

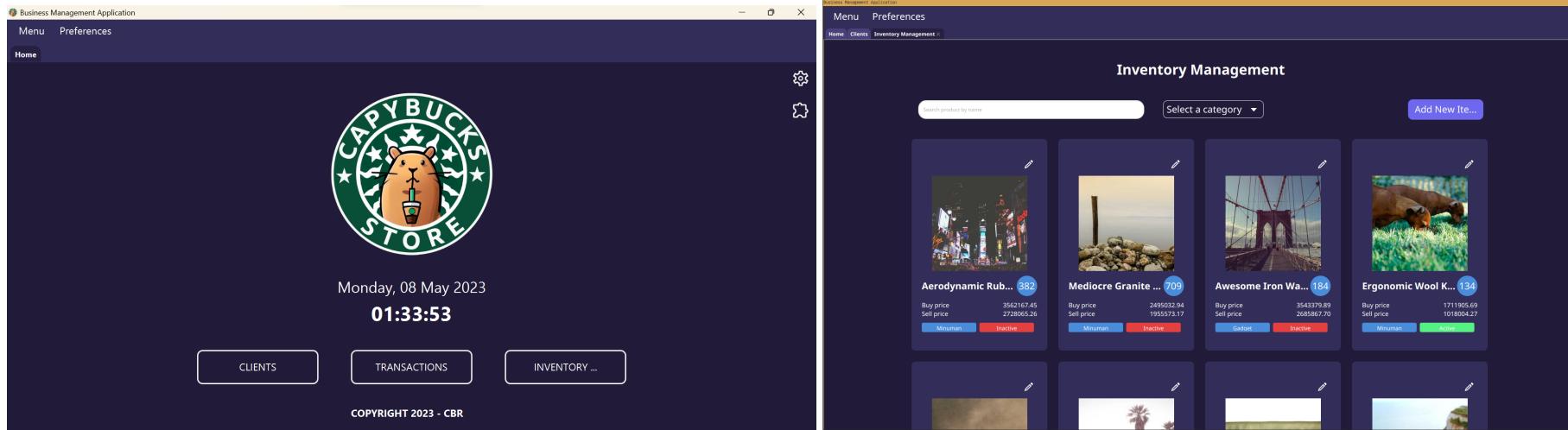
Kode Kelompok : CBR

1. 13516055 / Nathaniel Evan Gunawan
2. 13521044 / Rachel Gabriela Chen
3. 13521046 / Jeffrey Chow
4. 13521074 / Eugene Yap Jin Quan
5. 13521094 / Angela Livia Arumsari
6. 13521100 / Alexander Jason

Asisten Pembimbing : 13519007 / Muhammad Tito Prakasa

1. Deskripsi Umum Aplikasi

Aplikasi CBR - Business Management App adalah aplikasi sistem penjualan (*point-of-sales/POS*) berbasis *desktop*, dibuat menggunakan bahasa pemrograman Java menggunakan *library* JavaFX untuk bagian antarmuka. Aplikasi ini memiliki fitur-fitur yang umumnya dapat diharapkan ada dalam aplikasi POS, seperti manajemen inventaris dan manajemen transaksi. Selain itu, aplikasi mendukung fitur *membership* untuk memberikan *reward* kepada pelanggan setia; pelanggan dibagi menjadi 3 kategori, yaitu *customers*, *members*, dan VIP, dengan *customer* sebagai tingkat terendah tanpa *reward* dan VIP sebagai tingkat tertinggi dengan *reward* terbanyak. Aplikasi CBR - Business Management APP juga dilengkapi dengan fitur *export* ke PDF, serta menyediakan dukungan *extension/plug-in*, untuk menambahkan fungsionalitas program.



2. Kakas GUI: JavaFX

Library GUI yang dipakai dalam pengerjaan tugas ini adalah JavaFX. JavaFX dirancang untuk menyediakan pengembang Java dengan platform grafis berperforma tinggi yang ringan dan baru. JavaFX dikembangkan oleh Oracle Corporation dan dirilis pertama kali pada tahun 2008 sebagai pengganti dari toolkit GUI sebelumnya yaitu Swing.

Untuk dapat menggunakan JavaFX dilakukan cara berikut.

- JavaFX digunakan dengan melakukan *import*
- Membuat kelas yang meng-*inherit* kelas Application dari package javafx.application
- Melakukan override untuk method start() untuk menampilkan GUI yang telah dibuat
- Menciptakan scene dengan menspesifikasiroot node
- Menampilkan scene tersebut dalam stage (stage.setScene())
- Menjalankan aplikasi dengan metode launch()
- Aplikasi akan berhenti jika memanggil metode Platform.stop() dari package javafx.application; metode ini dipanggil jika pengguna secara eksplisit meminta untuk menutup aplikasi ini.

Komponen utama yang diperlukan untuk membuat GUI dapat muncul dengan baik adalah sebagai berikut.

- a. Stage: komponen yang berupa jendela utama yang menampung semua komponen GUI.
- b. Scene: komponen yang merupakan kontainer untuk komponen-komponen GUI seperti button, label, textfield, dan lain-lain.
- c. Layout Manager: komponen digunakan untuk mengatur tata letak dari komponen-komponen GUI di dalam Scene.
- d. Control: komponen GUI seperti button, label, textfield, checkbox, dan lain-lain.

Selain jenis komponen utama yang dibutuhkan, beberapa komponen yang disediakan oleh JavaFX adalah sebagai berikut.

- a. Label: digunakan untuk menampilkan teks statis.
- b. Button: digunakan untuk membuat tombol.
- c. TextField: digunakan untuk memasukkan teks dari pengguna.
- d. CheckBox: digunakan untuk membuat pilihan ya atau tidak.
- e. RadioButton: digunakan untuk membuat pilihan eksklusif.
- f. ComboBox: digunakan untuk membuat pilihan dalam daftar drop-down.
- g. ListView: digunakan untuk menampilkan daftar item.
- h. TableView: digunakan untuk menampilkan data dalam bentuk tabel.
- i. ScrollPane: digunakan untuk membuat area scrollable.
- j. Image: digunakan untuk menampilkan gambar.
- k.MenuBar: digunakan untuk membuat menu navigasi.
- l. ToolBar: digunakan untuk membuat toolbar dengan tombol-tombol aksi.
- m. Dialog: digunakan untuk menampilkan pesan atau dialog interaktif.

