Lappeenranta University of Technology
Department of Information Technology

DYNAMIC RUNTIME VARIATION AND BRANDING OF S60 SOFTWARE

Supervisors Professor Jari Porras and M.Sc. Erik Simko
Instructor M.Sc. Jani Nurminen

Lappeenranta, 23.4.2007

Jere Antikainen
Notkokatu 1 B 7
FIN - 53850 Lappeenranta
Finland

ABSTRACT

Author: Antikainen, Jere

Subject: **Dynamic Runtime Variation and Branding in S60 Software**

Department: Information Technology

Year: 2007

Place: Lappeenranta

Master’s Thesis. Lappeenranta University of Technology. 51 pages, 23 figures, 2 tables, and 2 appendices.

Supervisor: Professor Jari Porras

Keywords: Symbian, S60, branding, variation of software

This thesis studies ways of branding and variation of S60 software dynamically during run-time. S60 is a platform used by several phone manufacturers and their phones are used by numerous operators. Operators want to differentiate a phone or selected applications on the phone using their own brand. This brings a need for a way of branding a whole phone or some of its applications. Some applications may need to change their variation depending for example on a used service provider. Also the variant data may need to be shared between several applications or modules. The work introduces Symbian operating system and S60 platform and considers the challenges and restrictions that platform security and data caging of Symbian operating system bring to variation of software and sharing variant data between several applications. Already existing methods are also considered as the basis of the thesis. The work includes a presentation of a project where an implementation was created for branding an S60 application dynamically, and for sharing its branded data with other applications.

TIIVISTELMÄ

Tekijä:	Antikainen, Jere
Nimi:	**Dynamic Runtime Variation and Branding in S60 Software**
Osasto:	Tietotekniikan Osasto
Vuosi:	2007
Paikka:	Lappeenranta

Diplomityö. Lappeenrannan teknillinen korkeakoulu. 51 sivua, 23 kuvaa, 2 taulukkoa ja 2 liitettä.

Tarkastaja: Professori Jari Porras

Hakusanat: Symbian, S60, brändäys, ohjelmistojen muuntelu

Diplomityössä tutkitaan keinoja brändätä ja varioida S60-ohjelmistoja dynaamisesti ja ajonaikaisesti. S60 on kehitysalusta, jota käyttävät useat puhelinvalmistajat ja heidän puhelimiaan käyttävät lukuisat eri operaattorit. Operaattorit haluavat puhelimiensa tai osan puhelimen sovelluksista erottuvan kilpailijoista heidän omalla brändillään ja tämän takia täytyy olla keinot joko koko puhelimen, tai valittujen sovellusten brändäykselle. Osa sovelluksista saatetaan haluta vaihtavan käytettyä brändiä sen käyttämien resurssien, kuten verkkopalvelimen, mukaan. Variointidataa tulee myös pystyä jakamaan eri sovellusten tai sovellusten osien kesken. Työssä esitellään Symbian käyttöjärjestelmä ja S60 kehitysympäristö, sekä pohditaan Symbianin turvallisuuskäytäntöjen tuomia haasteita variointidatan jakamiseen eri sovellusten välillä. Olemassaolevia variointitapoja tutkitaan työn mahdolliseksi pohjaksi. Työ sisältää esittelyn projektista, jossa kehitettiin erään S60 sovelluksen dynaaminen brändäystoteutus, joka myös mahdollistaa variointidatan jakamisen eri sovellusten kanssa.

PREFACE

I would like to thank SYSOPENDIGIA for giving me the idea and inspiration for this thesis. I would especially like to thank my former project manager Jari Hakulinen for creating the opportunity to write this thesis in the first place, and the rest of the project team for ideas and support during the implementation phase. But most of all I want to thank my instructor Jani Nurminen for having the patience to guide me through this process, and for great comments and guidance during the writing. Last but not least, special thanks to Kaisa for giving me the motivation to finally graduate.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABBREVIATIONS

2G, 2.5G, 3G	Mobile Communication Generations
AIF	Application Information File
API	Application Programming Interface
CONE	Symbian OS Control Environment
DBMS	Database Management System
DMA	Direct Memory Access
DLL	Dynamic Link Library
GUI	Graphical User Interface
IPC	Inter-Process Communication
IPSec	IP Security Architecture
IrDA	Infrared Data Association
ITC	Inter-Thread Communication
OS	Operating System
PIM	Personal Information Management
ROM	Read Only Memory
SID	Secure ID of an Application in Symbian OS
SSL	Secure Sockets Layer
TCB	Trusted Computing Base
TCE	Trusted Computing Environment
TLS	Transport Layer Security
UART	Universal Asynchronous Receiver/Transmitter
UI	User Interface
WAP	Wireless Access Protocol
XML	Extended Markup Language

1 INTRODUCTION

In the smartphone industry the products of a phone manufacturer are used by several operators. For operators it is essential to their business to bring forward their own brand in the applications of a phone, because building brand recognition amongst the consumers is one of the key factors in building a brand. [1] This brings challenges to the phone manufacturers, since creating the software for the smartphone is time consuming and expensive. Customizing the software for each operator is possible, but this can complicate the version control and configuration management of the software a lot which may lead to problems with the quality of the software and delays in the software development. [2]

In Symbian Operating System (OS) there is also a built-in system of having some of the appearance, behaviour and functionality of applications stored externally of the application executable. This system is the usage of resource files which are separate text files written in a Symbian OS-specific resource language. Resource files are compiled into a binary file format which then can be loaded and read by applications. Resource files make it possible for information to be loaded only when it is actually needed and they can be changed and recompiled without the need to recompile the main program. Resource files are used for example in localizing applications for different languages. [3]

There are, however, some problems in using resource files as a tool for branding an application. Backwards compatibility of the resource files is hard to maintain because resource files are compiled in a way that the order of the resource definitions is decisive. That means that adding definitions for newer versions and maintaining compatibility with older versions becomes a tedious task over time. Also, having several resource files active at the same time for one application is not possible, so any application which needs to have several different brands active at one time, for example in different views, would require continuous loading and unloading of resource files. The sharing of brand data with different applications is also problematic with resource files. If the branded resource files were to be shared between applications, they would have to be available in a public directory, available for reading to any application. [4] This would probably not be desirable by operators, since branding data usually includes copyrighted material which could be easily copied from a public directory.

There are also other methods for customizing the UI of the S60 smartphone, for example skin support [5]. With skins it's fairly easy to change the graphics of the whole phone or just a single application, but skins cannot be used for customizing the functionality of an application since skin support is only about customizing graphics. Also, there can only be one skin active at one time, so if for example two applications need to change their skin depending on the situation, or have two different sets of skins active at the same time, then this becomes a problem. And also the user of the phone can change the used skin of the phone at any time, so the phone is not branded permanently, which might not be the best choice for the operators who want their brand to be visible all the time on their device.

There are also methods of varying only the functionality of the software. The term varying (e.g. variant, variating etc.) in this work refers to the process of altering the appearance or functionality of software. Variating can be either static or dynamic. The variation methods which require the software or some parts of it to be recompiled are so-called static methods.

Dynamic variation can be performed during run-time of the software without the need to recompile the software.

A component called central repository [6] provides an API to allow one or more clients to open repositories, and to provision and retrieve information from those repositories. By having some functionality flags defined in the central repository, the varying can be dynamic but having a centralized method for varying the UI and the functionality of the software dynamically while the software is running would make branding the phone a lot less complex and time consuming process.

1.1 Branding

The word brand comes from an ancient Norwegian word *“brandr”* which means burning. Brands have been used for hundreds of years to distinguish products from another producer’s similar products. The first real brand is said to be born on an island of Greece long before the birth of Christ. At that time a good oil lamp was a lamp which lasted longer than other lamps. When buying a lamp it was impossible to tell which lamp was better than others.

According to the legend oil lamp producers on one island were able to produce lamps which lasted much longer than other lamps because the quality of clay on the island was superior. They started to mark their lamps with their symbol and the merchants selling their lamps started to get bigger profits for the lamps when word started to get around that those lamps lasted longer. Even though the legend does not tell the name of the product, the first brand in the world was born. [7]

1.1.1 Branding in General

The term brand is ambiguous. Brands are not only physical products or services. A brand may also be a company, such as Nokia, an event, or a person such as Björn Borg. A brand is the additional value a consumer is willing to pay for a product compared to a generic product which fulfills the same purpose. [7]

A brand signals a consumer about the origin of the product and protects both the consumer and the producer from competitors who are trying to offer similar products. The elements, such as name, logo, or symbol, which help recognising, noticing and remembering a trademark can be called elements of a brand. But a brand should not be considered only as a name or a symbol; the concept extends beyond the external features of a commodity. A brand consists of all the positive and negative impressions, which accumulate over time, about the product, distribution chain, staff and communication. So a brand is not just a symbol which differentiates a product, but it's all that a consumer thinks when seeing the symbol. [8]

Building a brand starts from the background factors; goals, market researches, and identity. Once the elements are chosen, image and reputation of the brand are created. Creating a brand cannot start without there being something that is built into a brand. A product has to have some distinguishing feature which can be advertised to the consumers. [7] In case of mobile software, the distinguishing features can be the appearance of the applications or small features and usability issues in them. [9]

Advertising has to be able to affect emotions since it helps people to remember the information. If the information is too confusing the mind easily ignores it totally. Especially, if the information given in an advertisement is new in addition to being confusing, the image of the advertisement fades very quickly. Information has best chances of getting through when it can be attached to something previously known and concrete. [7] Advertisement has an important role in building a brand. Quality of advertisement is very important; according to research the quality of an advertisement is four or five times more important than the amount of money spent in advertisement. [10]

Marketing is used to bring a new brand into the knowledge of consumers. The goal is not, however, creating something new, but rather enhancing the existing image. The *4P*-model created by Philip Kotler [7] breaks the marketing basics down into four segments – a so called marketing mix: product, price, promotion and place. Product is the most important of these since it is concrete and visible. The most critical is price since if the price does not meet the consumers' vision of the value of the product; he will easily go for the competitor's product. Promotion is all the actions of a company, with which it strives to promote the product to the desired customers. The last segment, place, consists of distribution channels, coverage area, and transportation. Figure 1 depicts the 4P-model.

