TỔNG LIÊN ĐOÀN LAO ĐỘNG VIỆT NAM
TRƯỜNG ĐẠI HỌC TÔN ĐỨC THẮNG
KHOA CÔNG NGHỆ THÔNG TIN

LÊ THÀNH TIẾN – 521H0485
PHẠM VĂN TIẾN ĐẠT – 521H0030

PHÁT TRIỂN ỨNG DỤNG DI ĐỘNG QUẢN LÝ HẠN SỬ DỤNG CỦA ĐỒ ĂN

ĐỒ ÁN CUỐI KỲ MÔN MẪU THIẾT KẾ KỸ THUẬT PHẦN MỀM

THÀNH PHỐ HỒ CHÍ MINH, 2024

TỔNG LIÊN ĐOÀN LAO ĐỘNG VIỆT NAM
TRƯỜNG ĐẠI HỌC TÔN ĐỨC THẮNG
KHOA CÔNG NGHỆ THÔNG TIN

LÊ THÀNH TIẾN – 521H0485
PHẠM VĂN TIẾN ĐẠT – 521H0030

PHÁT TRIỂN ỨNG DỤNG DI ĐỘNG QUẢN LÝ HẠN SỬ DỤNG CỦA ĐỒ ĂN

ĐỒ ÁN CUỐI KỲ MÔN MẪU THIẾT KẾ KỸ THUẬT PHẦN MỀM

Người hướng dẫn
ThS. Vũ Đình Hồng

THÀNH PHỐ HỒ CHÍ MINH, 2024

LỜI CẢM ƠN

Lời đầu tiên, chúng tôi xin gửi lời cảm ơn đến sự hỗ trợ nhiệt tình của Trường Đại học Tôn Đức Thắng và khoa Công nghệ thông tin vì đã cung cấp cho chúng tôi các tài liệu và thiết bị cần thiết, tạo điều kiện thuận lợi cho quá trình tìm hiểu và thử nghiệm các giải pháp trong quá trình phát triển đồ án.

Ngoài ra, chúng tôi xin gửi lời cảm ơn chân thành đến ThS. Vũ Đình Hồng, người đã cung cấp những hướng dẫn quý báu và mang tính xây dựng trong quá trình thực hiện. Những phân tích sâu sắc của thầy đóng vai trò quan trọng trong việc định hướng nghiên cứu và kết quả đầu ra của đồ án này. Bên cạnh đó, những kiến thức được thầy truyền đạt trong quá trình học tập giúp chúng tôi vượt qua những thử thách và khó khăn gặp phải trong quá trình thực hiện và phát triển đồ án.

Thành phố Hồ Chí Minh, ngày 30 tháng 04 năm 2024
Tác giả

Lê Thành Tiến

Phạm Văn Tiến Đạt

PHIẾU ĐÁNH GIÁ CỦA GIẢNG VIÊN HƯỚNG DẪN

Tên giảng viên hướng dẫn: .....

Ý kiến nhận xét: .....

.....

.....

.....

Điểm tổng theo phiếu đánh giá rubrik: .....

TP. Hồ Chí Minh, ngày tháng năm 20

Giảng viên hướng dẫn

(Ký tên và ghi rõ họ tên)

CÔNG TRÌNH ĐƯỢC HOÀN THÀNH
TẠI TRƯỜNG ĐẠI HỌC TÔN ĐỨC THẮNG

Nhóm chúng em xin cam đoan đây là công trình nghiên cứu của riêng chúng em và được sự hướng dẫn khoa học của ThS. Vũ Đình Hồng. Các nội dung nghiên cứu, kết quả trong đề tài này là trung thực và chưa công bố dưới bất kỳ hình thức nào trước đây. Những số liệu trong các bảng biểu phục vụ cho việc phân tích, nhận xét, đánh giá được chính tác giả thu thập từ các nguồn khác nhau có ghi rõ trong phần tài liệu tham khảo.

Ngoài ra, trong báo cáo còn sử dụng một số nhận xét, đánh giá cũng như số liệu của các tác giả khác, cơ quan tổ chức khác đều có trích dẫn và chú thích nguồn gốc.

Nếu phát hiện có bất kỳ sự gian lận nào nhóm chúng em xin hoàn toàn chịu trách nhiệm về nội dung Báo cáo Đồ án cuối kỳ của mình. Trường Đại học Tôn Đức Thắng không liên quan đến những vi phạm tác quyền, bản quyền do chúng em gây ra trong quá trình thực hiện (nếu có).

TP. Hồ Chí Minh, ngày 30 tháng 04 năm 2024

Tác giả

Lê Thành Tiến

Phạm Văn Tiến Đạt

TÓM TẮT

Báo cáo này phân tích về việc áp dụng các mẫu thiết kế khác nhau để cải thiện mã nguồn của một ứng dụng di động theo dõi hạn sử dụng của thực phẩm tươi sống. Trong dự án này, các mẫu thiết kế được áp dụng là Singleton, Strategy, Template, Factory, Command, Adapter, Prototype, Proxy, và Model - View – ViewModel (MVVM). Chúng tôi phân tích chức năng hiện tại của ứng dụng, đưa ra giải pháp là cách áp dụng từng loại mẫu thiết kế cụ thể nhằm cải thiện cách thức hoạt động trong ứng dụng, và trình bày cách thức mẫu thiết kế tương ứng được áp dụng trong ứng dụng.

MỤC LỤC

DANH MỤC HÌNH VẼ

DANH MỤC CÁC CHỮ VIẾT TẮT

API	Application Programming Interface
MVVM	Model-View-ViewModel
UI	User Interface

CHƯƠNG 1 – MỞ ĐẦU

1.1 Mục đích

Báo cáo này nhằm mục đích mô tả chi tiết lý do lựa chọn các mẫu thiết kế cụ thể vào một ứng dụng giúp quản lý hạn sử dụng của đồ ăn, bao gồm Singleton, Strategy, Template, Factory, Command, Adapter, Prototype, Proxy và Model-View-ViewModel (MVVM); đồng thời minh họa cách triển khai các mẫu thiết kế này nhằm tăng khả năng bảo trì và khả năng mở rộng của ứng dụng. Bằng việc sử dụng các mẫu thiết kế này, mã nguồn của ứng dụng trở nên có cấu trúc hơn, dễ dàng sửa đổi, cho phép bảo trì và mở rộng trong tương lai. Báo cáo đi sâu vào những lợi ích riêng biệt của từng mẫu và minh họa các chúng được áp dụng, góp phần tạo nên một thiết kế phần mềm có tính hiệu quả về mặt tổng thể hơn.

1.2 Phạm vi đồ án

Báo cáo tập trung phân tích và áp dụng một số mẫu thiết kế vào một ứng dụng di động dùng để theo dõi hạn sử dụng của các mặt hàng thực phẩm trên hệ điều hành Android và được xây dựng chủ yếu với ngôn ngữ lập trình Java. Mục đích của việc áp dụng này nhằm cải thiện kiến trúc, khả năng bảo trì và khả năng mở rộng của ứng dụng thông qua các mẫu thiết kế được áp dụng rộng rãi để giải quyết vấn đề cụ thể nào đó về thiết kế phần mềm.

Các mẫu thiết kế được lựa chọn cho dự án này bao gồm Singleton, Strategy, Template, Factory, Command, Adapter, Prototype, Proxy và Model-View-ViewModel (MVVM). Các mẫu thiết kế này được chọn vì mức độ phù hợp với các yêu cầu và chức năng của dụng. Mục đích của từng mẫu thiết kế được áp dụng có thể tóm tắt lại như sau:

- Singleton: Tạo đối tượng duy nhất của một API được sử dụng xuyên suốt trong ứng dụng.
- Strategy: Sắp xếp danh sách các nguyên liệu trong nhà bếp dựa trên một số tiêu chí nhất định trong quá trình người dung sử dụng ứng dụng.
- Template: Xuất nguyên liệu cần chuẩn bị của từng món ăn dưới ảnh hoặc tài liệu nhằm hỗ trợ người dùng tạo danh sách đi chợ,…
- Factory: Tạo các công thức món ăn dựa trên nguồn gốc, địa danh gắn liền với các loại món ăn (ví dụ: món ăn Ý, Hàn, Nhật,…)

- Command: Hỗ trợ các hoạt động có thể được hoàn tác lại, chẳng hạn khi xóa một nguyên liệu ra khỏi tủ lạnh.
- Adapter: Sử dụng các công thức món ăn từ một nhà cung cấp khác, và kết quả được gửi về có cấu trúc và định dạng khác so với cấu trúc và định dạng mà ứng dụng đang sử dụng.
- Prototype: Tạo ra một bản sao của một đối tượng phức tạp và trong ứng dụng này chính là đối tượng nguyên liệu thành phần trong tủ lạnh.
- Proxy: Kiểm soát quyền truy cập vào một nhà bếp được chia sẻ của người dùng khác, nhằm thêm một lớp bảo vệ cho dữ liệu của người dùng.
- MVVM: Phân tách các lớp UI (View) với lớp thực thi (ViewModel) trong mã nguồn một cách rõ rang, nhằm nâng cao khả năng kiểm thử và tính mô-đun của ứng dụng.

CHƯƠNG 2 – ÁP DỤNG CÁC MẪU THIẾT KẾ

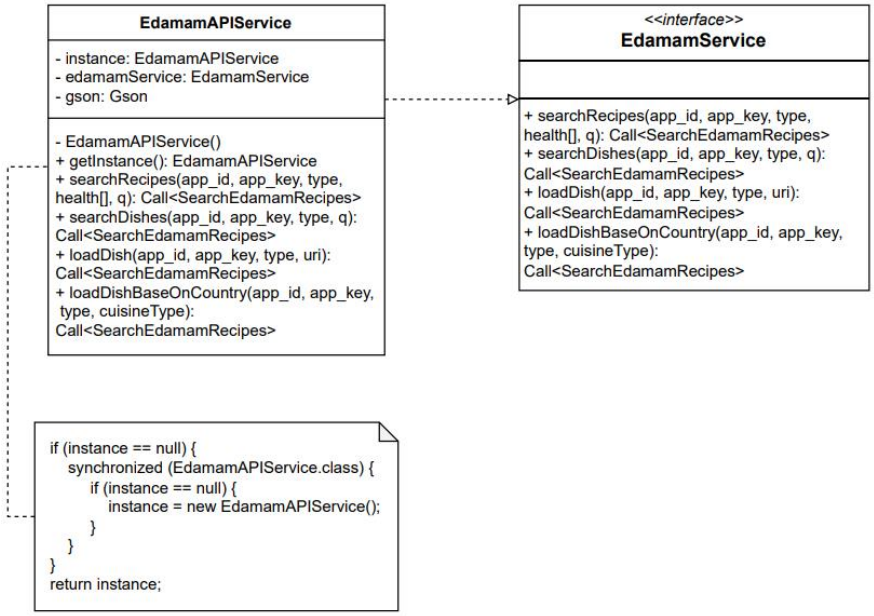
2.1 Singleton

2.1.1 Giới thiệu và lý do áp dụng

Mẫu thiết kế Singleton được sử dụng trong ứng dụng nhằm mục đích quản lý các kết nối API (Edamam API) về việc hiển thị các món ăn gợi ý dựa vào các nguyên liệu có sẵn trong tủ lạnh, và các hạn chế cá nhân của người dùng (dị ứng với dậu, trái cây, người ăn chay,..). Ngoài ra, API còn hỗ trợ vào việc hiển thị chi tiết các món ăn như thành phần, nguyên liệu, và quốc gia (món Á, Âu,..). Hơn thế nữa, API còn giúp người dùng tra cứu cách chế biến dựa vào tên món ăn, tên thành phần,... hỗ trợ người dùng lên kế hoạch cho những bữa ăn trong tuần, trong tháng.

