Mrudul V. Pendharkar

Fluid Home Screen for Mobile Phones

Helsinki Metropolia University of Applied Sciences
Master of Engineering
Information Technology
Thesis
12th April 2012

Acknowledgements

The idea for the thesis germinated when I was quiet annoyed by my mobile phone's home screen forcing me to launch application for even a small piece of information. This thesis is the result of work carried out to improve my and in turn every other similar mobile phone's home screen over the last few months.

I would like to thank Dr. Mahbub Rahman, my thesis instructor at Helsinki Metropolia University of Applied Sciences for his technical support, guidance and comments. I would also like to thank senior lecturer Taru Sotavalta for patiently reviewing the language of thesis and thereby helping to improve the presentation of the thesis.

I would also like to thank my friends and my family without whose support and encouragement, this thesis would never have been a reality.

Tampere, April, 2012

Mrudul V. Pendharkar

Metropolia

Abstract

Author(s) Title Number of Pages Date	Mrudul V. Pendharkar Fluid Home Screen for mobile phones 46 pages + 2 appendices 12 April 2012
Degree	Master of Engineering
Degree Programme	Information Technology
Specialisation option	Mobile Programming
Instructor(s)	MD.Mahbubur Rahman, Dr.

It is no more uncommon to find mobile devices with a large number of applications downloaded and installed. However a limited display capability makes it difficult to display a large number of applications or icons. Therefore optimisation of the display area may be desirable to enhance the user experience. This thesis attempted to propose a unique solution, by providing a framework which would ease developer experience as well as improve user experience.

In order to find the solution, usability study of current as well as classical home screens was taken. This study helped to identify three main gaps in the current model of home screens; larger view area needed for the application pinned to the home screen, accommodating more application icons on the home screen and interaction inside view area of the application.

After some iterations of mock up, two potential solutions were deduced. Both solutions are able to accommodate more application icons, provide a larger display area and interact inside the display area. The only difference between the two solutions was that one provides the display area alongside other application icons, while the other provides a display area floating above other application icons.

The only consideration with these solutions is that of memory consumption, which can be reduced by splitting and loading application states instead of the whole application.

Keywords	home screen, Qt, QML, user experience, usability

Contents

Acknowledgements 4

1 Introduction 6

2 Usability study of Home screens 7

2.1 Phones with Menu Grid 8
2.2 Home Screen with Idle Menu 9
2.3 Home screen with Widgets 9
2.4 Home screen with Tiles 11
2.5 Home screen icons with Notifications 12
2.6 Learning from usability study of home screens 13

3 Fluid home screen solution for mobile phones 14

4 Modelling Fluid Home Screen 18

4.1 Hardware 18
4.2 Software 18
4.2.1 Symbian 18
4.2.2 Qt application development framework 19
4.3 Design Ideas 20
4.3.1 Filmstrip 20
4.3.2 Bubble Home screen 22
4.3.3 Fluid Home Screen Solution 24

5 Technical Details of Implementing Solution 26

5.1.1 Running applications on the Home screen 26
5.1.2 Qt Graphics View Framework 27
5.1.3 Resizing and Repositioning 29
5.1.4 Custom Positioner 31
5.1.5 Drawing two surfaces 31
5.1.6 Interaction on surfaces 32
5.1.7 Architecture 32
5.1.8 Class Diagram 34
5.1.9 Activity Diagram 36

6 Conclusion and discussion **42**

References **44**

1 Introduction

The modern communication era has brought a tremendous expansion of wireless and wired line networks. Computers, televisions, telephone networks, television networks are all experiencing an unprecedented expansion, fuelled by consumer demand. Telephone networks are seeing a fundamental shift in communication. Wireless and mobile networks are replacing wired connections resulting in mobility to the consumer.

Mobile devices, such as cellular telephones, have become lighter and faster. They are now capable of performing tasks which far exceed a traditional voice call. Mobile devices are becoming small, portable computing devices that are capable of running a variety of applications. The processing power, memory capacity and hard disk capacity make them comparable to desktop computers or laptops. They have the capability to support different applications and services ranging from engaging games to productivity software such as the Microsoft Office. Users are also installing more and more applications as a result of application store present on device. Thus one can find lots of applications installed on mobile devices these days. However limited display capability makes it difficult to display a large number of applications or icons. Therefore optimisation of the display area is desirable to enhance the usability of devices.

Various approaches have evolved over time. Very basic scrollable windows, application shortcuts, home screen with widgets, home screen with tiles are some of the approaches that have evolved. These approaches although improve user experience one way or the other, do not take into consideration one tacit user need: need to skim over an application before launching of the application. Home screen with widgets does solve some of these problems, but needs a developer to write a completely new application for home screen itself. Hence a newer approach to home screen is needed to address the problem of skimming over application without writing a new application just for home screen.

This thesis attempts to provide a solution to the problem stated above, by providing a framework which would ease developer experience as well as improve user experience.

2 Usability study of Home screens

Experience is an almost overwhelmingly rich concept. The construction of experiences as stories from moment-by-moment experience is not straightforward. For example, experiences tend to improve over time [1]. Thus, experience is neither about creating good industrial design nor about fancy interfaces. But, it is about experience through the product.

User experience (UX) is defined as the way a person feels about using a product, system or service. User experience highlights the experiential, affective, meaningful and valuable aspects of human-computer interaction and product ownership, but it also includes a person's perceptions of the practical aspects such as utility, ease of use and efficiency of the system. User experience is subjective in nature, because it is about an individual's feelings and thoughts about the system. User experience is dynamic, because it changes over time as the circumstances change. [2]

Usability is the ease of use and learnability of a human –made object. The object of use may be a software, application, website, book, tool, machine etc. [3].

Both Usability and User experience are clearly overlapping concepts which one needs to understand in order to build better and better products. Mobile devices come in various forms and factors. Thus it comes as no surprise that experience on device always is evolving. Whether it is adding a shortcut page or notifications area or status pane every aspect results in a more usable product than the previous one.

Sometimes looking at history teaches us to look at the problem in present. By unfolding brief history of device home screens, one can understand user experience of the device. This study will also reveal the usability issues associated with various existing device home screens.

2.1 Phones with Menu Grid

Mobile phones in the early days did not have many features. The processing power and memory capacity were poor. As a result a number of applications installed on the device were also less. So a simple menu grid was enough to navigate through the applications that were on the device. This menu grid used to contain application shortcuts and thus clicking on the shortcuts would result in the actual application being launched.

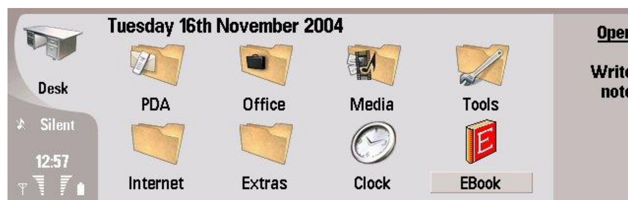

Figure 1. Menu grid on the phone.

Figure 1 illustrates a menu grid on the phone. Menu grids also had small scroll bars which could be used to navigate to next page. As a result more application icons could be accommodated than what would fit on the visible area of the display.

A typical usage of these phones is to open a menu, then launch the application. Once the application is launched, user interacts with the application. After the user is satisfied he or she closes the application. Also switching between applications on these phones takes place by closing the existing application, opening the menu and then launching a new application.

Thus access to program data needs launching of the complete application. Switching between applications needs a closure of the application. If the framework supports state saving, applications can save state just before the user interface gets closed. Due to the full screen launch, accidental launches of programs would make it difficult to switch between two applications.

2.2 Home Screen with Idle Menu

S60 2.0 phone from Nokia had a menu grid and an idle menu. One could configure an idle menu to set the favourite application shortcuts. The idle menu was also capable of displaying short alerts such as next appointment, WLAN networks found or not found.

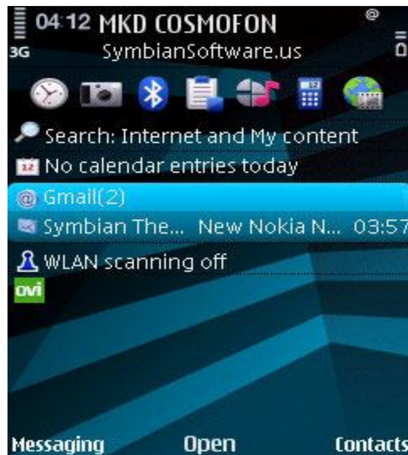

Figure 2. Home screen with Idle Menu.

Figure 2 illustrates a home screen with idle menu. This is the figure of typical S60 2.0 phone from Nokia. Thus the user using these phones has faster access to applications when compared to phones with a menu grid. Also the user gets regular notifications about updated information of the applications. Although this type of home screen is an improvement over previous home screens, still interaction on the application data or inside the application is not possible as the idle menu only contains application shortcuts. Also, application switching is inconvenient. Applications are launched in full version before one can see the data inside it. Thus one needs to close the application before he/she can switch to the next application.

2.3 Home screen with Widgets

S60 5.0 SDK provides concept called home screen in addition to the menu grid.

Figure 3. Home screen with Widgets and shortcuts.

Figure 3 illustrates typical home screens found two of the popular S60 5.0 device. The home screen contains not only shortcuts but also widgets. A software widget is a generic type of software application comprising of portable code intended for one or more different software platforms. The widget has a graphical user interface that displays data from the application. They may or may not update themselves. Widgets on the home screen have a fixed size predefined by the framework. On N8 size is around 2-3 lines while on the new Symbian devices (Belle) it is around 8 lines. One can have one or more home screens but N8 limits the home screen pages to three pages whereas Belle has approximately four pages.