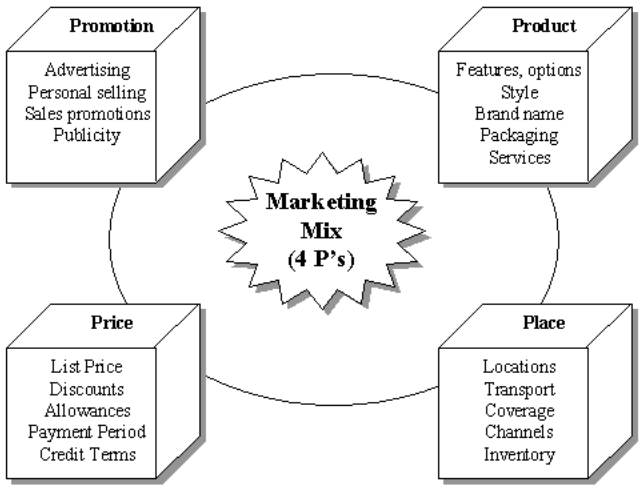


Figure 1: The 4P-model

A successful brand product has a higher price than similar non-branded products, but even so the consumers are willing to buy the brand product. Price is one of the values a brand has to have. If the price is lowered the brand starts to go downhill towards the generic commodities. This tells, that in a modern consumer society, a name and image alone don't sell; a product has to have a status which a high price brings along with it. [7]

Brand recognition starts with a consumer remembering having heard of the brand or seen the brand name or logo, or even connecting the brand into the right product group. Real signs of brand recognition are a large scale advertisement of the brand, the brand having been on the market for a long time, easy availability of the brand product, and that the brand is successful. A good slogan, a distinguishing melody, or a successful use of a symbol as a logo can be decisive elements in building brand recognition, because they are all easier to remember than just words. Also well built media publicity may work well in building brand recognition, even better than advertisements. A paid advertisement comes from the company, but an image given by the media origins form outside the company. [7]

1.1.2 Branding of Software

Branding of software is quite a new thing. Its history is analogous with the general history of branding; it has been introduced with a sort of mass production of software. The smartphone industry is probably the first industry starting to mass produce software in this way. One generic commodity, e.g. an application, is turned into different products, with the similar kind of functionality just by branding it. For example a generic instant messaging application is branded with different operator's graphics and features and it suddenly is a totally different product than the same application with a different set of graphics and branded features.

Branding software is basically just like branding any other product; the same rules apply to software as well. The software being branded is differentiated from the competitors by graphics, features, and functionality. The difference in branding software is on the technical side – how to enable software to be branded. [9]

1.2 Objectives

The objective of this thesis is to study methods of varying S60 software dynamically during runtime to enable flexible ways of branding a complete S60 smartphone or some of its applications. It would be ideal to be able to use some of the existing software variation methods provided by the S60 Platform, but the study should not be limited to them. The thesis also presents a project where an implementation was created for branding an S60 application and for sharing its branded data with other applications.

1.3 Structure of the Work

The Symbian OS and some of its aspects which are more closely related to this thesis are overviewed in Chapter 2. Chapter 3 describes the S60 UI framework and the methods of varying software in S60 applications and the limitations of those methods. The implementation described in Chapter 4 is the practical part related to this thesis. Chapter 5 considers the results of the project and draws conclusions of the whole study.

2 SYMBIAN OS

Symbian OS, produced by Symbian Ltd, is an advanced, open operating system licensed by the leading mobile phone manufacturers in the world. It is designed for the specific requirements of advanced 2G, 2.5G and 3G mobile communication devices. It combines the power of an integrated applications environment with mobile telephony, bringing advanced data services to the mass market. Symbian OS was developed from EPOC by Psion Software and it runs exclusively on ARM processors. Symbian OS is structured like many desktop operating systems with pre-emptive multitasking, multithreading, and memory protection. The major advantage of Symbian OS in smartphones is that it was designed for handheld devices. [11]

Different phone manufacturers require very different kind of mobile phones. The Symbian OS allows its licensees to modify the user interface part to suit the needs of each one. Figure 2 shows the Symbian OS software layers. The kernel part of the Symbian OS supports multi-threading and the core kernel library includes support for all features that are essential for the operation system such as timers, direct memory access (DMA) engines, interrupt controllers, and universal asynchronous receiver/transmitter (UART) serial ports. [12]

2.1 Platform Security

Symbian OS introduced the concept of platform security in version 9.1. Platform security is a defence mechanism against malicious or badly implemented code. It is a fundamental concept addressing the security and integrity of data and applications on a smartphone. There is a need to prevent badly implemented applications from doing unwanted things on the phone, while still enabling an open environment for third party applications.

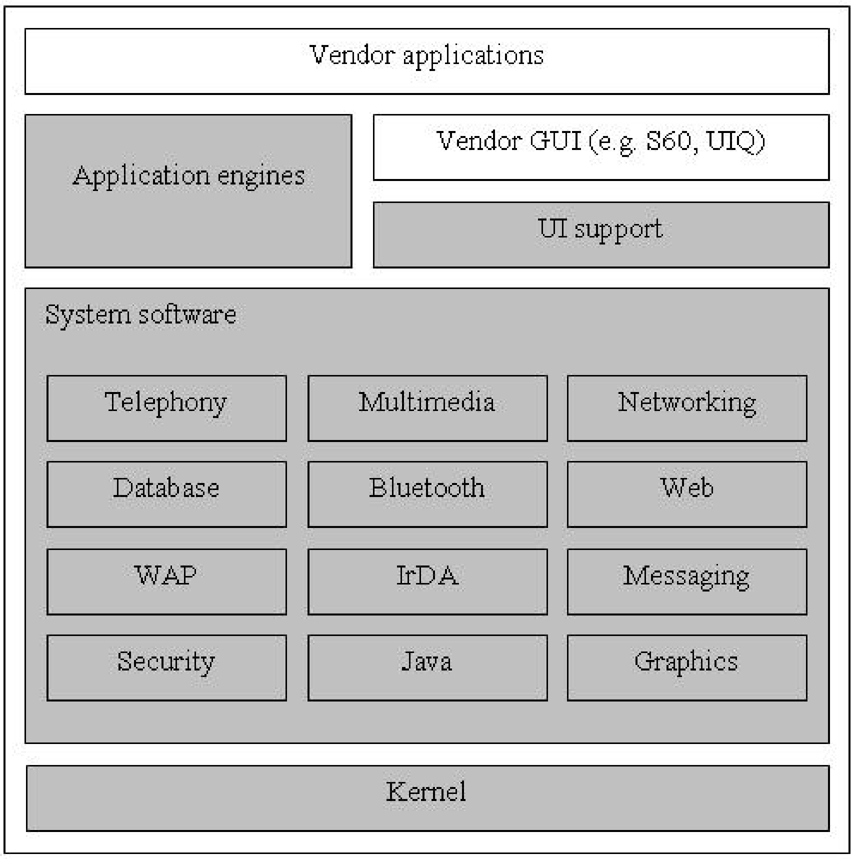


Figure 2: Symbian OS software layers [12]

This prevention is done by determining the trust level of applications before installation and after that ensuring that the installed applications cannot perform unwanted actions after their installation to the phone. [13] The security subsystem also enables data confidentiality, integrity and authentication by providing support for secure communications protocols, such as Transport Layer Security (TLS), Secure Sockets Layer (SSL), and IP Security Architecture (IPSec). It provides also support for authenticating installed software using digital signatures.

A so-called Trusted Computing Base (TCB) consists of the kernel, file server, and software installer. All components of TCB have unrestricted access to the all the resources of the device. TCB is responsible for maintaining the integrity of the device, and applying the fundamental platform security rules. Outside this core, other system components require access to some sensitive system resources, but not to all. The Trusted Computing Environment (TCE) consists of these key operating system components which protect the resources of the device against misuse. Figure 3 depicts the TCB and TCE.

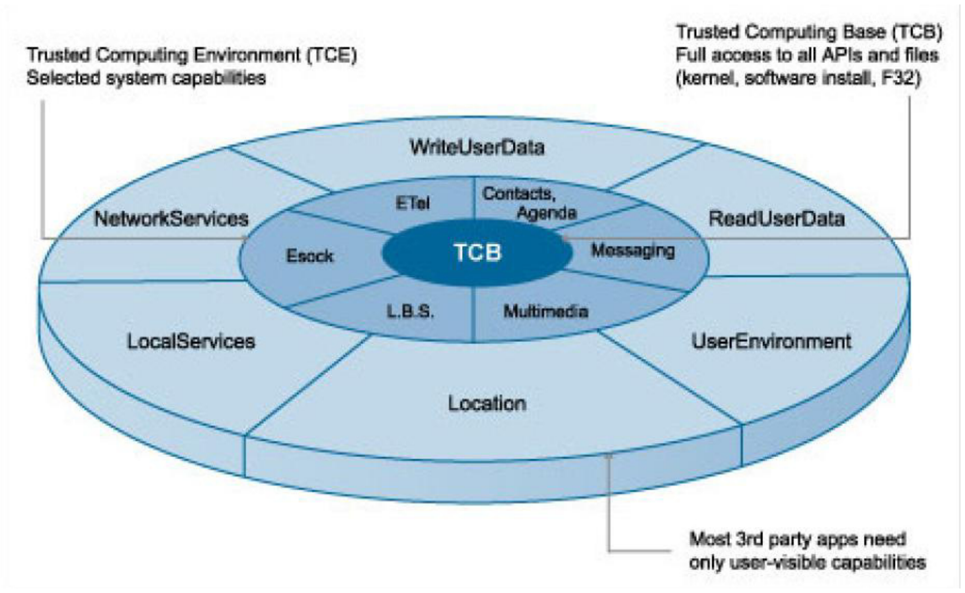

Trusted Computing Environment (TCE)
Selected system capabilities
Trusted Computing Base (TCB)
Full access to all APIs and files
(kernel, software install, F32)
WriteUserData
NetworkServices
ReadUserData
ETel
Contacts, Agenda
Esock
TCB
Messaging
L.B.S.
Multimedia
LocalServices
UserEnvironment
Location
Most 3rd party apps need
only user-visible capabilities

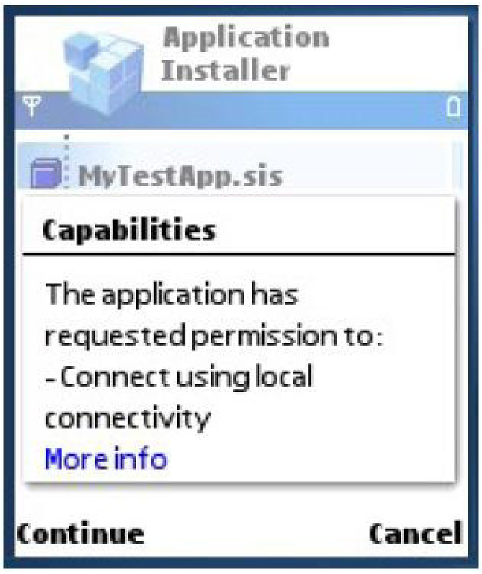
Figure 3: The TCB and TCE [11]

The main concern the platform security is trying to address is preventing unauthorized access to the user data and system services. As the users of smartphones use their phones as calendars and diaries it is imperative that the user's data is protected from being accessed by malicious software or corrupted by poorly implemented applications. As for system services the most important part to secure is the kernel executive itself. All other system resources, e.g. the telephony server, are used through user-side servers.