3. Plugin & Class Loader

Main App menyediakan interface Plugin yang memiliki method *void* yang harus di-implement oleh setiap plugin. Pengaturan dan pemuatan plugin pada Main App dilakukan oleh sebuah PluginManager.

PluginManager memiliki atribut `plugins`(List of Plugin) yang menyimpan daftar plugin yang telah dimuat dalam aplikasi. Selain itu, PluginManager memiliki *method*:

- `loadNewPlugin(String jarFile)`

Method ini digunakan untuk melakukan *loading* sebuah plugin baru dengan memanfaatkan *absolute path* dari plugin yang akan di-*load*. Dalam method ini, dilakukan pengecekan apakah plugin tersebut sudah pernah di-*load* atau belum. Jika ya, maka method ini akan *throw* `PluginException`. Jika tidak, maka plugin akan ditambahkan ke daftar plugin dan dipanggil method *load*-nya.

- `loadNewPlugin()`

Method ini digunakan untuk melakukan *loading* plugin-plugin yang pernah di-*load* di aplikasi sebelum aplikasi di-*restart*. Method ini dipanggil di awal *stantiate* aplikasi.

Kedua *method* di atas melakukan *loading* plugin dari sebuah file *jar* dengan memanfaatkan *absolute path* dari jar tersebut.

```

1 public void loadNewPlugin(String jarFile) throws PluginException, MalformedURLException {
2     URLClassLoader classLoader = new URLClassLoader(new URL[] { new File(jarFile).toURI().toURL() },
3             ClassLoader.getSystemClassLoader());
4     pluginServiceLoader = ServiceLoader.load(Plugin.class, classLoader);
5     for (Plugin p : pluginServiceLoader) {
6         boolean exists = plugins.stream()
7             .map(Plugin::getClass)
8             .anyMatch(c → c.getName().equals(p.getClass().getName()));
9         // If an object of the same class already exists, don't add the new object
10        if (!exists) {
11            // system.out.println(p.getClass().getName());
12            plugins.add(p);
13            p.load();
14        } else {
15            // system.out.println(p.getClass().getName());
16            throw new PluginException();
17        }
18    }
19}

```

Pertama, digunakan URLClassLoader untuk me-*load* semua *class* dari jar tersebut. URLClassLoader dimanfaatkan untuk melakukan *dynamic class loading* saat *runtime*. Artinya, *class* yang tidak didefinisikan di *main app* akan ter-*load* dari jar tersebut. Lalu, *method* ini memanfaatkan ServiceLoader untuk me-*stantiate* *class* yang memberikan *service* kepada package com.cbr.app dan mengimplementasikan Plugin. Untuk setiap *plugin* yang di-*load* dari *classLoader*, dipastikan bahwa *plugin* tersebut belum pernah di-*load* sebelumnya. Jika ya, maka akan dilempar PluginException, jika tidak maka plugin akan di-*load*.

Method *load* setiap *plugin* akan menambahkan *field-field* yang perlu pada Main App. Models pada Main App juga menyediakan Map additionalValues yang merupakan meta data dari setiap models sehingga bisa di-*extend* tanpa mengetahui keberadaan meta data tersebut. Field inilah yang digunakan oleh *plugin* untuk memanipulasi Main App dan objek-objek yang berada di dalamnya.

Setiap plugin juga akan memulai sebuah *thread* yang akan meng-*update state* setiap detik untuk memastikan bahwa fitur-fitur dari plugin reaktif terhadap perubahan di aplikasi. Misalnya pada plugin sistem, jika setting diubah untuk mengganti nilai-nilai additional values, maka plugin juga akan meng-*update state* sesuai *value* yang berubah.

4. Class Diagram

Diagram kelas yang dibuat memiliki beberapa struktur utama yaitu Identifiable, DataStorer, Plugin, DataStore, dan Price. Kelas Identifiable di-*extends* oleh kelas-kelas yang memiliki ID dan berfungsi sebagai objek utama dari aplikasi: seperti Product, Customer, Invoice, dan DataList<T>. Dengan melakukan extends, kelas-kelas tadi akan memiliki atribut ID yang diturunkan dari kelas Identifiable. Kemudian, kelas BoughtProduct dan InventoryProduct adalah *extends* dari kelas Product. Kelas Customer di-*extends* menjadi kelas Member karena memiliki beberapa atribut tambahan, kemudian di-*extends* kembali oleh kelas VIP dengan atribut tambahan discount.

Interface DataStorer berfungsi untuk mendefinisikan proses yang berhubungan dengan database. Kelas ini di-implements oleh tiga kelas, yaitu XmlDataStore, JsonDataStore, dan ObjDataStore sesuai dengan tipe penyimpanan data. Kelas DataStore berfungsi sebagai penghubung database dengan Kelas-kelas lain. Kelas Plugin berfungsi sebagai penghubung plugin dengan aplikasi. Interface Price di-implements oleh BasePrice dan PriceDecorator. Selain seluruh hubungan yang ditampilkan pada diagram kelas, setiap kelas juga dapat memanfaatkan composition, yaitu menggunakan kelas lain sebagai atribut.



https://drive.google.com/file/d/1pJxi-76kXYF9Vewiz75y6H-bIN9Jo-rB/view?usp=share_link

5. Konsep OOP

Catatan tambahan: Seluruh file utama terletak pada folder App/src

5.1. Inheritance

Models:

- Kelas Identifiable di-*inherit* oleh berbagai kelas yang menggunakan sebuah *string id* sebagai identifikasi

```
public abstract class Product extends Identifiable implements Serializable {  
    @Getter  
    @Setter
```

- Kelas Customer (yang di-*inherit* oleh menjadi kelas Member, yang di-*inherit* lagi oleh kelas VIP)

```
@Getter  
@Setter  
public class Member extends Customer {  
    @Getter  
    @NotNull  
    protected String name;
```

```
@Getter  
@Setter  
public class VIP extends Member {
```

- Kelas Product (yang di-*inherit* oleh kelas BoughtProduct)

```

@NoArgsConstructor
@Setter
@Getter
public class BoughtProduct extends Product {
    @NotNull private Integer count;
}

