

Nik Zakirin

Development of an Application for Trade Item Data Verification

Helsinki Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Bachelor's Thesis

29 May 2015

Author(s) Title Number of Pages Date	Nik Zakirin Development of an Application for Trade Item Data Verification 52 pages + 1 appendix 29 May 2015
Degree	Bachelor Of Engineering
Degree Programme	Information Technology
Specialisation option	Mobile Programming
Instructor(s)	Peter Hjort, Senior Lecturer Samuli Mattila, VP Products
<p>Consumers rely on the accuracy of product information in order to make an informed purchase decision. The emergence of online grocery stores as well as the EU regulation on the provision of food information is driving the demand for product data to be made available online. In the interest of ensuring the integrity of the product data, a validation system for the present information is needed.</p> <p>The goal of this project was to design and develop a data verification application for trade item information. This project was carried out at Digital Foodie Oy and was commissioned by GS1 Finland.</p> <p>The system is a client-server application. The scope of this project was limited to the design and development of the client-side application running on the Apple iOS operating system.</p> <p>The result of the product development work in this final year project was the Foodie verification iPad application, which is fully operational and has been taken into production use by a team of product inspectors.</p>	
Keywords	iOS, mobile programming, Xcode, design patterns, Objective-C

Contents

List of Abbreviations	1
1 Introduction	2
2 Trade Item Information State of the Union	3
2.1 GS1 Organisation	3
2.2 Trade Item Classification	3
2.3 Global Trade Item Number	4
2.4 European Union Regulation	5
3 Design and Implementation	7
3.1 Data Model	7
3.1.1 Inspection Task Manager	8
3.1.2 Inspection Batch	8
3.1.3 Inspection Task	10
3.1.4 Product	11
3.2 System Architecture	13
3.2.1 Model View Controller	13
3.2.2 Singleton	16
3.2.3 Strategy	17
3.2.4 Observer	19
3.3 Key Design Strategies	20
3.3.1 Image Loading	20
3.3.2 Persistence	22
4 Foodie Verification Application	24
4.1 Inspector Use Cases	24
4.2 Functional Division of Foodie Verification	25
4.2.1 Access Control	26
4.2.2 Batch Management	27
4.2.3 Product Collection	29
4.2.4 Product Inspection	31
4.2.5 Product Reception	33
4.2.6 Product Discovery	35
4.2.7 Analytics	37
4.3 Client–Server Functionality	38

5	Usability	39
5.1	Learnability	39
5.2	Efficiency	43
5.3	Memorability	44
5.4	Errors	45
5.5	Satisfaction	46
6	Discussion	47
6.1	Results	47
6.2	Challenges and Solutions	47
6.3	Future Development	48
7	Conclusions	50
	References	51
	Appendices	
	Appendix 1. GS1 and GS1-8 Prefixes	

List of Abbreviations

API	Application Programming Interface
DI	Dependency Injection
DRY	Don't repeat yourself design principle
EAN	European Article Number
EU	European Union
FIFO	First In First Out
GS1	Global Standards One
GTIN	Global Trade Item Number
HTTP	Hypertext Transfer Protocol
IC	Inspection Center
IGD	The Institute of Grocery Distribution
iOS	Apple's mobile operating system
ISBN	International Standard Book Number
ISO	International Organization for Standardization
JAN	Japanese Article Number
JSON	JavaScript Object Notation
KVO	Key Value Observation
LRU	Least Recently Used
MVC	Model-View-Controller design pattern
OOP	Object-Oriented Programming
POS	Point of Sale
RAM	Random Access Memory
REST	Representational State Transfer
SOC	Separation of concern design principle
SSL	Secure Sockets Layer
UPC	Universal Product Code
Xcode	Apple's Integrated Development Environment

1 Introduction

The proliferation of online grocery stores from the traditional brick-and-mortar retail stores has given rise to the demand for the availability of product information online. A recent study carried out by the Institute of Grocery Distribution (IGD) has shown that there is an upward trend of people preferring to do their weekly grocery shopping online [1]. The convenience of building the shopping cart in the comfort of one's own home, having the groceries collected and delivered straight to the front door, and avoiding the long queues at the checkout counters are the prime reasons that contribute to the rising popularity of online grocery stores, especially among families and elderly people.

There are several key elements that an online grocery store must provide to facilitate consumers in making an online purchase. Product images and data allow consumers to search for and identify items in the web store. Currently, the availability and completeness of the product information online vary vastly between data suppliers and manufacturers. Even when the information is available, there are often errors and inconsistencies when compared to the information printed on the product packaging. Consumers are accustomed to reading the product labels in the stores. In particular, they rely on the accuracy and completeness of the ingredients and allergens, which is critical in helping them to make a purchase decision. In order to protect the consumers from false information, there needs to exist a quality assurance system that can guarantee the integrity of the product information in digital form.

The goal of this project is to design and develop a data verification system that would ensure the integrity of the consumer trade item data available in the GS1 data bank. The scope of this project is limited to the development of the front-end system which would be an application running on the Apple iPad device.

2 Trade Item Information State of the Union

2.1 GS1 Organisation

Global Standards One (GS1) is a global organisation responsible for the development and maintenance of the supply chain standards. Compliance to the standards allows companies worldwide to access and exchange information using a common communication language. GS1 is also the official source for supplying manufacturers with unique barcodes, allowing for easy identification of trade items in the supply chain. [2,16.]

The GS1 system was first established in the United States when the first barcode standard – Universal Product Code (UPC) – was introduced. The UPC is a 12-digit identification number, represented in a series of bars capable of being read by a machine. [2,16.] The first product ever scanned, bearing the UPC barcode, was a 10-pack Wrigley's Juicy Fruit chewing gum. The event took place on June 26, 1974 [3,209]. Leveraging on the success of the UPC, the European Article Numbering (EAN) Association was established three years later and expanded the barcode to 13 digits to accommodate for a wider range of barcode numbers to be used outside of North America. Afterwards, the UPC and EAN barcodes were quickly adopted globally until the two organisations merged in 2005, and were launched as the GS1 organisation. Eventually in 2009, the UPC, EAN, and the Japanese counterpart, JAN, were renamed to Global Trade Item Number (GTIN), thus, greatly simplifying the standards for the industry. [3,210.]

Today, GS1 exists in over 100 countries and has added many new global standards for facilitating commercial trading and communication [3,210].

2.2 Trade Item Classification

All trade items are classified into one of the following categories in the product information hierarchy:

- **Base:** Retail consumer items typically found on the store shelves such as a bottle of cola drink or a bag of crisps.
- **Case:** Wholesale items such as a big case of cola drink or a box of crisp bags.

- **Pallet:** Transportation items such as a pallet of cola drink cases.

These classifications describe the level of packaging and also define the product types in the Foodie verification application.

2.3 Global Trade Item Number

The Global Trade Item Number (GTIN) is the GS1 System Identification Number used for uniquely identifying products and services. A GTIN number can vary in its length between 8 and 14 digits. Consumers most often encounter GTINs in the form of barcodes on product packaging. In fact, the standard was initially conceived in the United States in 1973 to facilitate automatic checkout of grocery items at the point of sale (POS) for improving the checkout efficiency. The POS system is able to quickly and accurately identify a product by scanning its barcode using a laser barcode scanner. Additionally, the POS system can also read supplementary information, such as price and weight, if they are encoded in the GTIN. Today, GTIN is globally adopted and is considered to be highly successful. The International Standard Book Number (ISBN) used for publications is another ubiquitous use of GTIN. [3,8-9;4,1-2.]

The GTIN family consists of four symbols that can be represented in a barcode [4,3]:

- **GTIN-12:** This is used in the classic 12-digit UPC barcode originally conceived by the Uniform Product Code Council in 1973.
- **GTIN-8:** This is a compact form of the barcode that is useful for product packaging with limited space.
- **GTIN-13:** This is most commonly used in EAN-13 barcodes found in the European markets. In Japan, it is referred to as *Japanese Article Number (JAN)*, which also contains 13 digits.
- **GTIN-14:** This symbol has an additional digit to represent the packaging level.

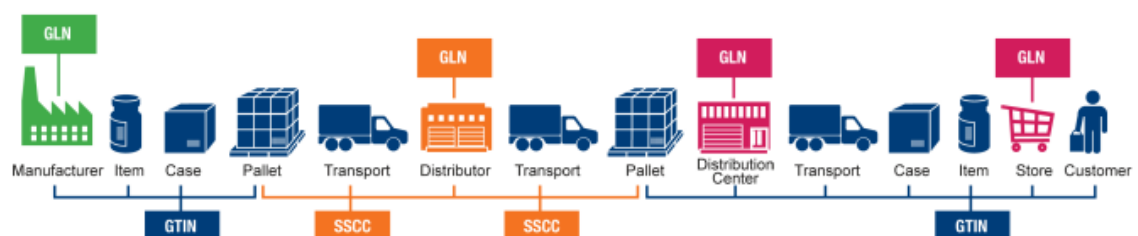


Figure 1 Trade Item Through the Supply Chain. Reprinted from Palazzolo (2013) [5]

Every company that adopts the GTIN standard is assigned a *company prefix*, which is given by a GS1 member organisation. A complete reference of the GS1 prefixes can be seen in appendix 1 [2,19]. The prefix is reflected in the first few digits of the GTIN. Each manufacturer is also allocated a range of numbers, which they can assign to their products. Figure 1 shows that, while the products get distributed through the supply chain to the end user, companies are able to trace and identify them using their GTINs, as they are guaranteed to be unique worldwide. Additional benefit for a company to be GTIN compliant is that it provides them with a common interface to process, store, and communicate about their products with trading partners. [4,1.]

2.4 European Union Regulation

A significant driving force for manufacturers and retailers to provide product information online came with a recent regulation published by the European Union (EU). The EU 1169/2011 Regulation – effective since December 2014 – mandates that all food and beverages sold must be accompanied with a significant amount of food information. The regulation is applicable to sales in brick-and-mortar retail stores as well as online stores across the EU markets. [6.] The compulsory information that must be displayed for the food and beverage products include [7]:

- The name of the product
- The list of ingredients
- Any ingredient or processing aid causing allergies or intolerances used in the manufacture or preparation of a food and still present in the finished product, even if in an altered form
- The quantity of certain ingredients or categories of ingredients
- The net quantity of the food
- The date of minimum durability or the ‘use by’ date
- Any special storage conditions and/or conditions of use
- The name and address of the food business operator under whose name the food is marketed (or the importer’s name if the food business operator is outside the EU)
- The country of origin or place of provenance where provided for in Article 26
- Instructions for use where it would be difficult to make appropriate use of the food in the absence of such instructions

- The actual alcoholic strength by volume for beverages containing more than 1.2% by volume of alcohol
- A nutrition declaration.

The objective of the regulation is to provide a standard for food labelling and to allow consumers to make an informed purchase decision. In particular, information such as the nutrition list, ingredients, and allergens are critical for the consumers. Failing to provide the food information to the consumers prior to the sale prohibits the store from selling the affected products. [6.]

3 Design and Implementation

A typical medium to a large-scale front-end application often consists of many functional blocks with several thousand lines of code. Moreover, new features and change requests can cause the code base to grow rapidly. Without proper planning, this can easily lead to a complex and unstructured code base. Therefore, employing robust design paradigms from the outset is important in order to build a strong foundation. This chapter discusses how the Foodie verification application was implemented using established best practices and how common pitfalls of application design were avoided. The aim was such that new developers, who are well versed with the established design paradigms and principles, would be able to contribute to the project with minimal effort and have a shallower learning curve.

3.1 Data Model

The primary structure of the inspection data model comprises nine classes pertaining to the management and storage of the inspection data.

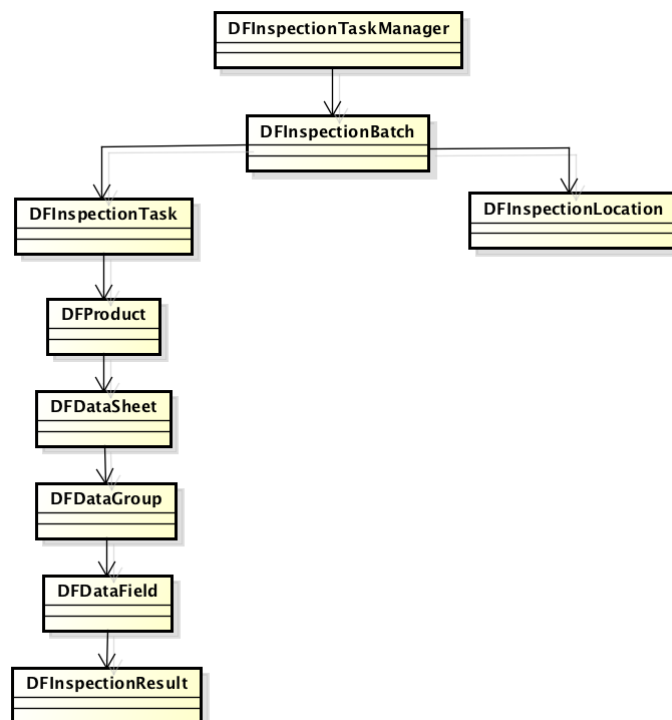


Figure 2 Class Diagram of the Main Blocks

Figure 2 illustrates a high-level view of the relationship between the main classes.

3.1.1 Inspection Task Manager

The inspection task manager was designed to be the main coordinator object responsible for maintaining the different types of inspection data. It is a singleton object, which signifies that there is only one instance of it in the application lifecycle. It does not hold any internal states of its own but carries the following functions:

- Holds references to the inspection data in the memory
- Persists the inspection data into the disk memory
- Decodes the inspection data from the disk memory after a cold start
- Fetches inspection locations from the server periodically.

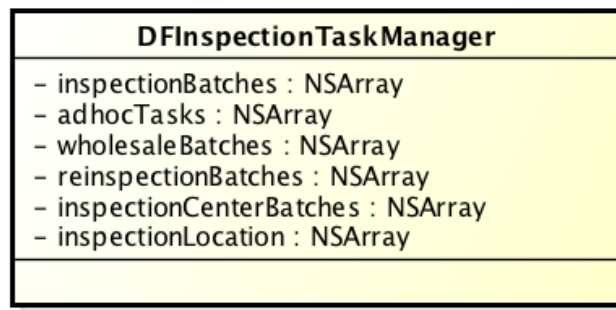


Figure 3 Inspection Task Manager Class Diagram

Figure 3 illustrates the interface of **DFInspectionTaskManager**, which exposes five array collections representing each of the supported inspection type in the application and an additional array collection for the inspection locations. Each collection is immutable, which means that consumers of this class will have read-only access. This is by design as to control and restrict all modifications to the collections to occur within the class implementation.

