PEMROGRAMAN LANJUT

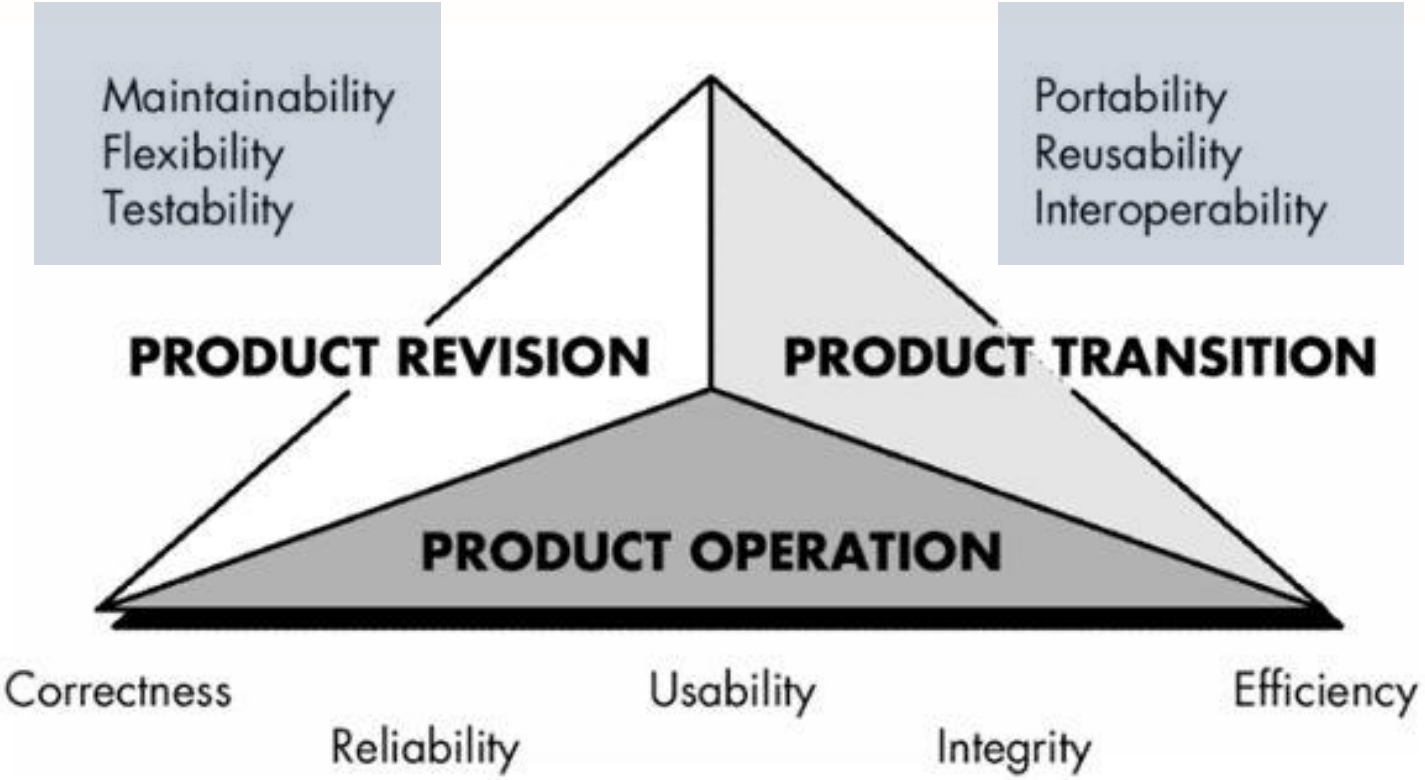
Programming Principles - 2

Oleh Politeknik Elektronika Negeri Surabaya

2021

Politeknik Elektronika Negeri Surabaya
Departemen Teknik Informatika dan Komputer

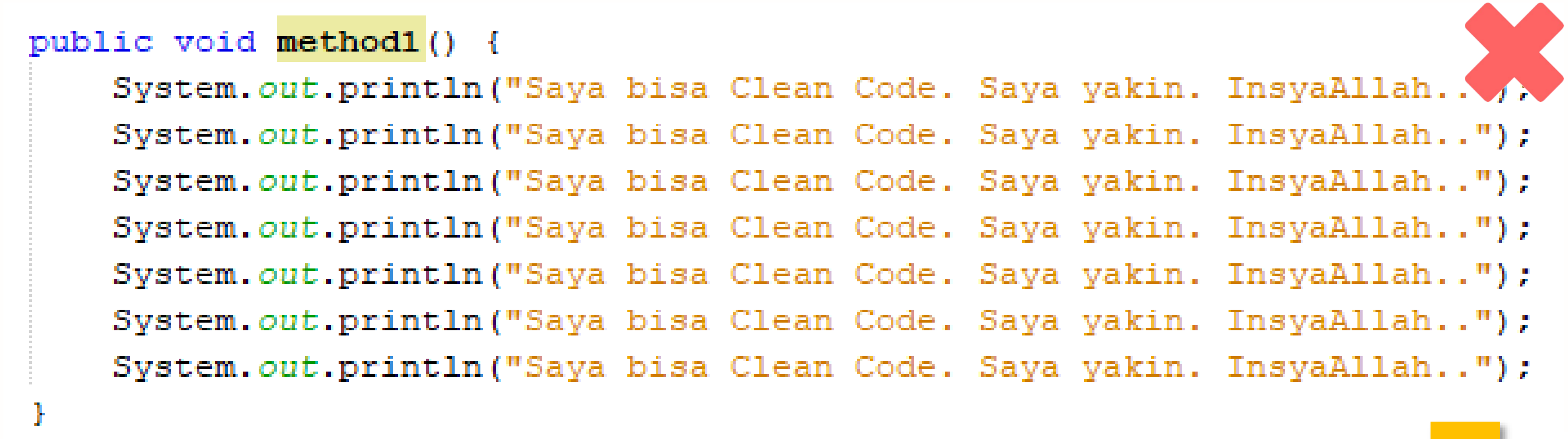
Review



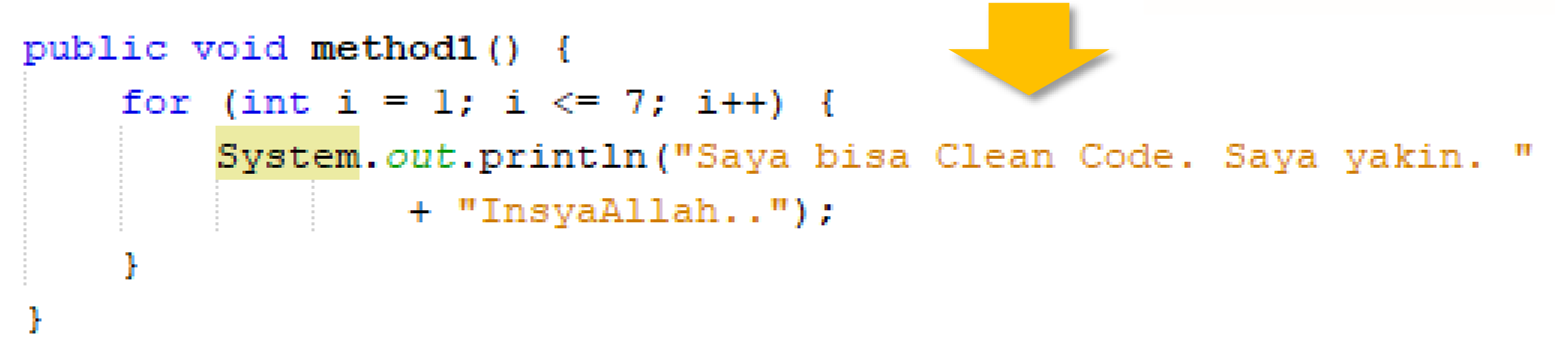
Mc Call Software Quality Metric

DRY (Don’t Repeat Yourself)

- Find and eliminate duplication wherever you can.



```
public void method1() {
    System.out.println("Saya bisa Clean Code. Saya yakin. InsyaAllah..");
    System.out.println("Saya bisa Clean Code. Saya yakin. InsyaAllah..");
    System.out.println("Saya bisa Clean Code. Saya yakin. InsyaAllah..");
    System.out.println("Saya bisa Clean Code. Saya yakin. InsyaAllah..");
    System.out.println("Saya bisa Clean Code. Saya yakin. InsyaAllah..");
    System.out.println("Saya bisa Clean Code. Saya yakin. InsyaAllah..");
    System.out.println("Saya bisa Clean Code. Saya yakin. InsyaAllah..");
}
```

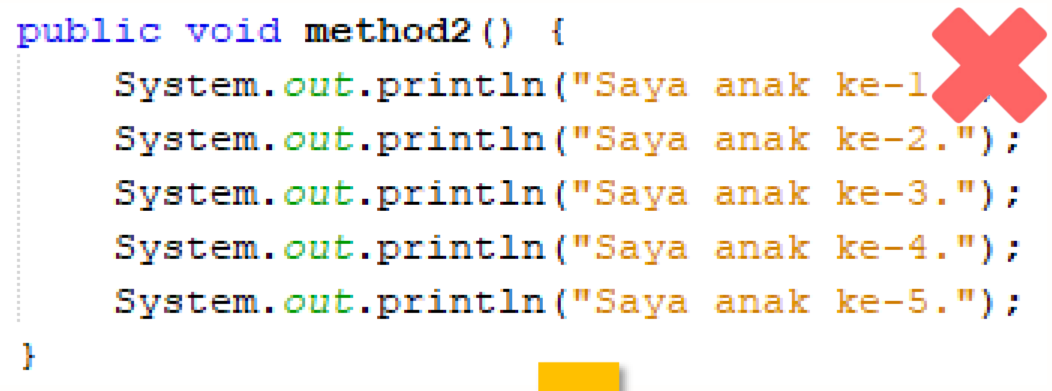


```
public void method1() {
    for (int i = 1; i <= 7; i++) {
        System.out.println("Saya bisa Clean Code. Saya yakin. "
                + "InsyaAllah..");
    }
}
```

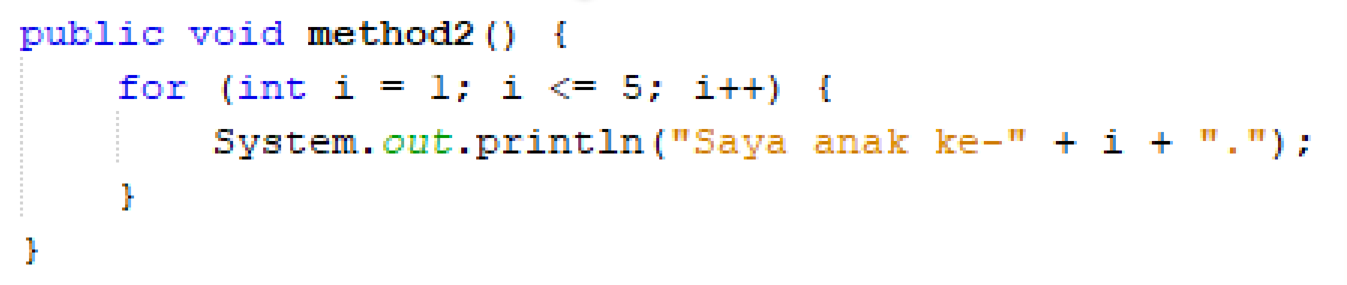
Smells:

- Duplicate code
- Data clumps

DRY (Don’t Repeat Yourself)

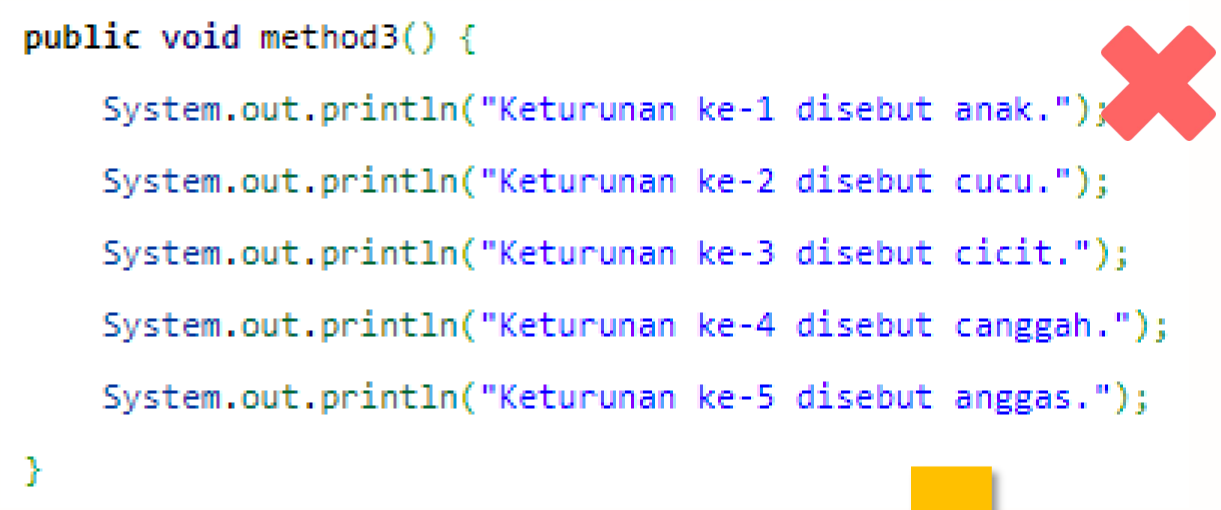


```
public void method2() {
    System.out.println("Saya anak ke-1.");
    System.out.println("Saya anak ke-2.");
    System.out.println("Saya anak ke-3.");
    System.out.println("Saya anak ke-4.");
    System.out.println("Saya anak ke-5.");
}
```

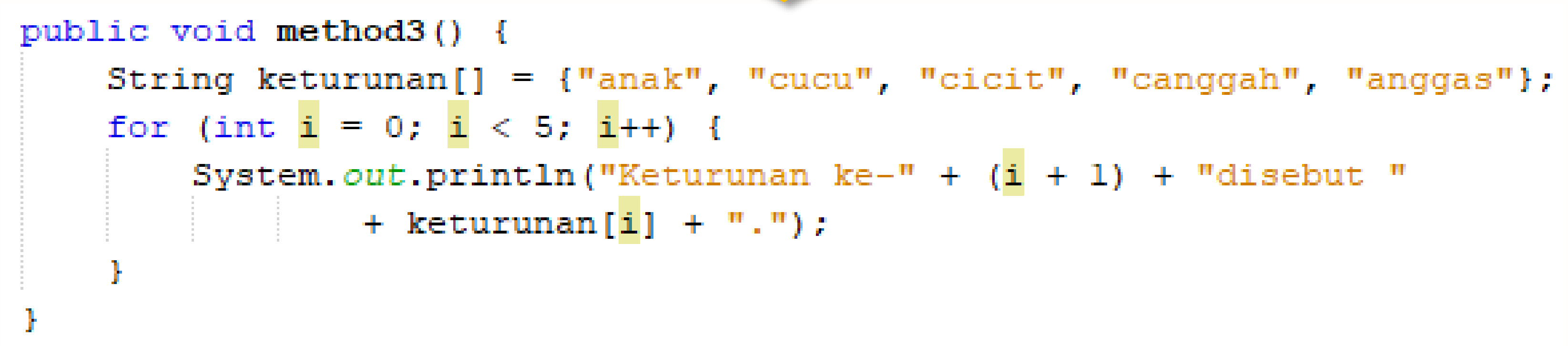


```
public void method2() {
    for (int i = 1; i <= 5; i++) {
        System.out.println("Saya anak ke-" + i + ".");
    }
}
```

DRY (Don’t Repeat Yourself)



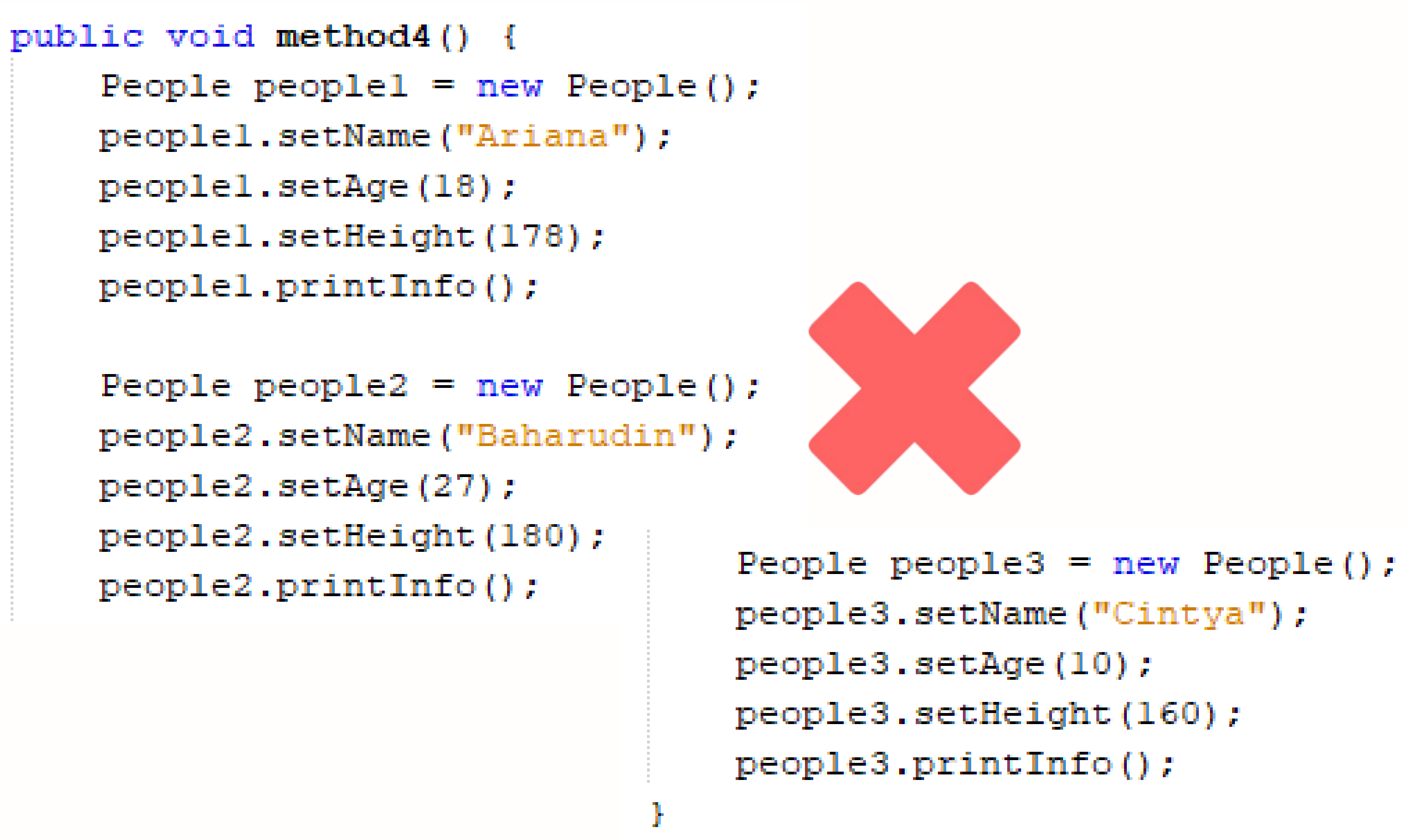
```
public void method3() {
    System.out.println("Keturunan ke-1 disebut anak.");
    System.out.println("Keturunan ke-2 disebut cucu.");
    System.out.println("Keturunan ke-3 disebut cicit.");
    System.out.println("Keturunan ke-4 disebut canggah.");
    System.out.println("Keturunan ke-5 disebut anggas.");
}
```



```
public void method3() {
    String keturunan[] = {"anak", "cucu", "cicit", "canggah", "anggas"};
    for (int i = 0; i < 5; i++) {
        System.out.println("Keturunan ke-" + (i + 1) + "disebut "
                + keturunan[i] + ".");
    }
}
```

