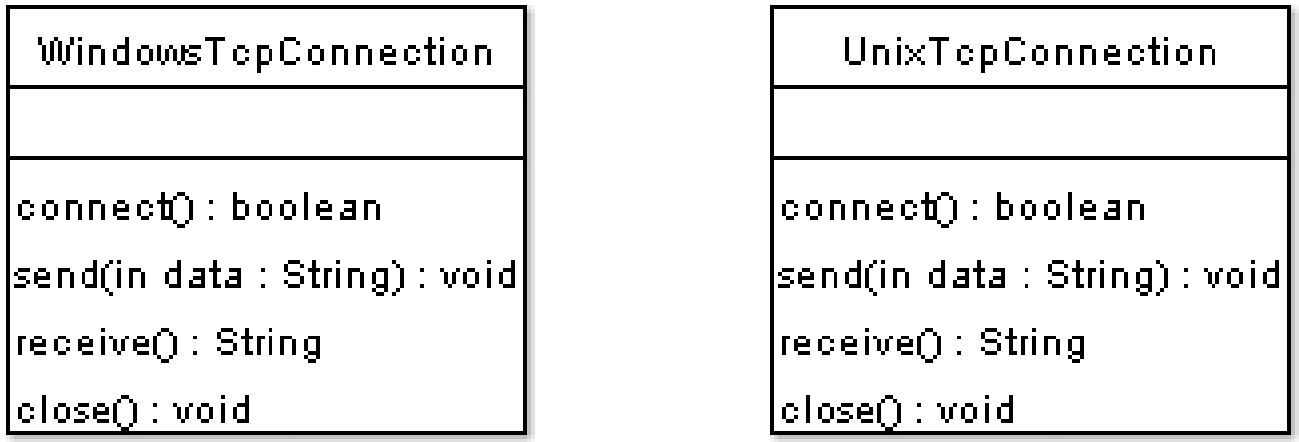

«Interface»
GenericClass
+ getFirstName(): String
+ getLastName(): String
FactoryClass
+ getInstance(): GenericClass
ConcreteClass1
ConcreteClass2

Παράδειγμα

- Γράφετε κώδικα και για Windows και για Unix.
- Πρέπει να γράψετε μια κλάση η οποία είναι υπεύθυνη για τη δημιουργία μιας σύνδεσης TCP
- Όμως η κλάση αυτή πρέπει να υλοποιεί διαφορετικά τη σύνδεση, αν πρόκειται για την υλοποίηση σε Windows και διαφορετικά αν πρόκειται για την υλοποίηση σε Unix

Χωρίς το Factory Pattern

- Ορίζετε δύο κλάσεις: `WindowsTcpConnection`, `UnixTcpConnection`



Χωρίς το Factory Pattern

- Και μέσα στον κώδικά σας, όποτε χρειάζεται να χρησιμοποιήσετε αυτήν την κλάση έχετε:

```
if (platform.equals("Unix")) {
  UnixTcpConnection connection
                  = new UnixTcpConnection();^^I
  connection.connect();
} else if (platform.equals("Windows")) {
  WindowsTcpConnection connection
                  = new WindowsTcpConnection();
  connection.connect();
}
```

Χωρίς το Factory Pattern

- Αν αργότερα αποφασίσετε να προσθέσετε και μια υλοποίηση για Mac, τότε θα προσθέσετε μια κλάση MacTcpConnection και παντού θα πρέπει να προσθέσετε τον αντίστοιχο κώδικα

Χωρίς το Factory Pattern

```
if (platform.equals("Unix")) {
    UnixTcpConnection connection
                    = new UnixTcpConnection();^^I
    connection.connect();
} else if (platform.equals("Windows")) {
    WindowsTcpConnection connection
                    = new WindowsTcpConnection();
    connection.connect();
} else if (platform.equals("Mac")) {
    MacTcpConnection connection
                    = new MacTcpConnection();
    connection.connect();
}
```

Χωρίς το Factory Pattern

- Τα πράγματα είναι ακόμα πιο δύσκολα αν αποφασίσετε να κρατάτε και μία λίστα με όλες τις ανοιχτές συνδέσεις σας.
- Σε αυτήν την περίπτωση θα πρέπει να γράψετε τον ακόλουθο κώδικα