```

- Kelas Invoice (yang di-*inherit* oleh kelas TemporaryInvoice dan FixedInvoice).

```

@Getter
@Setter
public class TemporaryInvoice extends Invoice implements Serializable {
    @NotNull protected Map<String, Integer> productFrequencies; // <ProductId, Count>
}

```

```

@Setter
@Getter
public class FixedInvoice extends Invoice {
    @NotNull private List<BoughtProduct> boughtProducts;
}

```

Page :

- Kelas AddItemPage (view/AddItemPage.java) yang di-*inherit* oleh ItemEditorPage (view/ItemEditorPage.java)

```

public class ItemEditorPage extends AddItemPage {
}

```

5.2. Composition

Models:

- Kelas FixedInvoice memiliki banyak kelas Price untuk atribut yang berupa harga. Kelas FixedInvoice juga memiliki kelas BoughtProduct.

```

@Setter
@Getter
public class FixedInvoice extends Invoice {
    @NotNull private List<BoughtProduct> boughtProducts;
    private Price discount;
    private Price usedPoint;
    private static Integer invoiceCount = 0;
    private Price getPoint;
    @NotNull private Map<String, String> additionalCosts;
    private Price grandTotal;
}

```

- Kelas App memiliki atribut berupa kelas DataStore. Kelas DataStore banyak memiliki kelas DataList.

```

public class App extends Application {
    @Getter
    @Setter
    private static Datastore datastore = new DataStore(mode:"JSON", folder:"assets/data/json");
}

```

```

public interface DataStorer {
    public DataList<Customer> loadClients();
    public DataList<InventoryProduct> loadInventory();
    public DataList<FixedInvoice> loadInvoices();
    public DataList<TemporaryInvoice> loadTemporaryInvoices();
}

```

Page :

- Kelas MainView memiliki kelas TransactionPage. Kelas TransactionPage memiliki kelas TransactionProductCardList.

```
public class MainView extends VBox {  
    private HomePage homePage;  
    @Getter  
    private SettingsPage settingsPage;  
    private ClientsPage clientsPage;  
    @Getter  
    private TransactionPage transactionPage;  
    private InventoryPage inventoryPage;
```

```
public class TransactionPage extends StackPane {  
    private HBox container;  
    @Getter  
    private HBox grandTotalContainer;  
    private TransactionProductCardList transactionProductCardList;  
    private TransactionInvoiceCardList transactionInvoiceCardList;  
    @Getter  
    private TemporaryInvoice temporaryInvoice;  
    @Setter  
    private Double grandTotal = 0.0;  
    private TitleDropdown temporaryInvoiceDropdown;  
    private TitleDropdown customerDropdown;  
    @Getter  
    private List<InventoryProduct> productList;  
    private Label grandTotalNumber;  
    private List<Member> membersVipsList;  
    private double discount;  
    private List<TemporaryInvoice> temporaryInvoiceList;  
    @Getter  
    private double managementContainerWidth;  
    @Getter  
    private VBox additionalCostsContainer;  
    private AdditionalCostCard discountContainer;
```

- Kelas App memiliki

5.3. Interface

- Interface Price yang diimplementasikan oleh kelas BasePrice dan kelas PriceDecorator.

```

@JsonSerialize(as=BasePrice.class)
@JsonDeserialize(as=BasePrice.class)
public interface Price {
    public Double getValue();
    public String toString();
}

```

- Interface DataStorer yang diimplementasikan oleh kelas XmlDataStore, ObjDataStore, dan JsonDataStore.

```

public interface DataStorer {
    public DataList<Customer> loadClients();
    public DataList<InventoryProduct> loadInventory();
    public DataList<FixedInvoice> loadInvoices();
    public DataList<TemporaryInvoice> loadTemporaryInvoices();

    public<T extends Serializable> List<T> loadAdditionalData(String dataName, Class<T> clazz);
    public void storeClients(DataList<Customer> records);
    public void storeInventory(DataList<InventoryProduct> records);
    public void storeInvoices(DataList<FixedInvoice> records);
    public void storeTemporaryInvoices(DataList<TemporaryInvoice> records);

    public<T extends Serializable> void storeAdditionalData(List<T> records, String dataName);
}

```

5.4. Method Overriding dan Method Overloading

Models:

- Method loadNewPlugin pada kelas PluginManager menerapkan method overloading untuk fungsi loadNewPlugin. Fungsi loadNewPlugin pertama memiliki signature parameter berjumlah satu bertipe String, sedangkan loadNewPlugin kedua memiliki signature tanpa parameter.

```

public void loadNewPlugin(String jarFile) throws PluginException, MalformedURLException {
    URLClassLoader classLoader = new URLClassLoader(new URL[] { new File(jarFile).toURI().toURL() },
        classLoader.getSystemClassLoader());
}

```

```

public void loadNewPlugin() throws MalformedURLException, PluginException {
    // system.out.println("this is load new plugin");
    int i = 0;
    for (String jarFile : AppSettings.getInstance().getPlugins()) {

```

- Kelas ExportPDF melakukan method overloading pada fungsi init dan exportPDF. Hal ini dilakukan karena fungsi export PDF memiliki perilaku yang sama, namun dapat dilakukan untuk dua tipe data yang berbeda, yaitu FixedInvoice dan List<FixedInvoice>.

```

    public void init(Button buttonExport, FixedInvoice invoice) {
        // create a new FileChooser dialog
        FileChooser fileChooser = new FileChooser();
        fileChooser.setTitle("Export Invoice");
        fileChooser.getExtensionFilters().add(new FileChooser.ExtensionFilter("PDF Files", "*.pdf"));

```

```

    public void init(MenuItem menuExport) {
        // create a new FileChooser dialog
        FileChooser fileChooser = new FileChooser();
        fileChooser.setTitle("Export Statement");
        fileChooser.getExtensionFilters().add(new FileChooser.ExtensionFilter("PDF Files", "*.pdf"));

```

```

    public static void exportPDF(String dest, FixedInvoice invoice) throws Exception {
        PdfDocument pdfDoc = new PdfDocument(new PdfWriter(dest));
        Document doc = new Document(pdfDoc, PageSize.LETTER);
        doc.setMargins(24, 24, 24, 24);