Typically, for a user there exist two types of user interactions: user interaction when a shortcut for the application program is pinned up to the home screen and user interaction when a widget for the application program is pinned up to the home screen. A major difference between these two interactions is that, for a shortcut, the user first needs to launch application and then interact inside the application, while in the case of a widget, if possible, the user first interacts inside the widget and then if needed launches the application.

As a result user is able to get regular updated information due to updatable widgets. Also a mixture of widgets and icons results in more application shortcuts pinned to the home screen. However the readable visible area inside a widget which is pinned to the home screen is very small for large data. Also in newer devices which have managed to enlarge this visible area still that area cannot be allocated to all applications. This is because only few widgets can fit in the given area. Also, one needs to write the widget itself, in order to experience large area. Switching between applications from home screen needs to close application and launch newer one.

2.4 Home screen with Tiles

Microsoft introduced the concept of home screen with tiles on Widows Phone 7. Home screen was called "Start Screen" which is made up of "Live Tiles". Tiles are links to applications, features, functions, and individual items. Users can add, remove or rearrange tiles. [4]

Figure 4. Windows mobile home screen.

Figure 4, illustrates a Windows mobile phone with tiles on the home screen. Tiles are dynamic and update in real time for example, the tile for an email account would display the number of unread messages or a tile could display a live update of the weather .[4]

User typically looks at the updated information in the tile of the application. He or she then opens the application to get more details. Then he or she closes the application to check more information in the next tile. As a result user gets continuous update without explicit need to refresh. He or she also gets more information about the updated data due to large tile size. However, still the display area of the tile is small; hence readability of the tile is poor. In order to switch between applications on the device one needs to close the launched application, and due to the predefined size of the tile, one can have a very small number of tiles on the display area.

2.5 Home screen icons with Notifications

The iPhone home screen is a graphical list of available applications [5]. In addition to that, there is a dock at the base where one can hold most favoured applications. One can have around 7 home screen pages where application icons or web clips are pinned up. Every home screen page holds up to sixteen icons and dock can hold up to four icons. At the top of the icon one can see a notification number indicating updated information for that application.

Figure 5. iPhone Home screen

Figure 5, illustrates a typical home screen of an iPhone. One can observe the current home screen page and dock at the bottom in the figure. On an iPhone user flicks

through pages of icons, then launches the application and once user no more requires the application, he or she may close the application to move to next application. With icons only pinned to the home screen the iPhone home screen is very similar to the menu grid. One can pin a number of applications to the home screen. He or she is also able to get continuous information about new updates with the help of notifications at the top of the icons. This information is useful but still is very small and in order to know more about the updated data one needs to always launch the application itself. Also in order to switch between applications on the device one needs to close the launched application.

2.6 Learning from usability study of home screens

Every home screen choice has its own advantage and disadvantage. Tiles, Widgets are good to have features on home screen. Their larger display area facilitates user to get more information about the updated data. Interaction on the home screen as in Widgets helps user to view more data. If there is a large amount of updated information, he or she can interact on the home screen itself and have a quick glance over the updated information. An icon with a small area helps one to accommodate more application shortcuts on the home screen. A fixed tile or widget area reduces the number of application shortcuts on the home screen.

Thus important user experiences on home screen that one should address are -

a) Larger display window to enhance viewing area

b) Interaction to make it possible to view more data

c) Smaller home screen footprint.

Accommodating all these three points will make home screen better. The proposed solution in fourth and fifth chapter, attempts to present a solution which is able to accommodate all the three points.

3 Fluid home screen solution for mobile phones

A fully launched application represents a full view of the program. In order to accommodate lots of applications on home screen, framework defines minimum and maximum size for partial view of individual program. Tiles, widgets are some of the ways how; the framework defines the size of visible views. Multiple programs added to the screen represent plurality of the programs. Home screen and menu grid are some of the examples of plurality of the program. Any program with a partial view is referred to as *tile* henceforth in discussion.

The proposed solution provides an improved method of presenting a partial view of a full version of the program. The method also includes providing for display of an enlarged version of the first tile in response to a first input. The enlarged version of the first tile is larger than the first partial view. Both the first partial view and enlarged partial view may be a representation of full version of the program.

The enlarged partial view not only improves the view area for the tile but is also supposed to be interactive allowing user to flick or click inside the view. An enlarged partial view is launched by panning or clicking over the tile on the screen. One is able to launch the enlarged view either alongside other tiles or as a floating tile above the existing tiles.

In order to launch the enlarged view alongside other tiles, as a response to the first tile becoming enlarged, repositioning of at least one of the tiles is needed. Also reducing the size of at least one of the pluralities of the tiles is needed, so that all the tiles in the plurality of the program still fit into the same screen view port. This brings in double benefit of not only experiencing the enlarged view but also keeping other application shortcuts accessible.

As alternative to the enlarged view alongside other tiles, one can also create an enlarged view as a floating bubble on the top of home screen icons. In order to launch the enlarged view as a floating tile above the existing tiles, as a response to the first tile, a separate enlarged view is created and it floats above other tiles. This makes the

tiles below the enlarged tile area to be partially or completely hidden. However at the same time one can have a bigger enlarged view as compared to enlarged view alongside other tiles.

In the subsequent section, concept based on drawings is explained in order to comprehend the whole idea.

Drawing based explanation

Typically a phone would contain more applications than those visible on the screen. The boxes at the bottom indicate there are more pages of the home screen. The user would configure his or her home screen pages based on his or her preferences. So one might have a home screen configured in the form of categories, while someone else might configure it based on the frequency of the usage of a specific application.

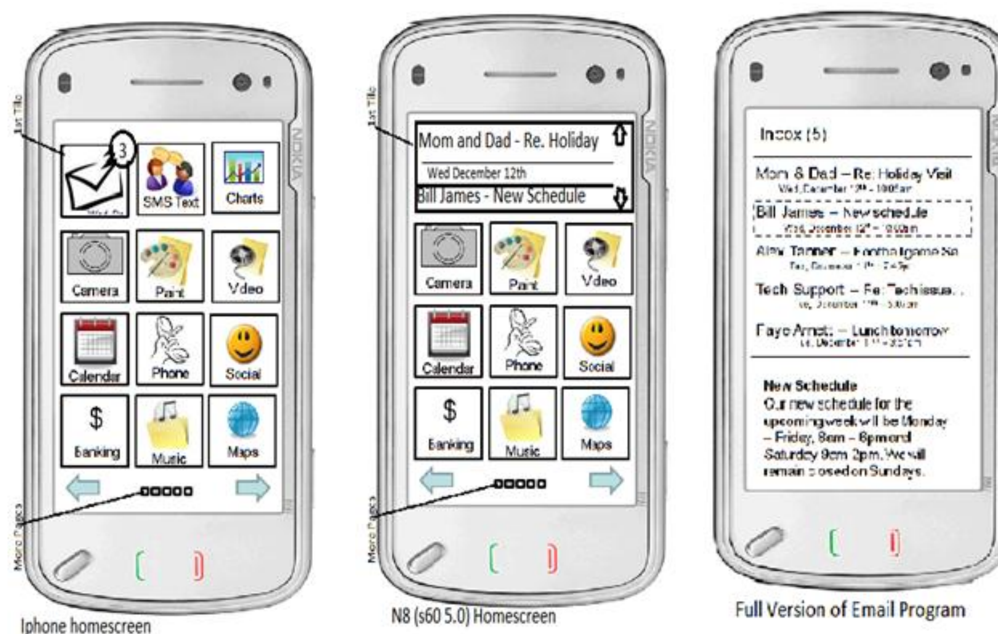

Figure 6. a)Depicting iPhone (b)N8 Home screen (c)Full version of Application

The first tile in all the three figures above (Figure 6) is that of email. Other tiles are representations of other application shortcuts. The home screen in figure 6 has nine visible tiles.

Figure 6a indicates a typical home screen of an iPhone. On an iPhone one can observe a notifications bubble hovering over the tile. This notifications bubble indicates how

many new messages have arrived. Similarly other shortcuts on the home screen will show notifications if needed to tell the user about new updates, missed calls, new SMS or missed appointments. On an iPhone one is able to pinup nine application shortcuts in the given example. Thus it is able to pinup maximum number of tiles possible in the visible view port.

Figure 6b indicates a home screen that can be seen on N8 or s60 5.0 devices. On the home screen one can configure a widget (if available) to display more data from the application. There can be a scroll widget which helps one to interact or navigate inside the widget. The maximum size of the widget is predefined for the home screen by the framework. As can be seen from the figure 6b only the basic subject information of the new mails is visible to the user. Then, using the scroll widget, which is at the extreme right corner of the widget, one can scroll through more information on updated data. One will notice that, it is not very convenient to scroll for more information, considering that the scroll widget is pretty small. At the same time one can configure shortcuts on the home screen. One can observe that by adding a widget to the home screen, numbers of shortcuts that can be pinned to the home screen are reduced to 6 + 1 widget. As the numbers of widgets grow, the number of shortcuts that can be pinned up will reduce.

Figure 6c shows full version of the program in the figure 6a. Here one can see a list of mails in the upper part of the window. At the bottom of the window, one can see current and future appointments. On clicking any of the mail or appointment, one is able to see details about of the mail or details about the appointment.

