

# Cohésion et couplage

# Introduction

- Identifier les niveaux de cohésion et de couplage
- Pour du code plus modulaire et lisible

# La Cohésion

*La cohésion mesure dans quelle mesure les éléments d'un module (ou classe) travaillent ensemble à une seule tâche bien définie.*

# Les types de cohésion

## La bonne cohésion

-  Communicative
-  Séquentielle
-  Fonctionnelle

## La mauvaise cohésion

-  Coïncidentielle
-  Logique
-  Temporelle
-  Procédurale



# La Cohésion Coïncidentielle

- Méthodes/groupes sans lien
- Aucune logique commune

Exemple typique : la classe `xxxUtils`

```
public class MiscUtils {  
    void sendEmail() { /* ... */ }  
    void convertTemperature() { /* ... */ }  
    void calculateTax() { /* ... */ }  
}
```



# La Cohésion Coïncidentielle

- ✕ SRP

⇒ On sépare les responsabilités

```
public class EmailService {  
    void sendEmail() { /* ... */ }  
}  
  
public class TemperatureConverter {  
    void convert(double celsius) { /* ... */ }  
}  
  
public class TaxCalculator {  
    void calculate() { /* ... */ }  
}
```



# La cohésion logique

- Du code qui partage un lien faible

```
public class NotificationManager {  
  
    public void sendEmail(String recipient, String message) {  
        System.out.println("Envoi d'un e-mail à " + recipient);  
    }  
  
    public void sendSms(String phoneNumber, String message) {  
        System.out.println("Envoi d'un SMS à " + phoneNumber);  
    }  
  
    public void logNotification(String type, String message) {  
        System.out.println("Log [" + type + "] : " + message);  
    }  
}
```

# La cohésion logique

- ✕ SRP

⇒ On sépare les responsabilités

```
public class EmailSender {  
    public void send(String recipient, String message) { /* ... */ }  
}  
  
public class SmsSender {  
    public void send(String phoneNumber, String message){ /* ... */ }  
}  
  
public class NotificationLogger {  
    public void log(String type, String message) { /* ... */ }  
}
```

# La cohésion temporelle

- Méthodes exécutées au même moment
- Aucune logique entre elles

```
public class AppInitializer {  
    void initLogger() { /* ... */ }  
    void loadConfig() { /* ... */ }  
    void connectToDatabase() { /* ... */ }  
}
```

# La cohésion temporelle

- ✕ SRP

⇒ On sépare les responsabilités

```
public class AppStartup {  
    void start() {  
        new LoggerInitializer().init();  
        new ConfigLoader().load();  
        new DatabaseConnector().connect();  
    }  
}
```

# La cohésion procédurale

- Méthodes regroupées par ordre d'exécution
- Chaque étape dépend de la précédente

```
public class OrderProcessor {  
    void validateOrder() { /* ... */ }  
    void saveOrder() { /* ... */ }  
    void notifyCustomer() { /* ... */ }  
}
```

# La cohésion procédurale

- ✕ SRP

⇒ On extrait les responsabilités

⇒ On gère l'ordre proprement dans une classe "chef d'orchestre"

```
public class OrderValidator { void validate() {} }  
public class OrderSaver { void save() {} }  
public class CustomerNotifier { void notifyCustomer() {} }  
  
public class OrderProcessingPipeline {  
    void process() {  
        new OrderValidator().validate();  
        new OrderSaver().save();  
        new CustomerNotifier().notifyCustomer();  
    }  
}
```

# La cohésion communicative

- Méthodes qui utilisent les mêmes données

```
public class Invoice {  
    private double total;  
  
    void calculateTotal() { /* ... */ }  
    void printInvoice() { /* ... */ }  
    void exportPdf() { /* ... */ }  
}
```



# La cohésion séquentielle

- Les méthodes forment un pipeline

```
public class PaymentService {  
  
    private final FundsChecker checker = new FundsChecker();  
    private final AccountDebitor debtor = new AccountDebitor();  
    private final TransactionRecorder recorder = new TransactionRecorder();  
    private final EmailNotifier notifier = new EmailNotifier();  
  
    public void process(PaymentRequest request) {  
        if (!checker.hasFunds(request)) { /* ERROR */ }  
        debtor.debit(request);  
        recorder.record(request);  
        notifier.notify(request);  
    }  
}
```



# La cohésion fonctionnelle

- Les méthodes ont une seule fonction bien définie

```
public class TaxCalculator {  
    double calculateTotalTax(Income income) {  
        double base = computeBase(income);  
        return applyRate(base);  
    }  
    private double computeBase(Income income) { /* ... */ }  
    private double applyRate(double base) { /* ... */ }  
}
```

# Le couplage

*Le couplage mesure la dépendance entre deux modules ou classes.*

# Les types de couplage

## Le bon couplage

-  Par les données
-  Par message

## Le mauvais couplage

-  Par le contenu
-  Commun
-  Externe
-  Par contrôle

# Par le contenu

- Accès à l'implémentation interne d'une autre classe

```
public class Engine {  
    private String status = "idle";  
    // ...  
    public void setStatus(String status) {  
        this.status = status;  
    }  
}  
  
public class EngineService {  
    void start(Engine engine) {  
        engine.status = "running";  
    }  
}
```