```

```
public static void exportPDF(String dest, List<FixedInvoice> invoices) throws Exception{
    PdfDocument pdfDoc = new PdfDocument(new PdfWriter(dest));
    Document doc = new Document(pdfDoc, PageSize.LETTER);
    doc.setMargins(topMargin:24, rightMargin:24, bottomMargin:24, leftMargin:24);
```

- Kelas Member yang merupakan extend dari kelas Customer melakukan method overriding pada method clone. Hal ini dilakukan karena perilaku method clone pada kelas Member berbeda dengan kelas Customer.

```
public Customer clone(){
    Customer newCustomer = new Customer();
    newCustomer.setType(this.type);
    newCustomer.setId(this.id);
    newCustomer.setInvoiceList(this.invoiceList);
    return newCustomer;
}
```

```
@Override
public Member clone(){
    Member newMember = new Member();
    newMember.setType(this.type);
    newMember.setId(this.id);
    newMember.setInvoiceList(this.invoiceList);
    newMember.setPoint(new BasePrice(this.getPoint().getValue()));
    newMember.setName(this.name);
    newMember.setAdditionalValue(this.additionalValue);
    newMember.setPhoneNumber(this.phoneNumber);
    newMember.setStatus(this.status);
    return newMember;
}
```

Page :

- ToolTipLabel menerapkan method overloading agar dapat dibuat dengan dua cara berbeda tergantung dengan signature parameter yang digunakan.

```
public class ToolTipLabel extends Label {  
    public ToolTipLabel(String text) {  
        super(text);  
  
        // Create tooltip with full label text  
        Tooltip tooltip = new Tooltip(text);  
  
        // Show tooltip when hovering over label  
        Tooltip.install(this, tooltip);  
    }  
  
    public ToolTipLabel(String title, String hover) {  
        super(title);  
  
        // Create tooltip with full label text  
        Tooltip tooltip = new Tooltip(hover);  
  
        // Show tooltip when hovering over label  
        Tooltip.install(this, tooltip);  
    }  
}
```

- Kelas App yang melakukan extends kelas Application dari JavaFX melakukan method overriding pada method start. Hal ini dilakukan untuk menginisialisasi UI yang diperlukan agar sesuai dengan tampilan yang telah dibuat.

```

    public class App extends Application {
        @Getter
        @Setter
        private static DataStore dataStore = new DataStore(mode:"JSON", folder:"assets/data/json");

        Run | Debug
        public static void main(String[] args) {
            // System.out.println("Starting App...");
            launch(args);
        }

        @Override
        public void start(Stage stage) throws Exception {
            stage.setTitle("Business Management Application");
            Image appIcon = new Image("file:assets/icons/capybucks.png");
        }
    }

```

5.5. Polymorphism

Models:

- Kelas FixedInvoice (models/FixedInvoice.java) dan TemporaryInvoice(models/TemporaryInvoice.java) extends Invoice.
- Kelas BroughtProduct (models/BroughtProduct.java) dan InventoryProduct (models/InventoryProduct.java) extends Product
- Kelas Member (models/Member.java) extends Kelas Customer (models/customer.java), tetapi pada pages/ProfileEditor.java, kelas Customer di-*polymorph* oleh kelas Member. Contohnya:

```

Customer newCustomer;
if (membershipDropdown.getValue().equals(anObject:"member")) {
    newCustomer = new Member(customer.getId(), customer.getInvoiceList(),
        nameForm.getContentTextField().getText(), phoneForm.getContentTextField().
        getText(), new HashMap<>());
}

```

Page :

- Kelas FormLabel extends kelas Label. Dalam penggunaannya sering dilakukan polymorph.

```
Label idLabel = new FormLabel(_content:"ID", idContainerWidth, idContainerHeight);
Label idContent = new FormLabel(customer.getId(), idContainerWidth, idContainerHeight);
```

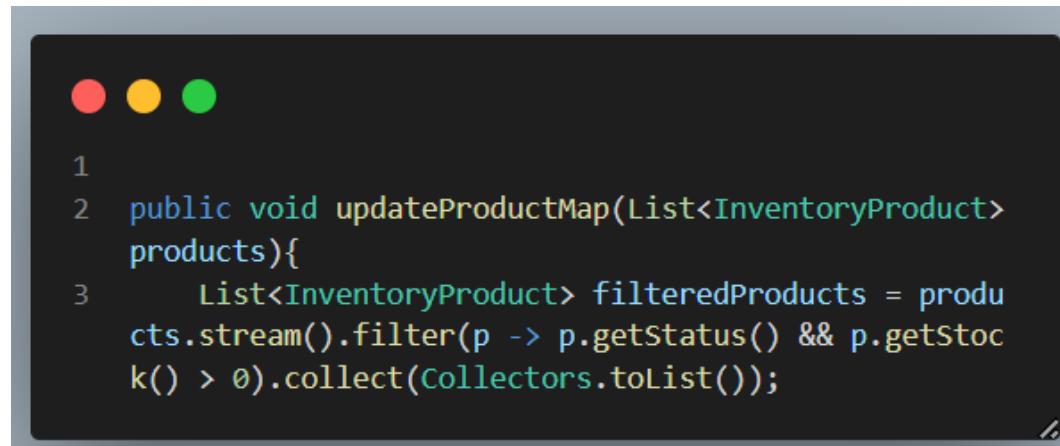
- Kelas DefaultButton (components/buttons/DefaultButton.java) *extends* Kelas Button pada JavaFx. Namun pada implementasinya, kelas Button sering di-*polymorph* oleh DefaultButton. Contohnya :

```
Button uploadButton = new DefaultButton(0.5 * uploadImageContainerWidth, uploadImageContainerHeight,
buttonName:"Upload");
```

5.6. Java API Collection

Models:

- Kelas TransactionProductCardList (components/cardlist/TransactionProductCardList.java) menggunakan List<InventoryProduct> dan java API stream filter untuk produk yang masih aktif
- Kelas Product (models/Product.java) menggunakan HashMap untuk menyimpan additional values seperti currency.



```
1
2 public void updateProductMap(List<InventoryProduct>
products){
3     List<InventoryProduct> filteredProducts = produ
cts.stream().filter(p -> p.getStatus() && p.getStoc
k() > 0).collect(Collectors.toList());
```

Page :

- Kelas InventoryPage menggunakan HashSet untuk membuat set category dari seluruh InventoryProduct.

