A Location Based Shopping List

Michael Mc Donnell

Kongens Lyngby 2009
IMM-B.Eng-2009-04

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

Abstract

People have always needed reminders, from tying a knot around the finger to writing a to-do list. The types of reminders have changed with the addition of modern technology, but the need is still the same. Technologies that use temporal events can be seen in many places, such as electronic calendars that remind users to go to a meeting a few minutes in advance.

With the recent release of Google's Android platform, developers now have the opportunity to create applications that use spatial events, such as location reminders. This thesis explores how location reminders can be used, by implementing a location based shopping list for the Android platform.

The primary and secondary use cases for a location based shopping list were identified, and the primary use cases were thouroughly described by studying how a paper shopping list works. Several user interface designs were described, and the best design was chosen by comparing the different designs' strengths and weaknesses.

A prototype that implemented the primary use cases was implemented and tested on an HTC G1 smart phone. The tests included an acceptance test, unit tests, a usability test, and a field test.

The results of the usability tests suggested that the prototype was easy to use and user friendly, but inconclusive in whether the implemented prototype was as fast to use as a paper shopping list.

Field testing showed that location reminders only worked well using GPS and not with Cell-ID positioning on the G1 smart phone. The smart phone, however, became unusable because the GPS used too much power.

The work done demonstrated that it is possible to implement applications that use location reminders for the Android platform, but the G1 hardware is still not capable to support location based applications.

Resumé

Mennesket har altid haft brug for påmindelser, lige fra at binde en sløjfe omkring en finger til at skrive huskelister. Typen af påmindelser har ændret sig i takt med introduktion af moderne teknik, men behovet er stadigvæk det samme. Teknologier som udnytter tidsafhængige hændelser kan ses mange steder, så som i elektroniske kalendere som kan give påmindelser om møder kort inden et møde finder sted.

Google har for nyligt frigivet Android platformen, som bl.a. nu har muligtgjort at udvikle programmer til mobiltelefoner som bruger stedafhængige hændelser. Dvs. at det nu er muligt for programmer på mobiltelefoner at give påmindelser når man ankommer til eller forlader en lokalitet. Dette eksamensprojekt udforsker muligheden for at bruge lokalitetspåmindelser ved at implementerer en lokalitetsbaseret indkøbsliste til Android platformen.

De primære og sekundære use cases der indgår i en lokalitetsbaseret indkøbsliste blev identificeret, og de primære use cases blev grundigt beskrevet ved at undersøge, hvordan en normal indkøbsliste bliver anvendt.

Der blev beskrevet tre mulige brugergrænseflade designs, og det bedste design blev valgt ved at sammenligne de forskellige designs styrker og svagheder.

En prototype blev implementeret der opfylder kravene der blev beskrevet i de primære use cases. Prototypen udviklet til og testet på en HTC G1 mobiltelefon. Der blev udført en sluttest, en brugervenlighedstest, unit tests og en marktest. Brugervenlighedstesten viste at prototypen var nem at lære, og use casesene kunne udføres hurtigt. Den gav dog ikke noget definitivt svar på hvorvidt prototypen er hurtigere og bedre end en normal indkøbsliste.

Testen i marken med prototypen på G1 mobiltelefonen viste at lokalitetspåmind-

elserne virkede godt ved brug af GPS, men var for upålidelige ved brug af Celle-ID positionering. GPS delen på G1 mobiltelefonen brugte dog så meget strøm at mobiltelefonen blev ubrugelig, da den ikke engang kunne holde til en hel dags brug.

Eksamensprojektet viste at selv om det er muligt at bruge lokalitetspåmindelser i programmer til Android platformen, så er de endnu ikke klar til udbredt anvendelse på den anvendte mobiltelefon pga. problemer med strømforbruget.

Acknowledgments

I would like to acknowledge the following persons for their contributions:

Ole Tange who briefly presented the idea of a location based shopping list at a local Linux user group meeting[1], where he gave a talk on the Openmoko project[2].

Teodor Filimon who is the author of the Tag ToDo application [4]. His to-do list provided inspiration in the user interface design.

The authors of the OI Shopping List[1]. Their shopping list provided inspiration in the user interface design.

Andrea Libelo for her support and help editing my thesis.

Jakob Eg Larsen my advisor at The Technical University of Denmark. Our regular meetings were a great motivator, and his comments and suggestions were invaluable.

[1]See `http://www.sslug.dk/emailarkiv/announce/2008_04/msg00001.html`
[2]See `http://www.openmoko.com/`

Contents

Chapter 1

Introduction

People have always needed reminders, from tying a knot around the finger to writing a to-do list. The types of reminders have changed with the addition of modern technology, but the need is still the same.

According to Malone [10] and Barreau et al. [2] computer users utilize visual reminders to organize their work. Placing both physical and virtual files currently used files where they would automatically stumble across them, e.g. on the desktop. This which suggests that the need for some form of reminder is inherentl within us.

Technologies that help remind people of certain events have become more widespread. For example Microsoft Outlook's calendar provides appointment reminders in the for of pop-up windows. Smart phones with built in calendars have appeared[1]. These can be synchronized with calendar systems like Outlook and online calendars such as Google Calendar[2].

PC programs and smart phone calendars are examples of de-centralized temporal[3] reminders. Centralized temporal reminder services such as *NemSMS*[4] have also appeared in Denmark. This is a service for public institutions to send reminders to mobile phones via SMS text messages. For example doctors can send appointment reminders to patients.

Location reminders are a second type of reminder that uses spatial events. These can remind people when they enter a location, leave a location, or are a certain distance away from a location. The Google Android platform has recently given developers the tools neccessary to build powerful location based applications for smart phones. This has been used in applications such as Locale [12]. Naone [11] gave the following description of Locale:

> *"The students' application, Locale, allows a user to program his or her phone to change its settings automatically depending on its location. For example, a phone might be set to change its ring to vibrate at the office but play a pop song when the user is at a favorite hangout."*

The Locale application can be programmed to trigger many of the smart phones functions, such as displaying a reminder or sending an SMS text message when the user enters a location.

This thesis describes a location based shopping list application prototype for the Google Android platform. The shopping list uses spatial reminders to remind people to buy items from their shopping list, when they arrive at the supermarket or other stores. The goal of the thesis is not only to make a shopping list

[1]E.g. the HTC Touch HD `http://www.htc.com/www/faqs.aspx?p_id=179&cat=0&id=85802`

[2]See HTC Dream features at `http://www.htc.com/www/product/dream/product-tour.html`

[3]Temporal events are time-triggered, e.g. one day left before an appointment

[4]See `http://oes.dk/sw52264.asp`

application prototype that uses location reminders, but one that is also fast and easy to use.

Chapter 2

Analysis

2.1 Replacing the Paper Shopping List

We use shopping lists as spatial reminders, allowing us to record items in our environment and access that information in another. But the conventional paper shopping list has the following drawbacks:

- You can forget to bring it to the store.
- You can easily lose it.
- You can forget to use it out once you are in the store.

So how can we solve the problems with a paper shopping list? One option is a location based shopping list application running on a smart phone. It solves the main problems of paper based lists, because:

- You will not forget to bring your shopping list, as long as you carry your smart phone.
- It is harder to lose a smart phone than it is to lose a piece of paper.
- An alert reminds you to use it when you enter the store.

In this way, a location based shopping list is preferable.

A paper shopping list does have a few advantages, however, that are important to address when creating a shopping list for a smart phone:

- It is easy to write a list on paper.
- It is easy to share it with another person.
- It can display all items at once, because it is not limited to a small screen.

The application must be able to compete with the advantages of the paper based shopping list if it is to be successful.

2.2 Paper Shopping List Scenarios

This section demonstrates how a regular pen and paper shopping list work using informal scenario descriptions. The scenario descriptions only describe the success cases. Failure scenarios are all described in Subsection 2.2.4.

To help describe the use of shopping lists we have created a persona called Andy. Andy is a geek who loves electronics and owns an advanced smart phone. He is a bit absent minded and often forgets appointments. He uses his smart phone's calendar to organize appointments and receive reminders. Andy lives alone, so he has not have anyone to help him with his shopping.

2.2.1 Scenario: Write a Paper Shopping List

Andy uses the last of the milk and wants to remember to buy more. He decides to write a shopping list. Andy does the following:

1. Finds a pen.
2. Finds a piece of paper.
3. Writes the title “Grocery”.
4. Writes “Milk”.
5. Places the shopping list where he can find it again, such as the side of the refrigerator.

When Andy later wants to add cornflakes to his shopping list, he must:

1. Find a pen.
2. Find his old shopping list.
3. Add ”‘Cornflakes”’.
4. Place the shopping list where he can find it again.

Andy adds several more items over the course of a few days. His list ends up containing the following items:

- Milk
- Cornflakes
- Apples
- Cheese

Andy tries to keep his shopping list stuck to his refrigerator, so that he will not lose it.

When it comes time to go to the store, Andy sometimes adds many items at once to the list before leaving. In this way Andy can quickly and easily make a shopping list with all the things he needs.

2.2.2 Scenario: Shopping With a Paper Shopping List

Andy wants to go shopping. He does not want to forget anything, so he brings a pen and the shopping list he stuck on his refrigerator. In the supermarket he looks at his list. He walks through the fruit section first, so he needs to find out what he has on his list from that section. He sees that he has apples on his list. He puts some apples into his shopping cart and crosses apples off the list.

He continues moving from section to section in the supermarket until he has found everything on his list. Andy walks to the cash register when he has checked off all the items on his list. He throws out his shopping list after leaving the supermarket.

2.2.3 Scenario: Writing multiple paper shopping lists

Andy already has a shopping list with his grocery items. He also needs to go to the hardware store to buy materials for a home improvement project. He does not want to write this on his grocery shopping list, as he might accidentally throw it away after leaving the supermarket. Also, he wants to keep the hardware shopping list in his garage instead of in his kitchen.

Andy does the following:

1. Finds a pen.
2. Finds another piece of paper.
3. Writes the title “Hardware”.
4. Writes “Screwdriver” and “Screws” below the title.
5. Places the shopping list in his garage.

Andy can easily add more items, as he put his shopping list in his garage.

2.2.4 Failure scenarios

Things do not always work out for Andy as described in the scenarios. Andy does not always remember to stick his shopping list back onto his fridge. He sometimes leaves it where he placed it when he added something to it. This makes it difficult to find it again later.

Andy often has trouble finding a pen when he needs to add something to his shopping list. He then wastes time looking for one, or gives up and forgets to write the item on his shopping list.

Andy sometimes goes shopping directly on his way home from work, but he sometimes forgets to bring his shopping list to work. He also frequently loses it among his papers on his desk. Also, he often forgets to bring a pen to the supermarket. This annoys him because he cannot cross off items he has already found, making it harder to scan the list for the next item.

Andy keeps multiple shopping lists, one on his refrigerator and one in his garage. This makes it easier to remember where they are and more accessible for adding items. This, however also makes it harder to remember to bring both of them. He rarely uses his garage and often forgets he has a hardware shopping list.

2.3 Vision

The vision described here defines the aim of the application to be implemented. The application could be extended further which is described in Section 7.

Imagine a shopping list applicatio running on Andy's smart phone, where he can enter which items he needs to buy. He always has his smart phone with him. When he remembers an item to record, he no longer needs a pen and he does not have to walk to the room to find the shopping list, and he never forgets to bring it with him. The application can even alert him to buy items, when he enters the store, by sensing his location (via GPS or other means of positioning).

Andy can add which items to get and where to buy them without an additional pen and paper. The application can have multiple shopping lists, and each shopping list can have multiple store locations attached to it. He can receive alerts to buy items whether he is in Whole Foods Market or Costco. The application displays a reminder, and the correct shopping list opens when he acknowledges the reminder.

The application sorts the shopping list by item category, e.g. "‘milk"’ is placed under "‘dairy"’. The application displays a category on the shopping list only if there have been added items belonging to that category. An early prototype test demonstrates the importance of this feature[1]. The small screen on the mobile device makes it hard to find which items are needed in a given section. Andy would need to scroll or otherwise shuffle through all the items on the list. This would be time-consuming and could cause Andy to revisit aisles at the store.

The application comes pre-loaded with the most common items and categories. For example the application automatically places "‘milk"’ in the "‘dairy"’ category when it is added. It can sort the categories based on standard stores layouts. This way Andy sees the correct categories as he passes through each section of the store. Andy can also add items that the application does not recognize. The application then lears to place that item in that category on subsequent uses. Andy can similarly add new categories to the application and choose where to display them on the shopping list. For example, he adds the category "‘flowers"’ to the system. At his local supermarket, the flower section is the first section he passes as he enters so he places this new category at the top of his list.

Andy can view the shopping list and tick off items as he puts them in the shopping cart. He can this way keep track of which items he has already gotten, and which ones he needs to get next. He can clear the list of the items he has already gotten, when he is almost done shopping, so he can see if he missed something. He does not need to delete his list, as the list will become empty,

[1]See section A.1 in the Appendix about the impact of having a small screen.

when cleared.

2.4 Use Case Analysis

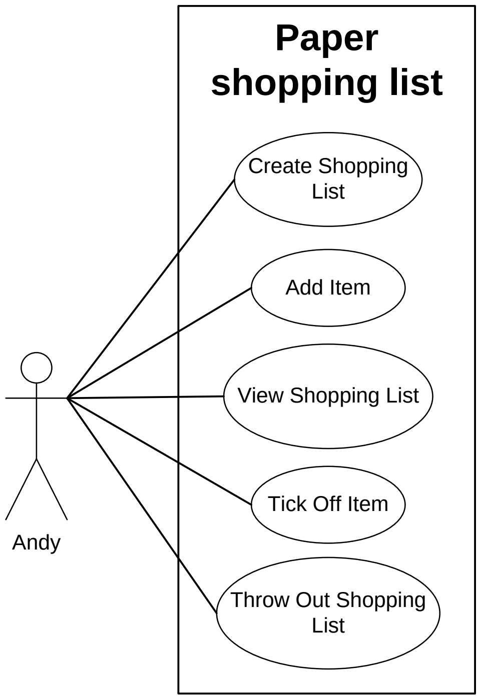


Figure 2.1: Basic use cases derived from the paper shopping list scenarios.