Từ những chức năng trên của phần mềm khi được áp dụng API thì mẫu thiết kế Singleton hoàn toàn phù hợp trong việc hỗ trợ ứng dụng quản lý API, đảm bảo an toàn cho việc chạy đa luồng (synchronized trong Java), tái sử dụng một kết nối duy nhất có thể cải thiện hiệu suất và tốc độ của ứng dụng, đặc biệt đối với Edamam API hỗ trợ thường siêng trong suốt quá trình sử dụng (Joshi 2016).

2.1.2 Sơ đồ lớp



Hình 2.1 Sơ đồ lớp áp dụng mẫu thiết kế Singleton

Lớp EdamamAPIService hoạt động như một Singleton, được cài đặt bằng phương pháp Double-Checked Locking, hỗ trợ cho việc chạy đa luồng, tương tự với phương pháp Lazy Initialization, nhưng sử dụng khóa kiểm tra kép để cải thiện hiệu suất đồng thời. Lớp này chỉ khởi tạo thông qua phương thức getInstance(). Lớp interface EdamamService chứa những phương thức trả về của Edamam API như gợi ý món ăn, hiển thị chi tiết về món ăn, gợi ý những món ăn theo các quốc gia, ...

2.1.3 Mã nguồn áp dụng mẫu thiết kế

```
public interface EdamamService {
    2 usages  1 implementation  Tien Le
    @GET("api/recipes/v2")
    Call<SearchEdamamRecipes> searchRecipes(@Query("app_id") String app_id,
                                            @Query("app_key") String app_key,
                                            @Query("type") String type,
                                            @Query("health") String[] health,
                                            @Query("q") String q);

    3 usages  1 implementation  Tien Le
    @GET("api/recipes/v2")
    Call<SearchEdamamRecipes> searchDishes(@Query("app_id") String app_id,
                                           @Query("app_key") String app_key,
                                           @Query("type") String type,
                                           @Query("q") String q);

    4 usages  1 implementation  Tien Le
    @GET("api/recipes/v2/by-uri")
    Call<SearchEdamamRecipes> loadDish(@Query("app_id") String app_id,
                                       @Query("app_key") String app_key,
                                       @Query("type") String type,
                                       @Query("uri") String uri);

    2 usages  1 implementation  Tien Le
    @GET("api/recipes/v2")
    Call<SearchEdamamRecipes> loadDishBaseOnCountry(@Query("app_id") String app_id,
                                                    @Query("app_key") String app_key,
                                                    @Query("type") String type,
                                                    @Query("cuisineType") String cuisineType);
```

Hình 2.2 Giao diện (interface) EdamamService

```java
public class EdamamAPIService implements EdamamService {
    4 usages
    private static EdamamAPIService instance;
    5 usages
    private final EdamamService edamamService;
    1 usage
    private static final Gson gson = new GsonBuilder()
            .setLenient()
            .create();

    1 usage  Tien Le
    private EdamamAPIService() {
        Retrofit retrofit = new Retrofit.Builder()
                .baseUrl("https://api.edamam.com/")
                .addConverterFactory(GsonConverterFactory.create(gson))
                .build();
        edamamService = retrofit.create(EdamamService.class);
    }

    Tien Le
    public static EdamamAPIService getInstance() {
        if (instance == null) {
            synchronized (EdamamAPIService.class) {
                if (instance == null) {
                    instance = new EdamamAPIService();
                }
            }
        }
```

Hình 2.3 Lớp của Singleton (EdamamAPIService)

```java
@Override
public Call<SearchEdamamRecipes> searchRecipes(String app_id, String app_key, String type, String[] health, String q) {
    return edamamService.searchRecipes(app_id, app_key, type, health, q);
}

3 usages  Tien Le
@Override
public Call<SearchEdamamRecipes> searchDishes(String app_id, String app_key, String type, String q) {
    return edamamService.searchDishes(app_id, app_key, type, q);
}

4 usages  Tien Le
@Override
public Call<SearchEdamamRecipes> loadDish(String app_id, String app_key, String type, String uri) {
    return edamamService.loadDish(app_id, app_key, type, uri);
}

2 usages  Tien Le
@Override
public Call<SearchEdamamRecipes> loadDishBaseOnCountry(String app_id, String app_key, String type, String cuisineType) {
    return edamamService.loadDishBaseOnCountry(app_id, app_key, type, cuisineType);
}
```

Hình 2.4 Một số phương thức trong lớp của Singleton (EdamamAPIService)

```
@Override
public void onBindViewHolder(@NonNull WeeklyMealDishesViewHolder holder, int position) {
    int curPosition = holder.getAdapterPosition();
    PlanForMeal planForMeal = data.get(curPosition);

    Tien Le *
    EdamamAPIService.getInstance().loadDish(APP_ID, APP_KEY,  type: "public", planForMeal.getDishUri()).enqueue(new Callback

        Tien Le
        @Override
        public void onResponse(@NonNull Call<SearchEdamamRecipes> call, @NonNull Response<SearchEdamamRecipes> response) {
            recipe = new ArrayList<>(Arrays.asList(response.body().getFoodRecipes()));
            holder.getRecipeTitleTextView().setText(recipe.get(0).getRecipeModel().getLabel());
            Glide.with(holder.getRecipeImageView().getContext()).load(recipe.get(0).getRecipeModel().getImage())
                    .centerCrop()
                    .diskCacheStrategy(DiskCacheStrategy.ALL)
                    .into(holder.getRecipeImageView());
            Log.i( tag: "TAG", recipe.get(0).getRecipeModel().getLabel());
        }
```

Hình 2.5 Singleton được khởi tạo để hiển thị chi tiết về từng món ăn

```
@Override
public void fetchRecipes(RecipesViewModel vm, String ingredients) {
    List<RootObjectModel> recipesList = new ArrayList<>();
    datit
    EdamamAPIService.getInstance().searchDishes(APP_ID, APP_KEY, type: "public", ingredients).enqueue(new Callback<SearchEdar
        datit
        @Override
        public void onResponse(@NonNull Call<SearchEdamamRecipes> call, @NonNull Response<SearchEdamamRecipes> response) {
            if (response.body() != null) {
                recipesList.addAll(Arrays.asList(response.body().getFoodRecipes()));
                vm.getRecipes().postValue(recipesList);
                vm.updateRecipes(recipesList);
            }
        }
```

Hình 2.6 Singleton được khởi tạo để hiển thị các món ăn gợi ý dựa vào thành phần có sẵn trong tủ

```
@Override
public void fetchRecipes(RecipesViewModel vm, String ingredients, String[] healthyLabels) {
    List<RootObjectModel> recipesList = new ArrayList<>();
    Tien Le
    EdamamAPIService.getInstance().searchRecipes(APP_ID, APP_KEY, type: "public", healthyLabels, ingredients).enqueue(new C
        Tien Le
        @Override
        public void onResponse(@NonNull Call<SearchEdamamRecipes> call, @NonNull Response<SearchEdamamRecipes> response) {
            if (response.body() != null) {
                recipesList.addAll(Arrays.asList(response.body().getFoodRecipes()));
                vm.getRecipes().postValue(recipesList);
                vm.updateRecipes(recipesList);
            }
        }
```

Hình 2.7 Singleton được khởi tạo để hiển thị các món ăn gợi ý dựa vào thành phần và những hạn chế của người dùng

```
public void prepareData(CuisineType cuisineType) {
    EdamamAPIService.getInstance().loadDishBaseOnCountry(
            APP_ID, APP_KEY,
            type: "public",
            Tien Le
            cuisineType.getCuisineType()).enqueue(new Callback<SearchEdamamRecipes>() {
        Tien Le
        @Override
        public void onResponse(Call<SearchEdamamRecipes> call, Response<SearchEdamamRecipes> response) {
            recipe = new ArrayList<>(Arrays.asList(response.body().getFoodRecipes()));
            mRecipes.postValue(Arrays.asList(response.body().getFoodRecipes()));
        }
```

Hình 2.8 Singleton được khởi tạo để hiển thị các món ăn theo quốc gia

2.2 Strategy

2.2.1 Giới thiệu và lý do áp dụng

Mẫu thiết kế Strategy được sử dụng trong ứng dụng nhằm sắp xếp danh sách nguyên liệu trong tủ thực phẩm một cách linh hoạt dựa trên các tiêu chí khác nhau trong quá trình sử dụng ứng dụng.

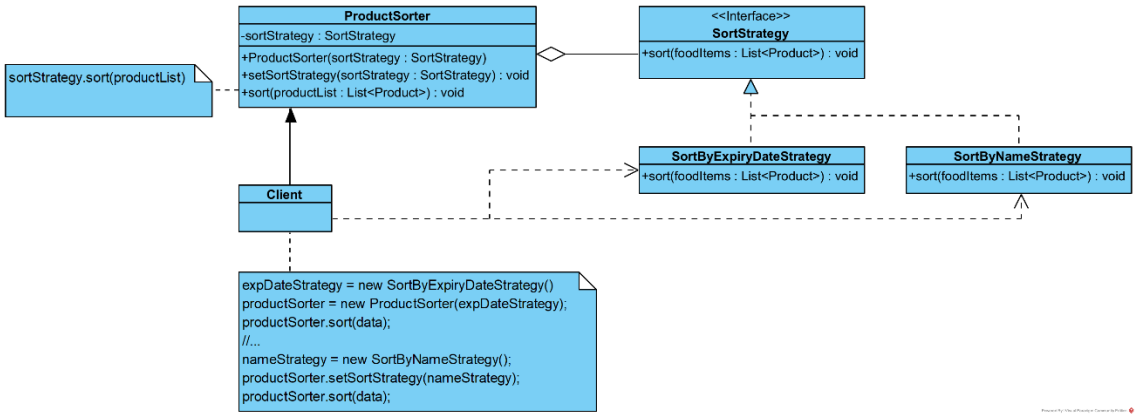
Strategy phù hợp cho trường hợp này vì nó cho phép định nghĩa nhiều thuật toán sắp xếp (chiến lược), được đóng góp trong các lớp riêng biệt và cho phép lựa chọn một chiến lược cụ thể tại thời gian chạy (Anon n.d.-c). Cách thiết kế này thúc đẩy tính linh hoạt và khả năng mở rộng vì các tiêu chí sắp xếp mới có thể được thêm vào mà không cần chỉnh sửa mã nguồn hiện tại (Hu 2023).