3.1.2 Inspection Batch

All inspection types, save for one, are allocated in batches. When an inspection work is requested, the server will return an inspection batch consisting of a number of inspection tasks corresponding to a predefined quantity or time availability set by the inspector.

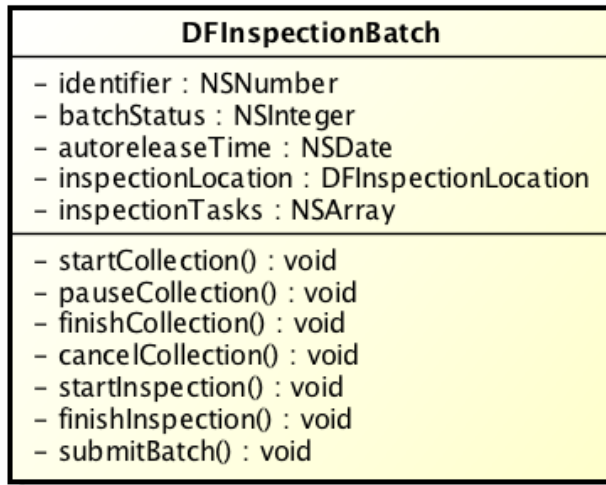


Figure 4 Inspection Batch Class Diagram

As shown in figure 4, an inspection batch is associated with a specific inspection location and is only valid for a certain period of time. The validity period is by default set at four days, though it may vary depending on the needs in the future. If a batch expires past its validity period, the server will automatically release the batch, freeing its inspection tasks for another inspector. This mechanism has been put in place to guarantee that all products get inspected and do not remain in the in-progress state for an indefinite amount of time.

A batch always holds a particular state, communicated via the `batchStatus` property. Table 1 lists the possible state enumerations for an inspection batch.

Table 1 Inspection Batch States

Status	Description
0	Batch has been assigned.
1	Batch has started its collection.
2	Batch has completed its collection.
3	Batch has started its inspection.
4	Batch has completed its inspection.
5	Batch is ready for submission.
6	Batch has been submitted.
7	Batch was cancelled.

The `batchStatus` is exposed as a read-only property. It reflects the internal state that is indirectly managed through the public instance methods listed in figure 4. For example, sending the `startInspection:` message to an instance of `DFInspectionBatch` will implicitly modify the internal state to 3. Conversely, the internal state also acts as a safeguard against invalid message invocation. As an example, sending the message `submitBatch:` when the internal state is other than 5 will return an error.

3.1.3 Inspection Task

An inspection task represents a single unit of inspection. It holds information about the product, store-specific aisle location, and inspection photos. Inspection tasks are commonly grouped into an inspection batch. However, its lifecycle is not dependent on one. An inspection task may also be allocated singularly, such as in the ad-hoc inspection mode.

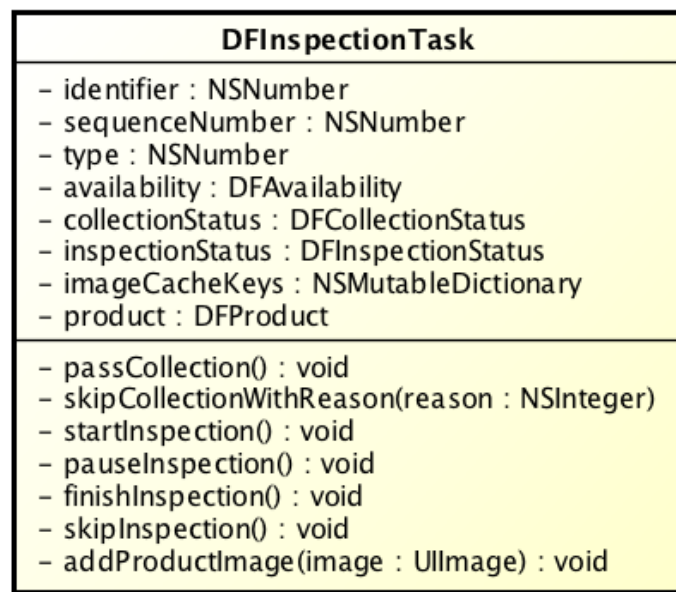


Figure 5 Inspection Task Class Diagram

As shown in figure 5, an instance of `DFInspectionTask` holds a `collectionStatus` state and an `inspectionStatus` state. Both states are managed inside the class implementation and are only exposed as read-only properties in the class interface. Table 2 lists the possible state enumerations for the `collectionStatus`.

Table 2 Collection States of Inspection Task

Status	Description
0	Task is not allocated to any inspector.
1	Task's product is pending for collection.
2	Task's product has been collected for inspection.
3	Task's product collection was skipped.

In the event that a product collection was skipped, the message `skipCollectionWithReason:` is sent to the object instance, along with an integer argument specifying the reason. The allowed possible values for the reason are *0 - Unable to Collect*, *1 - Out of Stock*, and *2 - No Longer Available*. If the product was successfully collected, the message `passCollection` is sent to the task, which will initialise its `inspectionStatus` state. The complete inspection state enumeration is listed in table 3 below.

Table 3 Inspection States of Inspection Task

Status	Description
1	Task is ready for inspection.
2	Task passed the inspection.
3	Task failed the inspection.
4	Inspection was skipped.

The inspection task will receive the message `finishInspection` when all of its attributes have been acknowledged. The final state will then be resolved depending on whether there were errors. If all of the attributes were passed, then the `inspectionStatus` will be set to 2. In case there were one or more errors, status 3 will be set. A timestamp record for every event performed on `DFInspectionTask` objects will be saved and will get submitted to the server for analytics purposes.

3.1.4 Product

Figure 6 shows the class diagram for `DFProduct` and the relationship to its constituent parts. A product represents a trade item, which can be a consumer, wholesale, or pallet type. It has a one-to-one relationship with an inspection task and is identifiable by its

GTIN number. Each instance of `DFProduct` has a strong reference to the latest imported datasheet, supplied by the manufacturer or the data supplier. A datasheet defines the set of inspectable attributes pertaining to a product and their rules as set forth by the GS1 requirements.

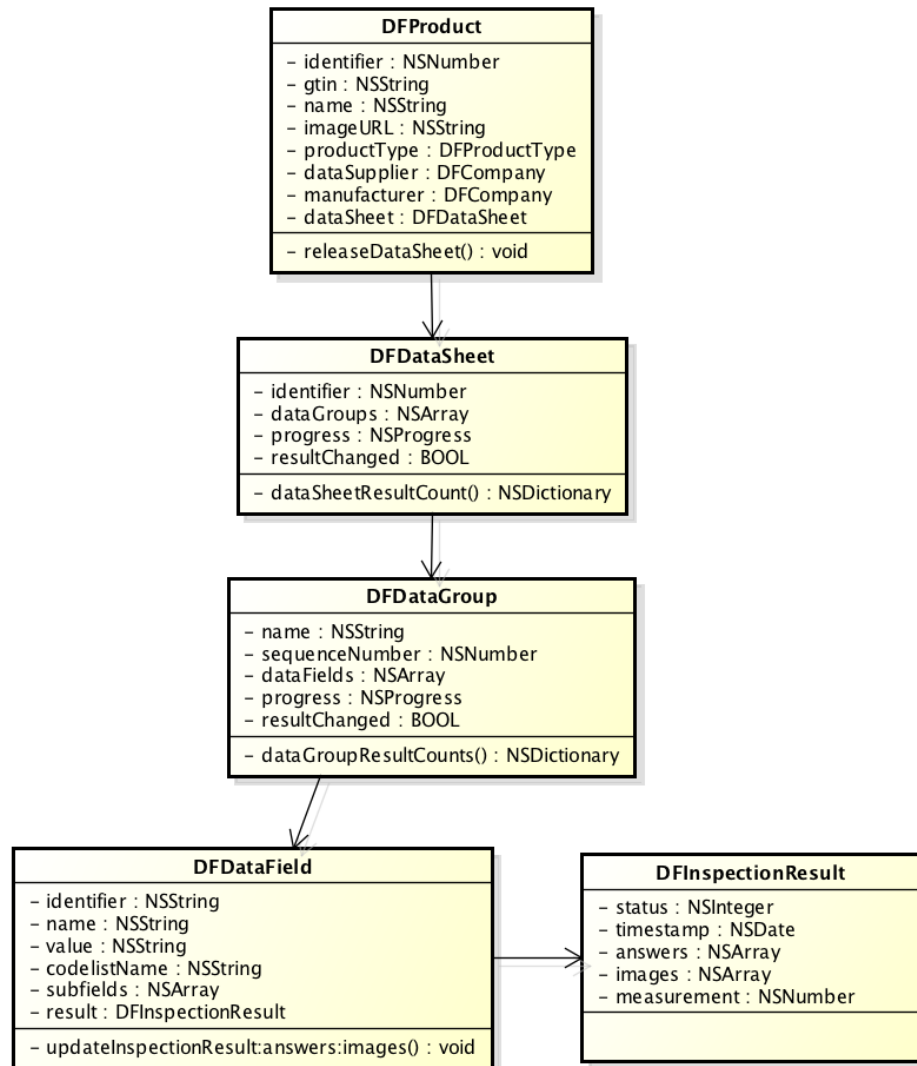


Figure 6 Class Diagram of `DFProduct` and its Compositions

A data field represents an attribute of the inspection data at the most granular level. Each attribute has an *identifier*, *name*, *value*, *rule*, and *result*. A data field can also form a hierarchical structure containing an arbitrary level of subfields. The cumulative result is consolidated by recursively iterating through the subfields and is stored in the parent field. Related data fields are categorised into groups where each group is assigned a sequence number that determines the suggested order of inspection. The `DFDataSheet` provides a convenience method, `datasheetResultCount`, that

returns the counts of *passed*, *failed*, *warning*, and *unable to validate* attributes. It aggregates the results by enumerating its array of data groups and sending the message `dataGroupResultCount` to each instance of `DFDataGroup`. This method call is particularly useful for reporting progress to different components used throughout the application.

3.2 System Architecture

3.2.1 Model View Controller

The most rudimentary prerequisite for any programmer wishing to develop on the Apple iOS platform is familiarity with the Model-View-Controller pattern. The iOS platform enforces the use of the MVC design pattern for custom objects designed for their applications [8,1].

The MVC design pattern was initially developed to facilitate the inclusion of a graphical user interface for front-end applications made using the Smalltalk programming language. Simultaneously, the pattern was also intended to promote developers to create reusable software components. [9,202.] This reusability is encouraged by logically separating the application logic from the user interface and interactivity components [10,121]. Today, the MVC pattern is widely used in practically all application domains such as on the web and in mobile. Several new and modified variants of it have also emerged to accommodate specific platform requirements.

MVC is a high-level design pattern that is more broadly known as the compound design pattern. The semantics of the compound pattern is such that each of its components adheres to a more basic design pattern. Together, the basic patterns interact with each other and work in synergy to form a high-level design solution. [11,500.] The structure of the MVC design pattern consists of three primary components: *view*, *model* and *controller* objects. Each object fulfils a role and carries a specific responsibility in an application. Figure 7 below illustrates the relationships between the three objects.

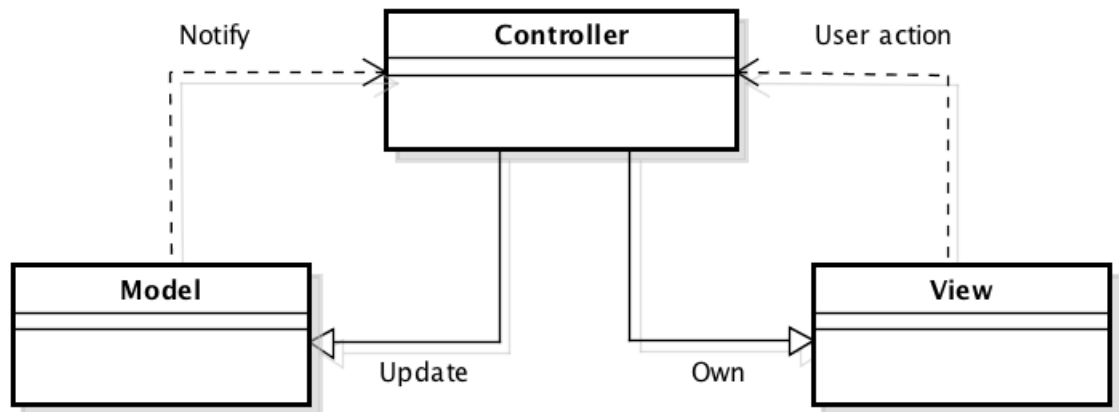


Figure 7 Relationship between Model-View-Controller objects. Reprinted from Apple (2012) [12]

As shown in figure 7, a view object in an application represents the object that is visible to the user. It conforms to the composite design pattern, where a view forms a hierarchy that may contain other views or interactive UI elements. Its role is simply to know how to draw and present itself on the screen. A view object is also responsible for maintaining different states that it might have and presenting the model data to the user. [10,121.]

The model adopts the observer pattern, which notifies its registered components any time its state changes. A registered component can be a controller, a view, or another model. A model object encapsulates the application logic and does not concern itself about how its data should be displayed [10,121]. For example, in a calculator application, the model might contain the algorithms for performing multiplication, addition, subtraction and division. However, it does not need to know how the calculation result will eventually be presented to the user.

The final piece of the puzzle is the controller object, which acts as the mediator between the view and the model objects. Its responsibility is to ensure consistency between the data from the model and the data being presented by the view [13,30]. Conversely, the view also delegates the handling of its user actions to the controller, which acts as the strategy plugged to the view. In an MVC-designed application, the controller is typically the least reusable component as it contains a tightly coupled code dependent on the view and the model objects [12].

In the compound design pattern where the controller acts as the mediator of data between the model and the view objects, it is not unexpected that the controller can

contain a significant amount of glue code. In a large-scale application, this will lead to an undesirable effect that the controller becomes very large and unwieldy. The controller, which is the least reusable component, holds too many responsibilities and is tightly coupled to logic that may be useful to other components. For example, a commonly used view in iOS applications, `UITableView`, often uses the controller object as its data source and delegate. At a minimum, there are three methods that a controller conforming to the `UITableViewDataSource` and `UITableViewDelegate` protocols has to implement in order to present data in a tabular format. That boilerplate code could be located in a dedicated object that could be reused by other controllers. Similarly, the code for fetching data from the web service and parsing the received data is also commonly placed inside the controller. To solve the problem, the Foodie verification application separates out the web service logic into a service layer.

