# S.O.L.I.D.

05506004

# Background

S.O.L.I.D.

- The Single Responsibility Principle
- The Open-Closed Principle
- The Liskov Substitution Principle
- The Interface Segregation Principle
- The Dependency Inversion Principle

**The solid principles are a set of best practices, transformed into a set of rules after dozens of years of cumulative development experience around the world done by software professionals**

https://stackify.com/solid-design-principles/

# Background

S.O.L.I.D.

- The Single Responsibility Principle
- The Open-Closed Principle
- The Liskov Substitution Principle
- The Interface Segregation Principle
- The Dependency Inversion Principle

- The SOLID principles were first introduced by the famous Computer Scientist Robert J. Martin (a.k.a Uncle Bob) in his paper in 2000. But the SOLID acronym was introduced later by Michael Feathers.
  - Uncle Bob is also the author of bestselling books Clean Code and Clean Architecture, and is one of the participants of the "Agile Alliance".
  - Therefore, it is not a surprise that all these concepts of clean coding, object-oriented architecture, and design patterns are somehow connected and complementary to each other.

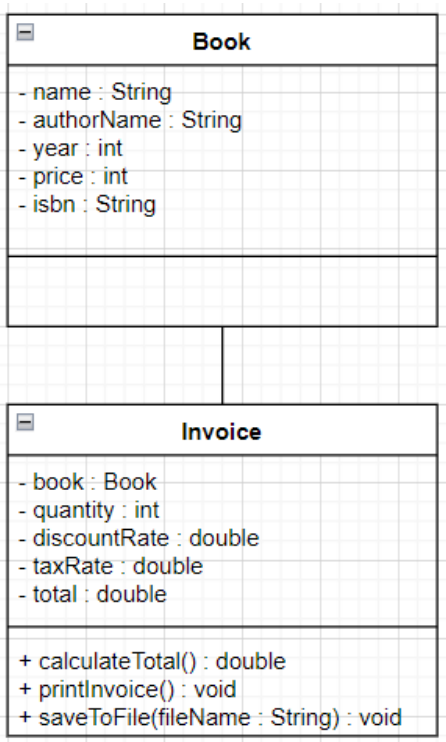
They all serve the same purpose:

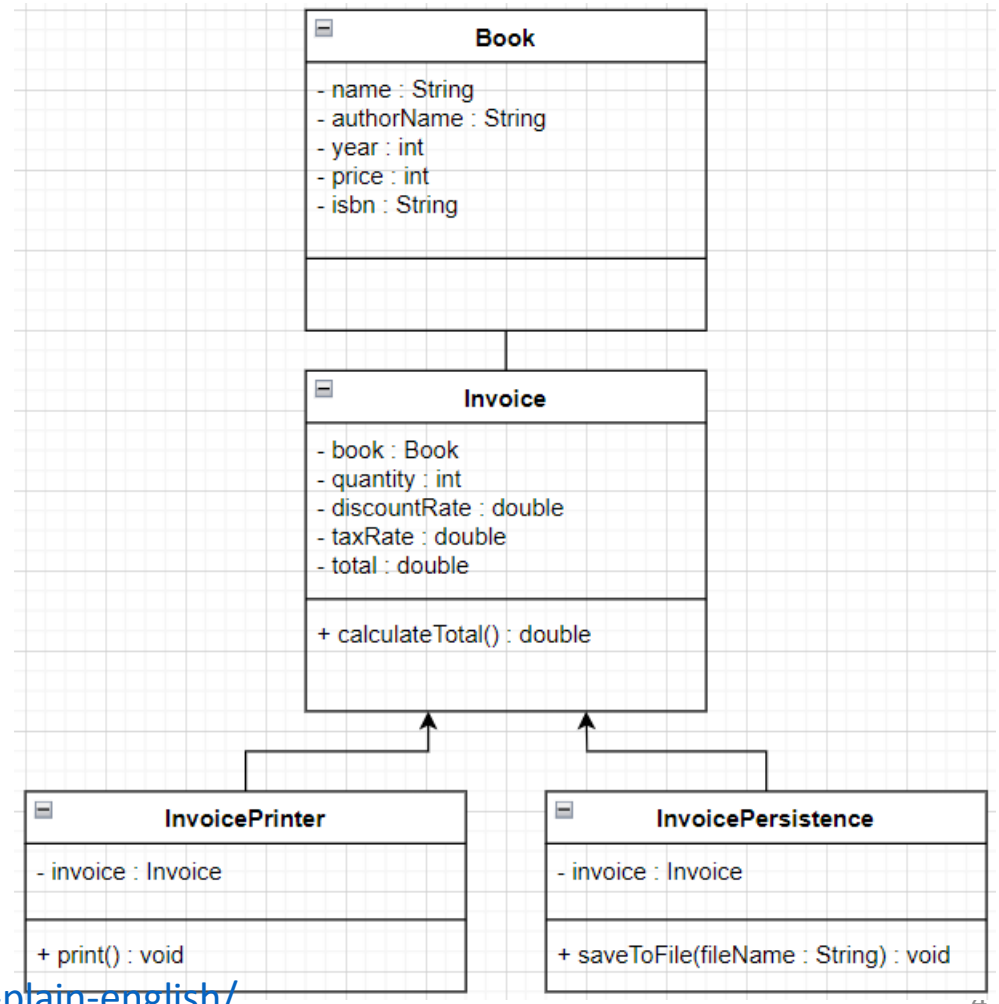
> "To create understandable, readable, and testable code that many developers can collaboratively work on."

# The Single Responsibility Principle

Robert C. Martin describes it:

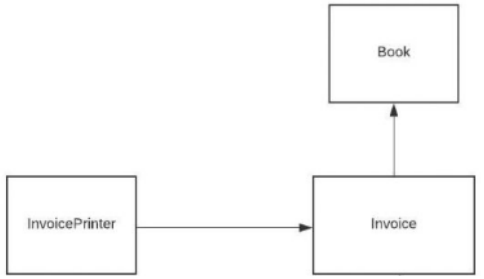
> A class should have one, and only one, reason to change.




https://www.freecodecamp.org/news/solid-principles-explained-in-plain-english/

# The Single Responsibility Principle




- The Single Responsibility Principle states that a class should do one thing and therefore it should have only a single reason to change.
- Following the Single Responsibility Principle is important. First of all, because many different teams can work on the same project and edit the same class for different reasons, this could lead to incompatible modules.
- … https://www.freecodecamp.org/news/solid-principles-explained-in-plain-english/

# The Single Responsibility Principle

```java
class Book {
    String name;
    String authorName;
    int year;
    int price;
    String isbn;

    public Book(String name, String authorName, int year, int price, String isbn) {
        this.name = name;
        this.authorName = authorName;
        this.year = year;
        this.price = price;
        this.isbn = isbn;
    }
}
```

```java
public class Invoice {

    private Book book;
    private int quantity;
    private double discountRate;
    private double taxRate;
    private double total;

    public Invoice(Book book, int quantity, double discountRate, double taxRate) {
        this.book = book;
        this.quantity = quantity;
        this.discountRate = discountRate;
        this.taxRate = taxRate;
        this.total = this.calculateTotal();
    }

    public double calculateTotal() {
            double price = ((book.price - book.price * discountRate) * this.quantity);

        double priceWithTaxes = price * (1 + taxRate);

        return priceWithTaxes;
    }

    public void printInvoice() {
            System.out.println(quantity + "x " + book.name + " " +          book.price
            System.out.println("Discount Rate: " + discountRate);
            System.out.println("Tax Rate: " + taxRate);
            System.out.println("Total: " + total);
    }

        public void saveToFile(String filename) {
    // Creates a file with given name and writes the invoice
    }

}
```

```java
public class InvoicePrinter {
    private Invoice invoice;

    public InvoicePrinter(Invoice invoice) {
        this.invoice = invoice;
    }

    public void print() {
        System.out.println(invoice.quantity + "x " + invoice.book.name + " " + invoice.boo
        System.out.println("Discount Rate: " + invoice.discountRate);
        System.out.println("Tax Rate: " + invoice.taxRate);
        System.out.println("Total: " + invoice.total + " $");
    }
}
```

```java
public class InvoicePersistence {
    Invoice invoice;

    public InvoicePersistence(Invoice invoice) {
        this.invoice = invoice;
    }

    public void saveToFile(String filename) {
        // Creates a file with given name and writes the invoice
    }
}
```

