Solutions to the SOLID Coding Challenges

## 01 Single Responsibility Principle

The given class was mixing up data loading and data formatting responsibilities, which should be separated in two different classes.

If you are ever tempted to use the word “and” as part of a class name, then you're probably about to break the Single Responsibility Principle, and you probably should try moving consistent methods to a different class.

**DataLoader.java:**

**public class** DataLoader {

**public** String loadData() {

*/\* Code that gets the original data;*

*in the current implementation, it always returns the same String of data \*/*

**return "-Washington:1789 -Adams:1797 -Jefferson:1801"**;

}

}

**DataFormatter.java:**

**public class DataFormatter {**

**public** String formatData(String unformattedData) {

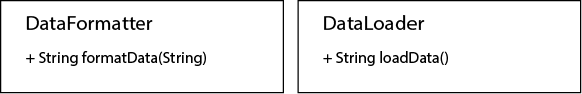
*/\* Formats the given data;*

*in the current implementation it just replaces whitespaces " " with new lines "\n" \*/*

**return** unformattedData.replace(**" "**, **"\n"**);

}

}



## 02 Single Responsibility Principle

The given class contained methods that dealt with managing customer accounts, plus a method that sent notifications to the user. It makes sense to separate them, so all the methods dealing with accounts remain together, while the messaging method goes to its own class (for example, a notification service class:)

**CustomerAccountService.java:**

**public class CustomerAccountService {**

**public void** activateAccount() {

*// Logic that activates a new account goes here*

}

**public void** closeAccount() {

*// Logic that closes an account goes here*

}

}

**NotificationService.java:**

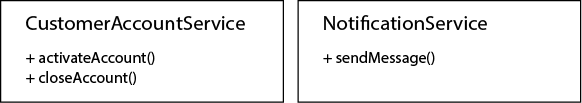
**public class NotificationService {**

**public void** sendMessage() {

*// Logic that sends a message to the customer goes here*

}

}



## 03 Open / Closed Principle

The problem with the original design is that the method that calculates interest rates for accounts is included as part of the Account class, embedding this business logic with the core model object. It is impossible to add new logical rules without recompiling the class.

Ideally, the way interest rates are calculated may be modified in the future without modifying the base Account class. Hence, creating an interface for the component that calculates the interest rates would make sense. Let's call it 'IInterestCalculator'. To be open for extension, this interface should allow to:

* Be able to add new logical rules used to calculate the interest rate (so these rules could be added or modified without recompiling the Account class.)
* And calculate the interest rate to apply to the given account (as we were originally intending.)

Again, the implementation of these rules may vary in the future, so for maximum flexibility, we only need them to implement a basic interface, 'IInterestRule' that:

* Determines if the rule can be applied to the given conditions.
* And obtains the interest rate that would be in effect if the rule applied.

This is the code of these new interfaces:

**IInterestCalculator.java:**

**import java.util.ArrayList;**

**public interface** IInterestCalculator {

**public void** addRule(IInterestRule rule);

**public double** calculateInterest(**int** yearOfCreation, **double** balance);

}

**IInterestRule.java:**

**public interface IInterestRule {**

**public boolean** isRuleApplicable(**int** yearOfCreation, **double** balance);

**public double** getInterestRate();

}

Then, we need to create concrete specific classes that implement those interfaces, and that we can use in our code.

First, all the logical rules that we were applying to calculate interest rates should now be moved apart. In this case, we create a new clase for each one of them, implementing the IInterestRule interface. We include a rule for the default case, and we also add a new rule, getting an interest rate for the new case defined with a balance over half a million:

**InterestRuleOverMillion.java:**

**public class** InterestRuleOverMillion **implements** IInterestRule {

@Override

**public boolean** isRuleApplicable(**int** yearOfCreation, **double** balance) {

**return** (balance > 1000000);

}

@Override

**public double** getInterestRate() {

**return** 2.1;

}

}

**InterestRuleOverHalfMillion.java:**

**public class** InterestRuleOverHalfMillion **implements** IInterestRule {

*// New Rule*

@Override

**public boolean** isRuleApplicable(**int** yearOfCreation, **double** balance) {

**return** (balance > 500000);

}

@Override

**public double** getInterestRate() {

**return** 1.4;

}

}

**InterestRuleOlderTwentyTen.java:**

**public class** InterestRuleOlderTwentyTen **implements** IInterestRule {

@Override

**public boolean** isRuleApplicable(**int** yearOfCreation, **double** balance) {

**return** (yearOfCreation < 2010);

}

@Override

**public double** getInterestRate() {

**return** 0.1;

}

}

**InterestRuleDefault.java**

**public class** InterestRuleDefault **implements** IInterestRule {

@Override

**public boolean** isRuleApplicable(**int** yearOfCreation, **double** balance) {

**return true**; *// Applies to any account*

}

@Override

**public double** getInterestRate() {

**return** 0.01;

}

}

If we had to modify the interest rate returned by one of these rules, we would only have to recompile that single rule – the interest calculator or account classes wouldn't change at all, being protected from these modifications.

The method to calculate the interest to apply (which was part of the old Account class) now should be part of an implementation of the IInterestCalculator interface, including all the previously defined rules:

**InterestCalculator.java:**

**import** java.util.ArrayList;

**public class** InterestCalculator **implements** IInterestCalculator {

**private** ArrayList<IInterestRule> **interestRules**;

**public** InterestCalculator() {

**this**.**interestRules** = **new** ArrayList<>();

**this**.addRule(**new** InterestRuleOverMillion());

**this**.addRule(**new** InterestRuleOverHalfMillion()); *// New rule added*

**this**.addRule((**new** InterestRuleOlderTwentyTen()));

**this**.addRule(**new** InterestRuleDefault());

}

@Override

**public void** addRule(IInterestRule rule) {

**this**.**interestRules**.add(rule);

}

@Override

**public double** calculateInterest(**int** yearOfCreation, **double** balance) {

**for**(**int** i = 0; i < **this**.**interestRules**.size(); i++) {

IInterestRule currentRule = **this**.**interestRules**.get(i);