Based on the paper shopping list scenarios described in Section 2.2, the following basic use cases can be identified as shown in Figure 2.1. The basic use cases represent the tasks performed when using a paper shopping list. These basic use cases are not enough if you want to create a shopping list for a smart phone. The primary use cases were extracted from the vision statement and can be seen on Figure 2.2 with short descriptions in Table 2.1. Finally a description of the secondary use cases can be found in Table 2.2. The primary use cases describe the core functionality of the application and should be implemented in the prototype, whereas the the secondary use cases can be left out of the prototype.

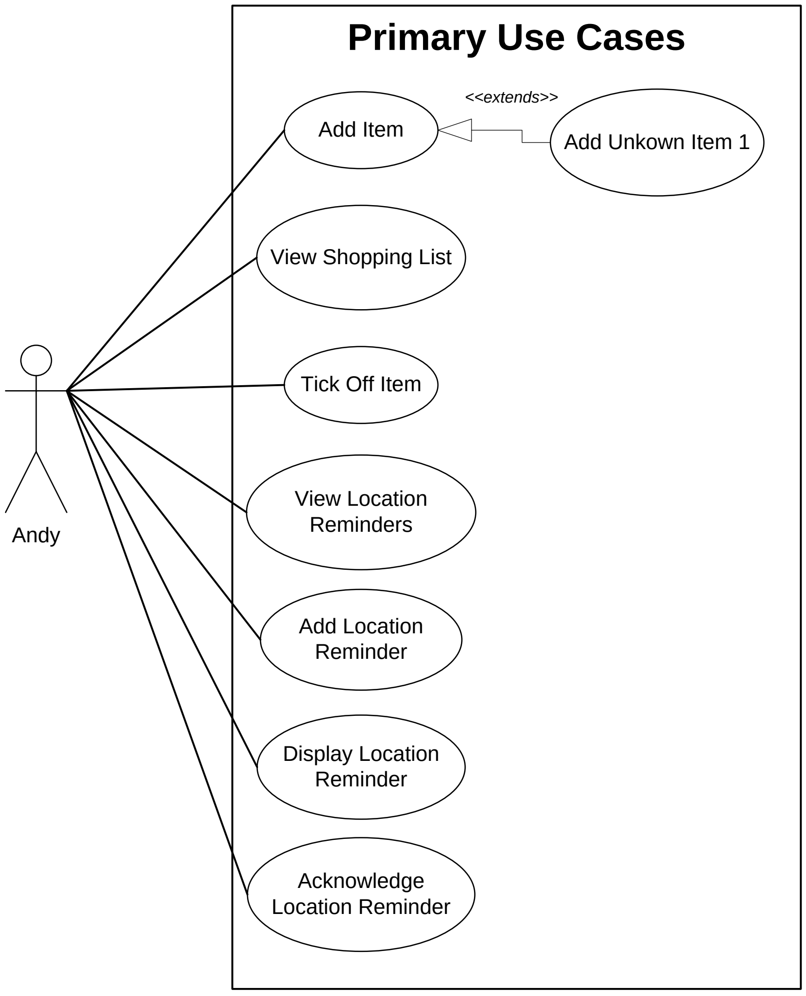


Figure 2.2: The primary use cases that should be implemented in the prototype.

Table 2.1: An overview of the primary use cases

Use Case	**Short Description**
Add Item	Andy wants to add an item, e.g. milk, to his shopping list.
Add Unknown Item 1	Andy wants to add an item to the system, which it does not recognize, but for which the category is known.
View shopping list	Andy is going shopping and wants to view his shopping list.
Tick Off Item	Andy bought an item and wants to tick it off from his shopping list.
Clear Ticked Items	Andy is done shopping and wants to remove all bought items from his shopping list.
View Location Reminders	Andy wants to see at which store locations he will receive reminders about items he needs to buy.
Add Location Reminder	Andy wants to receive location reminders when arriving at a particular store.
Display Location Reminder	The system discovers that Andy has arrived at a store where he needs to buy items, and alert him.
Acknowledge Location Reminder	Having been reminded, Andy wants to open the shopping list.

Table 2.2: An overview of the secondary use cases

Use Case	**Short Description**
Add Shopping List	Andy wants to add another shopping list to the system.
Delete Shopping List	Andy wants to remove a shopping list from the system.
Add Unknown Item 2	Andy wants to add an item to the system, which it does not recognize the item and for which the category is unknown.
Delete Item	Andy added an item that he did not need and wants to remove it from his shopping list.
Rename Item	Andy misspelled an item and wants to correct it.
Delete Category	Andy added a category that he no longer wants and wants to delete it from the system.
Rename Category	Andy added a category that he misspelled and wants to rename it.
Sort Categories	Andy added a category and wants to change where it appears on the shopping list in relation to the other categories.
Delete Location Reminder	Andy does not want to receive reminders anymore when he arrives at a particular store.
Edit Location Reminder	Andy wants to change the name or position of a location reminder.

2.5 Use Case Frequency Analysis

In the user interface design, the most frequently occuring use cases should be automated and accessible as much as possible, in order to increase the efficiency[2].

The best approach is to study the shopping list habits of several individuals, but this was not possible because of time and resource constraints. This frequency analysis is instead based on estimates of the developers own habits, despite of the limitations of this method.

I typically go shopping twice a week and purchase 8-12 items each time and use a paper shopping list. I usually buy:

- Bread
- Milk
- Cheese
- Fruit
- Meat
- Three kinds of vegetables

Items I buy less frequently:

- Cereals
- Household cleaning supplies
- Personal hygiene products
- Beverages

This means that I will have to add and tick off between 16-24 items each week.

As Andy does, I enter items on my shopping list over the course of the week rather than all at once. This means I use the shopping list several times a week. I only go shopping about twice a week, so I only strictly speaking need to view it twice a week.

I throw out the paper shopping list after each trip. This action is mapped to the use case **Clear Ticked Items**, which is used much as **View Shopping List**. It would however be useful to remove all ticked items from the list during a final check before going to the cash register, so I can be sure I have picked up every item. This means that the use case will be repeated about four times a week, because shop twice a week.

[2]A case of the classic 80-20 rule, where 80 percent of the time is often spent on 20 percent of the things you need to do.

Finally, each time I enter the store I will receive a location reminder, so the use case **Acknowledge Location Reminder** is performed twice a week.

The rest of the use cases and the secondary use case are rarely used. Some of them concern configuration of the system, and others are used for correcting mistakes. Their frequencies are difficult to estimate as they are performed irregularly.

The results of the use case frequency analysis is shown in Table 2.3. The frequency is estimated in uses per week. This does not take into account the duration of a particular use case. For example, a lot of time is spent viewing the shopping list, so this should receive more focus than the frequency of this use case might suggest. Therefor a use case frequency analysis is an incomplete meassure of the importanceof a use case.

Use case	**Frequency (Uses per week)**
Add Item	16-24
Tick Off Item	16-24
Clear Ticked Items	2-4
View Shopping List	2
Acknowledge Location Reminder	2
Add Location Reminder	Irregular
View Locations	Irregular
Secondary	Irregular

Table 2.3: An estimate of the frequency of the use cases.

2.6 Fully Dressed Use Case Descriptions

This section contains fully dressed use case descriptions. They describe the use cases in a formal manner with preconditions, postconditions(successful result), and what the user and system does. The fully dressed use case makes it easier to develop a prototype, as it is clear what happens. It also makes it easier to construct an acceptance test, as it should at least contain the steps described here.

2.6.1 Add Item

Precondition At least one shopping list has been created. The system knows which category the item, that is being added, belongs to.

Postcondition An item has been added to the chosen shopping list under a category.

1. The user selects which shopping list to add the item to.
2. The user enters the item.
3. The system adds the item to the selected list.

2.6.2 Add Unknown Item 1

Precondition At least one shopping list has been created. The system does not recognize the item of the category to which it belongs.

Postcondition An item has been added to the chosen shopping list under a category. The system will not ask the user which category the item belongs to if the item is re-entered again.

1. The user selects the shopping list to which he will add the item.
2. The user enters the item.
3. The system responds that it does not know to which category the item belongs.
4. The user selects the correct category.
5. The system registers the item as belonging to this category.
6. The system adds the item to the selected shopping list.

2.6.3 Tick Off Item

Precondition One or more items have been added to the selected shopping list.

Postcondition The system remembers that the item has been bought.

1. The user selects the bought item.
2. The system registers that the item has been bought.
3. The system responds that the item has been bought.

2.6.4 Clear Ticked Items

Precondition One or more items have been added to the selected shopping list, and one or more items have been ticked off.

Postcondition The selected shopping list only contains non-ticked items.

1. The user selects **Clear Ticked Items**.
2. The system removes all ticked items from the selected shopping list.

2.6.5 View Shopping List

Precondition At least one shopping list has been created.

1. The user selects the shopping list he wishes to view.
2. The system displays the shopping list.

2.6.6 Add Location Reminder

Precondition At least one shopping list has been created.

Postcondition The user is reminded if he enters the location.

1. The user selects **Add Location**.
2. The user types in the location name.
3. The user specifies the location coordinates.
4. The system adds the location reminder to the system.

2.6.7 View Locations

Precondition At least one shopping list has been created.

1. The user selects **View Location**.
2. The system displays the store locations that have been added and which shopping list they belong to.

2.6.8 Display Location Reminder

Precondition A location has been added to a shopping list. The shopping list has one or more items that needs to be bought.

Postcondition The user is reminded that he needs to buy items at the location.

1. The user moves close to the store where he need to buy items.
2. The system displays a reminder.

2.6.9 Acknowledge Reminder

Precondition The system has displayed a location reminder.

1. The user accepts the displayed reminder.
2. The system displays the shopping list where items needs to be bought.

Chapter 3

User Interface Design

The following section describes the chosen platform. Three user interface designs are described in the next sections:

- Separating each use case into separate screens.
- Using an alternative grid layout.
- Combining the use cases in list-centered design.

The platform is first chosen. It's Each design has advantages that must be addressed, such as:

- Ease of navigation.
- Screen real estate.
- Input speed.

The design is chosen at the end of the chapter based on these criterias.

3.1 Chosen Platform

Before a user interface can be designed the platform must be defined. Each platform has a number of common idioms that should be followed, for example a long press on an item on the Android platform brings up more options related to the item. Mobile devices have different ways of interacting with them, some have touch screens other do not. Some have full qwerty keyboards others on-screen keyboard, while some only have the basic numerical keyboard. The size of the screen is also very important as it limits how much you can display at once. This section describe the chosen platform.

The smart phone must have the following features:

- GPS or other means of positioning.
- Background processes that can continually monitor the user's position.

These features are essential for making a location based shopping list because the application cannot alert the user when he is in the right location, if it cannot continually monitor his position.

The HTC G1 had all of these capabilities. It runs Google's Android platform was released at the time when this thesis was first envisioned.

Some of the specifications that HTC lists for the G1 are:[1]

[1]Specifications taken from `http://www.htc.com/www/product/g1/specification.html`

- "*3.2-inch TFT-LCD touch-sensitive screen with 320 x 480 (HVGA) resolution.*"
- "*Slide-out 5-row QWERTY keyboard.*"
- "*GPS navigation capability with Google Maps.*"

It also has other features such as a SQLite database, WiFi, 3G, Bluetooth and a Camera.

There are many user interaction methods for the G1. Some notable methods are:

- Touch driven user interfaces.
- Large icons and widgets that are easy to touch.
- Long press brings up more actions related to the touched widget.
- The menu button on the phone brings up a pop-up menu at the bottom of the screen with more possible actions.
- The back button on the phone is used to return to the previous screen.

These means of interaction should be kept in mind when designing the application, to improve the ease of interaction for the user who is already familiar with the platform.

3.2 Separate Screens Design

This design separates each use case into a separate screen.

3.2.1 Add Item

One use case screen is shown on Figure 3.1. To add an item to a shopping list the user must enter the following:

- The item.
- The category to which the item belong.
- The name of the shopping list.

The design has three text fields and thus requires a lot of typing, but there are ways to reduce the need for typing. The application could remember what the user has previously typed and can make suggestions and auto-fill known fields.

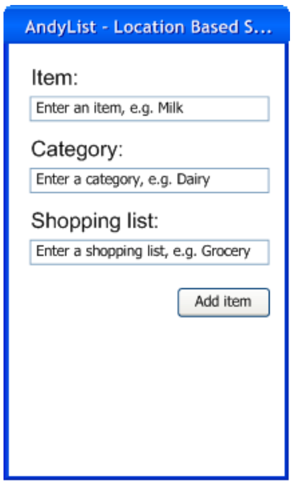

AndyList - Location Based S...
Item:
Enter an item, e.g. Milk
Category:
Enter a category, e.g. Dairy
Shopping list:
Enter a shopping list, e.g. Grocery
Add item

Figure 3.1: Adding an item to the shopping list.

Figure 3.2 on page 25 shows how the system suggests previously entered items when adding an item. The suggested items are shown in a list beneath the active text field. The user can select an item by touching the item on the suggestion list.

The list of suggestions becomes shorter as the user types more characters. For example the user types “m” and the application suggests several items starting with the letter “m”. The user then types “i” and the suggestion list is reduced to the items starting with “mi”.

Adding suggestions reduces the need to fully type longer words. However, it is not as helpful with short words, because it may be easier to continue typing instead.

