ΑΝΤΙΚΕΙΜΕΝΟΣΤΡΕΦΗΣ ΣΧΕΔΙΑΣΗ & SOLID PRINCIPLES

1

ΓΙΑ ΤΙ ΘΑ ΣΥΖΗΤΗΣΟΥΜΕ;

Εισαγωγή σε βασικές έννοιες

ΑΝΤΙΚΕΙΜΕΝΟΣΤΡΕΦΗΣ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ

- Χωρίζουμε ένα πρόβλημα σε οντότητες
  - Κάθε οντότητα είναι μία κλάση
  - Γνωρίσματα
  - Μέθοδοι

Η λογική διαφέρει από την “παραδοσιακή” προσέγγιση του διαδικαστικού προγραμματισμού

DESIGN PATTERNS

- Μοτίβα γραφής κώδικα
  - Singleton
  - Factory

Γιατί είναι χρήσιμα; Πόσο εύκολα είναι στην υλοποίηση;

The computing scientist's main challenge is not to get confused by the complexities of his own making.

E. W. Dijkstra, 1988

SOLID PRINCIPLES

- Αρχές γραφής κώδικα
  - Κατανόηση
  - Συντήρηση
  - Επεκτασιμότητα

Απαραίτητη προϋπόθεση για εργασία σε επίπεδο production (δείτε LinkedIn)

2

ΑΝΤΙΚΕΙΜΕΝΟΣΤΡΕΦΗΣ ΣΧΕΔΙΑΣΗ

Λογική, εισαγωγή σε design patterns

PROCEDURAL VS OBJECT ORIENTED

Procedural

```
typedef struct list* {
    int size;
    node head;
    node next;
} list;

void sort(list l);

int main() {
    list l;
    …
    sort(l);
}
```

Object oriented

```
public class List {
    private int size;
    private Node head;
    private Node next;

    public void sort() {
        ...
    }
}

public static void main
(String[] args) {
    List l = new List();
    l.sort();
}
```

Design patterns

Δεν τα συμπάθησε ποτέ κανείς!

SINGLETON

- Επιτρέπεται μόνο ένα στιγμιότυπο ενός αντικειμένου
  - Σύνδεση σε βάση δεδομένων
  - Εγγραφή σε αρχείο
  - Διαχείριση “κλειδωμένων” πόρων

- Στατική δομή, αρχικοποιείται μία φορά στην αρχή της εκτέλεσης

SINGLETON

```
public class DBConnection {
    private static final DBConnection INSTANCE
        = new DBConnection();

    private DBConnection() {
        // Open connection to database (e.g JDBC)
        ...
    }

    public static DBConnection getInstance() {
        return INSTANCE;
    }
}
```

FACTORY

- Θέλουμε να φτιάξουμε αντικείμενα αλλά δεν ξέρουμε ακριβώς τι τύπου
  - Cross-platform χαρακτηριστικά
  - Τυχαία δεδομένα

Υλοποιείται με Singleton!

FACTORY

```
public class FlappyBird {
    public static void main(String[] args) {
        Game g = Game.getInstance();
        while(!g.collision()) {
            g.generateObstacles();
        }
    }
}
```

FACTORY

```
public class Game {
    … // Singleton code
    public void generateObstacles() {
        …
    }
}
```

3

SOLID PRINCIPLES

Χρήσιμο εργαλείο για ποιοτικό κώδικα

SOLID PRINCIPLES

- Ας πούμε τα “αυτονόητα”
  - Single responsibility
  - Open-closed
  - Liskov susbitution
  - Interface segregation
  - Dependency inversion

Single responsibility principle

Κάθε κλάση πρέπει να κάνει ακριβώς μία δουλειά.