**if**(currentRule.isRuleApplicable(yearOfCreation, balance)) {

**return** currentRule.getInterestRate();

}

}

*// Default case (no rule matched)*

**return** 0;

}

}

Finally, we would modify the Account class to make use of any class implementing IInterestCalculator to obtain the appropriate interest rate:

**Account.java:**

**public class** Account {

**private int yearOfCreation**;

**private double balance**;

**private double interestRate**;

**public double** getInterestRate() {

**return interestRate**;

}

**public** Account(**int** yearOfCreation, **double** balance) {

**this**.**yearOfCreation** = yearOfCreation;

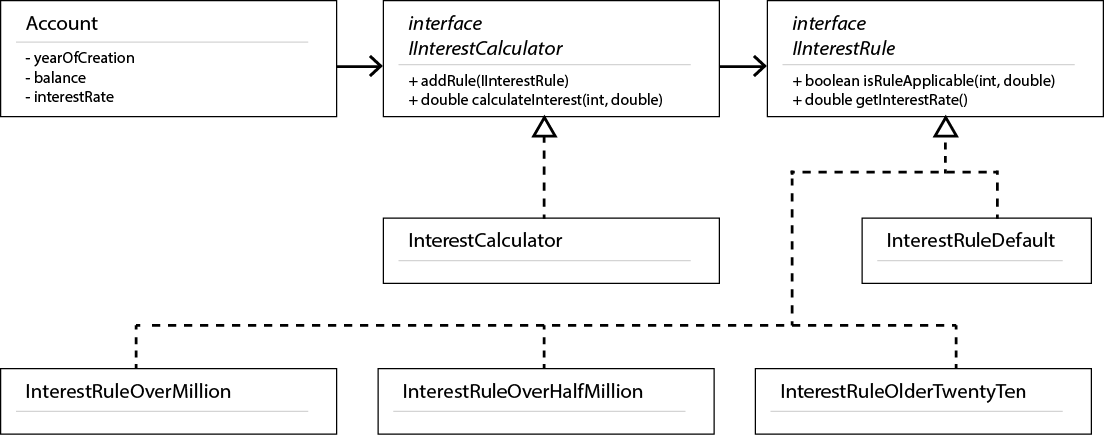
**this**.**balance** = balance;

IInterestCalculator accountInterestCalculator = **new** InterestCalculator();

**this**.**interestRate** = accountInterestCalculator.calculateInterest(yearOfCreation, balance);

}

}



## 04 Open / Closed Principle

There's one method from TaskList whose implementation should vary without having to modify the rest of the TaskList class. To solve this problem following the Open for Extension / Closed for Modification approach, the usage of getNextTask() has to be abstracted in some way, to provide different implementations without modifying the base class.

There are different ways to achieve this, and they rely on creating a common interface to implement, or a base class to extend.

In this case, we're going to turn the TaskList into an abstract class, where all other methods will already be implemented (closed for modification) but the getNextTask method will remain abstract and pending concrete implementations (open for extension) in what is sometimes called a Template Pattern:

**TaskList.java:**

**import java.util.ArrayList;**

**import** java.util.List;

**public abstract class** TaskList {

**protected** ArrayList<Task> **list**;

*/\*\**

*\* Constructor*

*\** ***@param list*** *an array filled with Tasks*

*\*/*

**public** TaskList(List<Task> list){

**this**.**list** = **new** ArrayList<Task>(list);

}

**public boolean** hasMoreTasks() {

**return** (**this**.**list** != **null** && (!**this**.**list**.isEmpty()));

}

**public abstract** Task getNextTask(); *// Other concrete classes need to implement this method so they can be instantiated*

}

**FIFOTaskList.java:**

**import** java.util.List;

*/\*\**

*\* First in, first out (FIFO)*

*\*/*

**public class** FIFOTaskList **extends** TaskList {

*/\*\**

*\* Constructor*

*\** ***@param list*** *an array filled with Tasks*

*\*/*

**public** FIFOTaskList(List<Task> list) {

**super**(list);

}

@Override

**public** Task getNextTask() {

**if**(**this**.hasMoreTasks()) {

**return this**.**list**.remove(0);

} **else** {

**return null**;

}

}

}

**LIFOTaskList.java:**

**import java.util.List;**

*/\*\**

*\* Last in, first out (LIFO)*

*\*/*

**public class** LIFOTaskList **extends** TaskList {

*/\*\**

*\* Constructor*

*\** ***@param list*** *an array filled with Tasks*

*\*/*

**public** LIFOTaskList(List<Task> list) {

**super**(list);

}

@Override

**public** Task getNextTask() {

**if**(**this**.hasMoreTasks()) {

*// indexes are zero-based*

*// so the index of the last element is the length minus one*

**int** lastTaskIndex = **this**.**list**.size() - 1;

**return this**.**list**.remove(lastTaskIndex);

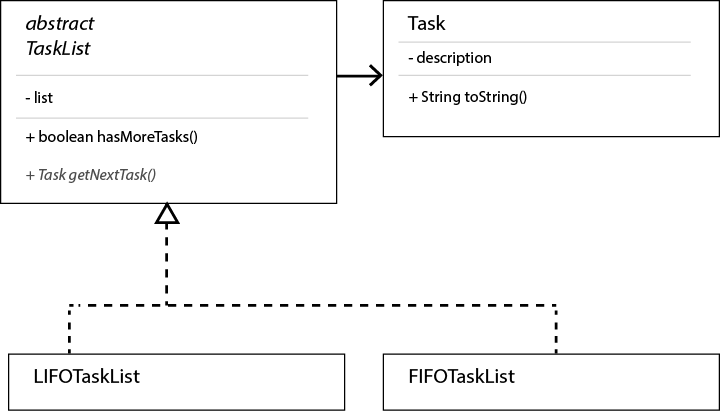
} **else** {

**return null**;

}

}

}



## 05 Liskov's Replacement Principle

The problem in the previous case is that, even when a Linear Equation IS A Quadratic Equation (with a zero 'a' coefficient,) a Linear Equation cannot always REPLACE a Quadratic Equation.