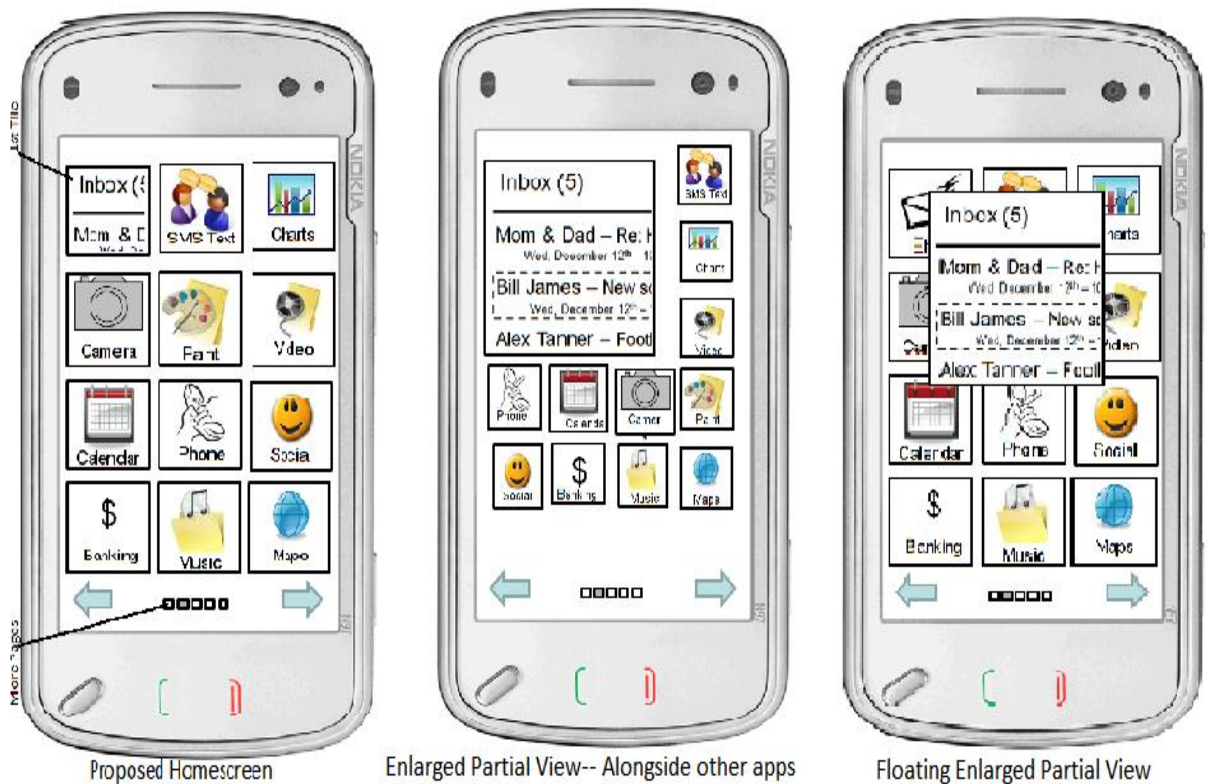

Figure 7. a) Proposed Home screen (b) Enlarged view embedded c) Enlarged view floating

Figure 7a, illustrates the proposed home screen. The proposed home screen looks very similar to the iPhone home screen. However on a closer look, one can observe that there is a partial view of the full version of the program. Thus in the same tile area, already there is a partial view to the actual data.

Figure 7b shows embedded enlarged partial view. One can also notice that other icons and tiles are still visible. They have also been repositioned and resized. This makes it possible to have an enlarged partial view and other tiles, alongside with each other on the same page. One is also able to interact inside the enlarged partial view.

Figure 7c shows a bubble of enlarged partial view that floats on top of other tiles on the home screen. The bubble is positioned in such a way that the bubble area should cover the touch point which enabled launching of the bubble. The bubble like the repositioned enlarged partial view is interactive.

4 Modelling Fluid Home Screen

Any system designed for people should be easy to use, easy to learn, easy to remember, and helpful to users. John Gould and Clayton Lewis (1985) recommend that designers striving for usability follow these three design principles, namely early focus on users and tasks, empirical measurement and iterative design

Iterative design is a methodology based on a cyclic process of prototyping, testing, analysing and refining a product or process. Based on the results of testing the most recent iteration of a design, changes and refinements are made. This process is intended to ultimately improve the quality and functionality of a design. [7]

This chapter hence forth will explain the hardware used for study and choices of technology made to implement the software. It will also show how iterative design was used to arrive at the proposed solution.

4.1 Hardware

N8 was chosen as a device for observing user behaviour. The Nokia N8-00 is a powerful combination of Internet, video, photos, music, and maps with the Symbian Anna operating system [8]. It has a large capacitive touch screen for pinch to zoom and other interactions on the device. N8 has a screen resolution of 640px*480px. It allows users to add application shortcuts and widgets. [8]

4.2 Software

4.2.1 Symbian

Symbian is a mobile operating system designed for smartphones. The Symbian platform is the successor to Symbian OS and Series 60 5th edition. The Symbian platform includes a user interface component based on Series 60 5th editions, while keeping the core technologies inherited from Symbian OS intact. The current platform version is Symbian^3. [9]

The native programming language to develop on Symbian is Symbian C++. It is non-standard C++ and learning curve is unfortunately steep [9]. On a Symbian^3 platform one can program with Qt, Web runtime, Flash or using java.

Widgets in computer programming are a reusable element of a graphical user interface that displays an information arrangement and provides standardized data manipulation [10]. On an N8 the size of widget is 312*82px. Widgets run on top of the web runtime. Widgets are lightweight applications that are based on standard Web technologies and are typically used to access information on the Web. They are similar to Apple's OS X Dashboard widgets, Google Gadgets, or Opera widgets in that they are built using Web standards such as XHTML, Cascading Style Sheets (CSS), and the JavaScript™ programming language. [11] From an end user perspective widgets appearance and functionality remains just like any other native application. Widgets provide an easy way to extend brand and provide high visibility. Home screen widgets allow users to dynamic data from the home screen itself without needing to launch the application in full screen. A home screen widget displays a non-interactive subset of data generated by full screen widget.

Thus apart from learning native application device technology, one also needs to learn a completely new technology, so that he or she can write a widget for home screen.

4.2.2 Qt application development framework

Qt is a cross platform application development framework. It is widely used to for developing application software with graphical user interface. [12]. The most code written in Qt should run unchanged across platforms. Qt uses standard C++. It also makes extensive use of a special code generator (called the Meta Object Compiler or moc) to enrich the language. The metaobject compiler, termed moc, is a tool that is run on the sources of a Qt program. It interprets certain macros from the C++ code as annotations, and uses them to generate additional C++ code with "Meta Information" about the classes used in the program. This meta information is used by Qt to provide programming features not available natively in C++: the signal/slot system, introspection and asynchronous function calls. [12] Original N8 devices were bundled with Symbian Qt 4.6.3. These devices were later updated with Symbian Anna patch.

Symbian Anna contains Qt 4.7 bundled in its platform. Qt 4.7 introduced Qt Declarative and Qt Quick framework.

Qt Quick is a framework that provides a declarative way of building custom, highly dynamic user interfaces with fluid transitions and effects, which are becoming more and more common especially in mobile devices. [13] Qt Declarative is a runtime interpreter that reads the Qt declarative user interface definition, QML data, and displays the UI that it describes. The QML syntax allows using JavaScript to provide the logic, and it is often used for this purpose. It is not the only way, however: logic can be written with native code as well. [13] Based on the above understanding of technology it was very natural to choose Qt/QML for making demo.

4.3 Design Ideas

After initial study of various home screens, it was observed that widgets solve the problem of viewing updated information partially. Although widgets update themselves at regular intervals, updates for certain applications cannot be fitted into the size defined for a widget. Also it was not possible to interact inside the widget. Thus the first two goals of design are - To introduce enlarged view and to introduce interaction in the enlarged view.