# Open-Closed Principle

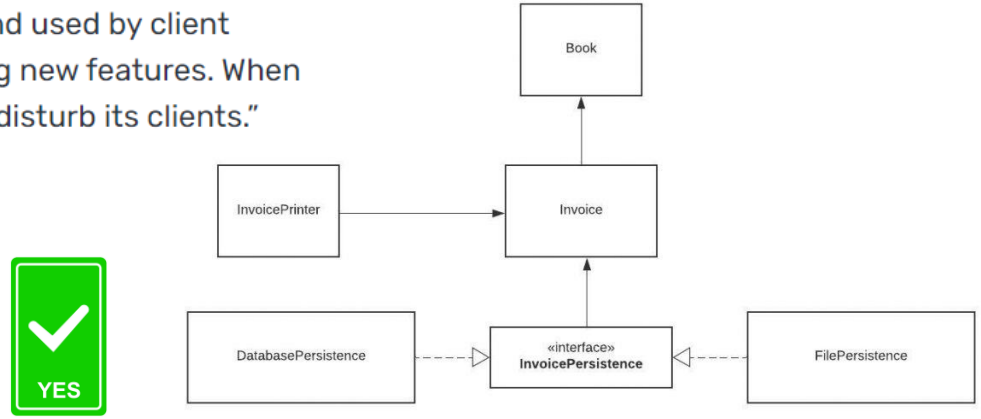
Robert C. Martin considered this principle as the "the most important principle of object-oriented design". But he wasn't the first one who defined it. Bertrand Meyer wrote about it in 1988 in his book Object-Oriented Software Construction. He explained the Open/Closed Principle as:

> "Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification."

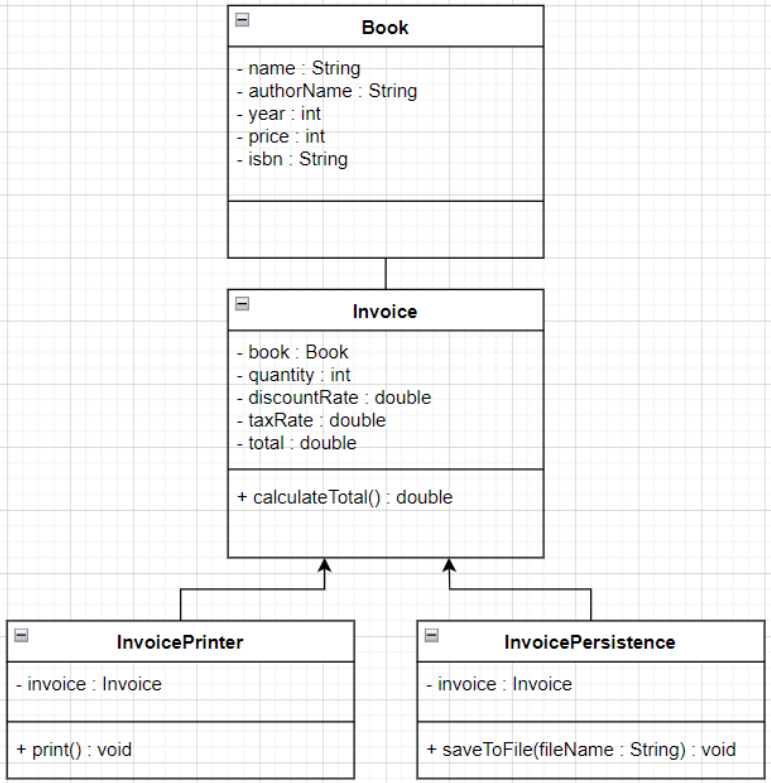
The general idea of this principle is great. It tells you to write your code so that you will be able to add new functionality without changing the existing code. That prevents situations in which a change to one of your classes also requires you to adapt all depending classes. Unfortunately, Bertrand Mayer proposes to use inheritance to achieve this goal:

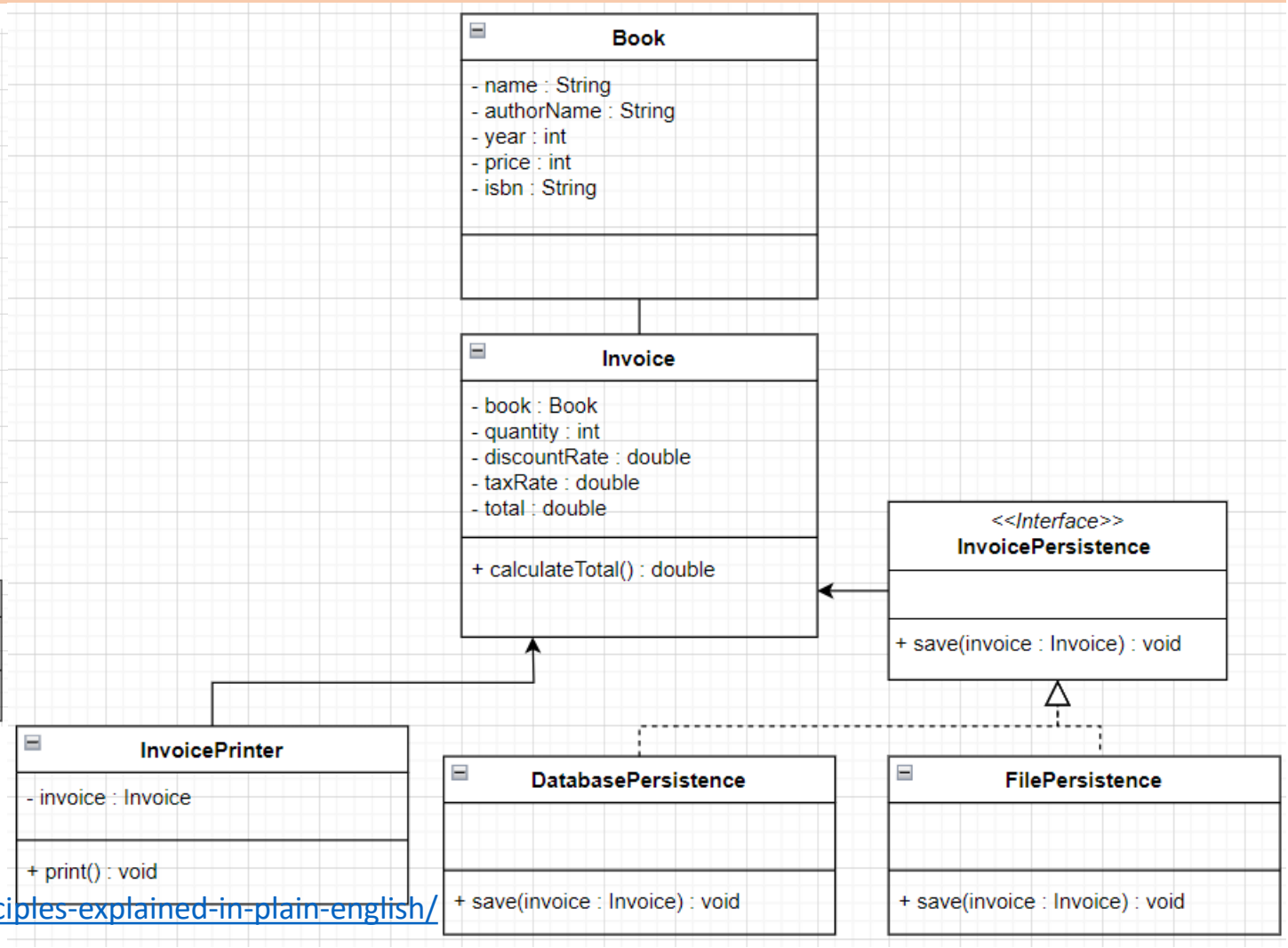
> "A class is closed, since it may be compiled, stored in a library, baselined, and used by client classes. But it is also open, since any new class may use it as parent, adding new features. When a descendant class is defined, there is no need to change the original or to disturb its clients."

```java
public class InvoicePersistence {
    Invoice invoice;

    public InvoicePersistence(Invoice invoice) {
        this.invoice = invoice;
    }

    public void saveToFile(String filename) {
        // Creates a file with given name and writes the invoice
    }

    public void saveToDatabase() {
        // Saves the invoice to database
    }
}
```

