UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

GIANEI LEANDRO SEBASTIANY

Smart Catalog: an Experience Report on the Development of a Software Product Line

Work presented in partial fulfillment
of the requirements for the degree of
Bachelor in Computer Science

Advisor: Prof. Dr. Ingrid Oliveira de Nunes

Porto Alegre
December 2016

ABSTRACT

Software Product Lines (SPLs) consist of a large-scale form of software reuse. Instead of developing single software products reusing components in an ad hoc way, in an SPL approach, a family of products is addressed. A common set of software assets is built, which allows derivation of different products in a prescribed way. In this work, this promising approach is exploited to develop an SPL of product catalogs, namely Smart Catalog, targeting mobile devices using the Android platform. An extractive and reactive approach is adopted, that is, previously developed applications are considered to extract a new SPL. Then, new features are incorporated based on new products that the SPL aims to derive. As result, the challenges and benefits identified in the work are reported, as well as lessons learned.

Keywords: SPL. Software Product Line. Software Engineering. Android. Clean Architecture. Catalog. App.

Smart Catalog: um Relatório de Experiencia no Desenvolvimento de uma Linha de Produto de Software

RESUMO

Linhas de Produto de Software (SPLs) são definidas como uma forma de reuso de software em larga escala. Ao invés de desenvolver produtos de software únicos reusando componentes de forma desorganizada, em uma abordagem SPL, uma família de produtos é definida. Um conjunto comum the artefatos de software é criado, o que permite a derivação em uma forma pré definida de diferentes produtos. Neste trabalho, esta abordagem promissora será explorada para desenvolver uma SPL de catálogo de produtos, nomeada Smart Catalog, que tem como alvo dispositivos móveis da plataforma Android. Uma abordagem reativa será adotada, o que significa que uma SPL será criada a partir de considerações de aplicações previamente desenvolvidas. Como resultado, os desafios e benefícios identificados neste trabalho serão reportados, bem como lições aprendidas.

Palavras-chave: Linha de Produto de Software, Enhenharia de Software, Android, Clean Architecture, Catálogo, App.

LIST OF FIGURES

LISTINGS

LIST OF ABBREVIATIONS AND ACRONYMS

SPL	Software Product Line
SPLE	Software Product Line Engineering
OS	Operational System
OOP	Object Oriented Programing
DCI	Data, Context, Interaction
BCE	Boundary, Control, Entity
MVP	Model, View, Presenter
IDE	Integrated development environment
DI	Dependency Injection
UI	User Interface
IO	Input Output
ERP	Enterprise Resource Planning

CONTENTS

1 INTRODUCTION

In the software industry, a promising development paradigm called *software product line engineering* (SPLE) is available. In most cases of software applications, it is possible to identify solutions that are crafted for a specific client and software solutions that are produced for a broad range of customers. Generally, specific solutions are expensive, while at the same time broad solutions do not have enough diversification. This situation created the need for mass customization, which is the production in large-scale of personalized goods. In order to produce such goods, it is necessary to have platforms that contain the main aspects of the product. The combination of platforms and mass customization resulted in *software product line engineering* (POHL; BÖCKLE; LINDEN, 2005).

A group of software applications which are very similar to each other and are thoroughly used by the industry is *products catalog applications*. E-commerce applications are examples of product catalog applications that show a large variety of products in order to make sales. Many product catalog solutions are already available and currently in use by various business, however, when a new solution is required for a slightly different business, it is generally created in an ad hoc way, that is, in an improvised way. This causes waste of development resources due to the inability to reuse code efficiently and increases the time-to-market.

Furthermore, creating and maintaining a family of similar software solutions that improves code reuse and time-to-market such as a software product line is complex. It is fundamental to determine formally what will the software product line (SPL) be capable of producing. It is also required to create a software architecture that is capable of taking advantage of the similarities by providing reusability. Furthermore, it is necessary to understand what is required for a specific client and instantiate the specific customization from the developed architecture. During the SPL creation, many concerns need to be accounted for, such as development costs, time of production, quality, maintenance and evolution.

Many software product lines were already created and analyzed, as discussed by Linden, Schmid and Rommes (2007), but few are aimed at the mobile environment. A mobile application consists of a software that can be executed on a mobile device, such as a smartphone. Business companies are benefiting from the usage of mobile applications (ISLAM; ISLAM; MAZUMDER, 2010). According to research about operational system market share in 2015 made by IDC (2015), the Android SO has a market share of 82.8 %, making it a desirable product target.

This work thus consists of the development of a software product line that provides the infrastructure needed to derive Android apps of product catalogs that show products efficiently among different business requirements. Due to existing solutions and a previously similar solution developed by the same author of this work, it is possible to approach the SPL creation in an extractive form, which is leveraging features of existing applications to create a new SPL in this area (KRUEGER, 2001). The existing solutions are used among a large variety of business and each of them has similarities and specific features, therefore the engineering of a SPL can leverage the commonalities existing in current solutions to create a core set of functionalities. The engineering process is capable of creating core parts of the software that are reusable and applicable to different client needs. With a platform and mass customization, it is possible to deliver specific needs for specific customers. All this is done while reducing development costs, time-to-market and maintaining a high quality of products.

This work provides a better understanding of the development of a SPL by creating a practical example and reporting insights of the process. The implementation is focused on the creation of the core assets with some variable features intended to exemplify the variability implementation in a SPL. It is organized this way: Chapter 2 defines the concepts of Software Product Lines. Chapter 3 analyzes existing software. Chapter 4 identifies the features of the SPL. Chapter 5 presents the design of the SPL architecture and describes its implementation. Finally, chapter 6 presents the instantiation of a final product from the SPL and chapter 7 explains the overall experience in developing this work.

2 BACKGROUND ON SOFTWARE PRODUCT LINES

"Software product line engineering is a paradigm to develop software applications (software-intensive systems and software products) using platforms and mass customization" (POHL; BÖCKLE; LINDEN, 2005). This chapter provides the concepts, requirements, limitations and benefits of SPLE. Additionally, an overview of the SPLE frameworks and lifecycle is provided.

2.1 Key Concepts

Software product lines came as a new architecture paradigm that enables the mass production of software applications while delivering customization in each application. This subject emerged as a conjunction of already known concepts: *mass customization*, *common platforms* and *program families*.

Mass production is the conception of goods in large quantities and in low cost. It allows lower production costs but not necessarily lower product quality. This goal is achieved by standardizing and automation of the production process. This concept had remarkable outcomes when Henry Ford designed assembly lines that drastically reduced time required to build automobiles.

A common platform is any base of technologies on which other technologies processes are built (POHL; BÖCKLE; LINDEN, 2005). Mass production, as created with Ford, reduced the possibility of customization, hence the need for a common platform. For example, in Ford production, all cars need a floor panel, a suspension system, and rocker panels. A platform containing these pieces can be created and reused by different cars. This approach enabled car manufacturers to increase their variety while reducing costs.

Program families is a concept related to SPL either. A program family is a set of software systems that has so many common properties that it is advantageous to study the common properties of the programs before analyzing individual members (PARNAS, 1976). Parnas proposed a new way of creating programs that is by taking an existing program at an intermediate point and making adaptations from that point. According to this practice, it is possible to reuse existing solutions and have software programs that share the same ancestor.

The conjunction of mass production, platforms and program families introduces the base concepts of a software product line. The SPL plan proactively for a systematic reuse, by creating reusable parts and managing the variability of the products, in order to deliver mass customization. The software product line requires the systematic construction of a consistent software platform to enable reuse, which is the core. It is necessary to make upfront planning to create correct platform artifacts, which are parts of the platform, such as test plans, test designs, requirements models, architectural models and software components.

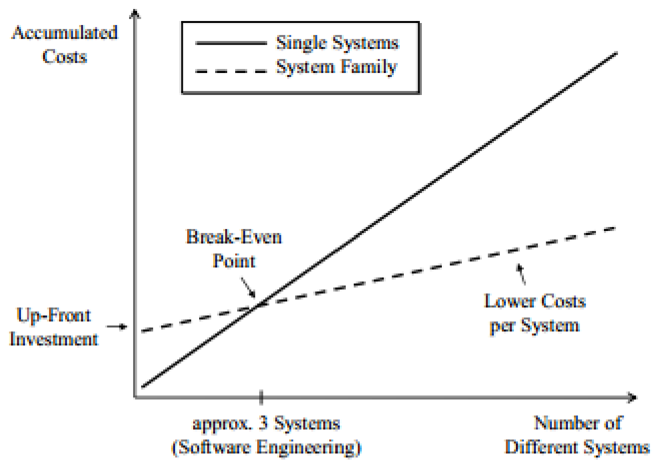
Generally, software is delivered in two forms: standard software such as commercial off the shelf (COTS) which aims to deliver a solution for as many customers as possible, allowing mass production but not being able to deliver specific customer needs. The other form is custom software, which aims to fulfill all needs of a single customer, implying elevated production costs. Software Product Lines (SPL) is a recent and promising effort to deliver mass customized software, with improved customer satisfaction and reduced time-to-market. Software product lines enable software companies to deliver a family of similar products efficiently. Just like a production line of cars, a software product line can deliver different applications that meet individual requirements, produced in large-scale within a certain variability. The Software Product Line paradigm can provide tools, techniques, and process guidelines to create, maintain and evolve a family of software. The SPL paradigm can help the creation of recent forms of software delivery such as Software as a Service and other cloud-based solutions but requires further investigation on the subject (SCHMID; RUMMLER, 2012).

2.2 Basic Requirements

The combination of common platforms and mass customization allows the reuse of existing technology and at the same time creates products that are closer to the customer expectation. The combination of these ideas changes the developing process as well as the organization of the company developing the SPL.

The first step is to build the platform where analysis of existing software can be used to generate a platform. Throughout the development process, it is necessary to identify and describe where and how products emerging from the same product line differ. This step is the creation of flexibility which is a precondition for mass customization. The flexibility defines which are the possible options of the SPL and where exactly they can vary.

Figure 2.1: Development cost of single system and SPLE.



Source: Pohl, Böckle and Linden (2005)

Reorganizing the company is necessary in order to accommodate these steps of development. It may be necessary to establish new organization units, for example, one that is responsible for the platform and another that is responsible for creating the variability.

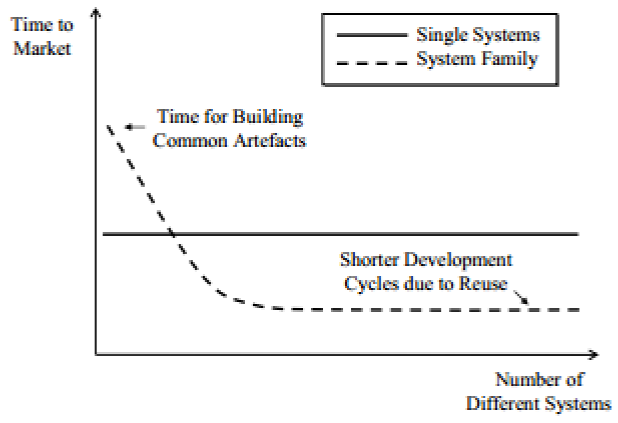
2.3 Outcomes

Many outcomes emerge from the use of product line engineering. One of the most prominent benefits is the reduction in development costs. When the software platform is created, it will reduce development costs of new applications due to the reuse of the platform. Although creating a platform that is ready to deliver the required variability has increased upfront costs, the accumulated costs when developing an increased amount of solutions is reduced. Figure 2.1 shows that there is a break even point where costs of creating new solutions outcome the upfront cost of creating the platform.

Similarly to production costs, the time-to-market of the solution in a software product line is longer in the initial phases but generates less time-to-market when more solutions are developed. There is an initial burden analyzing the variability and creating the platform of the production line, nevertheless, when the initial work is done, products can be created by reusing existing code. This drastically reduces the time that new solutions need to go to market. Figure 2.2 shows the initial burden of production compared to single family systems.

SPL improves the quality of the products. During the development process, the

Figure 2.2: Time-to-market with and without SPLE.



Source: Pohl, Böckle and Linden (2005)

platform is tested and reviewed throughout all emerging applications that use the platform. When an improvement is made to the platform due to a necessity discovered in one application, the improvement is spread across all products of the family, since all of them are generated from the same core. Therefore it is expected that all products of the production line have enhanced quality. Maintenance effort and evolution do also benefit by the same means: when an error correction or evolution is made on the platform, it is spread throughout the software product line.

Due to an increased amounts of functionalities emerging from different customers, the complexity to manage the SPL is greater. The code base is more extensive due to plenty of needed functionalities. SPLE offers adequate measures to cope with the complexity of all the features interactions and management of different but similar products. By creating a common platform, developing resources can be moved out from separate places into a concentrated artifact of the platform.