4.3.1 Filmstrip

Figure 8 a, b and c shows a design idea to implement Home screen based on film strips like icons. Figure 8a depicts the home screen with icons /application shortcuts

pinned to it. Since the language of choice is QML applications become

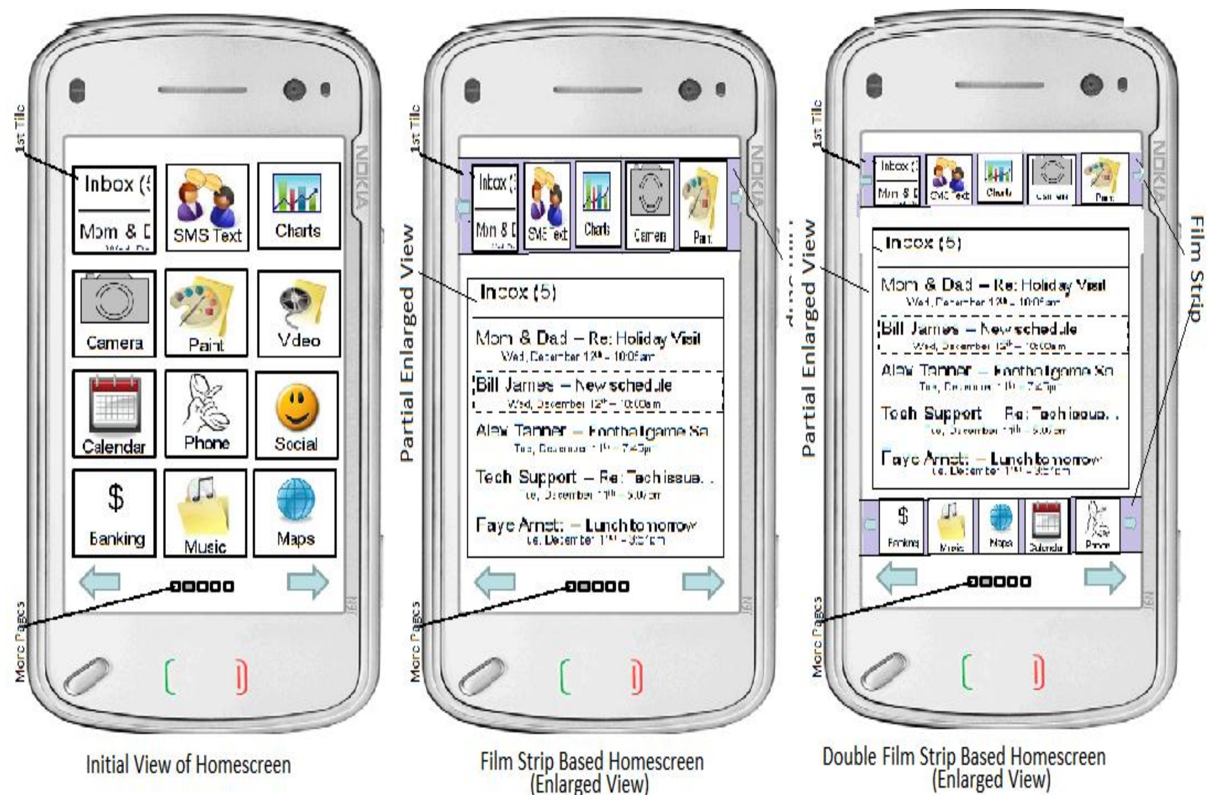

Figure 8. a)Initial View of home screen (b) Single film strip c) Double film strip

active as soon as they are pinned to the Home screen. As a result one is already able to have partial view of the full version of the application. It is now up to applications to decide how it should show its partial view. It can for e.g.: in case of camera or phone call delay instantiation of requisite native calls from the start up screen of the application. At the same time it can show some other text which indicates to user that he needs to instantiate the application. Here in figure 8a, only first tile is shown to be active. This is done in order to simplify and focus on the task.

Single Film Strip

Figure 8b illustrates the home screen, when the first tile is clicked. At the top there is a film strip which is a collection of all the icons on the home screen. One can observe icons from first and second row in the diagram. One can flick through the film strip to select other icons which were present on the home screen. Below the Filmstrip there is a partial enlarged view of the full version of application. One can interact in this area to look for more data. Also one can observe that the area is significantly large which

makes reading of data very easy. Thus with this solution there is an improvement in readability and interactivity over widgets.

This solution addresses the two core issues that were set at the start of finding of solution. But this solution does not suit the end user and he or she finds using the existing N8 home screen better. The reasons are pretty obvious when one looks at the deficiencies in this approach. All the icons are displaced from their existing positions making it difficult to user to find icon for other application. Single Film Strip view is not able to present all the icons from the current home screen page.

Double Film Strip

Figure on the extreme right hand side of Figure 8 shows idea of double Film strip. Figure on extreme left shows starting view of Home screen. When an application is clicked partial enlarged view is launched in the centre of the screen. At the top and bottom of the partial enlarged view there are film strips. This enables us to accommodate all the icons that are present on the home screen. This result in advantage that all home screen icons are visible along with enlarged partial view. Also readability and interactivity over widgets is immensely improved. But still all icons are displaced, making it difficult for user to locate icon for other application(s).

As a result one can observe, limitations that were present with single strip are eliminated. Also by starting with menu grid or iPhone like home screen, one is able to capture more application shortcuts on the home screen. Next iterations of design will try to eliminate the problem of displaced icons.

4.3.2 Bubble Home screen

Figure 9 a, b, c illustrates next iteration of designs which are evolving to our "wow" home screen or proposed solution. In figure 9a there is menu /iPhone like grid of icons. Applications are already active and are able to have partial view of the full version of the application.

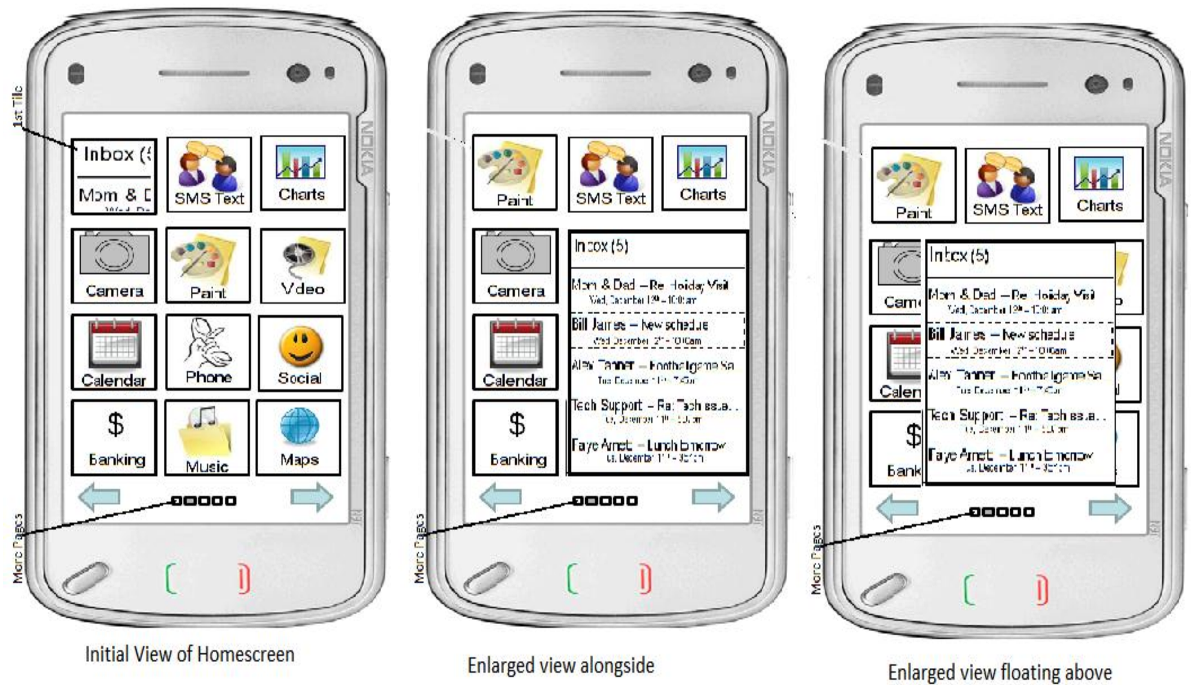

Figure 9. a) Initial View b) Enlarged view embedded c) Enlarged view floating

Applications decide how they should show their respective partial view. For e.g.: in case of camera or phone call delay instantiation of requisite native calls from the start up screen of the application. At the same time it can show some other text which indicates to user that he or she needs to instantiate the application. In order to maintain simplicity, only first tile in top left hand corner is shown to be active.

Embedded Bubble

Figure 9b depicts bubble which is embedded alongside of other icons on the home screen. One may notice that some icons have displaced from their position as a result of enlarged partial view which is launched. Launching of enlarged view can be done by clicking or panning over the area of first tile. The enlarged partial view or bubble in our case is interactive. Thus one can flick through or click buttons (if present in the view) and get more data. This results in large interactive view which is launched transparently. Also, almost all immediate neighbouring icons maintain their positions, making switching between application shortcuts on home screen easy.

But, one can notice that many icons on home screen are no more visible in current display area as a result of launching of enlarged partial view. And, some icons are repositioned from their original view making it difficult for user to locate some of the immediate neighbouring icons.

Floating Bubble

One will notice that although the launching of enlarged partial view is transparent, still access to all the icons, which were pinned to the home screen, is essential. With this in mind a unique solution is proposed in the figure 9c. The display is visualized as two surfaces instead of one. This means one can have home screen icons on lower surface while enlarged partial view on upper surface. The enlarged partial view like other experiments remains interactive and large. One can have much larger enlarged view or bubble compared to embedded bubble. Larger is the bubble much improved is the readability and interaction. As can be seen from picture, one is able to still access other icons on Home screen. Positions of icons are also not changed, making switching between icons, very convenient. Floating Bubble is launched at a prefixed location rather than at the point of initiation. This brings break in continuity. User needs to lift his finger to interact in the enlarged bubble rather than slide inside enlarged bubble. Also, partial hiding of icons could bring in annoyance to users and would need training to get used to such annoyance or distraction.

4.3.3 Fluid Home Screen Solution

The solution discussed in chapter 3 shows the fluid Home Screen Solution. These solutions build upon bubble Home Screen Solution and ensure that user experience is still rich.

Fluid Embedded Home Screen is depicted in the centre picture of Figure 7. It not only launches the enlarged view alongside other icons on home screen, but also ensures that all the icons that were originally pinned to home screen are visible. This it achieves by resizing and repositioning some of the icons on home screen. Framework support for repositioning and resizing of icons is discussed in chapter 5. Also the enlarged partial view is created exactly at the point of initiation.