## Par le contenu

- Tout changement interne dans Engine risque de casser l'EngineService
- Rend le code difficile à modifier et maintenir



# Par le contenu

⇒ On verrouille la classe

```
public class Engine {  
    private String status = "idle";  
  
    public void start() {  
        this.status = "running";  
    }  
  
    public String getStatus() {  
        return status;  
    }  
}  
  
public class EngineService {  
    void start(Engine engine) {  
        engine.start(); // interface propre  
    }  
}
```

# Le couplage commun

- Accès à des variables globales ou statiques partagées

```
public class GlobalCache {  
    public static Map<String, String> data = new HashMap<>();  
}  
  
public class UserService {  
    public void saveUser(String userId) {  
        GlobalCache.data.put(userId, "User saved");  
    }  
}  
  
public class NotificationService {  
    public void notifyUser(String userId) {  
        if (GlobalCache.data.containsKey(userId)) {  
            /* Send notification */  
        }  
    }  
}
```

## Le couplage commun

- Couplage implicite
- ✘ DI : la dépendance n'est pas clairement indiquée par une injection

⇒ On réintègre la fonctionnalité en local, via une classe et de l'injection de dépendance

# Le couplage commun

```
public interface Cache {
    void put(String key, String value);
    boolean contains(String key);
}

public class UserService {
    private final Cache cache;

    public void saveUser(String userId) {
        cache.put(userId, "User saved");
    }
}

public class NotificationService {
    private final Cache cache;

    public void notifyUser(String userId) {
        if (cache.contains(userId)) { /* Send notification */ }
    }
}
```

# Le couplage externe

- Dépendance à un format ou à un système extérieur

```
public class AlertService {  
    public void sendAlert(String number, String message) {  
        Twilio.init(accountId, token);  
        Message.creator(  
            new PhoneNumber(number),  
            new PhoneNumber(sender),  
            message  
        ).create();  
    }  
}
```



# Le couplage externe

- Rend difficile le remplacement par un autre format ou système
- Difficile à tester

# Le couplage externe

⇒ On crée une abstraction (Pattern "Adapter")

```
public interface Notifier {  
    void notify(String number, String message);  
}  
  
public class TwilioNotifier implements Notifier {  
    public void notify(String number, String message) {  
        /* Appel réel à Twilio */  
    }  
}  
  
public class AlertService {  
    private final Notifier notifier;  
  
    public void sendAlert(String number, String message) {  
        notifier.notify(number, message);  
    }  
}
```

# Le couplage par contrôle

- Contrôle du comportement interne d'une autre classe

```
public class ReportGenerator {  
    void generate(boolean asPdf) {  
        if (asPdf) {  
            // générer PDF  
        } else {  
            // générer HTML  
        }  
    }  
}
```

## Le couplage par contrôle

- En *pilotant* la classe par le paramètre, on crée un lien fort entre les deux classes

# Le couplage par contrôle

⇒ On met en place une abstraction (SRP/OCP/DI)

```
public interface ReportFormat {  
    void generate();  
}  
  
public class PdfReport implements ReportFormat {  
    public void generate() { /* PDF */ }  
}  
  
public class HtmlReport implements ReportFormat {  
    public void generate() { /* HTML */ }  
}  
  
public class ReportGenerator {  
    void generate(ReportFormat format) {  
        format.generate();  
    }  
}
```

## Le couplage par les données

- La classe utilise uniquement les données nécessaires

# Le couplage par les données

✗ Exemple incorrect :

On passe un paramètre Order pour utiliser uniquement l'adresse qu'il contient

```
public class Order {  
    public String orderId;  
    public String customerName;  
    public Address deliveryAddress;  
    public List<String> items;  
}  
  
public class ShippingService {  
    public double calculateShippingCost(Order order) {  
        String address = order.deliveryAddress;  
        return address.isDomTom() ? INTERNATIONAL_SHIPPING : NORMAL_SHIPPING;  
    }  
}
```

## Le couplage par les données

⇒ On extrait l'adresse pour simplifier et rendre la fonction réutilisable

```
public class ShippingService {  
    public double calculateShippingCost(Address deliveryAddress) {  
        return address.isDomTom() ? INTERNATIONAL_SHIPPING : NORMAL_SHIPPING;  
    }  
}
```

On peut maintenant l'utiliser un simulateur calculant les frais de livraisons par exemple.



# Le couplage par messages

- Les classes communiquent via des interfaces, évènements ou messages
- Le destinataire n'a pas besoin d'être connu

⇒ Pattern "Observer"



# Le couplage par messages

```
public interface EventListener {
    void onEvent(String event);
}

public class AuditLogger implements EventListener {
    public void onEvent(String event) {
        System.out.println("Audit: " + event);
    }
}

public class EventBus {
    private final List<EventListener> listeners = new ArrayList<>();

    void register(EventListener listener) {
        listeners.add(listener);
    }

    void send(String event) {
        listeners.forEach(l -> l.onEvent(event));
    }
}
```

# Cohésion vs Couplage

- On veut une haute cohésion
  - Chaque module fait une seule chose et le fait bien
- On veut un faible couplage
  - Chaque module dépend des autres via un couplage propre

Merci de votre attention