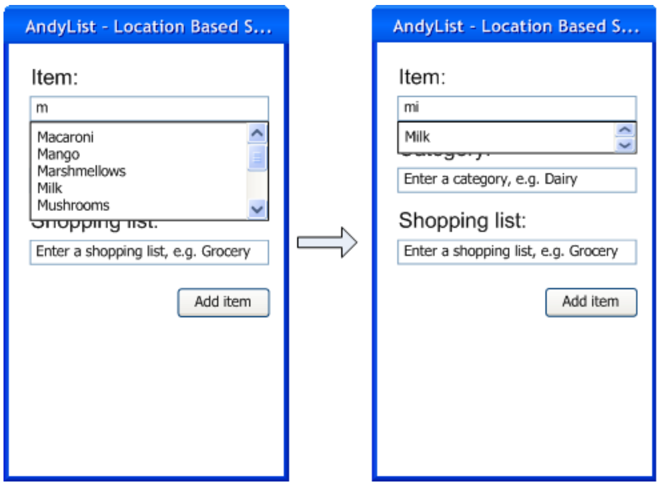
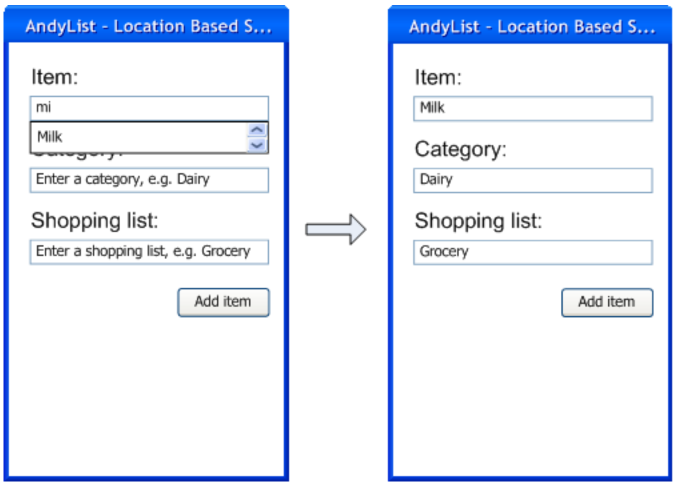

AndyList - Location Based S...
Item:
m
Macaroni
Mango
Marshmellows
Milk
Mushrooms
Shopping list:
Enter a shopping list, e.g. Grocery
Add item
AndyList - Location Based S...
Item:
mi
Milk
Enter a category, e.g. Dairy
Shopping list:
Enter a shopping list, e.g. Grocery
Add item

Figure 3.2: Previously added items are suggested to the user.

Figure 3.3 26 shows how the application can automatically fill text fields if a known item is entered in the item text field. It remembers which category an item belongs to and where it was last purchased. The user does not need to retype that information. The application then moves the focus to the “Add Item” button. This effectively cuts the needed user input to a third.


AndyList - Location Based S...

Item: mi

Milk

Category: Enter a category, e.g. Dairy

Shopping list: Enter a shopping list, e.g. Grocery

AndyList - Location Based S...

Item: Milk

Category: Dairy

Shopping list: Grocery

Figure 3.3: The form is auto filled if a previously added item is chosen.

3.2.2 View Shopping List

A shopping list, such as that in Figure 3.4 lists each item under its category, e.g. milk and cheese are listed under the dairy category. There is a tick box next to each item on the list. The user ticks these boxes as he puts each item on his list in his shopping cart, by pressing anywhere on the row containing the tick box and item name. This reduces mistakes by allowing the user a greater surface area on which to press.

The user can scroll through the list allowing him to see more items. He can change between different shopping lists by selecting them from the dropdown box in the top of the screen.

Figure 3.5 on page 27 shows a different way of switching between shopping lists, in which two screens are used. The first allows the user to select the shopping list, while the number next to each shopping list represents the number of items on that list. When the user selects a screen by pressing the appropriate row, the second screen displays the shopping list that was selected. The user can use the back button to return to the shopping list selection screen.

Both methods for switching between shopping lists carry advantages and disadvantages. The first method sacrifices some of screen real estate but also makes the switching more obvious. The second method allows more items to be viewed at a time and looks cleaner, but it also complicates navigation.

This method of displaying the shopping lists wastes screen real-estate as shown on Figure 3.6. This waste could be reduced by displaying the items in a grid like that in Figure 3.7. This, however, increases visual noise and can make finding an item more difficult.

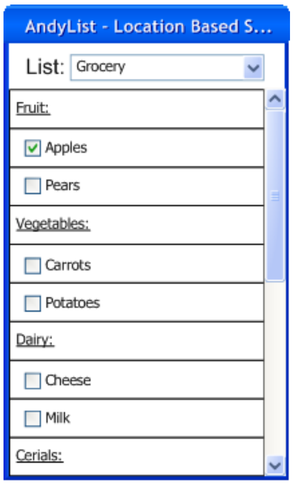


Figure 3.4: The shopping list where switching is done via a drop down list.

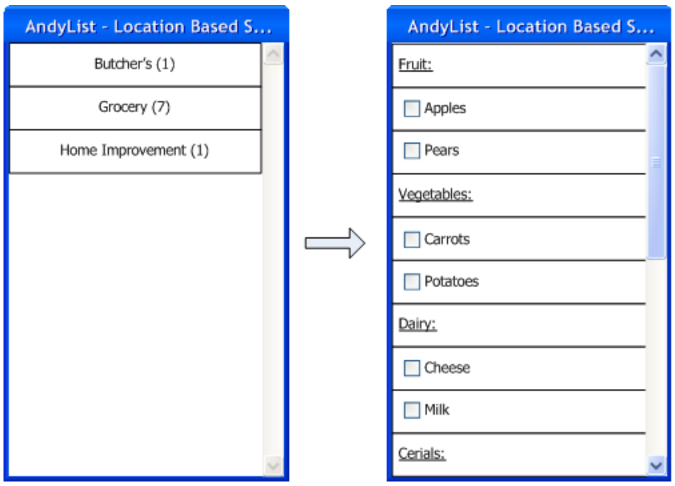


Figure 3.5: The shopping list where switching is done via nested lists.

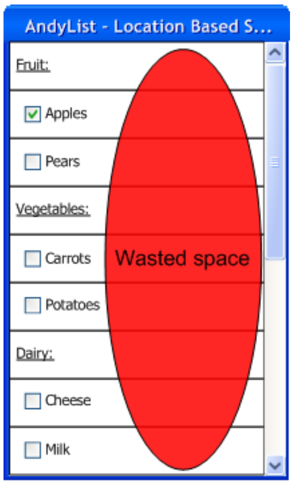
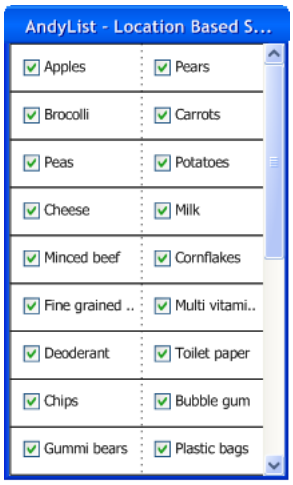


Figure 3.6: Screen real estate wasted to the right of each item.

Arranging the items in a grid also reduces the horizontal for space each item. Due to this loss of space, the text must be truncated and is harder to read.



✓ Apples	✓ Pears
✓ Brocolli	✓ Carrots
✓ Peas	✓ Potatoes
✓ Cheese	✓ Milk
✓ Minced beef	✓ Cornflakes
✓ Fine grained ..	✓ Multi vitami..
✓ Deoderant	✓ Toilet paper
✓ Chips	✓ Bubble gum
✓ Gummi bears	✓ Plastic bags

Figure 3.7: Items arranged in a grid.

3.2.3 Add Location

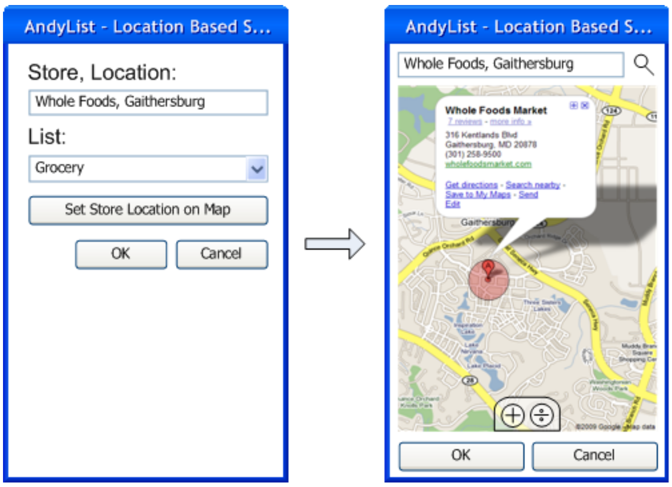
Users can add location reminders to a shopping list as shown in Figure 3.8. The described form can also be used for editing a location. The user types in the name of the location and chooses the shopping list. The user then presses the “Set Store Location on Map” button which searches for the location name on a map.

The user can move the pin to the desired location on the screen and extend by dragging it with his finger. He can also extend or reduce the radius by dragging the ring around the pin with his finger. The user will receive a location reminder when he enters the radius.

The zoom controls on the bottom zooms in and out of the map when touched.

The search box at the top of the screen can be used to search for a store. Search results are shown with small pins(not shown on the figure). The map pin that decides the location is automatically moved to one of the small pins when touched.

The OK button remains disabled until the user has entered the location name and chosen the location on the map. The user can alternatively press cancel if he decides not to add the location.



AndyList - Location Based S...

Store, Location: Whole Foods, Gaithersburg

List: Grocery

AndyList - Location Based S...

Whole Foods, Gaithersburg

Whole Foods Market
7 reviews - more info »
316 Kentlands Blvd
Gaithersburg, MD 20878
(301) 258-9500
wholefoodsmarket.com

Figure 3.8: Adding a location and setting the store location on a map.

3.2.4 Clear Ticked Items

The user can remove ticked off items from the list as shown on Figure 3.9. The ticked items will not be deleted entirely from the system, but only removed from the selected shopping list. The menu pops up at the bottom of the screen when the user presses the menu button.

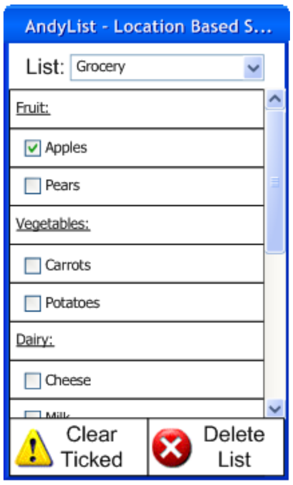


Figure 3.9: It is possible to remove all ticked items from the shopping list using the pop-up menu when viewing the shopping list, by pressing **Clear Ticked**.

3.2.5 Edit Category

Categories can be edited as shown on Figure 3.10. The edit actions are:

- Category deletion
- Category renaming.
- Category sorting.

A dialog with edit options appears when the user touches and holds a category New dialogs and screens are shown depending on which option he chooses.

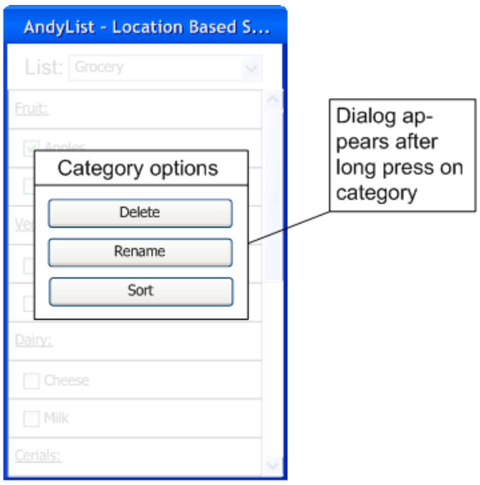


Figure 3.10: A long press on a category brings up a dialog with category options.

3.2.6 Edit Item

Te user can edit an item on a shopping list as shown on Figure 3.11. Similar to editing categories, the user touches the item and holds the item. A dialog appears where the user can choose to delete or rename the item. These options will lead to further dialogs and screens.

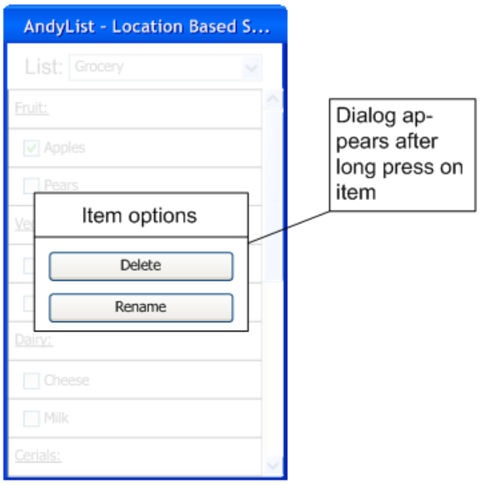


Figure 3.11: A long press on an item on the shopping list brings up a dialog where it is possible to delete and rename an item.

3.2.7 Sorting Categories

Categories are sorted by small up and down buttons on each side of the categories as shown on Figure 3.12. The user can move a category to its desired location by pressing the up button next to the category. The user can likewise move a category down by pressing the down button. The category moves up or down one row with each press.

Although successful, this method is tedious. The user must reposition his finger each time he wants to move a category another step, and multiple presses take time. A more efficient method allows the user to grab the category and place it directly in the correct place on the list. Figure 3.13 shows this alternative method. The user presses and holds a category to detach it from the list. He can then move it up or down the list by dragging it to the desired position and releasing his finger. In addition to its speed, this method is more spatial as people sort real-world objects in a similar manner, by picking them up to move them to a new location.

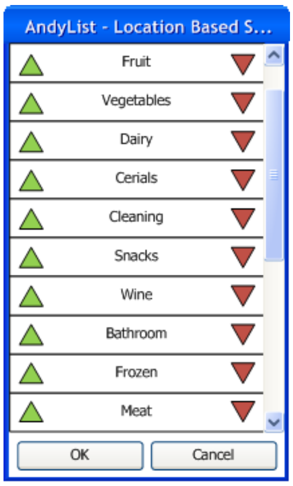


Figure 3.12: Move categories up on a list by pressing green arrow, and down on a list by pressing the red arrow.