SINGLE RESPONSIBILITY PRINCIPLE

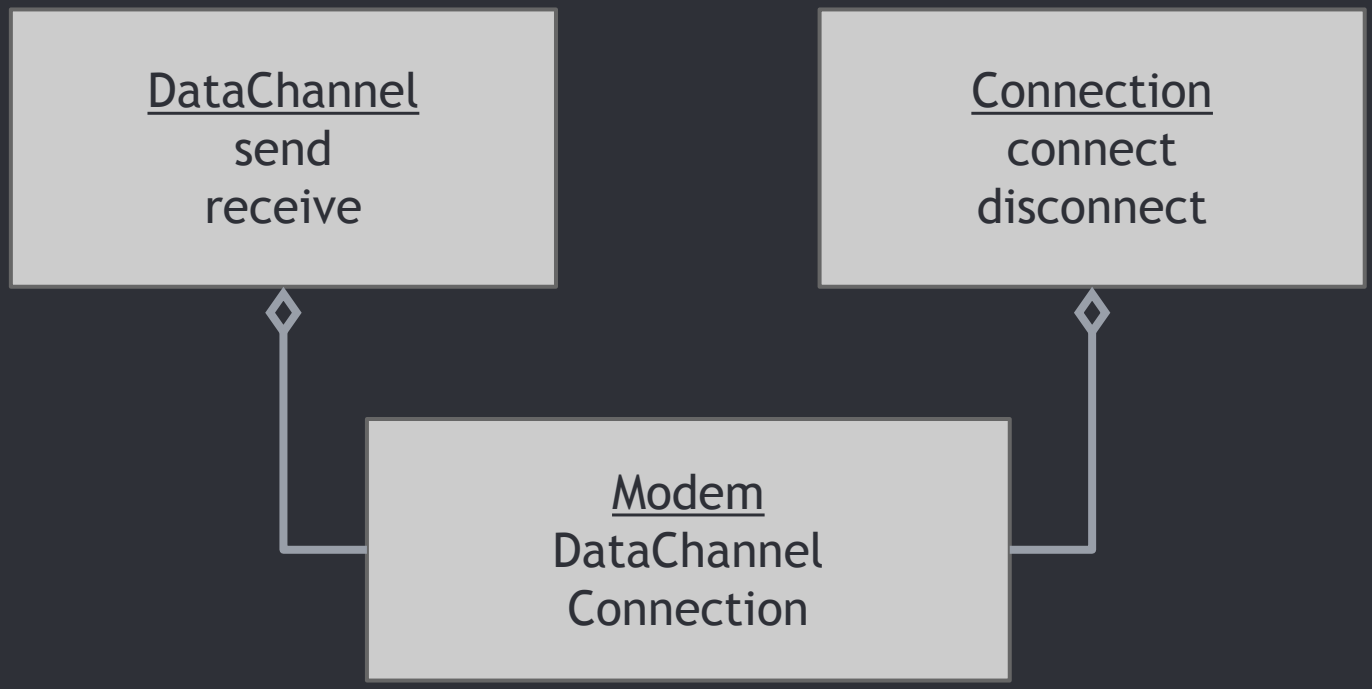
```
public interface Modem {
    public void connect(String host);
    public void disconnect();
    public void send(Packet data);
    public Packet receive();
}
```

Που είναι το λάθος; Γιατί;

SINGLE RESPONSIBILITY PRINCIPLE

```
public interface Modem {
    public void connect(String host);
    public void disconnect();
    public void send(Packet data);
    public Packet receive();
}
```

SINGLE RESPONSIBILITY PRINCIPLE



Open-closed principle

Κάθε κλάση πρέπει να είναι κλειστή σε αλλαγές, ανοιχτή σε επέκταση.

OPEN-CLOSED PRINCIPLE

```
public interface Shape {
    public void drawCircle();
    public void drawSquare();
    public void drawTriangle();
}
```

Που είναι το λάθος; Γιατί;

OPEN-CLOSED PRINCIPLE

```
public interface Shape {
    public void draw();
}

public class Circle implements Shape { … }
public class Square implements Shape { … }
public class Triangle implements Shape { … }
```

Liskov susbitution principle

Κάθε αντικείμενο στο πρόγραμμα πρέπει να μπορεί να αντικατασταθεί από ένα αντικείμενο υποκλάσης του, χωρίς να παρουσιάζει το πρόγραμμα σφάλμα εκτέλεσης.