Customers also benefit from software product lines. It generates products that are more adapted to their needs and wishes in shorter time. They can also take advantage of improvements that were made to other customers, inasmuch as some of these improvements can happen on the platform of the product line. The customers also have a better cost estimation, since the development of the platform has already been made, it is known that a big part of the application was already produced, thus reducing the amount of code to be estimated. The experience generated during the development of previous applications of the product line, help to pinpoint the cost estimation of new applications.

2.4 Challenges and Limitations

Software product line is a recent paradigm, many barriers existed to the adoption of SPL. The lack of technology was a strong barrier to software product line for a long time, programming languages that could enable the needs of product line did not exist. Object-oriented programming (OOP) is one of the most important technologies that enable software product line and only modern languages provide it. Encapsulation is a prerequisite to variability management and OOP help programmers to achieve it. Another enabling technology is the component technology which enables developers to create parts of software in a way that it is not deeply attached to other parts of the software. Conditional compilation (COUTO; VALENTE; FIGUEIREDO, 2011), model-driven and aspect-oriented programming (VOELTER; GROHER, 2007), and feature-oriented SPL (APEL et al., 2013) are techniques used to create SPL but are studied or provided with tools only recently. Another technological issue was the difficulty of managing the configuration of the software product line, it requires sophisticated tools to deal with the complexity.

Besides of the technology, another major prerequisite for SPL is the need for deep domain expertise. Identification of commonalities and variability among a specific market involves the knowledge of process, abstractions, and singularities of the market. A feature that is inserted in the product line which is not used by final products causes development waste. Similarly missing a needed feature causes additional costs to adapt the SPL in the future, besides of reducing the effectiveness of the SPL. The domain experts must have deep knowledge about the subject of the SPL, moreover, the abstractions used in the engineering process must be in an appropriate level according to the domain. Not knowing the abstractions and the domain correctly may lead to misunderstandings causing wrong choices during the development process. Regarding the market domain, it is also important that it is stable. Product lines require considerable up-front investment if the domain is volatile as much as to change all requirements, the work already done is wasted.

2.5 Process

There are two main development processes in the software product line engineering paradigm: domain engineering and application engineering.

- *Domain Engineering* is responsible for creating the platform of the SPL, it defines the variability and commonality of the product line. The activities involve analysis of the domain to identify required features, the realization of the platform as well as maintenance and evolution. The domain engineering also defines the scope of the SPL, that is, what types of software it will be capable of producing and what features it will cover. The sub-activities of domain engineering have the purpose of refining and detailing the variability defined in previous steps and provide feedback about the feasibility of the required variability.
- *Application Engineering* is responsible for creating products using the previously created platform, the variability is instantiated according to the platform possibilities, delivering a custom solution. The goal of application engineering is to achieve an as high as possible reuse of the platform artifacts when developing new products, by exploiting the commonality and variability of the SPL. The process is responsible for designing and realizing the application while estimating the impacts of differences existing on the application and the platform.

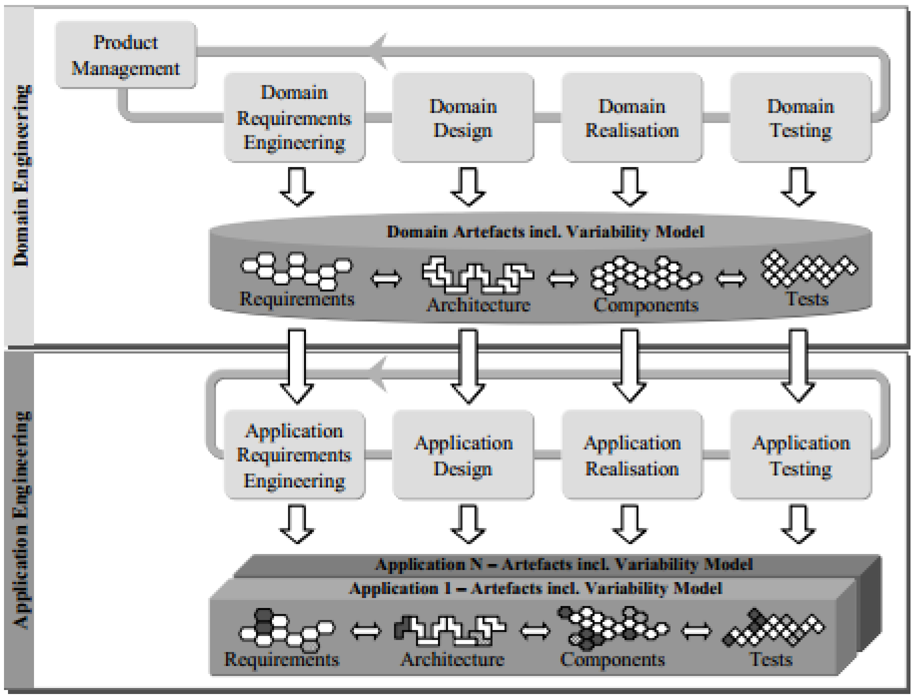
By splitting these processes, separation of concerns is obtained. The first process is concerned in building a robust platform while the latter is concerned with building a solution that is customer specific in short time. The concerns do also have relation to variability: domain engineering need to ensure that the proper variability is available whilst the application engineering binds the variability. Note that neither of the processes have to be performed in a sequential order.

There are available in literature many different methods to approach the SPLE, such as COPA, FAST, FORM, KobrA and QADA which are compared by Matinlassi (2004). The framework proposed by Pohl, Böckle and Linden (2005) defines the general processes of a software production line, making it usable in any context with due adaptations. Pohl et al.'s framework describe subprocess of the domain engineering and application engineering as described in figure 2.3.

The next five subprocesses define the domain engineering aspect of the framework.

- *Product Management* is responsible for enforcing the marketing goals throughout the development of the SPL. The product management accesses what the SPL is intended to create, defining the targets and stakeholders of the SPL. The major result of the product management subprocess is to create the product roadmap, that will guide the overall development of the product line and will be revisited in cycles.

Figure 2.3: Overview of the software product line engineering framework.



Source: Pohl, Böckle and Linden (2005)

The product roadmap defines the major features that the product line desires to create in an approximate time line.

- *Domain Requirements Engineering* is a continuous process of developing the common and variable requirements to the domain. This process will take as input the product roadmap to elucidate and document the commonality and variability of the SPL core. The output of this step is a domain variability model.
- *Domain Design* defines the reference architecture of the SPL from the requirements defined in the previous step. It is a high-level view of the architecture of the platform.
- *Domain Realization* is the detailed design and implementation of the SPL platform described in previous steps. The variability implementation of the SPL is possible by different means or with a combination of different solutions. The result of this process is loosely coupled configurable components.
- *Domain Testing* is responsible for validating the developed solutions against the definitions created in the previous steps. It produces tests that stress the configuration capability of the developed components.

The following four subprocesses define the application engineering aspect of the framework and are closely related to its counterparts on the domain engineering.

- *Application Requirements Engineering* is responsible for identifying features available from the SPL that are needed to the custom solution and reporting differences between application requirements and what is offered from the platform.
- *Application Design* sub-process produces the architecture specifications of the desired application. It uses the platform architecture definitions and incorporates definitions needed for the specific solution.
- *Application Realization* is the sub-process that develops what was previously designed. It reuses artifacts already produced from the domain realization and adds code that is specific to the application. The result of this sub-process is a running application that can be delivered to customers.
- *Application Testing* is responsible for validating the constructed application according to definitions previously created. It is similar to domain testing but focuses on features that are specific to the application.

However, Pohl, Böckle and Linden (2005) framework is intended to be used in large environments where a large quantity of personnel is involved, therefore due to project size restrictions, this work will use a subset of the process described by Pohl, Böckle and Linden (2005). The domain engineering consists of product management, domain requirements engineering and a union of domain design and realization. The application engineering is described as a single process that encompasses all sub-process.

3 PRODUCT MANAGEMENT

This chapter is the experience report of the product management subprocess as defined by Pohl et al. The scope of the SPL is specified by analyzing the market of catalog applications. An analysis of similarities among available solutions is performed in order to obtain knowledge about the domain. This software product line is named Smart Catalog.

3.1 Scoping

Scoping is a sub-activity of Product Management which fundamentally analyzes the commonality and variability to specify the key features of the software product line (POHL; BÖCKLE; LINDEN, 2005; CLEMENTS; NORTHROP, 2001).

The aim of this SPL is to create a family of software products that are digital versions of products portfolios. By creating a virtual form of product catalog, computational resources can be used in order to give dynamicity to the content, together with common resources such as real-time product availability synchronization, products recommendation, personalization, search, and filtering.

Product portfolios are used by a large variety of business, for example, the menu of restaurants; the product portfolio of a sales representatives; the products showcase of a hardware company. Each company has its own custom functional process, so the solution presented must be shaped for each business. Besides of having variability among different areas of business, there is variability among business of the same area. With that in mind, SPL is the approach that can deliver products for each specific case whilst reducing production and maintenance costs by taking advantage of the overall similarity of the required software.

Different types of markets are the target of this SPL. Wholesalers companies require catalogs to show their products to consumers. Independent salesman which represent other companies can also use catalogs to display their products. Stores in general also use catalogs as a helper to display their products, it is common to find flyers of products inside clothing stores. Restaurants and bars in general also have a form of showing their products, which is usually their menu which is a form of a catalog. Currently, the market is full of solutions, but most of them are specific to a customer and do not make use of improvements of other solutions.

A customer is a person that will acquire an instantiation of the SPL, with the desired customization according to its market area. The customers of this SPL are companies that work selling items in a specific market. There is also an intended customer that is the independent salesman, which would acquire the SPL software via an already available instantiation of the SPL in a common application store.

There is a growing use of mobile applications in an enterprise environment. The advent of mobile computing and its usage in the business environment makes it an area in need for new solutions. This SPL takes benefit of this trend by developing the software for mobile platforms.

For a complete understanding of the abstractions of this work scope, definitions about product catalogs are necessary. The following is a list of all definitions used throughout the remainder of this work:

- *Item* is the item being displayed. The catalog intends to display many items to consumers. For example TV; radio; Bluetooth player.
- *Category*, also referred to as *departments*, it is a set of items that share similar purposes and or attributes. For example, all items from the previous example are in the electronics category.
- *Item Set* is a group of items that are grouped by its category or factors that are not related to its category. For example, advertised items and recommended items are item sets.
- *Sub Categories* are departments defined inside a broader category in order to better distinguish peculiarities of a set of items
- *Category Tree* is the organization of the categories since a group of categories can belong to a broader category.
- *E-commerce* is a software solution that enables the purchasing and selling of any goods or services over the internet.
- *Drawer Menu* is a common user interface artifact in Android apps. It is the menu of the application, which can be accessed swiping from the left border of the app or via the menu button.
- *Tab layout* is another common user interface artifact in Android apps that consists of different views to be organized in a tab.

3.2 Analysis of Existent Systems

During the scoping process, according to the desired market and clients, some concerns about the main functionalities were found. These concerns are used to analyze each of the chosen systems:

- *Menus and Navigation* analyzes which options are available through the app menus, such as searches, filters, and sorting options. This dictates the way the user interacts with the app and in which ways the user can reach a specific functionality of the app.
- *Special set of items* verifies which set of items the app uses to display to the user, such as recommendations and advertised items.
- *Set of items UI* analyzes the layout on which the app present a set of items
- *Item detail UI* verifies how the app present information regarding one item to the user.
- *Home screen UI* verifies the elements shown to the user on the first and main view of the application.
- *Product Management* identifies if the app allows the user to register new items and if the app data is retrieved from external systems.
- *Connectivity* analyzes if the app can be used offline or if it requires a working connection to the internet.

Some aspects are not analyzed in this work. Features such as account management, purchasing, and shopping charts are not analyzed because they are out of the scope of this work.

As this SPL line is intended to be built for Android, all chosen applications are built for the Android OS. Apps were selected due to their similarity to the scope of this work. Nevertheless, there are plenty of apps available in the defined scope, but due to constraints, only a few solutions were selected. The chosen apps for analysis are Amazon, Ponto Frio, Mercado Livre, Enjoei and Ditlanta Catalog. The latter one is the solution that was developed previously by the same author of this work.

The analysis is split into three groups. The first group is e-commerce apps, the second group is community commerce apps and third is wholesaler app.

3.2.1 E-Commerce Apps

This group generally refers to large department stores. Usually, it is a mobile version of their e-commerce website. A user of this software can add items to a chart and make purchases. The products are registered automatically by integration with existing ERP (Enterprise Resource Planning) that controls items stock, prices, and sales.