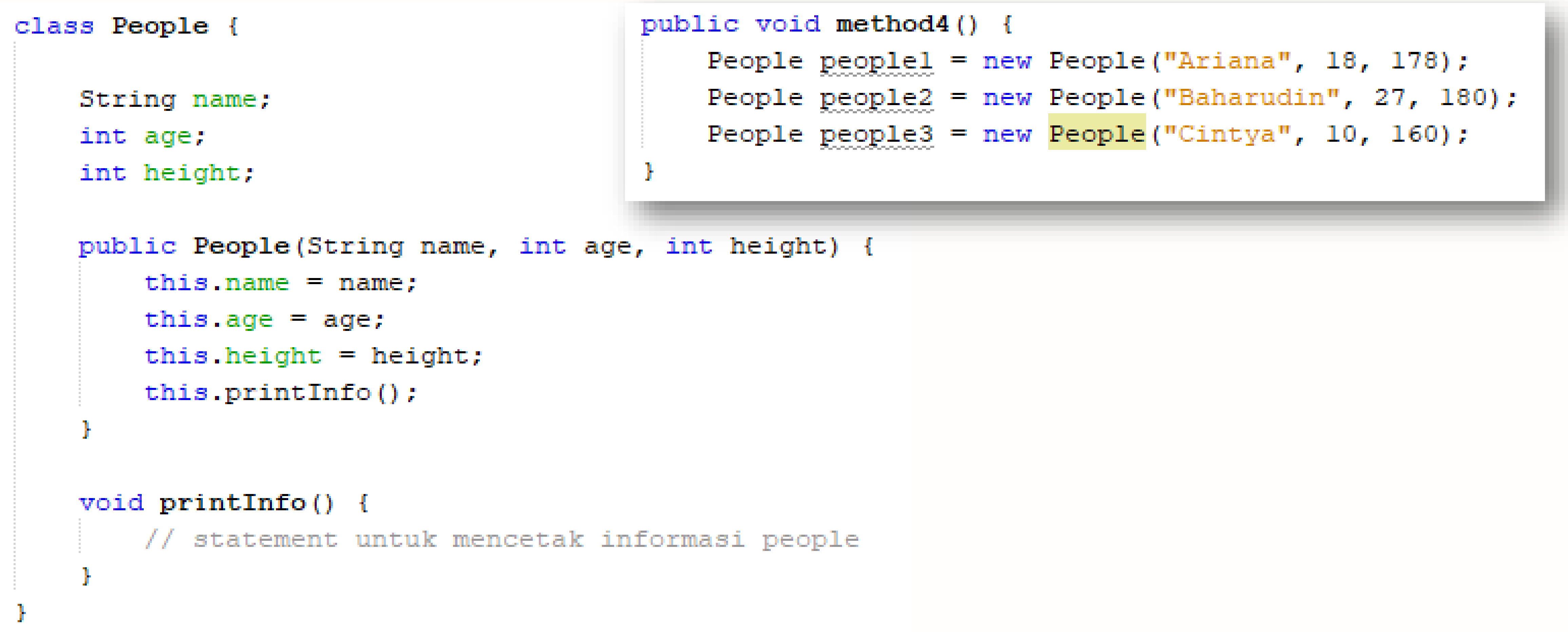
DRY (Don’t Repeat Yourself)

- The most obvious form of duplication is when you have clumps of identical code in various places.



```java
public void method4() {
    People people1 = new People();
    people1.setName("Ariana");
    people1.setAge(18);
    people1.setHeight(178);
    people1.printInfo();

    People people2 = new People();
    people2.setName("Baharudin");
    people2.setAge(27);
    people2.setHeight(180);
    people2.printInfo();

    People people3 = new People();
    people3.setName("Cintya");
    people3.setAge(10);
    people3.setHeight(160);
    people3.printInfo();
}
```

DRY (Don’t Repeat Yourself)

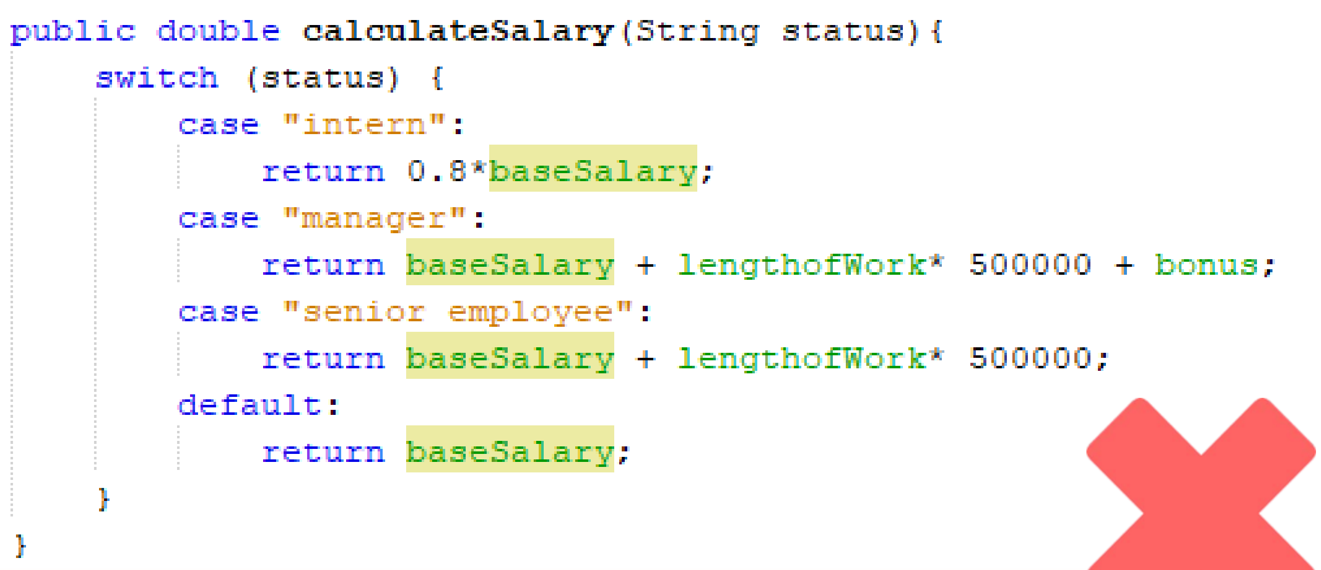


```java
class People {

    String name;
    int age;
    int height;

    public People(String name, int age, int height) {
        this.name = name;
        this.age = age;
        this.height = height;
        this.printInfo();
    }

    void printInfo() {
        // statement untuk mencetak informasi people
    }
}
```

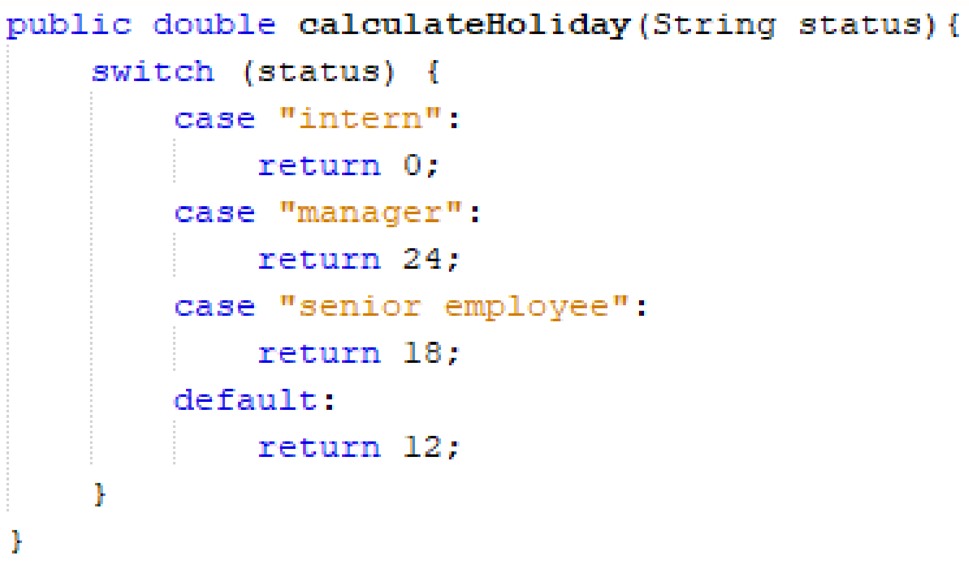
```java
public void method4() {
    People people1 = new People("Ariana", 18, 178);
    People people2 = new People("Baharudin", 27, 180);
    People people3 = new People("Cintya", 10, 160);
}
```

DRY (Don’t Repeat Yourself)

- A more subtle form is the switch/case or if/else chain that appears again and again in various modules, always testing for the same set of conditions. These should be replaced with polymorphism.



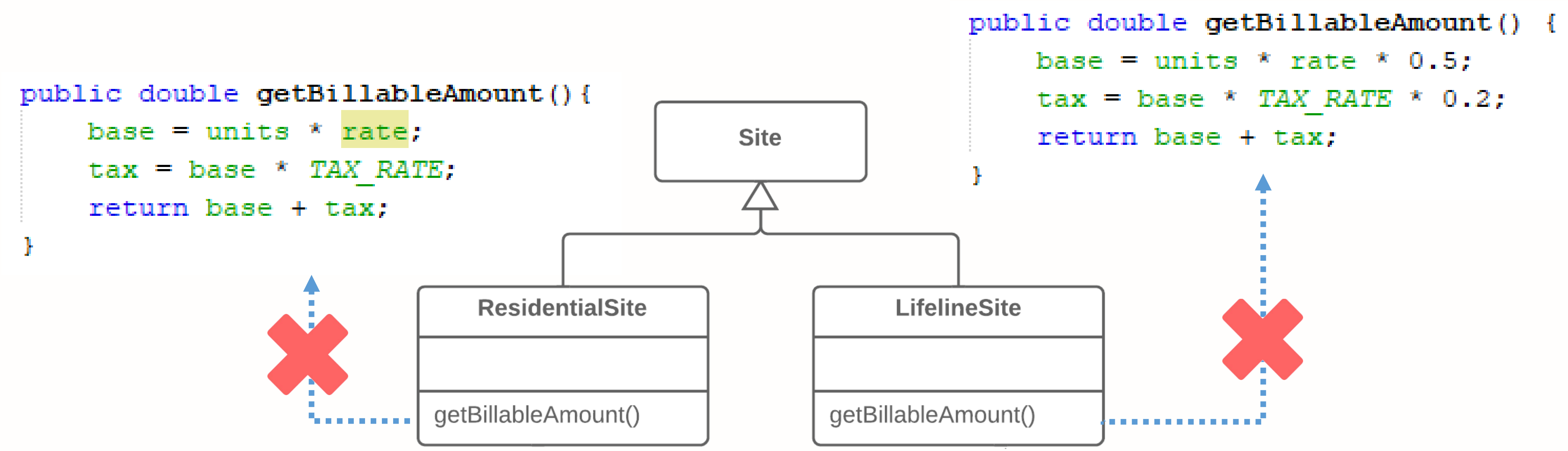
```
public double calculateSalary(String status){
    switch (status) {
        case "intern":
            return 0.8*baseSalary;
        case "manager":
            return baseSalary + lengthofWork* 500000 + bonus;
        case "senior employee":
            return baseSalary + lengthofWork* 500000;
        default:
            return baseSalary;
    }
}
```



```
public double calculateHoliday(String status){
    switch (status) {
        case "intern":
            return 0;
        case "manager":
            return 24;
        case "senior employee":
            return 18;
        default:
            return 12;
    }
}
```

DRY (Don’t Repeat Yourself)

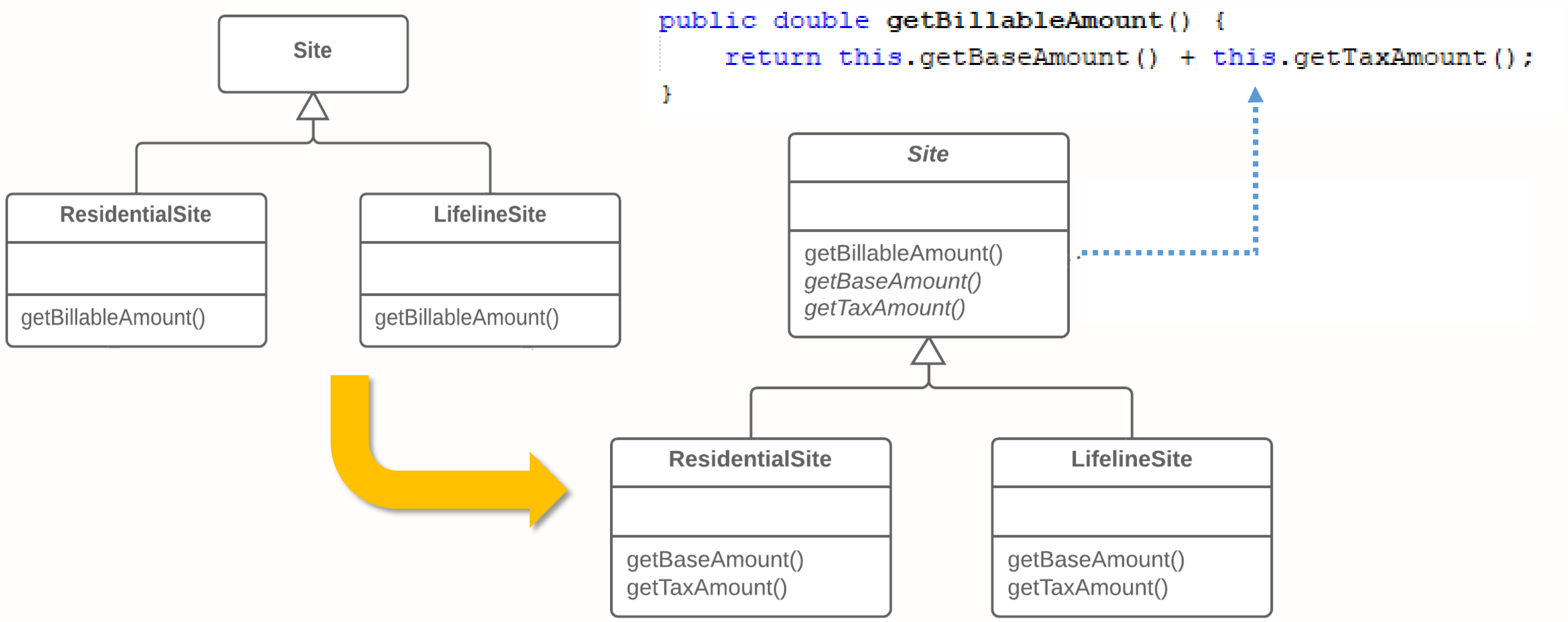
- Still more subtle are the modules that have similar algorithms, but that don’t share similar lines of code. This is still duplication.



```
public double getBillableAmount(){
    base = units * rate;
    tax = base * TAX_RATE;
    return base + tax;
}
```

```
public double getBillableAmount() {
    base = units * rate * 0.5;
    tax = base * TAX_RATE * 0.2;
    return base + tax;
}
```

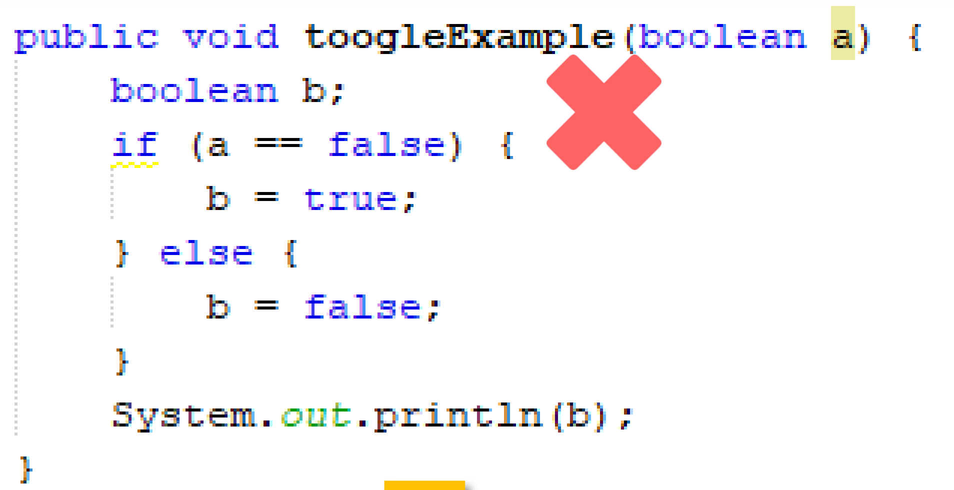
DRY (Don’t Repeat Yourself)



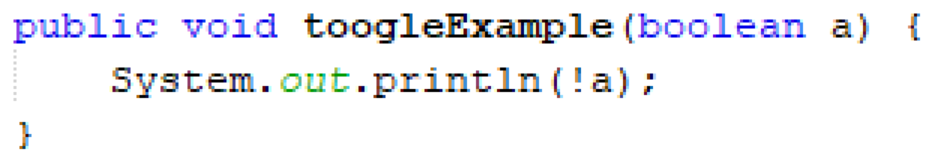
```
public double getBillableAmount() {
    return this.getBaseAmount() + this.getTaxAmount();
}
```

KISS (Keep It Simple St**id)

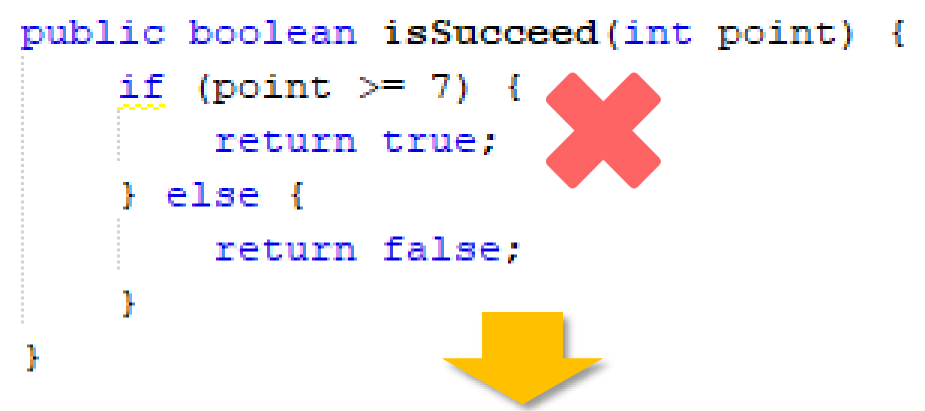
- Keep the code simple and clear, making it easy to understand.



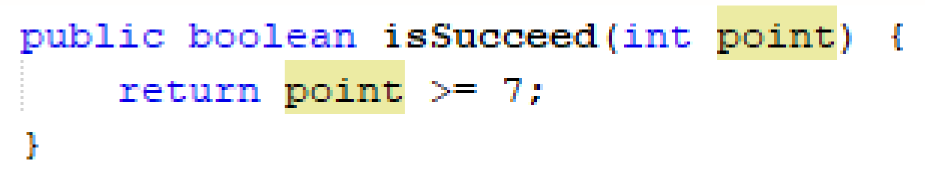
```
public void toogleExample(boolean a) {
    boolean b;
    if (a == false) {
        b = true;
    } else {
        b = false;
    }
    System.out.println(b);
}
```



```
public void toogleExample(boolean a) {
    System.out.println(!a);
}
```

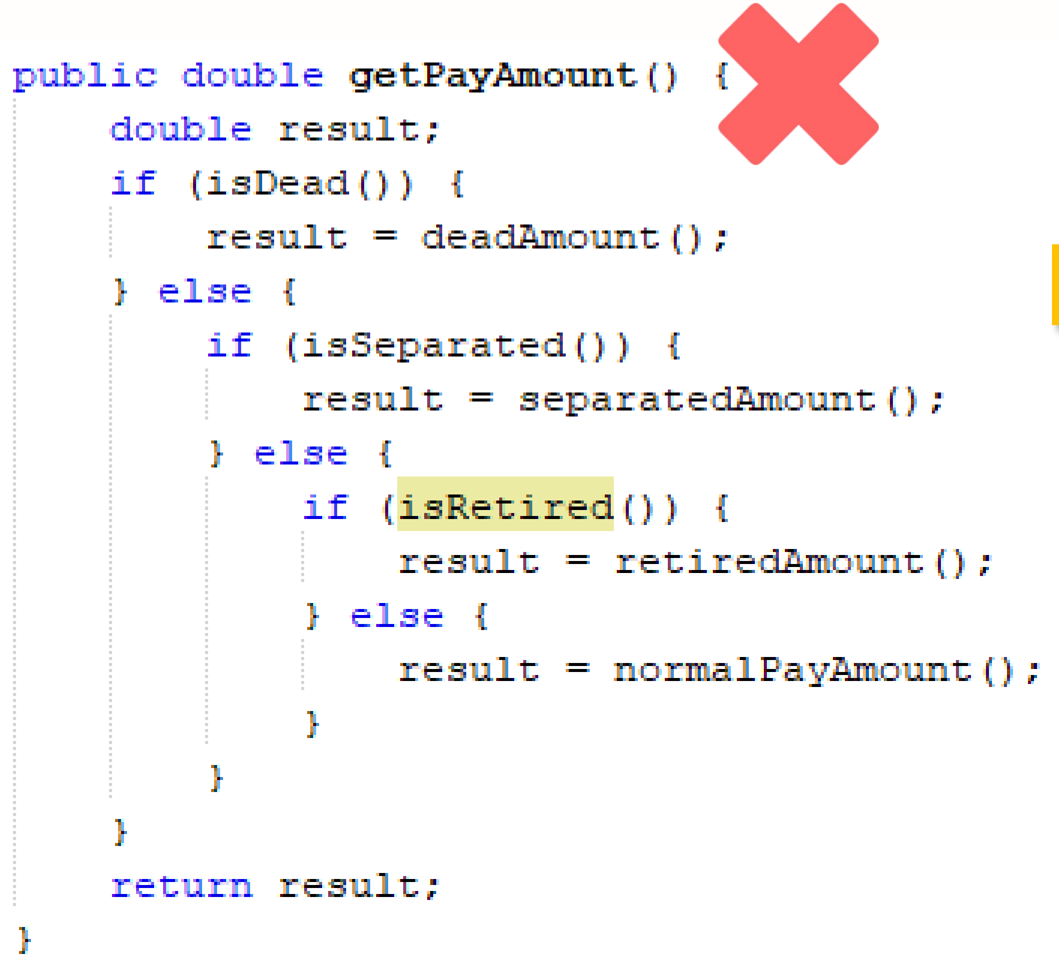


```
public boolean isSucceed(int point) {
    if (point >= 7) {
        return true;
    } else {
        return false;
    }
}
```