The access control to system services is done through the use of so-called capabilities. Capabilities are access tokens which correspond to permissions to undertake sensitive actions or groups of actions, or to access sensitive system resources. When an application is installed on the phone, it is granted a set of capabilities. These capabilities are managed by the kernel of the operating system and after the software is installed, they can no longer be changed. Capabilities are used for two things, they define permissions and limits for application’s access, but they also represent the level of trust the application has. [14]

Capabilities are used to verify the level of trust when linking static libraries or loading dynamic link libraries. A process always gets the capabilities of the executable file and those capabilities will never change during execution. A library can be loaded dynamically into a process only if the library has equal or higher capabilities than the loading process has, since for a library the given capabilities express the level of trust it has. But even if the library has more capabilities than the process it is loaded to, it can only perform operations according to the capabilities of the process.

Capabilities can be divided into two groups: basic capabilities and extended capabilities. Basic capabilities are capabilities that can be given by the user. Basic capabilities are confirmed from the user during software installation and are available until the software is uninstalled. Figure 4 shows the confirmation query shown to the user during software installation. Table 1 lists the user capabilities that can be granted to applications. [14]


Application Installer

MyTestApp.sis

Capabilities

The application has requested permission to:
- Connect using local connectivity

Figure 4: Capability confirmation query during software installation [4]

Table 1: User capabilities in Symbian OS Platform Security

NetworkServices	Grants access to remote services without any restriction on its physical location. Typically, this location is unknown to the phone user. Also such services may incur cost for the phone user. This typically implies routed protocols. Voice calls, SMS, internet services are good examples of such network services. They are supported by GSM, CDMA and all IP transport protocols including Bluetooth profiles over IP.
LocalServices	Grants access to remote services in the close vicinity of the phone. The location of the remote service is well-known to the phone user. In most cases, such services will not incur cost for the phone user. This typically implies a non-routed protocol. An application with this capability can normally send or receive

	information through Serial port, USB, IR and point-to-point Bluetooth profiles. Examples of services are IR beaming with the user's PC, Bluetooth gaming, and file transfer.
ReadUserData	Grants read access to data belonging to the phone user. This capability supports the confidentiality of user data. Todo-list, Contacts, messages and calendar data are always seen as user confidential data. For other content types such as images or sounds, it may depend on the context, and ultimately be up to the application owning the data to define.
WriteUserData	Grants write access to user data. This capability supports the management of the integrity of user data. Note that this capability is not necessarily symmetric with ReadUserData. For instance, one may wish to prevent rogue applications from deleting music tracks but may not wish to restrict read access to them.
Location	Grants access to the live location of the device. This capability supports the management of user’s privacy regarding the phone location. Location information protected by this capability can include map co-ordinates and street address, but not regional or national scale information.
UserEnvironment	Grants access to live confidential information about the user and his/her immediate environment. This capability also protects the user's privacy. Examples are audio, picture and video recording, and biometrics (such as blood pressure) recording.

Extended capabilities are more powerful capabilities that allow software to do more complicated tasks. These extended capabilities can only be granted to Symbian Signed software. Symbian Signed is a process launched by Symbian to verify and supply Public Key Infrastructure security model based signatures and certificates for application suppliers' software. Figure 5 shows the procedure for applying for Symbian Signed status. [14] See APPENDIX 1: System capabilities of Symbian OS for a list of system capabilities which are only granted to system services.

Capabilities are also used to verify permissions when using inter-process communication (IPC) or using a service provided by a server. IPC is designed for message delivery over process boundaries, which are also memory management boundaries. Kernel is responsible for memory management and message delivery and that makes it a secure and reliable way for processes to communicate.

In Symbian OS, every executable has its own Secure Identifier (SID). SID is used to identify the running process launched from the executable and it is stored in the executable binary itself. [14] There are different ways to implement the authentication in IPC. Client capabilities can be checked either run-time or build-time. Servers can also authenticate their clients based on their SIDs, but neither Platform security nor the client/server framework forces this: every server can decide whether or not to request capabilities from its clients. The clients have more limited options available: the server is usually authenticated by name. [14]

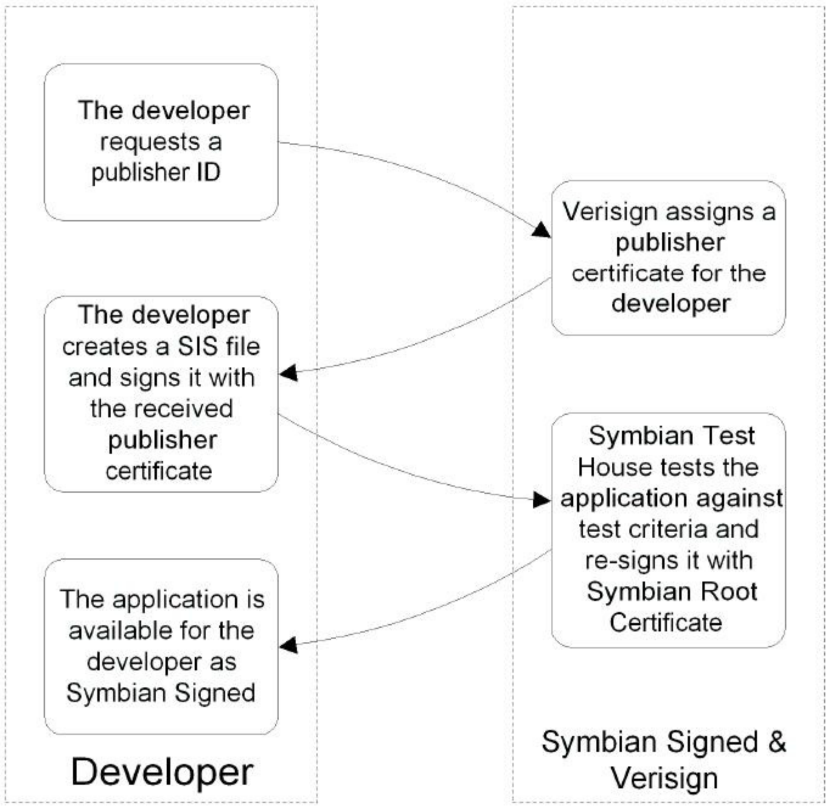

The developer requests a publisher ID
Verisign assigns a publisher certificate for the developer
The developer creates a SIS file and signs it with the received publisher certificate
Symbian Test House tests the application against test criteria and re-signs it with Symbian Root Certificate
The application is available for the developer as Symbian Signed
Developer
Symbian Signed & Verisign

Figure 5: Symbian Signed process [14]

Platform security also includes a feature called data caging which is meant to protect both application executable file and data files. There are predefined directories into which certain file types, such as executables and resources, are stored and only applications with powerful enough capabilities can access these directories. Data caging also allows applications on Symbian OS device to have private data, which is not accessible by other applications, in a so-called private directory. This directory exists for each executable on the phone memory. The private directory can only be accessed by the executable; only the executable containing the right SID can access the private folder assigned to it. Table 2 describes the protected directory structure.

Table 2: The protected directory structure [14]

\Sys	Directory which holds system-critical files and executables. This directory can only be modified by the Kernel, File Server, or Software Installer components of the platform.
\Sys\bin	Directory containing all the executables. This is the only directory where native C++ software can be run from.
\Resource	Directory which holds read-only resource files potentially shared by all applications. Only the Software Installer component can modify these files.
\Private	A process-specific directory that is identified by a SID of a process. Only processes with the same SID can access the directory.

All other directories in the system are not protected by data caging, which means that also unsigned applications without any capabilities have full access to them. [14]

As branding data usually is something the operators want to protect against unauthorized access, the private directory is a perfect place for storing data securely. However, the access restriction of this folder brings difficulties into sharing the branded data with other applications and so a separate process granting authorized access to the data would be ideal for this approach.

2.2 Client/Server Framework

Symbian OS client/server framework allows programs to offer services to multiple other programs. Many important Symbian OS system APIs use the client/server framework to provide services to client program: for example, the Window Server, File Server, Messaging and ETel.

Client/server framework has four key concepts: server, session, sub-session and message. A server is a separate executable which is run in its own process. It provides services to other processes through a client interface API which is contained in a separate DLL. Interaction between a client and a server is done with a message-passing protocol, which means that there are no direct pointers between them, leaving the integrity of the server and its resources untouched by clients.

Communication between a client and a server is managed by inter-thread communication (ITC). There is no pure IPC in Symbian OS, as the kernel ensures that ITC messages are delivered to corresponding threads, even over process boundaries. It is important to realize that message delivery is an unreliable method, since messages may be lost if the server’s message slots are full.

A session is the logical communication channel between a client and a server. A session can be reserved for a single client thread or shared between several. A server-side implementation defines how client messages should be handled. A sub-session is an efficient refinement of a session when a client wants multiple simultaneous uses of a server. For example, with the File Server, each opened file is handled through a separate sub-session. Figure 6 shows the basic structure of the Client/Server framework in Symbian OS.

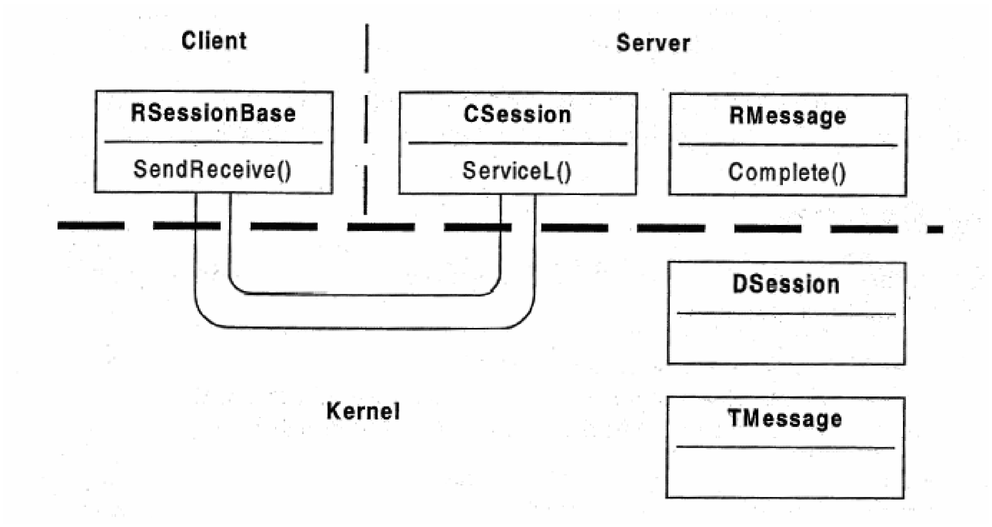


Figure 6: Client/Server framework in Symbian OS [18]

A message is a data structure passed between the client and the server through the session. It contains a code specifying the client request type and four 32-bit data parameters. The parameters can be pure number information or a pointer to a client-side descriptor. Decoding of the parameters is known only by the server and its clients.