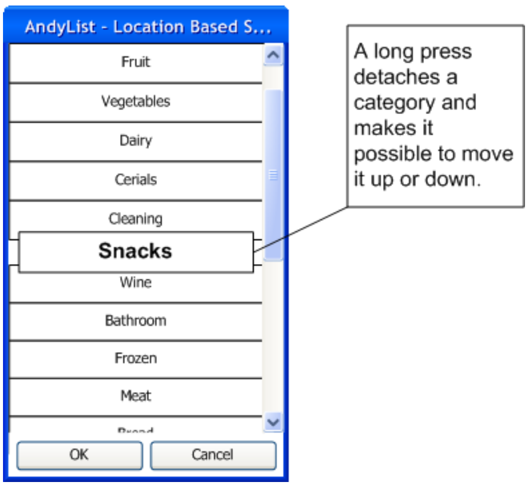

AndyList - Location Based S...
Fruit
Vegetables
Dairy
Cerials
Cleaning
Snacks
Wine
Bathroom
Frozen
Meat
OK
Cancel
A long press detaches a category and makes it possible to move it up or down.

Figure 3.13: Move a category up or down by first doing a long press and then moving it up and down, releasing it at the desired position.

3.2.8 Navigating Between Screens

Figure 3.14 shows a method of navigating among the screens via a home list. The user, for example, presses the "Add Item" button to go the the screen where he can add items to the shopping list. The back button returns user to the home list screen.

The disadvantage of this method is that the user will spend time navigating between use cases. The advantage is that each screen has ample screen real estate to itself because none of it is used by navigation.

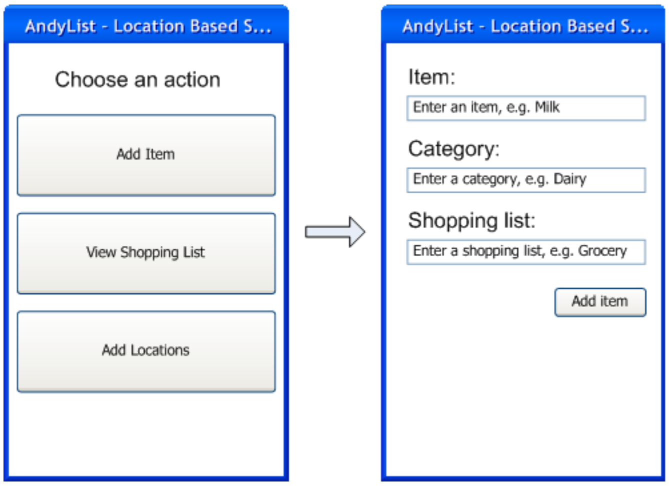

AndyList - Location Based S...
Choose an action
Add Item
View Shopping List
Add Locations
AndyList - Location Based S...
Item:
Enter an item, e.g. Milk
Category:
Enter a category, e.g. Dairy
Shopping list:
Enter a shopping list, e.g. Grocery
Add item

Figure 3.14: Navigating between the main use cases using a home list on the main screen.

Figure 3.15 shows a second method of navigating among screens. The user switches among screens that allow him to add items, view the shopping list and manage locations by pressing the corresponding tab. The user, for example, presses the "Add Item" tab to go the the screen where he can add items to the shopping list. The last used use case will be the default screen on start-up. The way of navigating is modeled after the built-in contact book and dialer for the Android platform.

The disadvantage of this method is that the navigation takes up screen real estate. The top part of the screen always contains navigation tabs. The advantage is that it is fast and easy to switch between uses cases. The user most often adds items to the list, so this will often[2] be the first screen on start-up because it shows the last used screen.

[2]The user only views the shopping list when he goes shopping.

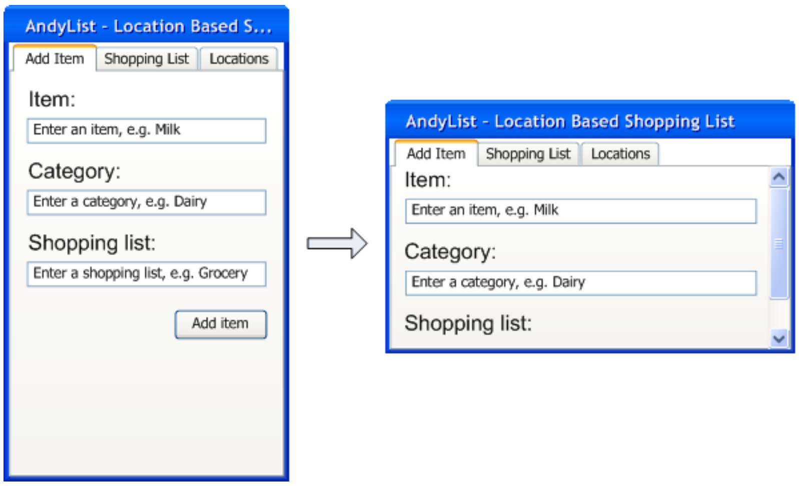


Figure 3.15: The last used use case is the main screen and navigation is done via tabs. Note that the tabs use a higher percentage of screen real estate in horizontal mode.

A third method of navigating between the use cases is shown on Figure 3.16. The user navigates by pressing options in a menu at the bottom of the screen. He can bring up the menu by pressing the menu button on the phone itself. The advantage of this method is that little space is wasted on navigation. The disadvantage is that it takes two presses to go to the most frequently used use case, **Add Item**. First the user must bring up the menu and then select "**Add Item**".

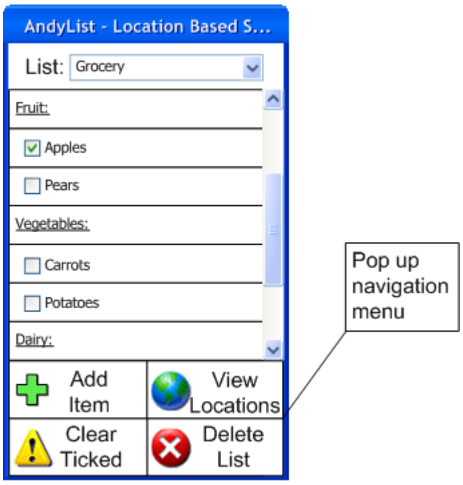


Figure 3.16: The shopping list is the main screen and navigation to other use cases is done via a pop-up menu.

3.3 Alternative Grid Design

The grid-based design was suggested by my advisor Jakob Eg Larsen. It displays small buttons as shown on Figure 3.17 representing each item, and arranges them in a grid a shown on Figure 3.18. The quantity can be increased by pressing the left side or decreased by pressing right side of the button. The button has a label with the item name and the quantity. Unselected items remain grey while items added to the shopping list turn white.

The alternative grid layout described has a few disadvantages. Adding items to the shopping from the long list is efficient than just typing them. The user must scroll through a long list of items to find what he needs. He must then press on the button once or multiple times.

There is more wasted space in this alternative layout than the other proposed layouts. The items that are not added to the list remain visible even if they are not needed.

Alternatively the proposed method could be used only for adding items to the list and then the list can be displayed in a separate screen as described in the previous section. Although this would improve screen real estate, it would not speed up the item searching.

One could speed up the the alternatively grid design by showing the most used items at the top. The user would then only need to scroll a long list if they wanted to add an uncommon item.

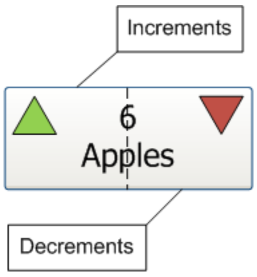


Figure 3.17: The button representing an item.

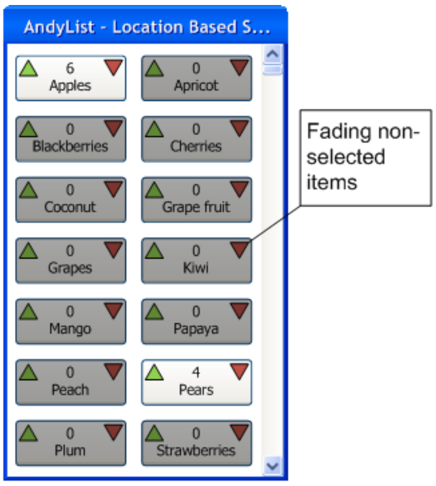


Figure 3.18: Items arranged in a grid.

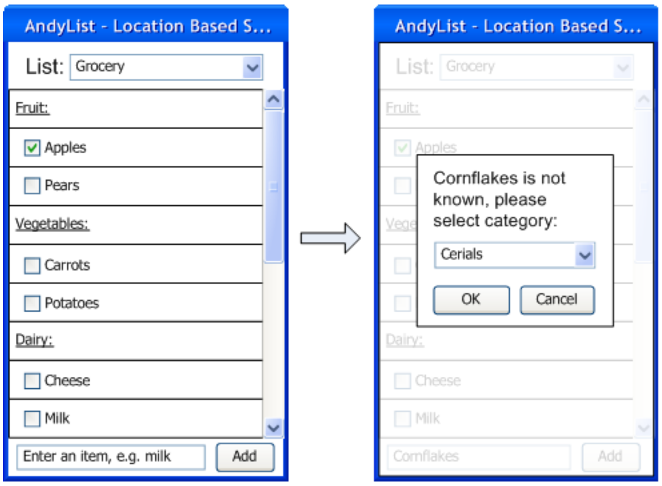
3.4 Combined List-Centered Design

The design described in Section 3.2.8, sacrificed screen real-estate for navigation. This space could be used to complete a use case instead. Figure 3.19 shows this possibility. The list has a text entry at the bottom of the screen where the user can add items. If the application does not recognize an item, a dialog is displayed where the user can enter the category. The item is added to "Uncategorized" if the user presses cancel. The number of inputs required to add an item to the shopping list is reduced by showing a pop-up list with suggestions as explained in Section 3.2.1. The application adds the item to the currently selected shopping list and remembers to which category the item belongs. This reduces the number of text entries needed compared to the method described on figure 3.1 on page 24.

Figure 3.20 shows the pop-up menu that appears with a press of the menu button. The menu contains options to:

- Remove the ticked items from the list.
- View and add location reminders.
- Add a new shopping list.
- Delete the currently selected shopping list

A long press on the category names allows the user to delete categories, rename categories, and sort categories as seen on Figure 3.10. It is also possible to delete and rename an item by a long press on that item.



Cornflakes is not known, please select category:

Figure 3.19: Items are added while viewing the shopping list.

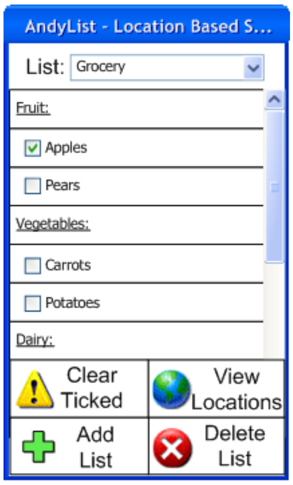


Figure 3.20: More actions are accessible through a context menu.

This design is efficient for adding new items to the list because it can be accomplished from the main screen. The most frequently used use cases (**Add Item** and **Tick Off Item**) are always accessible on the main screen. It is easy to switch between shopping lists because this is also done on the main screen. Less frequently used use cases have been reached by pressing the menu button, and the least frequently used use cases are accessible by long presses. This provides a good balance between how often a use case is used and how accessible it is.

3.5 Choice of UI Design

Creating a good user interface is not an exact science but a subjective process. Conducting usability tests that compare competing designs can help making the process more objective, but due to time and resource constraints, this was not feasible. Instead each design is rated by assigning point values to chosen criterias. The point values range from one to three, three representing the best possible performance. The rating system corresponds to: 1 for poor, 2 for satisfactory, and 3 for good. The following criteria where chosen:

- Ease of navigation.
- Screen real-estate usage.
- Input speed.

Table 3.1 shows the result of the evaluation of the designs described in Sections 3.2, 3.3, and 3.4.

The navigation used in the separate-screen design is clear, but it was slower than the combined list centered design, because it requires more button presses and screen changes. The combined list-centered design navigation is fast and requires fewer screen changes. Therefor the combined list-centered design received a higher navigation rating than the separate screens design. The alternative grid design was given the lowest score, because it wastes significant time scrolling.

The separate screen design was given a low score in screen usage because it wastes screen real estate. Each use case takes up an entire screen; the use case **Add Item**, for example, has three text entry fields and two buttons.The combined list-centered design is an improvement with only one text entry field and one button. The best, however, is the alternative grid design, which does not require any fields at all, and which can show more items at one time by the nature of the grid.

The input speed for all three designs is reasonably fast. However the combined list-centered design was rated as having the highest input speed because it required the least number of inputs of the designs.

The combined list-centered design received the highest overall score and was ultimately chosen as the design for implementation in a prototype of the shopping list application.

UI Design	**Navigation**	**Screen usage**	**Input**	**Score**
Separate-Screens	2	1	2	5
Alternative Grid Design	1	3	2	6
Combined List-Centered	3	2	3	8

Table 3.1: Rating of the proposed user interface designs.

Chapter 4

Design

4.1 The Android Platform

Applications for the Android platform must follow a few design rules, which will be covered in this section. An application can contain four different kinds of components[1]

- Activities
- Services
- Broadcast receivers
- Content providers

An application is divided into a number of screens, for example one showing the shopping list and one showing a map. Each of these screens must inherit from **Activity**, which is a requirement in the Android platform. Each activity can be started by sending an **Intent**, which is a built-in message passing class. This reduces coupling as each activity does not have to know anything about the other activities, but has to send the right intent. Activities can, this way, easily be substituted as long as the replacement responds to the same intent. This also has the benefit that each activity can be tested separately.

Services are a way to keep a part of the application running that does not have any graphical representation. It can carry out operations in the background for an application. Activities can connect to services and have them do work for them.

Broadcast receivers can listen to broadcasted intents just like activities, but they do not have any graphical representation like the services.