# Open-Closed Principle




https://www.freecodecamp.org/news/solid-principles-explained-in-plain-english/

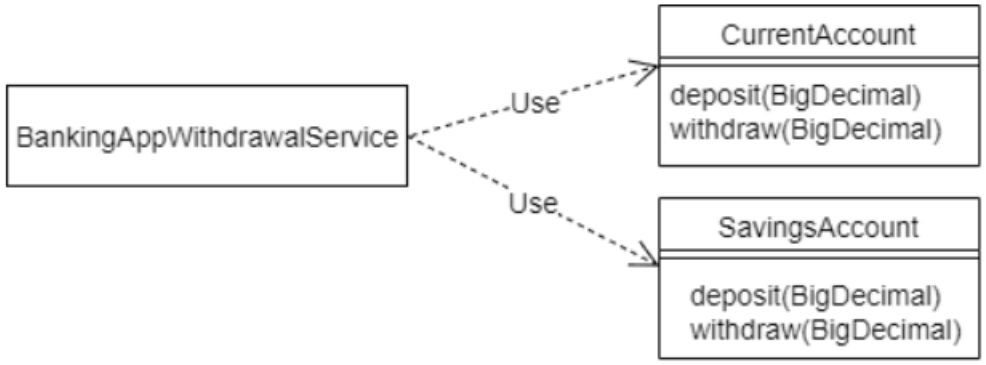
# Open-Closed Principle

- The Open-Closed Principle requires that classes should be open for extension and closed to modification.
- So what this principle wants to say is: We should be able to add new functionality without touching the existing code for the class. This is because whenever we modify the existing code, we are taking the risk of creating potential bugs. So we should avoid touching the tested and reliable (mostly) production code if possible.

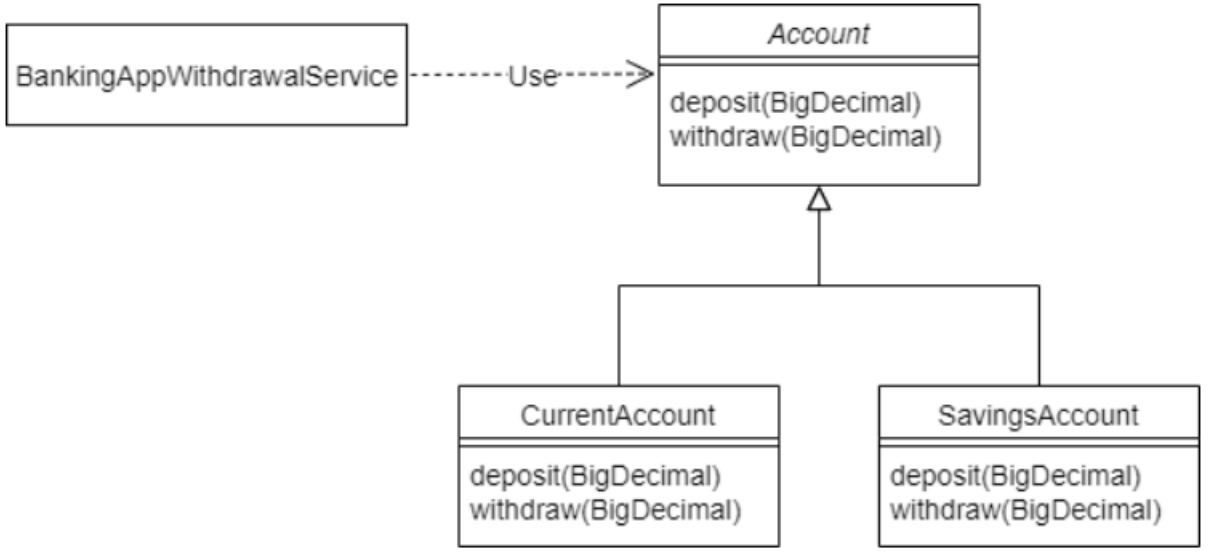
# Liskov Substitution Principle

## The Liskov Substitution Principle in practical software development

The principle defines that objects of a superclass shall be replaceable with objects of its subclasses without breaking the application. That requires the objects of your subclasses to behave in the same way as the objects of your superclass. You can achieve that by following a few rules, which are pretty similar to the design by contract concept defined by Bertrand Meyer. https://stackify.com/solid-design-liskov-substitution-principle/




**Without the Open/Closed Principle**



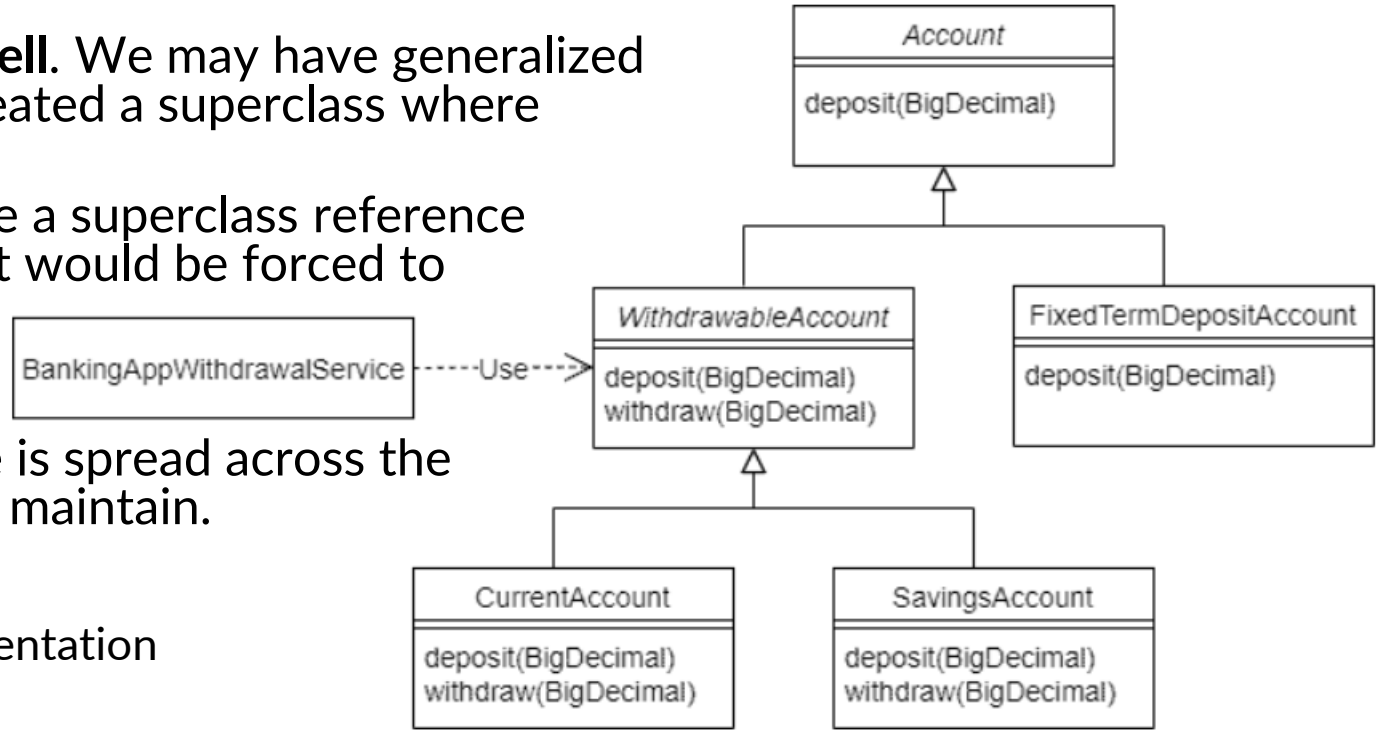

**the Open/Closed Principle to Make the Code Extensible**

https://www.baeldung.com/java-liskov-substitution-principle

# Liskov Substitution Principle

- At a high level, the LSP states that in an object-oriented program, if we substitute a superclass object reference with an object of any of its subclasses, the program should not break.
  - LSP violations are a design **smell**. We may have generalized a concept prematurely and created a superclass where none is needed.
  - If client code cannot substitute a superclass reference with a subclass object freely, it would be forced to do instanceof checks and specially handle some subclasses.
    If this kind of conditional code is spread across the codebase, it will be difficult to maintain.
- Fixing the Design
  - Program to interface, not implementation
  - Encapsulate what varies
  - Prefer composition over inheritance

https://reflectoring.io/lsp-explained/




**Revised Class Diagram**

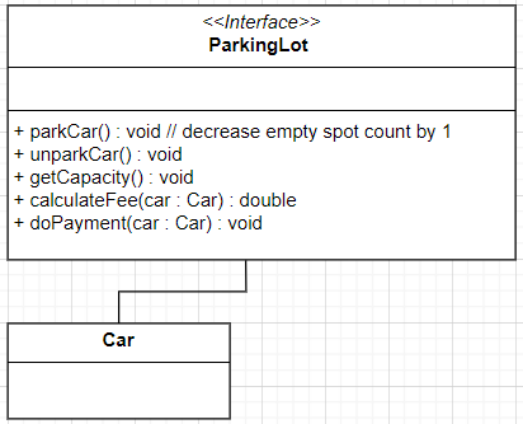
https://www.baeldung.com/java-liskov-substitution-principle