5.7. SOLID

a) **S (Single Responsibility Principle):**

- Kelas InventoryProduct (models/InventoryProduct.java) hanya bertugas untuk menangani data produk yang terdapat pada datastore inventory.json, dan menjadikan atribut pada datastore menjadi atribut pada dirinya.
- Kelas Customer (models/Customer.java) hanya bertugas untuk menangani pengguna yang terdapat pada datastore clients.json, dan menjadikan atribut pada datastore menjadi atribut pada dirinya.

b) **O (Open/Close Principle):**

- Kelas ClientsCardList (view/components/ClientCardList.java) merupakan kelas yang memiliki List<T> customer . Kelas ini terbuka untuk ekstensi tetapi tertutup untuk modifikasi, dalam arti kelas lain dapat diturunkan dari kelas ClientCardList tanpa mengubah isi dari kelas tersebut. Misalnya, jika hendak ditambahkan sebuah kelas Member yang menampung data pengguna

yang dideaktivasi, tidak perlu lagi mengubah isi dari kelas ClientCardList, tetapi kita dapat menambah perilaku dan tanggung jawab dari ClientsCardList pada Member.

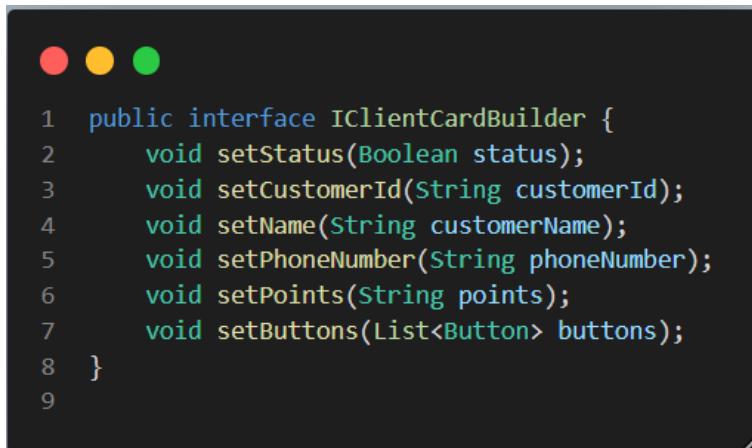
- Kelas DataList (models/DataList.java) merupakan kelas yang memiliki List<T> dataList yang extend dari Identifiable. Kelas ini terbuka untuk ekstensi tetapi tertutup untuk modifikasi, dalam arti kelas lain dapat diturunkan dari kelas DataList tanpa mengubah isi dari kelas tersebut. Misalnya, jika hendak ditambahkan sebuah kelas Product yang menampung data pengguna yang dideaktivasi, tidak perlu lagi mengubah isi dari kelas DataList, tetapi kita dapat menambah perilaku dan tanggung jawab dari DataList pada Product.

c) L (Liskov Substitution Principle):

- Pada kelas Customer (model/Customer.java) dapat diubah instancenya (Customer, Member, VIP) tanpa merubah fungsional program.

d) I (Interface Segregation Principle)

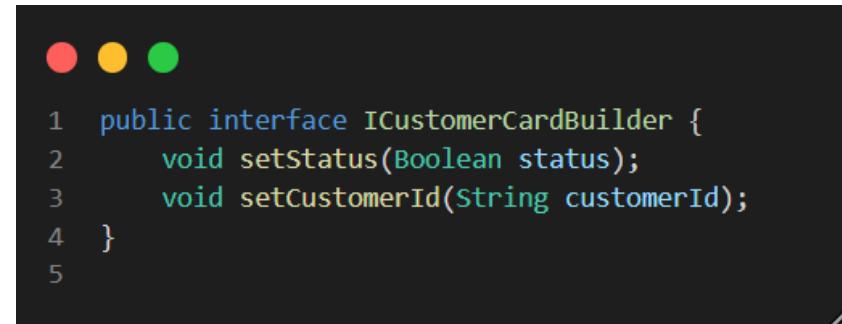
- Interface ICustomerCardBuilder (view/components/cards/clientcard/ICustomerCardBuilder.java) disegregasi dari IClientCardBuilder (view/components/cards/clientcard/IClientCardBuilder.java) karena Customer tidak perlu implement seluruh interface client.



```

1 public interface ICustomerCardBuilder {
2     void setStatus(Boolean status);
3     void setCustomerId(String customerId);
4     void setName(String customerName);
5     void setPhoneNumber(String phoneNumber);
6     void setPoints(String points);
7     void setButtons(List<Button> buttons);
8 }
9

```



```

1 public interface ICustomerCardBuilder {
2     void setStatus(Boolean status);
3     void setCustomerId(String customerId);
4 }
5

```

e) D (Dependency Inversion Principle)

- Kelas TemporaryInvoice (models/TemporaryInvoice.java) dan FixedInvoice (models/FixedInvoice.java) bergantung pada abstraksi. Kedua kelas tersebut bergantung pada kelas abstrak Invoice (models/Invoice.java)