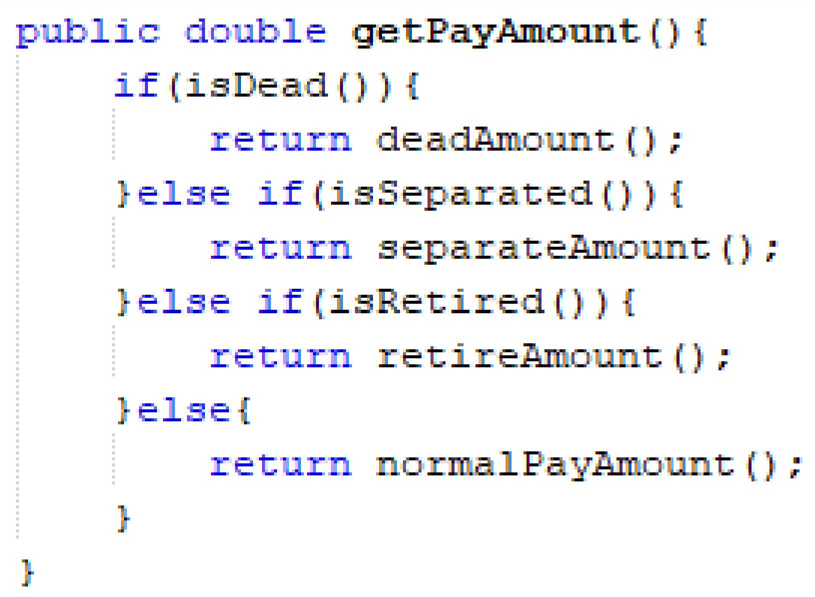


```
public boolean isSucceed(int point) {
    return point >= 7;
}
```

KISS (Keep It Simple St**id)

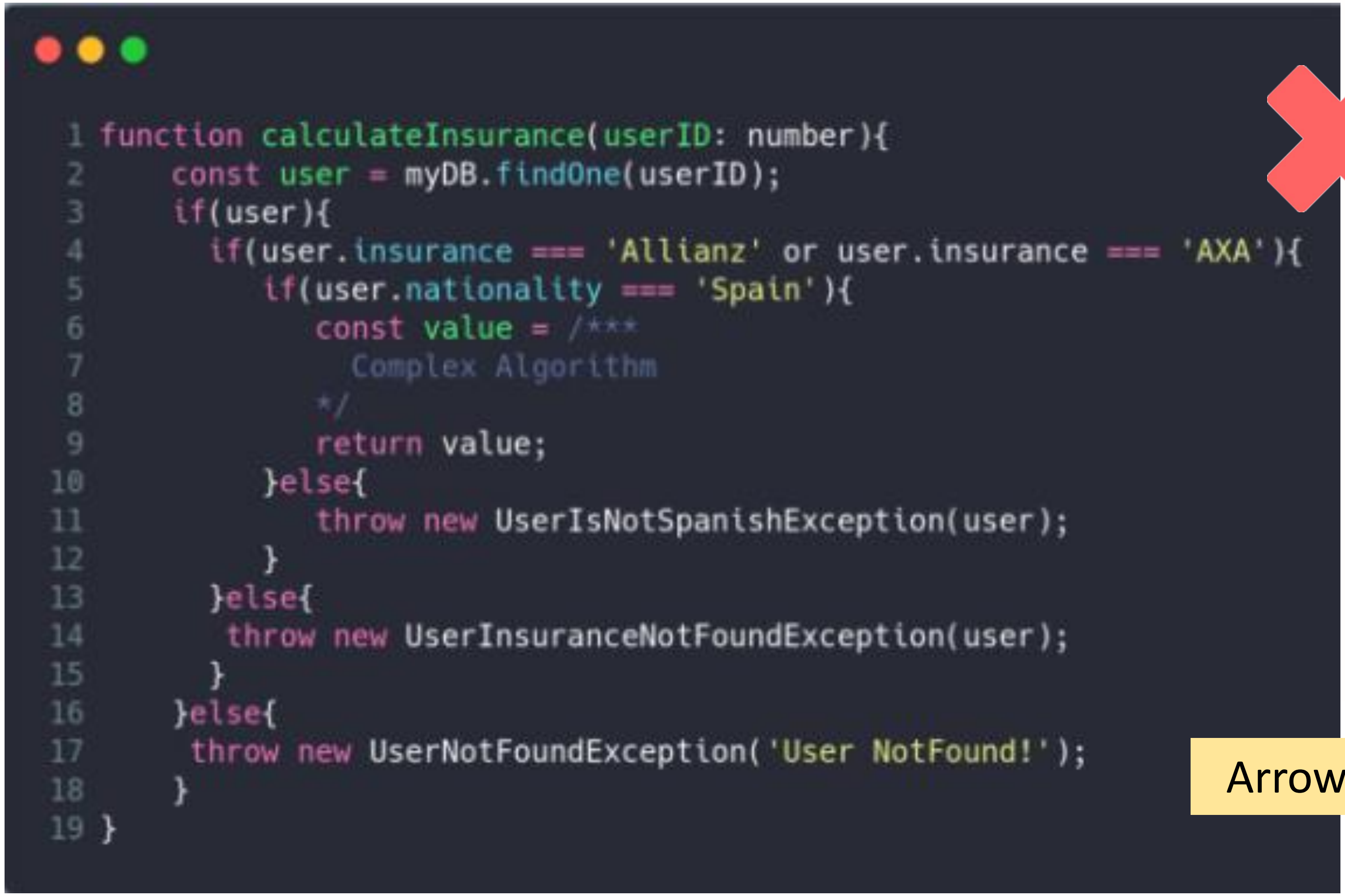


```
public double getPayAmount() {
    double result;
    if (isDead()) {
        result = deadAmount();
    } else {
        if (isSeparated()) {
            result = separatedAmount();
        } else {
            if (isRetired()) {
                result = retiredAmount();
            } else {
                result = normalPayAmount();
            }
        }
    }
    return result;
}
```



```
public double getPayAmount(){
    if(isDead()){
        return deadAmount();
    }else if(isSeparated()){
        return separateAmount();
    }else if(isRetired()){
        return retireAmount();
    }else{
        return normalPayAmount();
    }
}
```

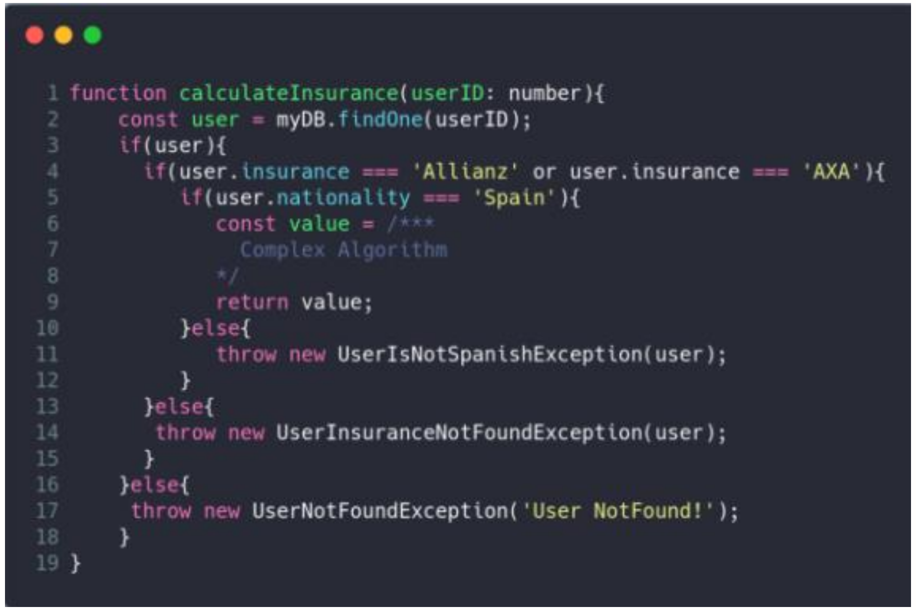
KISS (Keep It Simple St**id)



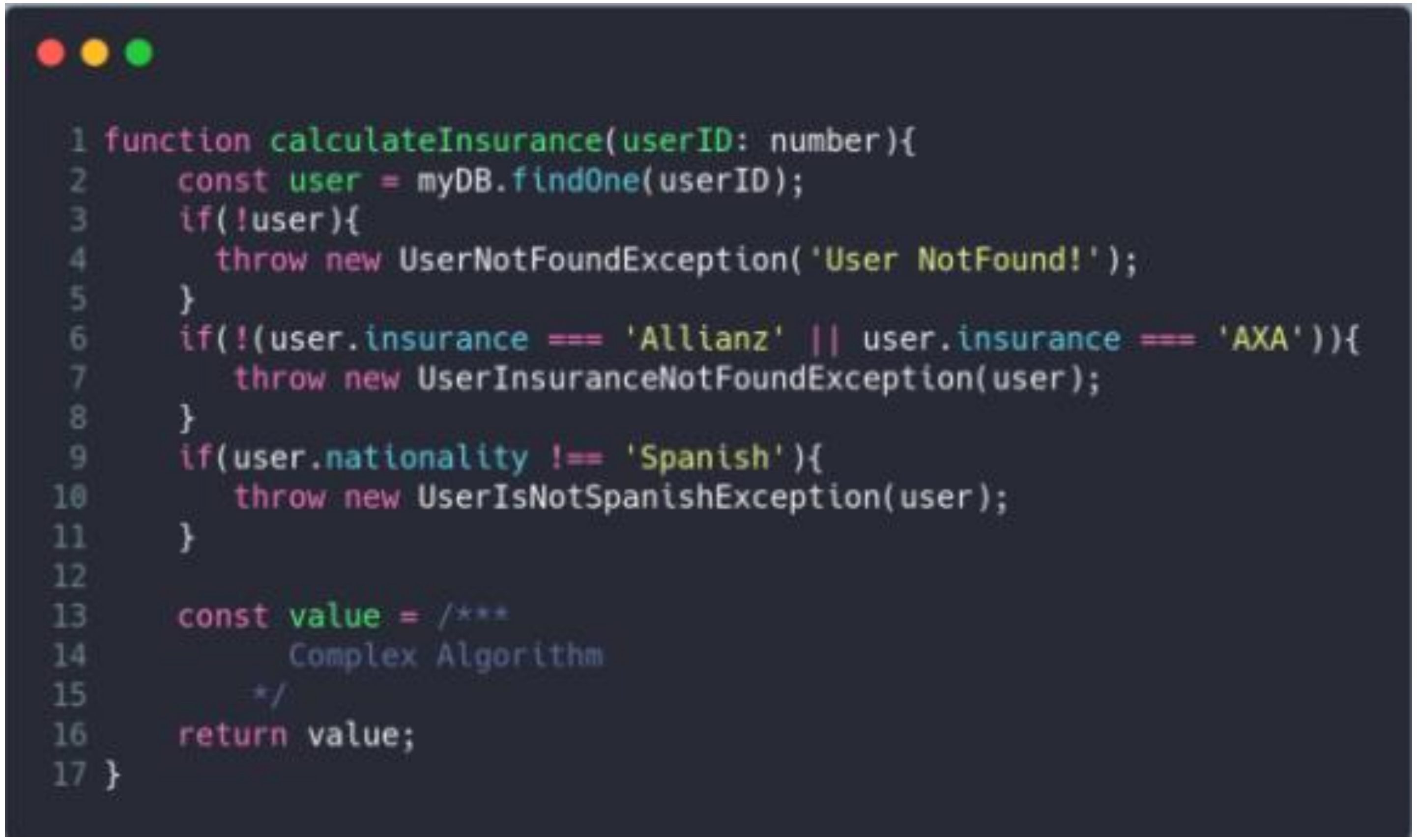
```
function calculateInsurance(userID: number){
    const user = myDB.findOne(userID);
    if(user){
      if(user.insurance === 'Allianz' or user.insurance === 'AXA'){
         if(user.nationality === 'Spain'){
            const value = /***
              Complex Algorithm
            */
            return value;
         }else{
            throw new UserIsNotSpanishException(user);
         }
      }else{
       throw new UserInsuranceNotFoundException(user);
      }
    }else{
     throw new UserNotFoundException('User NotFound!');
    }
}
```

Arrow Anti-pattern

KISS (Keep It Simple St**id)

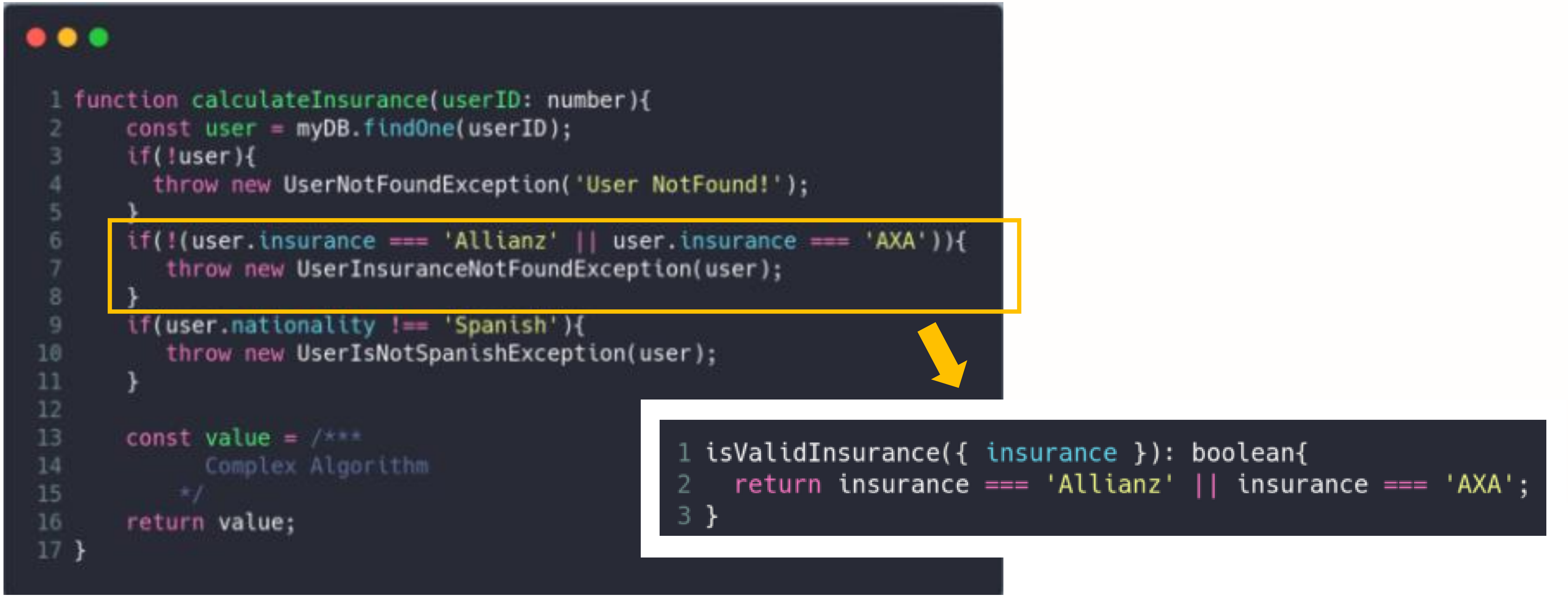


```
function calculateInsurance(userID: number){
    const user = myDB.findOne(userID);
    if(user){
      if(user.insurance === 'Allianz' or user.insurance === 'AXA'){
         if(user.nationality === 'Spain'){
            const value = /***
              Complex Algorithm
            */
            return value;
         }else{
            throw new UserIsNotSpanishException(user);
         }
      }else{
       throw new UserInsuranceNotFoundException(user);
      }
    }else{
     throw new UserNotFoundException('User NotFound!');
    }
}
```



```
function calculateInsurance(userID: number){
     const user = myDB.findOne(userID);
     if(!user){
       throw new UserNotFoundException('User NotFound!');
     }
     if(!(user.insurance === 'Allianz' || user.insurance === 'AXA')){
        throw new UserInsuranceNotFoundException(user);
     }
     if(user.nationality !== 'Spanish'){
        throw new UserIsNotSpanishException(user);
     }

     const value = /***
           Complex Algorithm
         */
     return value;
}
```

KISS (Keep It Simple St**id)



```
function calculateInsurance(userID: number){
    const user = myDB.findOne(userID);
    if(!user){
      throw new UserNotFoundException('User NotFound!');
    }
    if(!(user.insurance === 'Allianz' || user.insurance === 'AXA')){
       throw new UserInsuranceNotFoundException(user);
    }
    if(user.nationality !== 'Spanish'){
       throw new UserIsNotSpanishException(user);
    }

    const value = /***
          Complex Algorithm
        */
    return value;
}
```

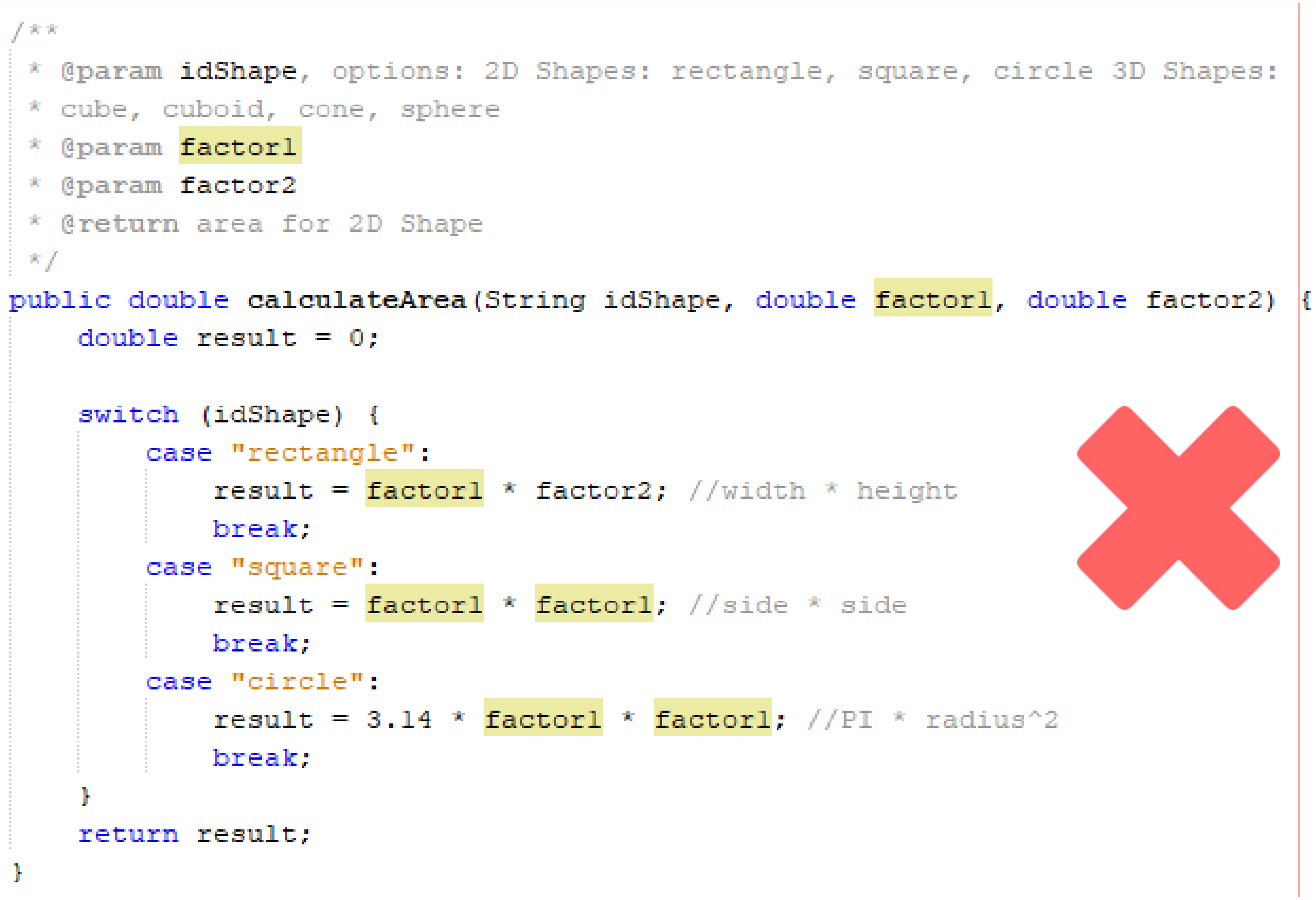
```
isValidInsurance({ insurance }): boolean{
  return insurance === 'Allianz' || insurance === 'AXA';
}
```

YAGNI (You Are Not Gonna Need It)

- Remove any parts which are unnecessary.
- Do not implement something until it is needed.