All data which does not fit into the provided integers must be represented as a descriptor, and an address of a package object (*TPckg*) containing the descriptor is delivered within the message. The server then uses safe inter-thread read and write functions to access the provided descriptor. Figure 7 shows the relationship of the descriptor class (*TDesC*) and the message package class (*TPckg*). [14]

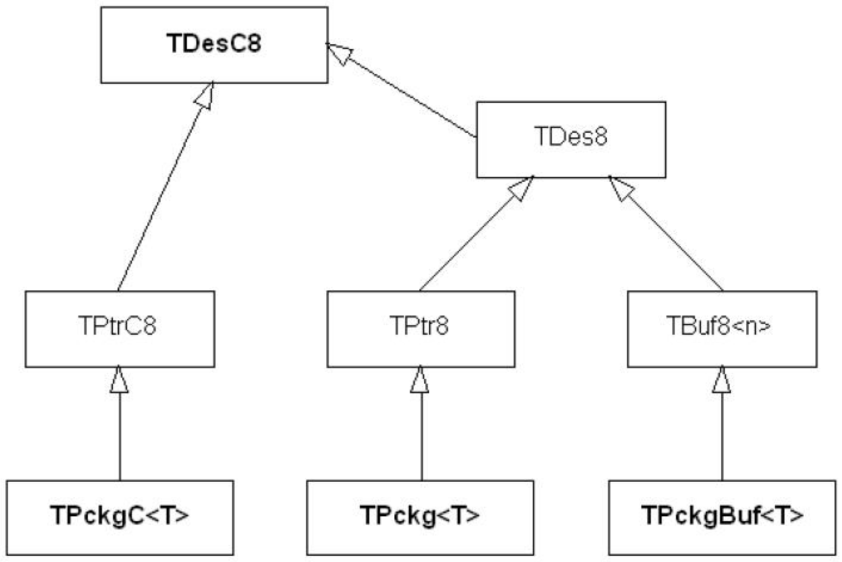


Figure 7: Relationship of a message package and a descriptor class [14]

2.3 Database

> "Symbian OS DBMS provides features for creating and maintaining databases, and implements reliable and secure data access to these databases via both native and SQL calls. These calls are supported by a transaction/rollback mechanism that ensures that either all data is written or none at all." [15]

There are two implementations provided by Symbian OS: a small and relatively lightweight client-side implementation and larger scale client/server implementation. DBMS uses basic Symbian OS features such as File Server, Permanent File Stores, and Streams. *RDbStoreDatabase*-API provides an interface to create and open a database exclusively, meaning that the database access is not shared. The operations to this database are done directly to a file. *RDbStoreDatabase*-API is only for client-side access. *RDbNamedDatabase*-API provides an interface for creating and opening a database which is identified by name and format.

This API can be used either as client-side or shared client/server database.

Symbian OS DBMS provides a server facility that allows multiple clients to access the same database at the same time. A transaction mechanism makes sure that only one client at a time can change data. The API has two key concepts, DBMS server session and database change notifier. The DBMS server session provided by *RDbs* allows databases to be shared by multiple clients with read/write access. Database clients can request notifications of events, such as transactions committed or rolled back, through *RDbNotifier*. [15]

The abstract base class *RDbDatabase* provides the transaction support. A transaction may be started explicitly, but if it is not, then all updates are made inside an automatic transaction which is started by the database itself. Automatically transactions are also committed by the database, but explicitly started transactions must be either committed or rolled back; that is, either permanently changing the current database state or restoring the previous state. [15]

The transaction mechanism does not provide isolation between clients. If one client is updating the database within a transaction, any other client can see the changes as they are made. This means that if a client is reading two different rows from the database, it is possible that the second row is changed in between the read operations, resulting in an "inconsistent read". This can be prevented by using a read-lock. A read-lock is done via enclosing the individual read operations within a transaction. Requesting a read-lock in the database fails if any other client has a write-lock to the database. Other clients with read-lock will not affect the completion of this request. [15]

As the shared database uses the Symbian OS client/server framework, the database can be protected against unauthorized access with platform security. The access to the database can be restricted to a number of SIDs and so only the defined applications can read from and/or write to the database. This enables a secure way of sharing variant data between several applications if branding would be implemented using DBMS. There were however some doubts about the performance of the DBMS, especially with large amounts of data. This was verified with prototyping in the pre-study phase of the project and as a result DBMS was not chosen as the basis of the implementation. See Chapter 4.2 about the prototyping.

3 Variation of Software in S60

The S60 Platform (previously known as Series 60 User Interface) is a platform for mobile phones which use Symbian OS. The S60 platform is a complete smartphone software package that provides a mandatory base of technology implementations. It is developed mainly by Nokia and it is currently amongst the leading smartphone platforms in the world. Figure 8 shows the high level architecture of S60 Platform.

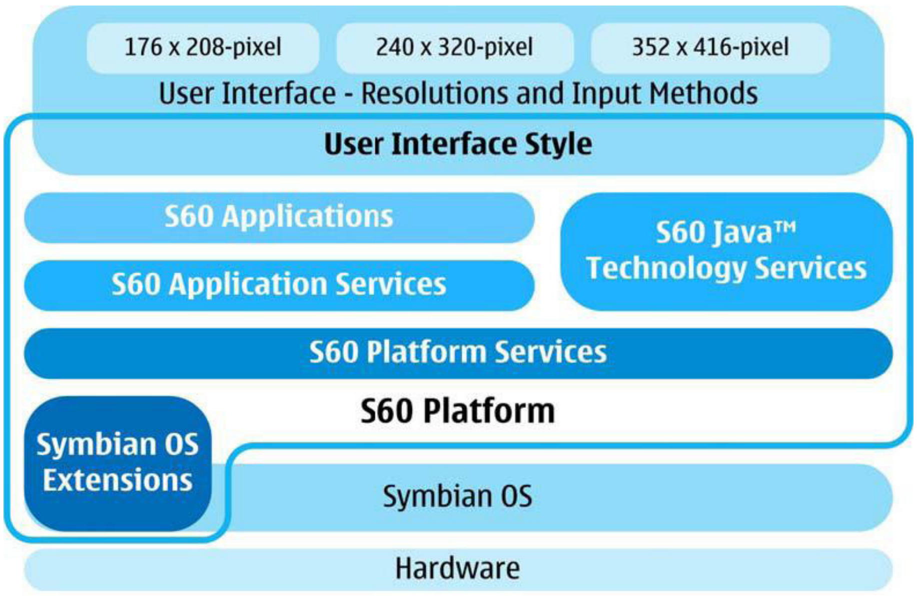


Figure 8: S60 high level architecture [16]

The S60 Platform offers a wide range of platform services and application services. Platform services are the fundamental services offered by the platform. [16] These include:

- Application framework services, which are the basic capabilities for launching applications and server processes

- UI framework services, which provide the concrete look and feel for UI components and handling UI events
- Location services, which allow the S60 platform to be aware of the location of the device
- Web-based services, which provide services to establish connections and interact with web-based services such as browsing, file download, and messaging
- Multimedia services, which offer the capabilities to play audio and video as well as support for streaming and speech recognition
- Communication services, which offer support for local and wide area communications, including Bluetooth, WLAN and voice calls

The application services provide the basic functionality for applications. [16] The application services include:

- Personal information management (PIM) services, which provide the fundamental features of PIM, such as contacts, task management, and calendar
- Messaging application services, which offer support for different messaging features including short messages, multimedia messages, e-mail, and instant messaging
- Browser application services, which provide the services for web content viewing, and video and audio rendering

3.1 Static Variation of Software

Static variation methods in S60 can include for example resource files, feature flags and the feature discovery API. All of these methods require parts of the software to be recompiled when changes in the variant data are to be taken into use.

3.1.1 Resource Files

The Symbian OS resource file system is a way of storing some of the application's appearance and functionality externally to the main body of the program. A resource file is a text file which is compiled into a binary file format using a Symbian OS resource compiler. The resource files can be compiled independently without the need to recompile the main program. There are two types of resources in Symbian OS: resource files (.rss) and application information files (.aif). Figure 9 shows the file types and tools used for resource compilation process. [3]

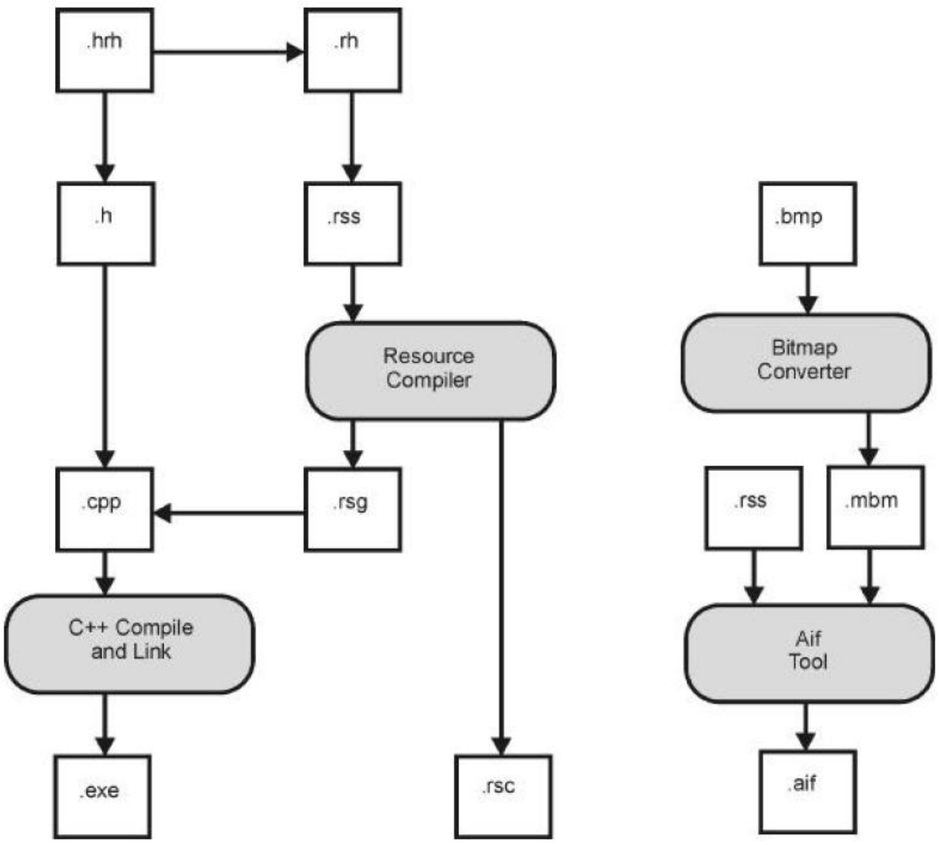


Figure 9: File types and tools used for resource compilation process [3]

An application information file defines the application behaviour in the system and resource files define behaviour within an application. When a resource file is compiled the output is an .rsc file. To access the resource in an application code an .rsg file containing indexes to the resource file is included using the *#include* directive of C++. The resource index file is automatically created by the resource compiler. [3]

As the index file is created every time the resource file is compiled, it is imperative that the ordering of the existing resource definitions is maintained if the resource file is expanded, or the source compatibility with older versions of the software is lost. This is because the old version of the software uses the old index file for loading resource items. If a newer version of the resource is introduced in such a way that the indexing of the items has changed for the part which is used by the older version of the software, then loading items using the old indexes will result in loading wrong types of items, e.g. reading a structure from the resource file when trying to read a string, leading into weird behaviour of the software and most likely crashing.