In our current implementation, the superclass (Quadratic Equation) allows to individually specify the value of each coefficient. In a Linear Equation, the 'a' coefficient must be zero, but this restriction isn't enforced, and the value is just ignored when solving it, rendering unexpected results for ideally the same scenario.

The best way to solve this problem is to separate Quadratic and Linear equations in two different classes, as one cannot be safely replaced by the other. However, since they still share some common structure, they can both inherit from the same interface, though using different constructors.

Let's create the new IEquation interface, that both kinds of equations implement:

**IEquation.java:**

**public interface** IEquation {

**public double**[] solve();

}

The new implementation of the original equations would be as follows:

**QuadraticEquation.java:**

**public class QuadraticEquation implements IEquation {**

**private int a**; *// ax2, quadratic coefficient*

**private int b**; *// bx, linear coefficient*

**private int c**; *// c, constant*

**public** QuadraticEquation(**int** a, **int** b, **int** c) {

**this**.**a** = a;

**this**.**b** = b;

**this**.**c** = c;

}

**public double**[] solve() {

**double** squareRootCalculation = Math.*sqrt*(Math.*pow*(**b**, 2) - 4 \* **a** \* **c**);

**double** root1 = (-**b** + squareRootCalculation) / (2 \* **a**);

**double** root2 = (-**b** - squareRootCalculation) / (2 \* **a**);

**double**[] solutions = **new double**[] {root1, root2};

**return** solutions;

}

**public** String toString() {

**return a** + **"x2 + "** + **b** + **"x + "** + **c**;

}

}

**LinearEquation.java:**

*// 0x2 + bx + c = 0*

*// A Linear Equation IS-An Equation and IS-SUBSTITUTABLE-FOR an Equation*

*// A Linear Equation IS-A Quadratic Equation*

*// but it may NOT SUBSTITUTE a Quadratic Equation that uses a quadratic coefficient other than zero*

**public class** LinearEquation **implements** IEquation {

*// private int a; // ax2, quadratic coefficient always zero in a linear equation*

**private int b**; *// bx, linear coefficient*

**private int c**; *// c, constant*

**public** LinearEquation(**int** b, **int** c) {

**this**.**b** = b;

**this**.**c** = c;

}

**public double**[] solve() {

**double** root = ((**double**)(-**c**) / (**double**)(**b**));

**double**[] solutions = **new double**[] {root};

**return** solutions;

}

**public** String toString() {

**return b** + **"x + "** + **c**;

}

}

Trying to use this new code is now confusion-free, as both equations inherit from the same interface, but use different constructors:

**Main.java:**

**public class** Main {

**public static void** main(String[] args) {

**int** a = 2;

**int** b = 10;

**int** c = 5;

IEquation quadraticEquation = **new** QuadraticEquation(a, b, c);

**double**[] solutions = quadraticEquation.solve();

System.***out***.println(**"Solutions for the equation "** + quadraticEquation);

**for**(**int** i = 0; i < solutions.**length**; i++) {

System.***out***.println(solutions[i]);

}

*// In a linear equation, a is always zero, and there is only a solution*

*// Now both implement the same interface, but enforce different restrictions*

IEquation linearEquation = **new** LinearEquation(b, c);

**double**[] solutionsForTwoEqualParameters = linearEquation.solve();

System.***out***.println(**"Solutions for a different equation "** + linearEquation);

**for**(**int** i = 0; i < solutionsForTwoEqualParameters.**length**; i++) {

System.***out***.println(solutionsForTwoEqualParameters[i]);

}

}

}

Prints:

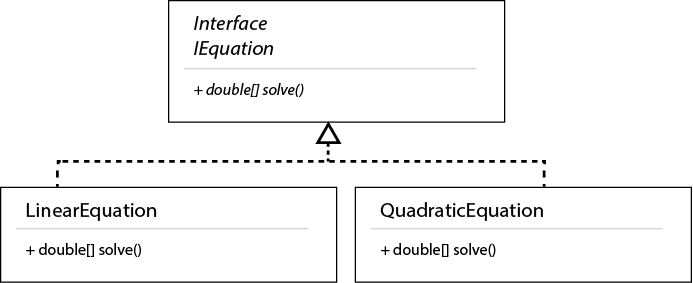
Solutions for the equation 2x2 + 10x + 5

-0.5635083268962915

-4.436491673103708

Solutions for a different equation 10x + 5

-0.5



## 06 Liskov's Replacement Principle

The problem in the previous case is that, even when an Equilateral Triangle IS A Triangle (with three identical sides,) an Equilateral Triangle cannot always REPLACE a Triangle.

In our current implementation, the superclass (Triangle) allows to individually specify the value of each side. In an Equilateral Triangle, the three sides must measure the same, but this restriction isn't enforced, and other side lengths are just ignored when calculating the perimeter, rendering unexpected results for ideally the same scenario.

The best way to solve this problem is to separate Triangles and Equilateral Triangles in two different classes, as one cannot be safely replaced by the other. However, since they still share some common structure, they can both inherit from the same interface (Shape, for example), while using different constructors.

**Shape.java:**

**public interface** Shape {

**public int** getPerimeter();

}

**Triangle.java:**

**public class Triangle implements Shape {**

**private int sideOneLength**;

**private int sideTwoLength**;

**private int sideThreeLength**;

**public** Triangle(**int** sideOneLength, **int** sideTwoLength, **int** sideThreeLength) {

**this**.**sideOneLength** = sideOneLength;

**this**.**sideTwoLength** = sideTwoLength;

**this**.**sideThreeLength** = sideThreeLength;

}

**public int** getPerimeter() {

**return sideOneLength** + **sideTwoLength** + **sideThreeLength**;

}

}

**EquilateralTriangle.java:**

**public class** EquilateralTriangle **implements** Shape {

**private int sideLength**;

**public** EquilateralTriangle(**int** sideLength) {

**this**.**sideLength** = sideLength;

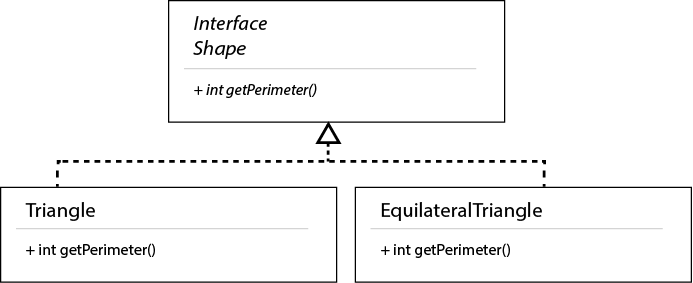
}

**public int** getPerimeter() {

**return** 3 \* **this**.**sideLength**;

}

}



When using these new classes, an EquilateralTriangle would only take the length of one side as a parameter, as they three are equal. It won't be able to substitute any Triangle, and it will return consistent results:

**Main.java:**

**public class** Main {

**public static void** main(String[] args) {

**int** sideOneLength = 10;

**int** sideTwoLength = 5;

**int** sideThreeLength = 7;

Shape myTriangle = **new** Triangle(sideOneLength, sideTwoLength, sideThreeLength);

**int** myTrianglePerimeter = myTriangle.getPerimeter();

System.***out***.println(**"The perimeter of my triangle is: "** + myTrianglePerimeter);

Shape equilateralTriangle = **new** EquilateralTriangle(sideOneLength);

**int** equilateralTrianglePerimeter = equilateralTriangle.getPerimeter();

System.***out***.println(**"The perimeter of my equilateral triangle is: "** + equilateralTrianglePerimeter);

}

}

The output of the previous code would now be consistent with the two different triangles defined:

The perimeter of my triangle is: 22

The perimeter of my equilateral triangle is: 30

## 07 Interface Segregation Principle

In the given case, our DefaultHeaderBanner class only needs to access three name-related methods from the user preferences. However, it is forced to implement an interface containing four more methods that it won't call. And a client class shouldn't be forced to implement methods from an interface that it doesn't use.

At the end, big interfaces, like our given IUserPreferences, may end up grouping not-so-closely-related methods (which would also violate the Single Responsibility Principle.) And the solution to this and the previous problem is to split them up in smaller, consistent interfaces.

Taking a closer look, we can see that IUserPreferences was containing methods related to the user name, the user security credentials, and the user favorite links. Hence, splitting this original interface in three, smaller, consistent interfaces would make sense:

**IUserNameStore.java:**

**public interface IUserNameStore {**

**public** String getFirstName();

**public** String getMiddleName();

**public** String getLastName();

}

**IUserSecurityCredentialsStore.java:**

**public interface IUserSecurityCredentialsStore {**

**public** String getPassword();

**public** String[] getSecurityQuestionAndAnswer();

}

**IUserFavoritesStore.java:**

**public interface** IUserFavoritesStore {

**public** String getFavoriteURL();

**public** String[] getBookmarkList();

}

In our current case, we're interested in name-related preferences, so we only need a new class that would implement IUserNameStore with default values:

**DefaultUserNameStore.java:**

**public class DefaultUserNameStore implements IUserNameStore {**

@Override

**public** String getFirstName() {

**return "John"**;

}

@Override

**public** String getMiddleName() {

**return "X."**;

}

@Override

**public** String getLastName() {

**return "Doe"**;

}

}

Our default header banner would now use this class only:

**DefaultHeaderBanner.java:**

**public class DefaultHeaderBanner {**

**public void** displayFullUserName() {

IUserNameStore defaultUserNameStore = **new** DefaultUserNameStore();

String fullUserName =

defaultUserNameStore.getFirstName() + **" "** +

defaultUserNameStore.getMiddleName() + **" "** +

defaultUserNameStore.getLastName();

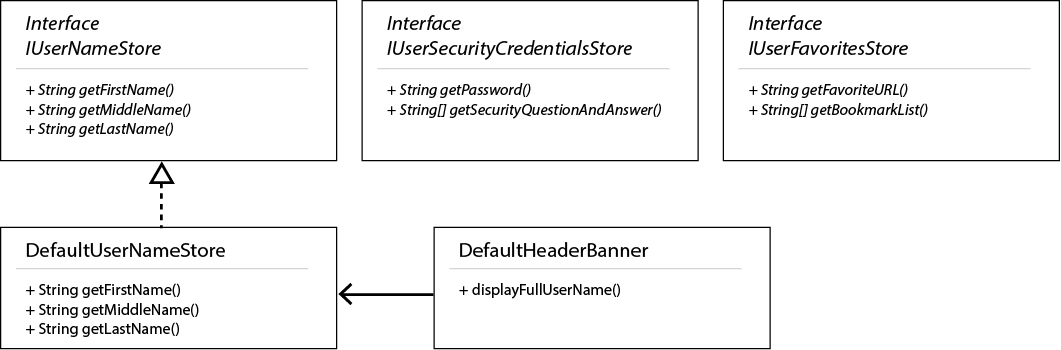
System.***out***.println(**"--------------------------------"**);

System.***out***.println(**"User --> "** + fullUserName);

System.***out***.println(**"--------------------------------"**);

}

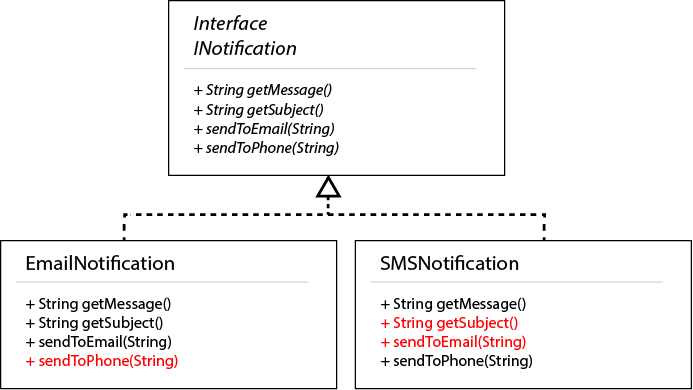
}



## 08 Interface Segregation Principle

According to the Interface Segregation Principle, no client class (a class that “uses” another class) should be forced to depend on methods of an interface that it doesn't need.