Tổng hợp lại, sau khi áp dụng mẫu thiết kế Strategy, dự án đạt được những lợi ích sau:

- Tính linh hoạt: Strategy cho phép các thuật toán sắp xếp được định nghĩa trong các lớp riêng biệt, thế nên, người dùng có thể chuyển đổi giữa các chiến lược sắp xếp một cách linh hoạt trong quá trình sử dụng.
- Tính mở rộng: Các tiêu chí hoặc thuật toán sắp xếp mới có thể được thêm vào ứng dụng mà không cần sửa đổi mã nguồn hiện có. Từ đó thúc đẩy khả năng mở rộng, cho phép ứng dụng bổ sung tính năng phù hợp với những yêu cầu mới.
- Tính bảo trì: Bằng cách đóng gói từng thuật toán sắp xếp trong lớp riêng của nó, mẫu thiết kế Strategy giúp duy trì một mã nguồn gọn gàng và có tính mô-đun.

Bên cạnh đó, sự thay đổi với một chiến lược sắp xếp nào đó sẽ không ảnh hưởng đến các chiến lược khác, giảm thiểu nguy cơ gây ra lỗi mới không mong muốn.

2.2.2 Sơ đồ lớp



Hình 2.9 Sơ đồ lớp áp dụng mẫu thiết kế Strategy

Chiến lược sắp xếp được áp dụng trong ứng dụng là sắp xếp dựa trên tên nguyên liệu và ngày sản xuất. Hai chiến lược này được khái quát thành giao diện (interface) SortStrategy. Lớp ProductSorter có thuộc tính chứa thông tin của chiến lược sắp xếp và hàm để bắt đầu thực thi việc sắp xếp. Lớp Client đại diện cho những nơi mà chiến lược sắp xếp được sử dụng và chiến lược sắp xếp có thể thay đổi trong quá trình người dùng sử dụng.

2.2.3 Mã nguồn áp dụng mẫu thiết kế

```
public interface SortStrategy {
    1 usage  2 implementations  Tien Le
    void sort(List<Product> foodItems);
}
```

Hình 2.10 Giao diện (interface) SortStrategy

```
public class SortByNameStrategy implements SortStrategy{
    1 usage  Tien Le
    @Override
    public void sort(List<Product> foodItems) {
        foodItems.sort(Comparator.comparing(Product::getTitle));
    }
}
```

Hình 2.11 Lớp của chiến lược sắp xếp theo tên nguyên liệu

```
public class SortByExpiryDateStrategy implements SortStrategy {
    1 usage  Tien Le
    @Override
    public void sort(List<Product> foodItems) {
        foodItems.sort(Comparator.comparing(Product::getExpiryDate));
    }
}
```

Hình 2.12 Lớp chiến lược sắp xếp theo hạn sử dụng

```
public class ProductSorter {
    3 usages
    private SortStrategy sortStrategy;

    1 usage  Tien Le
    public ProductSorter(SortStrategy sortStrategy) {
        this.sortStrategy = sortStrategy;
    }

    2 usages  Tien Le
    public void setSortStrategy(SortStrategy sortStrategy) {
        this.sortStrategy = sortStrategy;
    }

    2 usages  Tien Le
    public void sort(List<Product> productList) {
        sortStrategy.sort(productList);
    }
}
```

Hình 2.13 Lớp ProductSorter chứa các phương thức tương tác với các chiến lược

```
productSorter = new ProductSorter(new SortByExpiryDateStrategy());
sortButton = binding.sortProductBtn;
sortButton.setOnClickListener(v -> {
    PopupMenu popup = new PopupMenu(getContext(), v);
    popup.getMenuInflater().inflate(R.menu.product_sort_pop_menu, popup.getMenu());
    popup.setOnMenuItemClickListener(item -> {
        if (item.getItemId() == R.id.expiryDateOptionItem) {
            productSorter.setSortStrategy(new SortByExpiryDateStrategy());
        } else {
            productSorter.setSortStrategy(new SortByNameStrategy());
        }
        mViewModel
                .getProductsByKitchenType()
                .observe(getViewLifecycleOwner(), data -> {
                    productSorter.sort(data);
                    adapter.updateProductList(data);
                });
```

Hình 2.14 Đoạn mã nguồn chiến lược sắp xếp được khởi tạo và thực thi trong ứng dụng

2.3 Template

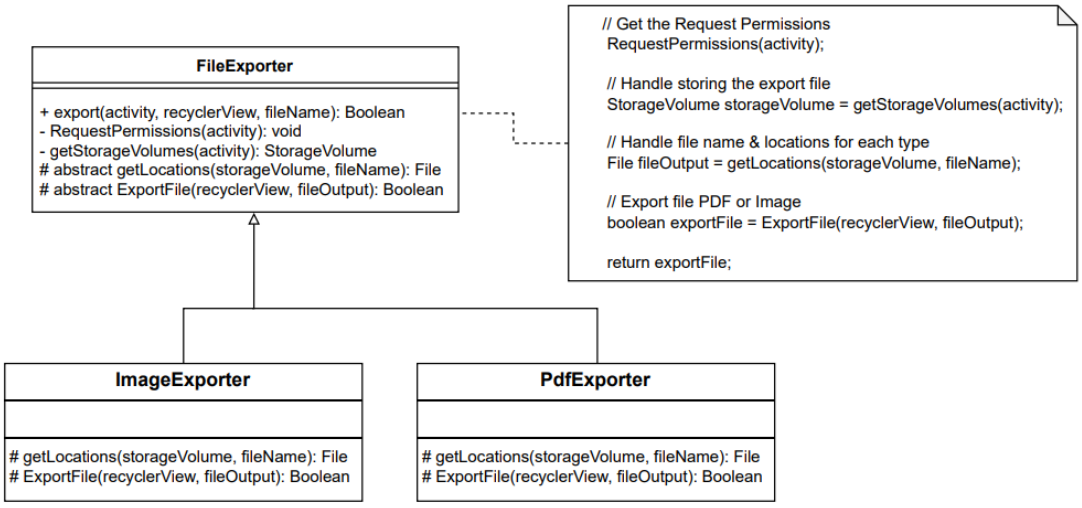
2.3.1 Giới thiệu và lý do áp dụng

Mẫu thiết kế Template được sử dụng trong ứng dụng nhằm mục đích xuất nguyên liệu cần chuẩn bị của từng món ăn, hỗ trợ người dùng dễ dàng mua sắm để nấu các món ăn cho từng bửa ăn đã được lên kế hoạch sẵn. Nguyên liệu sẽ được xuất ra hai dạng phổ biến là ảnh (png) và file (pdf), dễ dàng cho việc chia sẽ cho các thành viên trong gia đình.

Việc áp dụng mẫu thiết kế Template cho chức năng trên là hoàn toàn phù hợp, bởi nó cung cấp một khung sườn chung cho việc xuất ra danh sách các nguyên liệu thành phần, nhưng các lớp con có thể tùy chỉnh các bước cụ thể như file ảnh và pdf. Bên cạnh đó, Template thúc đẩy việc tái sử dụng mã bằng cách xác định các bước chung trong quy trình xuất danh sách nguyên liệu và triển khai chúng trong lớp cha (Hu 2023).

Ngoài ra, Template cho phép thêm các bước mới vào quy trình xuất danh sách nguyên liệu mà không ảnh hưởng đến các lớp con hiện có (tổng số lượng nguyên liệu cần thiết hoặc tính khả dụng của nguyên liệu tại địa phương), và dễ dàng bổ sung tính năng mới mà không cần thay đổi cấu trúc cơ bản của nó. Việc bảo trì mã ứng dụng sẽ trở nên dễ dàng bằng cách tập trung vào các bước chung ở lớp cha.

2.3.2 Sơ đồ lớp



Hình 2.15 Sơ đồ lớp áp dụng mẫu thiết kế Template

Mẫu thiết kế Template được áp dụng để tạo khung sườn của việc xuất nguyên liệu thành phần của từng món ăn ở dạng hình ảnh hoặc pdf. Ở lớp cha (FileExporter) chứa những bước chung của quy trình xuất file như yêu cầu cấp quyền ghi file ở bộ nhớ ngoài, xử lý việc lưu trữ tệp file. Những bước riêng như xác định vị trí lưu trên bộ nhớ, dạng file được xuất được thể hiện qua các lớp con (ImageExporter, PdfExporter).

2.3.3 Mã nguồn áp dụng mẫu thiết kế

```
public abstract class FileExporter {
    2 usages  datit
    public final boolean export(FragmentActivity activity , RecyclerView recyclerView, String fileName)
        // Get the Request Permissions
        RequestPermissions(activity);
        // Handle storing the export file
        StorageVolume storageVolume = getStorageVolumes(activity);
        // Handle file name & locations for each type
        File fileOutput = getLocations(storageVolume, fileName);
        // Export file PDF or Image
        boolean exportFile = ExportFile(recyclerView, fileOutput);
        return exportFile;
    }
    1 usage  datit
    private void RequestPermissions(FragmentActivity activity) {
        ActivityCompat.requestPermissions(activity,
                new String[]{READ_MEDIA_IMAGES, WRITE_EXTERNAL_STORAGE},
                PackageManager.PERMISSION_GRANTED);
    }
    1 usage  datit
    private StorageVolume getStorageVolumes(FragmentActivity activity) {
        StorageManager storageManager = (StorageManager) activity.getSystemService(STORAGE_SERVICE);
        StorageVolume storageVolume = storageManager.getStorageVolumes().get(0);
        return storageVolume;
    }
    1 usage  2 implementations  datit
    protected abstract File getLocations(StorageVolume storageVolume, String fileName);
    1 usage  2 implementations  datit
    protected abstract boolean ExportFile(RecyclerView recyclerView, File fileOutput);
```