Content providers provide a way to interface with databases or other files via URLs. They enable data to be shared across applications and activities in a uniform way. They support the Observer pattern[2] so the views are automatically updated when the underlying data source changes.

4.2 The Main Architecture

This sections describes the main architecture of the application. Figure 4.1 shows the main classes in the application. The application is divided into many separate screens, which all inherit from **Activity**. There is a service, **LocationService**, which inherits from **Service**, and there are two broadcast receivers, **OnBootServiceStarter** and **NotificationSpawner**, that inherit from

[1]See *Application Fundamentals* [7]

[2]See [9] page 210 for more information on the Observer pattern.

BroadcastReceiver. All of these classes' responsibilities will be described in this section.

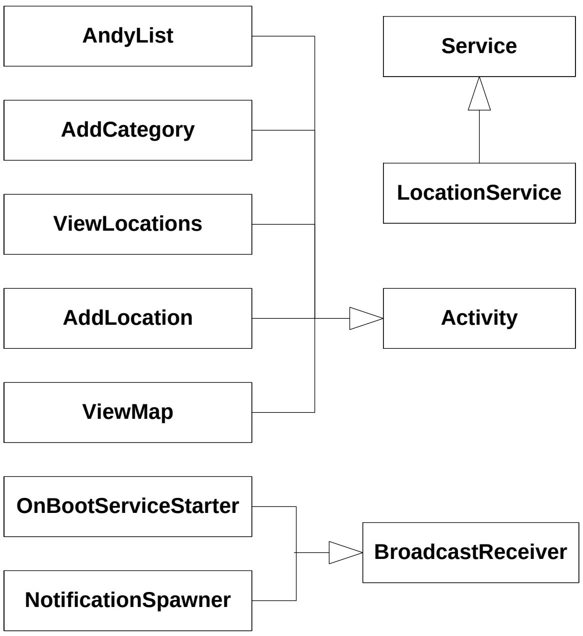


Figure 4.1: A simplified class diagram showing inheritance of classes.

The class **AndyList** is the main activity and the application starts in this activity. It shows the shopping list and starts other activies such as **ViewLocations**. It does not contain any functionality itself but is responsible of creating various objects and tying them together.

The classes **AddCategory**, **AddLocation**, **ViewLocations** and **ViewMap** are activities that each do what their name implies. The activity **AddCategory** is shown when the user needs to add to which category an item belongs. The activity **ViewLocations** displays all the location reminders and the shopping lists they are related to, but it can also spawn the **AddLocation** activity. The **AddLocation** activity in turn spawns the **ViewMap** activity when the user adds a new location reminder to the system.

The service **LocationSerivce** responsibilities are to insert, remove and update location reminders. It registers location reminders with the system service **LocationManager**. The **LocationService** is started on boot by the **OnBoot-**

ServiceStarter broadcast receiver. This is because services cannot listen to intents, so **OnBootServiceStarter** starts when booting has completed and then starts the service. The **OnBootServiceStarter** broadcast receiver is, therefor, only a workaround for the design limitations in the Android platform.

4.3 The Model-View-Control and the Observer Pattern

The Android platform uses several well known patterns such as Model-View-Control (MVC) and the Observer pattern. These patterns are built-in to the widgets and classes that come with the platform, so they were utilized in the design.

Figure 4.2 shows how these patterns are used in the application. The simplified class diagram shows five classes. These classes, except for the **Database-Helper**, class are not literal classes. They are generalizations of the types of classes that appear most often. The diagram shows how these classes interact.

The views are the classes that present information to the user, e.g. the shopping list with items. Each view displays data from a model that it observes. The view adapter is the model that the view observes. The view adapter provides a transformation of the data from the database adapters, so the real model is the database adapters. The view adapter, therefor, has to observe the database adapters so it can forward the change notifications to the view.

The use of the observer pattern makes the model inform its observers whenever it is manipulated. That means the views' display of the data is updated automatically, which reduces the chances of errors, as one does not have to manually update every view whenever the model is manipulated.

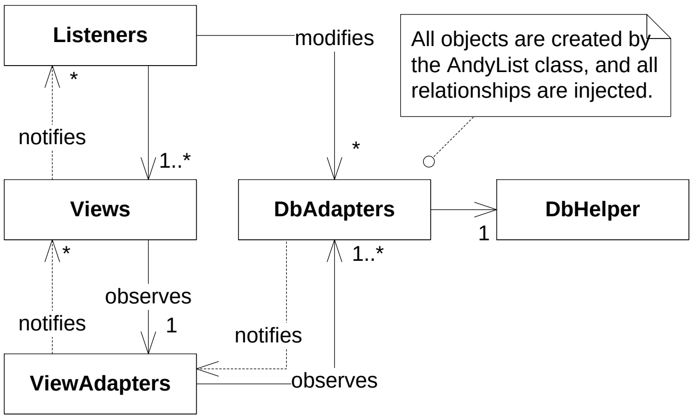


Figure 4.2: A simplified class diagram showing how MVC and the Observer pattern is used in the application.

The listeners[3] are what controls the program flow. They can read and write

[3]Also known as event handlers.

the state of a view and the model. The listeners contain the knowledge of what should be changed in the views and the models. For example, when the user clicks the add item button, the listener gets informed that the button was clicked. The listener reads the name of the item in the text entry view, clears the the text entry view, and then inserts the item in the shopping list model.

The use of MVC divides the responsibilities of the classes. This makes it easier to change to the implementation, as it is more obvious where the changes need to occur. It also makes it easier to test the implementation, because it is well defined what a class should be able to do.

It was mentioned in the previous section that the class AndyList was the main activity. Its responsibility is to instantiate all the needed objects and bind them together, which can be seen on Figure 4.3. This is how all the objects in the MVC pattern come to known about each other.

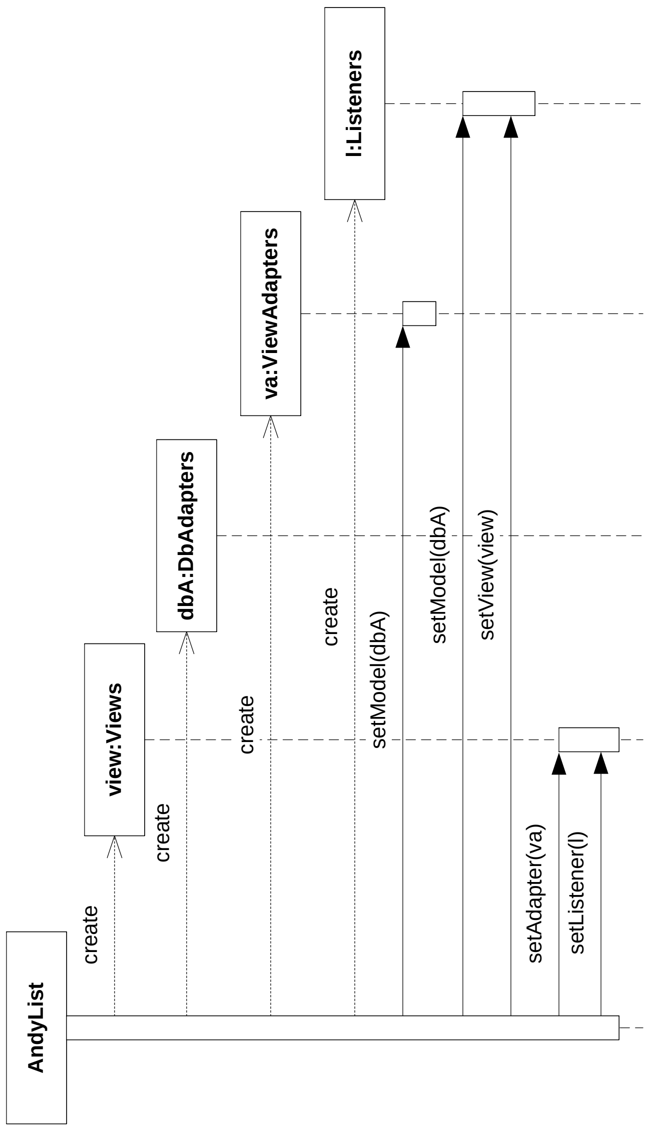


Figure 4.3: A sequence diagram showing how the objects are instantiated and tied together.

4.4 The Data Layer

The Android platform comes with SQLite, which is a small embedded SQL server[4]. SQLite is a fast, memory efficient and reliable solution, and it is being used in many embedded products, such as the Apple iPhone. Using the embedded database makes it easier to develop a data-heavy application. If one used custom XML-documents instead, then one would have to write a parser and functions to manipulate the documents. This would be error prone and create extra work, so SQLite is a good solution for persisting data.

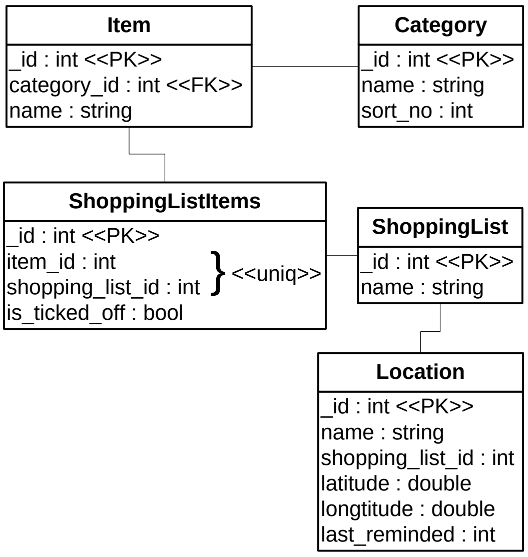


Figure 4.4: The database schema.

The database schema on Figure 4.4 describes what data is saved and how it is stored in the database. The items that go onto the shopping list have been split into two entities, **Item** and **ShoppingListItem**. This has been done so that it is possible to save and later retrieve the names of the items and which category they belong to, without them being on a shopping list.

The entity **Category** has also been split out from **Item** so that it is possible to have categories with out them necessarily being associated with any items.

Access to each table in the database is provided through database adapters as shown on Figure 4.5. As mentioned before, the database adapters are a part of the model. They contain methods to read and write data in the database. The

[4]Read more at http://www.sqlite.org/

methods are only shown for the **ItemDbAdapter** to make the diagram easier to read. All of the database adapters contain similar methods to read and write data in their associated table.

The database adapters are a light weight alternative to data mappers. The data mapper in a Data Mapper Pattern[5] can retrieve data from the database and use it to instantiate domain objects. It also persists domain objects to the database. The database adapters only work on database cursors so they do not waste memory or processing power on creating new objects. Google encourages developers to avoid unnecessary object creation [6] and uses database adapters in its own application examples.

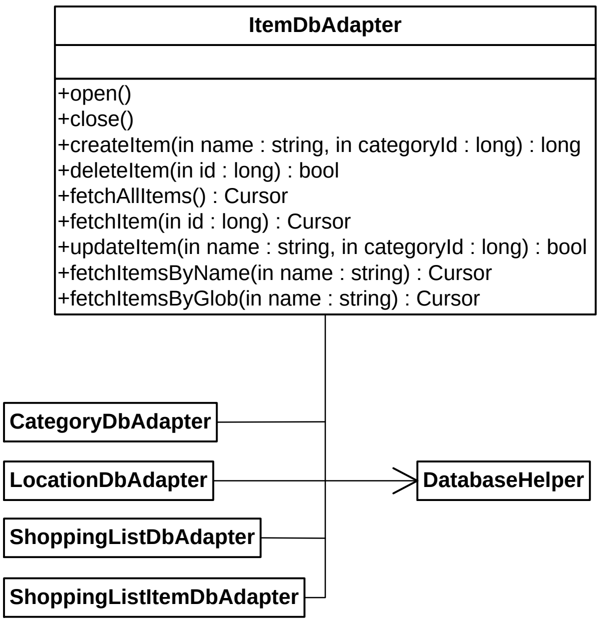

ItemDbAdapter
+open()
+close()
+createItem(in name : string, in categoryId : long) : long
+deleteItem(in id : long) : bool
+fetchAllItems() : Cursor
+fetchItem(in id : long) : Cursor
+updateItem(in name : string, in categoryId : long) : bool
+fetchItemsByName(in name : string) : Cursor
+fetchItemsByGlob(in name : string) : Cursor
CategoryDbAdapter
LocationDbAdapter
ShoppingListDbAdapter
ShoppingListItemDbAdapter
DatabaseHelper

Figure 4.5: A class diagram showing the database adapter classes. The methods are shown for the **ItemDbAdapter** class, but have been omitted for the others.

The database adapters could have been wrapped in content providers. There is not any interaction of the data between applications, so it was decided not to use content providers in order to save time on the implementation.

[5]See page 628 in [9]
[6]See the section *Designing for Performance* in [7]

4.5 Location Reminders

This sections describe how location reminders are enabled via proximity alerts. The **LocationService** is responsible for inserting, removing and updating location reminders. Figure 4.6 shows how the **LocationService** registers location reminders via proximity alerts in the system. It creates pending intents, those that can be fired at a later stage, that contain data about when it should be triggered and who should receive the intent. The **LocationService** registers the broadcast receiver **NotificationSpawner** to be started when the phone enters a radius of the desired position.

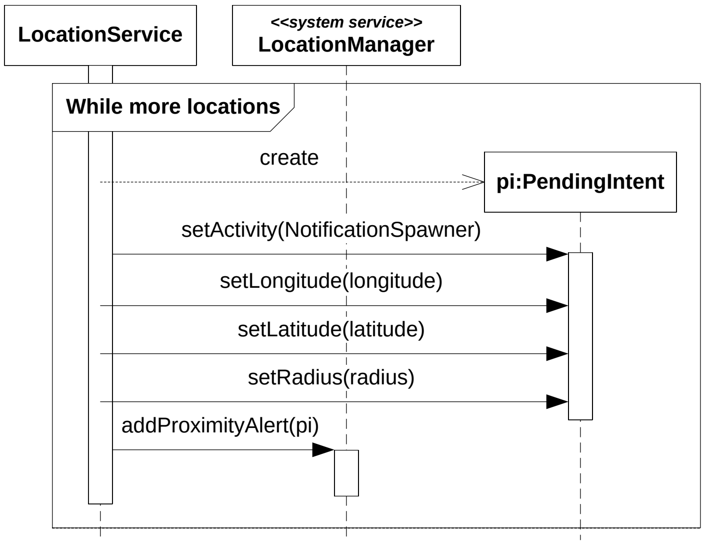


Figure 4.6: A sequence diagram showing how the service **LocationService** adds location reminders.