Amazon

Amazon App is a top quality app and has the highest number of downloads among the apps analyzed. This app only works online, where data is fetched from online systems on each view of the app. The home screen of the app has many views for recommendations, advertised items and items recommended to the user. The top view is a carousel (an automatic sliding view of images) that displays ads. The user can make a string search of items from many points of the app.

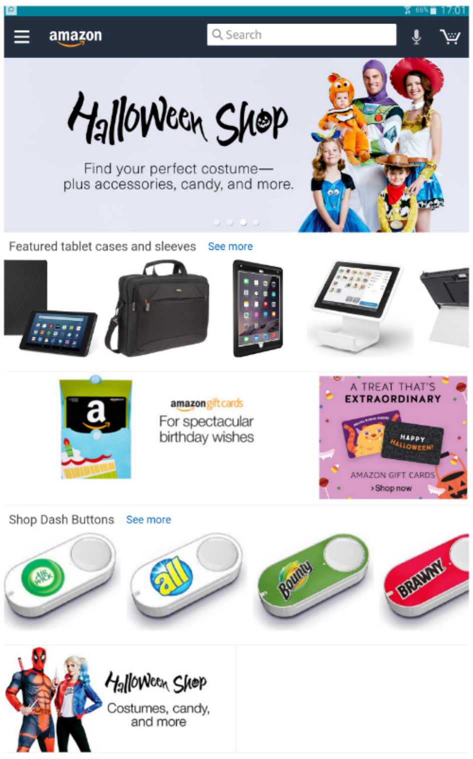
The drawer menu displays all available categories, grouped in category groups. The user can select to view all items from a top category or navigate deep in the hierarchy to visualize items of a subset of categories. The drawer also has options to visualize other information, such as the user account.

The Amazon App does a great work showing on many views a group of items that the system recommends to the user, as well as inserting promoted items to the user. Each department view has special content, such as ads, best sellers, new releases, top rated. Below this special content, there is a grid of the first items within that department. There is also a link to the full list of items.

When showing a group of items, the app can display it in lists or grids, where the user can select the type and size. The user can filter items with plenty of options that are relevant to the items being displayed as well as sort items in many aspects.

When displaying a specific item, the app shows all information in sections organized in a list. Bellow item description, the app display reviews, ratings and a selection of items that can be the previously visited and or recommendations. The user can share the item to other apps or have a link to the external website.

Figure 3.1: Amazon app screenshot.


amazon
Search
Halloween Shop
Find your perfect costume—
plus accessories, candy, and more.
Featured tablet cases and sleeves
See more
amazon gift cards
For spectacular birthday wishes
A TREAT THAT'S EXTRAORDINARY
AMAZON GIFT CARDS
› Shop now
Shop Dash Buttons
See more
Halloween Shop
Costumes, candy, and more

Ponto Frio

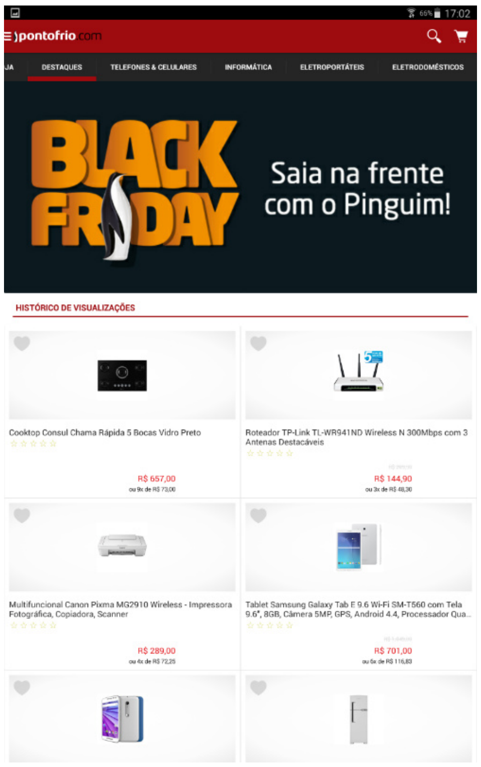
This app works only when online. There is no way of using the app in the offline mode. The home screen is a view pager where the starting selected page has a carousel of adds followed by a selection of previously visualized and advertised items. The leftmost tab is a category tree view. The other tabs display a small list of items from featured categories.

The app menu drawer contains only options related to the user account. Categories navigation is made via the special categories tab in the main screen. The user can make string search to find specific items.

The app only displays sets of items in large grids of items. The user has sorting functionality and a few filter options to narrow the item set visualization.

When displaying details of an item, the app split information about the product in different views. For example, there is a specific tab for item specification and another specific tab for item reviews.

Figure 3.2: Ponto Frio app screenshot.


17:02
pontofrio.com
JA
DESTAQUES
TELEFONES & CELULARES
INFORMÁTICA
ELETROPORTÁTEIS
ELETRODOMÉSTICOS
BLACK FRIDAY
Saia na frente com o Pinguim!
HISTÓRICO DE VISUALIZAÇÕES
Cooktop Consul Chama Rápida 5 Bocas Vidro Preto
R$ 657,00
ou 9x de R$ 73,00
Roteador TP-Link TL-WR941ND Wireless N 300Mbps com 3 Antenas Destacáveis
R$ 144,90
ou 3x de R$ 48,30
Multifuncional Canon Pixma MG2910 Wireless - Impressora Fotográfica, Copiadora, Scanner
R$ 289,00
ou 4x de R$ 72,25
Tablet Samsung Galaxy Tab E 9.6 Wi-Fi SM-T560 com Tela 9.6", 8GB, Câmera 5MP, GPS, Android 4.4, Processador Qua...
R$ 701,00
ou 6x de R$ 116,83

3.2.2 Community Commerce Apps

The main aspect of this group is that users of the system can register their own items for sale and system users buy items from other users (and not from the company). The rest of the application is similar to the previous group.

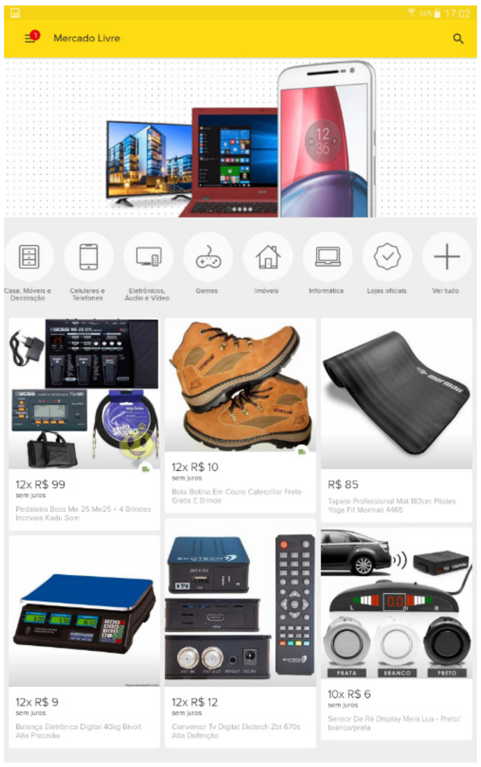
Mercado Livre

This app, similar to the previous app, can only work in online mode, the app asks for connection if it is being used without connection to the ethernet. The main view of the app is presented with a carousel of itemized categories followed by a staggered grid of items. The items displayed on the main screen are a selection of advertised items.

The drawer menu only has options related to the user account and specific platform options such as the item registration button. The categories can be accessed from the categories items on the main view or trough a special button on the main screen.

When displaying a set of items, the user can choose among the following display options: small staggered grid, large staggered grid or list view. The user can make

Figure 3.3: Mercado Livre app screenshot.


Mercado Livre
17:02
Casa, Móveis e Decoração
Celulares e Telefones
Eletrônicos, Áudio e Vídeo
Games
Imóveis
Informática
Lojas oficiais
Ver tudo
12x R$ 99
sem juros
Pedaleira Boss Me 25 Me25 + 4 Brindes Incriveis Kadu Som
12x R$ 10
sem juros
Bota Botina Em Couro Caterpillar Frete Gratis E Brinde
R$ 85
Tapete Professional Mat 183cm Pilates Yoga Fit Mormaii 4465
12x R$ 9
sem juros
Balança Eletrônica Digital 40kg Bivolt Alta Precisão
12x R$ 12
sem juros
Conversor Tv Digital Ekotech Zbt 670s Alta Definição
10x R$ 6
sem juros
Sensor De Ré Display Meia Lua - Preto/branco/prata
PRATA
BRANCO
PRETO

searches as well as select, order, and filter group of items.

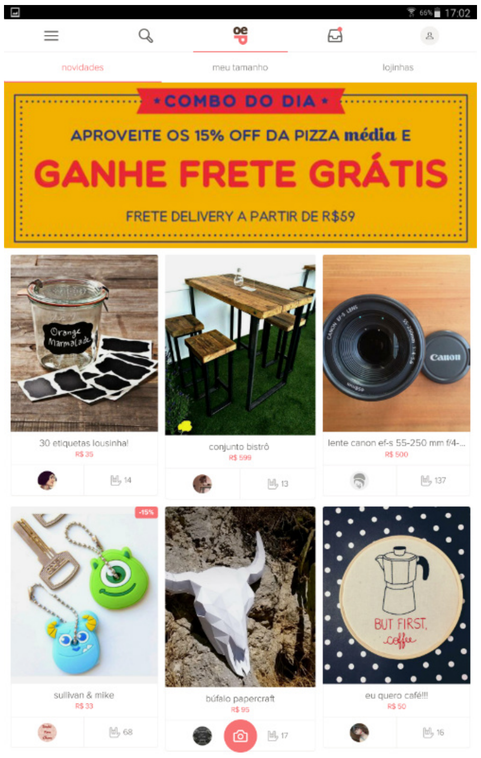
The product view is concentrated on a single view, where all info is laid in a list fashion followed by reviews of the vendor.

Enjoei

This app can only function with a working internet connection. The first view shows a grid of featured items where there are adds in between items. On the main view, there are three pages where the user can find items according to its preferences. The categories are accessed from a menu drawer where there is also available options related to the user account and the option to register a new item.

The items group layouts available to the user are a vertical list, grid, and staggered grid. When displaying details of an item, the app shows a large picture of the item, followed by the descriptions organized in a list. After the description, the app shows a set of recommended items.

Figure 3.4: Enjoei app screenshot.



3.2.3 Wholesaler apps

The app in this category is intended for usage by wholesale companies where sales representatives use the application to help in the company sales process. This type of app, as opposed to e-commerce, is not intended for final customers.

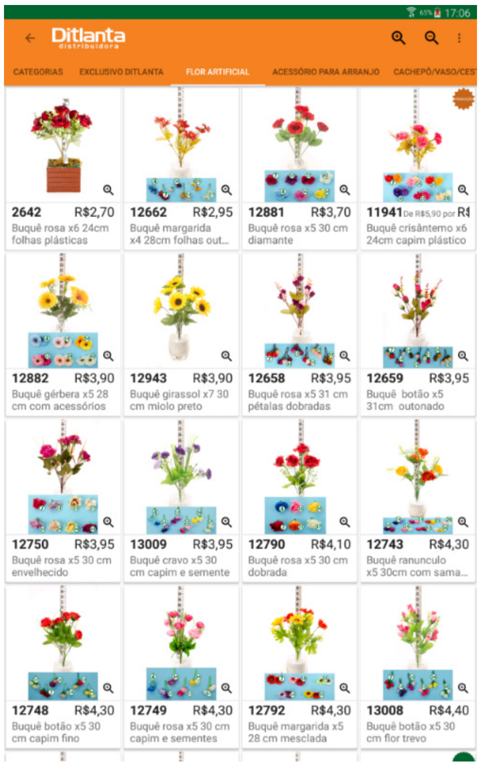
Ditlanta Catalog

This is the application previously developed by the same author of this work. It is the only application that works offline. When the app is online, it fetches latest data and updates its local database, making the app available in situations where there is no internet connection.

The main screen of the app consists of a grid of shortcuts to category groups. A category group leads to a tab pager view where each page contains a category from the shortcuts. The main view has a search option that allows the user to make string searches.

The drawer menu is available when displaying groups of items and it displays all available categories among the currently displayed items. There is no filter neither

Figure 3.5: Ditlanta Catalog app screenshot.