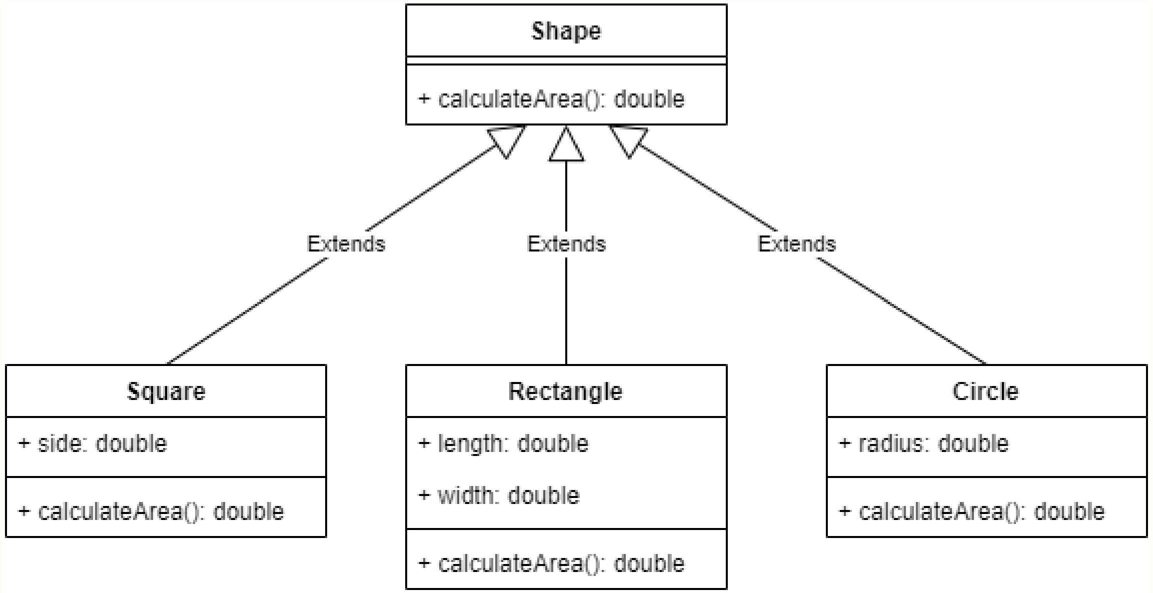
Smells:

- Dead code
- Speculative generality
- Lazy class
- Comments



```
/**
 * @param idShape, options: 2D Shapes: rectangle, square, circle 3D Shapes:
 * cube, cuboid, cone, sphere
 * @param factor1
 * @param factor2
 * @return area for 2D Shape
 */
public double calculateArea(String idShape, double factor1, double factor2) {
    double result = 0;

    switch (idShape) {
        case "rectangle":
            result = factor1 * factor2; //width * height
            break;
        case "square":
            result = factor1 * factor1; //side * side
            break;
        case "circle":
            result = 3.14 * factor1 * factor1; //PI * radius^2
            break;
    }
    return result;
}
```

YAGNI (You Are Not Gonna Need It)



Shape

+ calculateArea(): double

Extends

Extends

Extends

Square

+ side: double

+ calculateArea(): double

Rectangle

+ length: double

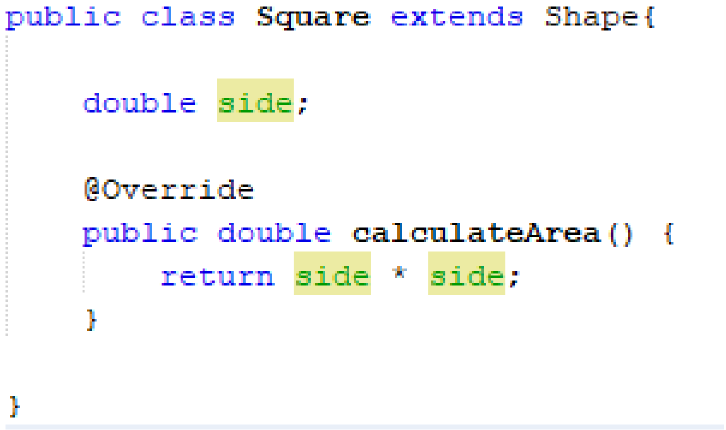
+ width: double

+ calculateArea(): double

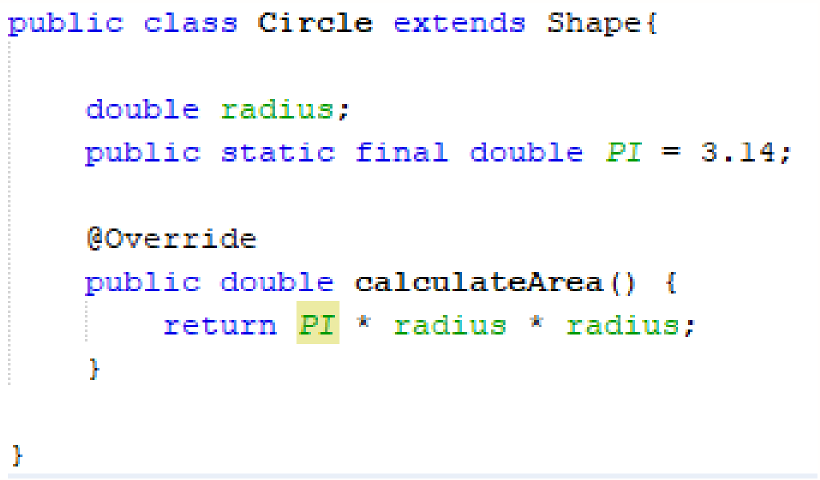
Circle

+ radius: double

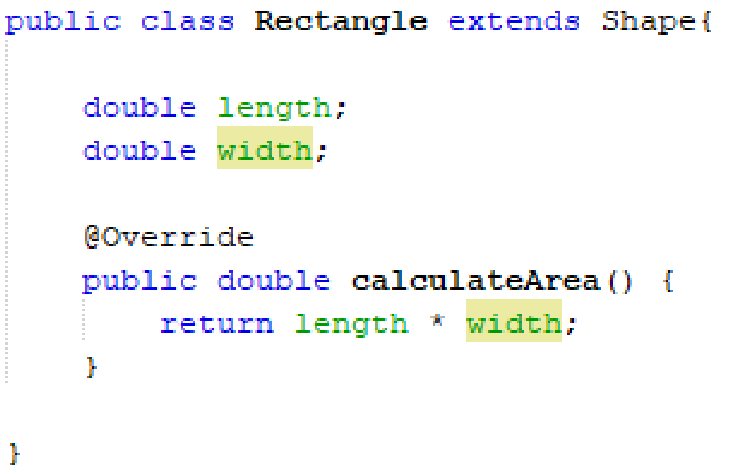
+ calculateArea(): double



```java
public class Square extends Shape{

    double side;

    @Override
    public double calculateArea() {
        return side * side;
    }

}
```



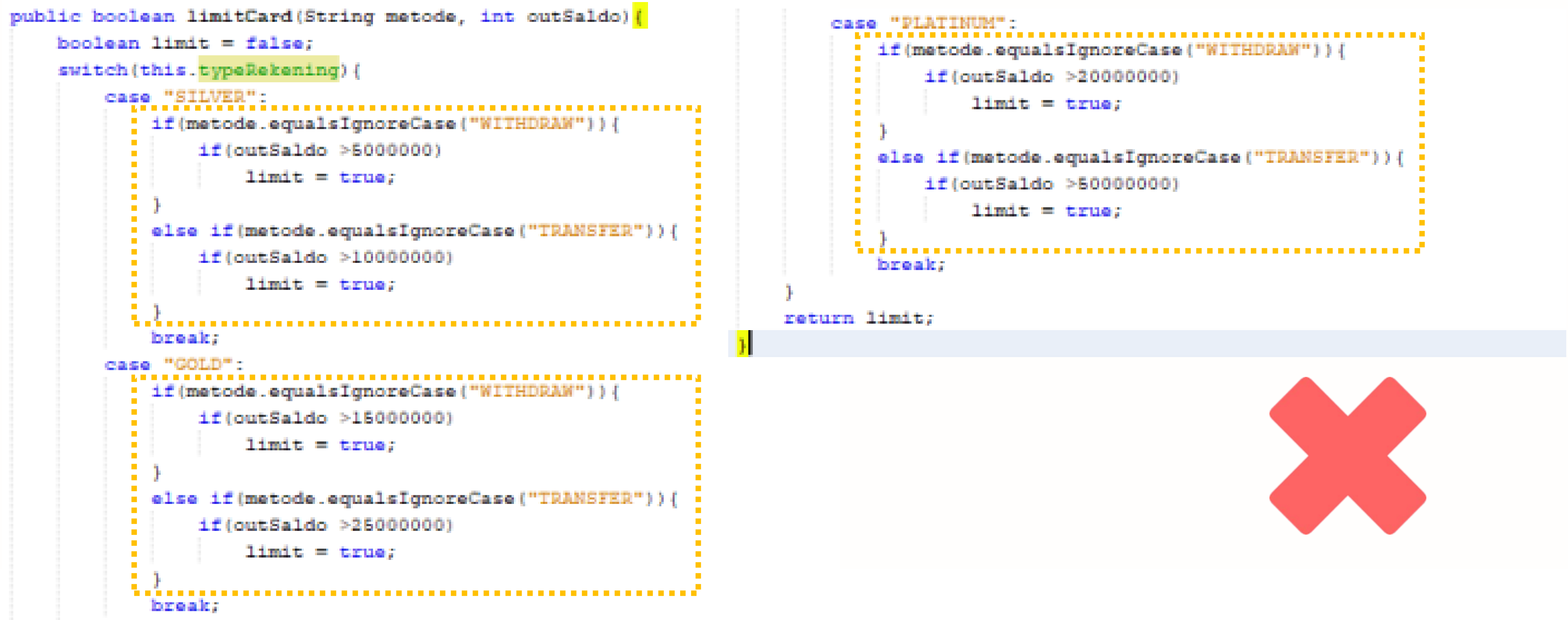
```java
public class Circle extends Shape{

    double radius;
    public static final double PI = 3.14;

    @Override
    public double calculateArea() {
        return PI * radius * radius;
    }

}
```



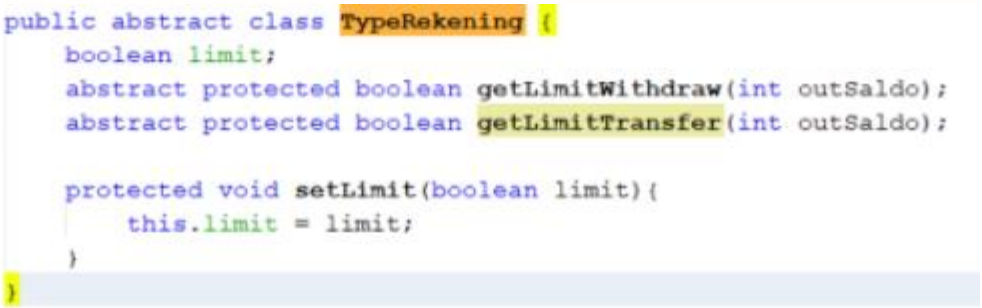
```java
public class Rectangle extends Shape{

    double length;
    double width;

    @Override
    public double calculateArea() {
        return length * width;
    }

}
```

Addendum

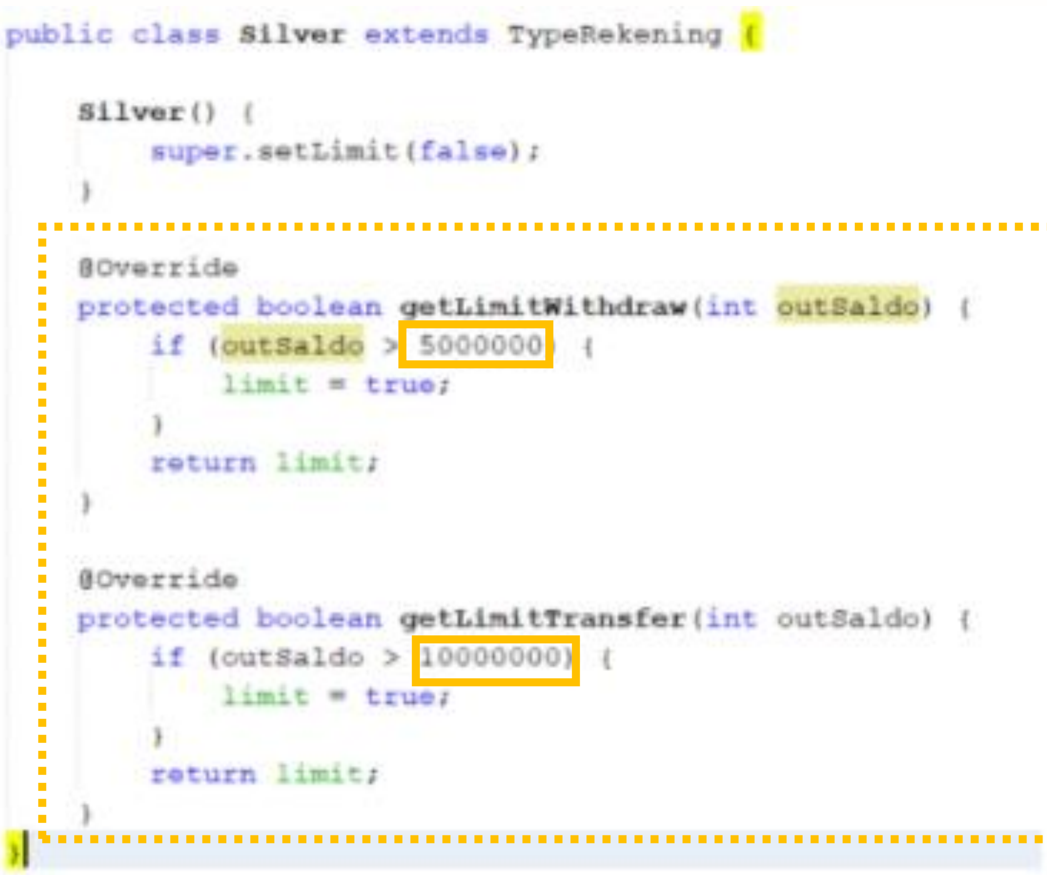
- DRY implementation in case study which is discussed at the previous meeting.



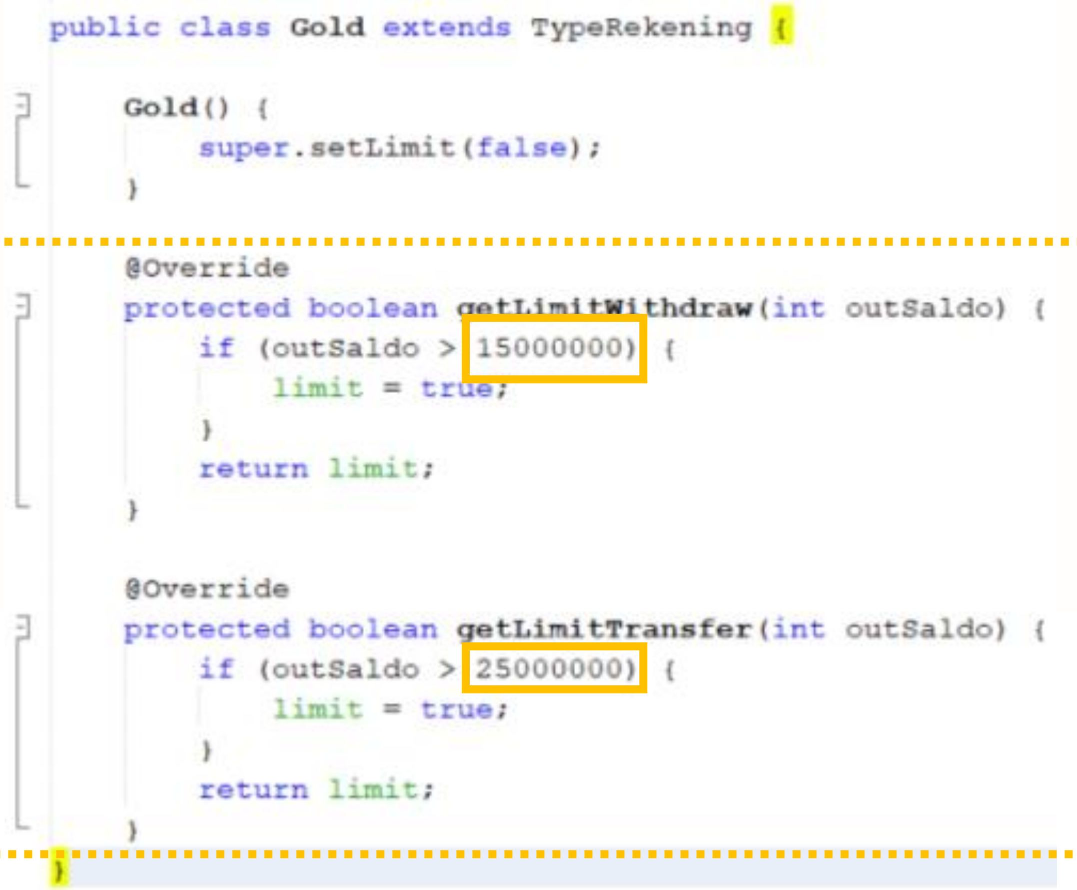
```
public boolean limitCard(String metode, int outSaldo){
    boolean limit = false;
    switch(this.typeRekening){
        case "SILVER":
            if(metode.equalsIgnoreCase("WITHDRAW")){
                if(outSaldo >5000000)
                    limit = true;
            }
            else if(metode.equalsIgnoreCase("TRANSFER")){
                if(outSaldo >10000000)
                    limit = true;
            }
            break;
        case "GOLD":
            if(metode.equalsIgnoreCase("WITHDRAW")){
                if(outSaldo >15000000)
                    limit = true;
            }
            else if(metode.equalsIgnoreCase("TRANSFER")){
                if(outSaldo >25000000)
                    limit = true;
            }
            break;
        case "PLATINUM":
            if(metode.equalsIgnoreCase("WITHDRAW")){
                if(outSaldo >20000000)
                    limit = true;
            }
            else if(metode.equalsIgnoreCase("TRANSFER")){
                if(outSaldo >50000000)
                    limit = true;
            }
            break;
    }
    return limit;
}
```



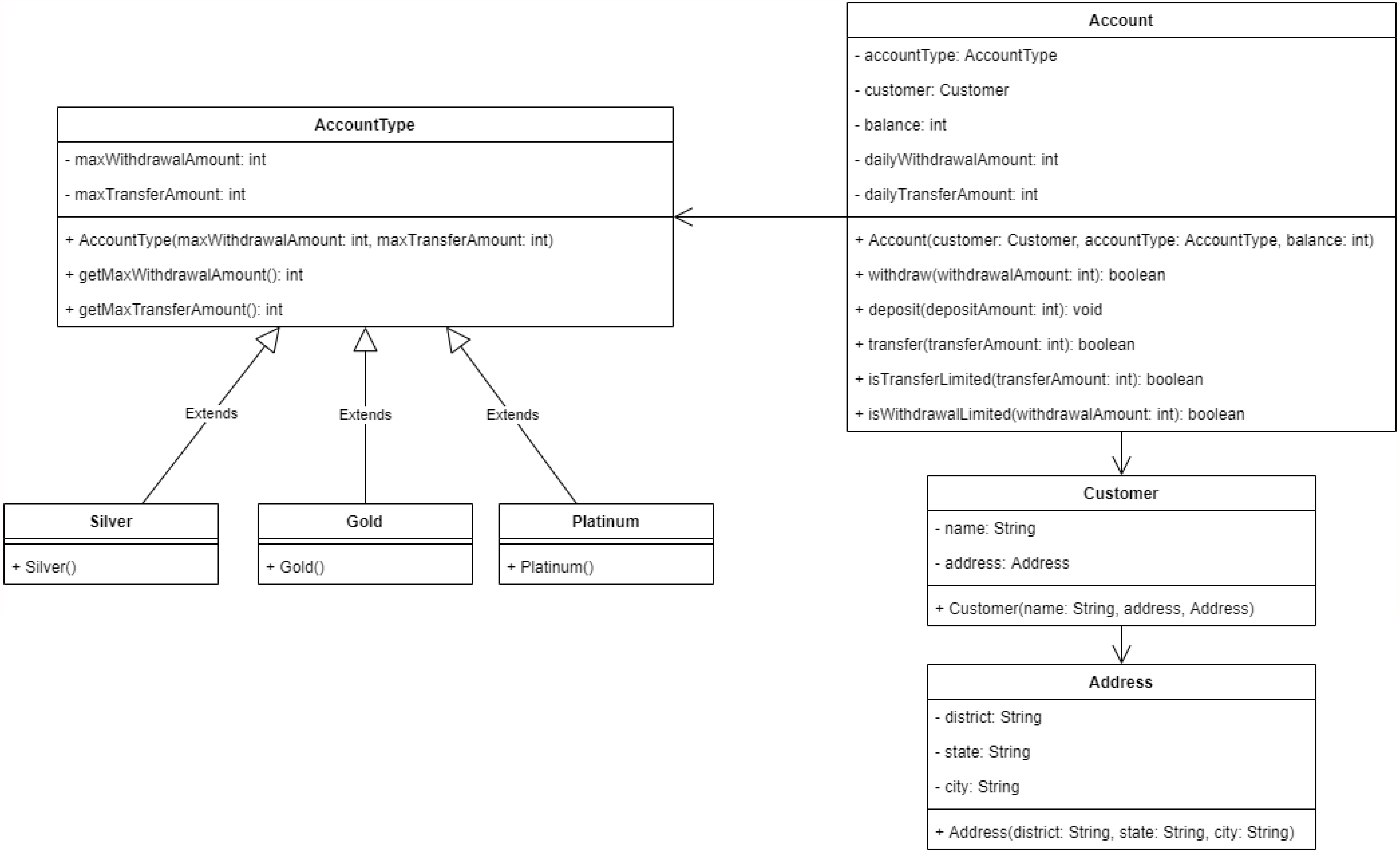
```
public abstract class TypeRekening {
    boolean limit;
    abstract protected boolean getLimitWithdraw(int outSaldo);
    abstract protected boolean getLimitTransfer(int outSaldo);

    protected void setLimit(boolean limit){
        this.limit = limit;
    }
}
```



```
public class Silver extends TypeRekening {

    Silver() {
        super.setLimit(false);
    }

    @Override
    protected boolean getLimitWithdraw(int outSaldo) {
        if (outSaldo > 5000000) {
            limit = true;
        }
        return limit;
    }

    @Override
    protected boolean getLimitTransfer(int outSaldo) {
        if (outSaldo > 10000000) {
            limit = true;
        }
        return limit;
    }
}
```



```
public class Gold extends TypeRekening {

    Gold() {
        super.setLimit(false);
    }

    @Override
    protected boolean getLimitWithdraw(int outSaldo) {
        if (outSaldo > 15000000) {
            limit = true;
        }
        return limit;
    }

    @Override
    protected boolean getLimitTransfer(int outSaldo) {
        if (outSaldo > 25000000) {
            limit = true;
        }
        return limit;
    }
}
```