Figure 4.7 shows how the system service **LocationManager** reads the position off the phones hardware. If it is near the location that the pending intent was registered with, it sends out the broadcast associated with a pending intent. This would send out a proximity alert event that is picked up by the **NotificationSpawner**. The **NotificationSpawner** in turn shows a notification by registering a pending intent with the system service **NotificationManager** (not shown on the figure). This will make the system open the application when the user acknowledges the location reminder.

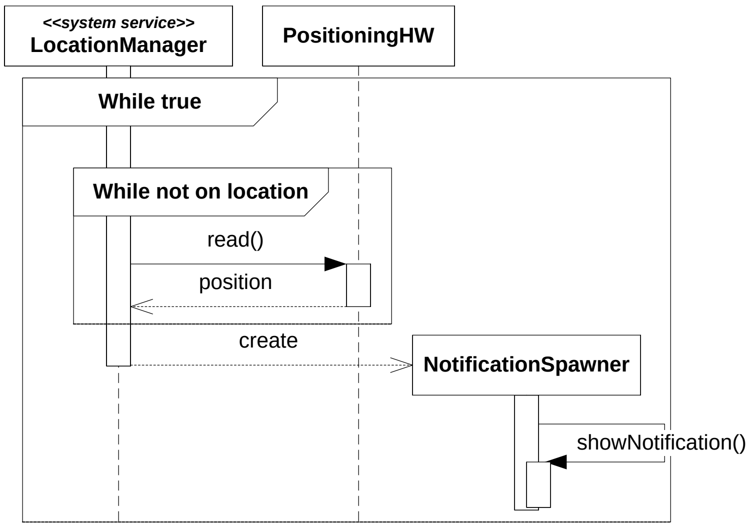


Figure 4.7: A sequence diagram showing how the system service **Location-Manager** handles changes to the location.

Chapter 5

Implementation

This section describes some of the implementation details concerning the implemented prototype application. More than 32 classes and 3500 lines of source code were implemented, so it is not possible to describe all of them within the scope of this thesis. I have decided to describe only those features that were most difficult to implement and that exposed bugs in the Android platform. The rest of the implementation is best understood by loading the source code in the Eclipse editor[1] that is used in the Android SDK, and running the application with the debugger enabled.

All of the primary use cases described in Section 2.4 were implemented in the prototype application. The acceptance test in Section 6.1 confirms that these use cases were implemented.

5.1 The Shopping List Adapter

The user interface mock up sketch on Figure 3.4 shows a list separated by categories. There was no titled list widget in the Android platform, so I chose to use the closest thing, an expandable list, because it can separate items by categories. This means if it was decided to later implement a flat titled list, one would be able to use the same view adapter.

The class **SLExpandableListAdapter** adapts the data from the database so that it can be used by the expandable list. It uses one database cursor to

[1]See Appendix D

retrieve all the categories and another for each category to fetch the associated items. It fetches the cursor containing all the items of a category, when a category is expanded. Listing 5.1 shows an excerpt of how the items are retrieved from the database when a category is expanded.

```
public Object getChild(int groupPosition, int childPosition)
{
    long categoryId = getGroupId(groupPosition);
    Cursor c = mSLIDb.fetchItems(mShoppingListId, categoryId);
    if (c.getCount() == 0)
        return null;

    c.moveToPosition(childPosition);

    return c;
}
```

Listing 5.1: An excerpt of the code that implements how items are retrieve for the shopping list.

During the implementation of the shopping list adapter I found out that the existing classes used for expandable lists[2] were inflexible. One was unable to control which widgets were displayed in each group. That meant that it was impossible to put check boxes with items under each category. I extended a base class called **BaseExpandableListAdapter** to change the displayed widgets. This undocumented so I had to look into the source code of the Android platform in order to find out how to extend this class.

The implementation of the expandable list adapter exposed a bug in the Android platform. I found out that if a group (category) was expanded and the shopping list switched to one with fewer categories, the application would crash. I found that it was because the expandable list in the Android platform did not forget which group had been expanded, eventhough it was notified that the underlying data had changed. When it tried to expand all the previously expanded groups, it would try to expand a group that did not exist. This bug was found and a solution created by studying the Android platform's source code. The created fix was simple, one needed to return "'-1"' if the call to **getGroupId** was out of bounds, but this undocumented. The code and the fix can be seen in listing 5.2.

[2]The view adapter **SimpleCursorTreeAdapter** in [5] provides an easy way to use expandable lists.

The bug has not been reported yet because of time constraints. It is unclear if the expandable list widget should completely forget which groups were expanded when the underlying data changes. It is good that the list keeps being expanded if there are minor changes so the user does not have to re-expand the groups. It, however, becomes problematic when the underlying data changes completely because it then tries to expand something that does not exist.

The Android developers should decide if this behavior is really wanted and make sure to document it. If they decide to keep the behavior they should make a notice in the documentation that the method **getGroupId** should return "‘-1"’. They could alternatively make the expandable list widget robust so that it checks that a group exists before trying to expand it.

```
public long getGroupId(int groupPosition)
{
    if (getGroupCount() > groupPosition)
    {
        mGroupCursor.moveToPosition(groupPosition);
        return mGroupCursor.getLong(mGroupIdIndex);
    }
    else
    {
        return −1;
    }
}
```

Listing 5.2: A fix for the expandable list bug. Code comments have been removed.

5.2 Item Autocompletion

The autocompletion described in Section 3.2.1 was implemented. The application comes pre-loaded with some common items and categories. These are inserted in the database by the class **DatabaseHelper** when the application is first installed.

The autocompletion is performed by a standard **AutoCompleteTextView** widget that uses a custom view adapter called **ItemAutoCompleteAdapter**. The view adapter instantiates views for the **AutoCompleteTextView** widget and binds the data from the database. The item names are fetched from the database using the database adapter **ItemDbAdapter**. Listing 5.3 shows how the item names are retrieved from the database. The called method, **fetchItemNamesByGlob**, returns a list of items where the item names match the first few characters the user has typed in the **AutoCompleteTextView** widget.

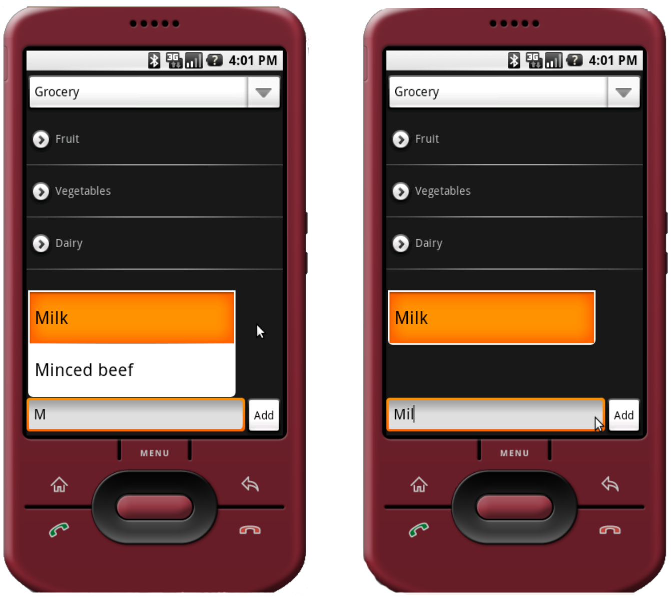
```
@Override
public Cursor runQueryOnBackgroundThread(CharSequence constraint) {
    if (constraint == null)
    {
        Log.d(TAG, "runQueryOnBackgroundThread constraint is null!");
        return mCursor;
    }

    return mItemDb.fetchItemNamesByGlob(constraint.toString());
}
```

Listing 5.3: The item names are retrieved by the **ItemAutoCompleteAdapter** in the background.

The implementation of the autocompletion feature exposed another bug in the Android platform. Figure 5.1a shows the autocompletion feature, where the user has typed the letter "'M"'. The user is given a list of items starting with the letter "'M"' from which to choose. This looks fine until the user then types in another letter. The choices that no longer matche the typed letters are removed, but the list now floats in mid air instead of being flush with the text entry box as seen on figure 5.1b. I reported the bug to the Android developer team as "'*Issue 2296 in android: AutoCompleTextView "drop up" gap*"' [3]. The developers responded that the bug had already been fixed and was scheduled to be included in a future release.

[3]See the bug report at http://code.google.com/p/android/issues/detail?id=2296



(a) Small (b) Bigger

Figure 5.1: Normal autocompletion behavior, and the bug that appears after typing more letters. Notice how the autocompletion list hovers in mid air.

5.3 The Map View

Figure 3.8 on page 30 describes how the user can add a location reminder by choosing a location on a map. The default map widget in the Android platform was used to implement it. The default map view did not, however, have map pins that could be moved. This was implemented by doing custom hit testing on the map pin picture and then looking at the type of the received event. Figure 5.2 shows the implemented map view. The search feature was not implemented because of time constraints, but was kept in the user interface design, in order to give an impression of how it would have worked. The OK and Cancel buttons at the bottom of the screen were also not implemented. There was a bug in the Android platform that made any widgets placed under the map view disappear. The bug was trivial to reproduce, but was not reported because of time constraints.

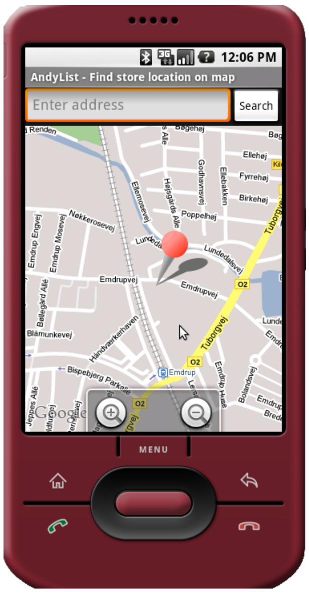


Figure 5.2: Map view with the custom map pin.

Figure 5.3 shows an activity diagram describing how the dragging of map pins was implemented. A custom **ItemizedOverlay** named **MapMarkerOverlay** was implemented. It is placed on the map view and receives events. It goes to a dragging state when it receives a down event, that is when the user touches the screen, and if the user has hit the map pin. When the **MapMarkerOverlay** is in the dragging state it will move the map pin picture to the new location if it receives a move event.

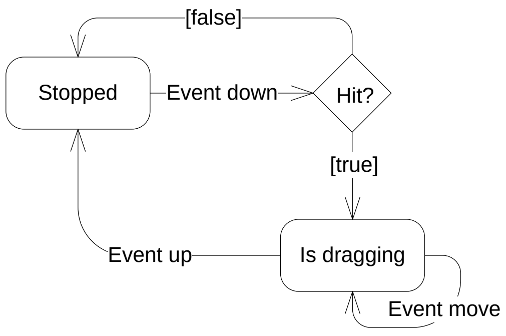


Figure 5.3: The activity diagram describes how the dragging of map pins was implemented.

I was surprised that Google did not include draggable map pins. There are no draggable pins in the default map application that comes with the Android platform, so I guess that was the reason the feature not being present. Extracting the feature from this application and including it in the Android platform would allow developers to create more kinds of applications that use the map view without having to re-implement the feature.

Chapter 6

Testing

6.1 Acceptance Test Results and Conclusion

This section describes the results from the acceptance test, which can be found in Appendix B. The acceptance test is used to verify that the system functions as expected. It provides step-by-step instructions on what a user is supposed to do and what the result should be, allowing a third party to replicate the results. A manual acceptance test was chosen because there were many interface changes during the development, and the development cycle was so short it would have been costly to produce an automatic acceptance test. One good side effect of this is that it describes how to use the system, so although it is not a user manual, one can easily deduce how to use the system by going through it. This would make writing a user manual easier.

It is important to note that this acceptance test does not test for various error cases. It only tests that it is possible to complete the implemented use cases. A finished product should also test for various error cases. It should also be subjected to a period of real-life usage by testers in order to uncover and weed out the worst bugs.

Table 6.1 contains a list of tests done in the acceptance test. Each row in the table states which use case was tested and whether the test passed or not. A detailed description of each step performed in the test can be found in Appendix B. It can be concluded that it is possible to perform all the use cases that were implemented. However, it does not mean that the system is bug free or that the system is feature complete.

Table 6.1: Short descriptions and results of the acceptance test.

Use Case	**Passed**
Add Item	✓
Add Unknown Item 1	✓
Tick Off Item	✓
Clear Ticked Items	✓
View Shopping List	✓
View Location Reminders	✓
Add Location Reminder	✓
Display Location Reminder	✓
Acknowledge Location Reminder	✓

6.2 Location Reminder Field Testing

This section contains a field test of the location based shopping list. It was already shown by the acceptance test that it was able to give location reminders using the emulator. However, would be preferable to field test it using real hardware because hardware does not always behave optimally.

The G1 phone is able to use three kinds of positioning: Cell-ID, WiFi and GPS[1]. The Android platform automatically uses both Cell-ID and WiFi if you choose a coarse positioning resolution. Carr [3] gives a good introduction to these technologies, on which the following explanation is based.

Cell-ID positioning is a method where the cell phone uses the information from the tower to which it is connected. It looks up the tower's position using its Cell-ID and learns that the cell phone is close to that tower. The precision is low, about 100-5000 meters, but the method is fast and does not drain power.

WiFi positioning works like Cell-ID positioning, but uses WiFi access points (AP) instead. The range of the APs are low, which means that you have to be close to see it. Therefor, the precision is better and Carr rates it to be 100-200 meters or less. However, it requires that there be a database of where the AP

[1]http://developer.android.com/reference/android/location/LocationManager.html

locations. It also requires a high density of APs in order to cover an area. This means that it is less reliable than Cell-ID or GPS.