It is a good idea to create an interface to group all kinds of notifications, so changing the way we notify users doesn't mean we need to change all code that uses notifications (Open for Extension, Closed for Modification.) However, the current implementation of INotification is forcing the classes that implement it to implement methods they cannot use – an EmailNotification that implements INotification would expose 'sendToPhone', even if the method won't work, which is deceiving and supposes an even bigger problem.



There are many solutions for this scenario, all of them revolving around not forcing subclasses to implement methods they don't need. Converting INotification into a simpler and more generic interface is a potential solution of the problem. Since any kind of notification can be sent, including a 'send' method in the interface makes sense, as it should be common and necessary for all possible implementing class. 'getMessage' can also considered common to any kind of notification sent, so it could have been included as well.

This is what the new architecture would look like:

**INotification.java:**

**public interface INotification {**

**public void** send();

}

**EmailNotification.java:**

**public class EmailNotification implements INotification {**

**private** String **message**;

**private** String **subject**;

**private** String **emailAddress**;

**public** EmailNotification(String message, String subject, String emailAddress) {

**this**.**message** = message;

**this**.**subject** = subject;

**this**.**emailAddress** = emailAddress;

}

@Override

**public void** send() {

**this**.sendToEmail(**this**.**emailAddress**, **this**.**message**, **this**.**subject**);

}

**private void** sendToEmail(String emailAddress, String message, String subject) {

*/\* Code to send email would go here \*/*

System.***out***.println(**"Sending email..."**);

}

}

**SMSNotification.java:**

**public class SMSNotification implements INotification {**

**private** String **message**;

**private** String **phoneNumber**;

**public** SMSNotification(String message, String phoneNumber) {

**this**.**message** = message;

**this**.**phoneNumber** = phoneNumber;

}

@Override

**public void** send() {

**this**.sendToPhone(**this**.**phoneNumber**, **this**.**message**);

}

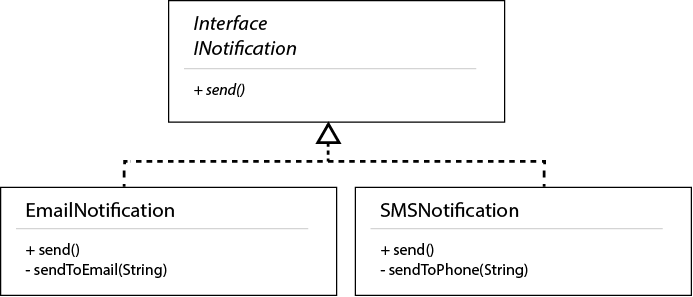
**private void** sendToPhone(String phoneNumber, String message) {

*/\* Code to send SMS text message would go here \*/*

System.***out***.println(**"Sending text message to phone..."**);

}

}



## 09 Dependency Inversion Principle

The main problem with our initially proposed architecture is that it makes our high-level DataLoader class depend on the specific details of the low-level MySQLDatabaseConnection. This implementation isn't prepared for changes, as trying to use a different database engine would imply modifying all classes (also violating the Open/Closed principle.) So all our these should depend only on abstractions.

Moreover, our DataLoader class depends on the underlying used database connection to get such data, and this dependency must be explicit. In the Spring framework we could use dependency injection to achieve this. In this simpler example, an easy way to achieve similar results is to provide this dependency as a mandatory parameter to include in the class constructor. What we want to avoid at all costs is to only introduce this dependency inside the body of a method, as it would become hard to track.

Taking all this into account, ther result would be as follows:

* Create an abstract Interface representing all possible database connections.
* Make specific database connections implement this interface.
* Make the original DataLoader class depend on this abstracted interface, and be sure to make this dependency explicit (in this case, including it in the constructor.)

**IDatabaseConnection.java:**

**public interface IDatabaseConnection {**

**public** String getData();

}

**MySQLDatabaseConnection.java:**

**public class MySQLDatabaseConnection implements IDatabaseConnection {**

**public** MySQLDatabaseConnection() {

*// A MySQL connection would be established here*

}

**public** String getData() {

*// Sample data, hardcoded*

**return "-Washington:1789 -Adams:1797 -Jefferson:1801"**;

}

}

**PostgreSQLDatabaseConnection.java:**

**public class PostgreSQLDatabaseConnection implements IDatabaseConnection {**

**public** PostgreSQLDatabaseConnection() {

*// A PostgreSQL connection would be established here*

}

**public** String getData() {

*// Sample data, hardcoded*

**return "-Washington:1789 -Adams:1797 -Jefferson:1801"**;

}

}

**DataLoader.java:**

**public class** DataLoader {

**private** IDatabaseConnection **databaseConnection**;

*/\*\**

*\* Constructor*

*\** ***@param connection*** *database connection needed to create this object (now an explicit dependency, not tied to a specific database engine)*

*\*/*

**public** DataLoader(IDatabaseConnection connection) {

**this**.**databaseConnection** = connection;

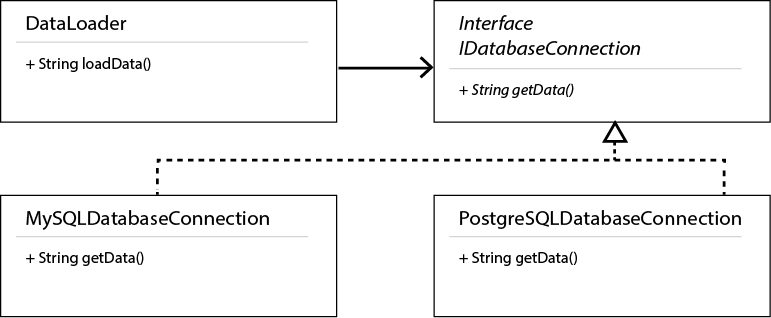
}

**public** String loadData() {

**return this**.**databaseConnection**.getData();

}

}



## 10 Dependency Inversion Principle

The starting point to create a clean architecture is to divide our original code into the three main layers used in this application, while grouping consistent methods together in:

* A data access layer
* A service layer
* A presentation (GUI) layer

Additionally, our original “Investment” class can be considered a data “model” object.