Hình 2.16 Giao diện (abstract) FileExporter

```
public class ImageExporter extends FileExporter{
    1 usage  datit
    @Override
    protected File getLocations(StorageVolume storageVolume, String fileName) {
        File imageFile = null;
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.R) {
            imageFile = new File( pathname: storageVolume.getDirectory().getPath() + "/Download/" + fileName + ".png");
        }
        return imageFile;
    }
    1 usage  datit
    @Override
    protected boolean ExportFile(RecyclerView recyclerView, File fileOutput) {
        Bitmap bitmap = Bitmap.createBitmap( width: 1080,  height: 1920, Bitmap.Config.ARGB_8888);
        Canvas canvas = new Canvas(bitmap);
        canvas.drawColor(Color.WHITE);
        recyclerView.draw(canvas);
        try {
            FileOutputStream outputStream = new FileOutputStream(fileOutput);
            bitmap.compress(Bitmap.CompressFormat.PNG,  quality: 100, outputStream);
            outputStream.flush();
            outputStream.close();
            return true;
        } catch (IOException e) {
            return false;
        }
    }
}
```

Hình 2.17 Giao diện ImageExporter

```
public class PdfExporter extends FileExporter {
    1 usage   datit
    @Override
    protected File getLocations(StorageVolume storageVolume, String fileName) {
        File filePDFOutput = null;
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.R) {
            filePDFOutput = new File( pathname: storageVolume.getDirectory().getPath() + "/Download/" + fileName + ".pdf");
        }
        return filePDFOutput;
    }
    1 usage   datit
    @Override
    protected boolean ExportFile(RecyclerView recyclerView, File fileOutput) {
        PdfDocument pdfDocument = new PdfDocument();
        PdfDocument.PageInfo pageInfo = new PdfDocument.PageInfo.Builder( pageWidth: 1080,  pageHeight: 1920,  pageNumber: 1).create();
        PdfDocument.Page page = pdfDocument.startPage(pageInfo);
        recyclerView.draw(page.getCanvas());
        pdfDocument.finishPage(page);
        try {
            pdfDocument.writeTo(new FileOutputStream(fileOutput));
            pdfDocument.close();
            return true;
        } catch (IOException e) {
            pdfDocument.close();
            return false;
        }
    }
}
```

Hình 2.18 Giao diện PdfExporter

```
private void templateExportFile(RecyclerView recyclerView, String fileName) {
    Button downloadBtn;
    downloadBtn = binding.downloadBtn;

    downloadBtn.setOnClickListener(v -> {
        PopupMenu popup = new PopupMenu(getContext(), v);
        popup.getMenuInflater().inflate(R.menu.download_ingredients, popup.getMenu());
        popup.setOnMenuItemClickListener(item -> {
            if (item.getItemId() == R.id.pdfOptionItem) {
                FileExporter filePDF = new PdfExporter();
                if(filePDF.export(getActivity(),recyclerView, fileName))
                    Toast.makeText(getContext(),  text: "Complete PDF export, check on Download storage", Toast.LENGTH_SHORT)
                else Toast.makeText(getContext(),  text: "Failure PDF export", Toast.LENGTH_SHORT).show();
            } else {
                FileExporter fileIMG = new ImageExporter();
                if(fileIMG.export(getActivity(),recyclerView, fileName))
                    Toast.makeText(getContext(),  text: "Complete IMG export, check on Download storage", Toast.LENGTH_SHORT)
                else Toast.makeText(getContext(),  text: "Failure IMG export", Toast.LENGTH_SHORT).show();
            }
            return true;
        });
        popup.show();
    });
}
```

Hình 2.19 Xuất nguyên liệu thành phần dựa vào lựa chọn của người dùng

2.4 Factory

2.4.1 Giới thiệu và lý do áp dụng

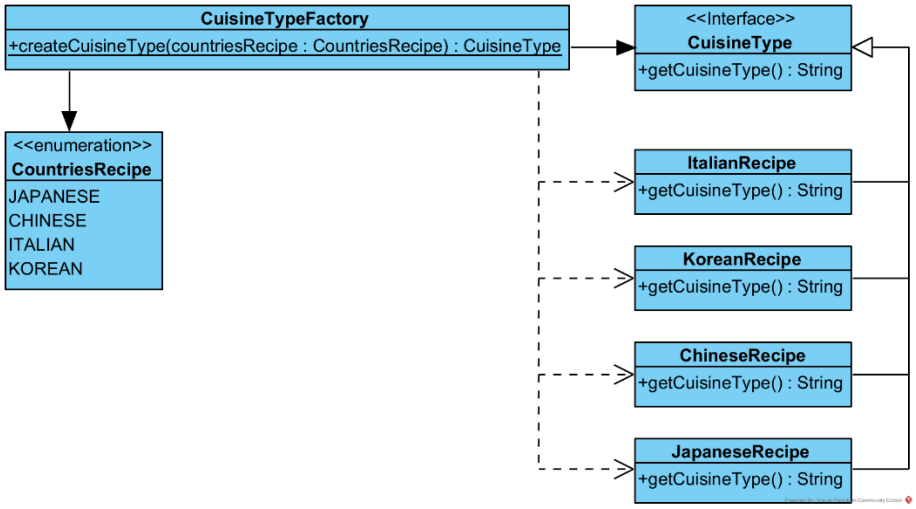
Mẫu thiết kế Factory được sử dụng để tạo ra các công thức món ăn dựa trên quốc gia và khu vực, chẳng hạn như công thức từ Ý, Hàn Quốc, và Nhật Bản,…

Factory được chọn để đóng gói việc tạo các đối tượng công thức dựa trên các quốc gia hoặc khu vực khác nhau. Bằng cách sử dụng các phương thức và lớp factory, ứng dụng tập trung logic tạo đối tượng, giúp việc thêm các loại công thức mới dễ dàng hơn trong tương lai mà không cần sửa đổi mã nguồn (Joshi 2016). Điều này nâng cao khả năng bảo trì, mở rộng và thúc đẩy việc phân tách các nhiệm vụ một cách rõ ràng bằng cách ủy quyền trách nhiệm tạo đối tượng công thức cho các lớp factory, giúp ứng dụng tập trung vào việc sử dụng các đối tượng được tạo chứ không phải quan tâm các chi tiết khởi tạo chúng (Hu 2023).

Tổng hợp lại, sau khi áp dụng mẫu thiết kế Factory, dự án đạt được những lợi ích sau:

- Tính đóng gói các logic khởi tạo: Factory đóng gói việc tạo các đối tượng phức tạp (công thức món ăn) trong các lớp factory riêng biệt. Điều này thúc đẩy tính đóng gói và phân tách mối quan tâm, vì phía sử dụng các đối tượng này không cần biết các chi tiết phức tạp về cách các đối tượng được khởi tạo.
- Tính mở rộng: Các loại công thức mới dựa trên quốc gia và khu vực được thêm vào một cách dễ dàng hơn. Lúc này, phía sử dụng vẫn tách biệt khỏi quá trình khởi tạo và tập trung vào việc sử dụng các đối tượng này.
- Cấu hình tập trung: Factory tập trung logic cấu trình và khởi tạo, giúp việc quản lý và bảo trì dễ dàng hơn. Bất kỳ thay đổi nào đối với quá trình tạo có thể được thực hiện trong các lớp factory, tránh phát sinh những sửa đổi trong mã nguồn của những nơi sử dụng các đối tượng này (các công thức món ăn).

2.4.2 Sơ đồ lớp



Hình 2.20 Sơ đồ lớp áp dụng mẫu thiết kế Factory

Mẫu thiết kế factory đơn giản được áp dụng để tạo các đối tượng là công thức món ăn tại các quốc gia là Ý, Hàn Quốc, Nhật Bản và Trung Quốc. Các lớp này được khái quát bởi giao diện (interface) CuisineType. Lớp factory tên CuisineTypeFactory chứa phương thức tạo các đối tượng dựa vào tham số truyền vào có kiểu là CountriesRecipe – là một enum trong Java bao gồm JAPANESE (món ăn tại Nhật Bản), CHINESE (món ăn tại Trung Quốc), ITALIAN (món ăn tại Ý), KOREAN (món ăn tại Hàn Quốc).

2.4.3 Mã nguồn áp dụng mẫu thiết kế

```
public interface CuisineType {
    1 usage  4 implementations  Tien Le
    String getCuisineType();
}
```

Hình 2.21 Giao diện (interface) CuisineType

```
1 usage  Tien Le
public class JapaneseRecipe implements CuisineType {
    1 usage  Tien Le
    public JapaneseRecipe() {}

    1 usage  Tien Le
    @Override
    public String getCuisineType() { return "japanese"; }
}
```

Hình 2.22 Lớp JapaneseRecipe

```
public class ChineseRecipe implements CuisineType {
    1 usage  Tien Le
    public ChineseRecipe() {}

    1 usage  Tien Le
    @Override
    public String getCuisineType() { return "chinese"; }
}
```

Hình 2.23 Lớp ChineseRecipe

```
public class ItalianRecipe implements CuisineType {
    1 usage  Tien Le
    public ItalianRecipe() {}

    1 usage  Tien Le
    @Override
    public String getCuisineType() { return "italian"; }
}
```

Hình 2.24 Lớp ItalianRecipe

```
public class KoreanRecipe implements CuisineType {
    1 usage  Tien Le
    public KoreanRecipe() {}

    1 usage  Tien Le
    @Override
    public String getCuisineType() { return "Korean"; }
}
```

Hình 2.25 KoreanRecipe

```
public class CuisineTypeFactory {
    1 usage  Tien Le
    public static CuisineType createCuisineType(CountriesRecipe countriesRecipe) {
        if (countriesRecipe == CountriesRecipe.JAPANESE) {
            return new JapaneseRecipe();
        } else if (countriesRecipe == CountriesRecipe.CHINESE) {
            return new ChineseRecipe();
        } else if (countriesRecipe == CountriesRecipe.ITALIAN) {
            return new ItalianRecipe();
        } else {
            return new KoreanRecipe();
        }
    }
}
```

Hình 2.26 Lớp CuisineTypeFactory

```
public enum CountriesRecipe {
    2 usages
    JAPANESE,
    2 usages
    CHINESE,
    2 usages
    ITALIAN,
    1 usage
    KOREAN
}
```

Hình 2.27 Enumeration CountriesRecipe

```
if (args.getSerializable(RecipesFragment.ARG_COUNTRY) != null) {
    country_name = (CountriesRecipe) args.getSerializable(RecipesFragment.ARG_COUNTRY);
    mViewModel.prepareData(CuisineTypeFactory.createCuisineType(country_name));
}
```

Hình 2.28 Ví dụ về nơi gọi lớp factory nhằm tạo đối tượng chứa thông tin đất nước/khu vực của công thức món lấy về

```
public void prepareData(CuisineType cuisineType) {
    EdamamAPIService.getInstance().loadDishBaseOnCountry(
            APP_ID, APP_KEY,
            type: "public",
            datit *
            cuisineType.getCuisineType()).enqueue(new Callback<SearchEdamamRecipes>() {
        datit *
        @Override
        public void onResponse(Call<SearchEdamamRecipes> call, Response<SearchEdamamRecipes> response)
            recipe = new ArrayList<>(Arrays.asList(response.body().getFoodRecipes()));
            mRecipes.postValue(Arrays.asList(response.body().getFoodRecipes()));
        }
```

Hình 2.29 Cách đối tượng cung cấp thông tin về đất nước/khu vực để lấy danh sách các công thức món ăn thông qua API

Các công thức món ăn của ứng dụng được lấy từ một nhà cung cấp thứ ba thông qua API của họ. Thế nên, các đối tượng được tạo ra thông qua lớp factory chứa thông tin đến đất nước/khu vực cần được lấy về các công thức món ăn; các thông tin này sẽ được sử dụng để tương tác với API và lấy về các công thức với đất nước/khu vực tương ứng.

