Classes - 10

MOHAMED ABDUL

CS 5800

Disclaimer

- Some of the concepts, and examples are derived from the clean code textbook.

Agenda

- Class Organization
- Classes Should be Small!
- Single Responsibility Principle
- Cohesion
- Maintaining Cohesion Results in Many Small Classes
- Organizing for Change
- Isolating from Change
- Summary

Class Organization

- **List the variables at the top**
  - First, list of public static variables.
  - Second, list of private static variables.
  - Third, list of instance variables.
- **Public functions should follow the variables**
  - Any utilities functions should follow right after the caller.
- **Encapsulation**
  - Try to keep variables, and utilities as private
    - Sometimes, its necessary to keep them as protected for testing purpose.

```
class DateUtil {

  public static final CST_TIMEZONE = "America/Chicago";
  public static final PST_TIMEZONE = "America/Seattle";
  public static final EST_TIMEZONE = "America/New_York";
  private static final DEFAULT_DATE_FORMAT_YYYY_MM_DD = "yyyy-MM-dd";
  private String dateformat;

  DateUtil(String dateFormat) {
    this.dateFormat = dateFormat;
  }

  public String getCurrentDate(String timeZone) throws DateException {
    if (!isValidTimeZone(timeZone)) {
        throw new DateException("Not a Valid Time Zone");
    }
    ZonedDateTime date = ZonedDateTime.now(ZoneId.of(timeZone));
    DateTimeFormatter formatter = DateTimeFormatter.ofPattern(getDateFormat());
    return date.format(formatter);
  }

  private boolean isValidTimeZone(String timeZone) {
    return CST_TIMEZONE.equals(timeZone)
        || PST_TIMEZONE.equals(timeZone)
        || EST_TIMEZONE.equals(timeZone);
  }

  private String getDateFormat() {
    if (dateFormat == null || dateFormat.isEmpty()) {
        return DEFAULT_DATE_FORMAT_YYYY_MM_DD;
    }
    return dateFormat;
  }

}
```

Class Organization Example

An util class which generates current date based on given time zone.

Classes Should be small!

- **Rule of a class**
  - First, the class should be small
  - Second, the class should be smaller than that
- **How do we measure the class?**
  - We measure function by number of physical lines are there.
  - But, for class we measure by responsibilities.

Single Responsibility Principle (SRP)

- SRP states that class should have only one reason to change.
- SRP states that class should have only one responsibility.
- Having single responsibility, makes the class easily maintainable and testable.
- If the class is coupled with too many responsibility, its hard to make changes without affecting other functionalities within the class.
- Cleaning up, and refactoring is important as programming tasks.