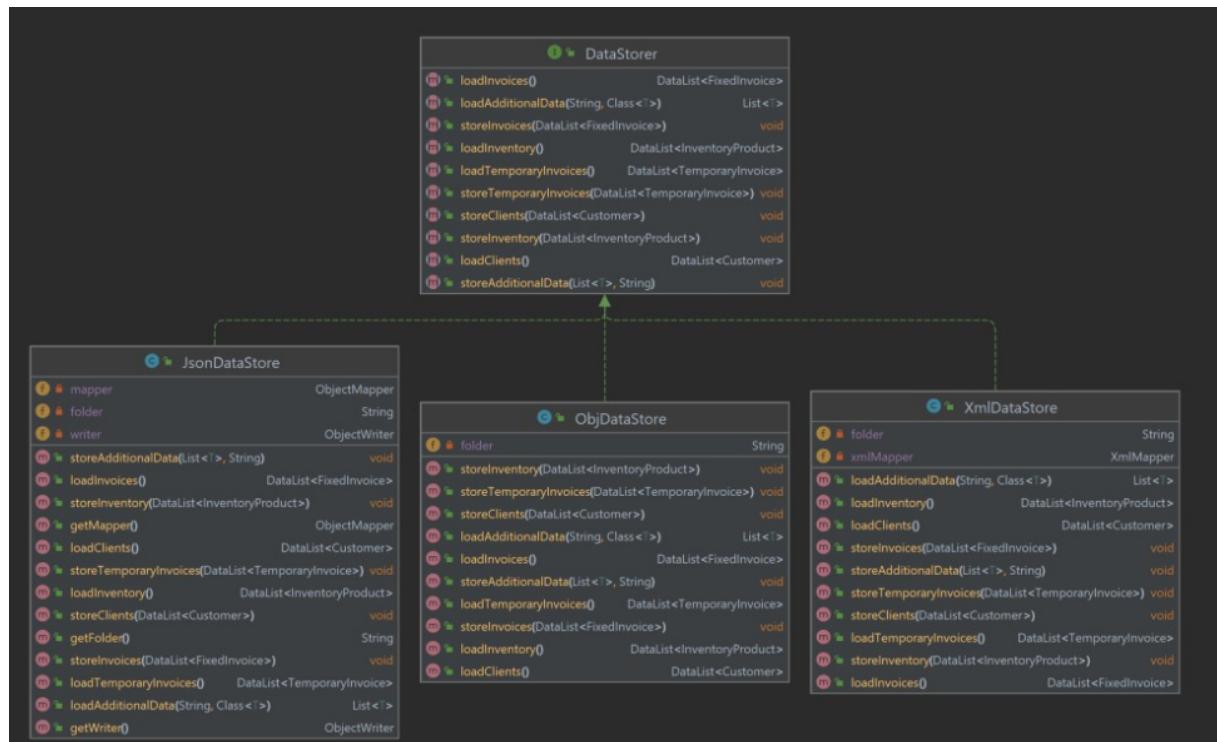
- Kelas BoughtProduct (models/BoughtProduct.java) dan InventoryProduct (models/InventoryProduct.java) bergantung pada abstraksi. Kedua kelas tersebut bergantung pada kelas abstrak Product (models/Product.java)

5.8. Design Pattern

a. Adapter Pattern

Adapter Pattern diimplementasikan pada DataStore. DataStore merupakan *class* khusus dalam aplikasi yang menyimpan data-data sementara aplikasi dan juga memiliki peran penting sebagai jembatan antara aplikasi dan juga data-data yang disimpan dalam file. Namun, karena *file-file* penyimpanan bisa berupa banyak jenis, maka DataStore menggunakan sebuah adapter bernama DataStorer yang merupakan *interface* yang digunakan untuk berinteraksi dengan file sesuai dengan mode yang dipilih.

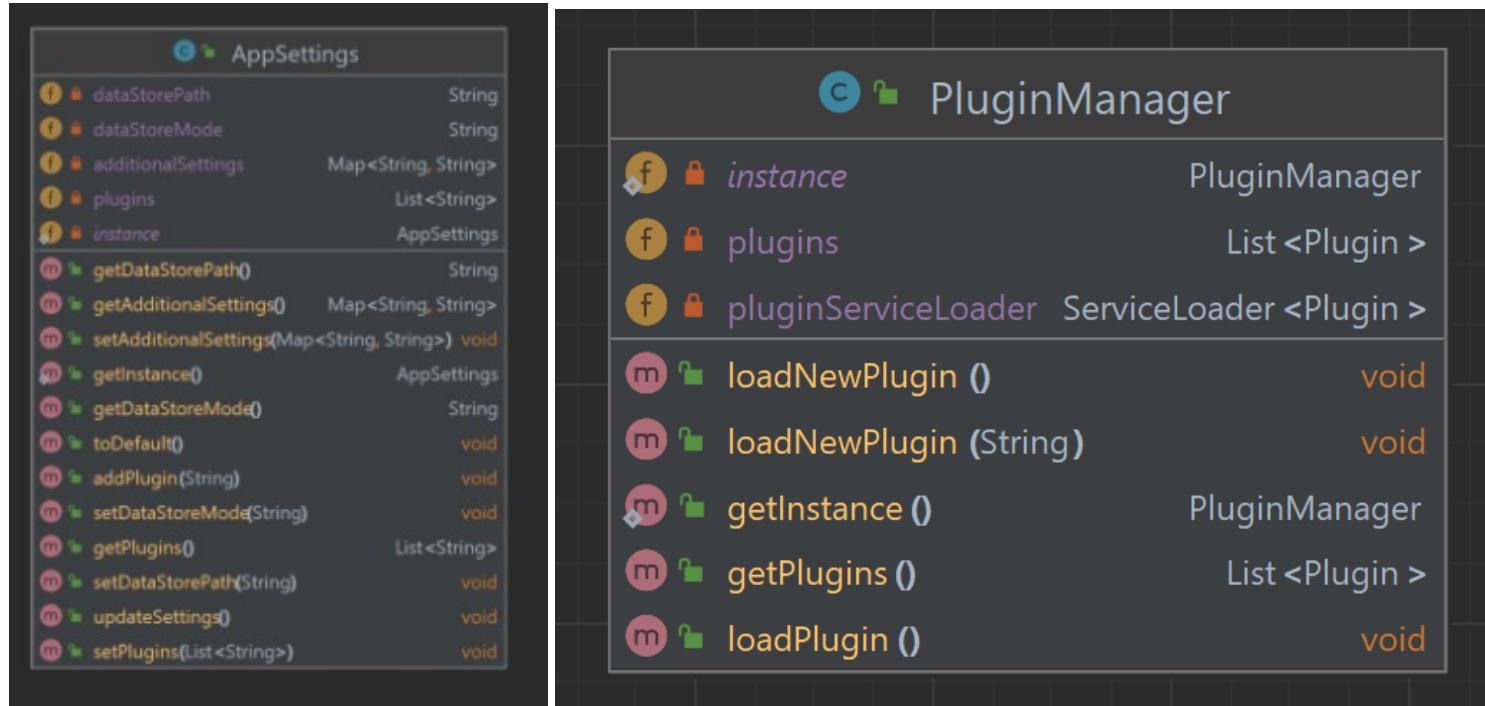
JsonDataStorer, XmlDataStorer, ObjDataStorer adalah *adapter* untuk masing-masing mode sehingga dapat digunakan dengan mudah oleh DataStore. Setiap *class* ini melakukan “*adapting*” method-method yang disediakan oleh library yang digunakan untuk masing-masing mode DataStore sehingga dapat dengan mudah digunakan oleh DataStore tanpa perlu mengetahui bagaimana setiap DataStorer berinteraksi dengan file.



b. Singleton Pattern

Singleton Pattern adalah pattern yang memastikan class hanya memiliki 1 *instance* dalam kelangsungan aplikasi. Hal ini berguna untuk beberapa class yang dipastikan hanya ada satu di aplikasi, seperti MainView yaitu window utama aplikasi, AppSettings yaitu class yang menyimpan settings dari aplikasi, PluginManager, HeaderMenuBar.