2.5 Command

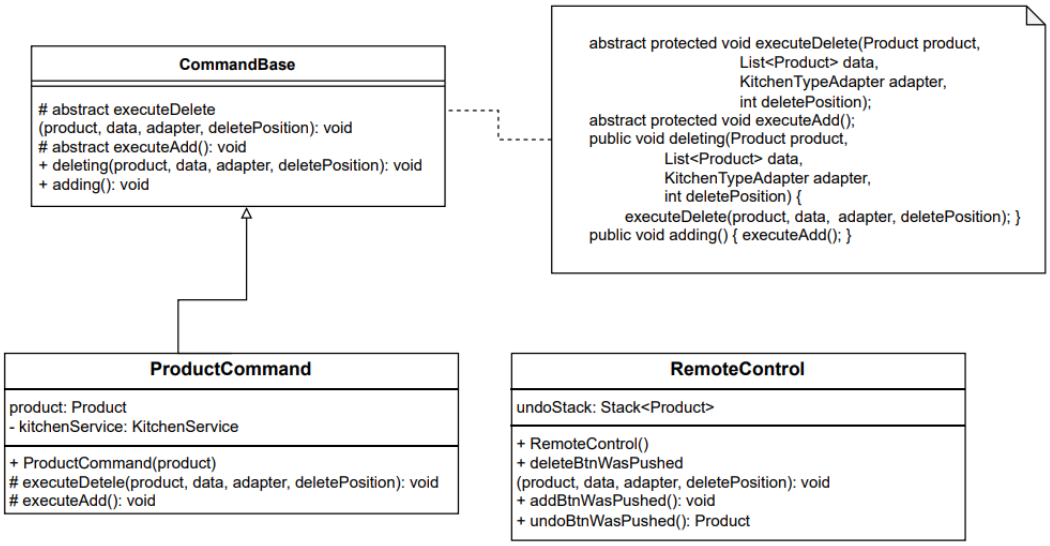
2.5.1 Giới thiệu và lý do áp dụng

Mẫu thiết kế Command được sử dụng trong ứng dụng nhằm hỗ trợ các hoạt động có thể được hoàn tác lại, chẳng hạn khi xóa một nguyên liệu ra khỏi tủ lạnh.

Việc áp dụng mẫu thiết kế Command cho chức năng trên hoàn toàn hợp lý bởi vì Command giúp việc tách biệt logic xử lý khỏi giao diện người dùng, cho phép dễ dàng thay đổi cách thức thực hiện các hành động mà không ảnh hưởng đến phần còn lại của ứng dụng. Bên cạnh đó, Command dễ dàng lưu trữ lịch sử các hoạt động được thực hiện bởi người dùng, từ đó dễ dàng hoàn tác hoặc làm lại các thao tác trước đó (Hu 2023).

Ngoài ra, mẫu thiết kế còn được áp dụng vào việc thêm hay xóa một nguyên liệu thành phần ra khỏi tủ lạnh, nhằm tăng hiệu quả trong việc thực hiện hoạt động phức tạp khi liên quan đến truy vấn cơ sở dữ liệu từ Firebase, chẳng hạn như hoàn tác sau khi xóa (tương ứng với chức năng thêm), hoặc làm lại sau khi hoàn tác (tương ứng với chức năng xóa); sau khi xóa một nguyên liệu, một lời nhắc hoàn tác sẽ được hiện lên trong vòng 5 giây, nếu người dùng ấn hoàn tác, thì nguyên liệu sẽ được thêm vào lại, tuy nhiên sẽ không hiện lời nhắc làm lại bởi vì người dùng có thể dễ dàng xóa một nguyên liệu, vì khi xóa luôn kèm một lời nhắc hoàn tác.

2.5.2 Sơ đồ lớp



Hình 2.28 Sơ đồ lớp áp dụng mẫu thiết kế Command

Lớp CommandBase chứa các phương thức trừu tượng thực thi (execute) được thể hiện cụ thể ở lớp con (ProductCommand). Việc triển khai được thôngqua lớp RemoteControl nhằm cụ thể hóa các nút hiển thị trên ứng dụng.

2.5.3 Mã nguồn áp dụng mẫu thiết kế

```
public abstract class CommandBase {
    1 usage  datit
    public CommandBase() {}
    1 usage  1 implementation  datit
    abstract protected void executeDelete(Product product,
                                          List<Product> data,
                                          KitchenTypeAdapter adapter,
                                          int deletePosition);
    1 usage  1 implementation  datit
    abstract protected void executeAdd();
    1 usage  datit
    public void deleting(Product product,
                         List<Product> data,
                         KitchenTypeAdapter adapter,
                         int deletePosition) { executeDelete(product, data, adapter, deletePosition); }
    1 usage  datit
    public void adding() { executeAdd(); }
}
```

Hình 2.30 Giao diện (abstract) CommandBase

```
public class ProductCommand extends CommandBase {
    5 usages
    Product product;
    no usages
    String newId;
    1 usage
    private KitchenService kitchenService = new SharedKitchenProxy();
    2 usages  datit
    public ProductCommand(Product product) { this.product = product; }
    1 usage  datit
    @Override
    protected void executeDelete(Product product,
                                 List<Product> data,
                                 KitchenTypeAdapter adapter,
                                 int deletePosition) {
        kitchenService.removeProductItem(product, data, adapter, deletePosition);
    }
    1 usage  datit
    @Override
    protected void executeAdd() {
        FirebaseFirestore db = FirebaseFirestore.getInstance();
        String productId = product.getProductId();
        if(productId != null && !productId.isEmpty()) {
            db.collection( collectionPath: "products") CollectionReference
                    .document(product.getProductId()) DocumentReference
                    .set(product);
        } else {
            db.collection( collectionPath: "products").add(product);
        }
    }
}
```

Hình 2.31 Giao diện ProductCommand

```
public class RemoteControl {
    4 usages
    Stack<Product> undoStack;
    3 usages  datit
    public RemoteControl() { undoStack = new Stack<>(); }
    1 usage  datit
    public void deleteBtnWasPushed(Product product,
                                   List<Product> data,
                                   KitchenTypeAdapter adapter,
                                   int deletePosition) {
        CommandBase commandBases = new ProductCommand(product);
        commandBases.deleting(product, data, adapter, deletePosition);
        undoStack.push(product);
    }
    2 usages  datit
    public Product addBtnWasPushed(Product product) {
        CommandBase commandBases = new ProductCommand(product);
        commandBases.adding();
        return product;
    }
    1 usage  datit
    public Product undoBtnWasPushed() {
        Product product = undoStack.pop();
        if (product != null) {
            addBtnWasPushed(product);
            undoStack.remove(product);
        }
        return product;
    }
}
```

Hình 2.32 Giao diện RemoteControl

```
holder.getShowMoreBtn().setOnClickListener(v ->  {
    PopupMenu popup = new PopupMenu(v.getContext(), v);
    popup.getMenuInflater().inflate(R.menu.more_option_product, popup.getMenu());
    popup.setOnMenuItemClickListener(item -> {
        if (item.getItemId() == R.id.deleteProduct) {
            int productIndex = data.indexOf(curProduct);
            remoteControl.deleteBtnWasPushed(curProduct, data,  adapter: this, productIndex);
            Snackbar sbUndo = Snackbar.make(v,  text: "Successfully deleted " + curProduct.getTitle(), Snackbar.LENGTH_LONG)
            datit
            sbUndo.setAction( text: "UNDO", new View.OnClickListener() {
                datit
                @Override
                public void onClick(View v) {
                    Product product = remoteControl.undoBtnWasPushed();
                    if(!data.contains(product)) {
                        data.add(productIndex, product);
                        notifyItemInserted(productIndex);
                    }
                }
            }).show();
```

Hình 2.33 Command áp dụng khi xóa và thêm gián tiếp qua nút hoàn tác

```
confirmAddingBtn.setOnClickListener(v -> {
    if (validateAllInput()) {
        categorizes = new ArrayList<>();
        checkedChipIds = chipCategoryGroup.getCheckedChipIds();
        Chip selectedChip;
        for (Integer id : checkedChipIds) {
            selectedChip = chipCategoryGroup.findViewById(id);
            categorizes.add(selectedChip.getText().toString());
        }
        mProduct.setTitle(productTitleTextInput.getText().toString());
        mProduct.setBrand(brandTextInput.getText().toString());
        mProduct.setBarcode(barcodeTextInput.getText().toString());
        mProduct.setProductCategorizes(categorizes);
        mProduct.setPantry(selectedPantry);
        mProduct.setKitchenId(MainActivity.getCurKitchenId());

        remoteControl.addBtnWasPushed(mProduct);
        sendResultActivity( isSuccess: true);
    }
});
```

Hình 2.34 Command áp dụng khi thêm trực tiếp qua nút thêm trên màn hình

2.6 Adapter

2.6.1 Giới thiệu và lý do áp dụng

Mẫu thiết kế Adapter được áp dụng để sử dụng kết quả trả về là các công thức món ăn từ API của bên thứ ba và có định dạng khác so với định dạng được sử dụng trong ứng dụng hiện tại.

Adapter được chọn để tạo điều kiện tích hợp các API bên ngoài với các định dạng dữ liệu khác nhau vào kiến trúc hiện có của ứng dụng. Adapter đóng vai trò trung giam, chuyển đổi giao diện các API bên ngoài thành định dạng tương thích với ứng dụng hiện tại, do đó đảm bảo giao tiếp liền mạch với các chi tiết cụ thể của API bên ngoài, thúc đẩy khả năng tương tác và cho phép bảo trì và cập nhật dễ hơn (Hu 2023). Mẫu thiết kế này cũng tuân theo Nguyên tắc Dependency Inversion) bằng cách tách rời ứng dụng khỏi các chi tiết của dịch vụ bên ngoài, cho phép kiểm thử và sửa đổi trong tương lai dễ dàng hơn (Anon n.d.-a).