Although resource files can be used for varying both the UI and the behaviour of an application there is a limitation to the system; there can be only one resource file active at one time for an application. Multiple resource files can of course be used for an application by loading and unloading them when needed, but this can eat up much of the processor time of the system if multiple resource files are needed constantly. Resource files are counted here as static variation of software since changing data in resource files requires the resource file to be compiled again into the binary file format before the changed data can be used by the applications.

3.1.2 Feature Flags

The software functionality can be varied by flagging some of its code so that some parts of the code are only compiled if a feature flag has a certain value. The code to compile is therefore selected at compile time and cannot be changed unless the code is recompiled.

Using feature flags can be problematic for version controlling and testability of the software. If there are several feature flags, then all the possible combinations of the flags should be tested. Increasing the number of flags increases the time needed for testing exponentially. Also when making changes, e.g. fixing bugs in the software, each part enclosed within a feature flag has to be carefully considered to make sure that the change is included in all possible feature flag combinations. [17]

3.1.3 Feature Discovery API

One varying mechanism used in S60 software is a centralized component, Feature Discovery API, which provides flag values to applications. These flag values are then used to determine programmatically which optional features are present on the devices executing the application. For example saving data to a memory card is only possible if memory card access is supported by the system. However this is not dynamic variation as such, since flag values can only be changed in the ROM-image creation phase. Using Feature Discovery API brings the same challenges to testing as feature flags.

3.2 Dynamic Variation of Software

Dynamic variation of software means that the variant data can be changed dynamically and not only at compile time. In S60 platform this can include for example the use of skinning, local variation, or the varying of applications based on a used service provider.

3.2.1 Skinning

Skinning is a method of varying the UI of applications on the phone. It defines bitmaps and icons that are shown on the phone, but it cannot be used for varying functionality of the software. Also skins can be changed by the user and they affect the whole phone when they are active. So for example some applications that would require their appearance to be different according to the service provider they were using, e.g. an instant messaging program having different looks when connected to different servers would be impossible to implement with skins. Figure 10 shows an example of how skins can change the appearance of the application shell of the phone.

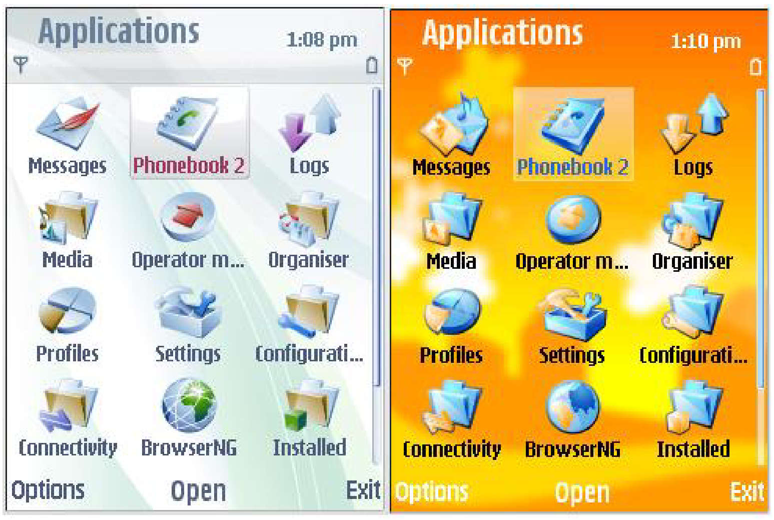


Figure 10: Example of skins

3.2.2 Local Variation

Symbian OS's central repository can be used to store functionality flags and the value of the flags can be changed dynamically. Central repository is basically a key/value-pair system and one of the biggest limitations of this system is that the keys need to be pre-defined. Also, central repository cannot be used for storing large amounts of data or for example files or bitmaps. [14] Local variation has the same problems with version controlling and testing as using feature flags.

3.2.3 Variation Based on Used Service Provider

Some applications vary their functionality according to parameters they get from a service provider they are using, for example a messaging server to which an instant messaging application is connected to. However, storing large amounts of data would require storage space from the network resource and sharing this data between different processes might not be practical as either every process would need to have a connection to the server, or there would have to be some kind of sharing framework implemented.

Also, fault tolerance of this kind of solution is limited; what to do when the network resource is not available? All in all using this kind of variation method has very limited use. The best way to implement this method would be to use it as semi-static variation method – meaning that the parameters needed from the network resource are pre-defined locally and they are just taken into use when connected to the service provider in question. However this makes this method very similar to using the Feature Discovery API.

Just as local variation, this method for varying software requires basically flagging the code and this brings again the same problems with testing and version controlling as using feature flags.

3.3 The Limitations in Current Variation Techniques

All of the varying methods described above have their advantages, but they also have their limitations. Also, comprehensive branding of software requires using several of those methods together. And even when combining all of the existing variation methods, it is very hard to create a solution for branding software with all the features a branding solution should have, e.g. ensuring backwards compatibility with earlier versions, support for multiple clients with several sets of branding data active at the same time, and support for skinning. See Chapter 4.1 for requirements set for the branding solution implemented in the practical part of this work.

Using resource files for branding provides one mechanism for branding the user interface and functionality of the software, but maintaining compatibility with older versions of the software can prove difficult. Compile time feature flags and the Feature Discovery API can be used for branding features and functionality, but they both require recompiling parts of the software when changes to the brand data are made. And both of these methods bring challenges to testing and version controlling of the software.

Skinning is a good way of branding the user interface, but it cannot be used for branding features or functionality of the software. With skinning it is also difficult to brand single applications and the user has a possibility to turn off the branding. Local variation and varying the software based on used network resource are feasible ways of branding features and functionality, but they cannot really be used for user interface branding. And they also bring challenges to testing and version controlling.

4 BRANDING SERVER

This Chapter presents a project for creating a solution for branding a single S60 application. The project consisted from several phases; pre-study including prototyping, design, implementation, testing, and documentation including writing of this thesis. There was a legacy system for branding the application preceding this project, but it could not fulfil all the requirements that the platform security and the S60 platform set for branding.

4.1 Requirements

The requirement for this project was to design and implement a **solution for branding** an S60 application. The solution would have to include **a way of defining the branded data, installing it on the phone ROM**, **updating the branded data** that already exists on the phone, support for **language variation**, and also **a simple API for using the data**. There was also a need to **have several instances** of a single brand and also several different brands active at the same time. The solution would also have to be implemented in a way that **backwards compatibility** with older versions of the clients using the data would be easy to maintain.

Support for skinning framework of S60 Platform was also one requirement. The rules for loading bitmaps when skinning is used are:

- If the requested bitmap is varied in the brand, then the bitmap from the brand is loaded
- If the bitmap is not varied in the brand, then the bitmap is loaded from the used skin, if it is available there
- Else the default bitmap is used

It has to also be taken into consideration that files may need to be varied according to the language as well.

4.2 Prototyping

As a part of a pre-study for this project there was a prototyping phase. A performance analysis was conducted with the prototypes and the results were reflected against the other requirements set for the project. Three types of prototypes were created; a **resource file based prototype**, **database based prototype**, and **client/server framework based prototype**. The prototypes are described in more detail in the following chapters.

4.2.1 Resource File Based Prototype

This prototype was a quite simple implementation of a DLL which used Symbian OS resource files as the variant data. The used variant data can be changed by changing the resource file in use. There were however clear limitations in this architecture with the requirements set to the project. Expanding the variant data is quite challenging since the resource file index is created every time the resource is compiled. That means that if the indexing was to change, every client would have to be recompiled to prevent use of wrong indexes. Using wrong indexes in fetching data would most likely result in crashing the client.

Also, updating the variant data which is in use by clients is also quite difficult. In order to install new resource files while the data is in use would require unloading the resource file, and the availability of the data would not be continuous. Having several brands active at the same time for one client would also result in performance issues, since one client loading data from different brands would require constant loading and unloading of resource files.

From a security point of view having this kind of architecture would be problematic, since there is no way to limit the access to the data as the resource files that are shared would have to be in a public folder.

4.2.2 Database Based Prototype

In this prototype the variant data is stored in a relational database. Clear benefits are well defined APIs for fetching the data, security through limiting the access by capabilities or SIDs, easy data updating and support for multiple clients simultaneously. Also using a database enables to a client access to several brands' data with no delays when changing the used brand.

However the performance of Symbian OS DMBS is not great, especially when using large databases as seen in Figure 11. Fetch times per item start to grow linearly when growing the database table size and even when fetching several items at a time the fetch time per item is still very high.

4.2.3 Client/Server Framework Based Prototype

In this prototype, the data is stored in a binary file, which is used by the server. The client API, implemented as a separate DLL, provides methods for reading the data. The Symbian OS client/server framework is used for message passing and it provides means for controlling the access to the server.

There are clear advantages in using this architecture. The security aspect is taken care of by the framework itself, the brand data is accessible from several different client processes simultaneously and, as seen in Figure 12, the fetch time per item is not very high, especially when fetching multiple items at a time.

4.2.4 Performance Measurements

Performance of the prototypes was measured by having several different sized sets of brand data. Each prototype was used to fetch items in sets of 1, 50 and 200 items per fetch operation from each set of brand data. The fetch times per item were calculated from the results.