Singleton Pattern yang diimplementasikan juga sudah *thread-safe* sehingga dapat dipastikan hanya terdapat satu *instance* meskipun beberapa thread berjalan bersamaan. Pattern ini berguna untuk kemudahan mendapatkan instansiasi dari sebuah *class* tanpa perlu melakukan banyak *passing reference* ke komponen-komponen dalam App karena dipastikan hanya ada satu instansiasi.

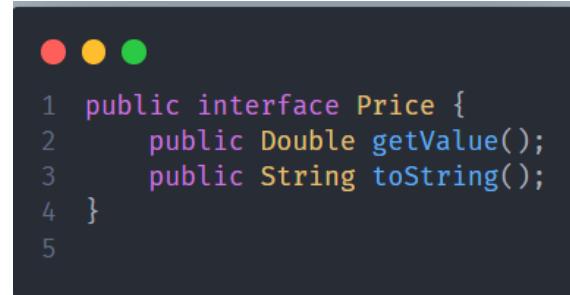


c. Decorator Pattern

Decorator Pattern adalah sebuah design pattern yang memberikan kapabilitas untuk menambahkan informasi ke sebuah objek dengan cara memanfaatkan sebuah *interface*. Pattern ini digunakan untuk plugin Currency dan juga untuk melakukan *rendering* additional values (metadata) dari models yang ada.

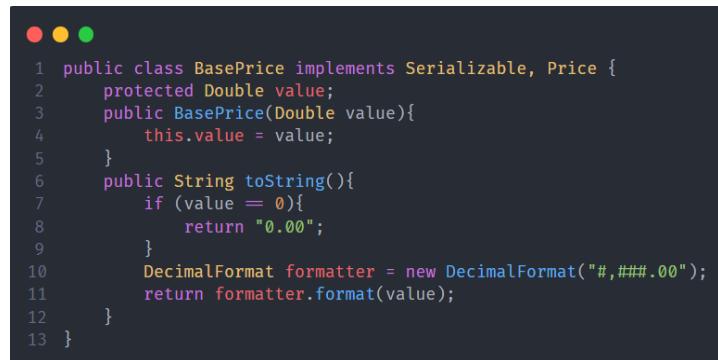
Berikut adalah penggunaan Decorator Pattern untuk Currency:

Main App hanya mengetahui adanya sebuah interface Price



```
● ● ●
1 public interface Price {
2     public Double getValue();
3     public String toString();
4 }
5
```

Price ini hanya memiliki method getValue yang merupakan nilai dari Price tersebut dan juga toString yaitu bagaimana cara Price dicetak sebagai sebuah String. Kemudian, Main App juga mengetahui adanya BasePrice yaitu harga dasar yang diketahui oleh App.



```
● ● ●
1 public class BasePrice implements Serializable, Price {
2     protected Double value;
3     public BasePrice(Double value){
4         this.value = value;
5     }
6     public String toString(){
7         if (value == 0){
8             return "0.00";
9         }
10        DecimalFormat formatter = new DecimalFormat("#,###.00");
11        return formatter.format(value);
12    }
13 }
```

Main App juga menyediakan sebuah interface decorator sehingga Price bisa didecorate:

```
● ● ●  
1 public abstract class PriceDecorator implements Price{  
2     @Getter protected Price price;  
3     public PriceDecorator(Price price){  
4         this.price = price;  
5     }  
6     public String toString(){  
7         return price.toString();  
8     }  
9     public Double getValue(){  
10        return price.getValue();  
11    }  
12 }
```

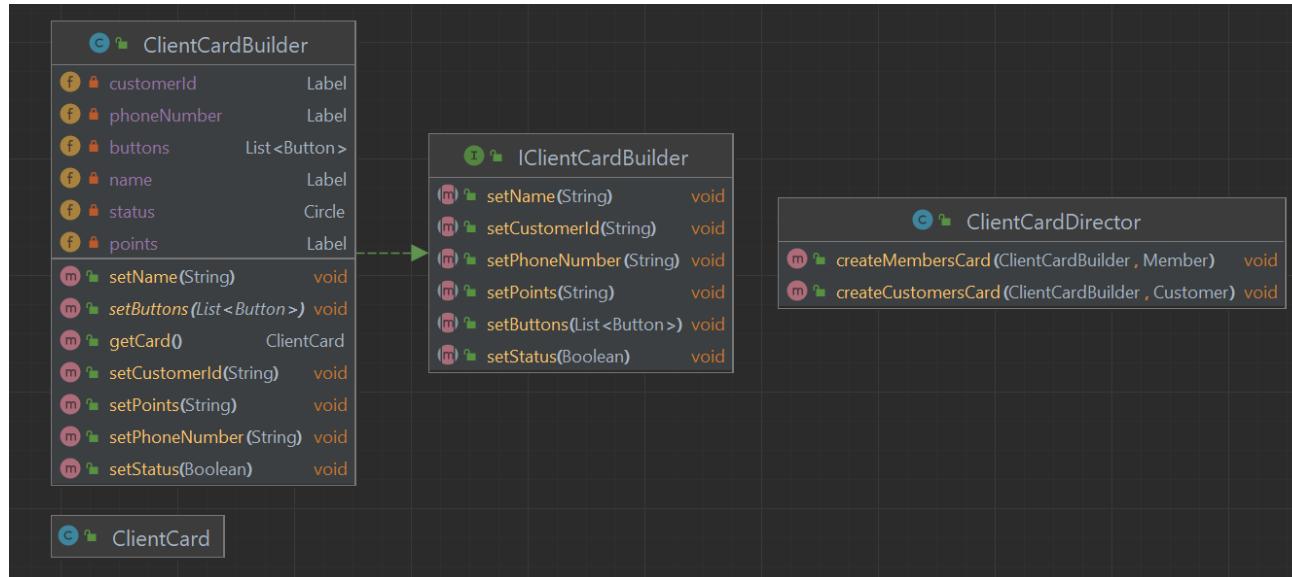
Kemudian, currency plugin memiliki class CurrencyPrice yang mengimplementasikan Price sehingga MainApp dapat menunjukkan Price dengan currency tanpa perlu mengetahui *decorator* atau informasi baru apa yang dapat ditambahkan ke Price.