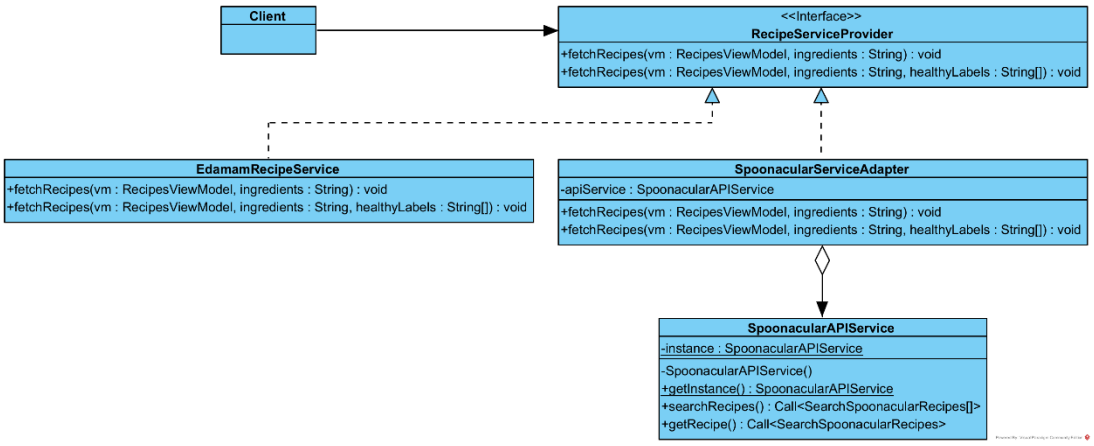
Tổng hợp lại, sau khi áp dụng mẫu thiết kế Adapter, dự án đạt được những lợi ích sau:

- Tính tương tác: Adapter đóng vai trò cầu nối giữa các giao diện không tương thích, cho phép ứng dụng giao tiếp với các hệ thống bên ngoài một cách liền

mạch. Adapter chuyển đổ giao diện của API bên thứ ba thành định dạng tương thích với giao diện hiện tại mà ứng dụng đang sử dụng, đảm bảo tích hợp và trao đổi dữ liệu trơn tru.

- Tính linh hoạt: Mô hình Adapter thích ứng với những thay đổi trong các API hoặc dịch vụ bên ngoài mà không ảnh hưởng đến chức năng cốt lõi của ứng dụng. Nếu định dạng hoặc giao diện của dịch vụ bên ngoài thay đổi, thì chỉ cần cập nhật phần hiện thực của adapter, trong khi các phần còn lại của ứng dụng không bị ảnh hưởng.
- Tính tái sử dụng: Adapter thúc đẩy khả năng tái sử dụng mã nguồn bằng cách đóng gói logic tích hợp trông các thành phần có thể tái sử dụng. Sau khi được hiện thực, adapter có thể được sử dụng lại trên các thành phần khác nhau của ứng dụng hoặc trong các dự án khác có yêu cầu tích hợp tương tự.

2.6.2 Sơ đồ lớp



Hình 2.35 Sơ đồ lớp áp dụng mẫu thiết kế Adapter

Ban đầu ứng dụng nhận thông tin công thức món ăn với định dạng là các lớp với tên RootObjectModel và RecipeModel; các lớp này chứa các thuộc tính tương ứng với kết quả được trả về từ bên thứ ba tên là Edamam thông qua API. Các lớp này được sử dụng rộng rãi hầu hết trong mã nguồn của ứng dụng. Thế nên, thêm một nguồn công thức món ăn mới với định dạng khác với API của Edamam gây khó khăn trong việc chỉnh sửa mã nguồn của những nơi sử dụng các lớp với định dạng ban đầu.

Adapter được áp dụng bằng cách khái quát cách thức lấy công thức món ăn từ các nguồn thành giao diện (interface) RecipeServiceProvider. Giao diện này yêu cầu kết quả trả về là các lớp tương thích với định dạng mà ứng dụng sử dụng hiện tại. Một lớp adapter được tạo ra tên là SpoonacularServiceAdapter hiện thực giao diện RecipeServiceProvider. Lớp adapter này sẽ làm việc với SpoonacularAPIService là lớp chứa các cách thức giao tiếp với API của nguồn cung cấp công thức mới (nguồn mới tên là Spoonacular) và thực hiện một số bước chuyển đổi về định dạng mà ứng dụng yêu cầu.

2.6.3 Mã nguồn áp dụng mẫu thiết kế

```
public interface RecipeServiceProvider {
    1 usage  2 implementations  new *
    void fetchRecipes(RecipesViewModel vm, String ingredients);
    1 usage  2 implementations  new *
    void fetchRecipes(RecipesViewModel vm, String ingredients, String[] healthyLabels);
}
```

Hình 2.36 Giao diện (interface) RecipeServiceProvider

```
public class SpoonacularServiceAdapter implements RecipeServiceProvider {
    1 usage
    private final SpoonacularAPIService apiService = SpoonacularAPIService.getInstance();

    1 usage  new *
    @Override
    public void fetchRecipes(RecipesViewModel vm, String ingredients) {
        List<SearchSpoonacularRecipes> spoonacularRecipesList = new ArrayList<>();
        new *
        apiService.searchRecipes(ingredients).enqueue(new Callback<SearchSpoonacularRecipes[]>() {
            new *
            @Override
            public void onResponse(
                    @NonNull Call<SearchSpoonacularRecipes[]> call,
                    @NonNull Response<SearchSpoonacularRecipes[]> response) {
                if (response.body() != null) {
                    spoonacularRecipesList.addAll(Arrays.asList(response.body()));
                    List<RootObjectModel> covertedList = new ArrayList<>();
                    for (SearchSpoonacularRecipes spoonacularRecipe: spoonacularRecipesList) {...}
                    vm.getRecipes().postValue(covertedList);
                    vm.updateRecipes(covertedList);
                }
            }
```

Hình 2.37 Lớp adapter tên SpoonacularServiceAdapter

```
public class SpoonacularAPIService implements SpoonacularService {
    4 usages
    private static SpoonacularAPIService instance;
    3 usages
    private final SpoonacularService spoonacularService;
    1 usage
    private static final Gson gson = new GsonBuilder()
            .setLenient()
            .create();

    1 usage  new *
    private SpoonacularAPIService() {...}

    new *
    public static SpoonacularAPIService getInstance() {...}

    2 usages  new *
    @Override
    public Call<SearchSpoonacularRecipes[]> searchRecipes(String ingredients) {
        return spoonacularService.searchRecipes(ingredients);
    }

    2 usages  new *
    @Override
    public Call<SearchSpoonacularRecipes> getRecipe(String id) {
        return spoonacularService.getRecipe(id);
    }
}
```

Hình 2.38 Lớp SpoonacularAPIService chứa cách thức giao tiếp với API mới

```
// Existing API service
recipeServiceProvider = new EdamamRecipeService();
recipeServiceProvider.fetchRecipes( vm: this, query, healthyLabels);

// New API service with different JSON format
recipeServiceProvider = new SpoonacularServiceAdapter();
recipeServiceProvider.fetchRecipes( vm: this, query);
```

Hình 2.39 Ví dụ về cách lấy công thức món ăn từ API với adapter

2.7 Prototype

2.7.1 Giới thiệu và lý do áp dụng

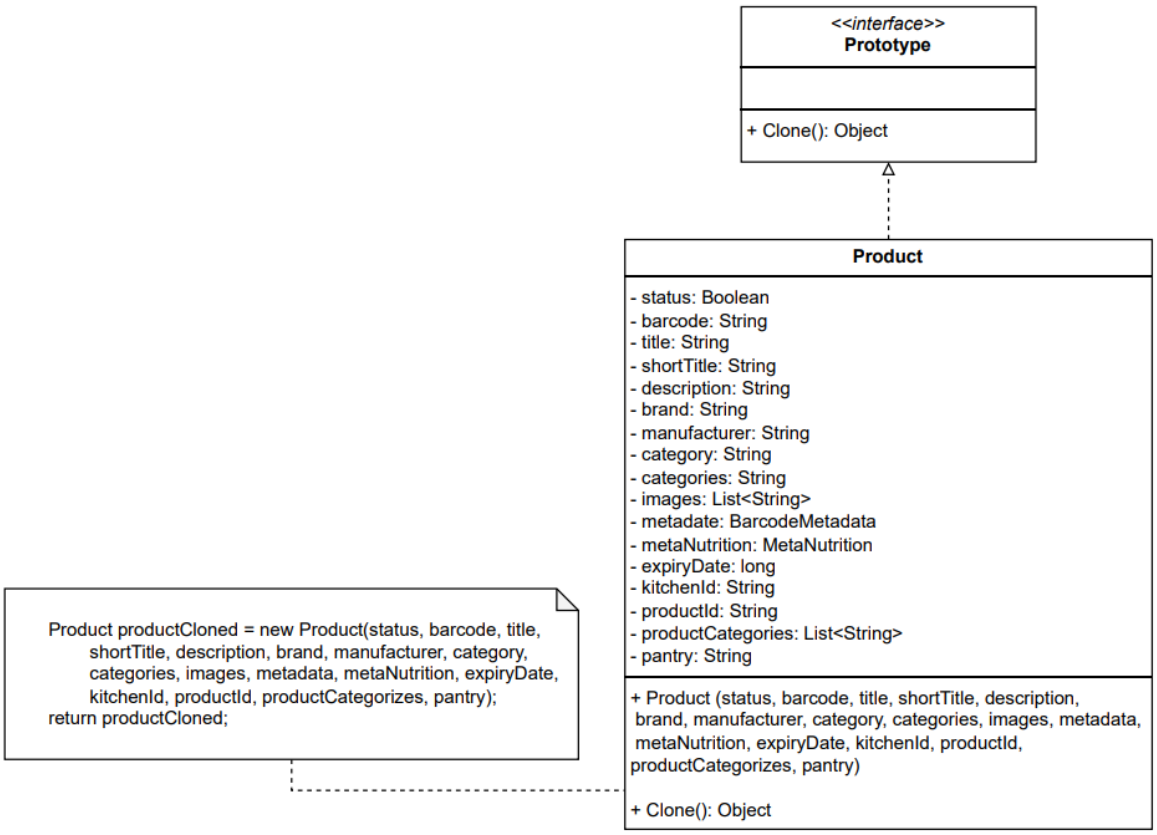
Mẫu thiết kế Prototype được sử dụng trong ứng dụng nhằm mục đích tạo ra một bản sao của nguyên liệu thành phần đã có trong tủ lạnh, khi người dùng mua một nguyên liệu

giống với nguyên liệu đã mua trước đó vài ngày, nhưng nguyên liệu mới có thể tươi hơn, hoặc có hạn sử dụng khác với nguyên liệu trước đó, việc thêm lại nguyên liệu sẽ mất nhiều thời gian nhập liệu hoặc quét mã, và phải chọn lại các phân loại (loại tủ, loại thực phẩm, ...).