While breaking down the original code in three specific blocks would work with the Single Responsibility Principle, that wouldn't work well with the Dependency Inversion Principle, as we would be making our high-level components (our GUI, RecommendedInvestmentDisplayGUI) depend on specific implementations (for example, a HardcodedInvestmentLoader in the data access layer, and a BalancedRiskProfitRecommendationService in the service layer.)

To follow the Dependency Inversion Principle, we have to create abstract interfaces for every one of the layers:

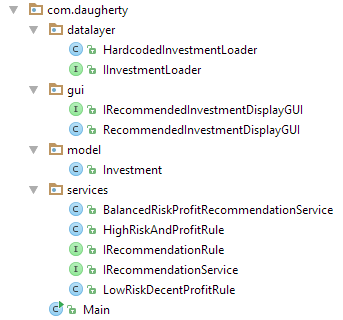
* An interface for the data access layer, IInvestmentLoader.
* An interface for the service layer, IRecommendationService.
* And ideally, an interface for the presentation layer, IRecommendedInvestmentDisplayGUI.

This way, all classes depend on abstractions, easy to modify with minimum impact. Then, at least an implementation for every one of these interfaces has to be provided:

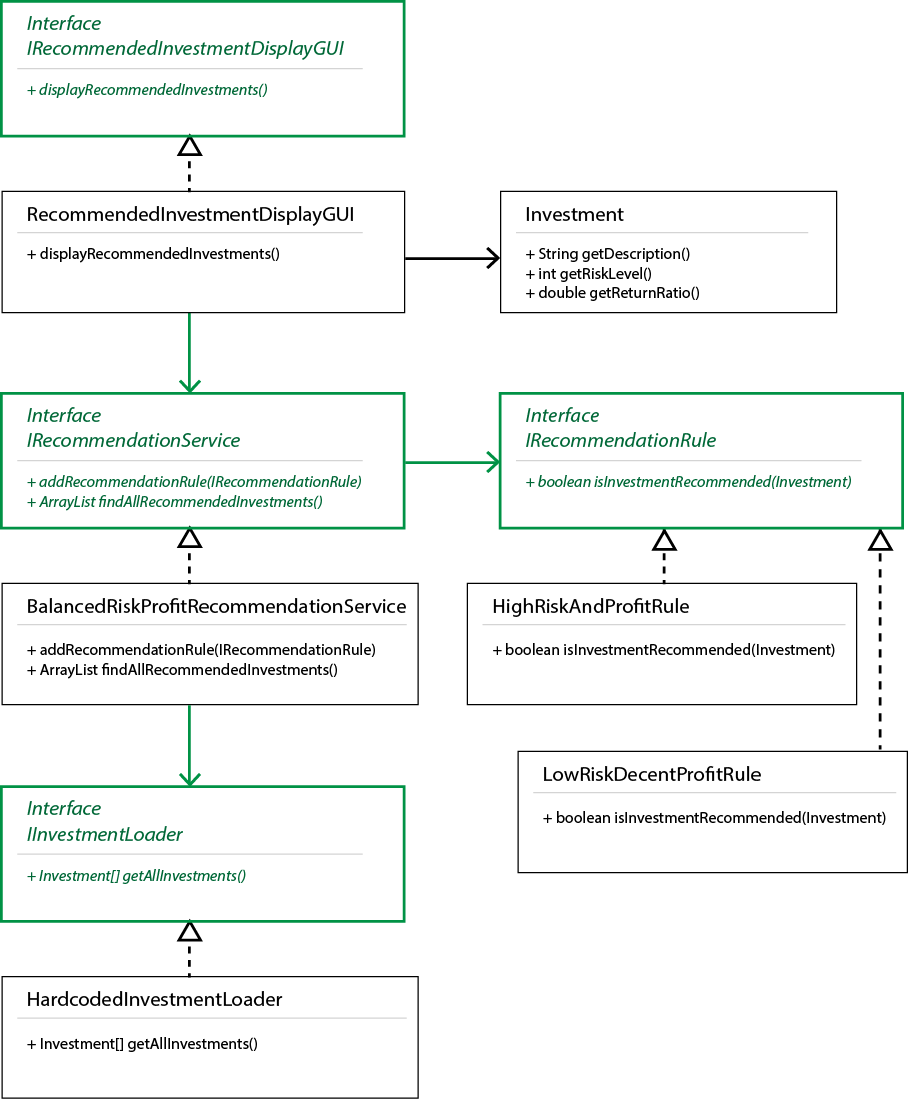
* A HardcodedInvestmentLoader that implements IInvestmentLoader.
* A BalancedRiskProfitRecommendationService that implements IRecommendationService.
* And a RecommendedInvestmentDisplayGUI that implements the IRecommendedInvestmentDisplayGUI.

Finally, to follow the Open/Close principle as much as possible, we should split the logical rules used to define what investments are worth recommending, as we already did in a previous example. We can include every one of the existing rules in its own, clearly named class, and make them all implement a common IRecommendationRule interface. This way, new rules could easily be added without impacting anything more than the service itself, and existing rules can be modified only impacting each specific rule.