Floating Fluid Home Screen also ensures that the enlarged partial view is launched exactly at the point of initiation.

5 Technical Details of Implementing Solution

The technical details of implementation will be discussed here. There are basically three problems that fluid home screen solution will try to address. The problems are running applications on the home screen, without need for developer to write another application like widget, resizing and repositioning of icons, draw two surfaces for popup bubble, so that both surfaces are visible and ensuring both the surface is interactive.

5.1.1 Running applications on the Home screen

Solution to first problem lies in using Qt Quick. Qt Quick is a collection of technologies that are designed to help developers create the kind of intuitive, modern, and fluid user interfaces that are increasingly used on mobile phones, media players, set-top boxes, and other portable devices. Qt Quick consists of a rich set of user interface elements, a declarative language for describing user interfaces, and a language runtime. A collection of C++ APIs is used to integrate these high level features with classic Qt applications. [14]

QML is a high level, scripted language. Its commands, more correctly elements, leverage the power and efficiency of the Qt libraries to make easy to use commands that perform intuitive functions. Drawing a rectangle, displaying an image, and application events -- all are possible with declarative programming. The language also allows more flexibility of these commands by using JavaScript to implement the high level user interface logic. [14]

To make Qt Quick possible, Qt introduces the QtDeclarative module. The module creates a JavaScript runtime that QML runs under with a Qt based backend. Because QtDeclarative and QML are built upon Qt, they inherit many of Qt’s technology, namely the signals and slots mechanism and the meta-object system. [14]

Thus when the home screen application is up and running, the user interface elements inside the application also become active with the help of JavaScript runtime. User

interface elements can be as simple as a button or as complex as photo viewer element/application. This means that applications can run in same process as that of home screen instead of creating a separate process. This is very much unlike to regular procedural language, where applications are usually separated by process boundary.

In order to determine solution for next two problems, one need to understand at Qt Graphics View architecture, as resizing and repositioning and drawing on surface involves Qt Graphics view.

5.1.2 Qt Graphics View Framework

Graphics View provides a surface for managing and interacting with a large number of custom-made 2D graphical items, and a view widget for visualizing the items, with support for zooming and rotation. [15]

The framework includes an event propagation architecture that allows precise double-precision interaction capabilities for the items on the scene. Items can handle key events, mouse press, move, release and double click events, and they can also track mouse movement. [15]

Graphics View uses a BSP (Binary Space Partitioning) tree to provide very fast item discovery, and as a result of this, it can visualize large scenes in real-time, even with millions of items. [15]

Graphics/View provides an item-based approach to model-view programming. Graphics/View architecture provides a non-visual class (QGraphicsScene) which holds all the item data and acts as model. A class called QGraphicsView is used to visualize the items on the scene. QGraphicsView can visualise different parts of scene. Thus one single scene can have multiple views observing. Graphics Scene holds items derived from QGraphicsItem class. Items can be of varying geometric shapes and sizes. Events generated on the scene are propagated to individual items. It maintains the state of item such as selection and focus handling. The architecture also supports animation and drag and drop.

The Scene

QGraphicsScene provides the scene for Graphics View. The scene serves as container for QGraphicsItem objects. One can add or remove items to scene using API provided by QGraphicsScene. One can also render parts of scene into a paint device. Scenes can be rotated, scaled, rendered and printed without a big hit on performance. One can do rendering on the scene using Qt's rendering engine or using OpenGL. Graphics scene can be used to present anything from just a few items up to tens of thousands of items or more. Scene is capable of notifying which items are colliding, which are selected and which items are top-level etc.

The View

QGraphicsView provides the view widget, which visualises the content of a scene. One can attach several views to the same scene, to provide several viewports into the same data set. Views can focus on separate items in the scene. It also provides scroll area to navigate large scenes. Input events from keyboard and mouse are translated to scene events before sending the events to scene that is been visualized. Like QGraphicsScene one can rotate and zoom using transformation matrix.

The Item

QGraphicsItem is the base class for graphical items in a scene. Graphics View provides several standard items for typical shapes such as rectangles, ellipses, etc. Items live in local coordinate system. They support mouse hover, mouse double click, mouse click, keyboard input, and keyboard focus etc. events. Several predefined QGraphicsItem subclasses are provided, including QGraphicsLineItem, QGraphicsPixmapItem, QGraphicsSimpleTextItem (for styled plain text), and QGraphicsTextItem (for rich text). One can also create custom classes which are derived from QGraphicsItem. Custom items are painted with the help of QPainter.

The Painter

The QPainter class performs low-level painting on widgets and other print devices. QPainter provides highly optimized functions to do most of the drawing GUI programs require. It can draw everything from simple lines to complex shapes like pies and chords. QPainter can operate on any object that inherits the QPaintDevice class.

The Declarative Subsystem

From Qt Graphics View Framework point of view, declarative Subsystem consists of three main classes – QDeclarativeView, and QDeclarativeItem. The QDeclarativeView class provides a widget for displaying a Qt Declarative user interface. QDeclarativeItem objects can be placed on a standard QGraphicsScene and displayed with QGraphicsView. QDeclarativeView is a QGraphicsView subclass provided as a convenience for displaying QML files, and connecting between QML and C++ Qt objects. [16]

The discussion about Qt Graphics View until now is going to be used to implement my solution.

5.1.3 Resizing and Repositioning

QML provides positioners which are container items that manage the positions and sizes of items in a declarative user interface. Various positioners that are provided by QML are row, column, grid and Flow. Also QML provides List view, Grid view, and Path view to provide different predefined visualisations. In order to resize and reposition icons in same view, positioners are better suited than various view elements provided by QML. Also creating different views and loading them is very heavy for the device when compared to resizing and repositioning on same view. Thus views are used to visualize different items of same scene, as recommended by Qt Graphics View framework. Hence forth only positioners would be discussed.

QML Row positioner positions child items so that they are horizontally aligned and are not overlapping. On the other hand, QML Column positioner positions child items so that they are vertically aligned and not overlapping. Both Row positioner and Column

positioner do not wrap around child items when cumulative width of children run over the view width. This means both row and column positioners cannot be used. Let's look at QML Grid positioner and QML Flow positioner if they are suitable for our need.

QML Grid positioner positions child items in a grid. Child items are aligned in Grid and are not overlapping. The grid positioner calculates a grid of rectangular cells of sufficient size to hold all items, placing the items in the cells, from left to right and top to bottom. Each item is positioned in the top-left corner of its cell with position (0, 0). [17] Grid positioner defaults to four columns if number of columns is not specified. New row is added if all child items do not fit into columns available on the row. But again like column or row positioners, it does not take into consideration the width of view. As a result if cumulative width of children on a row overruns width of view, then certain children are partially hidden. Another drawback of this is layout is that, cells take largest width of children in given column and largest height of children in given row. This means when icons are resized there would be an ugly effect of having lot of empty spaces around most of the cells.

Flow on the other hand places child items side by side from left to right wrapping as and when required. Flow item positions its child items like words on a page, wrapping them to create rows or columns of items that do not overlap. [18] Thus, Flow positioner seems to be at least better at handling the view space. It takes into consideration view width and based on that, decides whether it wants to place the child in same row or move it to next row. But it still suffers from same ugly limitation of having empty spaces around cells, as that of grid. This is a result of cell height been determined by maximum height of child cell in given row. So when flow decides to place next child on next row, it does not go a step further to check if it can actually fit the new child in vacant space on same row.

Thus one needs a positioner which is not only able to take into account width of enclosing view but also be able to identify empty spaces in the grid. It means one needs to write a custom positioner which would be able to meet all the requirements.

5.1.4 Custom Positioner

Icons are placed on the home screen with the help of Positioners. All elements that are displayed in Qt Quick are specialisation of QtDeclarativeItem QDeclarativeItem is derived from QGraphicsObject and QDeclarativeParserStatus. QGraphicsObject in turn is a specialisation of QGraphicsItem and QObject. This means that our positioner can be added to scene and a QDeclarativeView which is a specialisation of QGraphicsView can be used to display our positioner item. All applications that need to be pinned up to home screen need to be added as child of our custom positioner. This would enable us to take control of positioning of application shortcuts as and when it is needed. QDeclarativeItem is by default is not drawn on the view. Hence one would need to unset flag ItemHasNoContents indicating to the Graphics Framework that this item needs to be drawn. In addition to this, one also needs to override paint function of QGraphicsItem. This will ensure that our item and its children are painted by Graphics Framework.

5.1.5 Drawing two surfaces

The custom positioner is a specialisation of QGraphicsItem and it uses Qt Graphics Framework to draw, reposition etc. It makes solution to third problem of creating two surfaces on same scene for popup bubble easy. Since QGraphicsScene which is a 2D surface on which drawing is done, one does not really create two surfaces. What in turn one does is that, he or she draws one more item on top of existing child items of positioners. He or she needs to ensure, the item that would be represented as popup bubble would be the last item that is drawn. This way from end user perspective, the popup bubble would appear to be on top of other applications icons.

5.1.6 Interaction on surfaces

Interaction on items is a result of propagation of mouse, keyboard events from scene (QGraphicsScene) to individual items (QGraphicsItem). Since positioner and other applications are specialisation of QGraphicsItem drawn on QGraphicsScene, one can very well use this feature of Qt Graphics Framework. This event propagation being a Qt Graphics Framework feature, one actually does not need to do any separate implementation to enable this. All visible items will by default start responding to events from user. It is up to the home screen application to decide how it would like to use that event.