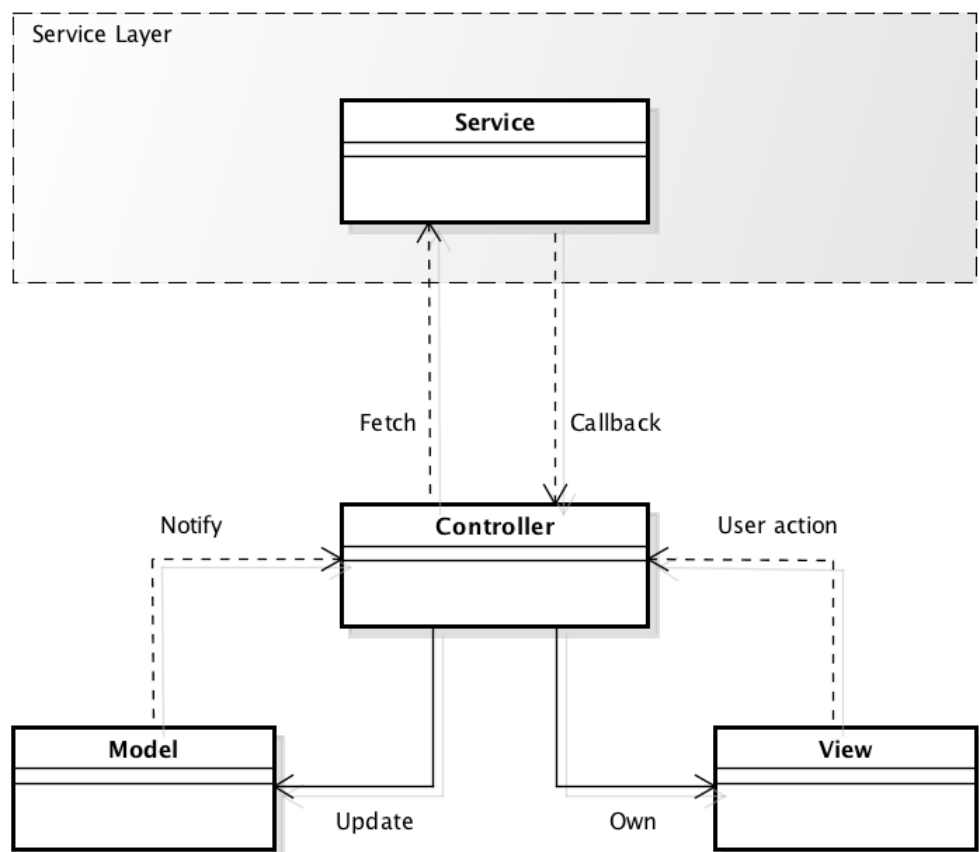


Figure 8 Model View Controller with Service Layer

As illustrated in figure 8, the service object interfaces with the controller and returns the fetched data using the block callback mechanism. Its main responsibility is to offload and encapsulate the web service logic and data parsing previously residing in the

controller. The net result is that the controller's complexity is reduced and the service object exists as a standalone component, which can be extended and shared with other components.

3.2.2 Singleton

Singleton is a pattern for ensuring that only one instance of a class is instantiated in the application lifecycle [11,177]. It is a useful pattern because it provides a convenient mechanism for sharing data between different code modules. Usage of the pattern is ubiquitous throughout Apple Cocoa frameworks. For example, sending the message `sharedApplication` to `UIApplication` from anywhere in the program will return an instance of the currently active application. Similarly, `NSUserDefaults`, `UIScreen`, and `NSFileManager` classes provide shared instances with methods to obtain the common preferences, screen objects, and file manager, respectively. This ubiquity makes it an important pattern for Cocoa developers to learn and understand.

Despite its prevalent adoption, the pattern is criticised to be akin to a global variable, which is generally considered to be a bad coding practice. Global variables can easily lead to errors that hard to debug due to their mutable and stateful nature. In addition, OOP advocates the concept of encapsulation, which is about limiting the scope of mutable states whereas the central idea of the singleton pattern directly negates that goal.

An alternative method to singleton is to manually inject data into the modules, more formally known as Dependency Injection (DI). However, the drawback of DI is that it requires all controllers' interfaces to be modified to accept the dependent objects, sacrificing the simplicity of sharing common resources. For this reason, the Foodie verification application utilises the singleton pattern for its service components as well as the manager components. The advantage of services and managers to be singletons is that their consumers can rely on a centralised location where specific data can reside. In the case of `DFInspectionTaskManager`, consumers of the singleton object can safely retrieve inspection batch resources, knowing that duplicate copies do not exist elsewhere. Having multiple copies of the same data would be detrimental, as the information can appear out of sync when displayed in different views. To overcome the drawbacks discussed earlier, all singleton objects in the Foodie verification

application only expose immutable shared resources or they do not contain any states at all.

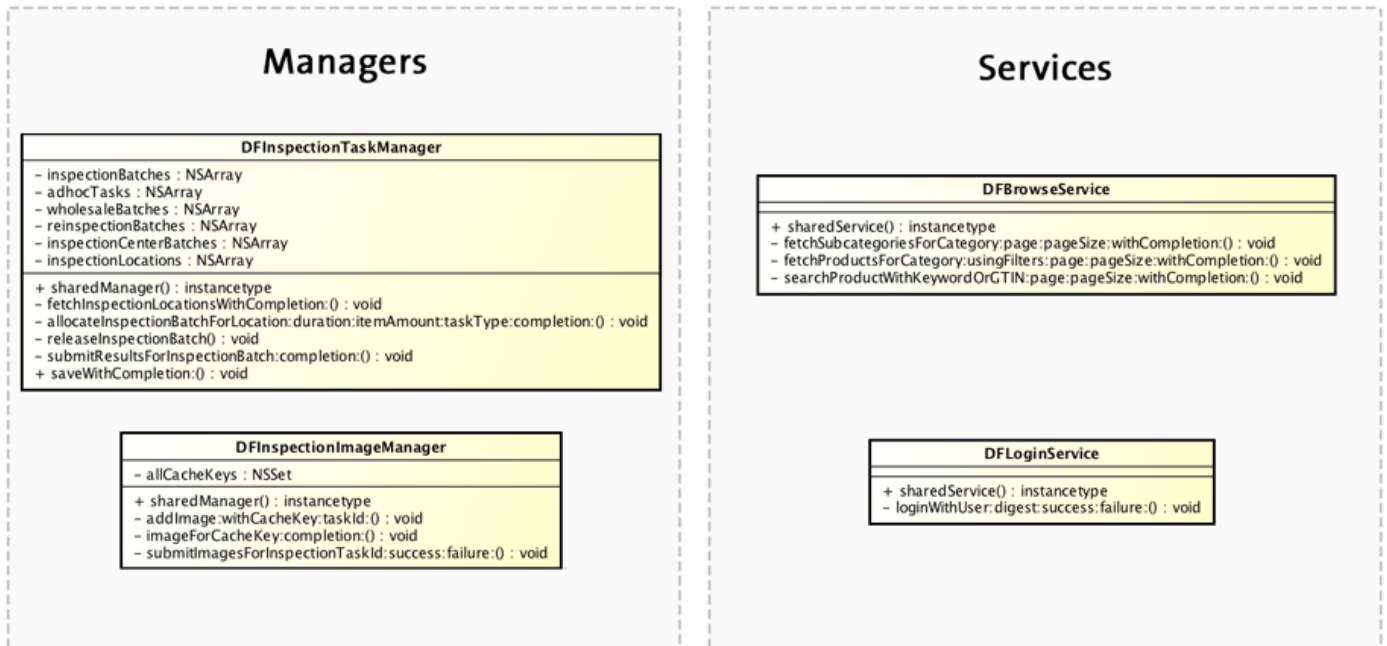


Figure 9 Manager and Service Singletons

As figure 9 shows, `DFInspectionTaskManager` exposes six immutable shared arrays and `DFInspectionImageManager` has one immutable `allCacheKeys` set accessible to the public. The `DFBrowseService` and `DFLoginService` singleton services do not hold any states of their own and only have methods for retrieving data from the network. It is worth noting that all manager and service classes in the application follow the platform standard naming convention to denote that they conform to the singleton pattern. Thus, any Cocoa developer working on these classes should be aware of the implications of this pattern.

3.2.3 Strategy

The strategy pattern is defined as a family of encapsulated sets of algorithms, which allows its clients to use a set of algorithm interchangeably [11,24]. This pattern is utilised in the Foodie verification application to accommodate data sources that need to download their data in small chunks from the API.

```

@interface DFTableViewController()
@property (nonatomic, strong) NSMutableArray *items;
@end

@implementation DFTableViewController

- (NSInteger)collectionView:(UICollectionView *)collectionView
numberOfItemsInSection:(NSInteger)section
{
    return [self reachedLastPage] ? [self.items count] :
[self.items count] + 1;
}

- (UITableViewCell *)tableView:(UICollectionView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    UITableViewCell *cell = ...

    if (indexPath.row == [self.items count]) {
        // Count the page and download more items
        int page = floorf(indexPath.row / kPageSize);
        [self downloadItemsForPage:page];
    } else {
        // Show downloaded item
    }

    return cell;
}

@end

```

Listing 1 Pagination Handling In Controller

Listing 1 demonstrates the simplest way to add pagination support in a controller object that displays its data in a tabular form. This implementation is inelegant because each controller has to handle the page calculation and is tightly coupled with the module that performs the data fetch. In case the controller needs to display a different set of data from another API source, the existing controller code will need to be altered, risking existing functionalities to break. An alternative approach that employs the strategy pattern is illustrated in figure 10.

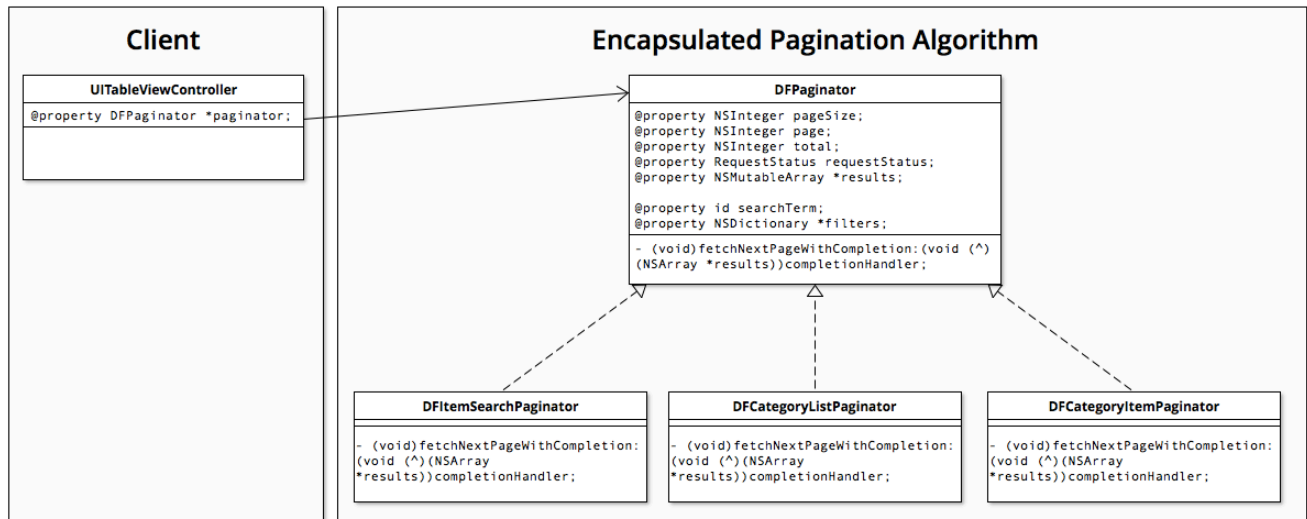


Figure 10 Encapsulated Pagination Behaviors

As figure 10 shows, the pagination logic has been extracted out of the controller and encapsulated into another class structure. The controller delegates its pagination functionality to the `DFPaginator` superclass, which defines the pagination interface. The `DFPaginator` object acts as a data source, where its subclasses will be responsible for fetching a collection of objects from the API and returning them to the requesting controller. Here, each `DFPaginator` subclass might use different services to support different sets of data from other APIs. It is worth noting that the controller is unaware of the concrete pagination implementation. In case a new paginator needs to be added, the existing code for the base `DFPaginator` or the existing concrete paginators do not need to be modified. A new paginator can simply be added and given to the controller. Through the use of polymorphism, the controller has the flexibility to swap out its pagination logic dynamically at runtime.

3.2.4 Observer

The observer pattern defines a mechanism for an object to loosely communicate its state changes to interested parties. The formal name given to the interested party is *observer* and the object that initiates the communication is called the *subject*. [11,51.] The observer pattern is a powerful abstraction technique, as the subject does not need to know the details of its observers. The Apple Cocoa framework provides four approaches on implementing the observer pattern [14]. The Foodie verification application utilises the Key-Value Observation (KVO) approach that allows object

instances to listen for changes on a particular key path. The use of KVO is primarily for displaying the inspection progress at two levels – data group and datasheet.