Account

- accountType: AccountType
- customer: Customer
- balance: int
- dailyWithdrawalAmount: int
- dailyTransferAmount: int

+ Account(customer: Customer, accountType: AccountType, balance: int)
+ withdraw(withdrawalAmount: int): boolean
+ deposit(depositAmount: int): void
+ transfer(transferAmount: int): boolean
+ isTransferLimited(transferAmount: int): boolean
+ isWithdrawalLimited(withdrawalAmount: int): boolean

AccountType

- maxWithdrawalAmount: int
- maxTransferAmount: int

+ AccountType(maxWithdrawalAmount: int, maxTransferAmount: int)
+ getMaxWithdrawalAmount(): int
+ getMaxTransferAmount(): int

Silver

+ Silver()

Gold

+ Gold()

Platinum

+ Platinum()

Customer

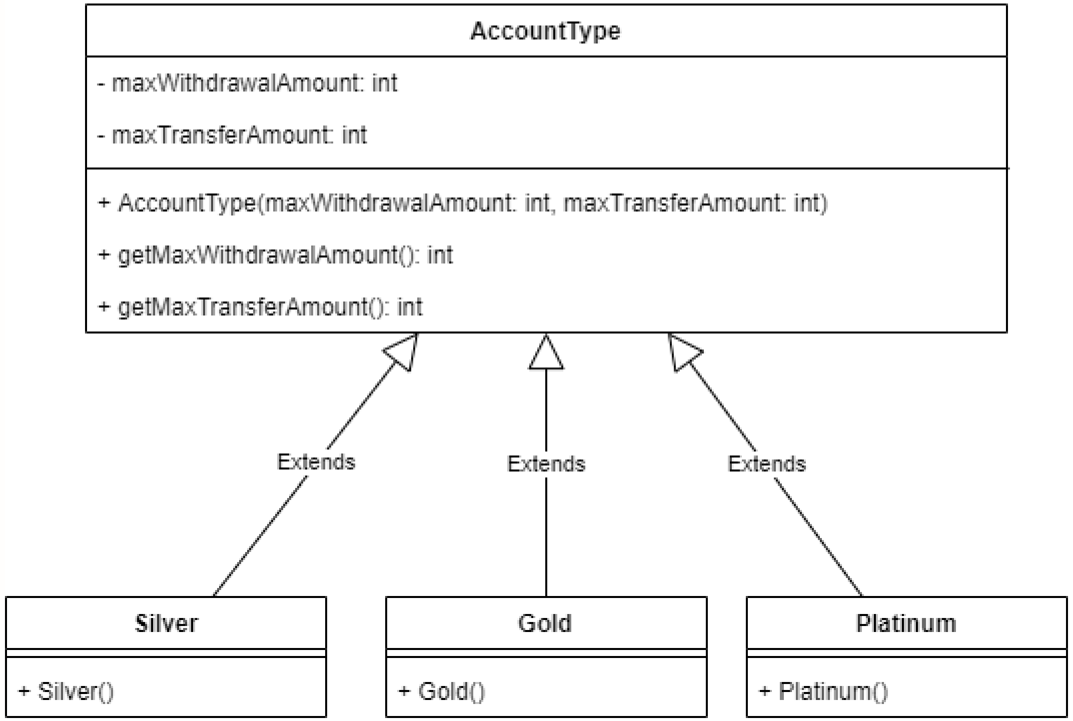
- name: String
- address: Address

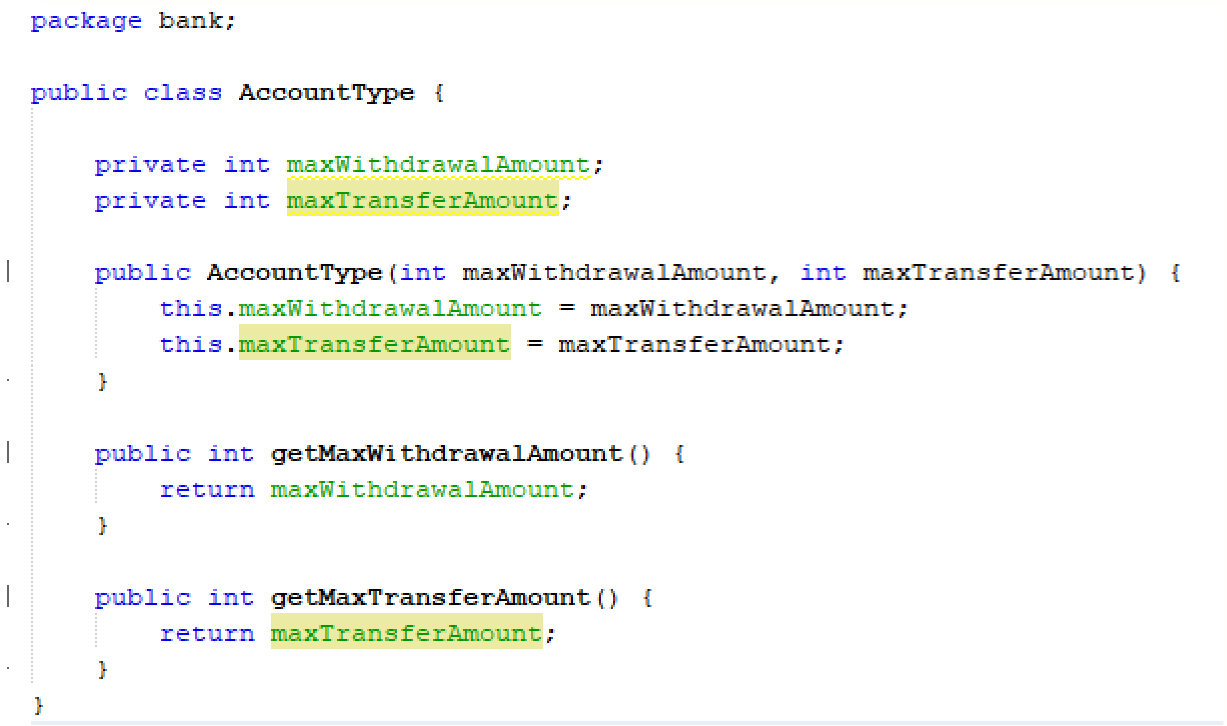
+ Customer(name: String, address, Address)

Address

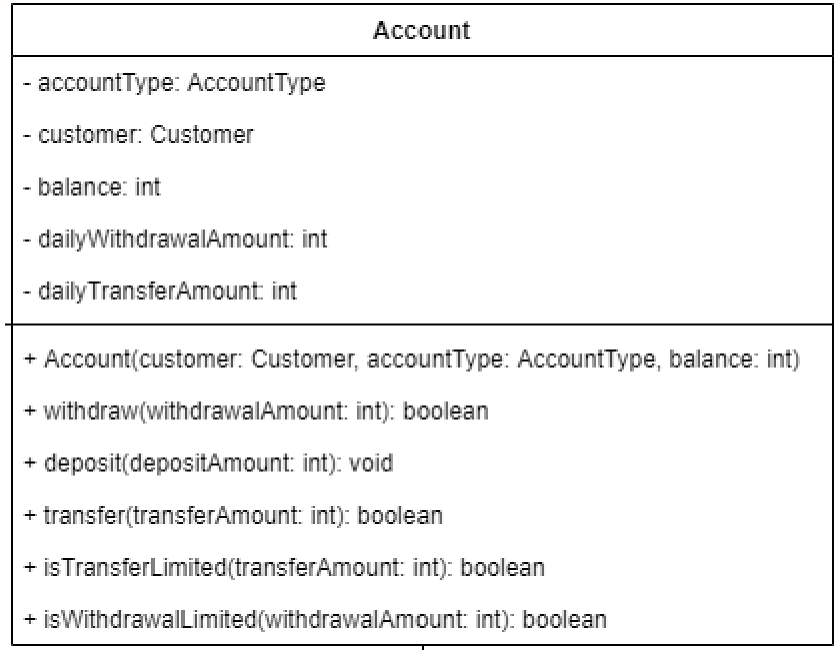
- district: String
- state: String
- city: String

+ Address(district: String, state: String, city: String)


AccountType
- maxWithdrawalAmount: int
- maxTransferAmount: int
+ AccountType(maxWithdrawalAmount: int, maxTransferAmount: int)
+ getMaxWithdrawalAmount(): int
+ getMaxTransferAmount(): int
Extends
Extends
Extends
Silver
+ Silver()
Gold
+ Gold()
Platinum
+ Platinum()



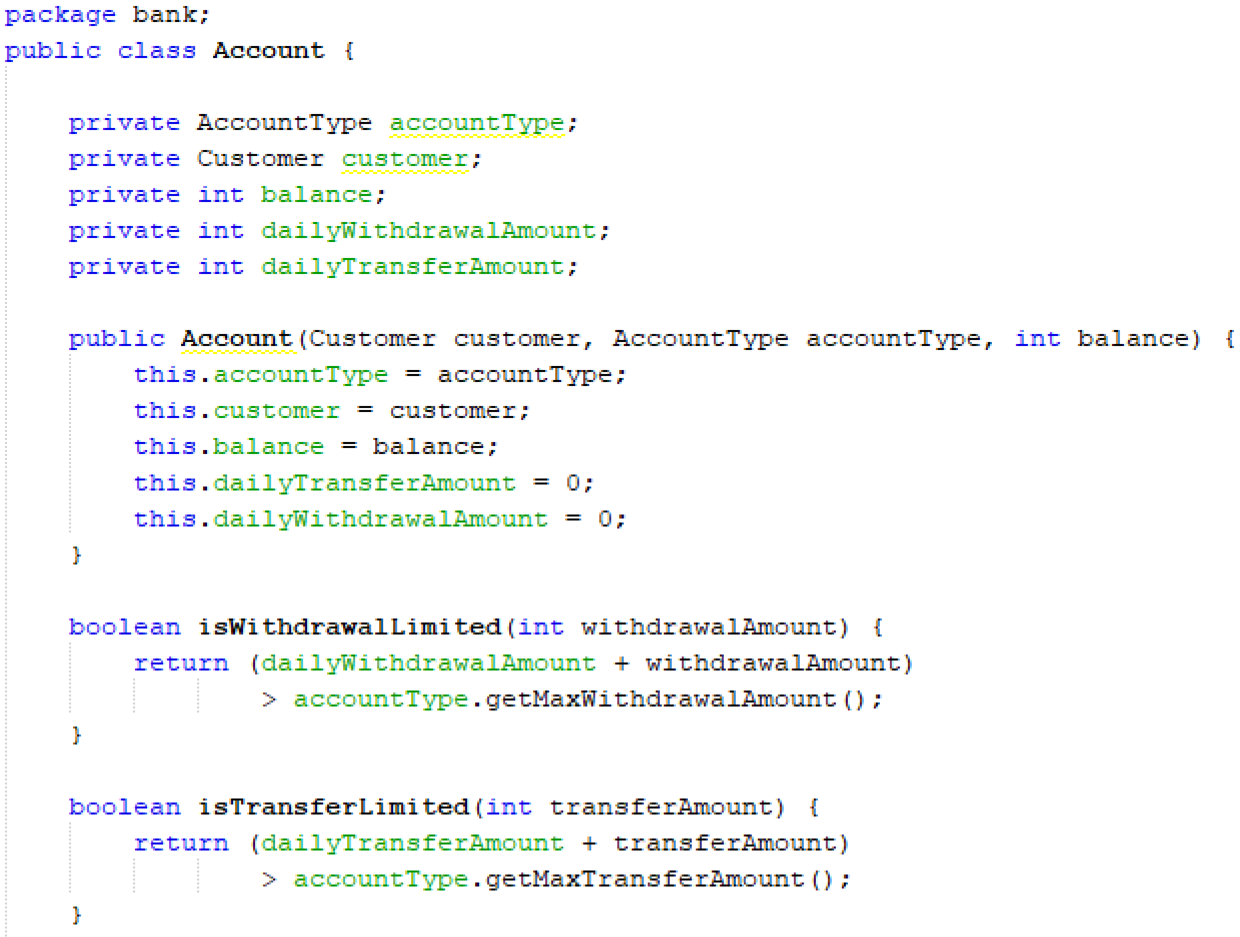
```
package bank;

public class AccountType {

    private int maxWithdrawalAmount;
    private int maxTransferAmount;

    public AccountType(int maxWithdrawalAmount, int maxTransferAmount) {
        this.maxWithdrawalAmount = maxWithdrawalAmount;
        this.maxTransferAmount = maxTransferAmount;
    }

    public int getMaxWithdrawalAmount() {
        return maxWithdrawalAmount;
    }

    public int getMaxTransferAmount() {
        return maxTransferAmount;
    }
}
```



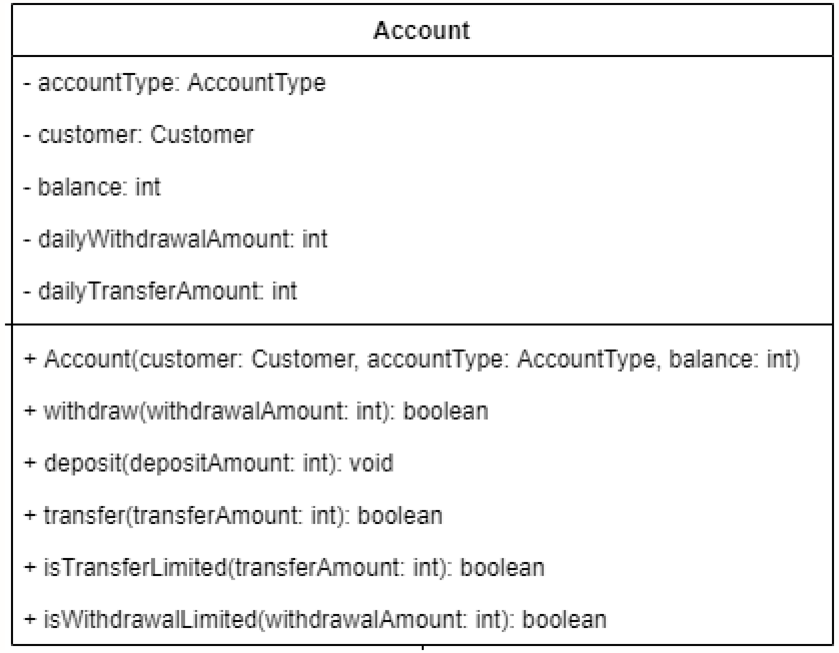
Account

- accountType: AccountType
- customer: Customer
- balance: int
- dailyWithdrawalAmount: int
- dailyTransferAmount: int

+ Account(customer: Customer, accountType: AccountType, balance: int)
+ withdraw(withdrawalAmount: int): boolean
+ deposit(depositAmount: int): void
+ transfer(transferAmount: int): boolean
+ isTransferLimited(transferAmount: int): boolean
+ isWithdrawalLimited(withdrawalAmount: int): boolean



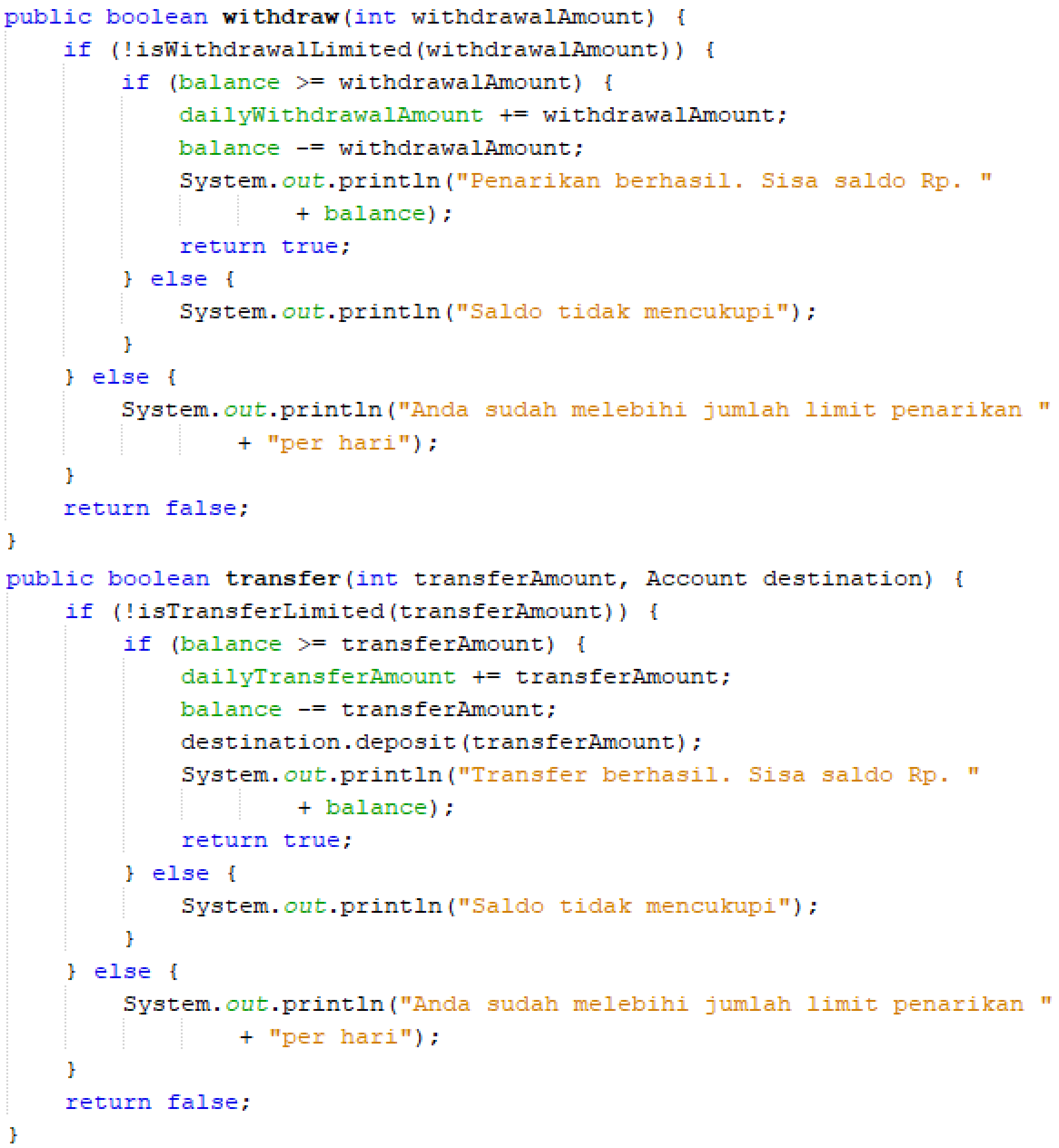
```
package bank;
public class Account {

    private AccountType accountType;
    private Customer customer;
    private int balance;
    private int dailyWithdrawalAmount;
    private int dailyTransferAmount;

    public Account(Customer customer, AccountType accountType, int balance) {
        this.accountType = accountType;
        this.customer = customer;
        this.balance = balance;
        this.dailyTransferAmount = 0;
        this.dailyWithdrawalAmount = 0;
    }

    boolean isWithdrawalLimited(int withdrawalAmount) {
        return (dailyWithdrawalAmount + withdrawalAmount)
                > accountType.getMaxWithdrawalAmount();
    }

    boolean isTransferLimited(int transferAmount) {
        return (dailyTransferAmount + transferAmount)
                > accountType.getMaxTransferAmount();
    }
```