LISKOV SUSBITUTION PRINCIPLE

```
public interface Shape {
    public void setWidth(int width);
    public void setHeight(int height);
}
```

LISKOV SUSBITUTION PRINCIPLE

```
public class Rectangle implements Shape {
    protected int width;
    protected int height;

    @Override
    public void setWidth(int width) {
        this.width = width;
    }

    @Override
    public void setHeight(int height) {
        this.height = height;
    }
}
```

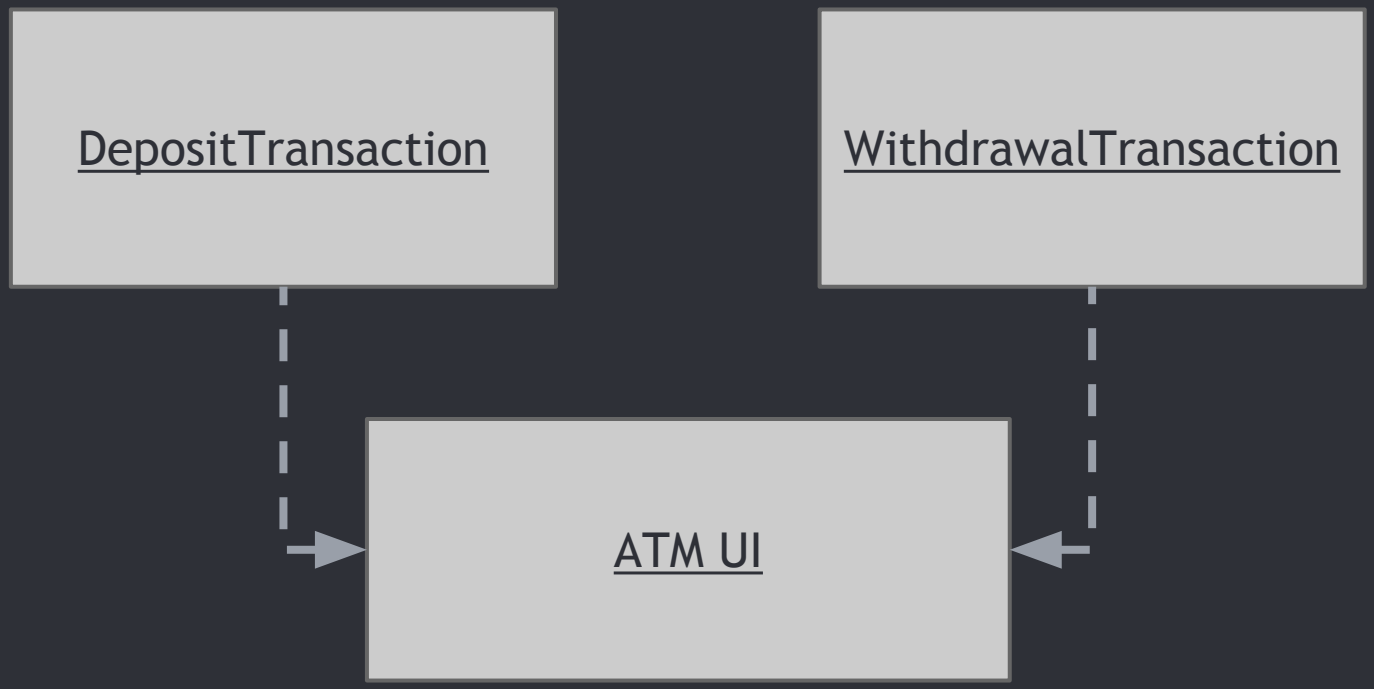
LISKOV SUSBITUTION PRINCIPLE

```
public class Square extends Rectangle {

    @Override
    public void setWidth(int width) {
        this.width = width;
        this.height = width;
    }

    @Override
    public void setHeight(int height) {
        this.height = height;
        this.width = height;
    }
}
```