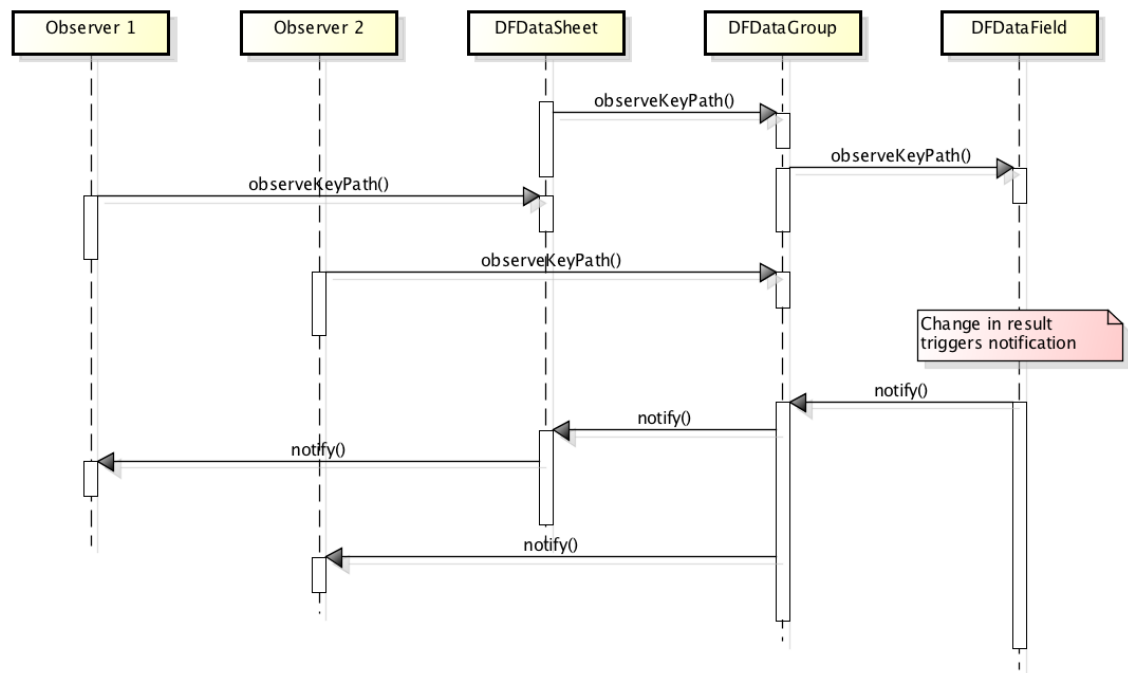


Figure 11 Sequence Diagram of Observer Pattern

Figure 11 illustrates the sequence diagram for the Foodie verification application's implementation of KVO. All views that would like to show the overall progress of a product inspection fall under the category of *Observer 1*. Conversely, the `DFDataGroupTableViewCell` is an example of *Observer 2* where the view needs to display the progress of a single data group. Any time an inspection result is updated, the corresponding `DFDataField` instance will modify its `resultChanged` property, triggering a chain of notifications to `DFDataGroup`, `DFDataSheet`, and their corresponding *Observer 1* and *Observer 2* subscribers.

3.3 Key Design Strategies

3.3.1 Image Loading

The Foodie verification application displays high-resolution product images in varying dimensions throughout the application. Having good quality images helps the inspector identify the correct product during the collection phase.

In order to facilitate the different presentations, the images ideally need to be optimised for the views where they are going to be displayed. Figure 12 demonstrates the functional relationship between the modules on a block level.

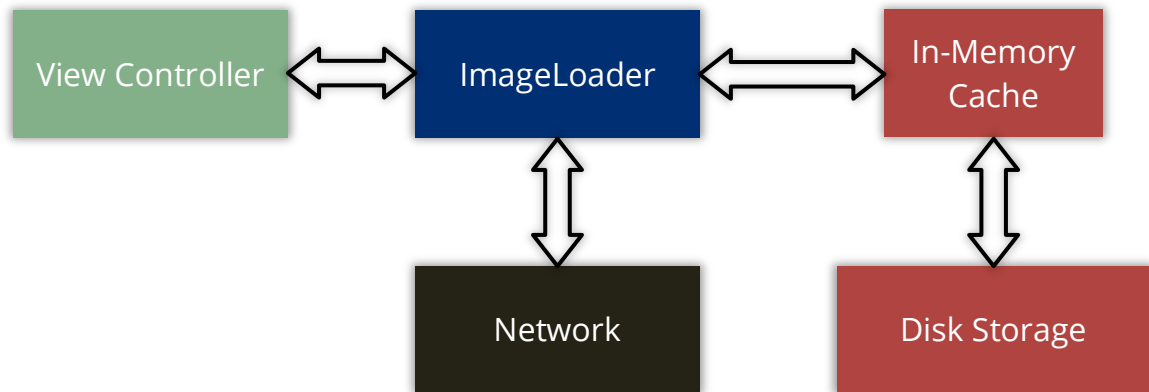


Figure 12 Functional Block Diagram of Image Loader

When a view wants to display a product image, it will first make a request from the image loader module. The image loader will then check if the image with the specified resolution is already available in the caches. In the event that it is not, the module will forward the request to the server. Finally, when the image loader receives a response from the server containing the image data, it will cache the information and provide the image to the original requestor.

This architecture was designed with the goal of high-performance for displaying images in varying resolutions. From the user's perspective, a fluid UI is reflected by the perceived responsiveness. This means that the application cannot lag and must remain responsive to touch inputs. Technically, this is achieved by performing the expensive and high latency processing operations asynchronously on a background thread.

A requested image can reside in several different locations. These locations can be thought as layers, as visualised in figure 13. The layer hierarchy helps to design a strategy for fetching the images. In the best-case scenario, the requested image lives in the heap memory, which would yield the smallest latency for the fetch. The next scenario is if the image is already downloaded and saved on the disk storage. Finally, as the slowest method, the image will need to be fetched and downloaded from the network.

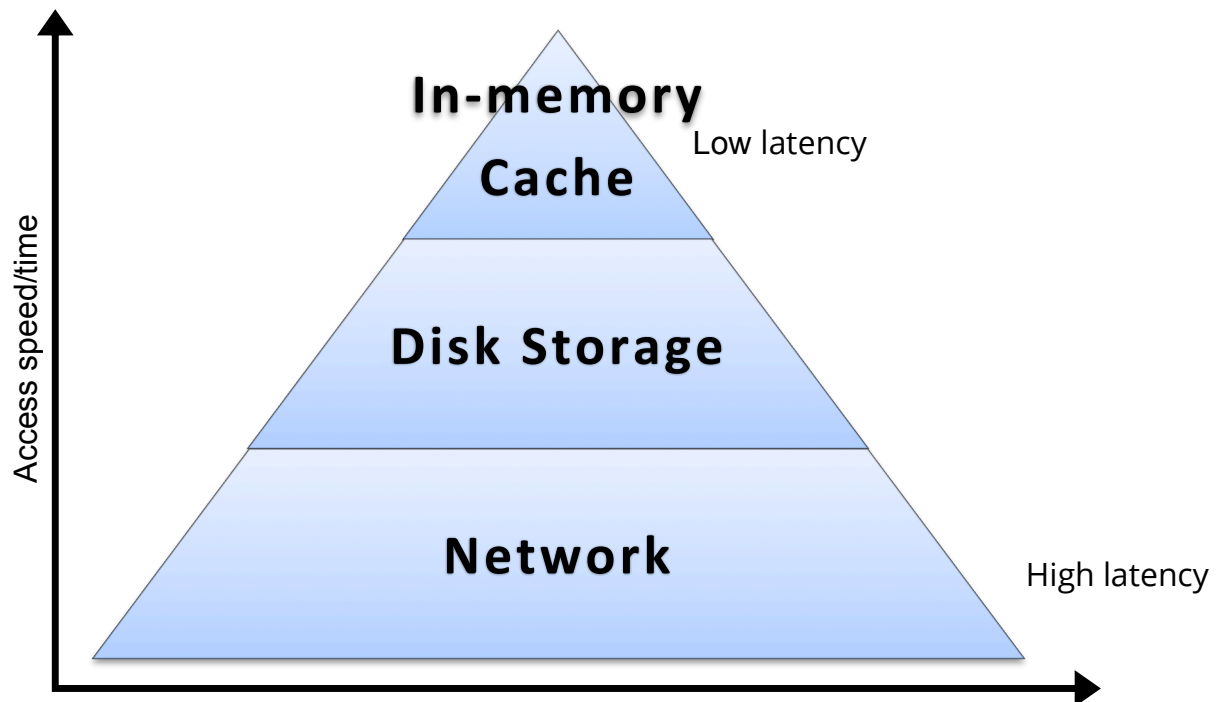


Figure 13 Memory hierarchy

In the event that the image needs to be fetched from the network, there is a cache replacement strategy that takes place when the image is saved. This replacement is necessary because of the limited amount of memory available for the application. The strategy employed by the image loader module is called Least Recently Used (LRU). LRU functions in the way that it does an internal bookkeeping of the images by keeping track of which images were used recently. If the allocated cache is full and a new image is about to be saved, LRU will replace the least recently used image with the new one.

3.3.2 Persistence

Despite being fundamentally a client–server application, the application needs to be able to function without network connectivity. In order to achieve this requirement, key design decisions had to be made early in the development. For example, all inspection data and images must be persisted on the disk cache. When the inspectors are working in the field, there is no guarantee that there will be network access. Therefore, the inspection results are designed to be stored on the device and can be submitted to the server when the connectivity is restored. Additionally, to prevent the application

from data loss, the inspection results are automatically saved when any of the following events occur:

- The inspection is paused
- The inspection is completed
- A product image is asked to be captured
- A new version of the application is available
- The application is sent to the background
- A task is allocated
- A task is collected
- A task is submitted
- The user is logging out.

`DFInspectionTaskManager` exposes a single interface for saving the object graph – `saveWithCompletion:(void (^)(void))completionHandler`. This method is safe to be called concurrently from multiple threads, which allows multiple events listed above to be triggered simultaneously.

4 Foodie Verification Application

Prior to the Foodie verification application, GS1 Finland relied on an electronic datasheet document that was filled by the manufacturers. This did not allow for an autonomous centralised information system as the large number of files sent back-and-forth could easily become unmanageable. Additionally, the document could not accommodate displaying the data in a dynamic manner and lacked advanced input validation.

The Foodie verification system consists of a server solution, which acts as a product information repository and a client application designed to run on Apple iPad devices. The iPad was chosen as the inspection device because of its ease of use and reliability.

4.1 Inspector Use Cases

The main users of the application are called inspectors. Figure 14 illustrates the main activities that can be performed by the inspector.

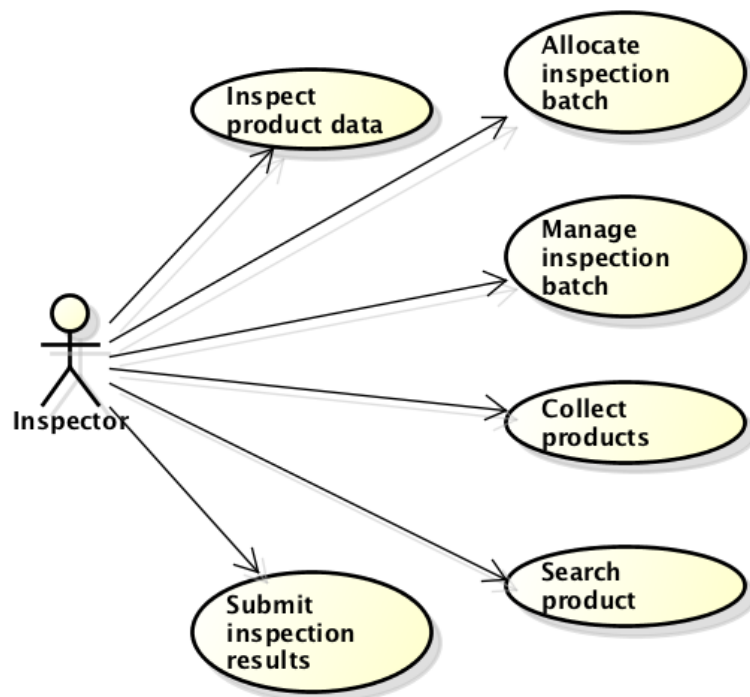


Figure 14 Use Case for Product Inspector

A typical scenario for an inspector is to start the workday by allocating an inspection batch consisting of products that have not been inspected. The server determines the number of products to allocate based on the available work hours defined by the inspector. The inspector then proceeds to collect the designated products in the warehouse or in the store with the help of the supplied product information and aisle location information. In order to eliminate human errors, the inspector utilises a barcode scanner to match with the correct product during collection. The application will respond with audible and haptic feedback to signal if the correct product was scanned.

Once the products are collected, inspection work can begin. Depending on the type, each product has a datasheet containing up to 100 attributes that must be verified by the inspector. Example attributes include the product name in relevant languages, physical dimensions, ingredients listing, and product barcode. Each attribute will be marked as passed, failed, warning, or cannot be validated. The product inspection is considered done once all attributes have results. To conclude the product inspection, the inspector is required to take a picture of the product using the built-in iPad camera. The picture is then submitted together with the results to the server for further processing.

4.2 Functional Division of Foodie Verification

The functions of the client application can be categorised into seven modules, as illustrated in figure 15. This chapter presents an overview of each module along with screen captures of their final implementations.

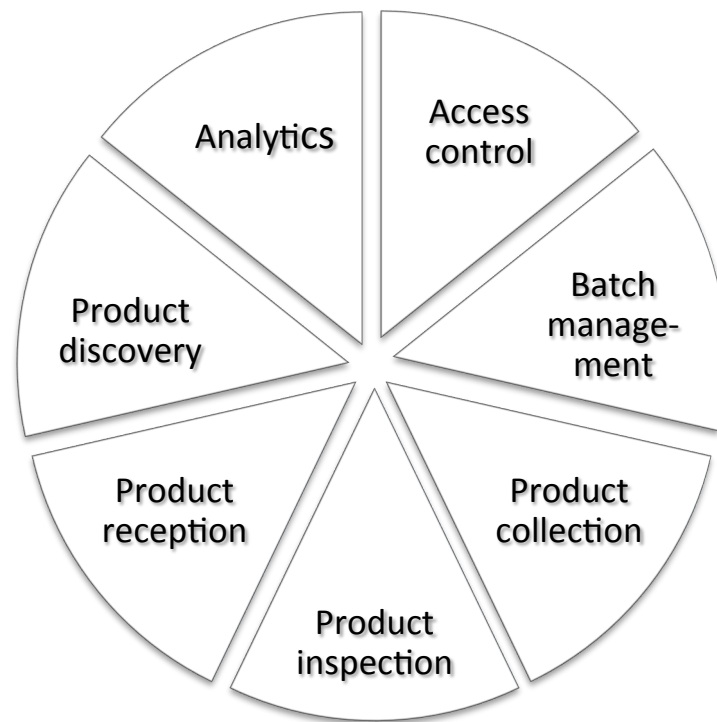


Figure 15 Functional Division of the Foodie Verification Application

The seven modules, seen in figure 15, define the functional blocks of the application, which have been designed to be independent of each other and can serve as standalone reusable modules.

4.2.1 Access Control

Access to the client application is granted by supplying a valid authentication credential. The usernames and passwords are securely stored on the device keychain store. The authentication server verifies the credential and grants the user a session key that is valid for a fixed period. The application handles session expiration by automatically signing the current user out and requiring the user to sign in again to gain access to the application.