SRP - Example

- An application util class which has functionalities to fetch weather information, stock information, and currency information.

```
class AppilcationUtil {
 public WeatherData getWeatherDataFor(City city);
 public List<WeatherData> getWeatherDataForAWeek(City city, Date fromDate);
 public WeatherData getWeatherDataForDay(City city, Date date);
 public WeatherData getWeatherDataForLocation(Location location);
 public WeatherData getWeatherDataForCountry(String country);
 public WeatherData getWeatherDataForCoordinates(double latitude, double longitude);
 public WeatherDate getWeatherDataForRange(City city, Date startDate, Date endDate);
 public List<WeatherData> getWeatherDataForDates(City city, List<Date> dates);
 public List<WeatherDate> getWeatherDateForMonthAndYear(City city, String year, String month);
 public StockData getStockDataFor(String stockName);
 public StockData getStockDataForDate(String stockName, Date date);
 public List<StockData> getStockForAWeek(String stockName, Date fromDate);
 public List<StockData> getStockForDates(String stockName, List<Data> dates);
 public List<StockData> getStockForMonth(String stockName, Date month);
 public List<StockData> getStockForAWeekWithBound(String stockName, Bounds prices, Date fromDate);
 public List<StockData> getStockForDatesWithBound(String stockName, Bounds prices, List<Data> dates);
 public List<StockData> getStockForMonthWithBound(String stockName, Bounds prices, Date month);
 public CurrencyData getCurrencyRateFor(String currency);
 public CurrencyData getCurrencyRateForDate(String currency, Date date);
 public List<CurrencyData> getCurrencyRateAWeek(String currency, Date fromDate);
 public List<CurrencyData> getCurrencyRateDates(String currency, List<Data> dates);
 public List<CurrencyData> getCurrencyRateMonth(String currency, Date month);
 public List<CurrencyData> getCurrencyRateAWeekWithBound(String currency, Bounds prices, Date fromDate);
 public List<CurrencyData> getCurrencyRateForMonthWithBound(String currency, Bounds prices, Date month);
}
```

```java
class WeatherClient {
 public WeatherData getWeatherDataFor(City city);
 public List<WeatherData> getWeatherDataForAWeek(City city, Date fromDate);
 public WeatherData getWeatherDataForDay(City city, Date date);
 public WeatherData getWeatherDataForLocation(Location location);
 public WeatherData getWeatherDataForCountry(String country);
 public WeatherData getWeatherDataForCoordinates(double latitude, double longitude);
 public WeatherDate getWeatherDataForRange(City city, Date startDate, Date endDate);
 public List<WeatherData> getWeatherDataForDates(City city, List<Date> dates);
 public List<WeatherDate> getWeatherDateForMonthAndYear(City city, String year, String month);
}
```

```java
class StockClient {
 public StockData getStockDataFor(String stockName);
 public StockData getStockDataForDate(String stockName, Date date);
 public List<StockData> getStockForAWeek(String stockName, Date fromDate);
 public List<StockData> getStockForDates(String stockName, List<Data> dates);
 public List<StockData> getStockForMonth(String stockName, Date month);
 public List<StockData> getStockForAWeekWithBound(String stockName, Bounds prices, Date fromDate);
 public List<StockData> getStockForDatesWithBound(String stockName, Bounds prices, List<Data> dates);
 public List<StockData> getStockForMonthWithBound(String stockName, Bounds prices, Date month);
}
```

```java
class CurrencyRateClient {
 public CurrencyData getCurrencyRateFor(String currency);
 public CurrencyData getCurrencyRateForDate(String currency, Date date);
 public List<CurrencyData> getCurrencyRateAWeek(String currency, Date fromDate);
 public List<CurrencyData> getCurrencyRateDates(String currency, List<Data> dates);
 public List<CurrencyData> getCurrencyRateMonth(String currency, Date month);
 public List<CurrencyData> getCurrencyRateAWeekWithBound(String currency, Bounds prices, Date fromDate);
 public List<CurrencyData> getCurrencyRateForMonthWithBound(String currency, Bounds prices, Date month);
}
```