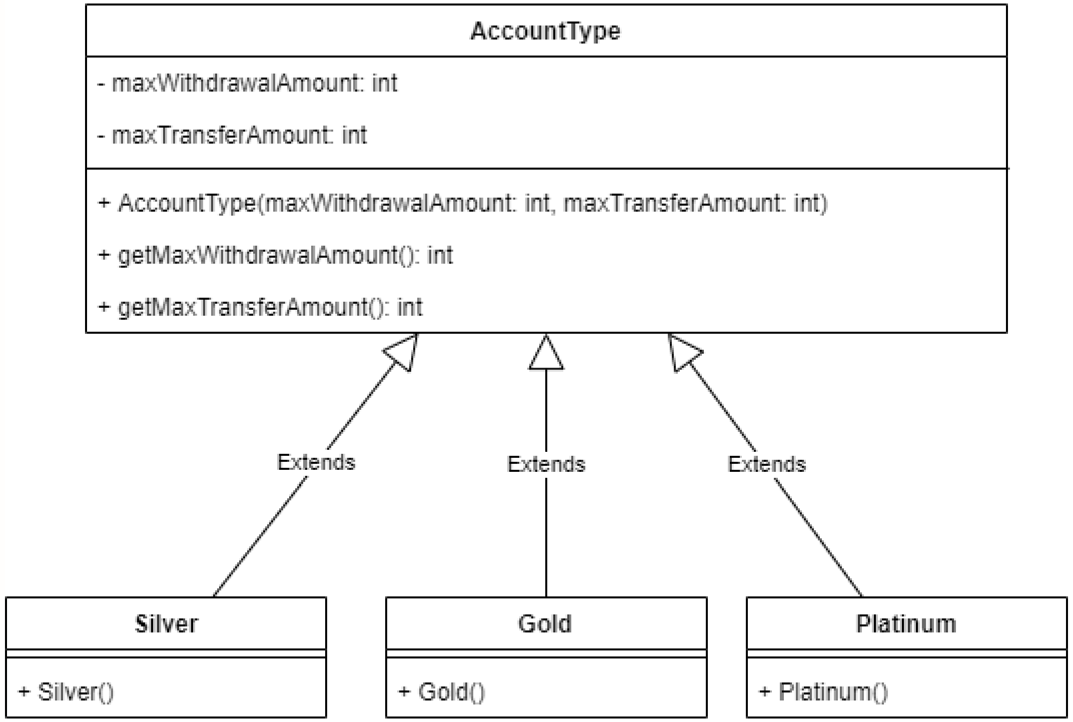
Account

- accountType: AccountType
- customer: Customer
- balance: int
- dailyWithdrawalAmount: int
- dailyTransferAmount: int

+ Account(customer: Customer, accountType: AccountType, balance: int)
+ withdraw(withdrawalAmount: int): boolean
+ deposit(depositAmount: int): void
+ transfer(transferAmount: int): boolean
+ isTransferLimited(transferAmount: int): boolean
+ isWithdrawalLimited(withdrawalAmount: int): boolean



```
public boolean withdraw(int withdrawalAmount) {
    if (!isWithdrawalLimited(withdrawalAmount)) {
        if (balance >= withdrawalAmount) {
            dailyWithdrawalAmount += withdrawalAmount;
            balance -= withdrawalAmount;
            System.out.println("Penarikan berhasil. Sisa saldo Rp. "
                    + balance);
            return true;
        } else {
            System.out.println("Saldo tidak mencukupi");
        }
    } else {
        System.out.println("Anda sudah melebihi jumlah limit penarikan "
                + "per hari");
    }
    return false;
}
public boolean transfer(int transferAmount, Account destination) {
    if (!isTransferLimited(transferAmount)) {
        if (balance >= transferAmount) {
            dailyTransferAmount += transferAmount;
            balance -= transferAmount;
            destination.deposit(transferAmount);
            System.out.println("Transfer berhasil. Sisa saldo Rp. "
                    + balance);
            return true;
        } else {
            System.out.println("Saldo tidak mencukupi");
        }
    } else {
        System.out.println("Anda sudah melebihi jumlah limit penarikan "
                + "per hari");
    }
    return false;
}
```



AccountType

- maxWithdrawalAmount: int
- maxTransferAmount: int

+ AccountType(maxWithdrawalAmount: int, maxTransferAmount: int)
+ getMaxWithdrawalAmount(): int
+ getMaxTransferAmount(): int

Silver

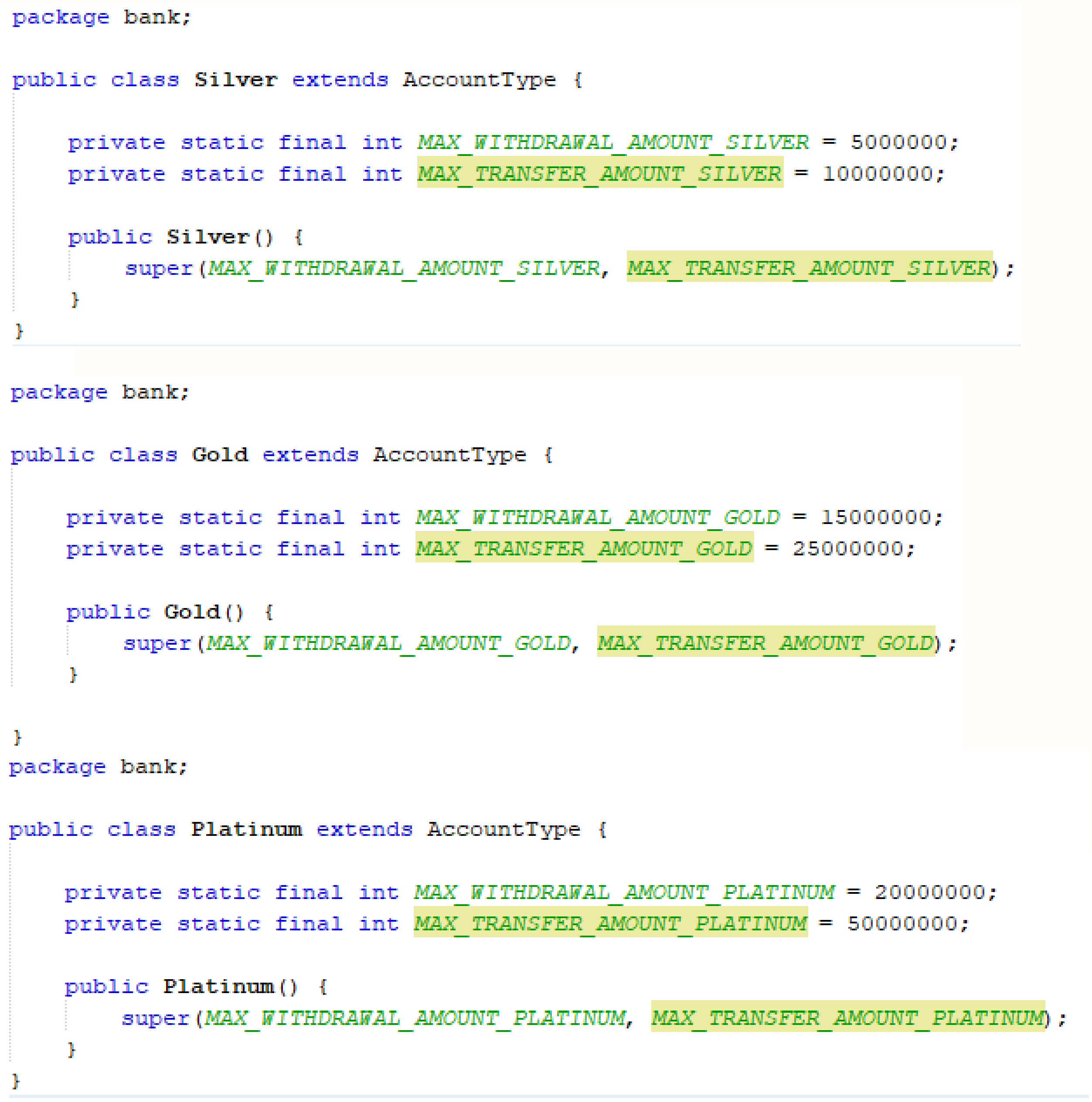
+ Silver()

Gold

+ Gold()

Platinum

+ Platinum()



```
package bank;

public class Silver extends AccountType {

    private static final int MAX_WITHDRAWAL_AMOUNT_SILVER = 5000000;
    private static final int MAX_TRANSFER_AMOUNT_SILVER = 10000000;

    public Silver() {
        super(MAX_WITHDRAWAL_AMOUNT_SILVER, MAX_TRANSFER_AMOUNT_SILVER);
    }
}
```

```
package bank;

public class Gold extends AccountType {

    private static final int MAX_WITHDRAWAL_AMOUNT_GOLD = 15000000;
    private static final int MAX_TRANSFER_AMOUNT_GOLD = 25000000;

    public Gold() {
        super(MAX_WITHDRAWAL_AMOUNT_GOLD, MAX_TRANSFER_AMOUNT_GOLD);
    }

}
```

```
package bank;

public class Platinum extends AccountType {

    private static final int MAX_WITHDRAWAL_AMOUNT_PLATINUM = 20000000;
    private static final int MAX_TRANSFER_AMOUNT_PLATINUM = 50000000;

    public Platinum() {
        super(MAX_WITHDRAWAL_AMOUNT_PLATINUM, MAX_TRANSFER_AMOUNT_PLATINUM);
    }
}
```

End of Review

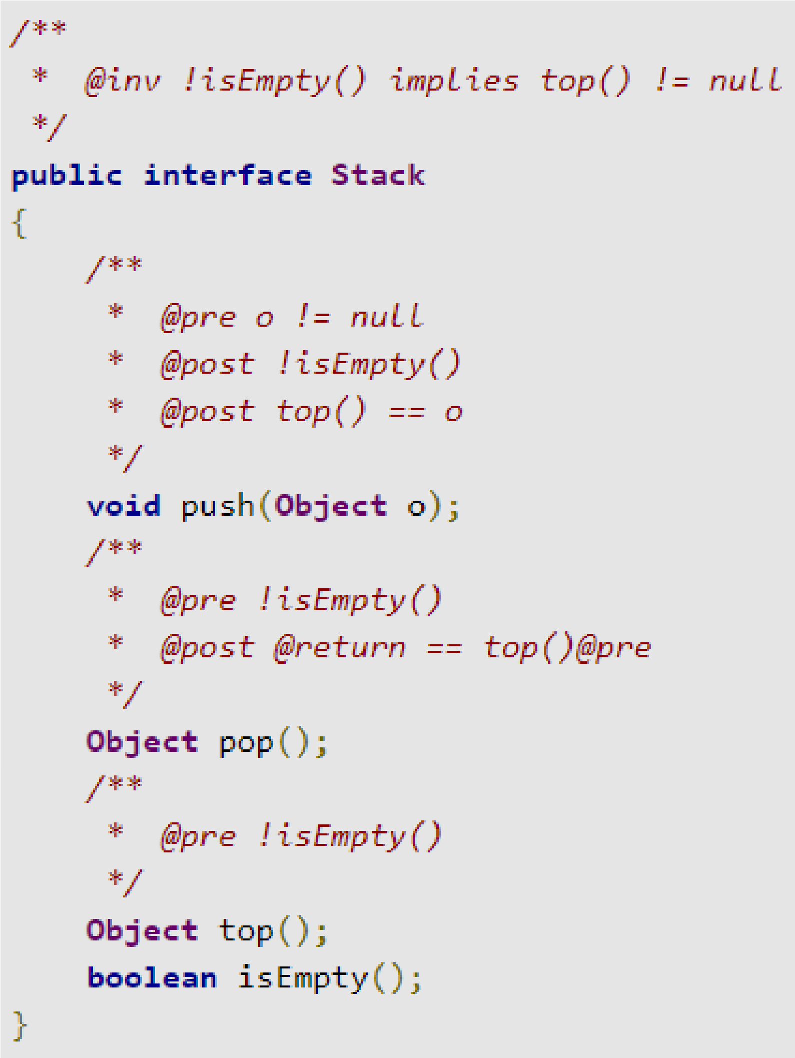
Design by Contract (DbC)

- The goal of DbC is to enable programmers to "build software specification into the software source code and make it self-checking at runtime." This is achieved through the introduction of "contracts" — executable code contained within the source that specifies obligations for classes, methods, and their callers.
- This principle views the relationship between a server and its clients as a formal agreement, expressing each party’s rights and obligations
- Methods should specify their pre- and post-conditions: what must be true before and what must be true after their execution, respectively.
- The server promises to do its job (defined by post-condition) as long as the clients uses the server correctly (defined by pre-condition)

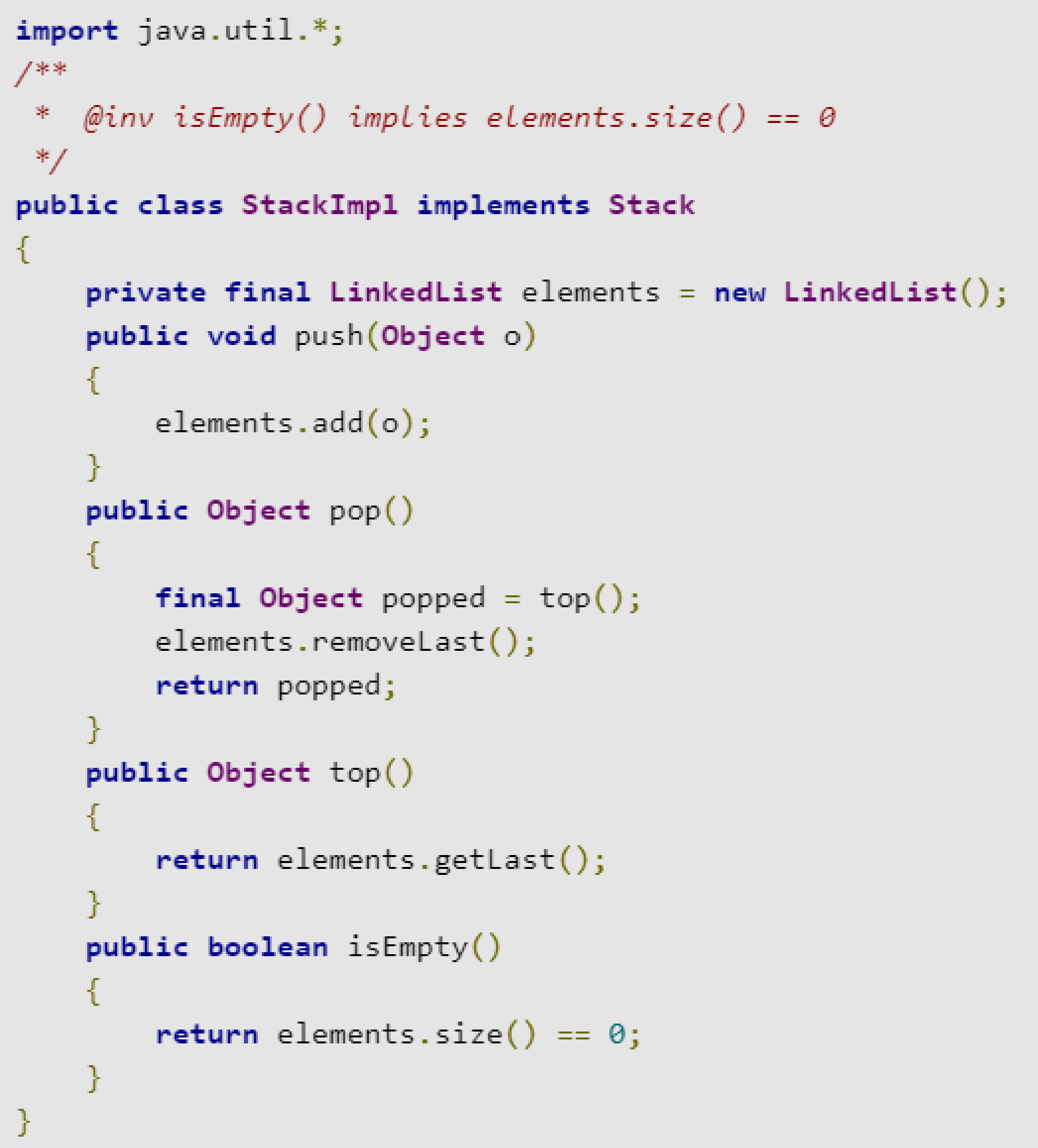
Design by Contract (DbC)

- If a method has specified some pre-condition then the failure of that condition is the responsibility of the client of the method.
- The client should do whatever is necessary to ensure it will meet the pre-conditions.
- Java: iContract, AssertMate, JASS, C4J, Cofoja, Annotated-contracts

Design by Contract: iContract

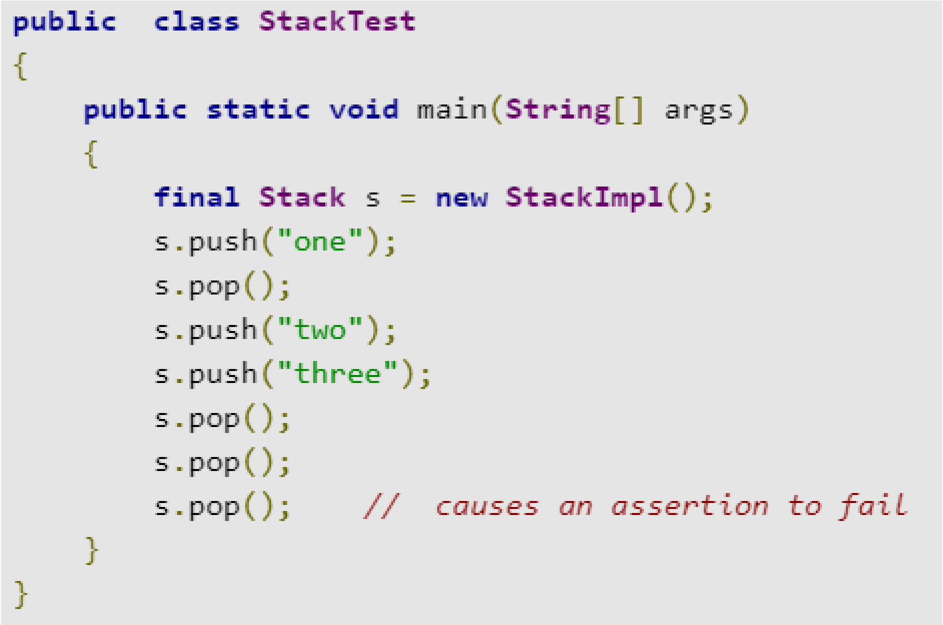


```
/**
 *  @inv !isEmpty() implies top() != null
 */
public interface Stack
{
    /**
     *  @pre o != null
     *  @post !isEmpty()
     *  @post top() == o
     */
    void push(Object o);
    /**
     *  @pre !isEmpty()
     *  @post @return == top()@pre
     */
    Object pop();
    /**
     *  @pre !isEmpty()
     */
    Object top();
    boolean isEmpty();
}
```

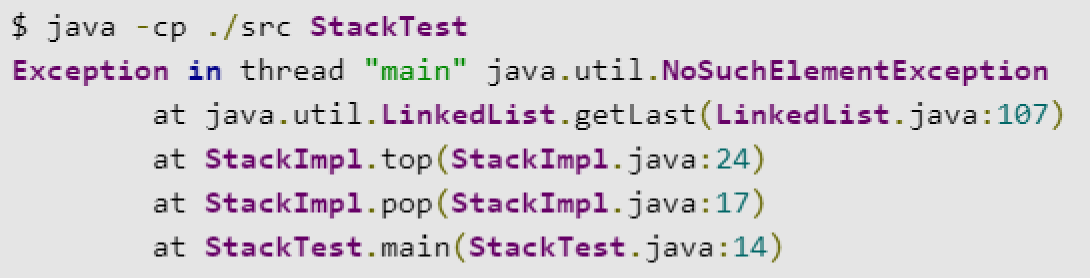


```
import java.util.*;
/**
 *  @inv isEmpty() implies elements.size() == 0
 */
public class StackImpl implements Stack
{
    private final LinkedList elements = new LinkedList();
    public void push(Object o)
    {
        elements.add(o);
    }
    public Object pop()
    {
        final Object popped = top();
        elements.removeLast();
        return popped;
    }
    public Object top()
    {
        return elements.getLast();
    }
    public boolean isEmpty()
    {
        return elements.size() == 0;
    }
}
```