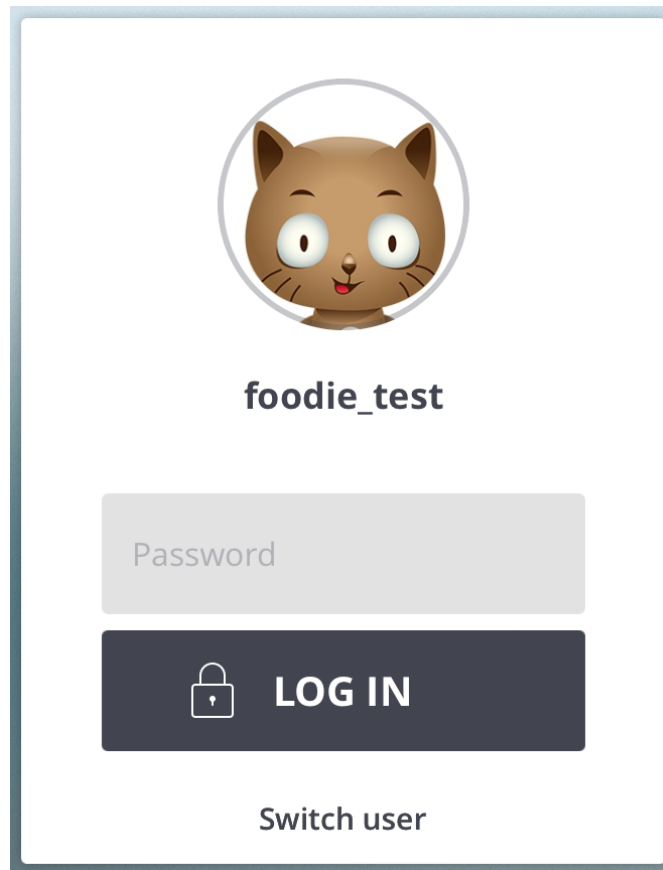


Figure 16 Access Control to the Application

Figure 16 illustrates the login screen presented to the inspector when a valid session is not present, such as when the application is started up for the first time. The application supports adding multiple accounts to allow for device sharing between inspectors. Upon successful login, only the inspection data relevant to the authenticated user will be loaded and visible.

4.2.2 Batch Management

The batch management module provides an overall view of the progress of past and present inspection batches allocated to an inspector. The batches are consolidated into three sections: *In Progress*, *Completed*, and *Cancelled*. In this view, seen in figure 17, most of the attention will be on the batches that are in progress. Inspectors need to remember to submit the inspection results before the batch expires. Submitted batches are saved as to allow for future reference. Cancelled batches do not hold any significant information other than to keep track of the cancellation time.

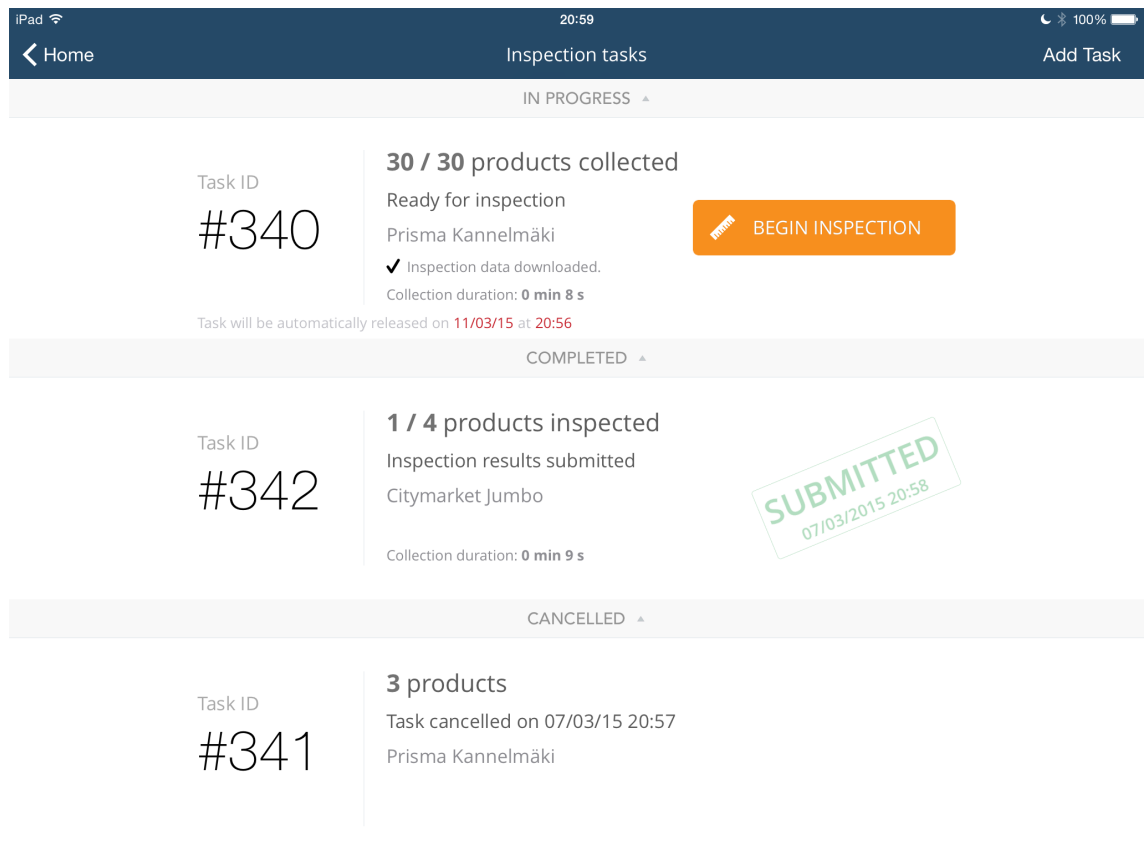


Figure 17 Batch Management View

The batch management view also provides the entry point for requesting a new inspection batch. Figure 18 shows the allocation view that appears when the “Add Task” button is pressed. An inspection batch is always requested for a specific location. This enables the server to narrow down the selection of products available for inspection. The location can be a brick-and-mortar store, a warehouse, or an inspection centre. The inspector is also required to either specify an explicit quantity or to provide an estimate of the time available for the inspection. Based on the value given, the server will determine the number of products to allocate. The batch of products retrieved from the server is then designated to the inspector. The same products will not be allocated to another inspector until their inspections have been completed or released.



Figure 18 Task Allocation View

Usage of the task allocation view, seen in figure 18, is consistent throughout the application such as when allocating inspection tasks for existing, wholesale, and new products.

4.2.3 Product Collection

Once an inspection batch has been allocated, the inspector then proceeds to collect the assigned products in the warehouse or in the store. The detail view, seen in figure 19, is composed of critical information, such as the product image, Global Trade Item Number (GTIN), package size, and the aisle location, to assist in identifying the correct product during collection.

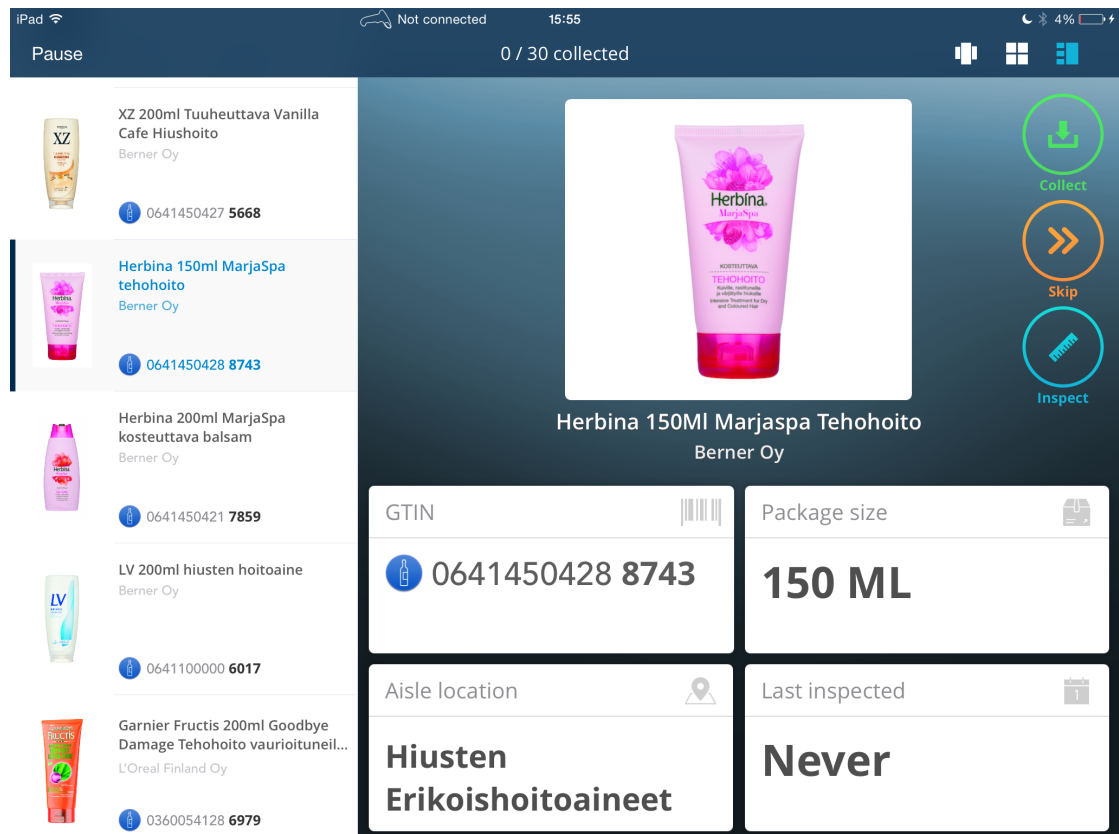


Figure 19 Product Collection View

The inspector is assured that the correct product was collected if the GTIN information matches the string of numbers printed under the product barcode. Alternatively, the inspector can utilise a barcode reader to scan the barcode on the product packaging. If the scanned barcode is found, the corresponding product will be automatically marked as collected in the collection view. Usage of the scanner is preferred as it eliminates any human error when comparing the numbers.

In the event that the product could not be located, it is marked as skipped by selecting an option specifying the reason for the bypass. This scenario typically occurs when the product is out of stock or has been removed from the inventory. A bypassed product will trigger the server to release its inspection and possibly re-schedule for inspection at a later time.

4.2.4 Product Inspection

The verification application allows for five different modes of product inspection work:

- **Wholesale inspection:** Case-level product inspection carried out in batches at warehouses.
- **New product inspection:** Base- or case-level product inspection carried out at inspection centres.
- **Existing product inspection:** Base- or case-level product inspection carried out in batches at warehouses or brick-and-mortar stores.
- **Product reinspection:** Base- or case-level product inspection carried out in batches at brick-and-mortar stores or inspection centres.
- **Ad-hoc inspection:** Base- or case-level product inspection carried out at warehouses or brick-and-mortar stores.

Figure 20 shows the entry points for the inspection modes on the front-page view of the application.

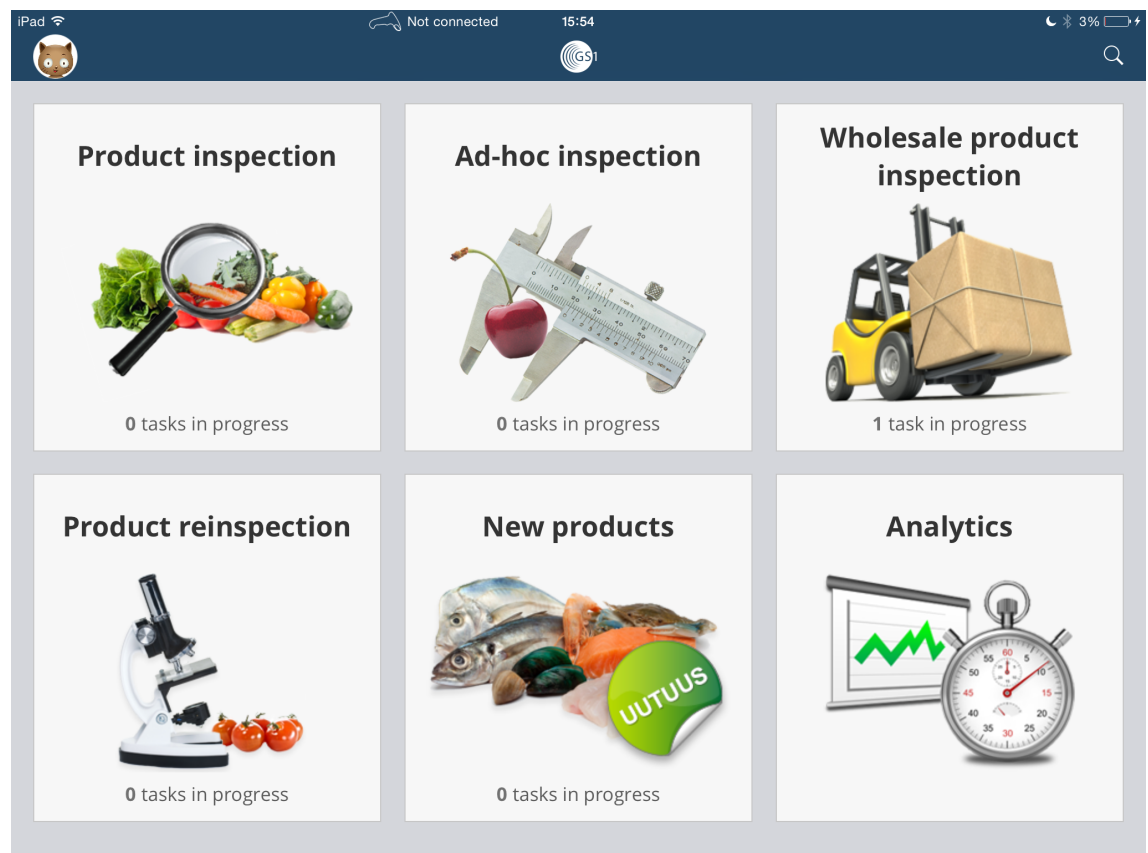


Figure 20 Front Page View

The Foodie verification application is a quality assurance tool used for validating the accuracy of consumer goods data available in the GS1 data bank. Each product is associated with a set of attributes, which are displayed in the inspection view shown in figure 21. Each attribute will in turn be validated by comparing it with the physical product in accordance with its rules. A rule serves to instruct and guide the inspector on how to verify the correctness of an attribute's data.