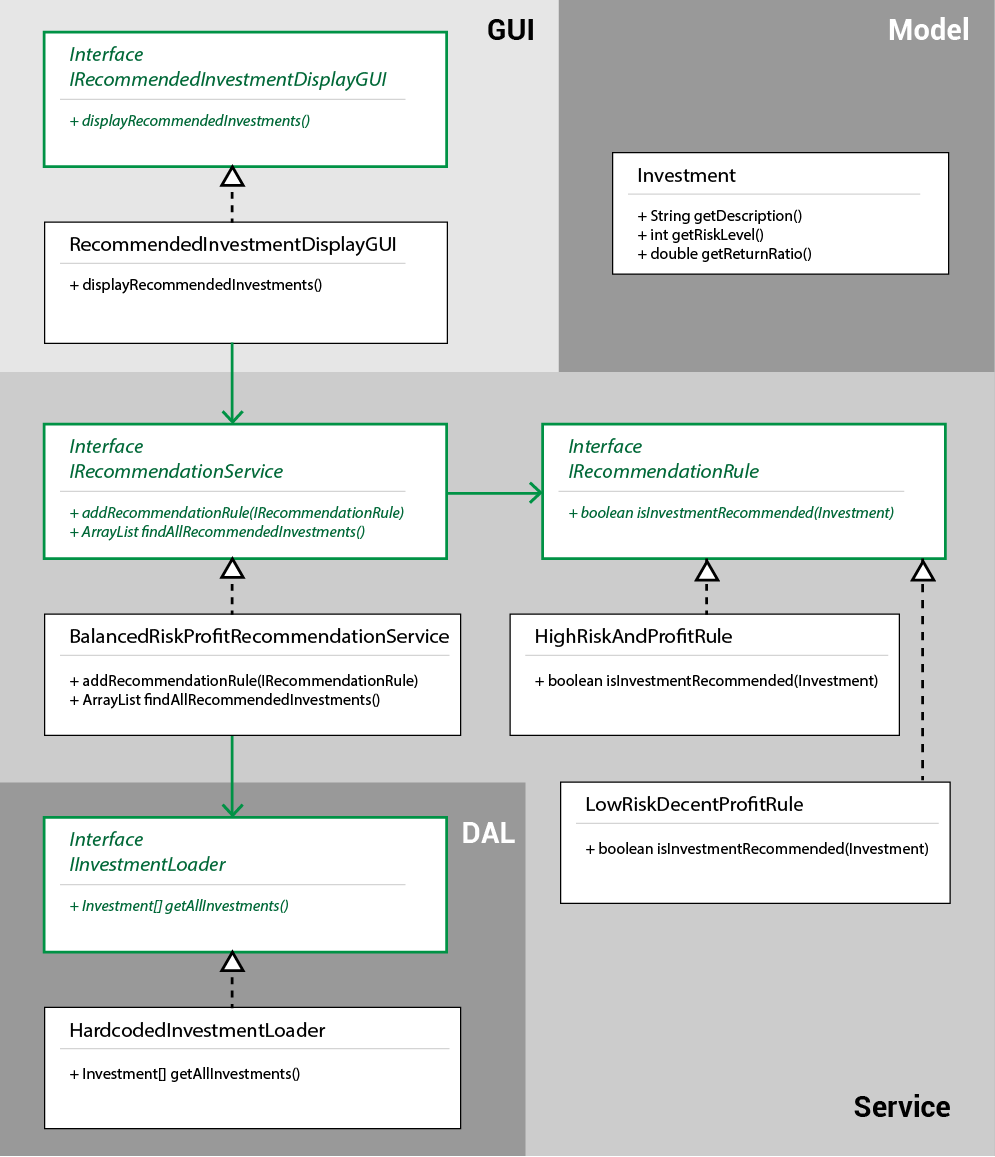
This is what the new packages and files would look like:



And this diagram of the new system architecture, highlighting how we depend on abstractions in all levels, while providing concrete implementations for them:



This version of the previous diagram displays the different layers and the classes that implement them:



These are the details of every one of the final classes:

**IInvestmentLoader.java:**

**package com.daugherty.datalayer;**

**import** com.daugherty.model.Investment;

**public interface** IInvestmentLoader {

**public** Investment[] getAllInvestments();

}

**HardCodedInvestmentLoader.java:**

**package com.daugherty.datalayer;**

**import** com.daugherty.model.Investment;

**public class** HardcodedInvestmentLoader **implements** IInvestmentLoader {

**public** Investment[] getAllInvestments() {

*// Returns a hardcoded list of possible investments (other implementations may connect to databases for the same purpose)*

**return new** Investment[] {

**new** Investment(**"Risky business"**, 100, 200),

**new** Investment(**"Startup"**, 50, 175),

**new** Investment(**"Savings account"**, 1, 2),

**new** Investment(**"Stocks account"**, 8, 7),

**new** Investment(**"High-frequency trading"**, 40, 20)

};

}

}

**IRecommendationService.java:**

**package com.daugherty.services;**

**import** com.daugherty.model.Investment;

**import** java.util.ArrayList;

**public interface** IRecommendationService {

**public void** addRecommendationRule(IRecommendationRule recommendationRule);

**public** ArrayList<Investment> findAllRecommendedInvestments();

}

**BalancedRiskProfitRecommendationService.java:**

**package com.daugherty.services;**

**import** com.daugherty.datalayer.HardcodedInvestmentLoader;

**import** com.daugherty.datalayer.IInvestmentLoader;

**import** com.daugherty.model.Investment;

**import** java.util.ArrayList;

**public class** BalancedRiskProfitRecommendationService **implements** IRecommendationService {

**private** ArrayList<IRecommendationRule> **ruleList**;

**private** IInvestmentLoader **investmentLoader**;

**public** BalancedRiskProfitRecommendationService() {

**this**.**investmentLoader** = **new** HardcodedInvestmentLoader();

**this**.**ruleList** = **new** ArrayList<IRecommendationRule>();

**this**.**ruleList**.add(**new** HighRiskAndProfitRule());

**this**.**ruleList**.add(**new** LowRiskDecentProfitRule());

}

@Override

**public void** addRecommendationRule(IRecommendationRule recommendationRule) {

**this**.**ruleList**.add(recommendationRule);

}

@Override

**public** ArrayList<Investment> findAllRecommendedInvestments() {

Investment[] allInvestments = **this**.**investmentLoader**.getAllInvestments();

ArrayList<Investment> recommendedInvestments = **new** ArrayList<Investment>();

**for**(**int** i = 0; i < allInvestments.**length**; i++) {

Investment currentInvestment = allInvestments[i];

**if**(**this**.isInvestmentRecommended(currentInvestment)) {

recommendedInvestments.add(currentInvestment);

}

}

**return** recommendedInvestments;

}

**private boolean** isInvestmentRecommended(Investment investment) {

**for**(**int** i = 0; i < **this**.**ruleList**.size(); i++) {

IRecommendationRule currentRule = **this**.**ruleList**.get(i);