Interface segregation principle

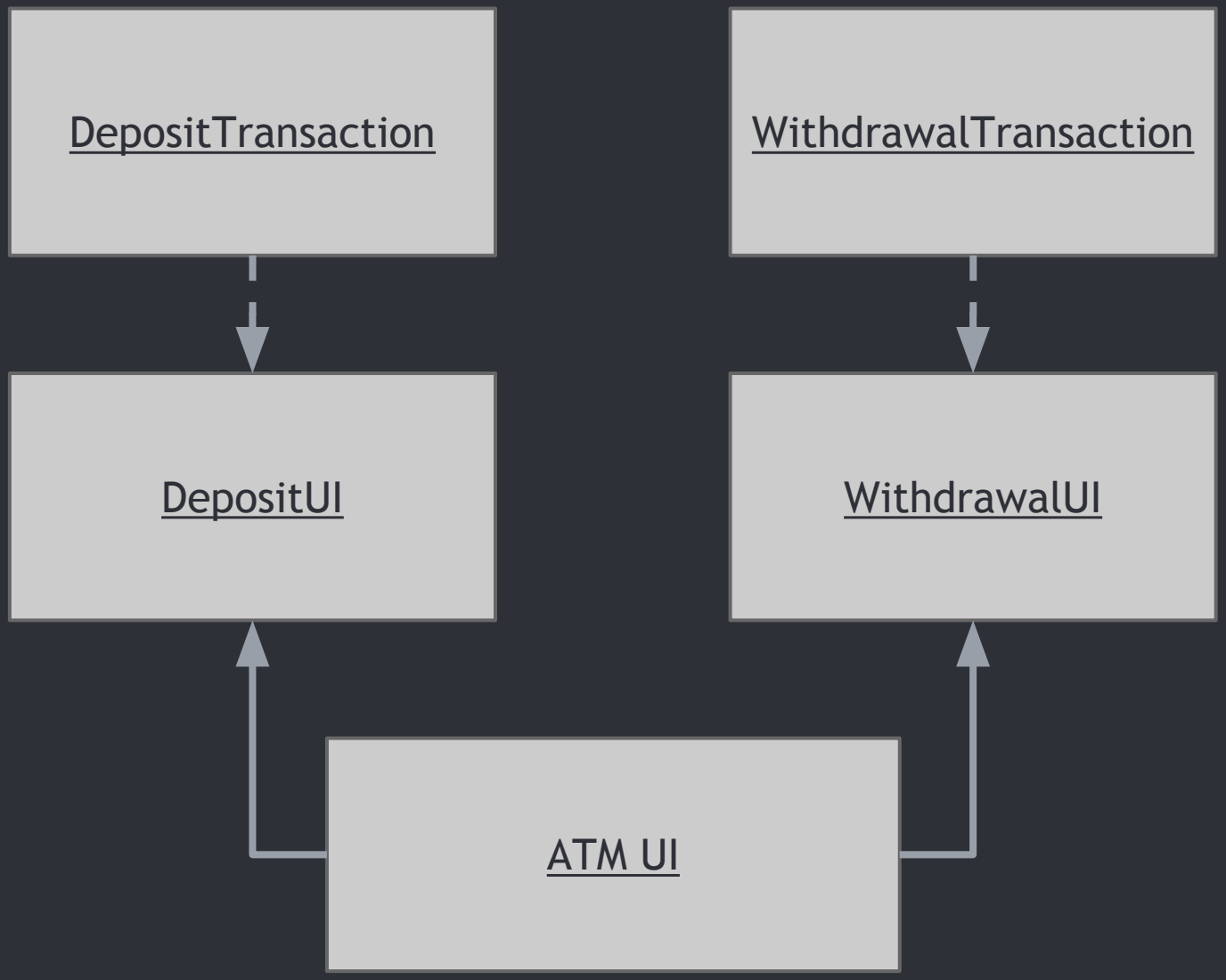
Πολλά συγκεκριμένα interfaces είναι καλύτερα από ένα γενικευμένο.

INTERFACE SEGREGATION PRINCIPLE



Τι μπορεί να γίνει καλύτερο;

INTERFACE SEGREGATION PRINCIPLE



Dependency inversion principle

Οι κλάσεις υψηλού επιπέδου δεν πρέπει να εξαρτώνται από κλάσεις χαμηλού επιπέδου. Και τα δύο πρέπει να εξαρτώνται από abstractions.