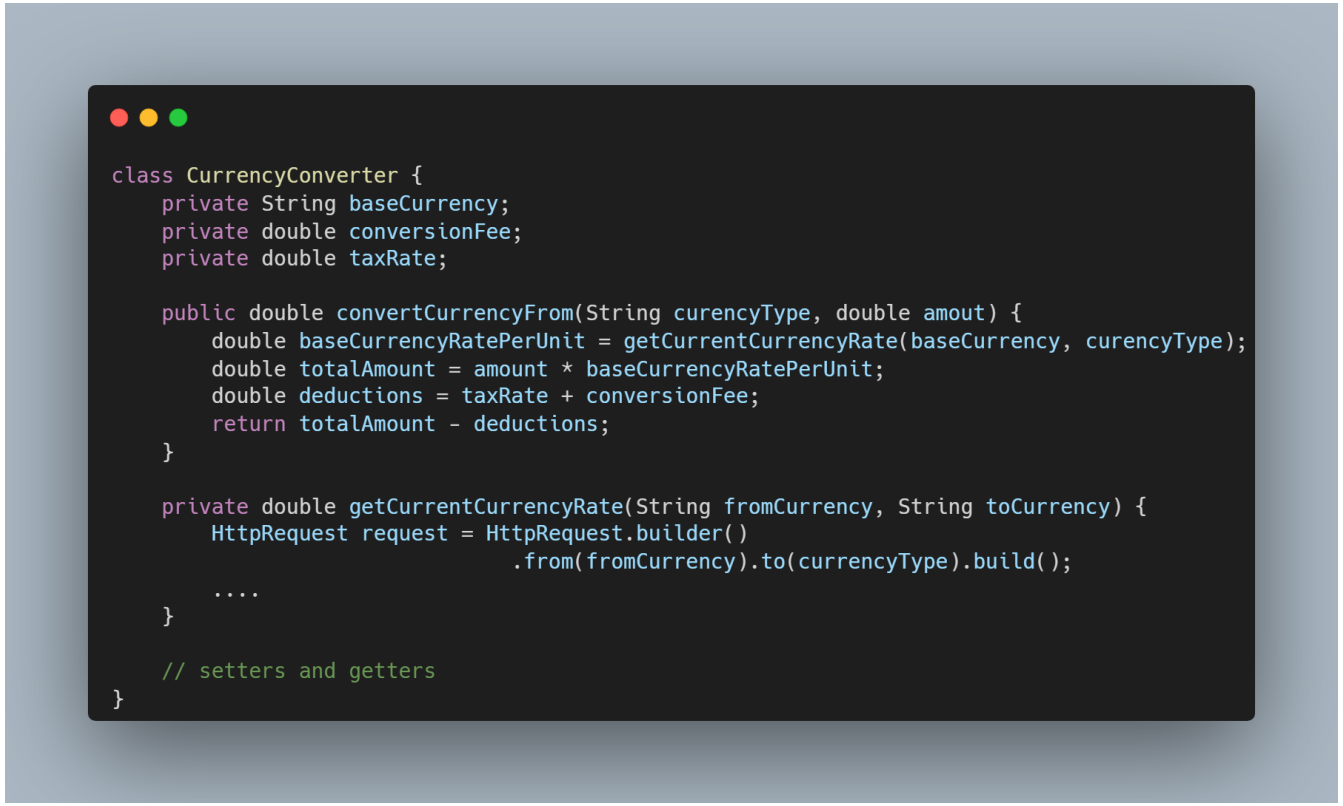
SRP - Example

Each class serves only specific purpose

- WeatherClient
- StockClient
- CurrencyRateClient

Cohesion

- Classes should have small number of instance variables
- Each methods in the class should utilize the instance variables.
- More variable a method manipulates, that method is more cohesive to the class.
- A class with each method using each instance variable is maximally cohesive



```java
class CurrencyConverter {
    private String baseCurrency;
    private double conversionFee;
    private double taxRate;

    public double convertCurrencyFrom(String curencyType, double amout) {
        double baseCurrencyRatePerUnit = getCurrentCurrencyRate(baseCurrency, curencyType);
        double totalAmount = amount * baseCurrencyRatePerUnit;
        double deductions = taxRate + conversionFee;
        return totalAmount - deductions;
    }

    private double getCurrentCurrencyRate(String fromCurrency, String toCurrency) {
        HttpRequest request = HttpRequest.builder()
                              .from(fromCurrency).to(currencyType).build();
        ....
    }

    // setters and getters
}
```

Cohesion

Currency converter

Maintaining Cohesion Results in Many Small Classes

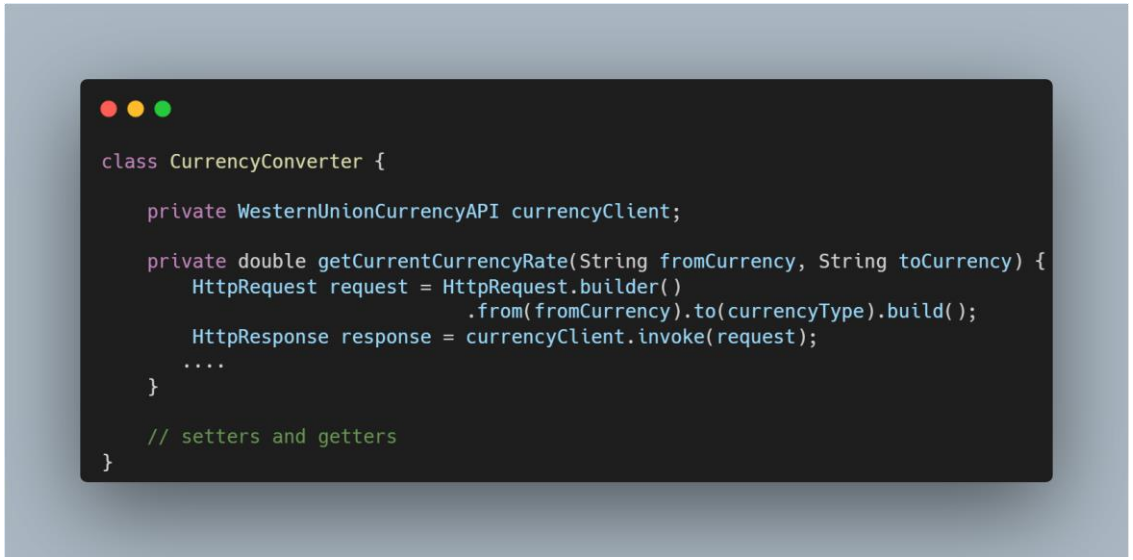
- When a class loses a cohesion and split them.
- Breaking a large function into many smaller functions often gives us the opportunity to split several smaller classes out as well.
- Progressively make changes and test the changes accordingly.

Organizing for Change

- Change is inevitable!
- We should maintain the SRP and as well Open Closed Principle
- New functionalities, should be incorporated by extending the existing class not by modifying them.