Việc áp dụng mẫu thiết kế cho chức năng trên là hoàn toàn hợp lý, bởi vì Prototype cho phép người dùng tạo bản sao của các nguyên liệu có sẵn, điều này giúp tăng trải nghiệm của người dùng, đặc biệt là khi trong tủ có rất nhiều nguyên liệu giống nhau, chỉ khác nhau về độ tươi hoặc hạn sử dụng.

Bên cạnh đó, Prototype giúp đảm bảo tính nhất quán của dữ liệu bản sao so với dữ liệu gốc, giúp hiệu quả trong việc giảm thiểu sai sót do nhập liệu thủ công (Joshi 2016). Ngoài ra, nó còn cho phép người dùng dễ dàng chỉnh sửa thông tin của bản sao dựa trên dữ liệu gốc, nhưng không thay đổi đến giá trị của dữ liệu gốc.

2.7.2 Sơ đồ lớp



Hình 2.40 Sơ đồ lớp áp dụng mẫu thiết kế Prototype

Lớp Prototype chứa phương thức Clone(). Lớp Product thực thi phương thức được cung cấp bởi Prototype nhằm nhân bản chính nó. Lớp này chính là thể hiện cụ thể của phương thức Clone(), ở đây Product cho phép nhân bản với cách sao chép là Sallow bởi vì Product chứa một số thuộc tính là một đối tượng, các đối tượng này sinh ra bởi mã vạch trên sản phẩm, sẽ là cố định.

2.7.3 Mã nguồn áp dụng mẫu thiết kế

```
public interface Prototype {
    1 usage  1 implementation  datit
    Object Clone();
}
```

Hình 2.41 Giao diện (interface) Prototype

```
public class Product implements Prototype {
    new *
    public Product(boolean status, String barcode, String title, String shortTitle, String description,
                   String brand, String manufacturer, String category, String categories, List<String> images,
                   BarcodeMetadata metadata, MetaNutrition metaNutrition, long expiryDate, String kitchenId,
                   String productId, List<String> productCategorizes, String pantry) {
        this.status = status;
        this.barcode = barcode;
        this.title = title;
        this.shortTitle = shortTitle;
        this.description = description;
        this.brand = brand;
        this.manufacturer = manufacturer;
        this.category = category;
        this.categories = categories;
        this.images = images;
        this.metadata = metadata;
        this.metaNutrition = metaNutrition;
        this.expiryDate = expiryDate;
        this.kitchenId = kitchenId;
        this.productId = productId;
        this.productCategorizes = productCategorizes;
        this.pantry = pantry;
    }
```

Hình 2.42 Lớp Product phức tạp được áp dụng mẫu thiết kế Prototype

```
// Shallow Clone
1 usage  datit
@Override
public Object Clone() {
    Product productCloned = new Product(status, barcode, title, shortTitle, description, brand, manufacturer,
            category, categories, images, metadata, metaNutrition, expiryDate,
            kitchenId, productId, productCategorizes, pantry);
    return productCloned;
}
```

Hình 2.43 Phương thức Clone

```
} else if (item.getItemId() == R.id.cloneableProduct) {
    final Calendar c = Calendar.getInstance();
    int mYear = c.get(Calendar.YEAR);
    int mMonth = c.get(Calendar.MONTH);
    int mDay = c.get(Calendar.DAY_OF_MONTH);
    DatePickerDialog datePickerDialog = new DatePickerDialog(v.getContext(),
            datit
            new DatePickerDialog.OnDateSetListener() {
                1 usage  datit
                @Override
                public void onDateSet(android.widget.DatePicker view, int year,
                                      int monthOfYear, int dayOfMonth) {
                    Calendar selectedDate = Calendar.getInstance();
                    selectedDate.set(year, monthOfYear, dayOfMonth);

                    // Create a clone product
                    Product productCloned = (Product) curProduct.Clone();

                    // Reset Id for a new product
                    productCloned.setProductId(null);

                    // Set new value in clone product (expiry date)
                    productCloned.setExpiryDate(selectedDate.getTimeInMillis());
```

Hình 2.44 Prototype áp dụng khi nhân bản một nguyên liệu (1)

```
                    db = FirebaseFirestore.getInstance();
                    db.collection( collectionPath: "products") CollectionReference
                            .add(productCloned) Task<DocumentReference>
                            datit
                            .addOnCompleteListener(new OnCompleteListener<DocumentReference>() {
                                datit
                                @Override
                                public void onComplete(@NonNull Task<DocumentReference> task) {
                                    if (task.isSuccessful()) {
                                        // Get the clone product after completely adding
                                        DocumentReference documentReference = task.getResult();

                                        // Get index added of clone product
                                        int indexLast = data.size();

                                        // Add Id (firebase) to clone product to add on list<Product>
                                        String newId = documentReference.getId();
                                        productCloned.setProductId(newId);
                                        data.add(productCloned);
                                        notifyItemInserted(indexLast);
                                    }
                                }
                            });
                }
            }, mYear, mMonth, mDay);
    datePickerDialog.getDatePicker().setMinDate(System.currentTimeMillis());
    datePickerDialog.show();
}
return true;
```

Hình 2.45 Prototype áp dụng khi nhân bản một nguyên liệu (2)

2.8 Proxy

2.8.1 Giới thiệu và lý do áp dụng

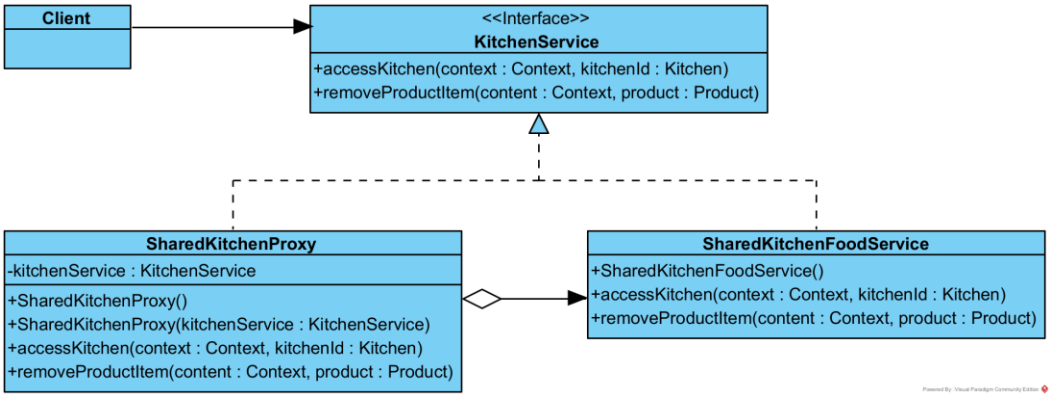
Mẫu thiết kế Proxy được sử dụng để kiểm soát quyền truy cập vào tủ đựng thực phẩm được chia sẻ bởi người dùng.

Proxy được sử dụng để cung cấp một đối tượng thay thế để kiểm soát quyền truy cập vào tài nguyên tủ đựng được chia sẻ. Điều này cho phép ứng dụng triển khai kiểm soát quyền truy cập một cách minh bạch mà không cần sửa đổi chức năng cốt lõi của tủ đựng chung (Hu 2023). Ngoài ra, Proxy hỗ trợ Nguyên tắc Single Responsibility bằng cách đóng gói logic kiểm soát quyền truy cập trong các đối tượng proxy, từ đó cải thiện cách thức tổ chức mã nguồn (Anon n.d.-b).

Tổng hợp lại, sau khi áp dụng mẫu thiết kế Adapter, dự án đạt được những lợi ích sau:

- Kiểm soát truy cập: Proxy cung cấp một đối tượng thay thế để kiểm soát quyền truy cập vào các tủ đựng thực phẩm chung. Bằng cách can thiệp vào các yêu cầu từ phía sử dụng, Proxy có thể thực thi việc xác thực người dùng hoặc thực hiện xác nhận bổ sung trước khi cấp quyền truy cập vào tủ đựng chung.
- Truy cập minh bạch: Proxy duy trì giao diện giống như các đối tượng thực mà chúng đại diện, đảm bảo minh bạch trong truy cập cho phía sử dụng. Điều này cho phép Proxy thay thế các đối tượng thực một cách một cách liền mạch mà không ảnh hưởng đến mã nguồn, tạo điều kiện thuận lợi cho thiết kế và triển khai mô-đun.

2.8.2 Sơ đồ lớp



Hình 2.46 Sơ đồ lớp áp dụng mẫu thiết kế Proxy

Lớp SharedKitchenFoodService chứa cách thức người dùng truy cập và tương tác với nhà bếp được chia sẻ bởi những người dùng khác. Nhằm kiểm soát quyền truy cập vào các nhà bếp chung này, mẫu thiết kế Proxy được áp dụng bằng cách thêm một lớp tên là SharedKitchenProxy chứa cách thức xác nhận thêm quyền của người dùng, và hai lớp này được khái quát bởi giao diện (interface) KitchenService. Lớp Client đại diễn cho những nơi liên quan sử dụng dịch vụ này và lớp sẽ tương tác với lớp SharedKitchenProxy để thực hiện việc kiểm soát quyền truy cập.

2.8.3 Mã nguồn áp dụng mẫu thiết kế

```
public interface KitchenService {
    2 usages  2 implementations  new *
    void accessKitchen(Context context, String kitchenId);
    2 usages  2 implementations  new *
    void removeProductItem(
            Context context,
            Product product,
            List<Product> data,
            KitchenTypeAdapter adapter,
            int deletePosition);
}
```

Hình 2.47 Giao diện (interface) KitchenService

```
public class SharedKitchenFoodService implements KitchenService {
    2 usages
    FirebaseFirestore db;

    1 usage  new *
    public SharedKitchenFoodService() { db = FirebaseFirestore.getInstance(); }

    2 usages  new *
    @Override
    public void accessKitchen(Context context, String kitchenId) {...}

    2 usages  new *
    @Override
    public void removeProductItem(
            Context context,
            Product product,
            List<Product> data,
            KitchenTypeAdapter adapter,
            int deletePosition) {...}
}
```

Hình 2.48 Lớp SharedKitchenFoodService

```
public class SharedKitchenProxy implements KitchenService {
    4 usages
    FirebaseFirestore db;
    4 usages
    private KitchenService kitchenService;

    2 usages  new *
    public SharedKitchenProxy() {...}

    no usages  new *
    public SharedKitchenProxy(KitchenService kitchenService) {...}

    2 usages  new *
    @Override
    public void accessKitchen(Context context, String kitchenId) {
        db.collection( collectionPath: "kitchens") CollectionReference
                .whereArrayContains( field: "subOwnerIds", MainActivity.getCurUser().getEmail())
                .get() Task<QuerySnapshot>
                .addOnCompleteListener(task -> {
                    if (task.isSuccessful()) {
                        kitchenService.accessKitchen(context, kitchenId);
                    } else {
                        Toast.makeText(context,  text: "You are not authorized to access this",
                    }
                });
    }
```

Hình 2.49 Lớp SharedKitchenProxy

```
kitchenService = new SharedKitchenProxy();
holder.getItemView().setOnClickListener(v -> {
    kitchenService.accessKitchen(v.getContext(), kitchen.getKitchenId());
```

Hình 2.50 Ví dụ về cách sử dụng lớp proxy để kiểm soát truy cập

2.9 Model - View – ViewModel (MVVM)

2.9.1 Giới thiệu và lý do áp dụng

Model - View – ViewModel (MVVM) được áp dụng để tách logic hiển thị UI (View) khỏi logic nghiệp vụ cốt lõi (ViewModel).