The screenshot displays the 'Product Inspection View' on an iPad. The top status bar shows 'Not connected' and '15:56'. The app header includes a 'Pause' button, the product ID '#2659', and camera icons. The main content area is divided into two columns. The left column shows a product image, a progress indicator at 4%, and a list of attributes with their completion status: 'Tunnistetiedot' (3/14 DONE), 'Tuoteluokittelu' (0/4 DONE), 'Mitta- ja painotiedot' (0/11 DONE), 'Ainesosat ja valmistustiedot' (0/7 DONE), 'Ravintosisältö' (0/6 DONE), 'Säilyvyystiedot' (0/6 DONE), 'Alkuperä- ja pakkaustiedot' (0/9 DONE), 'Turvallisuuustiedot' (0/14 DONE), and 'Alkoholituotteen tiedot' (0/3 DONE). The right column displays the product name in multiple languages (EN, FI, SV) with corresponding status icons (checkmark, warning, or error). Below this, the 'INSPECTION RULES' section provides detailed instructions for verifying the product name and packaging. The bottom section lists the product name in 35MRK for EN, FI, and SV.

GTIN-KOODI
03600522191186

TÄYDELLINEN TUOTENIMI / ENGLANTI (EN)
L'Oréal Paris Elvital 200ml Color-Vive Instant Miracle treatment gel for colored hair

TÄYDELLINEN TUOTENIMI / SUOMI (FI)
L'Oréal Paris Elvital 200ml Color-Vive Instant Miracle tehohoitogeeli värjäytyille hiuksille

TÄYDELLINEN TUOTENIMI / RUOTSI (SV)
L'Oréal Paris Elvital 200ml Color-Vive Instant Miracle inpackningsgel för färgat hår

INSPECTION RULES

1 Tarkastetaan vastaako tuotteessa olevat nimitiedot tuotetietopankkiin annettua nimeä. Lisäksi katsotaan onko nimi annettu nimeämissääntöjen mukaisesti. Mikäli tuotenimessä on kaikki vaaditut nimen osat, mutta ne ovat väärässä järjestyksessä, tulee tarkastustulokseksi varoitus. Nimen osan puuttuessa tai ollessa virheellinen tulokseksi tulee hyläty. Mikäli tuotteessa on ilmoitettu esim. sekä gramma että millilitratieto on molempien löydyttävä tuotenimestä. Valutetun painon tuotteilla on valuttu paino oltava tuotenimessä.

TUOTENIMI (35MRK) / ENGLANTI (EN)
Elvit 200ml C-V instant miracle gel

TUOTENIMI (35MRK) / SUOMI (FI)
Elvit 200ml C-V instant miracle gel

TUOTENIMI (35MRK) / RUOTSI (SV)

Figure 21 Product Inspection View

The number of attributes an inspector has to go through varies depending on the type of the product. For a base-level product, the count is approximately 70 whereas for a case-level product, it is approximately 15 attributes. Related attributes are bundled into groups to help organise the information. For example, the measurement group consists of attributes concerning the physical dimensions and weight of the product and the nutritional information group comprises attributes related to the ingredients and allergens.

Table 4 Valid Results of the Inspection Attribute

Code	Result	Description
0	Unable to validate	The attribute is not applicable for the product.
1	Passed	The data matches the information on the product.
2	Warning	Low severity inconsistency between the data and the physical product.
3	Failure	Inconsistency between the data and the physical product constitute a failure.

An attribute can be validated into one of four valid results, as listed in table 4. An inspection is considered finished once all of its attributes have been acknowledged. The inspector is also required to take a picture of the product using the built-in camera to serve as proof of inspection. When at least one picture has been associated, the inspection results are ready to be submitted to the server.

4.2.5 Product Reception

All new products entering the Finnish market after April 1, 2015 are subjected to the *new product inspection* mode. The same mode is also applicable to products that have undergone any changes in their packaging, composition, or GTIN code. The manufacturer sends a sample copy of the case-level product to the inspection centre (IC). Using the verification application and a barcode scanner, the IC personnel scan all inbound products to enter them into the inventory. Figure 22 below shows the reception details view prompted after a new product is scanned. For each product, the receiver must specify the following parameters:

- **Condition:** If a product arrives in a broken condition, the manufacturer will be notified so that a replacement can be sent. It is important that the product is in a good condition as not to affect the physical measurements or the readability of the packaging labels.
- **Priority:** By default, all new arrivals will be assigned a normal priority.
- **Inspect before:** The receiver can select a date in case the manufacturer has specified a time constraint to bring the product to the market. This will flag the system to prioritise the product in the inspection queue.

- **Decommission type:** By default, products will be donated to charity after inspection. Products with a short expiration date will be disposed and the ones with sensitive information can be sent back to the manufacturer.
- **Storage location:** Shelf spaces in the inspection centre are all marked with barcodes. The receiver scans the barcode encoded with the storage location information to denote where the product was unloaded.

Reception on 2

Cancel Reception Details CONFIRM Case Base

SCANNED ITEM · 23 FEB 2015 13:58:13 · NARNIA

Procter & Gamble Finland Oy
head&shoulders 250ml Apple Fresh shampoo
0541007623 0181
CONSUMER PACKAGE

Condition
Normal Broken

Procter & Gamble Finland Oy
head&shoulders 144kpl 1/4-lava
0408450005 8491
WHOLESALE PACKAGE

Condition
Normal Broken

Priority
Low Normal High

Inspect before
Saturday 28 February 2015

Decommission type
Charity Garbage disposal Return to sender

Storage location
Scan storage location barcode

Figure 22 Product Reception View

Once the reception is confirmed, the system will send an acknowledgement to the manufacturer via email. All eligible products are entered into the work queue awaiting inspection. As a rule, all new products have zero tolerance towards any discrepancy between the data provided by the manufacturer and the actual product. The product can only be approved when all errors in the data have been corrected.

4.2.6 Product Discovery

All application modules discussed so far were designed to accommodate the flow of the inspection work. The product discovery module exists as an auxiliary module to the application, leveraging on its access to the full product information database on the GS1 data bank. Through the product discovery interface shown in figure 23, the user can browse and view products by navigating the category tree.

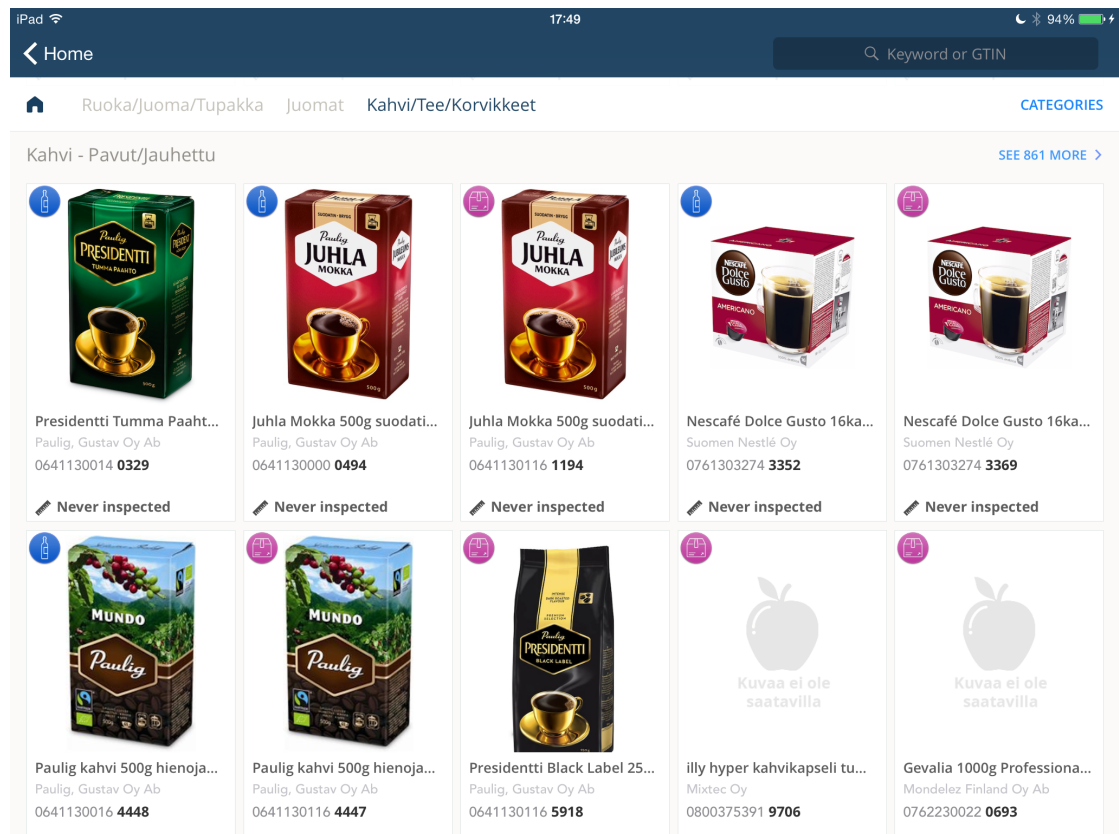


Figure 23 Product Discovery View

Selecting a product in the discovery view will show the product's details, as shown in figure 24. From the details view, the user can look up when the product was last inspected as well as the hierarchies the product belongs to.

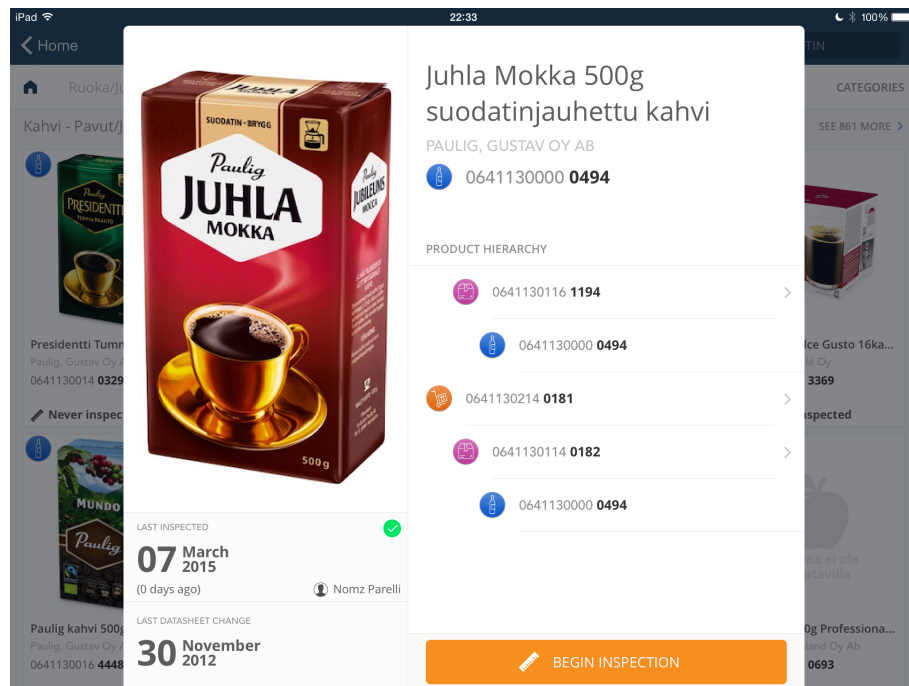


Figure 24 Product Details View

The product discovery view also contains a search control, which can be used for looking up product information by product name, brand, manufacturer, or GTIN code. An example of the search results view is illustrated in figure 25 below.

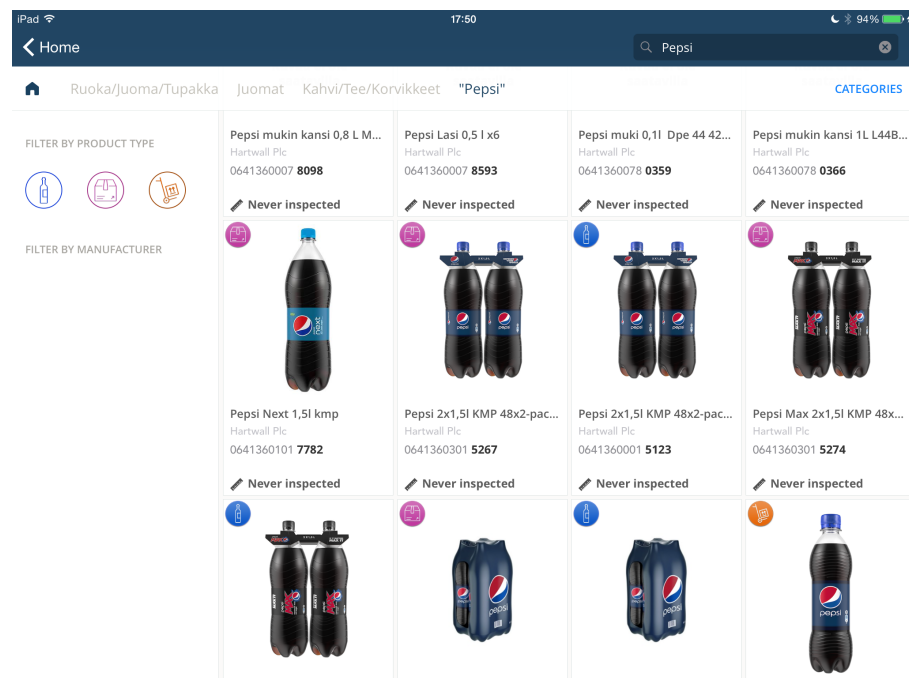


Figure 25 Search Results View

The search results can be narrowed down using a combination of search filters, such as by product type or by manufacturer.

4.2.7 Analytics

The analytics module is a reporting display, used by an inspector with a managerial role, for monitoring the overall progress and product inspection coverage. The tool can also be used for monitoring individual inspector performance. The server routinely records all changes made to the product data, enabling the system to trace the activities and calculate statistics.