d. Builder Pattern

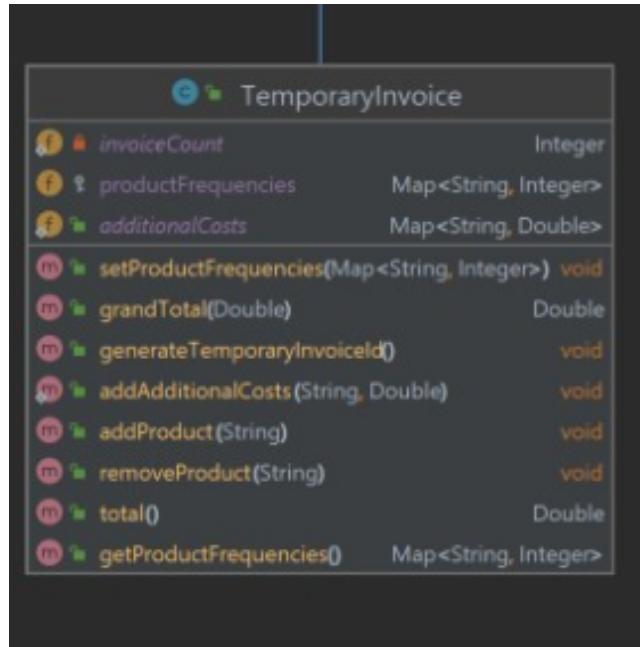
Builder pattern digunakan untuk meng-*construct* card Customer, Member, dan VIP. Hal ini dikarenakan *card* memiliki informasi dasar yang mirip tetapi komponen-komponen yang berbeda.

Builder pattern diimplementasikan menggunakan ClientCardDirector yang mendefinisikan cara meng-*construct* *card* untuk tipe customer yang bersesuaian. Pattern ini berguna sehingga tidak perlu dibuat banyak *class* berbeda untuk card Customer, Member, dan VIP, selain itu kita juga tidak perlu membuat *class* list berbeda untuk masing-masing card.



e. Flyweight

Flyweight pattern adalah pattern yang menyimpan separuh informasi dari objek yang berasal dari *class* lain tanpa menyimpan informasi tentang keseluruhan objek tersebut di *class* secara komposit. Flyweight digunakan untuk *TemporaryInvoice* dimana pada *TemporaryInvoice* hanya menyimpan ID dari *InventoryProduct* tanpa harus menyimpan objek dari *InventoryProduct* itu sendiri. Hal ini digunakan agar tidak dapat dengan mudah menyimpan dan mengubah models pada *DataStore*. Key pada *productFrequencies* adalah ID dari *InventoryProduct*.



5.9. Reflection

- Kelas PluginManager menggunakan ClassLoader dan juga method getClass
- Kelas ProfileEditor (view/ProfileEditor.java) menggunakan reflection ‘instanceof Member’ dan ‘instanceof VIP’ untuk memeriksa kelas parameter konstruktor dan menampilkan membership dan button yang sesuai

5.10. Threading

- Kelas ClockWidget mengimplementasikan threading untuk melakukan *update* jam
- Semua kelas yang mengimplementasikan plugin juga mengimplementasikan threading untuk meng-update state aplikasi

- Kelas ClientsPage dan InventoryPage mengimplementasikan threading agar daftar yang ditampilkan sesuai dengan perubahan yang terjadi di aplikasi

6. Bonus Yang dikerjakan

6.1. Annotation

6.1.1. Lombok

Implementasi Lombok dilakukan dengan menambahkan dependency Lombok pada pom.xml. Lombok yang digunakan adalah versi 1.28.26. Fitur lombok yang paling sering digunakan adalah `@Getter`, `@Setter`, `@AllArgsConstructor`, dan `@NoArgsConstructor`. Lombok sangat membantu agar programmer tidak perlu membuat function `getter`, `setter`, dan `constructor` secara manual. Pada file yang mengimplementasikan lombok, cukup melakukan `import lombok.<fitur>`, contohnya `import lombok.Setter;` Berikut adalah contoh implementasi fitur Lombok:

```
    @Getter  
    @Setter  
    private static DataStore dataStore = new DataStore(mode:"JSON", folder:"assets/data/json");
```

6.1.2. Null Safety dengan JetBrains Annotation

Implementasi Null Safety dilakukan dengan menggunakan JetBrains Annotation dan menambahkan dependency pada pom.xml. JetBrains Annotation yang digunakan adalah versi 24.0.1. Fitur Null Safety digunakan dengan menambahkan `@NotNull` pada suatu attribute yang tidak boleh bernilai Null. Pada file yang mengimplementasikan Null Safety, cukup melakukan `import org.jetbrains.annotations.NotNull;` Berikut adalah contoh implementasi fitur Null Safety:

```
@Getter  
@Setter  
public class Identifiable implements Serializable {  
    @NotNull protected String id;  
}
```

7. Pembagian Tugas

NIM	Nama	Tugas
13516055	Nathaniel Evan Gunawan	Page: History Transaksi
13521044	Rachel Gabriela Chen	Page: Client List, Plugin Pie Chart, Settings Page, Plugins Page Functionality: Datastore, Plugin Tax, Plugin Currency, Models, App Plugin Handling (Plugin & Plugin Manager), App Settings handling
13521046	Jeffrey Chow	Page: Transaction Page and related components Functionality: Datastore (functions used in Transaction Page), Lombok, Null Safety
13521074	Eugene Yap Jin Quan	Page: Homepage Functionality: Tab management, Plugin Bar & Line Chart

13521094	Angela Livia Arumsari	Page: Inventory Management Functionality: Export PDF
13521100	Alexander Jason	Page: Add Item, Edit Item, Edit/Upgrade Profile Functionality: Update Datastore (Customer and Inventory)

8. Foto Kelompok