Χωρίς το Factory Pattern

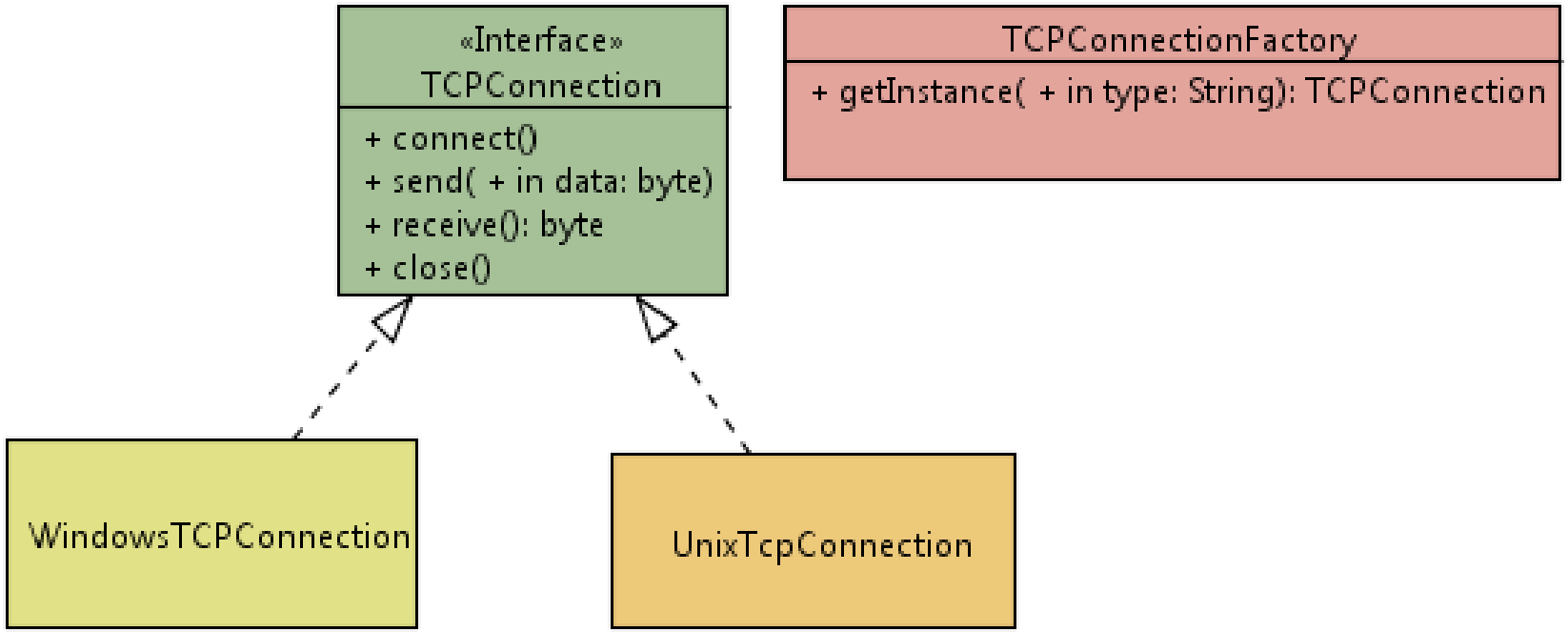
```
List<UnixTcpConnection> unixTcpConnections
  = new ArrayList<>();
List<WindowsTcpConnection> windowsTcpConnections
  = new ArrayList<>();
if (platform.equals("Unix")) {
  UnixTcpConnection connection
                = new UnixTcpConnection();^^I
  unixTcpConnections.add(connection);
  connection.connect();
} else if (platform.equals("Windows")) {
  WindowsTcpConnection connection
                = new WindowsTcpConnection();
  windowsTcpConnections.add(connection);
  connection.connect();
}
```

Προβλήματα

- Το κύριο πρόβλημα είναι ότι πρέπει να γράφετε τον κώδικά σας 2 φορές, μία για κάθε υλοποίηση.
- Αυτό τον κάνει πολύπλοκο, δυσανάγνωστο ενώ είναι εύκολο να γίνουν και λάθη

Με το Factory Pattern

- Ορίζεται κλάσεις όπως στο ακόλουθο διάγραμμα:



Με το FactoryPattern

- Η TcpConnection είναι *interface*, δεν περιέχει υλοποίηση δηλαδή των μεθόδων `connect()`, `send()`, `receive()`, `close()`.
- Ορίζει ουσιαστικά τη διεπαφή επικοινωνίας με τις κλάσεις
  - `WindowsTcpConnection`
  - `UnixTcpConnection`

Με το Factory Pattern

- Η κλάση TcpConnectionFactory περιέχει μια μέθοδο `getInstance()` η οποία μπορεί να οριστεί όπως:

```
class TcpConnectionFactory {
 public static TcpConnection getInstance(String type)
 TcpConnection connection;
 if (type.equals("Unix")) {
    connection = new WindowsTcpConnection();
  } else {
    connection = new UnixTcpConnection();
  }
  return connection;^^|^^|
}
```