GPS works by receiving signals from satellites orbiting earth and triangulating the position. This method is precise, it is typically 5-20 meters or less depending on the equipment. However, this requires a view of the sky to obtain signals.

The location reminders were tested in two different locations using both the coarse (Cell-ID and WiFi) and the fine (GPS) location methods. I tested the location reminder functionality with the following two stores:

- Døgnnetto Emdrupvej 107, København Nv 2400.
- Netto Lygten Lygten 53, København Nv 2400.

My home is about 200 meters from Døgnnetto, meaning it is close to the limit of precision of the coarse location method. Netto at Lygten is at least 1.5 km away from my home, making it more likely to work with the coarse location method. I tested the location reminders by placing the phone in my pocket when going to the stores. I traveled by foot when going to Døgnnetto and on my bicycle when going to Netto. Both methods of positioning were repeated three times and the results were noted in Table A.1 and Table A.2 in Appendix A.2.

	GPS	**Cell-ID**
Successes	6	0
Failures	0	6

Table 6.2: Condensed results from the location reminder field test.

Too few sample point were gathered to draw any statistically significant conclusion about the reliability of the location reminders. We can, however, see a clear pattern emerge in Table 6.2, which contains a condensed version of the results. The Cell-ID positioning was unreliable. It was unable to display a reminder before entering a store on any occasion, nor did it display any reminder inside the store. I did receive reminders after leaving the store, but this was too late to be of any use.

The GPS performed better and sent a reminder every time it was used. It worked well, but Google recommends to turn it off to conserve power. The GPS should be turned off, because HTC G1 battery did not even last a day with it turned on.

The test also showed that the user can receive multiple reminders. I received a new reminder when the phone lost its GPS signal and found a new one, for example when leaving the store. This was annoying and should be corrected in the final product. The functionality for suppressing multiple reminders was implemented. However, activating the feature caused more bugs so it was chosen not to activate it in the prototype.

It can be concluded that location reminders via GPS would have been viable were it not for the problem with the battery life.

6.3 Usability Test

This section describes the conducted usability test. Jones et al. [8] outlines some of the methods that should be used in designing usability tests [2]. The first thing is to decide what needs to be measured, and then followed a method chosen.

So which metrics could be measured? We could compare the speed of using application compared to using a paper based shopping list. This test would, however, be difficult to design, because we would not know which factors to include. For example, should the time to find a pen and a piece of paper be included in the time to use a paper based shopping list? Can we even decide how much time it takes to find a pen and a piece of paper? The conclusion would vary greatly depending on the estimate of this single factor. The input speed would also vary greatly depending on the users' experience with the G1's keyboard, which could skew the result greatly. My own informal tests clocked the time to write a shopping list with a title and the same items as in Task 1 in Appendix C.1 to 18 seconds. This, however, does not include the time to find a pen and paper.

Another possible metric is to measure the number of inputs, e.g. clicks and key presses, but this kind of testing is best suited for comparing two different solutions.

Although useful, these quantitative tests are not the most realistic for use in this situation. Jones et al. [8] describes an interview based qualitative test[3]. The purpose of such a test is to identify things that might confuse the user. Are buttons labeled correctly? Is the navigation logical? These kinds of questions cannot be definitively answered, but one can get a good idea about it by observing test persons using the prototype.

I will now give a short description of the interview test. The interviewer sits down with a test subject, the prototype, and a list of tasks to complete. Each

[2]See Chapter 7 in [8].

[3]See Section 7.5 in [8]

task should represent some use case that the test person should try to complete. The test subject must "think-aloud" or explain what they are seeing on the screen, and how they think they are supposed to complete the given task. The interviewer records whenever he observes that the test subject has problems or misunderstood something. The interviewer may ask what the subject is doing, but should be careful not to bias the test with his questions. The interviewer must ask the test person what they found difficult or confusing after the completion of each task. The tasks can be seen in Appendix C.1. The time the subject took to complete each task were recorded beginning from the moment he had finishing reading the task description.

The interview tests should normally be recorded by a video camera or an audio recorder, so it is possible to refer back for further analysis. There were not enough time to set up equipment or go back and do further analysis so this was skipped in the testing process.

One can end up with useless results because the user is confused by the interaction style of the smart phone. They then find the prototype they are testing confusing, not because there is something wrong with the prototype itself, but because they simply find the smart phone platform itself confusing. The G1 smart phone is different from conventional mobile phones. It has a touch screen and a slide out keyboard. It also has new idioms, such as "'long pressing"'. and item on the screen brings up new options to choose from. To familiarize the subject with the phone, he should be shown the default applications that come with the smart phone.

Three subjects were asked to perform the given tasks and the results were recorded in Appendix C.2. The results showed that the time to write a shopping list varied greatly from test person to test person. The slowest used 1 minute and 36 seconds, while the fastest test person used only 22 seconds. The slowest subject took his time and made many comments during the tasks. The fastest subject used almost the same time as I did writing a paper based shopping list. It would have been interesting to see how fast they could have completed the task if they had been told to hurry. For comparisonm, it would also have been interesting to see how much time the subjects would have used writing a paper based shopping list, but this was an afterthought.

The subject seemed to generally like the user interface and had no problems completing the tasks, except for trying to return from the map view in the last task. They were all looking for an OK button, but could not find one. The OK button had been left out because of a bug in the Android platform, which made widgets under a map view disappear. I tried using the back button instead, but this was not well received. Therefor, the OK button should be re-introduced in the map view, as soon as Google fixes the bug.

All of the subjects also had trouble hitting the tick boxes. They did manage to tick the items, but they had to touch some of the items more than once. The tick boxes were at the edge of the screen, where the screen is less sensitive, so the touches were not picked up. They did not know that they could have hit the whole row, instead of just the tick box itself. It is normal on the Android platform to hit a whole row, so they probably would not have had the problem if I had showed them this before the test. Another solution could be to simply move the tick boxes further away from the sceen edge so the G1 can more easily register the touches.

All of the test persons found a few labels where they thought the text should be changed for clarity, but they had no problems understanding the text.

The interview test showed that the application was efficient and user friendly except for the problems encountered with the map view. All of the subjects quickly learned how to complete each task on their own, as Jerôme Baltzersen put it:

> *"I did not have to spend much time figuring out the interface as it seemed very intuitive."*

The time to write a shopping list varied too greatly between the subjects to determine if it is faster than a paper based shopping list in the long run. The best way to meassure this would be to equip a larger number of people with the application and the phone and see if they would continue to prefer using the application over the paper shopping list.

6.4 Unit Tests

A series of unit tests were implemented in order to test the data layer. They were implemented so I could be confident that the data layer functioned as expected. The tests can be run with the **adb** tool that comes with the Android SDK. The application must first be installed in either the simulator or on the phone[4]. All of the tests can then be run by executing the following command on a command line:

```
$ adb shell am instrument -w dk.mcdonnell.andylist/ ↩
    android.test.InstrumentationTestRunner
```

This runs all the implemented unit tests. There are five unit tests in total and each of them contains five to seven test cases. It takes time to run all unit tests, so they can also be run separately by executing the following command:

```
$ adb shell am instrument -e class dk.mcdonnell. ↩
    andylist.tests.TheNameOfTheTest -w dk.mcdonnell. ↩
    andylist/android.test.InstrumentationTestRunner
```

Where `TheNameOfTheTest` must be replaced with one of the unit test names from Table 6.3.

Unit test name
CategoryDbAdapterTest
ItemDbAdapterTest
LocationDbAdapterTest
ShoppingListDbAdapterTest
ShoppingListItemDbAdapterTest

Table 6.3: The implemented unit tests that each contains about five to seven test cases.

All of the unit tests ran successfully, but they also only tested expected behavior. They should be extended to also test for error cases.

[4]See Appendix D on how to install the application.

Chapter 7

Further Work

This section further explores possibilities for work improvements of the location based shopping list. Three main areas of possible improvement are identified:

- Finish the prototype.
- Implement competing user interface designs.
- Additional interesting features.

The current prototype application does not include all the functionality a shopping list for a smart phone should contain. It lacks such features as adding, sorting, deleting and renaming of categories. These features are relatively easy to implement because much of the underlying functionality has already been implemented, as evidenced in the unit tests. Another important feature that was not implemented was the ability to limit the number of reminders received after entering a store. Field testing showed that it is possible to receive multiple repeat location reminders on one trip. This would be annoying. The functionality of this feature was implemented, but it was not enabled. Enabling it would have required changes to other parts of the application, that would have made the rest break[1]. More time is necessary to find solutions to he quirks of the Android platfrom in order to enable this feature.

The prototype came with a limited number of items and categories already known by the application. More items and categories available from the start would be useful.

There was not enough time to implement more than one prototype. So no tests were conducted to determine which user interface design was best. It would be ideal to implement all of the user interface designs and compare them in usability tests. It should be possible to re-use most of the implemented source code to implement prototypes with other user interfaces, as the code was separated into distinct layers.

With more time and resources, many additionally features could be implemented, and the application could be generally improved. The application could be tied to an online service, enabling synchronization of shopping lists between multiple devices, e.g. between partners. If this were tied to an online Google service, Google would know what you wanted to buy and in which area you shop. This kind of information would be valuable for potential advertisers, as they would be able to direct special offers for what users want to buy. This is much more effective than trying to show several different kinds of ads that the user may or may not be interested in.

[1]It would have required that the **NotificationSpawner** was made an activity instead of a **BroadcastReceiver**, but that made it steal the top level graphics.

It would also be possible to implement a delivery service by having the user enter what he wants to buy, the quantity and where to buy it.

Though the thesis was successful in reaching its goals, there is still a lot left to be explored.

Chapter 8

Conclusion

The implementation of the prototype application took longer than first expected. I managed to find three bugs in the Android platform, which halted the development on more than one occasion. It was apparent that the Android platform is not yet mature or fully tested. Four versions of the Android platform were released after the idea for the thesis was hatched.

The primary use cases for shopping list for a smart phone were identified and implemented in a prototype application, and found to be functional based on the results of the acceptance test. The secondary use cases were also identified, but not implemented. Most of the underlying functionality was implemented, but not integrated into the user interface.

Several user interface designs were described and evaluated according to a set of criteria. The design with the highest overall score was chosen for the implementation. The rating system for the competing designs was not entirely objective. A more objective usability test may have been more useful.

A usability test was conducted to see if the implemented application was better than a paper based shopping list, but was inconclusively. It did however show that the implemented prototype was efficient and user friendly. The test subjects did not uncover any major usability problems, but the sample size was small limiting the power of the results. The users generally enjoyed using the application, but a long-running real-world test with a larger sample size would be needed to determine the applications long term use potential.

Field testing showed that location reminders did not work reliably with Cell-ID positioning, but were successful using GPS. However the GPS used so much power that the phone became unusable.

The thesis showed that it is possible to create applications that use location reminders. They will, however, in the case of the HTC G1 smart phone, drain the battery too quickly. The Android platform is new and the HTC G1 was the only Android phone on the market at the time the thesis was written. Several new Android phones have been scheduled for release in 2009. It would be interesting to see if location reminders will be feasible on those devices.

Appendix A

Field Testing

A.1 Preliminary Field Test

This field test was done using the **Tag ToDo** program for the HTC G1. It was done at the start of the project, long before a prototype was ready. The goal was to try and identify any problems with a shopping list for a phone. The **Tag ToDo** program is a lot like a shopping list. It has multiple lists. You add things you need done to the list and tick them off as you complete them. This is very similar to a shopping list where you add things you need to buy to a list and tick them off as you buy them.

I wrote a shopping list in the **Tag ToDo** program with the following items:

- Shake-a-cake
- Chips
- Olives
- Red peppers
- Oregano
- Minced beef
- Shaving cream
- Toothpaste

- Cheese for pizza
- Chorizo
- Yeast
- Flour

The list was so long that it scrolled off the screen. The list was in the above order, which turned out to be a problem. I ended up walking a lot between the different sections, because I did not get everything in the section I was in. I could not keep track of all the items I needed in that section because I had to scroll to find them.

This early field test exposed a need for sorting the items by the section they are in, in order to avoid wasting time walking between the sections.

A.2 Location Reminder Field Testing Results

This section contains the results from the location reminder field test described in section 6.2.

Location	Døgnnetto Emdrupvej 107, København Nv 2400
Coarse 1	No reminder.
Coarse 2	Got a reminder half way home.
Coarse 3	No reminder.
Fine 1	Got a reminder a few meters outside the store and got multiple reminders while walking near the window in the store. Got a reminder when leaving the store.
Fine 2	Got a reminder a few meters outside the store. Did not go into the store.
Fine 3	Got a reminder 25 meters outside the store. Got another reminder when leaving the store.

Table A.1: Test results for Døgnnetto when using coarse and fine location methods.

Location	Netto Lygten 53, København Nv 2400
Coarse 1	Got a reminder reminder upon leaving the store and crossing the street.
Coarse 2	No reminder.
Coarse 3	No reminder.
Fine 1	Got a reminder on outside the store. Did not enter the store.
Fine 2	Got a reminder about 10 meters away from the store. Did not enter the store.
Fine 3	Got a reminder about 10 meters away from the store. Got a reminder inside the store when standing in front of the register. Also got a reminder after exiting the store.

Table A.2: Test results for Netto when using coarse and fine location methods.

Appendix B

Acceptance Test

This chapter contains a detailed step-by-step description of how to test the program. The results of the test is discussed in Section 6.1. All the test where run in the simulator that was shipped with the Android Platform. See Appendix ??? on how to set up the system for testing. All tests assume that the default test data has been loaded[1], and that none of it has been modified.

Test 1: Add item

In this test we will enter an item that is already known by the system. The item Pears has already been added to the system, and should be suggested when the user starts to type it. The system should also automatically add the item under the correct category on the shopping list.