Isolating from Change

- Once again, change is inevitable!
- Its hard to change or test something if it uses concrete implementation of a class.
- Instead of concrete implementation, the client code should use Abstract class or Interface.
- **Dependency Inversion Principle (DIP)**
  - Classes should depend upon abstractions, not on concrete details



```
class CurrencyConverter {

    private WesternUnionCurrencyAPI currencyClient;

    private double getCurrentCurrencyRate(String fromCurrency, String toCurrency) {
        HttpRequest request = HttpRequest.builder()
                              .from(fromCurrency).to(currencyType).build();
        HttpResponse response = currencyClient.invoke(request);
        ....
    }

    // setters and getters
}
```

Isolating from Change

- Every API call to **WesternUnionCurrencyAPI** gets charged.
- Every API call returns different rates as the currency rates fluctuates, and so its hard to test!

Isolating from Change

```
public interface CurrencyClient {
    HttpResponse<?> invoke(HttpRequest request);
}
```

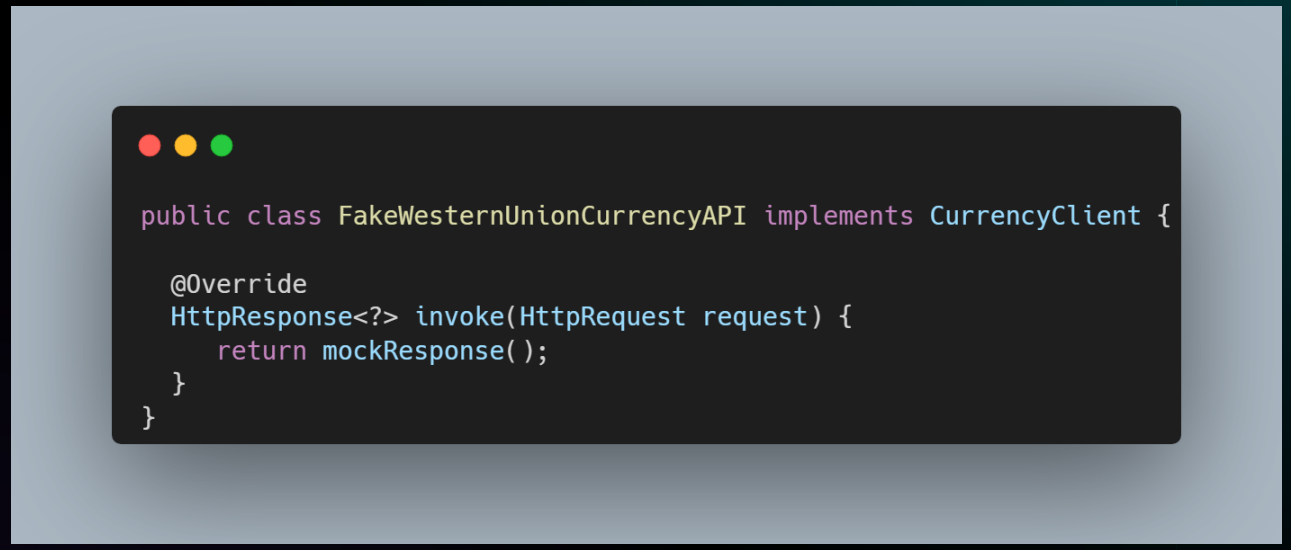
```
public class WesternUnionCurrencyAPI implements CurrencyClient {

  @Override
  HttpResponse<?> invoke(HttpRequest request) {
     ....
  }
}
```

Isolating from Change

```
class CurrencyConverter {

    private CurrencyClient currencyClient;

    public CurrencyConverter(CurrencyClient currencyClient) {
      this.currencyClient = currencyClient;
    }

    private double getCurrentCurrencyRate(String fromCurrency, String toCurrency) {
        HttpRequest request = HttpRequest.builder()
                                  .from(fromCurrency).to(currencyType).build();
        HttpResponse response = currencyClient.invoke(request);
       ....
    }

    // setters and getters
}
```

Isolating from Change

A separate implementation of CurrencyClient API specific to test CurrencyConverter.



```
public class FakeWesternUnionCurrencyAPI implements CurrencyClient {

  @Override
  HttpResponse<?> invoke(HttpRequest request) {
     return mockResponse();
  }
}
```

Summary

- Measure class by responsibility.
- **SRP** – Every class should have single responsibility.
- Analyze class for cohesiveness, if it lacks then split it.
- **Dependency Inversion Principle** – Client should not use concrete class and use interface or Abstract class.

Thank you