# Interface Segregation Principle

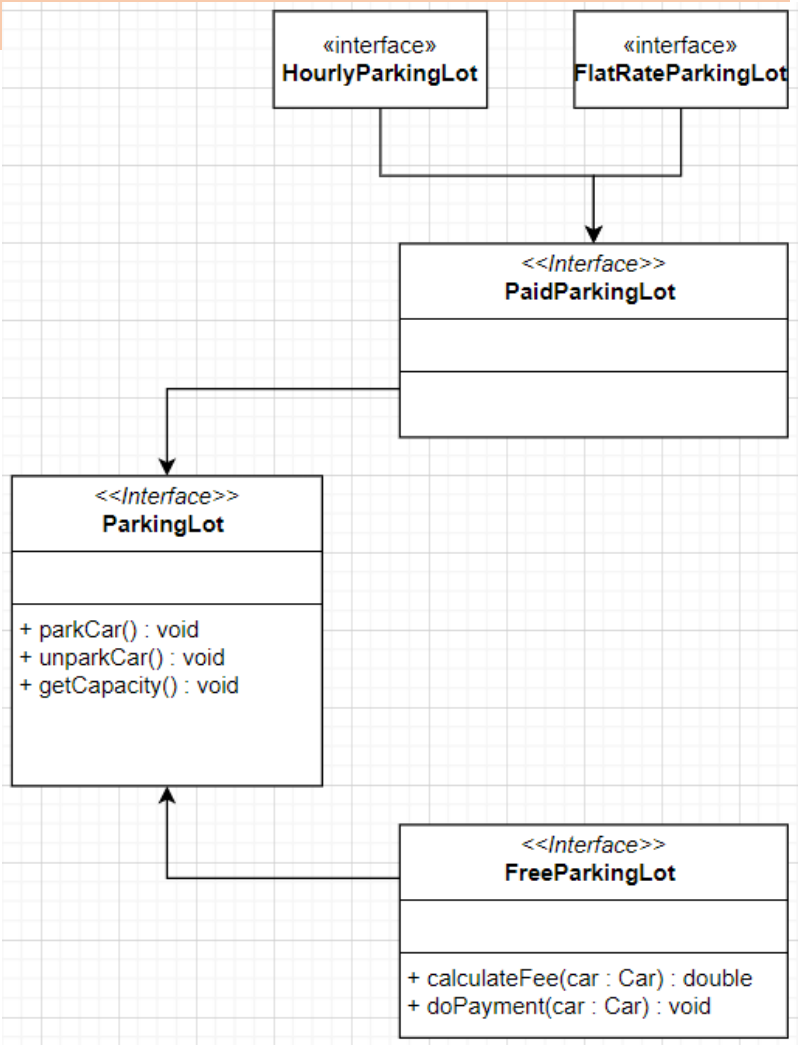
## Interface Segregation Principle

Segregation means keeping things separated, and the Interface Segregation Principle is about separating the interfaces.

The principle states that many client-specific interfaces are better than one general-purpose interface. Clients should not be forced to implement a function they do no need.





# Dependency Inversion Principle

The Dependency Inversion principle states that our classes should depend upon interfaces or abstract classes instead of concrete classes and functions.

In his article (2000), Uncle Bob summarizes this principle as follows:

> "If the OCP states the goal of OO architecture, the DIP states the primary mechanism".

https://www.freecodecamp.org/news/solid-principles-expla

https://stackify.com/dependency-inversion-principle/

**High-level modules should not depend on low-level modules**

## Definition of the Dependency Inversion Principle

The general idea of this principle is as simple as it is important: High-level modules, which provide complex logic, should be easily reusable and unaffected by changes in low-level modules, which provide utility features. To achieve that, you need to introduce an abstraction that decouples the high-level and low-level modules from each other.

Based on this idea, Robert C. Martin's definition of the Dependency Inversion Principle consists of two parts:

1. High-level modules should not depend on low-level modules. Both should depend on abstractions.
2. Abstractions should not depend on details. Details should depend on abstractions.

An important detail of this definition is, that high-level **and** low-level modules depend on the abstraction. The design principle does not just change the direction of the dependency, as you might have expected when you read its name for the first time. It splits the dependency between the high-level and low-level modules by introducing an abstraction between them. So in the end, you get two dependencies:

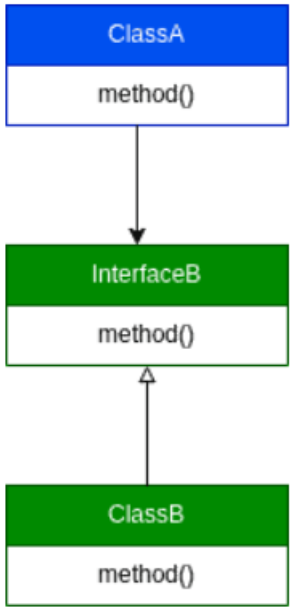
1. the high-level module depends on the abstraction, and
2. the low-level depends on the same abstraction.

# Dependency Inversion Principle

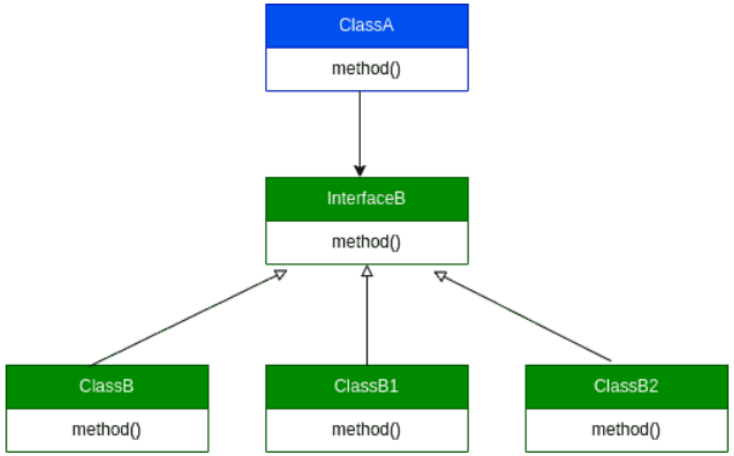
**Dependency Inversion is the strategy of depending upon interfaces or abstract functions and classes rather than upon concrete functions and classes.**

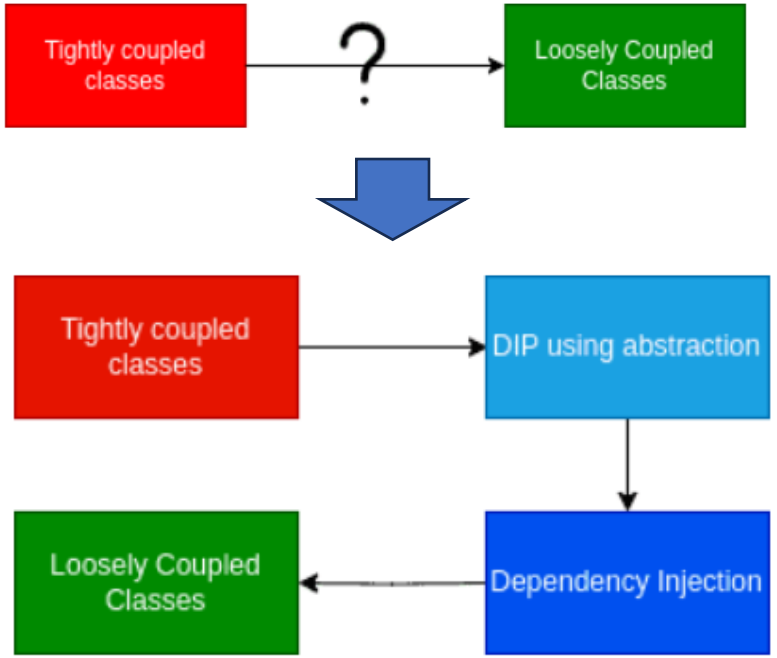
Simply put, when components of our system have dependencies, we don't want directly inject a component's dependency into another. Instead, we should use a level of abstraction between them. https://www.baeldung.com/cs/dip


```
class ClassB {
    // fields, constructor and methods
}

class ClassA {
    ClassB objectB;

    ClassA(ClassB objectB) {
        this.objectB = objectB;
    }
    // invoke classB methods
}
```