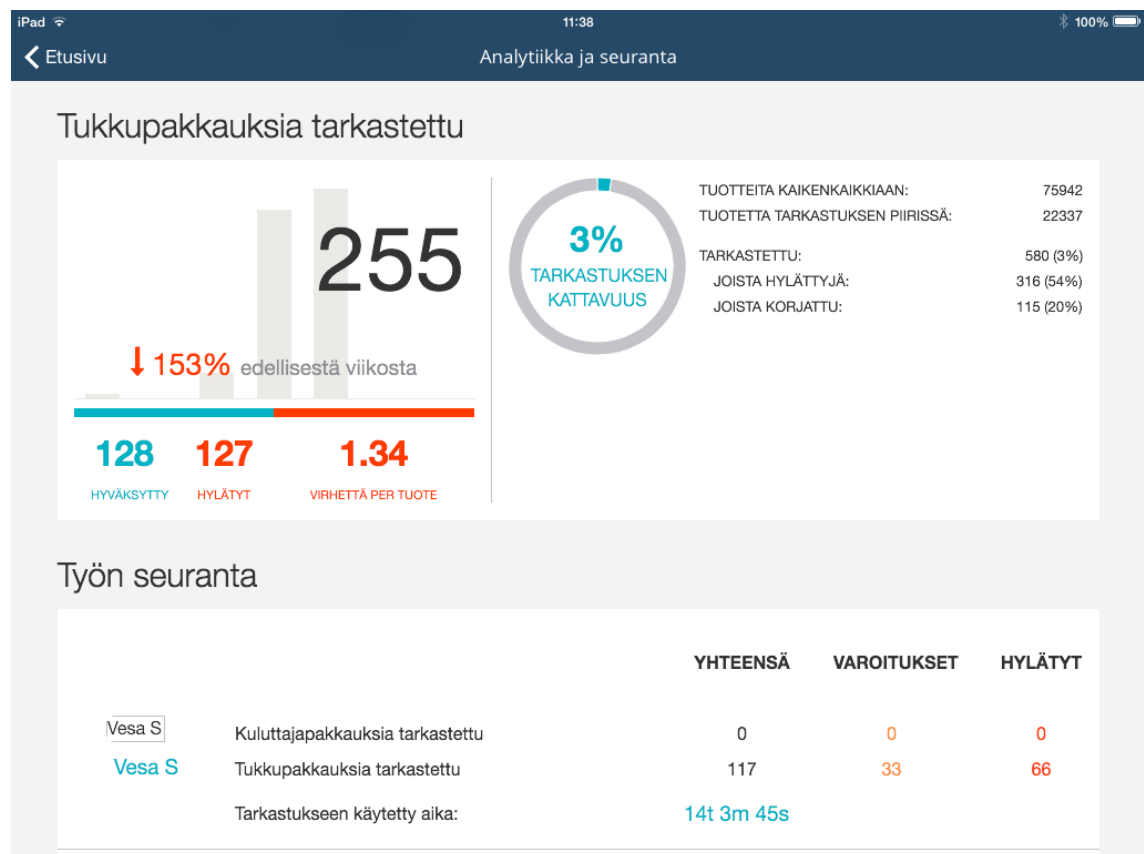


Figure 26 Analytics View

Figure 26 above shows the view presented to an inspector with the right privileges. The view provides a breakdown of the inspection progress for each product type and inspector. The data in the report can also be filtered for a specified time period.

4.3 Client–Server Functionality

All product information received from GS1 Finland is stored on the Foodie server, which acts as a central information repository. The server also contains the logic for determining which products are assigned to the inspectors according to a predefined order of priority. Additionally, the Foodie Server API provides methods for accessing resources such as *inspectionLocation*, *datasheet*, *codelist*, or *item*.

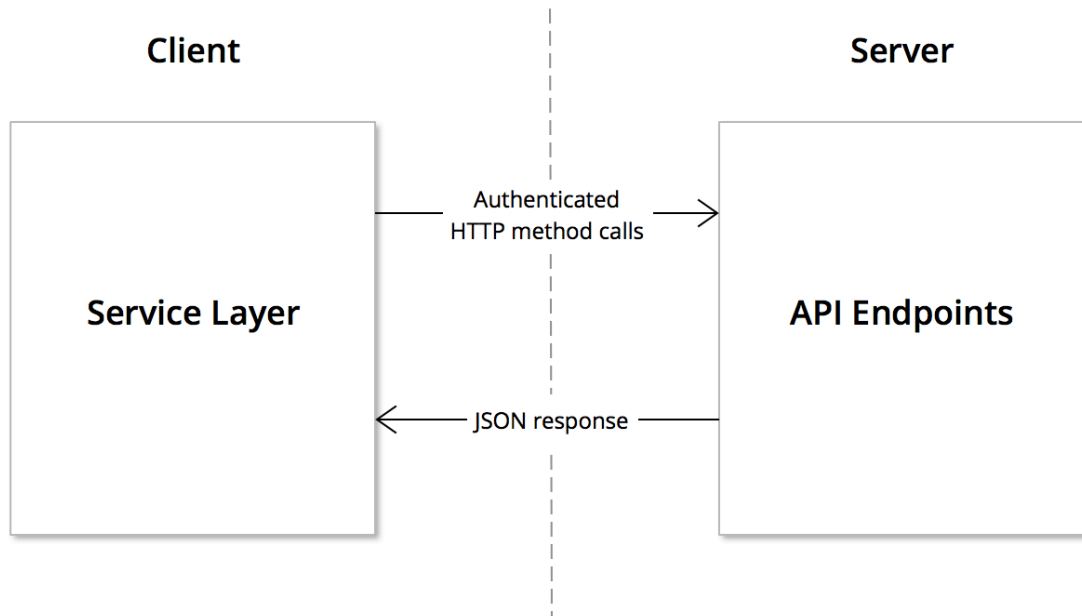


Figure 27 Client–Server Communication

As shown in figure 27, all communications between the client application and the server are authenticated and relayed using the Secure Sockets Layer (SSL) connection. API requests are signed with an application secret and a transaction identifier. The server has been configured to ignore all unauthenticated method calls.

5 Usability







An application is said to be usable when it performs according to the user's expectations. The ISO standard defines usability as to how well the users can use a product in a specified context to achieve certain goals in a manner that is effective, efficient, and satisfactory [15]. Assessing the usability of an application can be done by evaluating it against the general principles of user interface design, more broadly known as heuristics. According to Jakob Nielsen (1993), there are five quality components associated with usability – *learnability*, *efficiency*, *memorability*, *errors*, and *satisfaction* [16,26;17]. This chapter discusses the heuristics implemented and evaluates how well the Foodie verification application adheres to Nielsen's five quality traits.

Usability is very important for the Foodie verification application. On an average, it is estimated that an inspector is able to go through 100 wholesale products or 30 consumer products a day. That sum translates to approximately 1500 to 2100 attributes on a full workday. If the work is carried out over several consecutive weeks, the task can quickly become tedious and monotonous. Therefore, a significant amount of effort has been put into the design of the user interface and user interaction to ensure that the inspectors remain engaged with the application.

5.1 Learnability

Learnability is a measure of how easy it is for users to learn a new system to accomplish basic tasks [16,31]. At its core, the Foodie verification is a tool for performing inspection tasks. The majority of the time spent using the application will be in the inspection view. For this reason, most of the effort was focused on making the application as intuitive as possible. Starting an inspection task from the front page is never more than two taps away. User interface interactions and components were designed with delightful animations to not only appear attractive, but to provide affordances that guide the user in a logical flow. Additionally, all views were carefully designed to have only the relevant elements visible, while hiding inactive elements to reduce visual clutter. For example, the selected attribute expands a section to reveal its inspection rules and closes when the next attribute is selected. The result status control also animates out for the currently active attribute and collapses into its final result

indicating to the user that the attribute has been inspected. These examples are illustrated in figure 28 below.

GTIN-KOODI 03600522191186	
TÄYDELLINEN TUOTENIMI / ENGLANTI (EN) L'Oréal Paris Elvital 200ml Color-Vive Instant Miracle treatment gel for colored hair	
TÄYDELLINEN TUOTENIMI / SUOMI (FI) L'Oréal Paris Elvital 200ml Color-Vive Instant Miracle tehohoitogeeli värjätyille hiuksille	
TÄYDELLINEN TUOTENIMI / RUOTSI (SV) L'Oréal Paris Elvital 200ml Color-Vive Instant Miracle inpackningsgel för färgat hår	  


INSPECTION RULES

1 Tarkastetaan vastaako tuotteessa olevat nimitiedot tuotetietopankkiin annettua nimeä. Lisäksi katsotaan onko nimi annettu nimeämissääntöjen mukaisesti. Mikäli tuotenimessä on kaikki vaaditut nimen osat, mutta ne ovat väärässä järjestyksessä, tulee tarkastustulokseksi varoitus. Nimen osan puuttuessa tai ollessa virheellinen tulokseksi tulee hylätty. Mikäli tuotteessa on ilmoitettu esim. sekä gramma että millilitratieto on molempien löydyttävä tuotenimestä. Valutetun painon tuotteilla on valutettu paino oltava tuotenimessä.

Figure 28 Inspection Details View

Once an attribute is assigned a result, the next unanswered attribute will get selected automatically with its inspection rules section expanded. The colour of the attribute's value also changes to accentuate its selected state.

The automatic flow functionality also extends to the selection of data groups. Figure 29 below shows the next data group was automatically selected after the previous one was completed.


Tunnistetiedot

14 / 14 DONE

Tuoteluokittelu

3 / 5 DONE

Mitta- ja painotiedot

0 / 11 DONE

Figure 29 Automated Inspection Flow

As can be seen in figure 29, a data group cell comprises five pieces of useful information that are communicated to the user at a glance. First, the currently active cell is clearly conveyed with a vertical bar and a highlighted text colour, differentiating it from the rest of the cells. The progress is communicated with a horizontal progress bar, which gives a rough estimate of its completion. Additionally, a fraction figure is displayed next to the progress bar to cater for a more accurate state of the data group's progress. When the inspector scrolls through the list, a large and familiar checkmark symbol indents the cell's content, reducing the cognitive load on the user when trying to find the next incomplete data group. When the user wants to review the results, coloured dots were added to the cell to flag that its group contains at least one error or warning. The colour of the dot indicates the type of failure and matches the colours that the user has learned from the result status control. Despite being packed full of information, the cell presentation managed to remain organized and uncluttered.

Another common task that an inspector has to perform is to measure product dimensions and to enter the values into the inspection view. The standard Apple iOS virtual keyboard, seen in figure 30, does not allow for a restricted set of numeric keys. Approximately two thirds of the keys are redundant and the ones that are needed have too small of a hit area, inducing frustrations when the keys are mispressed.

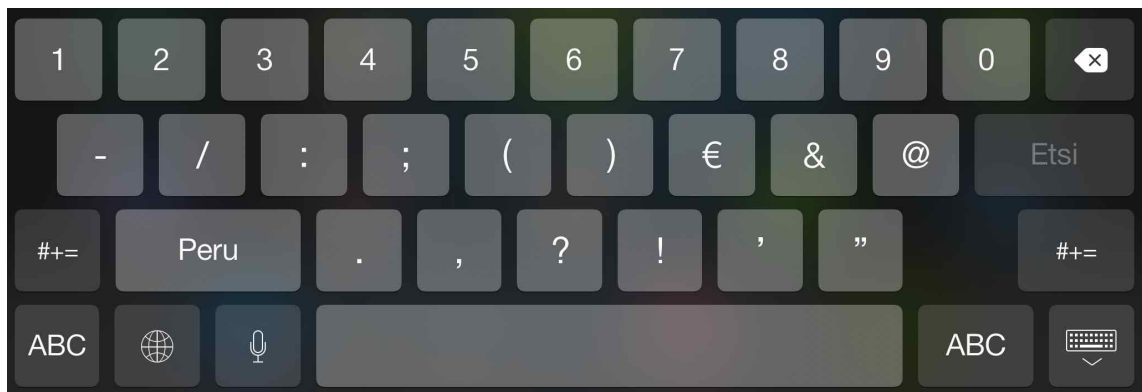


Figure 30 Standard Apple iPad Numeric Keyboard

To address this problem, a custom keyboard was designed, as seen in figure 31. The customized keyboard only includes the keys necessary for entering measurement values and a clearly labelled button for checking the result. The keys have also been arranged in the familiar style of a basic calculator to leverage on the user's existing motor proficiency in quickly entering numbers successively.

Pause #2659

L'Oréal Paris Elvital 200ml Color-Vive Instant Miracle
tehoitogeeli värjäytyille hiuksille

0360052219 1186 5%

2 FAILED 1 WARNING 1 PASSED 0 N/A

Tunnistetiedot
4 / 14 DONE

Tuoteluokittelu
0 / 4 DONE

Mitta- ja painotiedot

BRUTTOPAINO 260

EXPECTED MEASUREMENT 226 Gramma (GR) ± 10%

INSPECTION RULE
1 Tuote punnitaan pakkauksineen. Saatua tulosta verrataan tuotetietopankkiin ilmoitettuun bruttopainoon.
Bruttopainon toleranssi on tarkastuksen yhteydessä yleisistä säännöistä poiketen 10 %.

NETTOSISÄLTÖ

Input measurement: Check result

7 8 9 ←
4 5 6
1 2 3
0 ,

Figure 31 Custom Measurement Input Keyboard

Typically, the iPad is best suited for reading when held in the portrait orientation. Paragraphs are not too wide and users are comfortable with scrolling lengthy content vertically. Conversely, the landscape orientation is the more natural way when doing productive work. Accordingly, controls for performing repetitive tasks, such as collecting products or validating attributes, were strategically placed so that the thumbs could easily reach them when holding the device with both hands. Figure 32 helps to illustrate how such a small detail can vastly improve the ease of use and consequently make the user become more efficient.



Figure 32 Strategic Placement of Controls

Automatic cell selection, consistent use of colours for result and cell states, custom input keyboard, and strategic placement of key controls are examples of deliberate design decisions that make the Foodie verification application easy to use and allow the user to focus on getting the work done using the system.

5.2 Efficiency

An efficient system enables its experienced users to achieve a high level of productivity [16,31]. The Foodie verification application caters for beginners as well as experienced inspectors. The application obviates the need for an instruction manual by hiding the technical complexities and presenting the user with an interface that is self-explanatory. Barcode scanning capability in the system is a prime example of this. To the inspector, using the scanner is as simple as pointing the reader to a barcode. Behind the scenes, the application can respond in various ways, depending on the active context. For

example, in the product discovery mode, the read barcode invokes a product search based on the GTIN code. In the product inspection queue, scanning a product barcode initiates the corresponding product inspection. In the product collection mode, the application tries to match the read barcode with a product in the list and marks it as collected, if found. Figure 33 below illustrates the typical barcode types found on consumer product packaging.