1. Type the character **p** on the keyboard.
2. The system displays a pop-up list.
3. Choose **Pears** from the pop-up list.
4. Press enter twice.
5. The system adds **Pears** to the shopping list under **Fruit**.
6. Touch **Fruit** on the shopping list to expand it.

[1]The default source code loads the test data by default.

The item **Pears** should now be appear under the category **Fruit** on the shopping list.

Test 2: Add unknown item 1

In this test we will enter an item that is not known by the system, i.e. one that has not been entered before, or does not come as part of the standard items. The system should ask which category the unknown item belongs to and add it to the shopping list under the correct category.

1. Enter **ham** on the keyboard.
2. Press enter twice.
3. The system displays a dialog.
4. Touch the spinner in the dialog.
5. The system displays a list of categories.
6. Choose **Meat**.
7. Touch the OK button.
8. The system adds the category **Meat** to the shopping list.
9. Touch **Meat** on the shopping list to expand it.

The item **Ham** should be under the **Meat** category. The system should now be able to suggest **Ham**, if it is typed in again. Test this by:

10. Enter the character **h** on the keyboard
11. They system displays a pop-up list containing **Ham**.

Test 3: Tick off item

We want to make sure that the system remembers which items that got ticked off.

1. Touch the category Fruit to expand it.
2. Tick off Apples under Fruit.
3. Touch the category Vegetables to expand it.
4. Tick off Potatoes under Vegetables.

5. Press the back button on the phone.
6. The phone closes the application.
7. Touch and hold the application drawer on the bottom of the screen.
8. Touch the AndyList icon to start the program.
9. Touch the category Fruit to expand it.
10. Touch the category Vegetables to expand it.

The items Apples and Potatoes should both still be ticked off

Test 4: Clear ticked items

We want to remove all the ticked items on the list. This test should be performed with a random number of items ticked, to make sure that it functions as expected.

1. Tick off a few items by touching them.
2. Press the menu key on the phone.
3. The system displays a menu.
4. Touch the menu item Delete Ticked.

All the ticked off items should disappear from the shopping list. Categories with no items left should also disappear.

Test 5: View shopping list

We will change the shopping list that is currently shown from the default to another shopping list in the system. The system displays the Grocery shopping list per default.

1. Touch the spinner at the top of the screen.
2. The system displays a list of shopping lists.
3. Choose **Butcher** from the list of shopping lists.
4. The system displays the **Butcher** shopping list.

The **Butcher** shopping list should contain one item(**Minced meat**) with the default test data.

Test 6: View location reminders

We want to see which location reminders are present in the system, and which shopping lists they are associated with.

1. Press the menu button on the phone.
2. The system displays a menu.
3. Touch the menu item **Location Reminders**.
4. The system displays a list of locations reminders.

The displayed list should look like the shopping list, where each location is sorted by which shopping list it is associated with.

Test 7: Add location reminder

We want to add a location reminder to a shopping list.

1. Press the menu button on the phone.
2. The system displays a menu.
3. Touch the menu item **Location Reminders**.
4. The system displays a list of locations reminders.
5. Touch the button **Add store location**.
6. The system displays a form.
7. Enter **My butcher in Copenhagen** in the **Store, location** field.
8. Touch the spinner.
9. The system displays a list of shopping lists.
10. Choose the shopping list **Butcher**.
11. Touch the button **Set store location**.
12. The system displays a map with a pin.
13. Drag the map pin to where the streets **Frederiksborggade** and **Kultorvet** in Copenhagen cross each other.

14. Press the back button on the phone.
15. Touch the OK button.

The new location reminder should now appear under the shopping list **Butcher**. We have however not tested whether it is working as expected. This requires the use the built-in debugging tools in Eclipse.

16. Press back on the phone twice to stop the application.
17. Open the **Emulator Control** view in Eclipse.
18. Scroll down to **Location Controls**.
19. Choose the tab **Manual** and choose **Decimal**
20. Enter **12.573536** in the field **Longitude**.
21. Enter **55.682742** in the field **Latitude**.
22. Press the **Send** button.

The system should now show a notification in the notification bar on the phone. Acknowledging the reminder should open up the shopping list **Butcher** which did not previously have a reminder associated with it.

Test 8: Get a location reminder

We want to test that it is possible to receive a location reminder, once you arrive at a store. The chosen location was my local super market Døgnnetto at Emdrupvej, Copenhagen, Denmark.

1. Press back on the phone twice to stop the application.
2. Open the **Emulator Control** view in Eclipse.
3. Scroll down to **Location Controls**.
4. Choose the tab **Manual** and choose **Decimal**
5. Enter **12.541601** in the field **Longitude**.
6. Enter **55.722502** in the field **Latitude**.
7. Press the **Send** button.

The system should now show a notification in the notification bar on the phone. Acknowledging the reminder should open up the shopping list **Grocery**.

Test 9: Acknowledge location reminder

We want to acknowledge a location reminder that has appeared in the status bar[2].

1. Drag down the notification bar.
2. Touch the reminder (A little AndyList icon).
3. The system should open the shopping list associated with the reminder.

[2]See Test 8 on how to get a location reminder.

Appendix C

Interview Usability Test

C.1 Tasks for the Interview Usability Test

This section contains tasks for the usability test. Please read the following instructions for the usability test:

1. You have four minutes to complete each task.
2. Try to "think out loud" when you are performing the task. For example, try to explain what you think you have to do, e.g. which buttons you have to click.
3. Explain why something is confusing you if you get confused.
4. Please stop after completing each task, so that the supervisor will have time to write down your comments.

There are seven tasks in total, so the test should not take much more than 40 minutes.

Task 1: Add items to the shopping list

Add the following items to the grocery shopping list:

- Milk
- Apples

- Potatoes
- Cheese

Task 2: Add items to the shopping list (Advanced)

Add **Brocolli** to the grocery shopping list.

Task 3: Switch shopping list

Switch to the **Butcher** shopping list.

Task 4: Tick off items

You are out shopping and you have gotten the following items:

- Milk
- Apples
- Potatoes
- Cheese

Please tick them off your shopping list.

Task 5: Clear ticked off items

You have finished shopping and you want to remove all the bought items from the shopping list. Please remove all the bought items from the shopping list.

Task 6: View location reminders

Please find out at which stores you will receive location reminders for the grocery shopping list.

Task 7: Add a location reminder

Please add a location reminder for the **Butcher** shopping list. You can choose yourself where to place the location reminder on the map.

C.2 Results From the Interview Usability Test

This section contains the results from the interview usability test described in Section 6.3. Three males used the cell phone to perform a set of tasks described in Appendix C.1. The observations were written down in Table C.1, C.2 and C.3.

Table C.1: Results from the first interview.

Name:	Thomas Dixen Axel
Sex:	Male
Age:	25
Cell phone:	Nokia E51, E50, 6282
Task 1	1 min. 36 sec. to complete. No problems. Discovered the autocompletion feature after typing just one letter.
Task 2	58 sec. to complete. Was wondering if he had made a spelling error, because the item did not show up in the suggestions. Had no problem assigning the new item a category, but thought the label should say that the item was unknown.
Task 3	15 sec. to complete. He thought it would be nice if the list of shopping lists also said how many items you need to buy for each shopping list.
Task 4	33 sec. to complete. He had problems hitting the small tickbox. He did not know that he could hit the whole line instead. He wanted more response than the box just getting ticked, it could for example blink item text.
Task 5	1 min. 9 sec. to complete. Tried long pressing to delete an item. Tried to delete the whole list using the menu. Discovered the correct menu item and slapped himself on the head, because he meant it was pretty obvious, but had missed the menu item anyway.
Task 6	18 sec. to complete. He remembered which menu item to use from the previous task. He had no problem expanding the grocery list item to see which locations were associated with the grocery list.

Continued on next page

Continued from previous page

Task 7	3 min. 50 sec. He could not figure out how to move the map pin. He tried tapping the map and expected the pin to move there. He also tried to long press the map and expected the map pin to move there. He tried to use the search feature, but found out it was not implemented. He finally asked for a hint and was given the following "'Try moving the pin"', which immediately made him figure out what to do. He did not like the method of moving the map pin, because you could lose it if you scrolled it out of the view. He also did not like to hit the back button, which he said felt like canceling the operation. He also missed that there was not any response about that the location had been set when he returned from the map view. He found out that he needed to give the location a name when it would not let him press the OK button until he had entered a name. He did not like the labels, and thought that "'Set store location"' should say "'Find store location on map"' or similar. He also discovered that label "'Add store location"' should have said "'Add store location reminder"'.

Table C.2: Results from the second interview.

Name:	Kasper Lindberg
Sex:	Male
Age:	23
Cell phone:	Nokia N6131
Task 1	33 sec. to complete. Immediately discovered the auto-completion feature. No problems or comments.
Task 2	25 sec. to complete. No problem figuring out how to enter the category, but thought the text should be changed to reflect that the item was unknown by the system.
Task 3	9 sec. to complete. No problems. He said it was "*Very easy to figure out*".

Continued on next page

Continued from previous page

Task 4	35 sec. to complete. Got annoyed by having to expand each category. Tried to touch tickbox, and the system had trouble recognizing the touch. He did not try to touch the whole item line. He suggested that the tickboxes should be moved further from the edge of the screen. Also wanted to be able to get an alphabetical listing of the items, although he liked that the items were sorted after where he would have been in the store.
Task 5	36 sec. to complete. Tried to long press item and then tried to long press shopping list drop down. Then discovered that he had to use menu and was finally able to clear the list. He thought it was strange that he could not long press the shopping list title in the drop down.
Task 6	20 sec. No problems or comments.
Task 7	4 min. and 30 sec. to complete. Filled in the name of the location first and then continued on to set the store location. He had no problems figuring out that he had to drag the map pin to the desired location, but he had problems placing it exactly where he wanted it, and did not figure out he could zoom in for more precise control. He was almost done after 30 seconds or so, but then got stuck when he wanted to go back. He desperately tried to use the search feature, but saw that it was not implemented. I finally had to give the hint "How do you normally go back on this device?" after the time was up. He then figured out to hit the back button and complete the task, but he said that "*It felt wrong to hit the back button because it felt like a cancel button.*". He also noticed that there was no indication that the location had been set, after using the back button. He said "*It should display the coordinates or the address.*".

Table C.3: Results from the third interview.

Name:	Jerôme Baltzersen
Sex:	Male

Continued on next page

Continued from previous page

Age:	23
Cell phone:	Nokia E65
Task 1	21 sec. to complete. Used autocompletion immediately. He said he *"I did not notice the add button at first, but discovered it as soon as I started typing."*.
Task 2	15 sec. to complete. He had no problems and said *"I assumed it did not know the item when I did not show an autocompletion, so it was logical that I had to specify the category."*.
Task 3	5 sec. to complete. He said *"It was obvious that the drop down list changed the shopping list because it already said grocery."*.
Task 4	21 sec. to complete. He had trouble hitting the tickboxes as he only tried to hit the box itself instead of the whole item line.
Task 5	10 sec. to complete. He went directly for the menu item, eventhough he had not seen it before. He said *"I used the process of elimination, and I could not see where else it would have been."*.
Task 6	9 sec. to complete. No problems and no comments.
Task 7	1 min. and 4 sec. to complete. Tried using the search feature but discovered it was not implemented. Then moved the map pin by dragging it. It took him a while to figure out how to go back and he said *"How do I confirm it?"*. He managed to figure out to use the back button with out any hint by simply trying different buttons. He also figured out that he had forgotten to give location a name when he could not click the disabled OK button.
Other comments	He said *"I did not have to spend much time figuring out the interface as it seemed very intuitive."*

Appendix D

Setting Up the Software

This sections describes how to get started using the prototype application. First install Eclipse and the Android SDK:

1. Install Eclipse by following the instructions at `http://www.eclipse.org/`
2. Install the Android SDK by following the instructions at `http://developer.android.com/sdk/1.5_r1/installing.html`

Then import the source code from the accompanying CD-ROM into Eclipse by doing the following:

3. Open Eclipse.
4. Select **File→Import** from the menu.
5. A new window with import options appears.
6. Choose **General→Existing Projects into Workspace**.
7. A new window appears.
8. Browse and choose the directory called **AndyList** on the CD-ROM.
9. Press **Finish**.

You can now start the emulator and the prototype application by right clicking the project called **AndyList** and selecting **Run As→Android Application**.

Bibliography

[1] OI Shopping List - OpenIntents. http://www.openintents.org/en/node/19, Accessed April 2009.

[2] Deborah Barreau and Bonnie A. Nardi. Finding and reminding: file organization from the desktop. *SIGCHI Bull.*, 27(3):39–43, 1995.

[3] Eric Carr. Location Technologies Primer. http://www.techcrunch.com/2008/06/04/location-technologies-primer/, Accessed April 2009.

[4] Teodor Filimon. Tag ToDo List - An app for the Android mobile platform. http://teodorfilimon.com/android/Tag-ToDo-List/index.html, Accessed April 2009.

[5] Google Inc. Android 1.1 SDK, Release 1. http://developer.android.com/index.html, 2009.

[6] Google Inc. Android 1.1 SDK, Release 1, Sample Code. http://developer.android.com/guide/samples/index.html, 2009.

[7] Google Inc. The Developer's Guide. http://developer.android.com/guide/index.html, 2009.

[8] Matt Jones and Gary Marsden. *Mobile Interaction Design.* John Wiley & Sons, February 2006.

[9] Craig Larman. *Applying UML and Patterns : An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition).* Prentice Hall PTR, October 2004.

[10] Thomas W. Malone. How do people organize their desks?: Implications for the design of office information systems. *ACM Trans. Inf. Syst.*, 1(1):99–112, 1983.

[11] Erica Naone. App Class: An MIT team wins $300,000 in Google's Android Developer Challenge . `http://www.technologyreview.com/article/21844/`, January/February 2008.

[12] Two forty four a.m. LLC. Locale. `http://www.localeandroid.com/index.html`, 2009.