Ditlanta
distribuidora
CATEGORIAS
EXCLUSIVO DITLANTA
FLOR ARTIFICIAL
ACESSÓRIO PARA ARRANJO
CACHEPÔ/VASO/CES
2642 R$2,70 Buquê rosa x6 24cm folhas plásticas
12662 R$2,95 Buquê margarida x4 28cm folhas out...
12881 R$3,70 Buquê rosa x5 30 cm diamante
11941 De R$5,90 por R$ Buquê crisântemo x6 24cm capim plástico
12882 R$3,90 Buquê gérbera x5 28 cm com acessórios
12943 R$3,90 Buquê girassol x7 30 cm miolo preto
12658 R$3,95 Buquê rosa x5 31 cm pétalas dobradas
12659 R$3,95 Buquê botão x5 31cm outonado
12750 R$3,95 Buquê rosa x5 30 cm envelhecido
13009 R$3,95 Buquê cravo x5 30 cm capim e semente
12790 R$4,10 Buquê rosa x5 30 cm dobrada
12743 R$4,30 Buquê ranunculo x5 30cm com sama...
12748 R$4,30 Buquê botão x5 30 cm capim fino
12749 R$4,30 Buquê rosa x5 30 cm capim e sementes
12792 R$4,30 Buquê margarida x5 28 cm mesclada
13008 R$4,40 Buquê botão x5 30 cm flor trevo

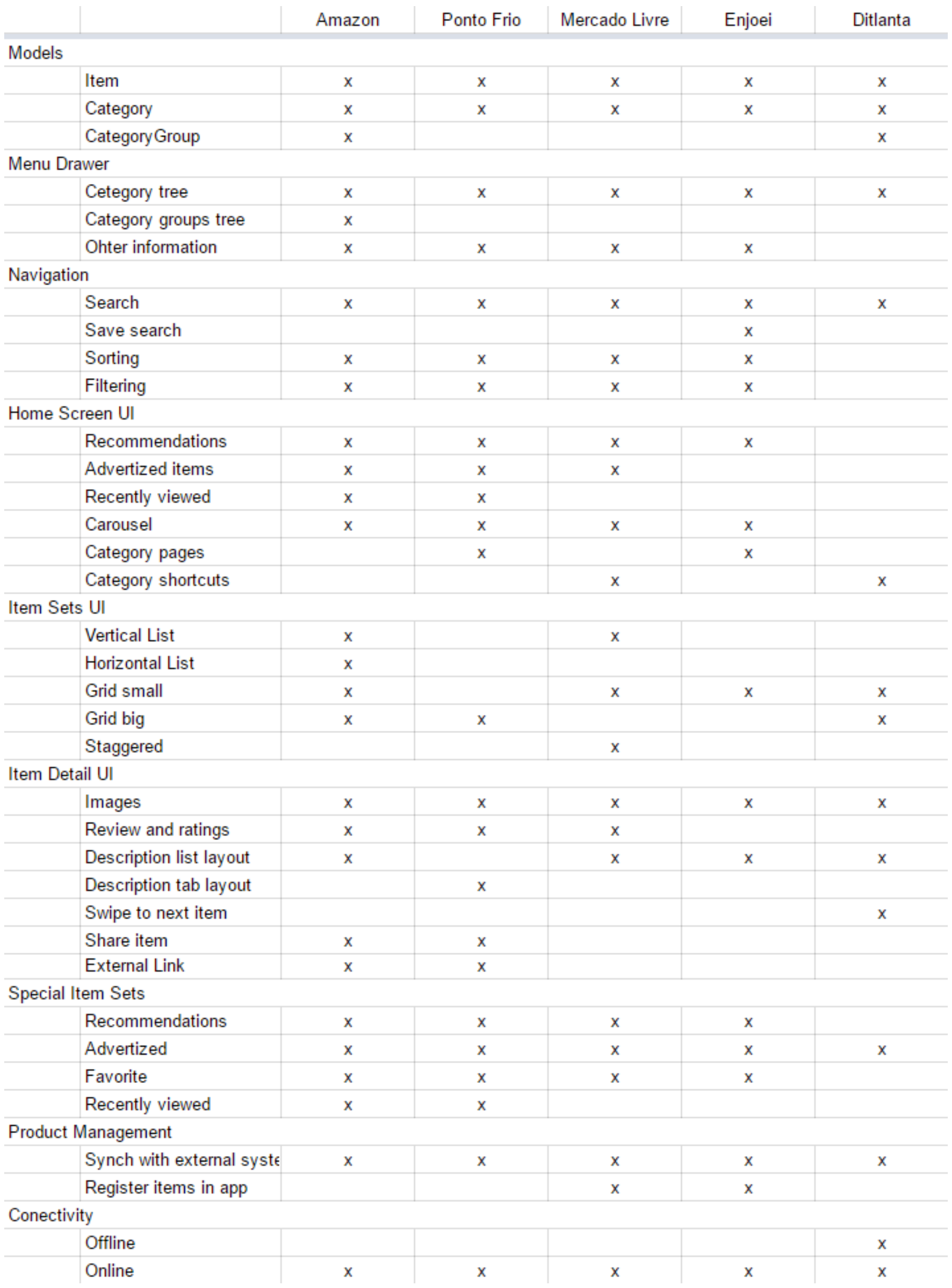
ordering options.

When displaying the details of items, the app only shows a large image of the item and very limited information. However, this is the only app that has swipe gestures that can move to the next or previous item without leaving the item detail view.

Features Summary

In order to better understand which features are available in the analyzed applications, Figure 3.6 shows a table of features present in each of them.

Figure 3.6: Features vs. analyzed apps.



		Amazon	Ponto Frio	Mercado Livre	Enjoei	Ditlanta
Models						
	Item	x	x	x	x	x
	Category	x	x	x	x	x
	CategoryGroup	x				x
Menu Drawer						
	Cetegory tree	x	x	x	x	x
	Category groups tree	x				
	Ohter information	x	x	x	x	
Navigation						
	Search	x	x	x	x	x
	Save search				x	
	Sorting	x	x	x	x	
	Filtering	x	x	x	x	
Home Screen UI						
	Recommendations	x	x	x	x	
	Advertized items	x	x	x		
	Recently viewed	x	x			
	Carousel	x	x	x	x	
	Category pages		x		x	
	Category shortcuts			x		x
Item Sets UI						
	Vertical List	x		x		
	Horizontal List	x				
	Grid small	x		x	x	x
	Grid big	x	x			x
	Staggered			x		
Item Detail UI						
	Images	x	x	x	x	x
	Review and ratings	x	x	x		
	Description list layout	x		x	x	x
	Description tab layout		x			
	Swipe to next item					x
	Share item	x	x			
	External Link	x	x			
Special Item Sets						
	Recommendations	x	x	x	x	
	Advertized	x	x	x	x	x
	Favorite	x	x	x	x	
	Recently viewed	x	x			
Product Management						
	Synch with external syste	x	x	x	x	x
	Register items in app			x	x	
Conectivity						
	Offline					x
	Online	x	x	x	x	x

4 DOMAIN REQUIREMENTS

This chapter organizes the features found in the product management phase. It defines a formal variability that is used in subsequent processes.

4.1 Commonalities and Variability

During the analysis of the applications, as intended, many similarities were found:

- All of the solutions can show a large set of items, having small images to represent each item.
- When the user clicks on an item being displayed on a set of items, a detailed view is opened.
- All apps have a drawer menu.
- The items are organized into categories, and the categories are organized in a tree structure. All apps have the tree hierarchy available to the user at some point.
- All of the solutions are able to perform string searches in the items.
- Images are used when displaying details of an item.

The studied apps have many variability points. Some aspects are different but similar, whereas other aspects are specific to each application. The following is a compilation of the differences found during the study:

- Four apps have specific sets of recommended items, advertised items and recently viewed items, but each of these sets is displayed in a different manner among the apps.
- Four apps use the carousel view in order to display ads.
- Two apps have the home screen split into different pages, and the pages content is different among them.
- The initial page differs at an increased level between the apps, there are some which show different contexts of items, whilst others can only show a single subject. Some apps use the main screen mainly for advertising, while others use it to show only the categories. The main view can contain a different combination of item sets and can have different layouts.
- Each app is unique when displaying a single item. There may be ads and recom-

mendations on the detail view. The apps may display items specifications in a list, or organized in tabs. The list layout is different among the apps. One software uses swipe gestures to change the current item being displayed. It also depends on the application whether to show or not user feedback and product ratings.

- One app allows the user to save searches.
- The system used to retrieve data is different.
- Two apps allow the registration of items inside the app.
- Four apps can filter items being displayed. All solutions are similar in the form of displaying the filter, it is a menu accessible by a button.
- The drawer menu content is different. The categories tree view can be on the drawer and each app has some platform specific options.
- The groups of item layout options available to the user are different. A single type of layout has small differences between apps.

4.2 Feature Model

Having studied the similarities and variability of the apps, it is possible to detail which features must be present in the SPL. The following is a list of mandatory features that the SPL has to deliver in order to be an efficient product catalog solution. The SPL must:

- Have a category organization, in form of tree. This is used on all apps as the app navigation backbone. The SPL must allow the grouping of categories so that the app can display a group of categories in different places.
- Have the item detail visualization fully customizable, in order to allow the use of the catalog by any customer.
- Provide means of search and filtering. It is preferable that the solution is smart enough to learn which options are available for a single SPL instantiation and display the filter and search options accordingly.
- Be able to create apps that work online and or offline.
- Provide a simple way of communication with other data providing applications.
- Allow items creation from inside the app.
- Provide various layout options when displaying groups of items.

Figure 4.1: Feature tree.

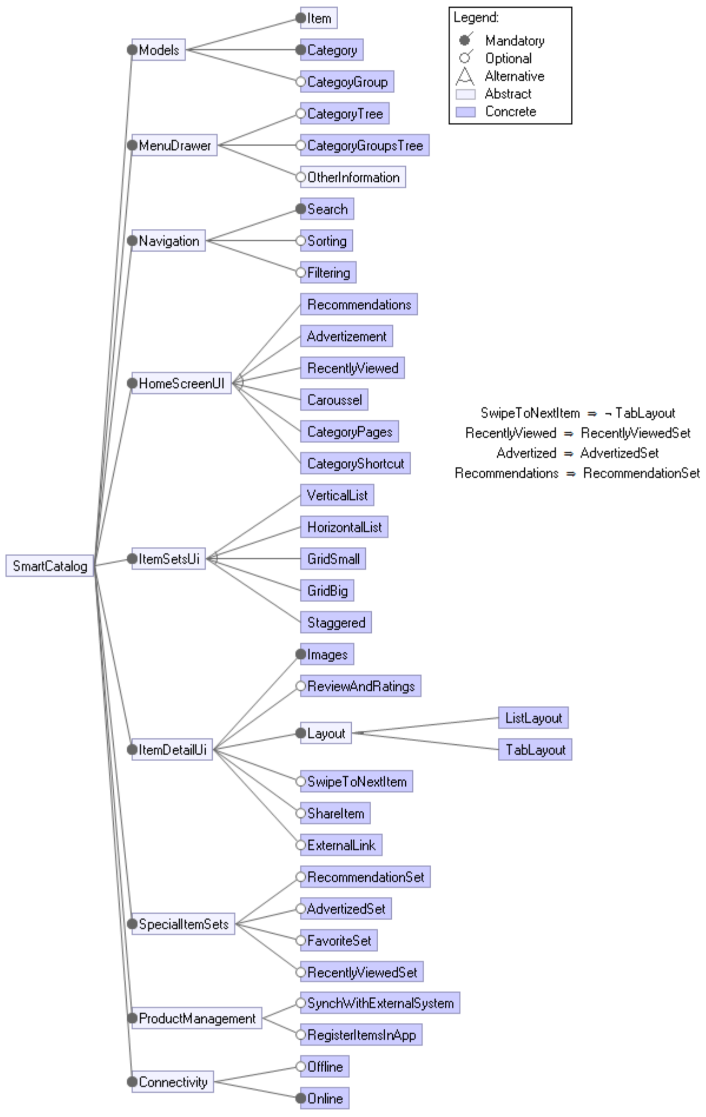


Figure 4.1 shows the identified features organized in a feature model, together with applicable restrictions.

5 DOMAIN DESIGN AND REALIZATION

This chapter describes the chosen architecture plan and its implementation details for the Android OS environment.