5.1.7 Architecture

There are three possible solutions technically to implement or write our custom positioner

1. Edit Qt sources so that our new positioner could be embedded in the sources. Qt been open source, it is easy to do this. But this brings a problem that one needs to install custom Qt libraries on all the devices that are available. This is rather an inconvenient and risky solution. Since one does not know exact situation of libraries on device, our custom library would run into risk of breaking existing applications that run on existing official libraries.

2. Other way to achieve this is by embedding a new type inside an executable. Since QML is built on top of Qt one can use Qt's meta-object system. To this object register our type using qmlRegisterType function. After registering the new type, one can easily access it from QML. Although this solution is pretty much self-sufficient it makes our application tightly coupled and needing one to continuously generate new executable if one has to make even a small change to application. Also the new positioner that is written might be useful to other clients as well. So it makes sense to keep our positioner independent of the Home screen application.

3. Third option that is to write a plugin. This plugin can be accessed from qml. Writing a plugin involves moving relevant code into the dll, which client can discover and access without prior knowledge.

Qt provides two APIs for creating plugins [19]. A higher-level API and lower level API. Higher level API is used for writing extensions to Qt itself: custom database drivers, image formats, text codecs, custom styles, etc. And, lower-level API is used for extending Qt applications.

In order to create a plugin library, one needs to declare a plugin class that inherits from QObject and from the interfaces that the plugin wants to provide. He/She also needs to tell Qt's meta-object system about the interfaces. This can be accomplished passing information to Q_INTERFACES() macro. After this, one should export the plugin using the Q_EXPORT_PLUGIN2() macro. And then build the plugin using a suitable .pro file.

Also Qt provides QDeclarativeExtensionPlugin an abstract class for writing custom QML plugins. In order to use plugin in our QML home screen application, it is derived from QDeclarativeExtensionPlugin class. QDeclarativeExtensionPlugin is a plugin interface that makes it possible to create QML extensions that can be loaded dynamically into QML applications. These extensions allow custom QML types to be made available to the QML engine. [20]

Steps to create a custom QML extension plugin [20]:
In order to write a custom QML extension plugin, one needs to subclass QDeclarativeExtensionPlugin, implement registerTypes() method to register types using qmlRegisterType(), and export the class using the Q_EXPORT_PLUGIN2() macro. One should then write an appropriate project file for the plugin and create a qmldir file to describe the plugin

After plugin is written it is deployed by copying the generated dll under Qt plugin path. When application using plugin is run, it will first look for plugin under application's executable directory. For e.g.: If the executable is running from /usr/bin application will try to locate plugin under /usr/bin. If it is unable to locate plugin at that location then it will look under path specified by QLibraryInfo::location (QLibraryInfo::PluginsPath). This path is usually QTDIR/plugins, where QTDIR is the path where QT is installed. One can also add path which are separate from these path using QCoreApplication APIs setLibraryPaths() and addLibraryPath().

5.1.8 Class Diagram

Figure 10 illustrates class diagram of the important classes that are involved in the solution. All the classes involved are not shown, for the sake of clarity.

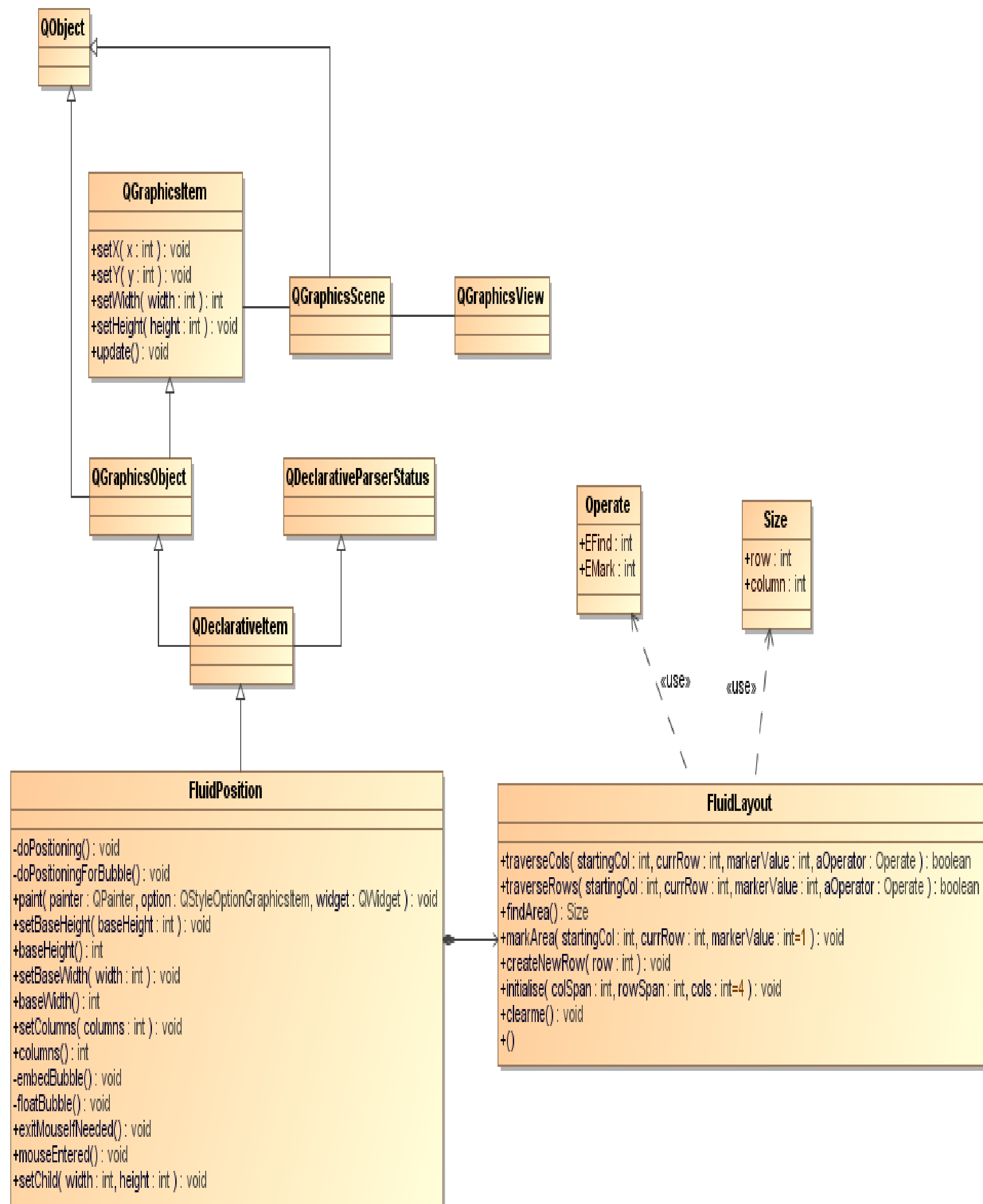

Figure 10. Class diagram

One can see from the class diagram that there are two classes which form the basis of framework implementation. FluidLayout class determines the x and y coordinates of

each item. It also maintains a map that helps in determining the new positions of items. FluidLayout can be specified to either find an empty space which suits the required size of item or can be specified to mark item area in the internal map. Logic to determine empty space or mark area is similar. It traverses horizontally to determine if empty spaces on the row are good enough to satisfy column requirement for the item. Once it finds its enough, it traverses vertically to determine any possible collision down. If it finds collision in the path, then current row is aborted and it moves on to next row to repeat same process.

FluidPosition class is derived from QDeclarativeItem. The QDeclarativeItem class provides the most basic of all visual items in QML. All visual items in Qt Declarative inherit from QDeclarativeItem. Although QDeclarativeItem has no visual appearance, it defines all the properties that are common across visual items - such as the x and y position, the width and height, anchoring and key handling. By default QDeclarativeItem does not draw any visual item. In order for our derived class to be visible, one needs to unset flag ItemHasNoContents. QDeclarativeItem is derived from QGraphicsObject which itself is derived from QGraphicsItem. Thus this enables us to listen to mouse events, mouse hovers, keyboard events, keyboard focus etc.

In order to draw item paint function inside the class needs to be overridden. Paint function uses QPainter to do the painting. QPainter performs low level painting on widgets and other paint devices. QPainter provides highly optimised functions to do most of the drawing that GUI programs require. Together with QPaintDevice and QPaintEngine classes, QPainter forms the basis for Qt's paint system.

FluidPosition also offers APIs for setting/getting base width and base height. Base width and base height determine minimum size of cell. Offering an option of setting base width and height, to client(s) of FluidPosition, brings in flexibility for client(s). Fixing minimum cell height and width beforehand also helps in eliminating ugly spaces that are seen with default positioners.

FluidPosition also offers API to set number of Columns for the positioner. By default it is set to 4, if nothing is specified explicitly. One will notice that an API for setting rows

is not offered. This is done intentionally. Rows are added based on children that are yet to be positioned in the view.

FluidPosition has two APIs doPositioning and doPositioningForBubble. DoPositioning() is supposed to determine position of the children, so that partial enlarged view is placed alongside of other children, when launched. Similarly doPositioningForBubble() is supposed to determine position of the children, so that partial enlarged view is placed on top other children as popup bubble, when launched.

Both doPositioning and doPositioningForBubble determine initial position of children with the help of FluidLayout APIs. The input for row span and column span comes from the child declaration. Child is expected, to provide this input in their declaration. If it is not provided, then row span and column span, is assumed to be one. In the application row span and column span is forced to be same during initialisation phase. This is done to keep visual appearance of the application simple.