**if**(currentRule.isInvestmentRecommended(investment)) {

**return true**;

}

}

*// No match with any rule means it isn't recommended*

**return false**;

}

}

**IRecommendationRule.java:**

**package com.daugherty.services;**

**import** com.daugherty.model.Investment;

**public interface** IRecommendationRule {

**public boolean** isInvestmentRecommended(Investment investment);

}

**HighRiskAndProfitRule.java:**

**package com.daugherty.services;**

**import** com.daugherty.model.Investment;

**public class** HighRiskAndProfitRule **implements** IRecommendationRule {

@Override

**public boolean** isInvestmentRecommended(Investment investment) {

**return** (investment.getRiskLevel() < 75 && investment.getReturnRatio() > 150);

}

}

**LowRiskDecentProfitRule.java:**

**package com.daugherty.services;**

**import** com.daugherty.model.Investment;

**public class** LowRiskDecentProfitRule **implements** IRecommendationRule {

@Override

**public boolean** isInvestmentRecommended(Investment investment) {

**return** (investment.getRiskLevel() < 10 && investment.getReturnRatio() > 6);

}

}

**IRecommendedInvestmentDisplayGUI.java:**

**package com.daugherty.gui;**

**public interface** IRecommendedInvestmentDisplayGUI {

**public void** displayRecommendedInvestments();

}

**RecommendedInvestmentDisplayGUI.java:**

**package com.daugherty.gui;**

**import** com.daugherty.model.Investment;

**import** com.daugherty.services.BalancedRiskProfitRecommendationService;

**import** com.daugherty.services.IRecommendationService;

**import** java.util.ArrayList;

**public class** RecommendedInvestmentDisplayGUI **implements** IRecommendedInvestmentDisplayGUI {

**private** IRecommendationService **recommendationService**;

**public** RecommendedInvestmentDisplayGUI() {

**this**.**recommendationService** = **new** BalancedRiskProfitRecommendationService();

}

**public void** displayRecommendedInvestments() {

*// (Other implementations may not echo the results to the console in plain text, or may present them with a different look)*

ArrayList<Investment> recommendedInvestments = **this**.**recommendationService**.findAllRecommendedInvestments();

System.***out***.println(**"Recommended investments"**);

System.***out***.println(**"-----------------------"**);

**for**(**int** i = 0; i < recommendedInvestments.size(); i++) {

System.***out***.println(recommendedInvestments.get(i).getDescription());

}

}

}

## 11 Don't Repeat Yourself

For both logical rules, we're comparing whether the current balance is higher than a given amount. Chances are we would be adding more rules in the future based on the same principle. So both rules could be moved to a single new class named something like **InterestRuleOverAmount.java**:

@Override

**public boolean** isRuleApplicable(**int** yearOfCreation, **double** balance) {

**return** (balance > **this**.**thresholdAmount**);

}

Where thresholdAmount is the amount to surpass, provided as a parameter to the constructor. This would be used as follows in **InterestCalculator.java**:

**public** InterestCalculator() {

**this**.**interestRules** = **new** ArrayList<>();

**this**.addRule(**new** InterestRuleOverAmount(1000000, 2.1));

**this**.addRule(**new** InterestRuleOverAmount(500000, 1.4));

}

Even more DRY optimization principles could be applied here, like getting rid of the “magic numbers” used as constant numbers in these rules – for both tresholds and interest rates - and extracted as constants (which could then be moved to their own rules or classes as well.)

## 12 Don't Repeat Yourself

There are two main elements that can be optimized in this class.

First, the function used to convert from Fahrenheit to Celsius is duplicated in two places. If, for example, a bug were discovered in that function, or it had to be replaced by a more precise calculation, it would have to be fixed in two spots. Hence, it is better to consolidate it under a single method.

Second, there are quite a few String and numeric values hardcoded in the logic – and several of them are just reused through the code. Extracting them to constant values would make maintaining them an easier task.

Here's a proposed refactoring of **TemperatureAlertService.java:**

**public class TemperatureAlertService {**

*// Extracts constant hardcoded values (Strings and numbers) to reusable constants*

**public static final** String ***TO\_ALERT\_EMAIL\_ADDRESS*** = **"alert.service@example.com"**;

**public static final** String ***TEMPERATURE\_ALERT\_MESSAGE\_PREFIX*** = **"Temperature alert "**;

**public static final double *HIGH\_TEMPERATURE\_CELSIUS\_THRESHOLD*** = 100;

**public static final double *LOW\_TEMPERATURE\_CELSIUS\_THRESHOLD*** = 0;

**public boolean** checkIsHighTemperature(**double** temperatureInFahrenheit) {

**double** temperatureInCelsius = convertToCelsius(temperatureInFahrenheit);

**if**(temperatureInCelsius > ***HIGH\_TEMPERATURE\_CELSIUS\_THRESHOLD***) {

**this**.sendAlert(***TEMPERATURE\_ALERT\_MESSAGE\_PREFIX*** + temperatureInCelsius, ***TO\_ALERT\_EMAIL\_ADDRESS***);

**return true**;

} **else** {

**return false**;

}

}

**public boolean** checkIsLowTemperature(**double** temperatureInFahrenheit) {

**double** temperatureInCelsius = (convertToCelsius(temperatureInFahrenheit));

**if**(temperatureInCelsius < ***LOW\_TEMPERATURE\_CELSIUS\_THRESHOLD***) {

**this**.sendAlert(***TEMPERATURE\_ALERT\_MESSAGE\_PREFIX*** + temperatureInCelsius, ***TO\_ALERT\_EMAIL\_ADDRESS***);

**return true**;

} **else** {

**return false**;

}

}

*// Extracts a re-used mathematical operation to its own function*

**private double** convertToCelsius(**double** temperatureInFahrenheit) {

**return** (temperatureInFahrenheit - 32) \* (5 / 9.0);

}

**private void** sendAlert(String message, String toAddress) {

*// It would use some email service to send the alert*

System.***out***.println(**"Sending alert..."**);

System.***out***.println(**"Message: "** + message);

System.***out***.println(**"To Address: "** + toAddress);

}

}