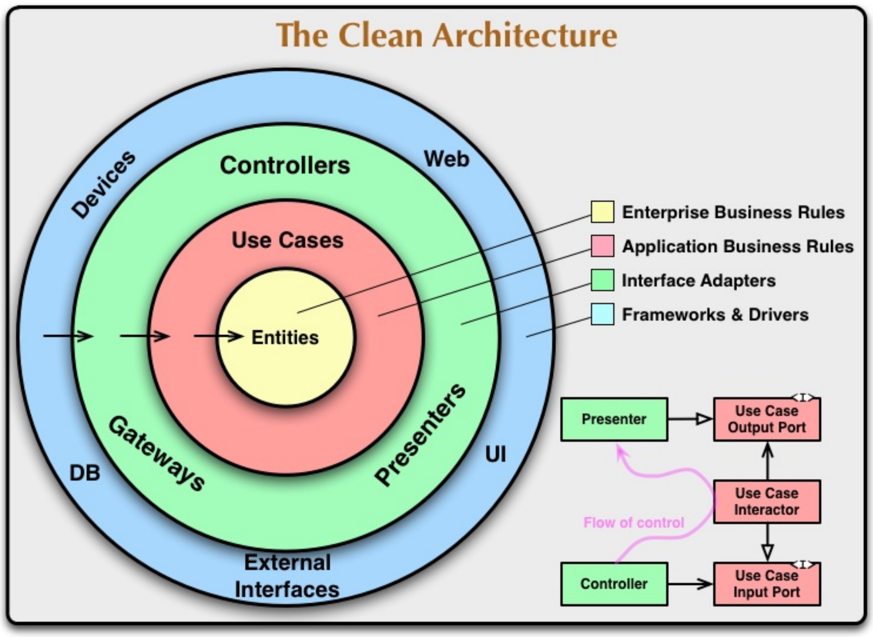
5.1 Domain Design

In order to take advantage of the commonality of the analyzed applications, it is necessary to create the core of the SPL with an architecture that can separate concerns efficiently. There is a large range of architecture ideas that share the objective of separation of concerns (Clean Architecture, Hexagonal Architecture (COCKBURN, 2005), Onion Architecture (PALERMO, 2008), DCI from Coplien and Bjørnvig (2010) and BCE from Jacobson et al. (1993)). All of them split the software into different layers where at least one layer is for the interface and another is for business rules. The Clean Architecture as described in Martin (2012) is the conjunction of all other previously cited architectures. This is the chosen base architecture for this Software Product Line. The Clean Architecture aims to be:

- *Independent of Frameworks*. It must not depend on specific libraries nor frameworks. The architecture must be self-contained and allow the use of different frameworks and libraries that are specific to each instance of the SPL.
- *Testable*. Each part of the software must be testable individually. UI, Database and Web Servers must not depend on each other and core elements should not depend on these factors.
- *Independent of UI*. The UI can change rapidly following new trends on user experience and new user interface guidelines. More soever each SPL customer will have its unique requirements regarding UI.
- *Independent of Database*. The system must be independent of the persistent storage system. Some solutions may require only local persistence whereas other may require online synchronization with other systems. The architecture must be able to save state independently of the system used to do it.
- *Independent of any external agency*. The business rules of the system must not know anything about the outside world.

The basic principle of the Clean Architecture is *The Dependency Rule* where it

Figure 5.1: The Clean Architecture.



Source: Martin (2012)

states that code dependency only points inwards. It means that layers from the inside can not know anything from layers from the outside. In particular, names of any software entity created on outer rings must not be mentioned in inner rings. Data formats used on the outside can not be used in the inside layers whereas outer circles changes do not impact inner circles. The circles of Figure 5.1 represents the layers of the architecture.

The lower right of the Figure 5.1 shows an example interaction between layers. The flow of data begins from the controller of the interface adapter and travels to the use cases which deliver data back to interface adapters into the presenters. Note as well the source code dependency represented by the arrows. In this interaction, there is an apparent dependency contradiction where a use case must deliver data to presenters, thus inflicting in main dependency rule that code must only depend inwards. This is solved with the Dependency Inversion Principle where interfaces and inheritance relationships are organized in a way that code dependency is different from the flow of control on layer boundaries. For example, the use case layer defines an interface making the methods available for use case usage, and the Adapters implement this interface giving the real implementation. This way the use cases do not need to know how the method will run on the adapter that is outside of its scope. The same technique is used on all layer boundaries. Polymorphism is thoroughly used to create code dependencies that are different from data control flow.

Key aspects of the domain architecture

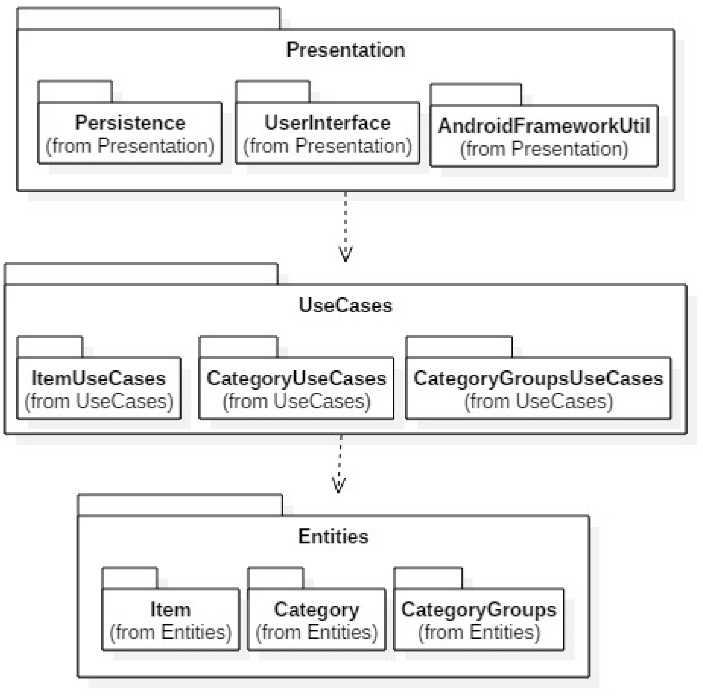
The Android framework which is used for app development introduces many limitations on the application architecture. The basic architecture imposed by the Android framework has flaws, such as views with very large responsibilities and complex logic when dealing with asynchronous tasks. A desirable architecture should be efficient while at the same time be adaptable to existing Android limitations in order to mitigate the inherited Android architecture problems. The principles of the Clean Architecture precisely fulfill the requirements of the architecture for the SPL core. This architecture can adapt to the restriction of the Android framework while allowing parts of the software to be built independently. For example, use cases can be built in the core of the SPL, without knowing where the data is saved or retrieved and removing complexity from the views. This is the reason to use the Clean Architecture for the Smart Catalog core architecture.

In order to fulfill the dependency rule stated by the clean architecture, dependency injection (DI) is used. DI allows client code to delegate the responsibility of providing dependencies to external code (such as the database service) to external entities (the injector). This is extremely useful when creating basic presentation classes in the core. For example, the presentation layer of the core may require at some point a list of all entities, although it does not need to know how such entities are stored or delivered. Each instance of the SPL is an injector, being responsible for delivering the correct dependencies to the core, allowing the core to work with interfaces or abstract classes.

As described in previous chapters, the core part of the SPL contains the software assets that are the base for all different product instances. After the existent software analysis and architecture definition, it is possible to define which features are available in the core and thus available for reuse in new applications. The core of the Smart Catalog SPL contains entities, the use cases and implementation of views that are common to many applications analyzed as well as many utility code.

In the Smart Catalog SPL, the entity layer of the Clean Architecture is implemented with interface models of `Items`, `Categories`, and `CategoryGroups`. The use cases layer is implemented with concrete use cases for each of the existing models, relying on repository interfaces for each model. The interface adapters are implemented on the same level as the framework and drivers layer, in the presentation layer. This outermost layer has implementations for the persistence systems, Android framework ex-

Figure 5.2: Core package diagram.


Presentation
Persistence (from Presentation)
UserInterface (from Presentation)
AndroidFrameworkUtil (from Presentation)
UseCases
ItemUseCases (from UseCases)
CategoryUseCases (from UseCases)
CategoryGroupsUseCases (from UseCases)
Entities
Item (from Entities)
Category (from Entities)
CategoryGroups (from Entities)

tensions and utilities, and implementation of user interface elements. Figure 5.2 shows the packet diagram of the core.

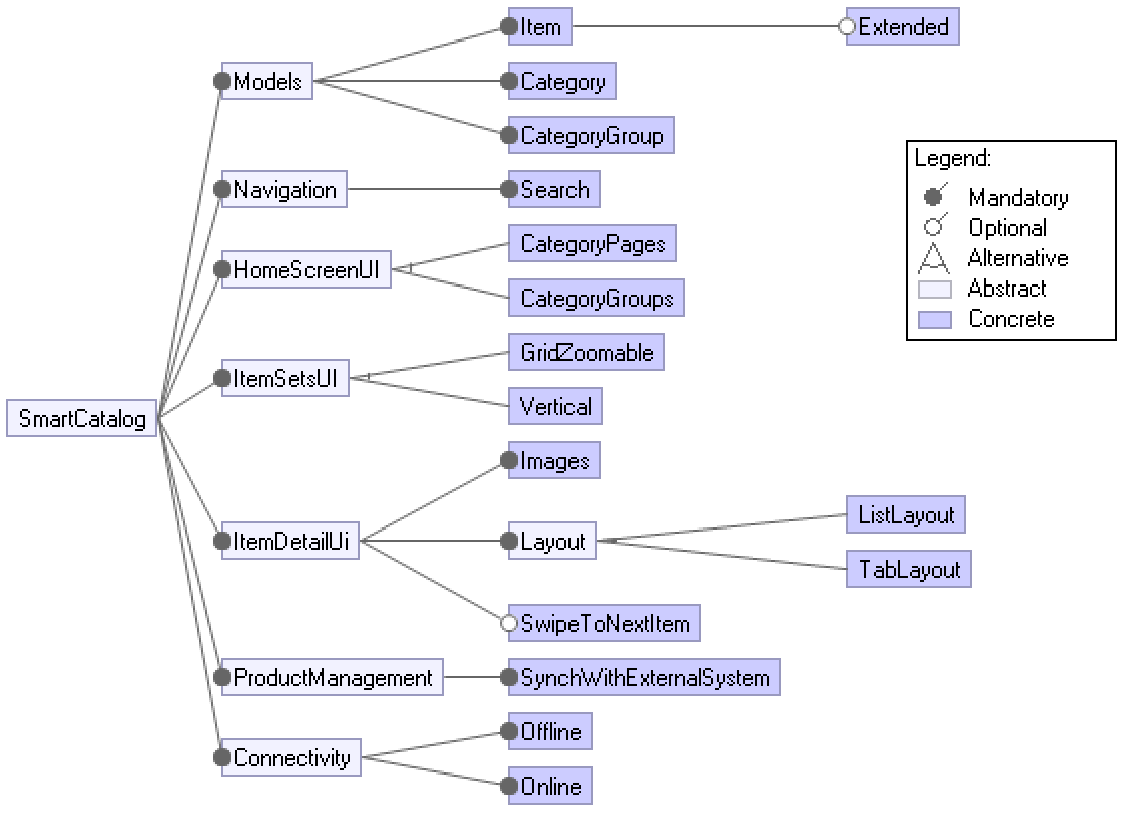
5.2 Domain Realization

This section describes the implementation of the architecture defined in the previous section.

The development of the SPL was entirely made on the Android Studio IDE which is the officially recommended tool for developing Android applications. The IDE has the ability to manage various application modules that can depend on each other to create an app. The core part of the SPL is in a specific module named core. The instances of the SPL are created each on a new module that depends on the core module. Features are only implemented on the SPL core module and the configuration is made on each instance module. This separation is necessary due to the Android environment.

Besides of the feature development, a large development effort is necessary to create the architecture organization of the SPL. This is why at this stage, some features of the SPL were removed due to time constraints. Figure 5.3 shows the features that are

Figure 5.3: Implemented feature tree.



implemented in this work.

The EXTENDED feature is an example implementation of what could be the abstract ITEM MODEL feature defined in the analysis. Each instance of the SPL may have a different item model with different fields, depending on the client specification. A sample specification is represented with the EXTENDED feature. The implemented features aim at the core functionality of the Smart Catalog and a sample of features that exemplify how the variability is implemented on software product lines.

In the implementation the architecture was simplified in three layers: *Entities* that contain the entities, *UseCases* containing use cases and *Presentation* responsible for user interface elements, persistence, and other utilities.

Entities

All the models defined in the core are interfaces with a method that must return the id of the entity and other methods related to the other entities fields. This interface is the type of data that is used to transfer data on crossing points. For example, if a UI layer needs a specific field of the entity, it will use methods from the entity interface. With this technique, the data does not need to be transformed to different objects when being transferred between layers. Listing 5.1 shows the code that defines an item model.

Listing 5.1: Entity example

```
public interface ItemBasicModel extends ItemId {
    @Override String getStringId();
    String getCategoryStringId();
    CategoryModel getCategory();
    String getName();
    float getPrice();
    String getDescription();
    String getImageUrl();
}
```

The defined features model has a mandatory ITEM MODEL feature and an optional EXTENDED MODEL feature that is related to optional fields of the item model that a SPL instance may require. In this work, only one extension of the item model was developed. This is done by creating another entity model with the same identification property and the additional fields. The two entities models are linked together by a one-to-one relationship defined by the id property. Listing 5.2 shows the extended item model.