Figure 33 Trade Item Barcodes

Additional complexity is introduced when the application has to parse the barcode in order to extract the product GTIN code. Determining the product type is trivial as base-level product packaging typically contains EAN-13 or EAN-8 barcodes, whereas case-level product packaging includes a GS1-128 barcode. A great variety of information can be encoded in a GS1-128 barcode, where each piece is tagged with an *application identifier* to specify its semantic meaning. The application gracefully handles all possible semantics and returns the expected product to the user, providing the confidence that the system behaves as expected.

As the barcode example above illustrates, the seamless interaction between the inspector and the application is achieved by intelligent system decision-making that is invisible to the user. Using the barcode scanner not only eliminates human errors, but also allows the inspector to become more efficient by saving time from not having to do manual comparison work.

5.3 Memorability

Memorability refers to how well users can use a system after a period of inactive use, without having to learn everything again [16,31]. This applies to inspectors returning from a holiday or managers who are only using the application intermittently, to extract statistical data or progress reports. The front page of the application, seen in figure 20, provides clear entry points for the available activities. Getting to the analytics view is

only a single tap away for the managers. Consistent workflow for allocating and managing inspection batches across different modes of inspection also allow the inspectors to immediately recall how to use the system. Each view has a clear purpose and all possible call-to-action buttons are visible and labelled, negating the inspector from having to remember the contents of a menu.

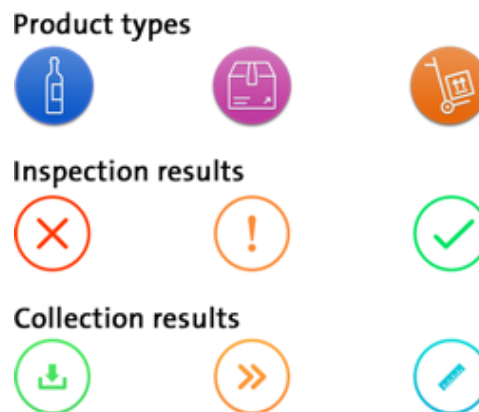


Figure 34 Icon Set

The extent of design consistency in the application is not limited to the workflow, but includes the usage of typeface, conformance to the platform standard guidelines, and usage of custom icons. Figure 34 shows the set of key icons that is used throughout the application. Users can easily recognise the icons and associate them with their actions, knowing that the same action will always have the same effect. This has the advantage of reinforcing the user's confidence when using and operating the application.

5.4 Errors

A high rate of failure is intolerable because users should be able to carry out the functions of a system without frequent errors [16,26]. Bugs in software are inevitable. Even with rigorous manual and automated testing, systems fail all the time and users will always find a way to use an application in a way the programmer never intended it to be used. Too often software is written to only handle situations where everything goes as planned. Error situations are equally important to be handled and the best way to prevent errors is to prepare for them. In a client-server architecture such as the Foodie verification, unplanned server outages, unexpected response data, no network coverage or programmer errors are examples of possible failures that could occur.

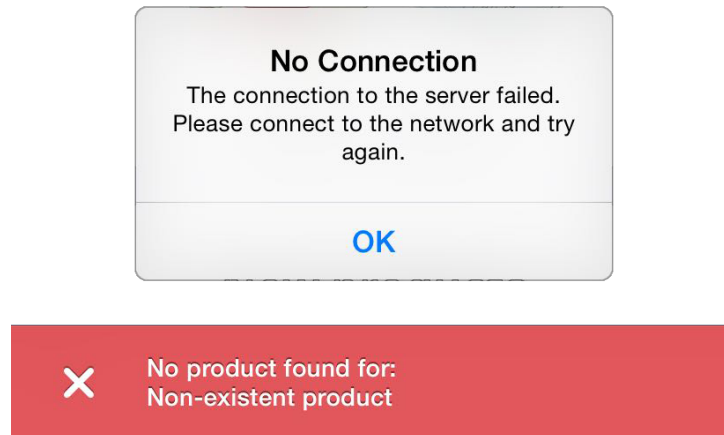


Figure 35 Error Messages

Figure 35 above shows example error messages displayed in the application. The user is always kept informed with descriptive messages and possible recommendations on what action to take. All inspection data are routinely saved to the disk memory. In the event of a crash, the application should be able to recover most, if not all, of the data from before the crash occurred.

5.5 Satisfaction

A system should be designed with the goal that it is pleasant to use [16,31]. Every touch input in the application is responded with a delightful animation. Entering an invalid password will cause the login screen to shake briefly. Selecting a cell will cause it to get depressed before bouncing back. Successfully submitting an inspection result will greet the user with a large “submitted” label, giving the user a sense of accomplishment. Navigating forward to a new view is done with a custom slide-in transition animation and navigating backward will have an inverse slide-out animation, as the user would expect to happen. Every action has a corresponding reaction. Users feel that the application reacts and behaves as it should. This symmetry can give a sense of satisfaction and comfort to the user. Making the extra effort to implement these custom animations may seem redundant and subjective. However, if the inspectors enjoy using the application, it can definitely help them to stay engaged and become more productive.

6 Discussion

6.1 Results

The Foodie verification application was deployed for production use and has received numerous incremental updates since its original release. All the required features were thoroughly tested by several inspectors at various warehouses. The feedbacks received on the usability of the application were positive. The application is said to be easy to use, unambiguous with an intuitive workflow, and most importantly, all the necessary actions function according to the inspectors' needs.

6.2 Challenges and Solutions

There were several technical challenges encountered during the development of the application. The first issue was regarding the automatic persistence of data to the disk memory. The application is expected to periodically save its inspection results and gracefully recover the data in the event of an application crash. The automatic saving was designed such that the entire object graph of the `DFInspectionTask` instance was encoded to disk when a key event took place. Occasionally, a crash would occur if the array collection were mutated when the persistence process was still in progress. The first attempt at fixing the issue was by throttling the frequency of the automatic save by trimming the key events. The trade-off meant that there was now a higher probability of data lost from a crash. Furthermore, the fix still did not prevent the issue from resurfacing, but only mitigated it by minimizing its likelihood to occur. Upon further introspection, a second attempt at addressing the root cause of the problem was made. Instead of persisting the same array collection that was being mutated, `DFInspectionTaskManager` now performs a deep copy of the inspection data array collections, which in turn, are serialized and persisted to the disk memory.

Another unforeseen issue that presented itself was a low memory condition on older generation devices. During the development phase, the iPad mini 2 was the primary device used for testing. The device is equipped with 1 GB of Random Access Memory (RAM). The memory footprint of the application on a normal use typically ranges between 30 MB to 80 MB. The memory consumption peaks when the inspector attaches several photos to the inspection results. Keeping the raw image data in the

heap causes the memory footprint to jump in the range of 150 MB to 250 MB. Unfortunately, the devices used by the inspectors were of the older generation iPad mini 1, which only feature a 512 MB of RAM. The inspectors were going through approximately 100 product inspections a day, or potentially up to 200 product images that are kept in memory. For this reason, the users were experiencing crashes immediately following an image capture. The issue was eventually resolved once I was able to pinpoint the root cause of the problem. Now, the application only keeps a maximum of two images in the memory and saves the rest on the disk. When a new image is captured, the `DFInspectionImageManager` component will purge old images based on the First-In-First-Out (FIFO) cache replacement strategy. By the same token, the component will also purge all images from the disk cache that have been submitted or cancelled to prevent them from accumulating over time.

Yet another challenge that was known during the design phase was related to sending large image files under poor network conditions. Instead of submitting the inspection results and the image data in a single network call, they are separated into two payloads. The application employs two network operation queues that function at different priority levels. The use of this method required the support from the API to be able to receive the image data non-sequentially. Consequently, this approach reduces the size of the results payload and allows it to be submitted even when the network connectivity is poor. In order to ensure that the image data are eventually submitted, a bookkeeping mechanism was implemented in the `DFInspectionImageManager` component, which keeps track of images in the submission queue. The component periodically checks the queue and attempts to process its items until the queue is empty.

6.3 Future Development

Even though the application was designed based on the requirements specified by GS1, its components were structured to be customer-agnostic. The application has a baseline implementation that makes deploying repackaged versions for new customers trivial. For example, the `DFThemeManager` component exists as a configuration point for handling the application-wide look and feel. Changing the typeface and customising the colour palette for a new customer can be easily accomplished as all font face and

brand colour method calls are routed through common `UIFont` and `UIColor` categories.

In its current state, the application is positioned as a front-end solution for verifying data that already exist in a data bank. However, it is no too far-fetched to consider that a customer might request the capability for capturing new product information. A new type of user, called *data supplier*, could be introduced into the use case. The application can be extended to function as a full-featured and dynamic input form used by data suppliers or manufacturers for entering new product data into the data bank.

7 Conclusions

The goal of developing a front-end system for assuring the integrity of trade item information in the GS1 data bank was met. An iPad application was written to work in conjunction with a server-side application that allows product inspectors to plan, manage, monitor, and carry out inspection work at warehouses and retail stores. The system was designed to include the complete feature set specified by GS1 Finland. Given the proper productisation opportunity, further development can be made to include product data capturing in the system, turning the application into a more versatile solution to manufacturers and potential new customers.

The emergence of online grocery stores has driven the need for product information to be made available digitally. Consumers rely on the accuracy of the product information displayed in order to make an informed purchase decision. The hope is that having a quality assurance system in place – enforced by an authoritative organization – will protect consumers from false information, thus increasing the consumer trust and accelerate the growth of the online grocery industry.

References

- 1 Henry V. The Rise and Rise of Online Grocery Shopping [online]. IGD Retail Analysis; 10 February 2015.
URL: <https://www.igd.com/Research/Shopper-Insight/Channels-and-In-store/25717/The-rise-and-rise-of-online-grocery-shopping/>. Accessed 23 March 2015.
- 2 GS1. The GS1 General Specifications. January 2015;15(2):16-17.
- 3 Drobnik O. Barcodes with iOS: Bringing together the digital and physical worlds. Shelter Island, NY: Manning Publications; 2015.
- 4 GS1. An Introduction to the Global Trade Item Number (GTIN). December 2006.
- 5 Palazzolo M. Focus on Traceability – Regulation and Consumer Demand [online]. Food Regulations and Labelling Standards Conference; December 2013.
URL: <http://www.slideshare.net/informaoz/maria-palazzolo>. Accessed 23 March 2015.
- 6 GS1. EU Regulation on Food Information to Consumers [online]. GS1 Europe; 2012.
URL: http://www.gs1.eu/docs/news/EU_Regulation_on_Food_Information_to_Consumers_-_What_you_need_to_know_-_V1.0.pdf. Accessed 23 March 2015.
- 7 Mason Hayes & Curran. Impact Assessment of EU Food Information Regulation (1169/2011) on Food Manufacturers and Retailers [online]. 2012.
URL: http://www.gs1.org/docs/freshfood/EU_FIR_Impact_Analysis.pdf. Accessed 23 March 2015.
- 8 Iulia-Maria T. Best practices in iPhone programming: Model-view-controller architecture — Carousel component development [online]. University of Timisoara, RO. EUROCON 2011;1-4.
URL: <http://ieeexplore.ieee.org.ezproxy.metropolia.fi/xpl/articleDetails.jsp?tp=&arnumber=5929308>. Accessed 24 March 2015.
- 9 Sasine JM, Toal RJ. Implementing the model-view-controller paradigm in Ada 95 [online]. Loyola Marymount University, Los Angeles, CA. ACM 1995;202-211.
URL: <http://dl.acm.org/citation.cfm?id=376571&bnc=1>. Accessed 24 March 2015.
- 10 Hansen S, Fossum TV. Refactoring model-view-controller [online]. Journal of Computing Sciences in Colleges 2005;21(1):120-129.
URL: <http://dl.acm.org.ezproxy.metropolia.fi/citation.cfm?id=1088791.1088812>. Accessed 24 March 2015.
- 11 Freeman Eric, Freeman Elizabeth, Sierra K, Bates B. Head First Design Pattern. Sebastopol, CA: O'Reilly Media; 2004.

- 12 Apple Inc. Model-View-Controller - General concepts [online]. iPhone Library Documentation, Apple Developer Center; 2012.
URL: <https://developer.apple.com/library/ios/documentation/General/Conceptual/CocoaEncyclopedia/Model-View-Controller/Model-View-Controller.html>. Accessed 24 March 2015.
- 13 Mahemoff MJ. Handling multiple domain objects with Model-View-Controller [online]. Melbourne University, AU. Technology of Object-Oriented Languages and Systems 1999;28-39.
URL: <http://ieeexplore.ieee.org.ezproxy.metropolia.fi/xpl/articleDetails.jsp?tp=&arnumber=809412>. Accessed 24 March 2015.
- 14 Apple Inc. Introduction to Key-Value Observing Programming Guide [online]. Mac Developer Library, Apple Developer Center; 2012.
URL: <https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/KeyValueObserving/KeyValueObserving.html>. Accessed 24 March 2015.
- 15 ISO 9241-11. Ergonomic requirements for office work with visual display terminals (VDTs) — Part 11: Guidance on usability. Geneva: International Organization for Standardization (ISO); 1998.
- 16 Nielsen J. Usability Engineering. Amsterdam: Morgan Kaufmann; 1993.
- 17 Nielsen J. 10 Heuristics for User Interface Design [online]. Nielsen Norman Group; 1995.
URL: <http://www.nngroup.com/articles/ten-usability-heuristics/>. Accessed 22 March 2015

GS1 and GS1-8 Prefixes

Synopsis of GS1 Prefixes	
GS1 Prefixes	Significance
000 - 019	GS1 Prefix* (used to create U.P.C. Company Prefixes)
02	GS1 Variable Measure Trade Item identification for restricted distribution
030 - 039	GS1 Prefix
04	GS1 restricted circulation number within a company
05	GS1 US Reserved for future use
060 - 099	GS1 Prefix (used to create U.P.C. Company Prefixes)
100 - 199	GS1 Prefix
20 - 29	GS1 restricted circulation number within a geographic region
300 - 976	GS1 Prefix
977	Allocated to ISSN International Centre for serial publications
978 - 979	Allocated to International ISBN Agency for books, portion of 979 sub-allocated to International ISMN Agency for music
980	GS1 identification of Refund Receipts
981-984	GS1 coupon identification for common currency areas
985 - 989	Reserved for further GS1 coupon identification
99	GS1 coupon identification

Synopsis of GS1-8 Prefixes	
GS1-8 Prefixes	Significance
0	Velocity Codes
100 - 139	GS1 Prefix
140 - 199	Reserve
2	GS1 restricted circulation number within a company
300 - 969	GS1 Prefix
97 - 99	Reserve