# Dependency Inversion Principle




Let's summarize how DIP can help us to get loosely coupled classes:

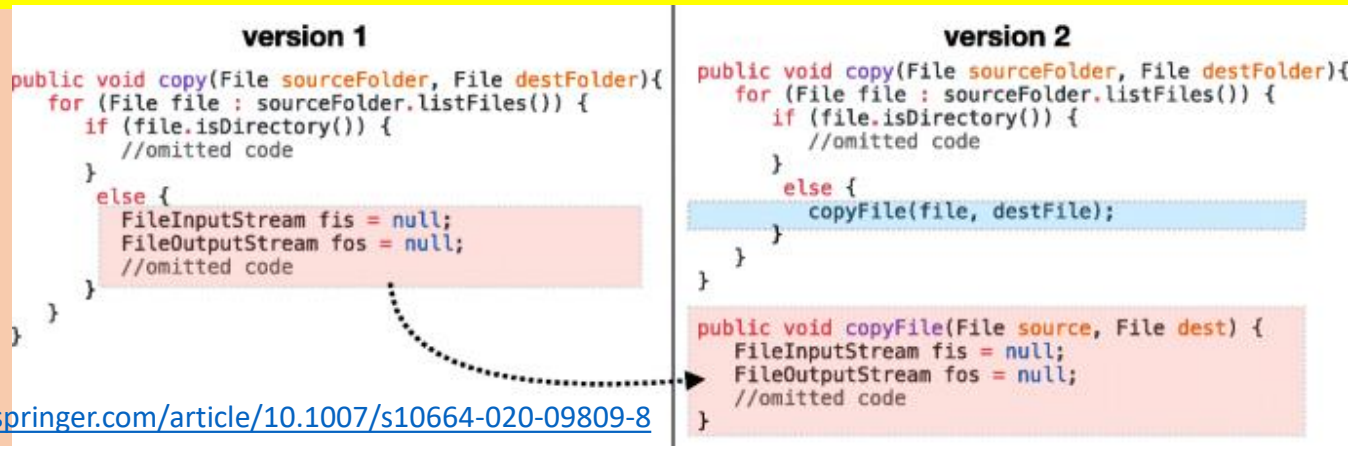
| With DIP | Without DIP |
|---|---|
| Easy to develop different components with TDD | Hard to test due to class dependencies |
| Easy to extend our components and apply OCP | Hard to extend as classes are tightly coupled |
| Easy to independently deploy parts of the system | Need to recompile all the software for a small fix |
| Easy to merge branches of work as changes are isolated | Hard to merge branches as code has dependencies |

# Summary

- SOLID encourage us to create more maintainable, understandable, and flexible software. Consequently, as our applications grow in size, we can reduce their complexity and save ourselves a lot of headaches further down the road!

# Refactor

05506004
https://dzone.com/articles/what-is-refactoring




https://link.springer.com/article/10.1007/s10664-020-09809-8

# Refactor

- The process of refactoring involves taking small, manageable steps that incrementally increase the cleanliness of code while still maintaining the functionality of the code.
  - It's the cumulative effect of many small refactorings performed toward a single goal that makes the difference.
- Refactoring is one of the most used terms in software development and has played a major role in the maintenance of software for decades. While most developers have an intuitive understanding of the refactoring process, many of us lack a true mastery of this important skill.

https://dzone.com/articles/what-is-refactoring

# Refactor

- Some tips to keep in mind when establishing a goal include:
  - **Keep it simple**: Start small and maintain focus on a reasonably-sized goal
  - **Break down large goals**: If a goal involves refactoring hundreds of classes at once, break down this goal into small pieces; pick a smaller subset of classes to refactor; once the refactor is completed on this subset, move onto another subset
  - **Be specific**: Do not fall into the trap of stating that a goal is to "tidy up a method" or "clean up a class" (vague and abstract goals); instead, set a more measurable goal, such as to remove nested loops, remove complicated conditionals, reduce the number of lines of a method by moving common code into a private method, etc.
  - **Focus on the end**: Do not get distracted by the intermediate steps that are required to reach a goal; if reducing an inheritance hierarchy requires the creation of a few classes and the removal of others, do not get bogged down and forget that the end goal is to refactor the hierarchy, not simply add and remove classes

https://dzone.com/articles/what-is-refactoring

# Refactor (cont.)

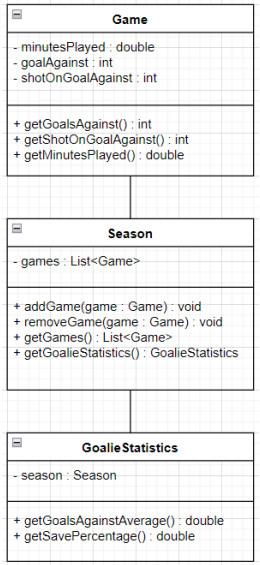
- The distinguishing feature of a well-designed program is its modularity, thanks to which it is enough to know only a small part of the code to introduce most modifications.
  - Modularity also makes it easier for new people to get in and start working more efficiently.
  - To achieve this modularity, related program elements must be grouped together, the connections being understandable and easy to find.
  - There is no single rule of thumb as to how this can be done.
- refactoring does not change how it works, but its structure, so all tests must be passed.

https://thecodest.co/blog/a-quick-primer-on-refactoring-for-beginners

# Beginner's refactoring techniques

- The transformations listed below are only examples but they are really helpful in daily programing:
  - Function extraction and variable extraction – if the function is too long, check if there are any minor functions that could be extracted. The same goes for long lines. These transformations can help with finding duplications in the code. Thanks to Small Functions, the code becomes clearer and more understandable.
  - Renaming of functions and variables – using the correct naming convention is essential to good programming. Variable names, when well-chosen, can tell a lot about the code.
  - Grouping the functions into a class – this change is helpful when two classes perform similar operations as it can shorten the length of the class.
  - etc

```java
public class Game {

    private final double minutesPlayed;
    private final int goalsAgainst;
    private final int shotsOnGoalAgainst;

    public Game(int goalsAgainst, int shotsOnGoalAgainst, double minutesPlayed) {
        this.goalsAgainst = goalsAgainst;
        this.shotsOnGoalAgainst = shotsOnGoalAgainst;
        this.minutesPlayed = minutesPlayed;
    }

    public int getGoalsAgainst() {
        return goalsAgainst;
    }

    public int getShotsOnGoalAgainst() {
        return shotsOnGoalAgainst;
    }

    public double getMinutesPlayed() {
        return minutesPlayed;
    }
}
```




```java
public class Season {

    private final List<Game> games;

    public Season(List<Game> games) {
        this.games = new ArrayList<>(games);
    }

    public Season() {
        this.games = new ArrayList<>();
    }

    public void addGame(Game game) {
        games.add(game);
    }

    public void removeGame(Game game) {
        games.remove(game);
    }

    public List<Game> getGames() {
        return games;
    }

    public GoalieStatistics getGoalieStatistics() {
        return new GoalieStatistics(this);
    }
}
```

```java
public class GoalieStatistics {

    private final Season season;

    public GoalieStatistics(Season season) {
        this.season = season;
    }

    public double getGoalsAgainstAverage() {

        if (season.getGames().isEmpty()) {
            return 0.0;
        }
        else {
            List<Game> games = season.getGames();
            int tga = 0;
            double mins = 0;

            for (Game game: games) {
                tga += game.getGoalsAgainst();
                mins += game.getMinutesPlayed();
            }

            return (tga / mins) * 60;
        }
    }
```

- **Goals Against Average (GAA)**: The number of goals scored against a goalie (called Goals Against, GA) divided by the number of minutes the goalie played for a season multiplied by 60 (the number of minutes per hockey game)
- **Save Percentage (SV%)**: The total number of saves (Shots on Goal, SOG, minus the number of goals scored on a goalie) divided by the total SOG.