Listing 5.2: Entity extended example

```
public interface ItemExtendedModel extends ItemId {
    @Override String getStringId();
    ItemBasicModel getItemBasicModel();
    boolean getIsPromoted();
    boolean getIsSale();
    boolean getIsAssembled();
    boolean getIsNew();
    float getPreviousPrice();
    boolean mustShowPreviousPrice();
    java.util.Date getCreationDate();
}
```

Use Cases

This layer accesses the persistence systems and delivers entities to the presentation layer. Since the persistence system must be implemented on the presentation layer, as per the Clean Architecture, the repository pattern is used. Each model has a repository interface described in the use cases layer, this allows the use cases implementation to use the repository methods without knowing how the real implementation is done. The concrete implementation of the repository is delivered by the dependency injection system. Listing 5.3 shows part of the implementation of the items use cases and Listing 5.4 shows the `ItemBasicRepository` interface. It is possible to notice that the repository is not instantiated by the use case, but injected by the dependency injection system.

Listing 5.3: Use case example

```
public class ItemBasicUseCases {
    @Inject  ItemBasicRepository itemBasicRepository;

    @Inject public ItemBasicUseCases(){}

    public Observable<ItemBasicModel> getAll() {
        return Observable.create(
                subscriber -> {
                    for (ItemBasicModel item :
                            itemBasicRepository.loadAll()) {
                        subscriber.onNext(item);
                    }
                    subscriber.onCompleted();
                }
        );
    }
    ...
}
```

Listing 5.4: Repository example.

```
public interface ItemBasicRepository {
    List<? extends ItemBasicModel> loadAll();
    ItemBasicModel load(String itemId);
    List<? extends ItemBasicModel> query(String query);
    void insert(ItemBasicModel itemBasicModel);
    void insertAll(List<ItemBasicModel> itemBasicList);
    void remove(String itemId);
    void removeAll();
}
```

The core also has use cases implementation and a repository interface for the EXTENDED ITEM MODEL, CATEGORY MODEL, and CATEGORY GROUP MODEL features.

Presentation

The presentation layer is split into three main packages: persistence, user interface, and android framework utilities.

Persistence

The persistence system is implemented on the presentation layer, it is created by implementing all the repositories interfaces defined by the use cases layer and all models defined by the entities layer. In this work, there is only one persistence implementation, which fetches data from an external system and saves it on the device to allow offline usage. With this, the offline and online features are set as mandatory on the feature model. Future iterations of the Smart Catalog SPL can easily implement new persistence systems without changing other parts of the core since it is only necessary to implement the interfaces.

Android Framework Utilities

This package contains utility methods related to the Android framework. This package also contains the base classes for the Model View Presenter (MVP) pattern used by the user interface package.

User Interface

The views of this SPL are organized with the MVP pattern. The usage of this pattern is intended to decrease the complexity of the Android framework usage by implementing basic code related to Android in the core framework and allowing SPL features implementation to focus solely on the feature itself.

Each available feature of the SPL that is a user interface, has a specific package that contains its implementation. For example, the HOME SCREEN UI feature is implemented in the package `core.presentation.ui.homescreen`. In this specific case where there are two sub-features available, each is implemented in a sub package, `homescreen.categorypages` and `homescreen.categorygroups`, extending the same base class. The implementation of the feature CATEGORY GROUPS as home screen is available in Appendix A.1.

The views may have events (such as click for details in an item) where new views need to be created. The interface `BaseAppDisplayFactory` is responsible for giving the variability of the UI elements that are created when such actions are performed. There are sub interfaces for each UI feature described in the feature model. The methods of these interfaces are used to start views related to the features. Listing 5.5 shows the definition of those interfaces.

Listing 5.5: Views configuration.

```
public interface BaseAppDisplayFactory {
    interface HomeScreenConfigurator{
        void startHomeScreen(AppCompatActivity activityBase);
    }

    interface ItemSetsConfigurator{
        ItemSetsCallbacks provideItemSetFragment(String searchQuery,
            boolean isCategoryIdQuery);
    }

    interface ItemDetailConfigurator{
        void switchToItemView(FragmentActivity fromActivity, String[]
            categoriesIds, int position);
        ItemDetailFragmentBase getItemDetailFragment(String itemId);
    }
}
```

To exemplify, the creation of the home screen is explained. After the initial splash screen of the app is showed to the user, the main screen must be loaded. The listing 5.6 shows the code used to perform the action.

Listing 5.6: UI action.

```
@Inject BaseAppDisplayFactory appDisplayFactory;
...
    baseAppDisplayFactory.startHomeScreen(this);
```

```
...
```

The instance of `AppDisplayFactory.HomeScreenConfigurator` that is present inside the `appDisplayFactory` field is injected by the dependency injection system. The `AppDisplayFactory.HomeScreenConfigurator` interface, according to the feature model, has two implementations described in Listing 5.7 and Listing 5.8, which can be chosen from on the configuration of new SPL instances.

Listing 5.7: Home screen factory one.

```
public class GategoryGroupsHome implements AppDisplayFactory.
    HomeScreenConfigurator {
    @Inject public GategoryGroupsHome() { }
    @Override
    public void startHomeScreen(AppCompatActivity activityBase) {
        Intent intent = new Intent(activityBase, MainActivityGroup.
            class);
        activityBase.startActivity(intent);
        activityBase.finish();
    }
}
```

Listing 5.8: Home screen factory two.

```
public class GategoryPagesHome implements AppDisplayFactory.
    HomeScreenConfigurator {
    @Inject public GategoryPagesHome() { }

    @Override
    public void startHomeScreen(AppCompatActivity activityBase) {
        Intent intent = new Intent(activityBase, MainActivityTabbed.
            class);
        activityBase.startActivity(intent);
        activityBase.finish();
    }
}
```

The dependency configuration system must be started at some point of the software execution. Android applications usually have a starting screen that is called splash screen. This view is the first view created by the application and is showed to the user while the application loads and configures the dependency system. Listing 5.9 shows the code of the splash screen used in the Smart Catalog SPL, which is used to start the dependency configuration system.

Listing 5.9: Dependency system start.

```
public abstract class SplashScreenBase extends AppCompatActivity {
    @Inject ItemBasicRepository itemBasicRepository;
    @Inject CategoryRepository categoryRepository;
    @Inject CategoryGroupRepository categoryGroupRepository;
    @Inject BaseAppDisplayFactory baseAppDisplayFactory;
    @Inject ItemBasicUseCases itemBasicUseCases;
    @Inject CategoryUseCases categoryUseCases;
    @Inject CategoryGroupUseCases categoryGroupUseCases;
    @Inject FirebaseAuth.AuthStateListener authStateListener;

    protected static SplashScreenBase instance = null;

    protected void onCreate(Bundle savedInstanceState){
        super.onCreate(savedInstanceState);

        if (instance == null) {
            synchronized (SplashScreenBase.class) {
                if (instance == null) {
                    instance = this;
                    instance.injectMe(instance);
                }
            }
        }
        baseAppDisplayFactory.startHomeScreen(this);
    }

    public static SplashScreenBase getInstance() {
        return instance;
    }

    protected abstract void injectMe(SplashScreenBase splashScreen);
}
```

The view classes of the Android framework are instantiated by the Android framework itself, thus not allowing the views to be created with the dependency injection system. The solution used is to make the splash screen a singleton that when created by the Android framework, injects all the fields of the SPL. The `onCreate` method is called by the Android framework, its implementation injects all fields and starts the singleton reference to make the injected fields available for other views.

An additional splash screen class that extends the base class has to be used when the EXTENDED ITEM MODEL feature is selected. This class extends the base splash screen class and adds injected fields related to this specific feature. In this way, the code related to the extended feature is only used when this feature is selected. Listing 5.10 describes the additional splash screen class.

Listing 5.10: Dependency system start extended.

```java
public abstract class SplashScreenExtended extends SplashScreenBase{
    @Inject public ItemExtendedRepository itemExtendedRepository;

    @Inject public ItemExtendedUseCases itemExtendedUseCases;

    protected void injectMe(SplashScreenBase splashScreenBase){
        injectMeInner(splashScreenBase);
        injectMeInner(this);
    }

    protected abstract void injectMeInner(SplashScreenBase
        splashScreenBase);
    protected abstract void injectMeInner(SplashScreenExtended
        splashScreen);

    public static SplashScreenExtended getInstance() {
        return (SplashScreenExtended) instance;
    }
}
```

Note that the `injectMe` method is abstract. This is because the configuration call must be made by each of the instances. The instances of the SPL must extend this method in order to configure the dependency system. The configuration of the dependency injection system, which is the configuration of a SPL instance, is explained on the next chapter.

6 APPLICATION ENGINEERING

This chapter describes the process of creating final product instances of the SPL, describing a general process and using a new app to explain it by example.

6.1 Configuration knowledge

When a new instance of the SPL needs to be created, a sequence of steps must be followed:

- The first step is to create a new module on the Android Studio IDE and make it depend on the existing core module.
- The DI configuration files must be generated and configured according to the features selected from the feature model.
- Configure the Android manifest file to register the views chosen from the feature model.

A new instance requires the creation of a new Android Studio IDE module and only tree configuration classes, besides of the Android Manifest file and build file. The configuration files are classes that configure the dependency injection system.

The configuration begins with the creation of an `ApplicationComponent`, which Listing 6.1 describes. The code of this file is a copy on all instances of the SPL, although, it must be created on each instance of the SPL due to requirements of packages names needed by the dependency injection system.

Listing 6.1: Dependency injection component.

```
@Singleton
@Component(modules = ApplicationModule.class)
public interface ApplicationComponent {
    Context context();
    void inject(SplashScreenBase splashScreenBase);
}
```

The next file to create is the `ApplicationModule` which contains the configuration of the `ApplicationComponent`. The configuration is done with `provides` annotated methods. Each of these methods is responsible for instantiating the correct implementation of the function's return type. For example, Listing 6.2 configures the system to use the

`ItemBasicGreendaoRepository` concrete implementation for the `ItemBasicRepository` interface.

Listing 6.2: Dependency injection configuration example.

```
@Provides @Singleton
ItemBasicRepository provideItemBasicRepository(
    ItemBasicGreendaoRepository repository){
     return repository;
}
```

The feature ITEM is a mandatory feature, thus the configuration explained before is mandatory on all instances of the SPL. This configuration is necessary in order to include the associated assets of the feature in the derived product. This is also the case for many other mandatory features used by the system. Having a configuration point even for mandatory features prepares the SPL for new features that may be developed on the future. For example, if a new repository that can fetch data from a XML file is created, the new feature can be easily inserted just by changing an existing configuration point.

The optional feature EXTENDED ITEM MODEL requires a configuration method only when it is selected. When selected, the system uses the classes related to that feature, and when it is not selected, the classes are not used. Listing 6.3 describes the optional feature configuration.

Listing 6.3: Dependency injection optional feature.

```
// Must be provided only if Extended feature is selected
@Provides @Singleton
ItemExtendedRepository provideItemPromotedRepository(
    ItemExtendedGreendaoRepository repository){
     return repository;
}
```

There are tree configuration points that vary according to selected features. For example, when configuring the feature HOME SCREEN UI, if the sub-feature CATEGORY GROUPS is selected, then the class `CategoryGroupsHome` must be provided. If feature CATEGORY PAGES is selected then, the class `CategoryPagesHome` must be provided. Listing 6.4 describes all the possible configurations according to a boolean logic, where the left side of the expression are the features selected in the feature tree and the right side is the name of the class that needs to be provided.

Listing 6.4: Dependency injection feature choice.

```
// CategoryGroups -> CategoryGroupsHome
// CategoryPages -> CategoryPagesHome
@Provides @Singleton
AppDisplayFactory.HomeScreenConfigurator homeScreenConfigurator(
   GategoryGroupsHome homeScreenConfigurator){
    return homeScreenConfigurator;
}

// Extended && SwipeToNextItem -> SwipeListExtendedDetail
// Extended && !SwipeToNexItem  -> ListExtendedDetail
// !Extended && SwipeToNexItem -> SwipeListDetail
// !Extended && !SwipeToNexItem -> ListDetail
@Provides @Singleton
AppDisplayFactory.ItemDetailConfigurator itemDetailConfigurator(
   ListDetail itemDetailConfigurator){
    return itemDetailConfigurator;
}

// Extended && GridZoomable -> GridZoomItemExtendedSet
// Extended && Vertical -> VerticalItemExtendedSet
// !Extended && GridZoomable -> GridZoomItemSet
// !Extended && Vertical -> VerticalItemSet
@Provides @Singleton
AppDisplayFactory.ItemSetsConfigurator itemSetsConfigurator(
   VerticalItemSet itemSetsConfigurator){
    return itemSetsConfigurator;
}
```

The last file to be created is the splash screen extension, this file is responsible for instantiating the configuration and injecting it. Listing 6.5 explains the implementation.