Design by Contract: iContract

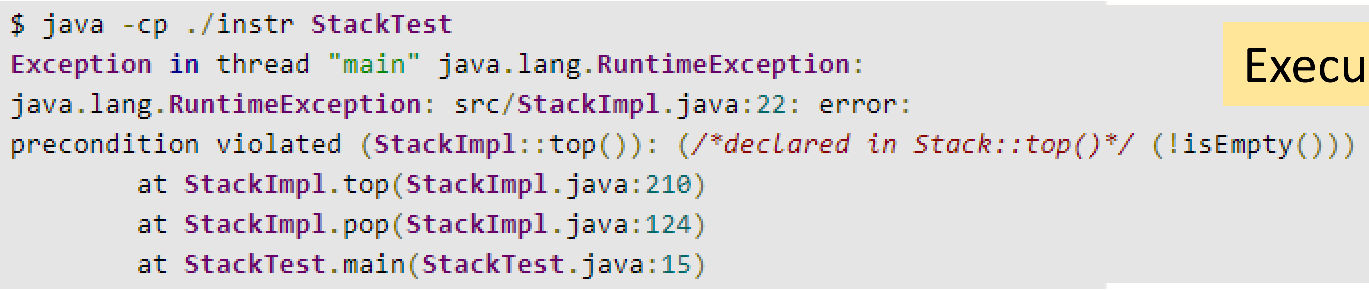


```
public  class StackTest
{
    public static void main(String[] args)
    {
        final Stack s = new StackImpl();
        s.push("one");
        s.pop();
        s.push("two");
        s.push("three");
        s.pop();
        s.pop();
        s.pop();    //  causes an assertion to fail
    }
}
```



```
$ java -cp ./src StackTest
Exception in thread "main" java.util.NoSuchElementException
        at java.util.LinkedList.getLast(LinkedList.java:107)
        at StackImpl.top(StackImpl.java:24)
        at StackImpl.pop(StackImpl.java:17)
        at StackTest.main(StackTest.java:14)
```

Executed without using iContract



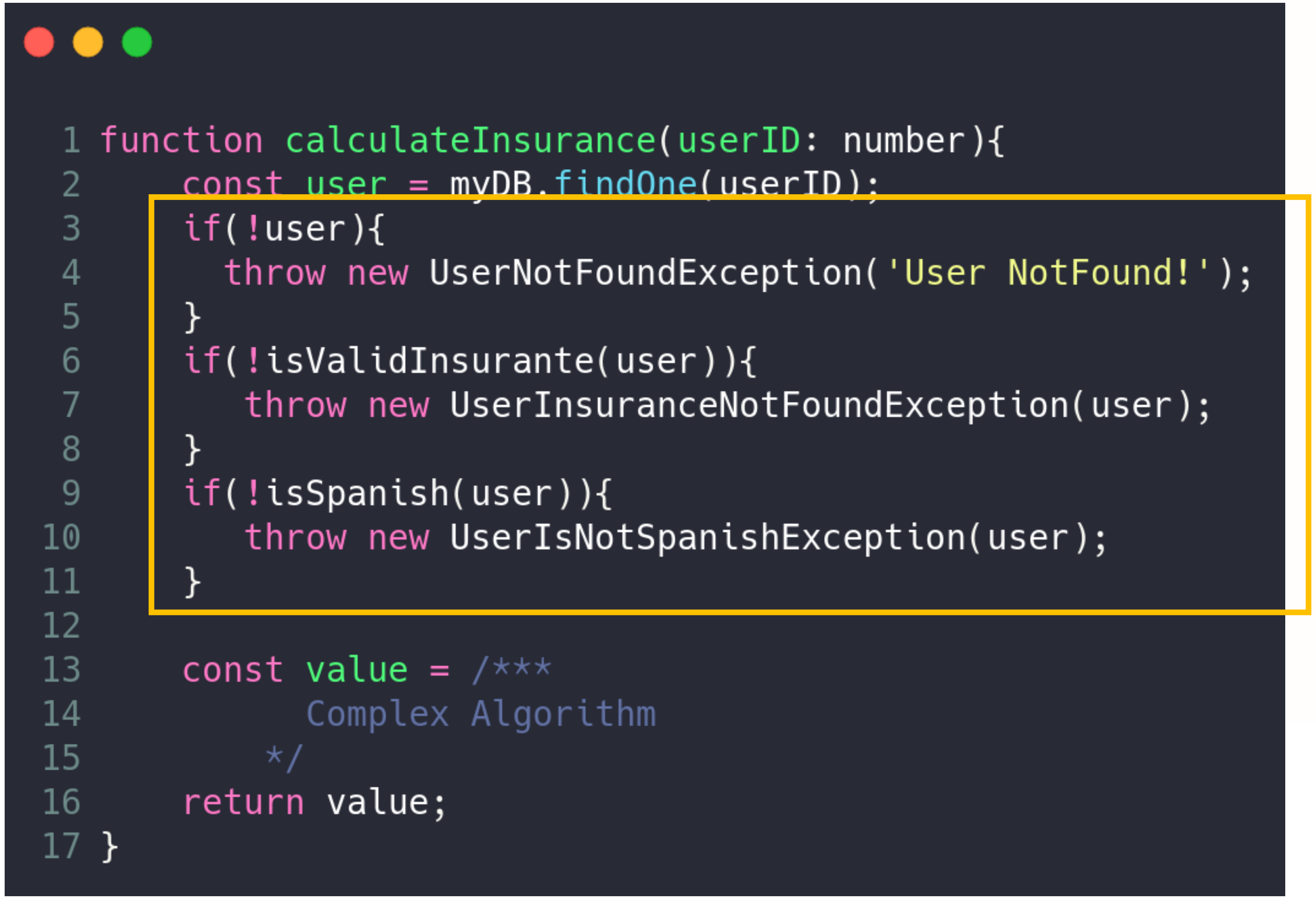
```
$ java -cp ./instr StackTest
Exception in thread "main" java.lang.RuntimeException:
java.lang.RuntimeException: src/StackImpl.java:22: error:
precondition violated (StackImpl::top()): (/*declared in Stack::top()*/ (!isEmpty()))
        at StackImpl.top(StackImpl.java:210)
        at StackImpl.pop(StackImpl.java:124)
        at StackTest.main(StackTest.java:15)
```

Executed using iContract

Defensive Programming

- Defensive Programming is based on the idea that every program module is solely responsible for itself.
- Defensive programming encourages each procedure to defend itself against errors.
- Assume that your program will be called with incorrect inputs, i.e.: files that are supposed to be open may be closed, that files that are supposed to be closed may be open, and so forth.

Defensive Programming



```
function calculateInsurance(userID: number){
    const user = myDB.findOne(userID);
    if(!user){
      throw new UserNotFoundException('User NotFound!');
    }
    if(!isValidInsurante(user)){
       throw new UserInsuranceNotFoundException(user);
    }
    if(!isSpanish(user)){
       throw new UserIsNotSpanishException(user);
    }

    const value = /***
          Complex Algorithm
        */
    return value;
}
```

Checking pre-condition

Common Closure Principle

Common Closure Principle

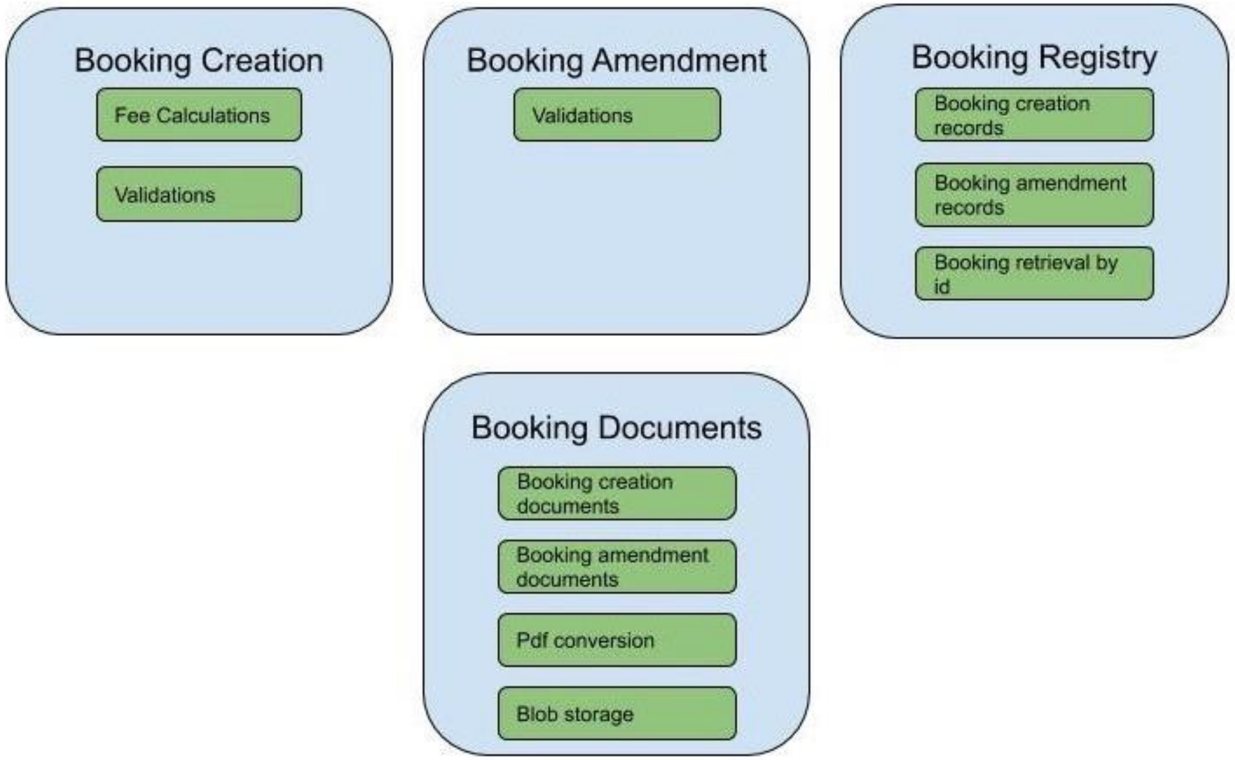
Common Closure Principle

- If the code in an application must change, you would rather that all of the changes occur in one component, rather than being distributed across many components
- If two classes are so tightly bound, that they always change together, then they belong in the same component.
- By following this principle each time we need to change our software the minimum number of components will be affected.

Common Closure Principle

Initial Design

- Booking Creation: Calculates fees, does some validations and checks for room availability.
- Booking Amendment: Checks for availability and does validations. There is no fee for changing a booking, so it does not need fee calculations.

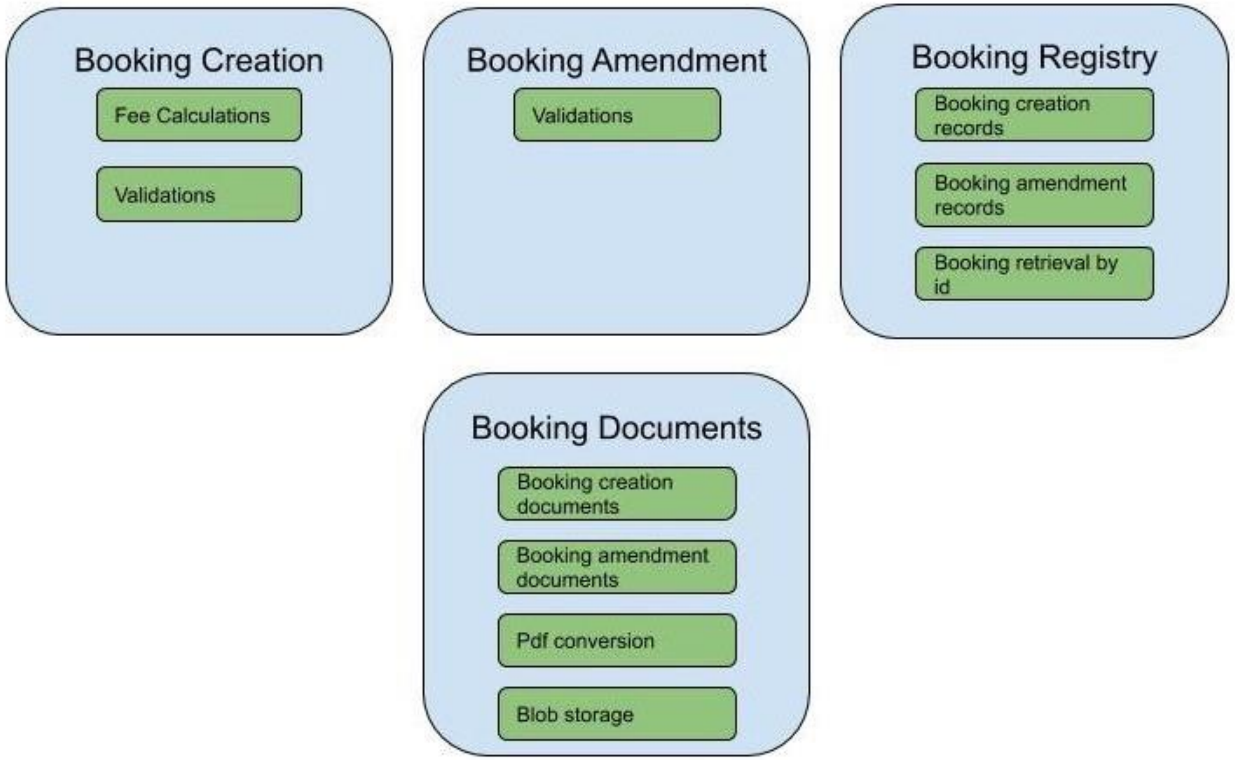


Initial Design

Common Closure Principle

Initial Design

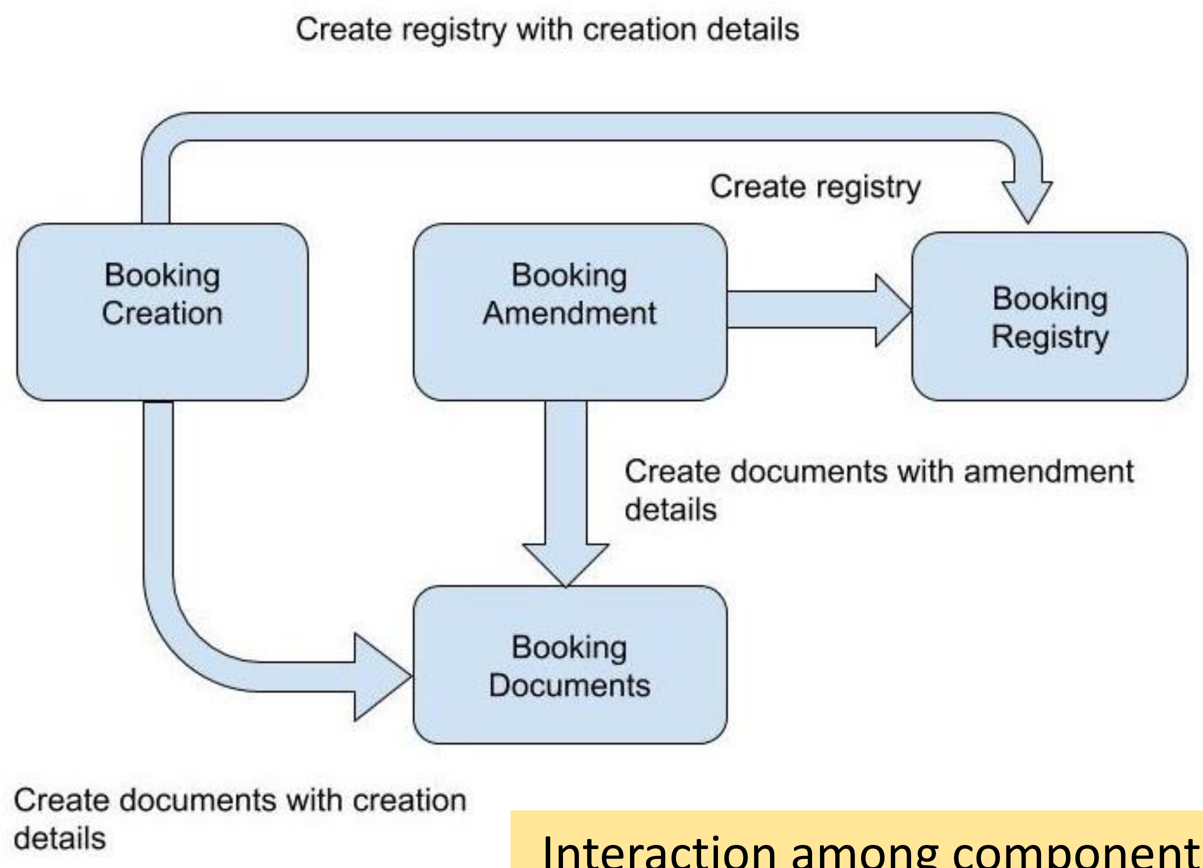
- Booking Documents: Creates booking creation and amendment document, converts to Pdf, as well as saves to blob.
- Booking Registry: Creates registry records required for booking creation and booking amendment. It also has a class which helps with the retrieval of those records;



Initial Design

Common Closure Principle

- Booking creation and Booking Amendment components initiate the call to the other two components to create documents and store the required records.

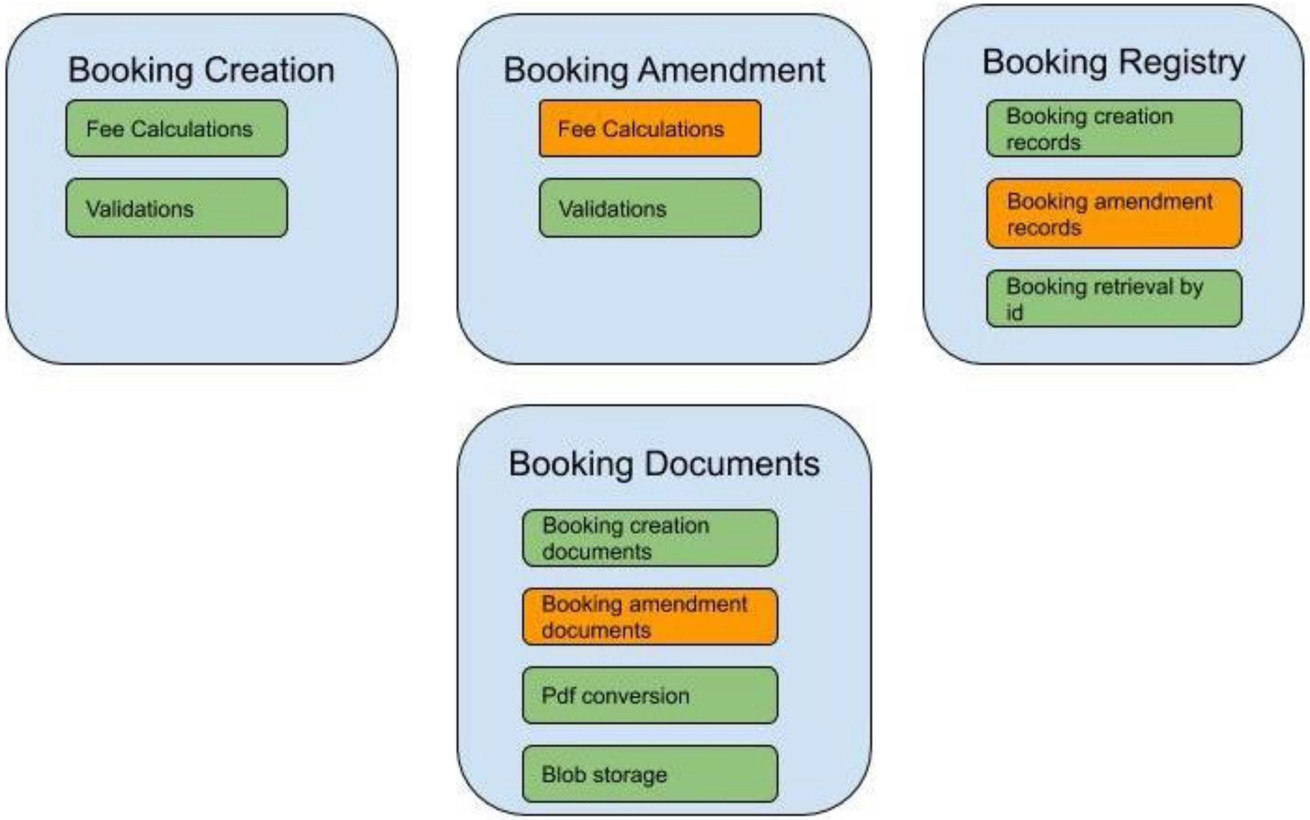

Create registry with creation details
Create registry
Booking Creation
Booking Amendment
Booking Registry
Create documents with amendment details
Booking Documents
Create documents with creation details

Interaction among components

Common Closure Principle

First Scenario:
There is a fee each time a booking amendment happens

- Booking Amendment needs to have the logic which calculates the total fee based on the details of each amendment.
- Booking Amendment Documents must reflect the incurred fee.
- Booking registry needs to store the calculated fee in its records.