  Stated formally:

$$GAA = 60\left(\frac{GA}{time\ played_{mins}}\right), \quad SV\% = \frac{SOG - Goals}{SOG}$$

```java
    public double getSavePercentage() {

        if (season.getGames().isEmpty()) {
            return 0.0;
        }
        else {
            List<Game> games = season.getGames();
            int g = 0;
            int tsoga = 0;

            for (Game game: games) {
                g += game.getGoalsAgainst();
                tsoga += game.getShotsOnGoalAgainst();
            }

            return ((double) tsoga - g) / tsoga;
        }
    }
}
```

# After Refactor (Season)

```java
public class Season {

    private final List<Game> games;

    public Season(List<Game> games) {
        this.games = new ArrayList<>(games);
    }

    public Season() {
        this.games = new ArrayList<>();
    }

    public void addGame(Game game) {
        games.add(game);
    }

    public void removeGame(Game game) {
        games.remove(game);
    }

    public List<Game> getGames() {
        return games;
    }

    public GoalieStatistics getGoalieStatistics() {
        return new GoalieStatistics(this);
    }
}
```

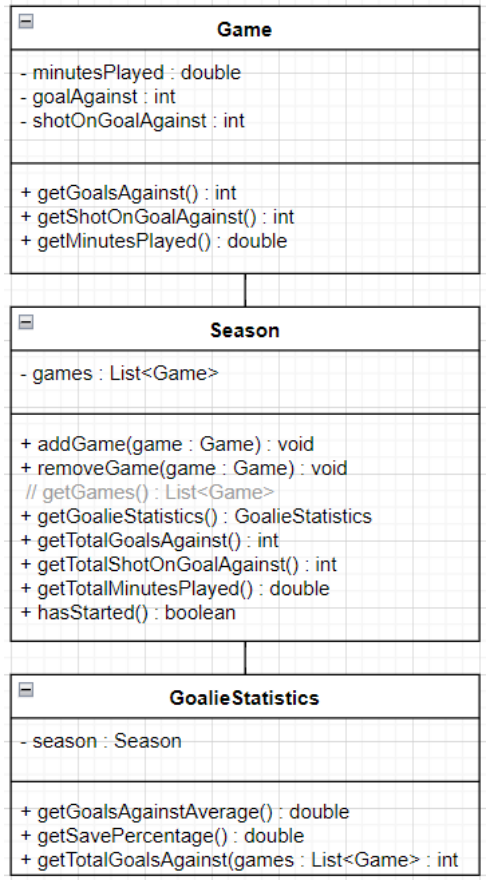
```java
public class Season {

    private final List<Game> games;

    public Season(List<Game> games) {
        this.games = new ArrayList<>(games);
    }

    public Season() {
        this.games = new ArrayList<>();
    }

    public void addGame(Game game) {
        games.add(game);
    }

    public void removeGame(Game game) {
        games.remove(game);
    }

/*    public List<Game> getGames() {
        return games;
    }
*/
    public GoalieStatistics getGoalieStatistics() {
        return new GoalieStatistics(this);
    }

    public int getTotalGoalsAgainst() {
        return games.stream().mapToInt(game -> game.getGoalsAgainst()).sum();
    }
    public int getTotalShotsOnGoalAgainst() {
        return games.stream().mapToInt(game -> game.getShotsOnGoalAgainst()).sum();
    }

    public double getTotalMinutesPlayed() {
        return games.stream().mapToDouble(game -> game.getMinutesPlayed()).sum();
    }
    public boolean hasStarted() {
        return !games.isEmpty();
    }
}
```

Not a good idea to let Goalies access games directly

# After Refactor (GoalieStat)

```java
public class GoalieStatistics {

    private final Season season;

    public GoalieStatistics(Season season) {
        this.season = season;
    }

    public double getGoalsAgainstAverage() {

        if (season.getGames().isEmpty()) {
            return 0.0;
        }
        else {
            List<Game> games = season.getGames();
            int tga = 0;
            double mins = 0;

            for (Game game: games) {
                tga += game.getGoalsAgainst();
                mins += game.getMinutesPlayed();
            }

            return (tga / mins) * 60;
        }
    }
```

```java
    public double getSavePercentage() {

        if (season.getGames().isEmpty()) {
            return 0.0;
        }
        else {
            List<Game> games = season.getGames();
            int g = 0;
            int tsoga = 0;

            for (Game game: games) {
                g += game.getGoalsAgainst();
                tsoga += game.getShotsOnGoalAgainst();
            }

            return ((double) tsoga - g) / tsoga;
        }
    }
}
```




```java
class GoalieStatistics {

    private final Season season;

    public GoalieStatistics(Season season) {
        this.season = season;
    }

    public double getGoalsAgainstAverage() {

        if (season.hasStarted()) {

            int tga = season.getTotalGoalsAgainst();
            double mins = season.getTotalMinutesPlayed();
            return (tga / mins) * 60;

        } else {
            return 0.0;
        }
    }

    public double getSavePercentage() {

        if (season.hasStarted()) {

            int g  = season.getTotalGoalsAgainst();
            int tsoga = season.getTotalShotsOnGoalAgainst();
            return ((double) tsoga - g) / tsoga;

        } else {
            return 0.0;
        }
    }
}
```

# Season and GoalieStat

```java
public class Season {

    private final List<Game> games;

    public Season(List<Game> games) {
        this.games = new ArrayList<>(games);
    }

    public Season() {
        this.games = new ArrayList<>();
    }

    public void addGame(Game game) {
        games.add(game);
    }

    public void removeGame(Game game) {
        games.remove(game);
    }

/*    public List<Game> getGames() {
        return games;
    }
*/
    public GoalieStatistics getGoalieStatistics() {
        return new GoalieStatistics(this);
    }

    public int getTotalGoalsAgainst() {
        return games.stream().mapToInt(game -> game.getGoalsAgainst()).sum();
    }
    public int getTotalShotsOnGoalAgainst() {
        return games.stream().mapToInt(game -> game.getShotsOnGoalAgainst()).sum();
    }

    public double getTotalMinutesPlayed() {
        return games.stream().mapToDouble(game -> game.getMinutesPlayed()).sum();
    }
    public boolean hasStarted() {
        return !games.isEmpty();
    }
}
```

```java
class GoalieStatistics {

    private final Season season;

    public GoalieStatistics(Season season) {
        this.season = season;
    }

    public double getGoalsAgainstAverage() {

        if (season.hasStarted()) {

            int tga = season.getTotalGoalsAgainst();
            double mins = season.getTotalMinutesPlayed();
            return (tga / mins) * 60;

        } else {
            return 0.0;
        }
    }

    public double getSavePercentage() {

        if (season.hasStarted()) {

            int g  = season.getTotalGoalsAgainst();
            int tsoga = season.getTotalShotsOnGoalAgainst();
            return ((double) tsoga - g) / tsoga;

        } else {
            return 0.0;
        }
    }
}
```

# Split (prepare to extract)

```java
68  public double getGoalsAgainstAverage() {
69
70      if (season.getGames().isEmpty()) {
71          return 0.0;
72      }
73      else {
74          List<Game> games = season.getGames();
75          int tga = 0;
76          double mins = 0;
77
78          for (Game game: games) {
79              tga += game.getGoalsAgainst();
80              mins += game.getMinutesPlayed();
81          }
82
83          return (tga / mins) * 60;
84      }
85  }
87  public double getSavePercentage() {
88
89      if (season.getGames().isEmpty()) {
90          return 0.0;
91      }
92      else {
93          List<Game> games = season.getGames();
94          int g = 0;
95          int tsoga = 0;
96
97          for (Game game: games) {
98              g += game.getGoalsAgainst();
99              tsoga += game.getShotsOnGoalAgainst();
100         }
101
102         return ((double) tsoga - g) / tsoga;
103     }
104 }
```

```java
public double getGoalsAgainstAverage() {

    if (season.getGames().isEmpty()) {
        return 0.0;
    }
    else {
        List<Game> games = season.getGames();

        int tga = 0;
        for (Game game : games) {
            tga += game.getGoalsAgainst();
        }

        double mins = 0;

        for (Game game: games) {
            mins += game.getMinutesPlayed();
        }

        return (tga / mins) * 60;
    }
}
```

```java
public double getSavePercentage() {

    if (season.getGames().isEmpty()) {
        return 0.0;
    }
    else {
        List<Game> games = season.getGames();

        int g = 0;
        for (Game game : games) {
            g += game.getGoalsAgainst();
        }

        int tsoga = 0;
        for (Game game: games) {
            tsoga += game.getShotsOnGoalAgainst();
        }

        return ((double) tsoga - g) / tsoga;
    }
}
```