Listing 6.5: Dependency injection splash screen.

```
public class BasicSplashScreen extends SplashScreenBase{
    @Override
    protected void injectMe(SplashScreenBase splashScreen) {
        DaggerApplicationComponent.builder()
                .applicationModule(new ApplicationModule(getApplication
                   ().getApplicationContext()))
                .build()
                .inject(this);
    }
}
```

Note that the extended class must be `SplashScreenBaseExtended` if the EXTENDED ITEM MODEL feature is selected for this instance.

The Android manifest file should also be configured by referring the views classes that are used by the SPL instance. This configuration step is not explained because it is specific to the Android environment.

6.2 Ditlanta Catalog - An Instance of the Smart Catalog SPL

This section describes the instantiation process of a new Ditlanta Catalog app. The same app analyzed to retrieve the features of the SPL is now instantiated with the features created by the SPL. One feature is selected differently from what is available in the original Ditlanta Catalog app in order to test if the other analyzed applications can also have reproduced features.

This app requires offline availability because its usage is made in regions with low internet connectivity. Although, the app must synchronize with an external database in order to keep its data updated.

The Ditlanta business works with a large variety of items that can be grouped in segments. This is mirrored to the app by creating a home screen that has shortcuts for those group of categories. Since groups of categories are used on the home screen, the menu drawer of categories is organized only with categories and not groups.

The layout that will be available when displaying a set of items is a vertical list layout, which is different from the original version. The layout used when displaying details of an item is the list layout. Since the detail view of an item is very simple, this app allows the usage of the SWIPE TO NEXT ITEM feature.

Figure 6.1 shows the selected features of this instance from the implemented features tree. The result of the implementation can be verified on Figures 6.2, 6.3, and 6.4. The configuration file used by this instance is available in Appendix A.2.

Figure 6.1: Ditlanta Catalog app features selection.

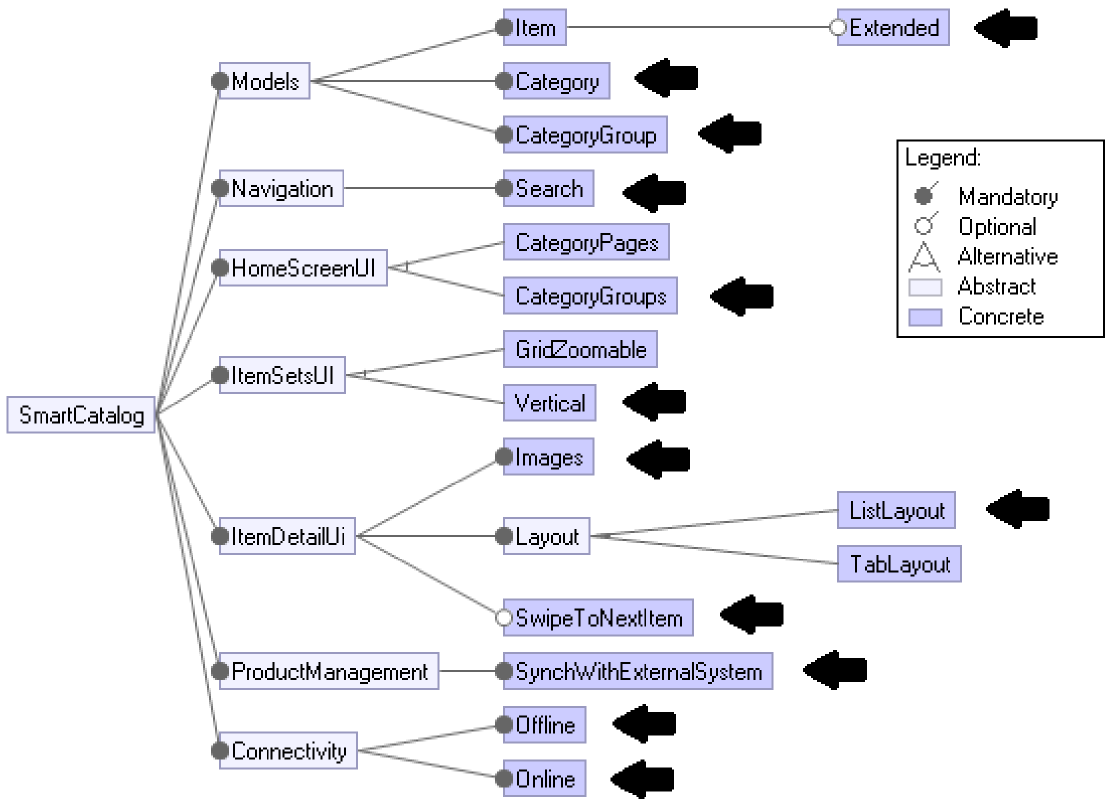


Figure 6.2: Ditlanta Catalog instance home screen screenshot.

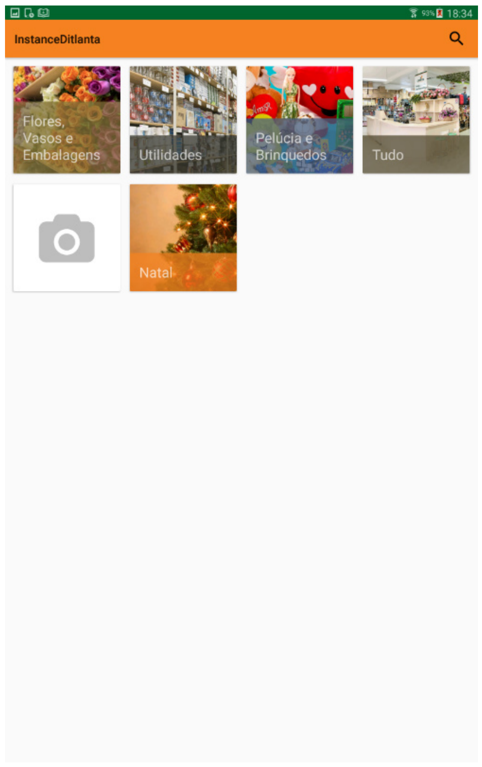
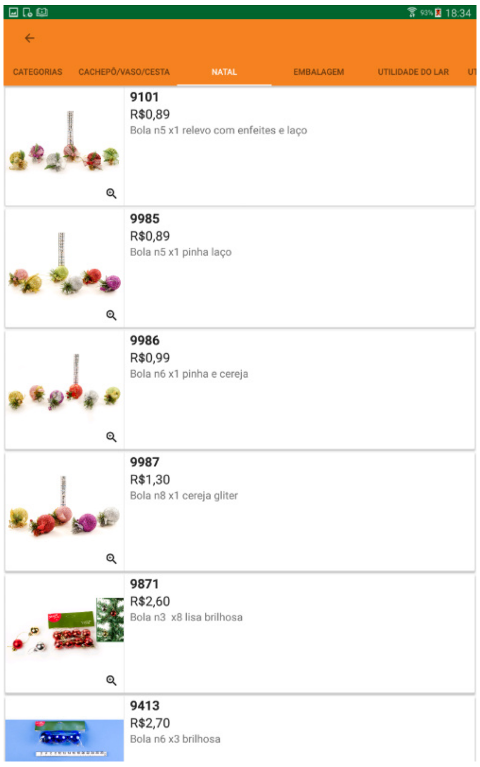


Figure 6.3: Ditlanta Catalog instance item set screenshot.



9101
R$0,89
Bola n5 x1 relevo com enfeites e laço

9985
R$0,89
Bola n5 x1 pinha laço

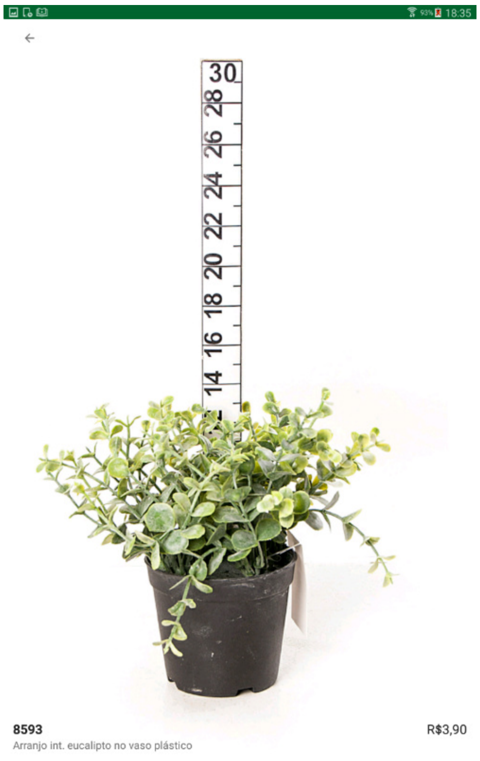
9986
R$0,99
Bola n6 x1 pinha e cereja

9987
R$1,30
Bola n8 x1 cereja gliter

9871
R$2,60
Bola n3 x8 lisa brilhosa

9413
R$2,70
Bola n6 x3 brilhosa

Figure 6.4: Ditlanta Catalog instance item detail screenshot.



8593
Arranjo int. eucalipto no vaso plástico
R$3,90

7 DISCUSSION

This chapter discusses in more details the experience of developing the Smart Catalog software product line

The SPL architecture planning and creation was underestimated in the initial phases of development. It happens that the architecture planning and creation required more work time than the feature analysis and development phases. This reinforces the idea brought from the literature that a SPL creation has greater up-front cost, and this knowledge should be taken into account when developing new software product lines.

The Android environment has peculiarities and restrictions. It is difficult to manage the framework activities and fragments, which are the basic views elements. The classes related to these views have specific and complicated life cycle and are instantiated by the Android environment. This required the special setup of the dependency injection, provided by the splash screen.

There is as well a problem with the Android manifest file, which is a configuration file required for all Android applications. This file needs to be recreated in each instance of the SPL whereas, on the environment of this work, this file have a similar code on all instances. A simple mechanism to create inheritance of this configuration file is not available. On the other side, the Gradle build system that is used to make the builds of Android applications turned out to be very useful and customizable. Information that was similar to the core and new instances could be centralized in one single configuration file.

Some tools available for Android can only work correctly in modules that are of the application type. The core of the SPL must be implemented in a library type module which hinders the usage of some libraries. It is not possible to make an app module depend on another app module in Android studio, an instance module must be of type application and depend on the core library module.

A new SPL instance creation is simple, only requiring the creation of configuration files, which are similar in every instance of the SPL. Besides, if a new instance requires a feature that is not yet present in the SPL core, it can create the implementation and make the configuration use this instance implementation instead of the core implementation. Although, further cycles of development should generalize the feature and implement it in the core.

The instantiation process of this SPL, besides of being simple, is made manually, thus, prone to errors. There are currently available many tools for configuration knowledge and product derivation, nevertheless, these tools are recent and do not focus on the Android environment. The Android environment requires many files to be set up which are not common in other development environments, these specificities should be mapped to existing tools.

Further instances of the SPL are expected to trigger new core features. The ideal process is to develop a feature into a single instance and when it is identified that this feature is needed in another instance, that feature would be moved to the core and reused in the new instance.

In comparison with the development of the features in the previous Ditlanta Catalog application, the adaptation of existing features into the SPL platform of this work required a very increased amount of development time. This is comprehensible because, on the SPL platform, the features must be implemented according to the architecture defined in this SPL in order to allow it to work in conjunction with all the features in the feature tree. Moreover, it is expected that new features required by customers of the SPL require more development time due to the adaptation of the features into the existing SPL core. On the other side, the creation of new similar products is drastically smaller when using the SPL. Instead of copying the existing application and adapting the existing features in an ad hoc manner, leading to code duplication and substantial development effort, one can now simply make a different configuration of the SPL and generate a newly adapted software solution. In this way, features are not duplicated, allowing the evolution and maintenance of the products trough the SPL platform.

8 CONCLUSION

Software product line engineering creates a platform of common reusable software parts that are used to deliver new products and reduces the time-to-market of new products after initial steps of creation. In this work, five existing Android catalog applications were analyzed and derived a new software product line of catalog applications. The implemented features in this work created the mandatory features necessary for catalog applications and provided an example of how variability can be implemented. This new software product line is at the first stage of creation and contains a small features subset of the analyzed applications, yet, this SPL can generate new similar catalog applications with reduced production costs.

The core and architecture of this SPL still have room for improvements. Many features identified in the analysis phase were not implemented but should be, since its necessity was already identified.