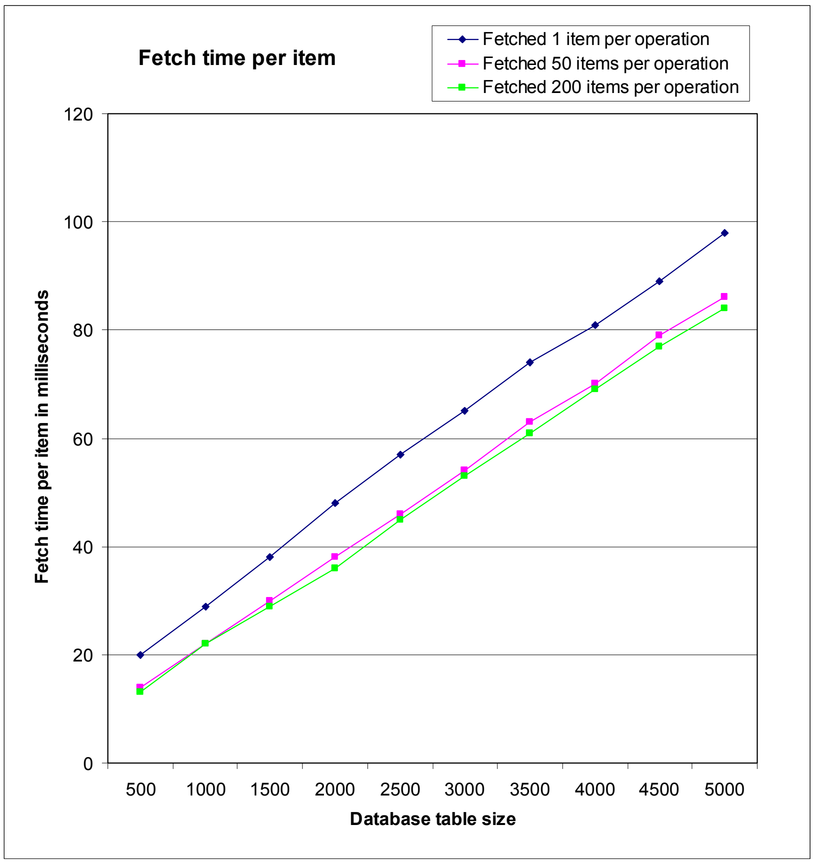


Figure 11: Fetch times per item from a database

Figure 11 shows the measurements for the database prototype. It is clearly seen from the results that the fetch times per item grow linearly in relation to the database table size. Also, the overhead caused by the DBMS for each fetch operation is quite high, since even when fetching several items in one operation the fetch time for single item is not particularly low.

In Figure 12 there are measurements for the client/server prototype and for the resource file based prototype. The fetch times per item in the resource file based prototype do not vary when fetching more items at a time, since the fetch operations are still separate read operations from the resource file, even though for the client application the API provides a way of fetching several items at a time.

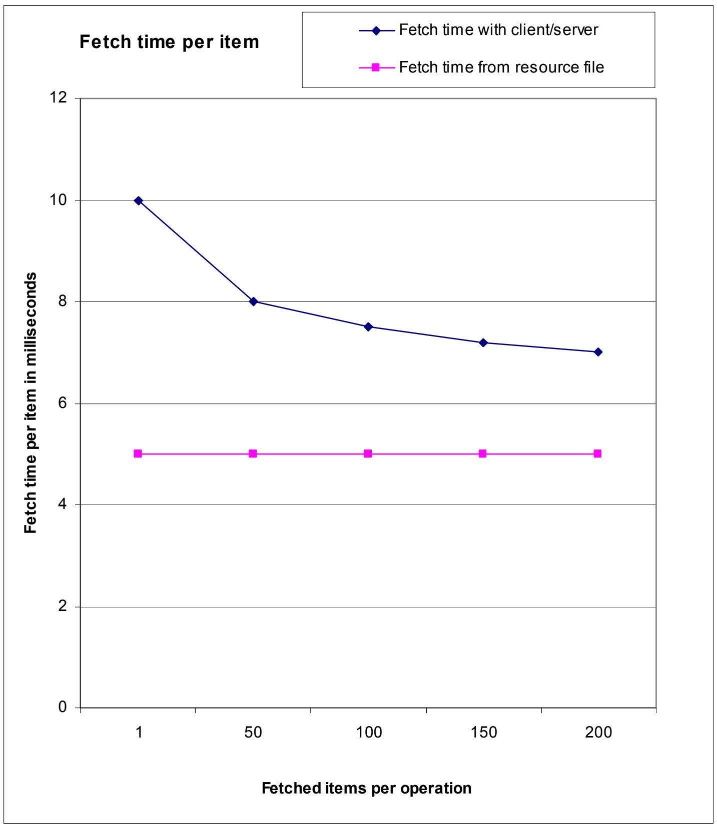


Figure 12: Fetch times for a client/server and a resource file prototypes

For the client/server prototype the fetch times per item are lower when fetching more items at a time. This is due to the slow startup of the server process and the time it takes for the processor for context switches between client and server processes for each client/server-operation.

4.2.5 Results from Prototyping

The results of performance measurements and other features of the different prototype architectures were reflected against the requirements set for the project. As a result the client/server prototype was selected as the base for the implementation of the project. The clear benefits are the inherent security of the client/server framework, the capability to handle multiple clients simultaneously, and the relatively low fetching times of the brand data items.

4.3 Variant Data Definition

For branding of the software to be possible, there has to be a way for the operators to define the variant data. XML was chosen as the format of data definition, as it enables having structured data definitions. There are also readily available parsers for XML in S60 Platform. Also it would be quite easy to implement a tool for generating the XML data definitions or even to modify existing tools used in customizing other parts of the phone software into producing the new XML data definitions. As the XML structures are not verified in the implementation by using an XML schema, but rather during parsing, an XML schema for the data definitions was not created in this project.

The first step in designing the data definitions was to decide what kind of data structures were needed for branding and what kind of identifiers and other parameters the data needs. In this implementation, one set of brand data is thought to be for a single application. Of course this does not limit several applications for using the same set of data, but generally the set of brand data needs an identifier and it is in this case the application identifier.

The application identifier is a string identifying the logical owner of the set of brand data. As one single application can have several different sets of brand data, the brand data needs also an identifier for the brand itself. Also as the requirements define that the brand data has to be language variated the set of brand data needs an identifier for the language variant. For versioning of the data there also has to be a version id.

For single brand data items two identifiers are needed. The identifier for the item itself is needed so that the item can be fetched. Also the type of the item has to be defined.

Figure 13 shows the structure of a brand data definition. Within one XML-file there can be several different brand definitions each enclosed within their own <*brand*> tags. Within the brand definition there are the general identifiers and brand element definitions.

```
<branding>
          <brand>
                      <application_id>example_app</application_id>
                      <brand_id>example_brand</brand_id>
                      <brand_language_id>01</brand_language_id>
                      <brand_version>1</brand_version>

                      <element type="xxx" id=”yy”>
                                  <element_value></element_value>
                      </element>
          </brand>
<branding>
```

Figure 13: The XML structure of brand data definition

In this implementation there are six separate data types defined for the brand data items. The element value can be a single data item or it can be a structure defining more complex data.

An example brand data file can be seen in APPENDIX 2: An example of a brand definition XML-file. The XML-file defines a brand with a simple integer-type element and a more complex list-type element, containing an integer-type element, text-type element, file-type element, and a bitmap type element. It is worth noticing how the bitmap element is linked to the file element with the tag *<bitmap_file_id>.* That means that the bitmap-element can be loaded from that particular file.

4.3.1 Text Element

Text type element is a simple id-value pair as seen in Figure 14. Text element data is treated as 16-bit data in the Branding Server and it has to be UTF-8 encoded. The reason for differentiating 16-bit text element data and 8-bit buffer element data is that in Symbian OS all user input is usually handled as 16-bit since 8 bits are not enough to hold all Unicode characters.

```
<element type="text" id=”text1”>
           <element_value>text element </element_value>
</element>
```

Figure 14: XML definition of a text element

4.3.2 Integer Element

Integer type element is also a simple id-value pair.

```
<element type="integer" id=”int1”>
           <element_value>123 </element_value>
</element>
```

Figure 15: XML definition of an integer element

4.3.3 Buffer Element

A buffer element is very similar to the text element. The only difference is that the buffer element is treated as 8-bit data within the Branding Server. The data itself can be anything, simple 8-bit text or even binary data.

```
<element type="buffer" id=”buffer1”>
           <element_value>somedata</element_value>
</element>
```

Figure 16: XML definition of a buffer element

4.3.4 File Element

The file element has a more complex structure as its value. The value is a path to the file from where the Branding Server can copy the file when installing the brand data. The handling of the file element in the client side API is described in Chapter 4.1.

```
<element type="file" id="file1">
          <element_value>
                <file_name>c:\private\12345678\import\file1.mbm</file_name>
          </element_value>
</element>
```

Figure 17: XML definition of a file element

4.3.5 Bitmap Element

The bitmap element is not a simple element. The value is a structure of five different values. For a bitmap element to be used as a brand element in Branding Server there has to be a branded file element installed to the server. So the first data element in the bitmap element value is the identifier of the file element installed to the server. The two following data elements are the bitmap identifier and bitmap mask identifier used for loading the bitmap from the file. The two last data elements are used for skinning support. The major and minor skin identifiers are used for loading the bitmap from a skin, if one is used.

```
<element type="bitmap" id="bitmap1">
          <element_value>
                      <bitmap_file_id>file1</ bitmap_file_id>
                      <bitmap_id>1234</bitmap_id>
                      <mask_id>5678</mask_id>
                      <skin_id_major>9876</skin_id_major>
                      <skin_id_minor>5432</skin_id_minor>
          </element_value>
</element>
```

Figure 18: XML definition of a bitmap element

4.3.6 Structure Element

A structure element is a complex structure, which can hold any number of any type of elements, even other structures. This element type enables branding of virtually any kind of data, especially lists of data.

```
<element type="list" id="list1">
          <element_value>
               <element type="integer" id="int1">
                      <element_value> 12345</element_value>
               </element>

               <element type="text" id="text1">
                      <element_value>some text</element_value>
               </element>
          </element_value>
</element>
```

Figure 19: XML definition of a structure element

4.4 Data Storage

The data storage is the internal representation of the data structures described in Chapter 4.3. A server has a directory structure under its private folder. There are separate directories for each application and under each application's directory there are directories for the brands for that application. Each language variant of a single brand is stored in a binary format file under those directories. The files included in the brand are stored under a separate folder under the folder of the brand. The folder structure is shown in Figure 20.

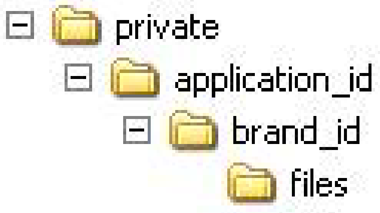


Figure 20: Brand data folder

Branding Server includes a versioning system which enables continuous usage of the brand data even when newer versions of the data are updated into the server. Brand data is contained in a single binary format file. If there are no clients using the data when an update of the data is installed the newer version simply replaces the old file. But if the file is in use by some client, a temporary file of a similar format is created and the client(s) using the data are informed that there is a newer version of the brand data available. Each client can then decide whether to take the newer version into use or not. The temporary data file will persist in the system until the old version is no longer in use, and then it will replace the old version.

There can be several temporary files, each representing a version of the data. If new clients start using the brand data while there are temporary files, the newest version of the data will be automatically selected. Whenever an older version is no longer in use it will be replaced by the newest version.

4.5 Architecture

This chapter describes the architecture of the Branding Server. There are two main components in the architecture; the server which contains most of the functionality of the Branding Server, and the client which is the implementation of the interface provided by the server.