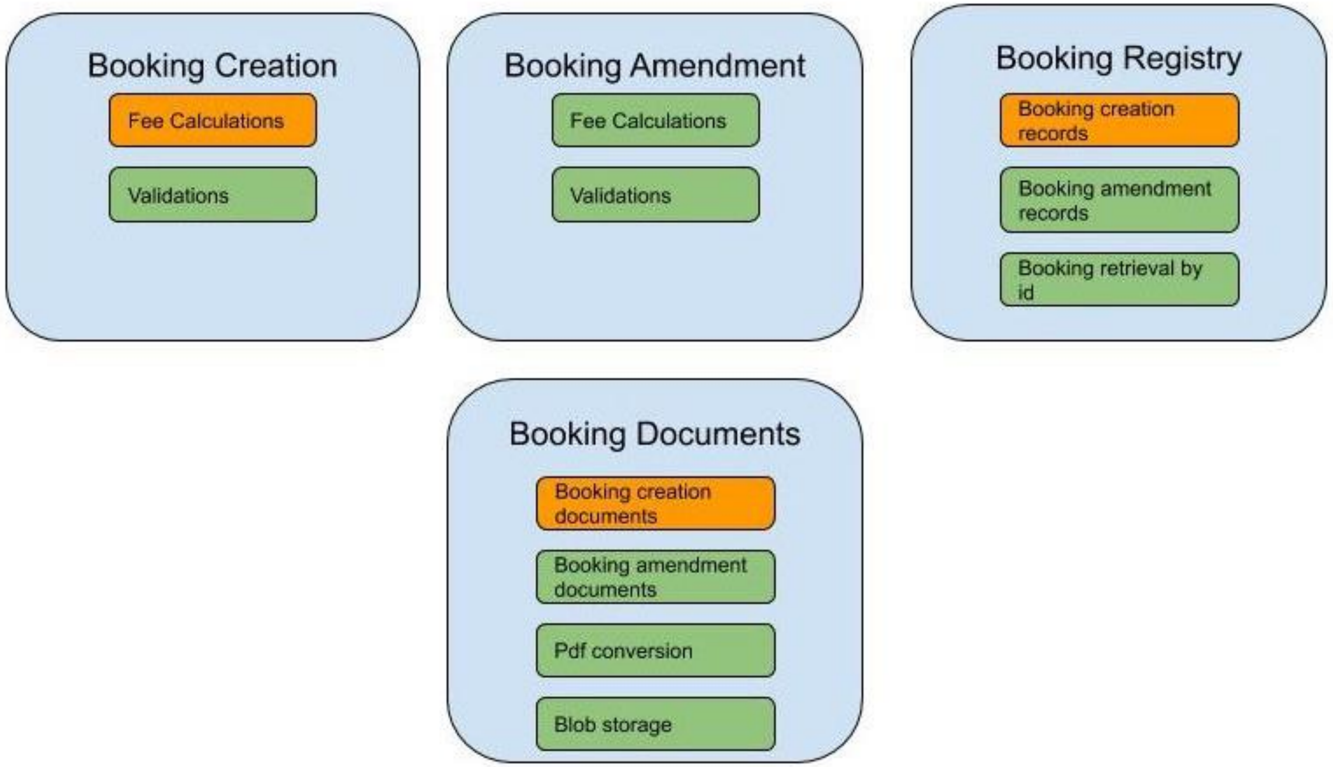


First Scenario

Common Closure Principle

Second Scenario:
There is a promotion code for booking.

- Booking Creation: Needs to be able calculate the discounted fee if there is a promotion code.
- Booking Creation Documents must show the discount when we are creating a booking.
- Booking registry needs to store the promotion code whenever the operation is booking creation and includes a promotion code.

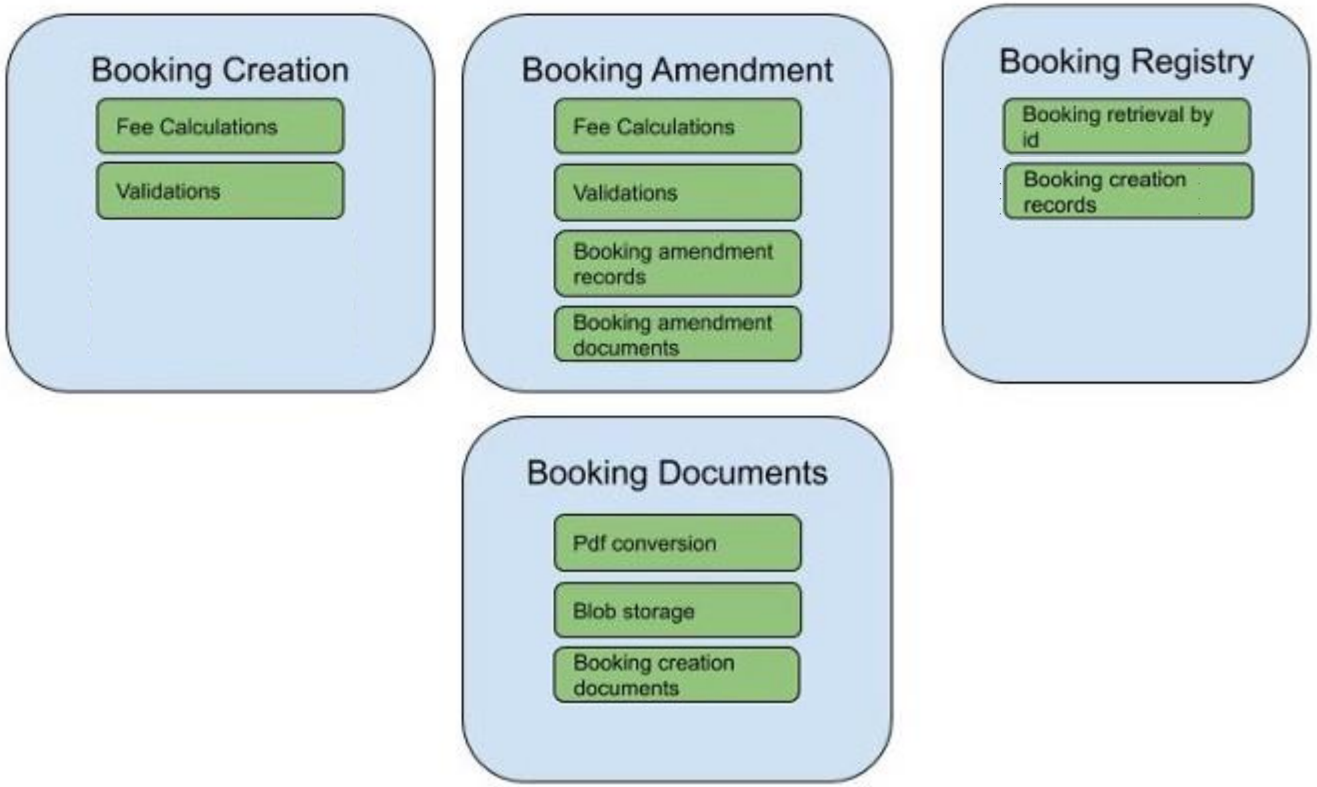


Second Scenario

Common Closure Principle

Revised Design (after applying Common Closure Principle)

- Move the classes which change at the same time and with the same reason to the same component.
- Move the Booking Amendment Document and Booking Amendment Record classes into the Booking Amendment to accommodate first scenario.

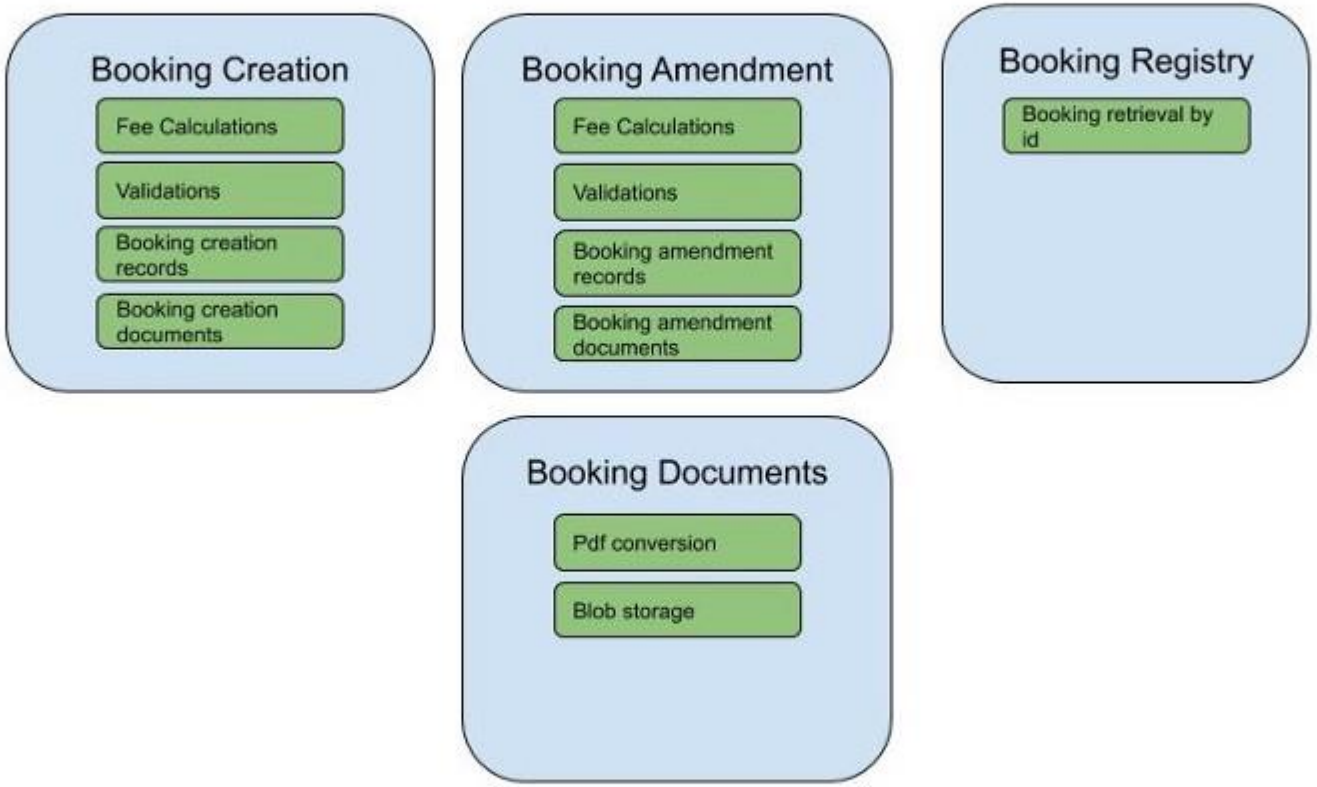


Revised Design

Common Closure Principle

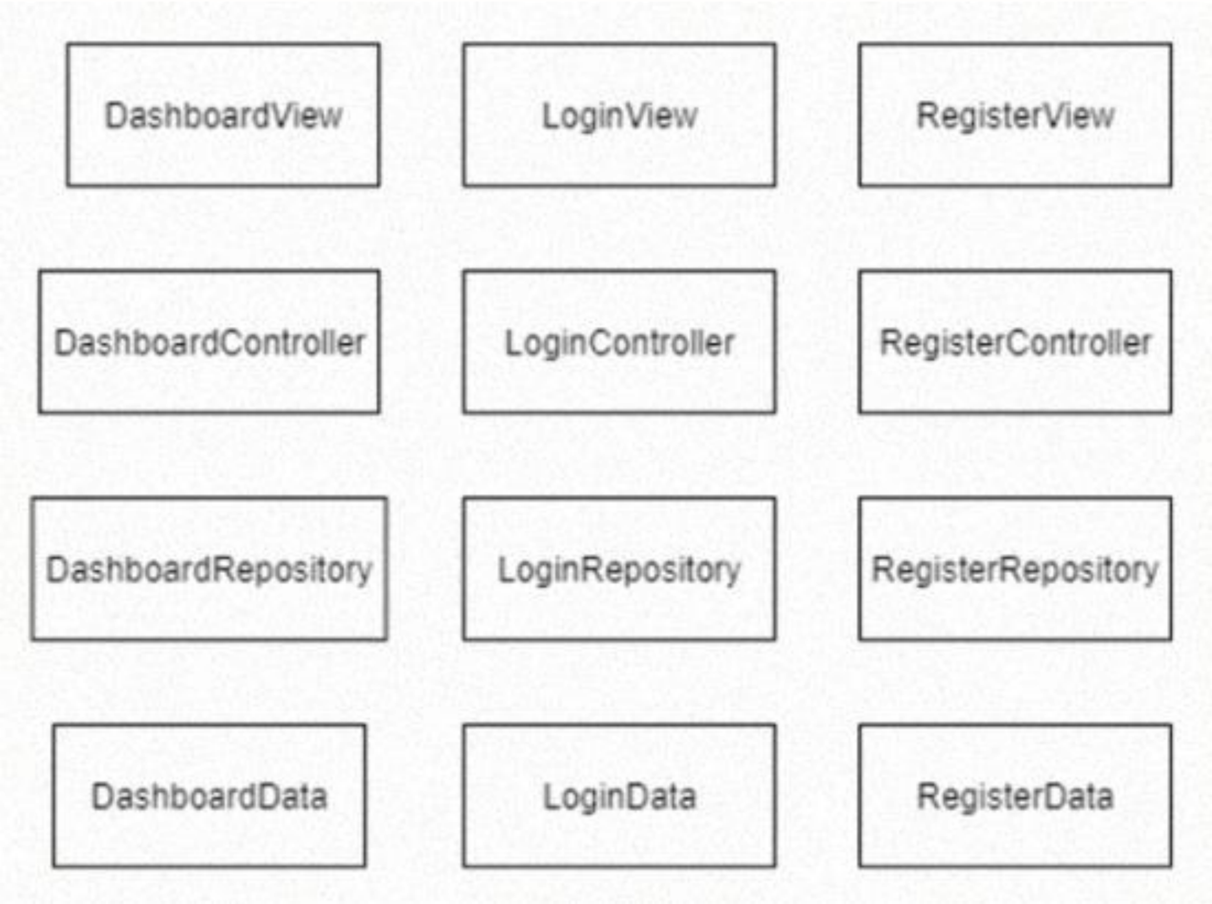
Final Design (after applying Common Closure Principle)

- Move the classes which change at the same time and with the same reason to the same component.
- Move the Booking Creation Document and Booking Creation Record classes into the Booking Creation to accommodate second scenario.



Final Design

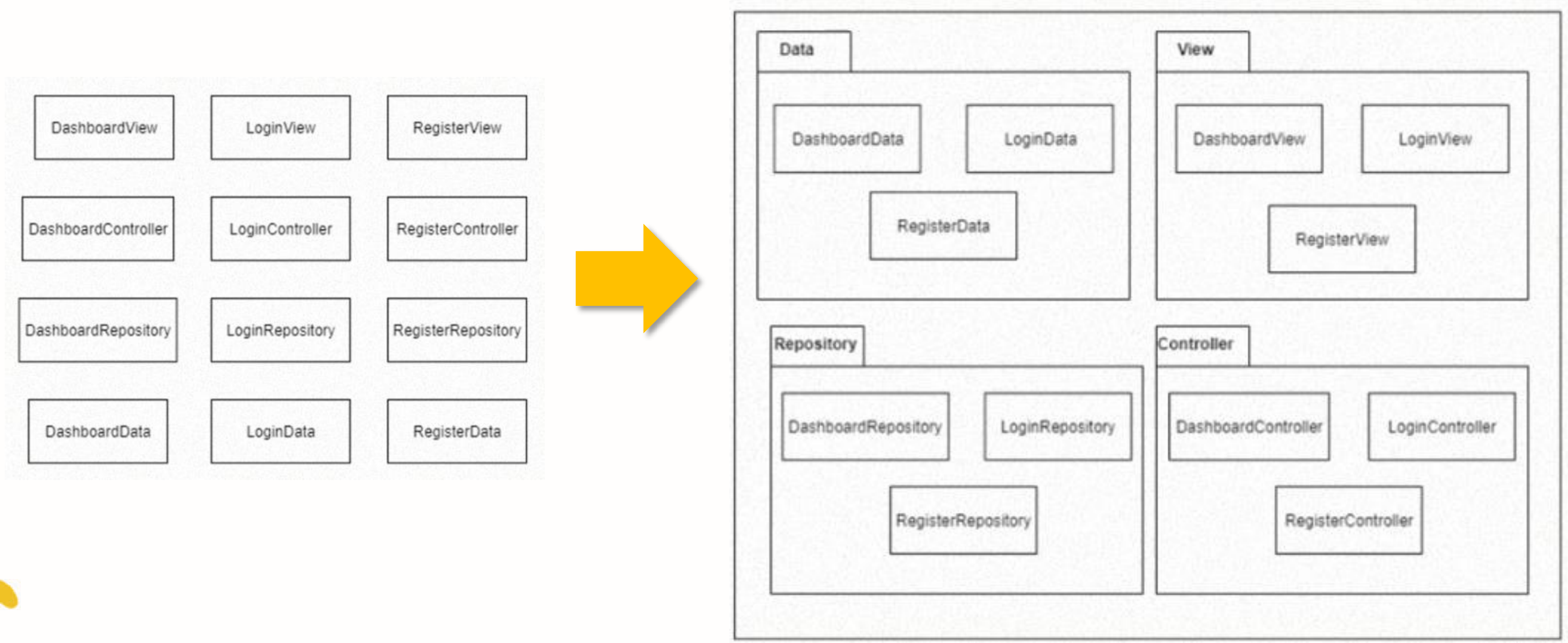
Common Closure Principle



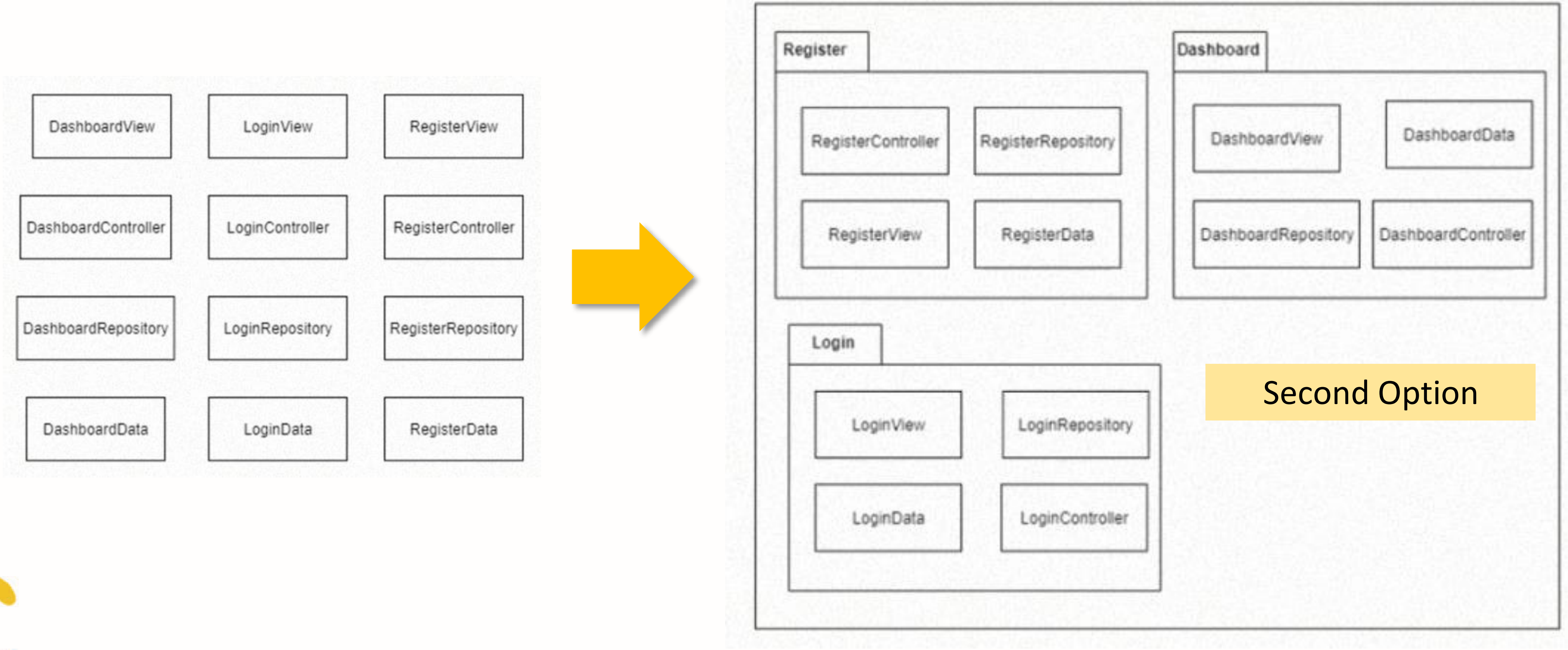
DashboardView	LoginView	RegisterView
DashboardController	LoginController	RegisterController
DashboardRepository	LoginRepository	RegisterRepository
DashboardData	LoginData	RegisterData

Common Closure Principle

First Option



Common Closure Principle



Second Option

References

- Rasyid Institute. Modul Workshop Clean Code. 2019.
- Bertrand Meyer. Object-Oriented Software Construction (2nd Edition). Pearson College Div, 2000.
- Martin, Robert C. Clean Architecture: A Craftsman's Guide to Software Structure and Design. Pearson. 2017.
- https://www.leadingagile.com/2018/05/design-by-contract-part-one/
- https://www.leadingagile.com/2018/05/design-by-contract-part-two/
- https://www.infoworld.com/article/2074956/icontract-design-by-contract-in-java.html?page=2
- https://betterprogramming.pub/refactoring-guard-clauses-2ceeaa1a9da
- https://medium.com/dev-genius/common-closure-principle-the-story-of-an-evolving-architecture-6919b452c8db

bridge to the future

http://www.eepis-its.edu