5.1.9 Activity Diagram

Since we have two approaches to launch a partial enlarged view, we will discuss two activity diagrams in this section.

Approach 1: Partial Enlarged View launched alongside of other application shortcuts.

Figure 11 illustrates the sequence of calls that take place when an event takes place to launch enlarged partial view of the application.

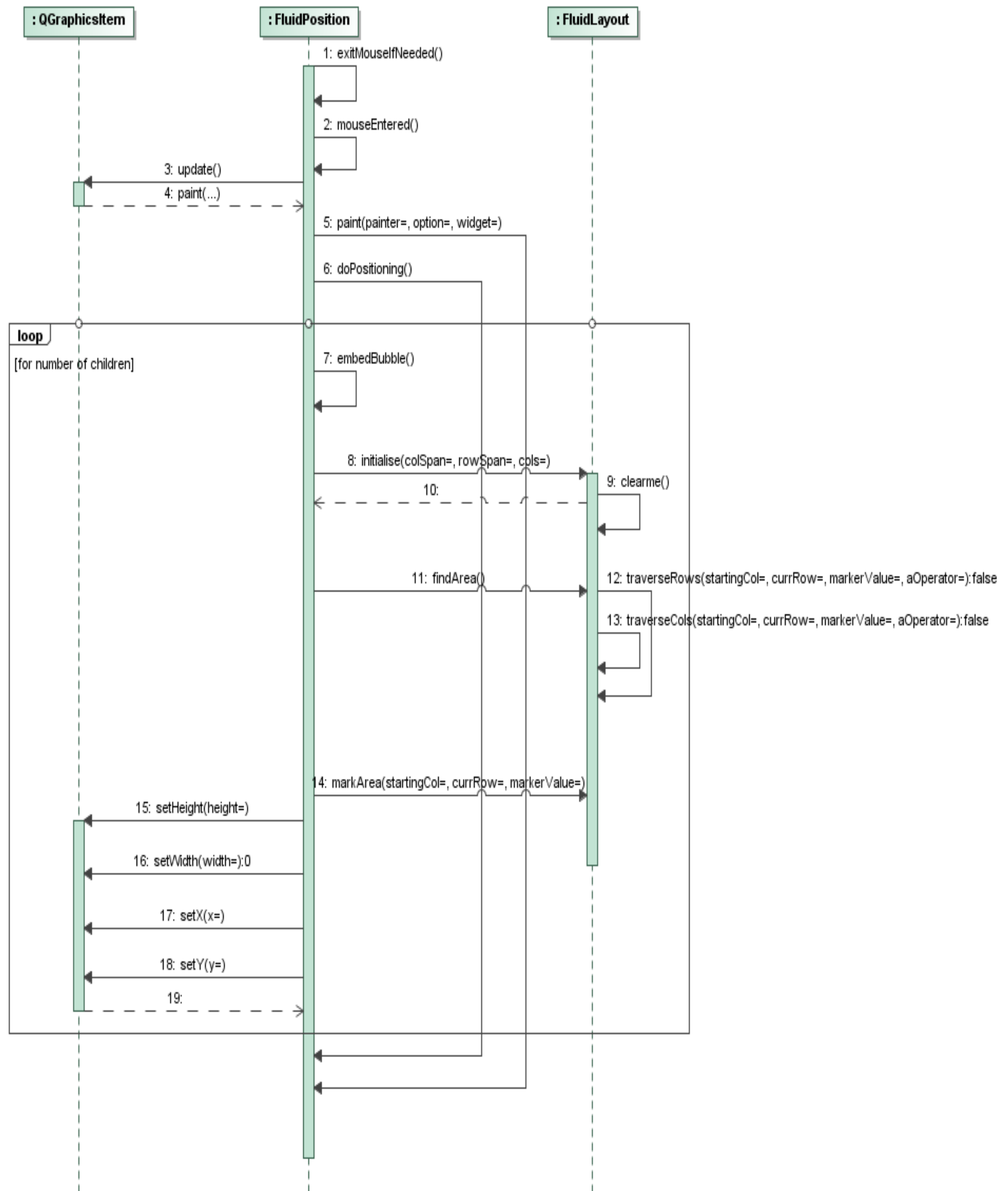

Figure 11. Activity Diagram: Partial Enlarged View is launched alongside other application shortcuts.

Client application or in this case home screen application which is not shown in the diagram above, will initiate the process for displaying enlarged partial view on a Mouse Click Event.

Initially it checks if there was any previous mouse event that needs cleaning up. This is done by call to exitMouseIfNeeded(). This function inturn sends update request to the Qt Graphics View Framework. Before sending update request, internal flag is set to indicate that all the items need to be reset. Update call to Qt Graphics View Framework indicates that repainting of Graphics Item is needed. Hence virtual function paint is called by the framework. On receiving the paint call, all the children items inside the item are reset or repainted to default values. All this is not shown in the above diagram in order to simplify the diagram and focus on more relevant and important part of information.

After child items are reset, again internal flag is set to indicate that request for enlarged partial view is ongoing. Update function call is sent to the Qt Graphics View Framework. Framework in turn calls overridden paint function of FluidPosition class.
Since an enlarged partial view is launched alongside of other applications, doPositioning function of FluidPosition class is called. Inside this function all the remaining repositioning and resizing of children elements takes place.

EmbedBubble function locates the child which needs to be expanded. Expansion size is hard coded to be half the number of columns. Also inside this function, based on position of the child, other children are relocated. This is needed as child size is expanded at same position as that of child.

Expanded area is nothing but new bigger column span and row span of the child. Inorder to store new size initialise function of FluidLayout class is invoked.Initialise call sets the column span and row span for the child. This information is used by findArea to determine right location for the child in the view’s children map.
FindArea method as mentioned above is used to locate right location for child in the children map. Find Area method of FluildLayout class determines this by calling traverseRows() function. This function in turn calls traverseCols(). This way traversing column by column and then row by row, ideal location for the current child item is determined.

This determined location is then passed to markArea() call of FluildLayout class. This way information of all the children and their locations in the map is captured. This information is also used by findArea function to locate vacant spots inside map.
After a suitable empty location is located height, width, X and Y coordinates for the child item is set, by calling virtual functions of QGraphicsItem.This information is used by Qt Graphics View Framework when it paints item on the view.

Since expanded partial view consumes a large real estate of device view, there is less space left for other children. This is compensated by reducing the column span and row span of other children. This way all the children are accommodated in the same view, as well as are able to paint an enlarged partial view for one of the child.

Approach 2: Partial Enlarged View launched as a popup bubble floating above other application shortcuts.

Figure 12 depicts the sequence diagram for popup bubble floating above other application shortcuts.

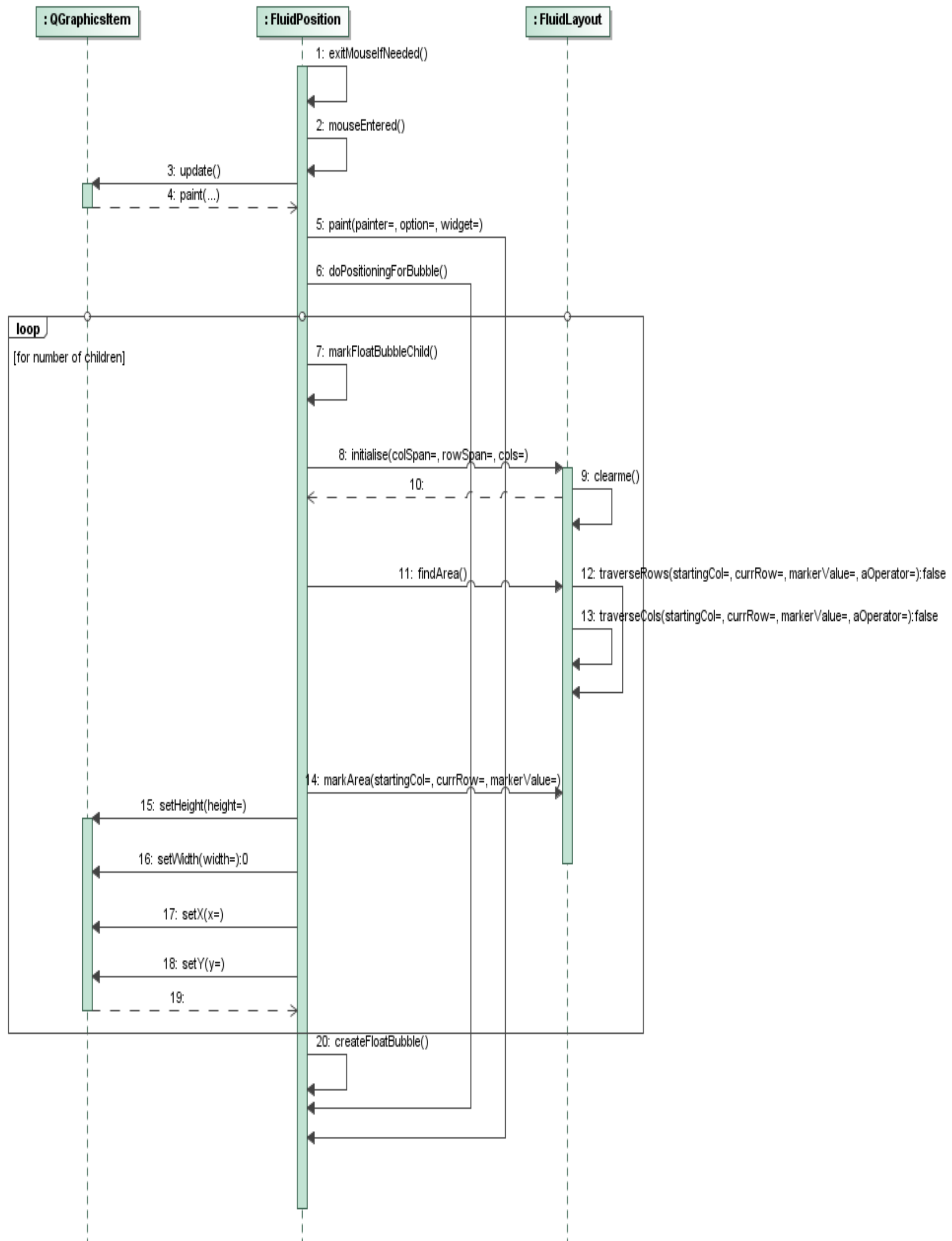

Figure 12. Activity Diagram: Partial Enlarged View is launched as popup above other application shortcuts.