4.5.1 The Client

In the client the main parts are four abstract interfaces and their implementations, a factory class which is used for creating the interfaces and the client implementation responsible for the communication between the client and server using the client/server framework. The class hierarchy of the client can be seen in Figure 21.

MBSAccess is the abstract interface, implemented in *CBSAccess*, for accessing brand data. It has separate methods for fetching each of the element types described in Chapter 4.3, a method for fetching several brand data elements at a time, and methods for registering and unregistering an observer for brand data updates. The observer, which uses the Observer-pattern [19], has to implement the abstract interface *MBSBrandChangeObserver*. *MBSUpdater*, implemented by *CBSUpdater*, is an interface used for updating the brand data. It can be used to install a new brand and update or replace an existing brand. *MBSElement*, implemented by *CBSElement*, represents the brand data elements within the Branding Server. *RBSClient* is the client implementation responsible for message passing between the client and the server.

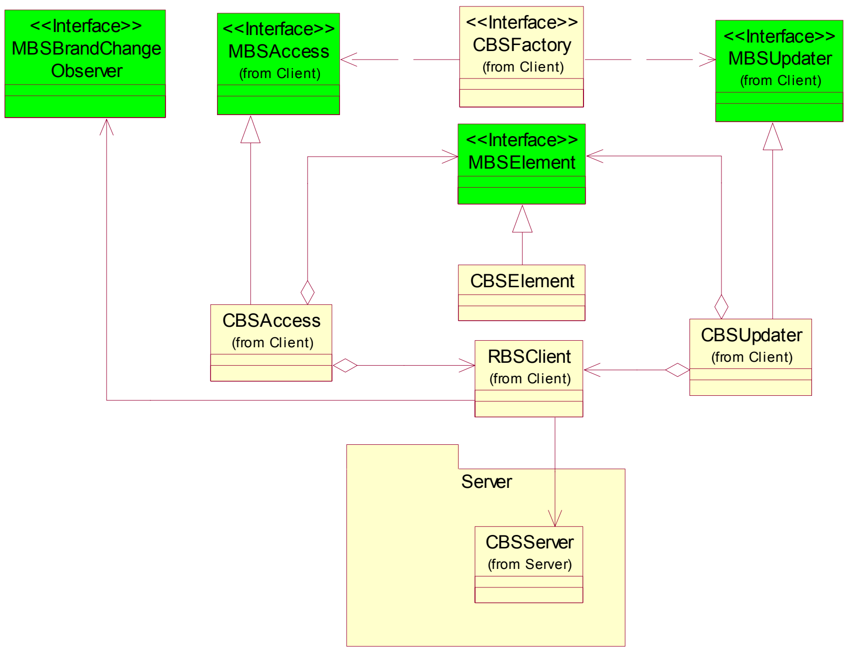


Figure 21: Class diagram of the client

4.5.2 The Server

CBSServer is the server implementation. It is responsible for creating a session for each client and for communication between the sessions. Most of the functionality of the server is contained in *CBSSession,* which is responsible for handling the messages sent by the client. For each client there is an instance of *CBSSession*. *CBSBrandHandler* is the handle to the brand data for each session. It is used for fetching the data from the actual binary format file.

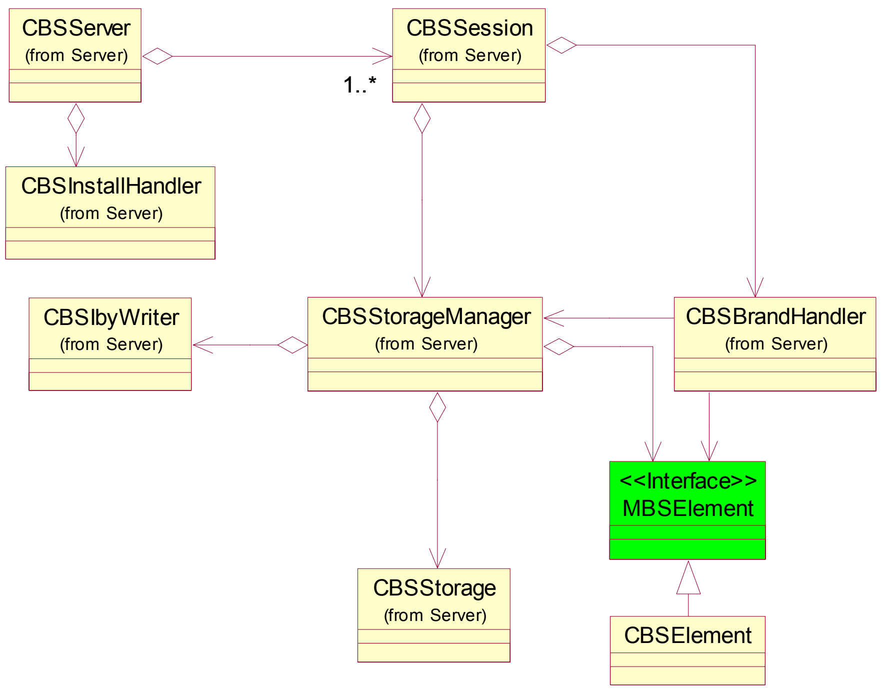

CBSServer (from Server)
CBSSession (from Server)
1..*
CBSInstallHandler (from Server)
CBSIbyWriter (from Server)
CBSStorageManager (from Server)
CBSBrandHandler (from Server)
<<Interface>> MBSElement
CBSStorage (from Server)
CBSElement

Figure 22: Class diagram of the server

All data elements are handled as instances of *MBSElement,* just like in the client. *CBSStorageManager* is responsible for handling the versioning of the brand data files and it uses *CBSStorage* to represent the brand data files. For installing and updating the brand data there is *CBSInstallHandler* and *CBSIbyWriter,* which is only used in ROM image creation phase for writing IBY files needed to install the data files into the ROM image. The class hierarchy of the client can be seen in Figure 22.

4.5.3 Relations to Outside Components

The key relationships between the Branding Server and the components in its environment are shown in Figure 23. The client uses Symbian OS control environment (CONE), and AknSkins and AknBitmaps from Avkon for bitmap skinning support. Avkon is a user interface layer provided by S60 platform for the underlying Symbian OS application. [20] The server uses file store from Symbian OS for the storage handling of the brand data files.

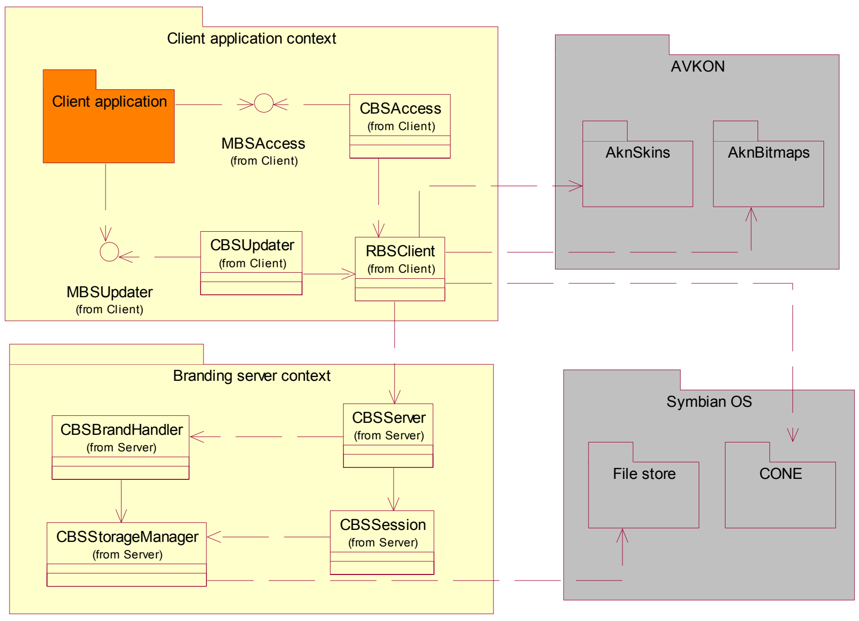


Figure 23: Logical module relations

4.6 Installation System

The installation system has two separate parts: the importing tool for creating brand data files from the XML, and the IBY-files for importing the data files into the ROM-image, and the installer tool which is used for installing brand data on the phone with SIS-packages. SIS-packages are software installation packages used by the Symbian OS.

The importing tool is used in the building phase of the software. It consists of an XML-parser, a client which uses the the *MBSUpdater*-interface for importing the data, and an IBY-file writer. The XML-parser parses the brand data definition XML-files into internal data structures and then uses the updater API of the Branding Server to convert the data into the binary data files. Once the data files are created, the IBY-writer writes the IBY-files needed for importing the data files into the ROM-image. The IBY-files need to be included in the client application's own IBY-file.

The installer tool handles the SIS-package installation and uninstallation of the brand data on the phone. The SIS-package copies the XML-files into the import-folder of the installer from where the installer parses them. The installer then uses the *MBSUpdater*-interface to install the data into the Branding Server. When the SIS-package is uninstalled, the installer uses the same interface to delete the brand data from the Branding Server. The installer can also be used for updating existing sets of brand data. The update can include changed values of the existing data elements or totally new elements.

4.7 Results

As the results of the project there is a working implementation of the Branding Server. Even though it was created for a specific application, it is generic enough to be used by any application with any type of brand data. Due to reasons outside the scope of the project, the integration of the Branding Server to the application it was implemented for was not conducted during the project. However, having tested the APIs and integrating the implementation into test applications, we can safely say that the requirements set for the project were met in a satisfactory level.

Branding Server **enables branding** an S60 application and sharing the brand data of the application with other applications. The server **supports multiple clients using the same brand data simultaneously** and also multiple clients using multiple sets of brand data in the same fashion. The implementation provides **a way of defining the brand data** in a user friendly format, **tools for installing the data in the ROM-image**, and for **updating existing data** on the phone. The server also guarantees **backwards compatibility** of the data with old clients if the brand element ids are not reused, meaning that if any id within a set of brand data always represents the same data there will be no data compatibility issues. Also, by having separate methods in the API for fetching each type of brand data elements, it is not possible for clients to crash trying to load wrong type of data elements, since if the id being fetched is representing a different type of data than what the fetch method is meant to be used for, the method will simply return an error code.

The Branding Server also enables **language variation of all brand data**, including files and bitmaps. It also provides **support for skinning** the bitmap elements and it follows the rules for bitmap loading listed in the requirements in Chapter 4.1. It should be noted that as the skinning support of S60 platform requires the control environment of the Symbian OS application framework to work, then skinning support of Branding Server will not work if the client-side of the Branding Server is run in a process which does not have access to the control environment of the operating system. Generally all UI applications have access to the control environment and regular executables do not. If the Branding Server API methods for fetching bitmaps are used in a process with no access to the control environment, they return the original bitmap and do not even try to fetch the skinned bitmap. But if the control environment is available, the bitmap is searched from the active skin, if so required.