Με το FactoryPattern

- Και πώς χρησιμοποιείται; Απλά:

```
TcpConnection connection =
    TcpConnectionFactory.getInstance("Windows");
connection.connect()
```

Με το Factory Pattern

- Αν θέλετε να κρατάτε και μια λίστα με όλες τις συνδέσεις σας;

```
List<TcpConnection> connections =
        new ArrayList<>();
TcpConnection connection = TcpConnectionFactory
    .getInstance("Windows");
connections.add(connection);
connection.connect();
```

Βελτιώσεις στη σχεδίαση του Factory Pattern

- Η προηγούμενη σχεδίαση απαιτούσε αλλαγή στο Factory όποτε θέλαμε να προσθέσουμε νέες κλάσεις
- Μπορούμε να σχεδιάσουμε το Factory χρησιμοποιώντας ένα χαρακτηριστικό της Java που ονομάζεται *reflection*
- Η ιδέα είναι να μπορούμε να επιτρέπουμε μια κλάση να εγγραφεί ως κομμάτι του Factory

```
class SomeFactory
{
private HashMap<String,Class> aClassMap =
                                        new HashMap<>();
public void registerClass(String name, Class class)
{
  aClassMap.put(productID, productClass);
}

public SomeClass createClass(String productID)
{
  Class aClass = aClassMap.get(productID);
  return (SomeClass)aClass.newInstance();
}
}
```

Singleton (Anti)Pattern

- Πολλές φορές θέλουμε μόνο ένα αντικείμενο από μια συγκεκριμένη τάξη σε ένα πρόγραμμα
- Παράδειγμα
  - Μόνο ένα αντικείμενο της τάξης Connection
  - Μόνο ένα αντικείμενο της τάξης Logger

Singleton vs Global variables

- Σε πολλές περιπτώσεις το Singleton χρησιμοποιείται (κακώς) σε περιπτώσεις όπου χρειάζεστε μια global variable.
- Το Singleton μπορεί να καταλήξει ως εύκολη λύση σε μια κακή σχεδίαση

Παράδειγμα υλοποίησης Singleton

```
class MySingletonClass {
 public static MySingletonClass getInstance() {
   static MySingletonClass instance
    = new MySingletonClass;
      return instance;
  }
  /** There can be only one. */
  private MySingletonClass() {       }
}
```

Singleton σε UML

Παράδειγμα

- Έστω ότι γράφετε ένα πρόγραμμα που πρέπει να έχει πρόσβαση στο πληκτρολόγιο.
- Για το σκοπό αυτό ορίζετε μια κλάση Keyboard την οποία και χρησιμοποιείται όταν θέλετε να διαβάσετε από εκεί.

Χωρίς το Singleton pattern

```
class Keyboard {
  public:
    Keyboard() {
      //initialisation to connect to hardware
    }
    char getKeystroke() {
     //code to return a single keystroke
   }
};
```

Χωρίς το Singleton pattern

- Προφανώς πρέπει να υπάρχει μόνο μία κλάση Keyboard (υποθέτουμε ότι υπάρχει ένα πληκτρολόγιο)
- Πρέπει να εμποδίσουμε τη δημιουργία περισσοτέρων από μίας κλάσεων (σκεφτείτε τι θα γίνει αν κάνουμε αρχικοποίηση το hardware ενώ κάποιος άλλος περιμένει να διαβάσει ένα keystroke)

Χωρίς το Singleton pattern

- Η μόνη λύση που έχουμε αν δε χρησιμοποιήσουμε τον singleton pattern είναι μια γενική μεταβλητή

```
Keyboard keyboard;
```

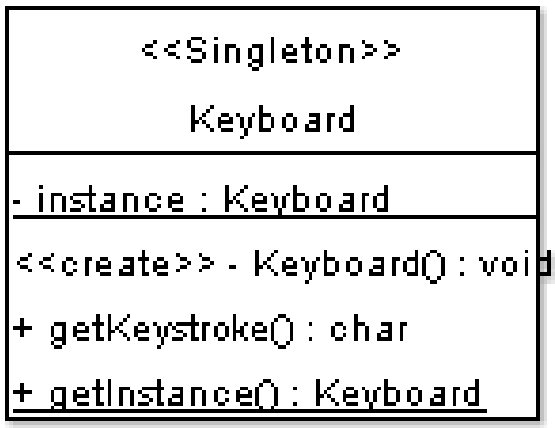
- και η ελπίδα ότι ο προγραμματιστής θα κάνει τους κατάλληλους ελέγχους όποτε χρησιμοποιεί την keyboard

```
if (keyboard==null) {
        keyboard = new Keyboard();
}

keyboard.getKeystroke();
```