DEPENDENCY INVERSION PRINCIPLE

```
public class Lamp {
    private byte light;

    public void turnOn() {
        light = 1;
    }

    public void turnOff() {
        light = 0;
    }
}
```

DEPENDENCY INVERSION PRINCIPLE

```
public class Button {
    private Lamp lamp = new Lamp();

    public void detect() {
        boolean pressed = getPhysicalState();
        if(pressed) {
            lamp.turnOn();
        }
        else {
            lamp.turnOff();
        }
    }
}
```

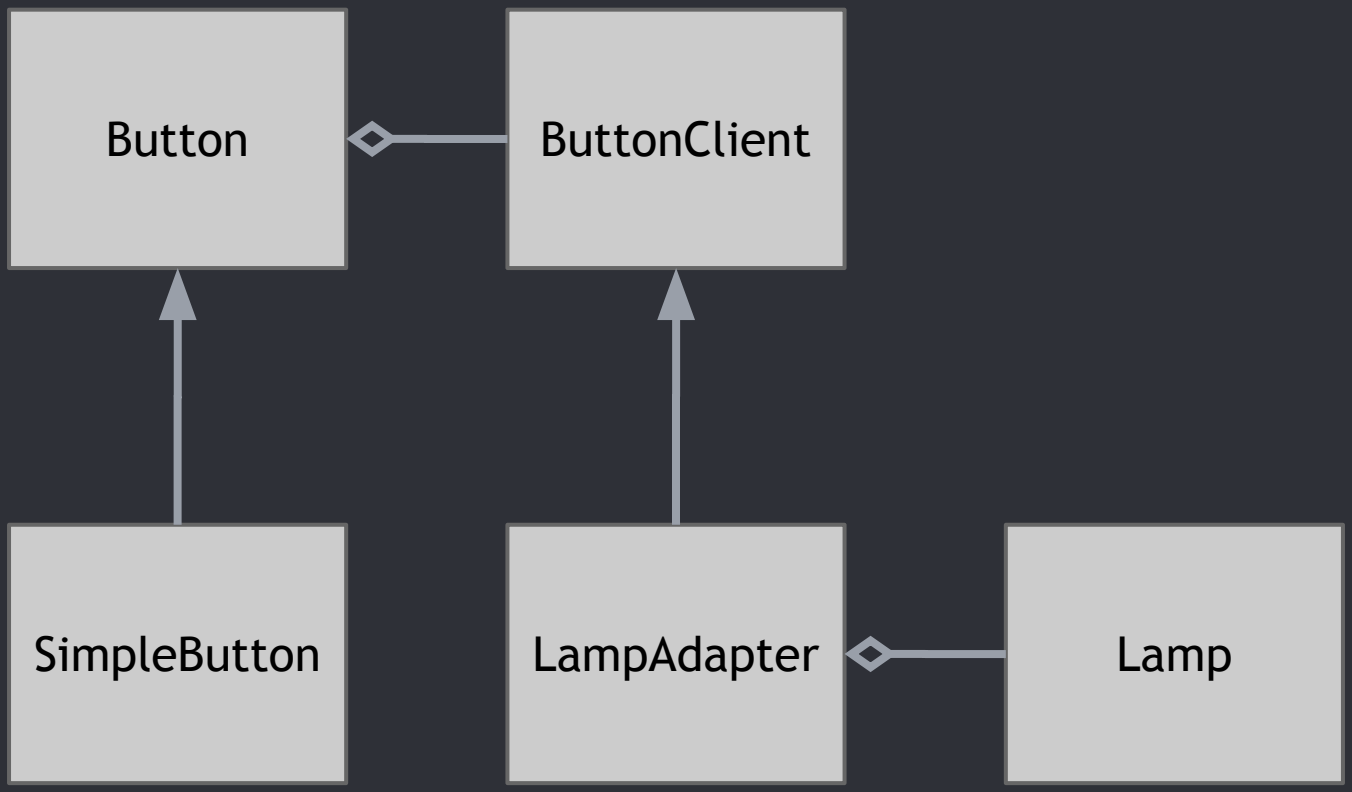
DEPENDENCY INVERSION PRINCIPLE

```
public abstract class ButtonClient {
    public abstract void turnOn();
    public abstract void turnOff();
}

public class Lamp extends ButtonClient {
    … // Lamp implementation
}
```

DEPENDENCY INVERSION PRINCIPLE

```
public abstract class Button {
    public abstract void detect();
}

public class SimpleButton extends Button {
    … // Button implementation
}
```

INTERFACE SEGREGATION PRINCIPLE



4

ΕΥΧΑΡΙΣΤΩ

Ερωτήσεις, και μετά εργαστήριο