# Extract

```java
public double getGoalsAgainstAverage() {

    if (season.getGames().isEmpty()) {
        return 0.0;
    }
    else {
        List<Game> games = season.getGames();

        int tga = 0;
        for (Game game : games) {
            tga += game.getGoalsAgainst();
        }

        double mins = 0;

        for (Game game: games) {
            mins += game.getMinutesPlayed();
        }

        return (tga / mins) * 60;
    }
}
```

```java
public double getSavePercentage() {

    if (season.getGames().isEmpty()) {
        return 0.0;
    }
    else {
        List<Game> games = season.getGames();

        int g = 0;
        for (Game game : games) {
            g += game.getGoalsAgainst();
        }

        int tsoga = 0;
        for (Game game: games) {
            tsoga += game.getShotsOnGoalAgainst();
        }

        return ((double) tsoga - g) / tsoga;
    }
}
```

```java
public double getGoalsAgainstAverage() {

    if (season.getGames().isEmpty()) {
        return 0.0;
    }
    else {
        List<Game> games = season.getGames();

        int tga = getTotalGoalsAgainst(games);

        double mins = 0;

        for (Game game: games) {
            mins += game.getMinutesPlayed();
        }

        return (tga / mins) * 60;
    }
}
private int getTotalGoalsAgainst(List<Game> games) {
    int g = 0;

    for (Game game: games) {
        g += game.getGoalsAgainst();
    }
    return g;
}
public double getSavePercentage() {

    if (season.getGames().isEmpty()) {
        return 0.0;
    }
    else {
        List<Game> games = season.getGames();

        int g = getTotalGoalsAgainst(games);

        int tsoga = 0;
        for (Game game: games) {
            tsoga += game.getShotsOnGoalAgainst();
        }

        return ((double) tsoga - g) / tsoga;
    }
}
```

# (Use stream thus) the method belonged to season

```java
public double getGoalsAgainstAverage() {

    if (season.getGames().isEmpty()) {
        return 0.0;
    }
    else {
        List<Game> games = season.getGames();

        int tga = getTotalGoalsAgainst(games);

        double mins = 0;

        for (Game game: games) {
            mins += game.getMinutesPlayed();
        }

        return (tga / mins) * 60;
    }
}

private int getTotalGoalsAgainst(List<Game> games) {
    int g = 0;

    for (Game game: games) {
        g += game.getGoalsAgainst();
    }
    return g;
}
public double getSavePercentage() {

    if (season.getGames().isEmpty()) {
        return 0.0;
    }
    else {
        List<Game> games = season.getGames();

        int g = getTotalGoalsAgainst(games);

        int tsoga = 0;
        for (Game game: games) {
            tsoga += game.getShotsOnGoalAgainst();
        }

        return ((double) tsoga - g) / tsoga;
    }
}
```

```java
public class Season {

    private final List<Game> games;

    public Season(List<Game> games) {
        this.games = new ArrayList<>(games);
    }

    public Season() {
        this.games = new ArrayList<>();
    }

    public void addGame(Game game) {
        games.add(game);
    }

    public void removeGame(Game game) {
        games.remove(game);
    }

    public List<Game> getGames() {
        return games;
    }

    public GoalieStatistics getGoalieStatistics() {
        return new GoalieStatistics(this);
    }

    public int getTotalGoalsAgainst() {
        return games.stream().mapToInt(game -> game.getGoalsAgainst()).sum();
    }
}
```

```java
public double getGoalsAgainstAverage() {

    if (season.getGames().isEmpty()) {
        return 0.0;
    }
    else {
        List<Game> games = season.getGames();

        //int tga = getTotalGoalsAgainst(games);
        int tga = season.getTotalGoalsAgainst();

        double mins = 0;

        for (Game game: games) {
            mins += game.getMinutesPlayed();
        }

        return (tga / mins) * 60;
    }
}
/*
private int getTotalGoalsAgainst(List<Game> games) {
    int g = 0;

    for (Game game: games) {
        g += game.getGoalsAgainst();
    }
    return g;
}
*/
 public double getSavePercentage() {

    if (season.getGames().isEmpty()) {
        return 0.0;
    }
    else {
        List<Game> games = season.getGames();

        //int g = getTotalGoalsAgainst(games);
        int g = season.getTotalGoalsAgainst();

        int tsoga = 0;
        for (Game game: games) {
            tsoga += game.getShotsOnGoalAgainst();
        }

        return ((double) tsoga - g) / tsoga;
    }
}
```

# Apply to getTotalShotsOnGoalAgainst() and getTotalMinutesPlayed()

```java
public class Season {

    private final List<Game> games;

    public Season(List<Game> games) {
        this.games = new ArrayList<>(games);
    }

    public Season() {
        this.games = new ArrayList<>();
    }

    public void addGame(Game game) {
        games.add(game);
    }

    public void removeGame(Game game) {
        games.remove(game);
    }

    public List<Game> getGames() {
        return games;
    }

    public GoalieStatistics getGoalieStatistics() {
        return new GoalieStatistics(this);
    }

    public int getTotalGoalsAgainst() {
        return games.stream().mapToInt(game -> game.getGoalsAgainst()).sum();
    }
}
```

```java
/*
private int getTotalGoalsAgainst(List<Game> games) {
    int g = 0;

    for (Game game: games) {
        g += game.getGoalsAgainst();
    }
    return g;
}
*/
```

```java
public double getGoalsAgainstAverage() {

    if (season.getGames().isEmpty()) {
        return 0.0;
    }
    else {
        List<Game> games = season.getGames();

        //int tga = getTotalGoalsAgainst(games);
        int tga = season.getTotalGoalsAgainst();

        double mins = 0;

        for (Game game: games) {
            mins += game.getMinutesPlayed();
        }

        return (tga / mins) * 60;
    }
}
```

```java
public double getSavePercentage() {

    if (season.getGames().isEmpty()) {
        return 0.0;
    }
    else {
        List<Game> games = season.getGames();

        //int g = getTotalGoalsAgainst(games);
        int g = season.getTotalGoalsAgainst();

        int tsoga = 0;
        for (Game game: games) {
            tsoga += game.getShotsOnGoalAgainst();
        }

        return ((double) tsoga - g) / tsoga;
    }
}
```

```java
public int getTotalGoalsAgainst() {
    return games.stream().mapToInt(game -> game.getGoalsAgainst()).sum();
}
public int getTotalShotsOnGoalAgainst() {
    return games.stream().mapToInt(game -> game.getShotsOnGoalAgainst()).sum();
}

public double getTotalMinutesPlayed() {
    return games.stream().mapToDouble(game -> game.getMinutesPlayed()).sum();
}
```

```java
public double getGoalsAgainstAverage() {

    if (season.getGames().isEmpty()) {
        return 0.0;
    }
    else {
        //List<Game> games = season.getGames();

        int tga = season.getTotalGoalsAgainst();

        double mins = season.getTotalMinutesPlayed();

        return (tga / mins) * 60;
    }
}
public double getSavePercentage() {

    if (season.getGames().isEmpty()) {
        return 0.0;
    }
    else {
        //List<Game> games = season.getGames();

        int g  = season.getTotalGoalsAgainst();

        int tsoga = season.getTotalShotsOnGoalAgainst();

        return ((double) tsoga - g) / tsoga;
    }
}
```

# Create hasStarted()

```
public int getTotalGoalsAgainst() {
    return games.stream().mapToInt(game -> game.getGoalsAgainst()).sum();
}
public int getTotalShotsOnGoalAgainst() {
    return games.stream().mapToInt(game -> game.getShotsOnGoalAgainst()).sum();
}

public double getTotalMinutesPlayed() {
    return games.stream().mapToDouble(game -> game.getMinutesPlayed()).sum();
}
public double getGoalsAgainstAverage() {

    if (season.getGames().isEmpty()) {
        return 0.0;
    }
    else {
        //List<Game> games = season.getGames();

        int tga = season.getTotalGoalsAgainst();

        double mins = season.getTotalMinutesPlayed();

        return (tga / mins) * 60;
    }
}
public double getSavePercentage() {

    if (season.getGames().isEmpty()) {
        return 0.0;
    }
    else {
        //List<Game> games = season.getGames();

        int g  = season.getTotalGoalsAgainst();

        int tsoga = season.getTotalShotsOnGoalAgainst();

        return ((double) tsoga - g) / tsoga;
    }
}
```

```
public int getTotalGoalsAgainst() {
    return games.stream().mapToInt(game -> game.getGoalsAgainst()).sum();
}
public int getTotalShotsOnGoalAgainst() {
    return games.stream().mapToInt(game -> game.getShotsOnGoalAgainst()).sum();
}

public double getTotalMinutesPlayed() {
    return games.stream().mapToDouble(game -> game.getMinutesPlayed()).sum();
}
public boolean hasStarted() {
    return !games.isEmpty();
}
public double getGoalsAgainstAverage() {

    //if (season.getGames().isEmpty()) {
    if (!season.hasStarted()) {
        return 0.0;
    }
    else {

        int tga = season.getTotalGoalsAgainst();

        double mins = season.getTotalMinutesPlayed();

        return (tga / mins) * 60;
    }
}

public double getSavePercentage() {

    //if (season.getGames().isEmpty()) {
    if (!season.hasStarted()) {
        return 0.0;
    }
    else {

        int g  = season.getTotalGoalsAgainst();

        int tsoga = season.getTotalShotsOnGoalAgainst();

        return ((double) tsoga - g) / tsoga;
    }
}
```