One can observe that sequence diagram is pretty similar to enlarged partial view created alongside other icons. However there are some minor differences to the earlier implementation which are discussed in the following section.

One can observe that doPositioningForBubble() is called instead of doPositioning(). This is because a bubble is created on top of other child items. While traversing through children that are present, the item which needs to be enlarged is marked. After all the children are painted, a new child which floats at location under the point of initiation of event is created.

Also the column span and row span of this new child is larger than regular children. Since the child is drawn on top of all children, it has possibility of expanding up to n columns and n rows. But doing that would actually look like application is launched in full screen. Hence the column span and row span is chosen in such a way only few children are hidden below the bubble but at the same time the floating bubble is large enough for user to have comfort of interaction.

Floating bubble has advantage over embedded bubble due to the possibility of having a larger expanded partial view. But hiding of certain application shortcuts due to presence of Floating bubble, does not make it a natural choice when compared to embedded bubble.

6 Conclusion and discussion

The project focused on creating a home screen which would be able to accommodate several of application shortcuts on a single home screen page and at the same time is able to offer a large and interactive window for quick access to application data without the need for launching the whole application.

The proposed solution provides an improved method of presenting a partial view of a full version of the program. The method includes providing an enlarged display of the first tile in response to a first input. The method also includes an interactive area inside the enlarged view.
Thus with the enlarged partial view not only the view area for the tile is improved. The interaction inside the view in turn mitigates unnecessary launching of the application there by increasing the overall user experience.

Such a solution has its advantages and disadvantages. This solution provides detailed information on updated data. As a result the user is able to get more information about the updated data. Since the enlarged partial view provides enough information about the updated data, one does not need to launch the application for getting details of the updated data. As a result task switching needed to get similar updated information from various applications is greatly reduced. With this solution one can have far greater live application accelerators pinned to home screen than the current home screen solutions. This is possible because the enlarged view can be launched alongside other application accelerators dynamically. One can easily add more live accelerators to the home screen compared to for example The Windows mobile home screen.

However, this increase in live accelerators results in more usage of RAM. This would mean the hardware needs to support more RAM size in order to support the increased load. QML is a declarative language which means code is interpreted on runtime. Although this makes application active, interpreting on runtime increases the load time of application. If there are a number of QML elements in an application, then the application loading time would be significantly increased. Since the home screen has various applications pinned up, the loading time of home screen is significantly increased. Also a declarative module needs to keep runtime parser in memory in order

to interpret QML files. This brings an increased load on the memory and in turn affects performance of the home screen application. The enlarged partial view as a floating bubble hides some shortcuts under it. It could be inconvenient for the user who would want to switch between applications. This solution relies heavily on panning and flicking for launching and navigating inside the partial enlarged view. Thus such a home screen provides for better interaction and readability, but it is only suitable for touch based devices rather than type devices.

References

1. Hassenzahl, Marc; User Experience and Experience Design [online]. The Interaction-Design.org Foundation; 2011.
   URL:http://www.interaction-design.org/encyclopedia/user_experience_and_experience_design.html
   Accessed 28 January 2012

2. User Experience [online]. Wikipedia. 5 August 2003.
   URL: http://en.wikipedia.org/wiki/User_experience
   Accessed 28 January 2012

3. Usability [online]. Wikipedia. 5 August 2003.
   URL: http://en.wikipedia.org/wiki/Usability
   Accessed 28 January 2012

4. Windows Phone [online]. Wikipedia. 18 April 2008.
   URL: http://en.wikipedia.org/wiki/Windows_Phone
   Accessed 28 January 2012

5. Iphone [online]. Wikipedia. 29 June 2005.
   URL: http://en.wikipedia.org/wiki/IPhone
   Accessed 28 January 2012

6. Iterative Design [online]. Wikipedia. 21 March 2005.
   URL: http://en.wikipedia.org/wiki/Iterative_design
   Accessed 28 January 2012

7. N8-00 Specification[online]. Developer.Nokia.com. 27 April 2010.
   URL: http://www.developer.nokia.com/Devices/Device_specifications/N8-00/
   Accessed 28 January 2012

8. Symbian [online]. Wikipedia. 27 March 2003.
   URL: http://en.wikipedia.org/wiki/Symbian
   Accessed 28 January 2012

9. Widget [online]. Wikipedia. 19 September 2001.
   URL: http://en.wikipedia.org/wiki/Widget
   Accessed 28 January 2012

10. Web developer's library [online]. developer.nokia.com.
    URL:http://library.developer.nokia.com/index.jsp?topic=/Web_Developers_Libra
    ry/GUID-044FA5DD-71AC-4BF1-869D-74BA1C776DFA_cover.html
    Accessed 28 January 2012

11. Qt (framework) [online]. Wikipedia. 24 August 2001.
    URL: http://en.wikipedia.org/wiki/Qt_%28framework%29
    Accessed 29 January 2012

12. Qt Quick [online]. Wikipedia. 15 July 2010.
    URL: http://en.wikipedia.org/wiki/Qt_Quick
    Accessed 29 January 2012

13. Qml Introduction [online]. Doc.qt.nokia.
    URL: http://doc.qt.nokia.com/4.7-snapshot/qml-intro.html
    Accessed 29 January 2012

14. Graphics View Framework [online]. Doc.qt.nokia.
    URL: http://developer.qt.nokia.com/doc/qt-4.8/graphicsview.html
    Accessed 29 January 2012

15. QDeclarativeView Class [online]. Doc.qt.nokia.
    URL: http://doc.qt.nokia.com/4.7-snapshot/qdeclarativeview.html
    Accessed 29 January 2012

16. Qml Grid Element [online]. doc.qt.nokia.
URL: http://doc.qt.nokia.com/4.7-snapshot/qml-grid.html
Accessed 29 January 2012

17. Qml Flow Element [online]. doc.qt.nokia.
URL: http://doc.qt.nokia.com/4.7-snapshot/qml-flow.html
Accessed 29 January 2012

18. How to create Qt plugins [online]. doc.qt.nokia.
URL: http://developer.qt.nokia.com/doc/qt-4.8/plugins-howto.html
Accessedd 29 January 2012

19. QDeclarativeExtensionPlugin Class [online]. doc.qt.nokia.
URL:http://developer.qt.nokia.com/doc/qt-4.8/qdeclarativeextensionplugin.html#details
Accessed 29 January 2012

Developer Guide

This developer guide is meant to be a guideline for developers who would like to make applications for Home screen. It draws heavily from learnings under Conclusion and Discussion section.

- Partial Loading - Simultaneous loading of all applications leads to significantly high load times. In order to reduce this, a developer should ensure that only relevant parts are loaded at start up. Other parts of the application should be loaded as and when required.

- Relinquish Resources – -The developer should ensure that a minimum number of resources is used and they are relinquished as soon as their use is completed.

- Design Enlarged Partial View – The enlarged partial view is capable of showing a full view of the application in the partial view itself. However this would bring undesirable effects and a bad user experience. So to avoid this, it would be desirable that the developer creates an enlarged partial view, which is loaded at the time of displaying the enlarged partial view. This way the developer can control the user experience he would desire in that view.

- Avoid scroll bars for navigation – Such a home screen is suitable for touch based devices. On any touch based devices using scroll bars for navigation is a bad user experience. If at all scroll bars need to be shown, they should be shown to indicate the displacement of the context from the starting point of the application.

- Be concise and clear – The enlarged partial view is not a place to slap all the details of the application. It is a place for quick and useful information. So display a minimum amount of relevant information which is very easy for the user to read.

- Avoid inputs from the user – The enlarged partial view has very limited space even though it is an enlarged version of the shortcut. So having text boxes or buttons can make it difficult for the user to click and enter.

- Support Panning, Flicking and/or Dragging – The home screen is suitable for touch-based devices. Navigating for more information is usually recommended using panning, flicking or dragging. Make it a point to support this in the enlarged partial view.
- Lower case all files – With Linux-based operating systems becoming popular on mobile devices, the case of the name of the file becomes essential. In order to avoid problem after application installation, it is recommended that all the files have their name in lower case.
- Entry point QML or Main Qml of your application should be named as "accelerator.qml". This is essential as framework looks for this file as the starting point of the application.