Με το singleton pattern

- Καταρχήν ορίζουμε την κλάση μας διαφορετικά:



Με το Singleton pattern

```
class Keyboard {
public:
   public char getKeystroke() {
     //returns keystroke
   }
  public static Keyboard getInstance() {
    if (instance == null) {
        instance = new Keyboard();
    }
    return instance;
  }
  private Keyboard() {
^^|^^| //hardware initialisation
^^| }
  private static Keyboard instance = null;
};
```

Με το Singleton pattern

- Και η χρήση πλέον του Keyboard είναι απλή:

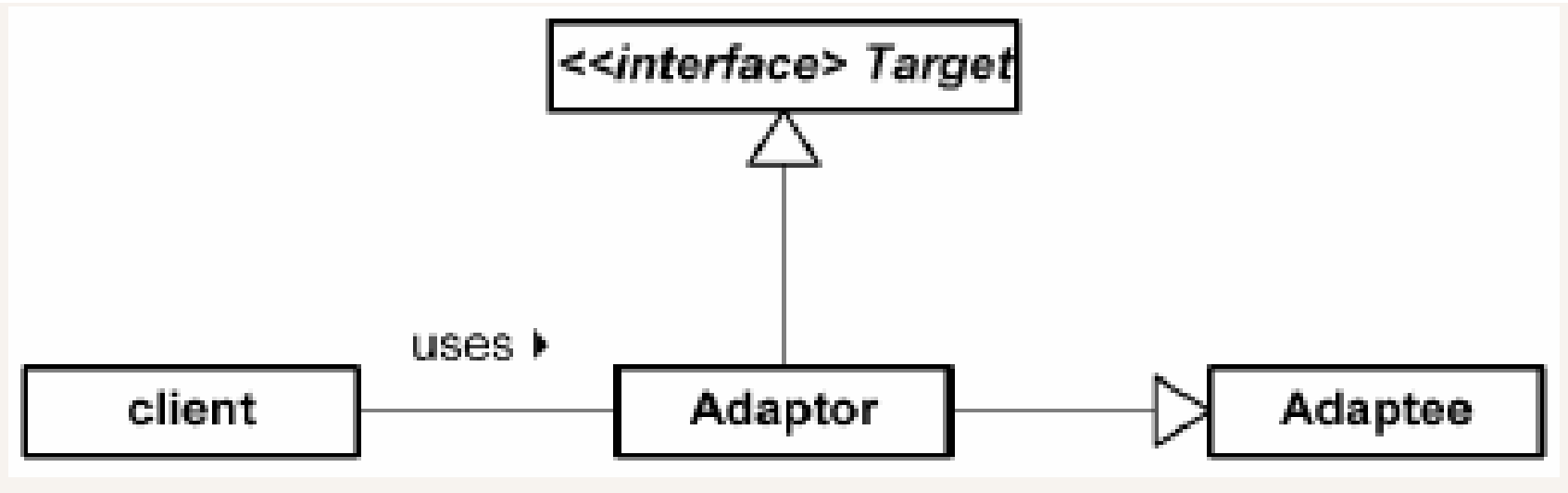
```
keyboard = Keyboard.getInstance();
```

Η υλοποίηση του Singleton στη Java

- Τα τελευταία χρόνια οι περισσότερες υλοποιήσεις του Singleton είναι με `enum`.
- Δείξτε μια υλοποίηση του Singleton με `enum`

Adapter (Wrapper) Pattern

- Ο Adapter Pattern χρησιμοποιείται για να αλλάξει τη διεπαφή μιας τάξης προκειμένου να φαίνεται συμβατή με κάποια άλλη



Παράδειγμα

- Το παράδειγμα της ουράς και της προσομοίωσης της με λίστα είναι το πιο διαδεδομένο.
- Στο παράδειγμα αυτό έχουμε υλοποίηση τη λίστα σε μια κλάση LinkedList αλλά θα θέλαμε και μια υλοποίηση ουράς
- Η λύση είναι να μη γράψουμε μια καινούργια κλάση αλλά
- Ορίζουμε ότι η CAQueue κληρονομεί από τη LinkedList και υλοποιεί τη διεπαφή μιας ουράς

Ο ορισμός της CAQueue

```
class CAQueue<T> extends LinkedList<T>
    implements Queue {
 public void enqueue(T something) {
  super.add(something);
 }
 public T dequeue() {
  return super.remove(size()-1);
}
 public boolean isEmpty() {
    return super.isEmpty();
};
```

Η υλοποίηση της enqueue/dequeue

- Η προηγούμενη υλοποίηση που δείξαμε ήταν με κληρονομικότητα
- Δείξτε μια υλοποίηση του Adapter με περιεκτικότητα