This work is the initial phase of the construction of the Smart Catalog product line. With the features developed in this work, it is possible to generate many different catalogs applications easily, which can adjust to some customers. The currently implemented features can have sixteen possible configurations, but the full feature model defined by the apps analysis could have more than five thousand possible configurations. With bigger features set, more customers can be attended efficiently with low time-to-market.

The variability implementation on a SPL is a complex task. Although an example implementation is described in this work, there are many different approaches available for variability implementation. The complexity of implementation increases with new features being added to the platform, thus, the creation of more features is intended on future cycles of the Smart Catalog SPL implementation.

This project improved the researcher knowledge on SPL immensely. It gave powerful insights on the analysis and implementation of software product lines. Difficulties were found during the process and lessons were learned, as well as technologies used in the Android environment.

The Ditlanta Catalog app instance is already being used by the real company Ditlanta. This gives trust to the solution delivered. The SPL aspect of Smart Catalog can be used to deliver specific needs of specific clients. With this in mind, the Smart Catalog SPL could be a business, and use SPL methodology as a market strength to deliver great individualized products for their customers while not requiring a lot of workforce and maintaining a high quality of products.

For a future work, the implementation of all features found by the analysis is intended, as well as the automation of the configuration and instantiation process. Moreover, further studies can be made on the development of this SPL in a multi-platform environment, which could aim the creation of Smart Catalog into different mobile platforms.

REFERENCES

APEL, S. et al. **Feature-Oriented Software Product Lines**. [S.l.]: Springer, 2013.

CLEMENTS, P.; NORTHROP, L. **Software Product Lines: Practices and Patterns**. [S.l.]: Addison-Wesley Professional, 2001.

COCKBURN, A. Hexagonal architecture. (website). 2005. Available from Internet: <http://alistair.cockburn.us/Hexagonal+architecture>.

COPLIEN, J.; BJØRNVIG, G. **Lean architecture: for agile software development**. [S.l.]: John Wiley & Sons, 2010.

COUTO, M. V.; VALENTE, M. T.; FIGUEIREDO, E. Extracting software product lines: A case study using conditional compilation. In: IEEE. **Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on**. [S.l.], 2011. p. 191–200.

IDC, R. I. Smartphone os market share, 2015 q2. (website). 2015. Available from Internet: <http://www.idc.com/prodserv/smartphone-os-market-share.jsp>.

ISLAM, M. R.; ISLAM, M. R. I.; MAZUMDER, T. A. Mobile application and its global impact. In: INTERNATIONAL JOURNALS OF ENGINEERING & SCIENCES, 2010, Rawalpindi, Pakistan. **Proceedings...** [S.l.]: IJENS, 2010. p. 72–78.

JACOBSON, I. et al. **Object-Oriented Software Engineering-A use-case Driven Approach**. [S.l.]: Addison-Wesley, New York, 1993.

KRUEGER, C. Easing the transition to software mass customization. In: SPRINGER. **International Workshop on Software Product-Family Engineering**. [S.l.], 2001. p. 282–293.

LINDEN, F. J. van der; SCHMID, K.; ROMMES, E. **Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering.** [S.l.]: NJ, USA: Springer-Verlag New York, Inc, 2007.

MARTIN, R. C. The clean architecture. (website). 2012. Available from Internet: <https://8thlight.com/blog/uncle-bob/2012/08/13/the-clean-architecture.html>.

MATINLASSI, M. Comparison of software product line architecture design methods: Copa, fast, form, kobra and qada. In: IEEE COMPUTER SOCIETY. **Proceedings of the 26th International Conference on Software Engineering**. [S.l.], 2004. p. 127–136.

PALERMO, J. Onion architecture. (website). 2008. Available from Internet: <http://jeffreypalermo.com/blog/the-onion-architecture-part-1/>.

PARNAS, D. L. On the design and development of program families. **IEEE Trans. Softw. Eng.**, IEEE Press, Piscataway, NJ, USA, v. 2, n. 1, p. 1–9, jan. 1976. ISSN 0098-5589. Available from Internet: <http://dx.doi.org/10.1109/TSE.1976.233797>.

POHL, K.; BÖCKLE, G.; LINDEN, F. J. van der. **Software product line engineering: foundations, principles and techniques**. [S.l.]: Springer Science & Business Media, 2005.

SCHMID, K.; RUMMLER, A. Cloud-based software product lines. In: ACM. **Proceedings of the 16th International Software Product Line Conference-Volume 2**. [S.l.], 2012. p. 164–170.

VOELTER, M.; GROHER, I. Product line implementation using aspect-oriented and model-driven software development. In: IEEE. **Software Product Line Conference, 2007. SPLC 2007. 11th International**. [S.l.], 2007. p. 233–242.

APPENDIX A — SOURCE CODE SAMPLES

A.1 Category Pages Feature Implementation

A.1.1 MainActivityBase

```
public abstract class MainActivityBase<P extends Presenter> extends
   MvpRxActivityBase<P> {
    public ProgressBar progressBar;
    public Toolbar toolbar;
    public BaseAppDisplayFactory baseAppDisplayFactory;
    private FirebaseAuthentication firebaseAuthentication;

    @Override
    protected void onPause() {
        super.onPause();
        firebaseAuthentication.setOnPause();
    }

    @Override
    protected void onResume() {
        super.onResume();
        firebaseAuthentication.setOnResume();
    }

    @Override
    @CallSuper
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        baseAppDisplayFactory = SplashScreenBase.getInstance().
           baseAppDisplayFactory;
        firebaseAuthentication = new FirebaseAuthentication(this,
           baseAppDisplayFactory);
        progressBar = (ProgressBar) findViewById(R.id.progressBar);
        toolbar = (Toolbar) findViewById(R.id.toolbar);
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        MenuInflater menuInflater = getMenuInflater();
        menuInflater.inflate(R.menu.menu_search, menu);
        MenuItem searchMenuItem = menu.findItem(R.id.menu_search);
        setupSearchMenu(searchMenuItem);
        return super.onCreateOptionsMenu(menu);
    }

    void setupSearchMenu(MenuItem searchMenuItem){
        SearchManager searchManager = (SearchManager) getSystemService(
            Context.SEARCH_SERVICE);
        SearchView searchView = (SearchView) MenuItemCompat.
            getActionView(searchMenuItem);
        searchView.setSearchableInfo(searchManager.getSearchableInfo(
            getComponentName()));
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        int itemId_ = item.getItemId();
        if (itemId_ == R.id.menu_switch_lock_task) {
            Utils.switchLockTasMode(this);
            return true;
        }
        return super.onOptionsItemSelected(item);
    }

    protected void setupToolbar() {
        setSupportActionBar(toolbar);
    }

    public void stopLoading() {
        progressBar.setVisibility(View.GONE);
    }
}
```

A.1.2 MainActivityCategoryPages

```
public class MainActivityCategoryPages extends MainActivityBase<
    MainActivityCategoryPagesPresenter> {
```

```
public TabLayout tabLayout;
public ViewPager viewPager;

private TabbedGalleryPageAdapter pagerAdapter;

@Override
protected void onCreate(Bundle savedInstanceState) {
    setContentView(R.layout.activity_main_tabbed);
    super.onCreate(savedInstanceState);

    progressBar = (ProgressBar) findViewById(R.id.progressBar);
    toolbar = (Toolbar) findViewById(R.id.toolbar);
    tabLayout = (TabLayout) findViewById(R.id.slidingTabLayout);
    viewPager = (ViewPager) findViewById(R.id.pager);
    afterViews();
}

public void afterViews(){
    setSupportActionBar(toolbar);

    pagerAdapter = new TabbedGalleryPageAdapter(
       getSupportFragmentManager(), baseAppDisplayFactory);

    presenterAfterView();
    setupToolbar();
    setupPager();
}

private void setupPager(){
    viewPager.setAdapter(pagerAdapter);
    tabLayout.setupWithViewPager(viewPager);
    tabLayout.setTabGravity(TabLayout.GRAVITY_CENTER);
    tabLayout.setTabMode(TabLayout.MODE_SCROLLABLE);
}

public void stopLoading() {
    progressBar.setVisibility(View.GONE);
    viewPager.setVisibility(View.VISIBLE);
}

public void addItem(CategoryModel suitCase) {
        pagerAdapter.addItem(suitCase);
    }
}
```

A.1.3 MainActivityCategoryPagesPresenter

```
public class MainActivityCategoryPagesPresenter extends Presenter<
    MainActivityCategoryPages> {

    private static int OBSERVABLE_ID = 0;

    CategoryUseCases categoryUseCases;

    private Observable<CategoryModel> categoryModelObservable;

    public MainActivityCategoryPagesPresenter(){
        categoryUseCases = SplashScreenBase.getInstance().
            categoryUseCases;
    }

    @Override
    protected void onCreatePresenter(Bundle savedState) {
        categoryModelObservable = ObservableHelper.setupThreads(
                categoryUseCases.getAll().cache());
    }

    @Override
    protected void onAfterViews() {
        makeSubcription();
    }

    private void makeSubcription() {
        restartable(OBSERVABLE_ID,
                () -> categoryModelObservable.subscribe(new Observer<
                    CategoryModel>() {
                    @Override
                    public void onCompleted() {
                        if (getView() != null)
                            getView().stopLoading();
                    }

                    @Override
                    public void onError(Throwable e) {
                        throw new RuntimeException(e);
                    }

                    @Override
                    public void onNext(CategoryModel categoryModel) {
                        if (getView() != null) {
                            getView().stopLoading();
                            getView().addItem(categoryModel);
                        }
                    }
                })
        );

        if (isUnsubscribed(OBSERVABLE_ID))
            start(OBSERVABLE_ID);
    }
}
```

A.2 Ditlanta app configuration module

```
@Module
public class ApplicationModule {
    private final Context context;

    public ApplicationModule(Context context) {
        this.context = context;
    }

    //--------------------------
    //mandatory provides

    @Provides  @Singleton
    Context provideApplicationContext() {
        return this.context;
    }
```

```
//Persistence for Item model
@Provides @Singleton
ItemBasicRepository provideItemBasicRepository(
   ItemBasicGreendaoRepository repository){
    return repository;
}

//Persistence for Category model
@Provides @Singleton
CategoryRepository provideCategoryRepository(
   CategoryGreendaoRepository repository){
    return repository;
}

//Persistence for CategoryGroup model
@Provides @Singleton
CategoryGroupRepository provideCategoryGroupRepository(
   CategoryGroupGreendaoRepository repository){
    return repository;
}

//Helper container with options about UI
@Provides @Singleton
BaseAppDisplayFactory provideAppDisplayFactory(AppDisplayFactory
   appDisplayFactory){
    return appDisplayFactory;
}

//External sync
@Provides @Singleton
public DatabaseReference provideFirebase(){
    FirebaseDatabase.getInstance().setLogLevel(Logger.Level.DEBUG);
    DatabaseReference firebase = FirebaseDatabase.getInstance().
       getReferenceFromUrl(BuildConfig.FIREBASE_URL);
    return firebase;
}

//External sync authentication
@Provides @Singleton
```

```
public FirebaseAuth.AuthStateListener authStateListener(
    LoginAuthStateListener authStateListener){
     return authStateListener;
}

//---------------------------
//Optional feature

// Must be provided only if Extended feature is selected
@Provides @Singleton
ItemExtendedRepository provideItemPromotedRepository(
    ItemExtendedGreendaoRepository repository){
     return repository;
}

//---------------------------
// Choice features

// CategoryGroups -> CategoryGroupsHome
// CategoryPages -> CategoryPagesHome
@Provides @Singleton
AppDisplayFactory.HomeScreenConfigurator homeScreenConfigurator(
    GategoryGroupsHome homeScreenConfigurator){
     return homeScreenConfigurator;
}

// Extended && SwipeToNextItem - > SwipeListExtendedDetail
// Extended && !SwipeToNexItem  - > ListExtendedDetail
// !Extended && SwipeToNexItem - > SwipeListDetail
// !Extended && !SwipeToNexItem - > ListDetail
@Provides @Singleton
AppDisplayFactory.ItemDetailConfigurator itemDetailConfigurator(
    SwipeListExtendedDetail itemDetailConfigurator){
     return itemDetailConfigurator;
}

// Extended && GridZoomable -> GridZoomItemExtendedSet
// Extended && Vertical -> VerticalItemExtendedSet
// !Extended && GridZoomable -> GridZoomItemSet
// !Extended && Vertical -> VerticalItemSet
@Provides @Singleton
```

```
    AppDisplayFactory.ItemSetsConfigurator itemSetsConfigurator(
        VerticalItemExtendedSet itemSetsConfigurator) {
        return itemSetsConfigurator;
    }
}
```