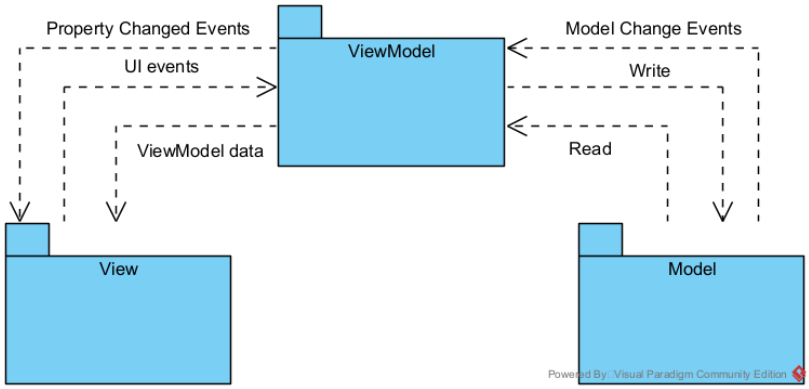
MVVM được chọn để cung cấp sự tách biệt rõ ràng giữa giao diện người dùng (View), đối tượng (Model) và logic hiển thị (ViewModel). Bằng cách cấu trúc ứng dụng kiến trúc MVVM, mã nguồn của dự án tăng khả năng mở rộng, dễ bảo trì và dễ kiểm thử. ViewModel đóng vai trò trung gian giữa View và Model xử lý tương tác của người dùng, ràng buộc dữ liệu và các tác vụ liên quan đến UI (Anderson 2012). Sự tách biệt này thúc đẩy khả năng tái sử dụng của mã nguồn, cho phép thực hiện kiểm thử dễ dàng hơn và nâng cao khả năng làm việc chung giữa các nhà phát triển bằng cách xác định ranh giới rõ rang giữa các thành phần khác nhau của ứng dụng.

Tổng hợp lại, sau khi áp dụng MVVM, dự án đạt được những lợi ích sau:

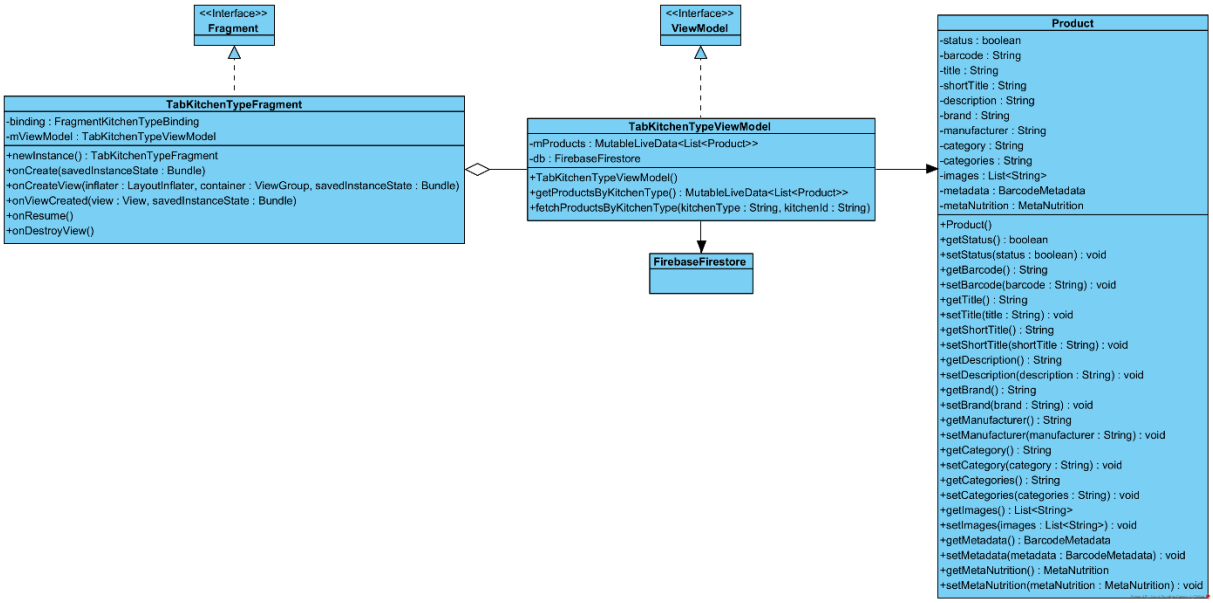
- Phân tách mối quan tâm: MVVM chia dự án thành ba lớp riêng biệt bao gồm giao diện người dùng (View), các đối tượng (Model) và các logic hiện thị (ViewModel). Sự phân tách này cải thiện cách tổ chức code và khả năng bảo trì.
- Kiểm thử: MVVM thúc đẩy khả năng kiểm thử bằng cách cô lập logic hiển thị trong ViewModel, có thể được kiểm thử đơn vị (unit-testing) độc lập với giao diện người dùng. Điều này cho phép cho phép thực hiện kiểm thử để xác minh hành vi của logic nghiệp vụ của ứng dụng mà không cần sử dụng các thư viện kiểm thử UI.
- Liên kết dữ liệu: MVVM tận dụng liên kết dữ liệu để thiết lập giao tiếp hai chiều giữa View và ViewModel, đảm bảo rằng các thay đổi trong dữ liệu được tự động cập nhật trên giao diện người dùng và ngược lại. Điều này đơn giản hóa việc

phát triển UI và giảm lượng mã nguồn cần thiết để đồng bộ dữ liệu giữa các lớp khác nhau của ứng dụng.

2.9.2 Sơ đồ lớp



Hình 2.51 Sơ đồ kiến trúc áp dụng mô hình MVVM



Hình 2.52 Sơ đồ lớp áp dụng mô hình MVVM với chức năng liên quan đến màn hình bếp

2.9.3 Mã nguồn áp dụng mẫu thiết kế

```
public class Product {
    2 usages
    @SerializedName("success")
    private boolean status;
    2 usages
    @SerializedName("barcode")
    private String barcode;
    2 usages
    @SerializedName("title")
    private String title;
    2 usages
    @SerializedName("alias")
    private String shortTitle;
    2 usages
    @SerializedName("description")
    private String description;
    2 usages
```

Hình 2.53 Ví dụ về Model (lớp Product)

```
public class TabKitchenTypeFragment extends Fragment {

    2 usages
    public static final String ARG_OBJECT = "object";
    5 usages
    private FragmentKitchenTypeBinding binding;
    4 usages
    private TabKitchenTypeViewModel mViewModel;
    5 usages
    private ProductSorter productSorter;

    2 usages
    Button sortButton;
    4 usages
    KitchenTypeAdapter adapter;
    no usages
    TabLayout tabLayout;
    2 usages
    Bundle args;

    Tien Le
    public static TabKitchenTypeFragment newInstance() { return new TabKitchenTypeFragment(); }

    Tien Le
    @Override
    public void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
    }
```

Hình 2.54 Ví dụ về View (lớp TabKitchenTypeFragment)

```
public class TabKitchenTypeViewModel extends ViewModel {
    3 usages
    private MutableLiveData<List<Product>> mProducts;
    2 usages
    FirebaseFirestore db;

    no usages  Tien Le
    public TabKitchenTypeViewModel() { this.mProducts = new MutableLiveData<>(); }

    1 usage  Tien Le
    public void fetchProductsByKitchenType(String kitchenType, String kitchenId) {
        db = FirebaseFirestore.getInstance();
        List<Product> products = new ArrayList<>();

        db.collection( collectionPath: "products") CollectionReference
                .whereEqualTo( field: "kitchenId", kitchenId) Query
                .whereEqualTo( field: "pantry", kitchenType)
                .get() Task<QuerySnapshot>
                .addOnCompleteListener(task -> {...});
    }

    2 usages  Tien Le
    public MutableLiveData<List<Product>> getProductsByKitchenType() { return mProducts; }
}
```

Hình 2.55 Ví dụ về ViewModel (lớp TabKitchenTypeViewModel)

CHƯƠNG 3 – TỔNG KẾT

Như vậy, việc áp dụng các mẫu thiết kế vào ứng dụng di động hiện tại đã cải thiện về cách tổ chức mã nguồn, tăng khả năng thích ứng với những yêu cầu mới và dễ dàng mở rộng trong tương lai. Nhìn chung dự án này không chỉ minh họa cho việc ứng dụng thực tiễn các mẫu thiết kế mà còn nhấn mạnh tầm quan trọng của chúng trong việc phát triển các hệ thống phần mềm phức tạp như ứng dụng di động của chúng tôi.

TÀI LIỆU THAM KHẢO

English

1. Anderson, Chris. 2012. “The Model-View-ViewModel (MVVM) Design Pattern.” Pp. 461–99 in *Pro Business Applications with Silverlight 5*, edited by C. Anderson. Berkeley, CA: Apress.

2. Anon. n.d.-a. “Adapter.” Retrieved April 30, 2024 (https://refactoring.guru/design-patterns/adapter).

3. Anon. n.d.-b. “Proxy.” Retrieved April 30, 2024 (https://refactoring.guru/design-patterns/proxy).

4. Anon. n.d.-c. “Strategy.” Retrieved April 30, 2024 (https://refactoring.guru/design-patterns/strategy).

5. Hu, Chenglie. 2023. “Software Design Patterns.” Pp. 231–75 in *An Introduction to Software Design: Concepts, Principles, Methodologies, and Techniques*, edited by C. Hu. Cham: Springer International Publishing.

6. Joshi, Bipin. 2016. “Creational Patterns: Singleton, Factory Method, and Prototype.” Pp. 87–109 in *Beginning SOLID Principles and Design Patterns for ASP.NET Developers*, edited by B. Joshi. Berkeley, CA: Apress.

7. Lou, Tian. n.d. “A Comparison of Android Native App Architecture – MVC, MVP and MVVM.”