4.8 Further Development

The current version of Branding Server is just the first delivery of it. It provides the basic functionality for branding, but is missing some of the features originally planned for it. Several helper methods were planned for the *MBSElement*-interface, e.g. an iterator for easy browsing through the items of a list type element. Also it was planned that the installing system of the Branding Server would be integrated into the tool which is used for customizing a ROM-image. As an alternative to this approach, an own tool could be implemented for the XML-data creation.

A caching solution and indexing of the file storage were planned for the server side, but they were both dropped from the project due to schedule pressures. Caching fetched items into memory might make the memory consumption too high, but indexing the file storage and then caching the index locations of fetched identifiers might bring considerable performance improvements, especially if the client using the Branding Server loads the same identifiers repeatedly.

In the long term, development plan could be to make the architecture of the server part totally different. An ECOM-plugin [21] based server, where each set of brand data would be a plugin to the server, could be the direction to go to. ECOM is a client/server architecture which provides services for handling plugins. Using this architecture might make data importing and installation a bit more complicated operation, having to compile the plugin for each set of brand data, but it could help make the versioning and updating the data easier.

5 CONCLUSIONS

The purpose of this thesis was to find a way for branding an S60 application or possibly even the whole phone with all the software in it. The work introduced the Symbian Operating System and the S60 Platform along with some of the ways it provides for varying software. Some aspects of branding in general were also briefly introduced. The practical part of the thesis was a project that presented a solution for branding an S60 application and even comprehensive branding of the whole phone.

There is a growing need for branding applications on S60 phones. Existing methods of varying the software provided by the platform are not enough to cope with the requirements for branding and with the limitations set by Symbian platform security. New solutions of varying both the graphics and functionality of the software are needed. These solutions have to be easy to use and they have to enable easy installation of the variant data into the ROM image so that neither developing the software nor installing the ROM image into the phones on the production lines would be slowed down.

Defining the branding data in XML-format enables existing tools for customizing the phone software to be easily modified to produce the new data definitions. Installation system, which at the same time produces the branding data files to be installed into the ROM-image and the needed IBY-files to enable the installation, is a clear benefit. And as this installation system can be integrated into the build tool chain of the platform, it makes the brand data creation and installation automatic with no need for manual installation of SIS-packages or other ways of installing software after ROM-image flashing.

Using the client/server framework solves the problems with sharing the brand data between applications and it also provides security through the capability-based access control. Client/server framework does add some overhead to the data fetching, mostly due to the amount of processor time the context switching between client and server processes requires, but with good design of the client application using the server, e.g. fetching multiple items with the operation, this can be reduced.

REFERENCES

[1] Marcus, A., Branding 101, Interactions 11, 5 (Sep. 2004), 14-21, 2004, DOI= http://doi.acm.org/10.1145/1015530.1015539

[2] Jézéquel, J., Reifying Variants in Configuration Management, ACM Trans. Softw. Eng. Methodol. 8, 3 (Jul. 1999), 284-295, 1999, DOI= http://doi.acm.org/10.1145/310663.310668

[3] Digia, Inc.: Programming for the Series 60 platform and Symbian OS, Johan Wiley and sons Ltd., 2002

[4] Nokia Corporation: S60 Platorm: Symbian Platform Security FAQ, Version 2.0, 2006, [Internet], available: http://www.forum.nokia.com/info/sw.nokia.com/id/81cc0db0-6b3b-4510-b108-802d6215363a/S60_Platform_Symbian_Platform_Security_FAQ_v2_0_en.pdf.html [referenced 20.03.2007]

[5] Nokia Corporation: Series 60 Developer Platform: Application UI Customization, Version 1.0, 2004, [Internet], available http://www.forum.nokia.com/info/sw.nokia.com/id/9bb2a055-9599-42fe-b2ae-b2e8dd0e878c/Series_60_DP_App_UI_Customization_v1_0_en.pdf.html [referenced 20.03.2007]

[6] Nokia Corporation: S60 Platform: System Initiated Events, Version 2.1, 2006, [Internet], available http://www.forum.nokia.com/info/sw.nokia.com/id/904fbab3-f692-42fb-adb8-c169301ed0fc/S60_Platform_System_Initiated_Events_v2_1_en.pdf.html [referenced 20.03.2007]

[7] Laakso, H. & Kauppakari Oy, Brandit kilpailuetuna, miten rakennan ja kehitän tuotemerkkiä. Gummeruksen kirjapaino Oy, 1999, ISBN 952-14-0088-9

[8] Klein, N.: No Logo, tähtäimessä brandivaltiaat, WS Bookwell Oy, 2001, ISBN 951-0-26275-7

[9] Rondeau, D. B., For mobile applications, branding is experience, Commun. ACM 48, 7 (Jul. 2005), 61-66, 2005, DOI= http://doi.acm.org/10.1145/1070838.1070867

[10] Aaker, D. A. & Joachimsthaler, E., Brandien johtaminen, WS Bookwell Oy, 2000, ISBN 978-951-0-24994-9

[11] Symbian Ltd.: Symbian OS Version 9.1 Product description, Revision 1.1, 2005, [Internet], available http://www.symbian.com/symbianos/releases/v91/functionaldescription.html [referenced 20.03.2007]

[12] Timo Rouvinen, Dynamic application development in Symbian OS, M.Sc. thesis, Lappeenranta University of Technology, 2004

[13] Symbian Ltd.: Symbian OS v9 Security Architecture, Revision 2.0, 2005, [Internet], available http://www.symbian.com/Developer/techlib/v9.1docs/doc_source/guide/N10022/SGL.SM0007.013_Rev2.0_Symbian_OS_Security_Architecture.doc [referenced 20.03.2007]

[14] Nokia corporation: Symbian OS: Overview To Security, Version 1.1, 2006, [Internet], available http://www.forum.nokia.com/info/sw.nokia.com/id/5e713b29-fe0e-488d-8fc6-b4dd1950f3c2/Symbian_OS_Overview_To_Security_v1_1_en.pdf.html [referenced 20.03.2007]

[15] Nokia Corporation: S60 Platform: Using DBMS APIs, Version 2.0, 2006, [Internet], available http://forum.nokia.com/info/sw.nokia.com/id/e0a66f34-092a-4a52-8003-

6bbc3aa83c8f/S60_Platform_Using_DBMS_APIs_v2_0_en.pdf.html [referenced 20.03.2007]

[16] Nokia Corporation: S60 Platform: Introductory Guide, Version 1.2, 2006, [Internet], available http://www.forum.nokia.com/info/sw.nokia.com/id/b8613f0a-21c2-4dff-a828-d1bc9c4987c9/S60_Platform_Introductory_Guide_v1_2_en.pdf.html [referenced 20.03.2007]

[17] Kolb, R. and Muthig, D, Making Testing Product Lines More Efficient by Improving the Testability of Product Line Architectures, In Proceedings of the ISSTA 2006 Workshop on Role of Software Architecture For Testing and Analysis (Portland, Maine, July 17 - 20, 2006). ROSATEA '06. ACM Press, New York, NY, 22-27. DOI= http://doi.acm.org/10.1145/1147249.1147252

[18] Tasker, M. et al.: Professional Symbian Programming: Mobile Solutions on the EPOC platform, Wrox Press Ltd., 2000, ISBN 1-861003-03-X

[19] Gamma, E. et al: Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley Professional, 1995, ISBN 0-201-63361-2

[20] Nokia Corporation: S60 Platform: Application Framework Handbook, Version 2.0, 2006, [Internet], available http://www.forum.nokia.com/info/sw.nokia.com/id/8cc94f4a-cbd9-4f82-b4d1-790b7569733e/S60_Platform_Application_Framework_Handbook_v2_0_en.pdf.html [referenced 25.3.2007]

[21] Nokia Corporation: S60 Platform: Ecom Plug-in Architecture, Version 2.0, 2007, [Internet], available http://www.forum.nokia.com/info/sw.nokia.com/id/53a369e8-

14c7-4f52-9731-577db4e0d303/S60_Platform_ECom_Plug-in_Architecture_v2_0_en.pdf.html [referenced 23.3.2007]

APPENDIX 1: System capabilities of Symbian OS

Tcb	Used by the Trusted Computing Base only, it gives access to the location where executables are stored and therefore they can change their capabilities. Only the Symbian OS kernel, the file server (including the loader), and the software installer are granted this privilege.
CommDD	Grants direct access to all communication device drivers, including phone baseband software.
PowerMgmt	Grants the right to kill any process in the system, to switch the machine into standby state, wake it up again or power it down completely.
MultimediaDD	Grants direct access to all multimedia device drivers.
ReadDeviceData	Grants read access to device, network operator, and phone manufacturer confidential settings or data.
WriteDeviceData	Grants write access to settings that control the behaviour of the device.
Drm	Grants access to digital rights-protected content.
TrustedUI	Grants the right to create a trusted UI session, and therefore to display dialogs in a secure UI environment.
ProtServ	Grants the right to a server to register within the protected name space – to limit scope for malware to spoof sensitive system servers.
DiskAdmin	Grants access to disk administration operations that affect more than one file or one directory (or overall filesystem integrity/behaviour, etc).
NetworkControl	Grants the right to modify or access network protocol controls, in a way that might affect more than one client

	application or transport connection / session.
AllFiles	Grants read access to entire file system. Grants write access to other processes' private directories.
SwEvent	Grants the right to generate software key & pen events and to capture any of them regardless of the foreground status of the application.
SurroundingsDD	Grants access to device drivers that provide input information about the surroundings of the device.

APPENDIX 2: An example of a brand definition XML-file

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<branding>
  <brand>
    <application_id>
      APP_ID
    </application_id>
    <brand_id>
     BRAND_ID
    </brand_id>
    <brand_language_id>
      99
    </brand_language_id>
    <brand_version>
     1
    </brand_version>
    <element type="integer" id="integer1">
            <element_value>
             10
            </element_value>
    </element>
    <element type="list" id="list1">
      <element_value>
        <element type="integer" id="int1">
            <element_value>
           12345
            </element_value>
        </element>
        <element type="text" id="text1">
            <element_value>
          example-text
            </element_value>
        </element>
        <element type="file" id="file1">
            <element_value>
             <file_name>c:\test-file.mbm</file_name>
            </element_value>
        </element>
        <element type="bitmap" id="bitmap1">
                        <element_value>
                        <bitmap_file_id> file1</bitmap_file_id>
                        <bitmap_id>      1234</bitmap_id>
                        <mask_id>        1234</mask_id>
                        <skin_id_major>  1234</skin_id_major>
                        <skin_id_minor>    1234</skin_id_minor>
            </element_value>
        </element>
      </element_value>
    </element>
  </brand>
</branding>
```