# Make if easier to understand

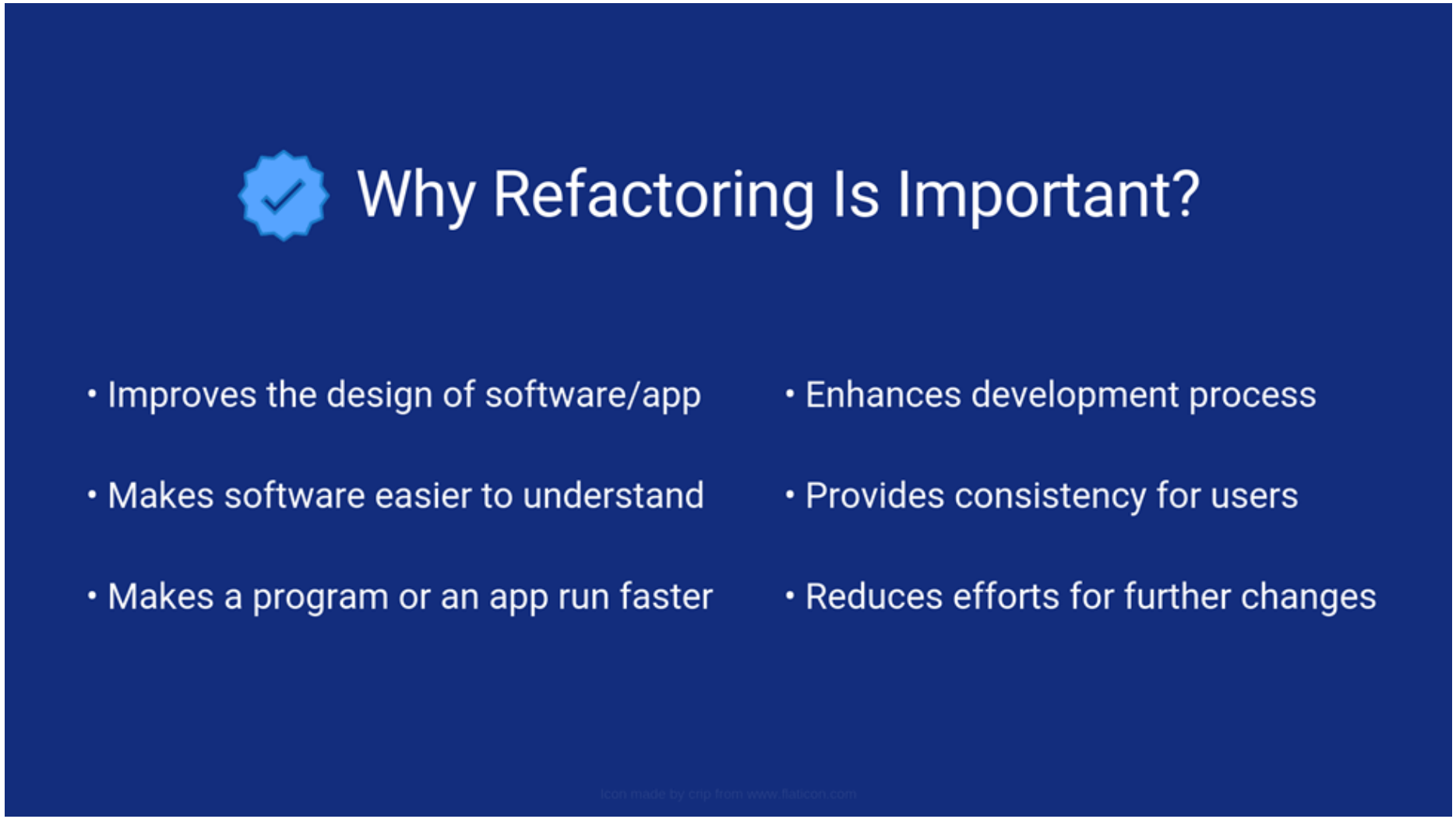
```java
public int getTotalGoalsAgainst() {
    return games.stream().mapToInt(game -> game.getGoalsAgainst()).sum();
}
public int getTotalShotsOnGoalAgainst() {
    return games.stream().mapToInt(game -> game.getShotsOnGoalAgainst()).sum();
}

public double getTotalMinutesPlayed() {
    return games.stream().mapToDouble(game -> game.getMinutesPlayed()).sum();
}
public boolean hasStarted() {
    return !games.isEmpty();
}
public double getGoalsAgainstAverage() {

    //if (season.getGames().isEmpty()) {
    if (!season.hasStarted()) {
        return 0.0;
    }
    else {

        int tga = season.getTotalGoalsAgainst();

        double mins = season.getTotalMinutesPlayed();

        return (tga / mins) * 60;
    }
}

public double getSavePercentage() {

    //if (season.getGames().isEmpty()) {
    if (!season.hasStarted()) {
        return 0.0;
    }
    else {

        int g  = season.getTotalGoalsAgainst();

        int tsoga = season.getTotalShotsOnGoalAgainst();

        return ((double) tsoga - g) / tsoga;
    }
}
```

```java
9   public double getGoalsAgainstAverage() {
10
11      if (season.hasStarted()) {
12          int totalGoalsAgainst = season.getTotalGoalsAgainst();
13          double totalMinutesPlayed = season.getTotalMinutesPlayed();
14
15          return (totalGoalsAgainst / totalMinutesPlayed) * 60;
16      }
17      else {
18          return 0.0;
19      }
20  }
21
22  public double getSavePercentage() {
23
24      if (season.hasStarted()) {
25          int totalGoalsAgainst = season.getTotalGoalsAgainst();
26          int totalSogAgainst = season.getTotalShotsOnGoalAgainst();
27
28          return ((double) totalSogAgainst - totalGoalsAgainst) / totalSogAgainst;
29      }
30      else {
31          return 0.0;
32      }
33  }
```

# More on code refactoring

- The indisputable benefits of refactoring:
  - Improving the objective readability of the code by reducing or restructuring it.
  - Encouraging developers to write software more thoughtfully, following a given style and criteria.
  - Preparing a springboard for creating reusable pieces of code.
  - Making it easier to find bugs in a large amount of code.
- When to refactor
  - When adding features -> By cleaning the code base before adding new features or updates, it helps to make the product more robust and easier to use in the future.
  - During a code review
    - This is the last chance to clean up the code before the product is launched. Doing so can help you identify areas of the code that are worth fixing.
    - It's also a good time to start refactoring after the product has been launched. It allows them to get more work done before going on to the next project.

https://maddevs.io/blog/code-refactoring/




https://deepinspire.com/blog/in-clean-code-we-trust-why-and-how-to-do-code-refactoring/

# Recommended SOLID Readings

- https://reflectoring.io/single-responsibility-principle/
- https://reflectoring.io/open-closed-principle-explained/
- https://reflectoring.io/lsp-explained/
- https://reflectoring.io/interface-segregation-principle/
- https://stackify.com/dependency-inversion-principle/

# Further Readings

- https://dzone.com/articles/what-is-refactoring
- https://thecodest.co/blog/a-quick-primer-on-refactoring-for-beginners
- Refactoring
- https://refactoring.guru/smells/long-method
- Pros And Cons Of Code Refactoring
- https://www.c-sharpcorner.com/article/pros-and-cons-of-code-refactoring/
- Code Refactoring: Meaning, Benefits and Best Practices
- https://maddevs.io/blog/code-refactoring/
- Code Refactoring
- https://youtube.com/playlist?list=PLGLfVvz_LVvSuz6NuHAzpM52qKM6bPlCV
- Learn Java Programming - Coupling Tutorial
  - https://youtu.be/Eq5ReWFlc6w
  - Daniel Ross