Building consumer products with open source

Ari Jaaksi, Nokia

Introduction

Nokia launched its first Linux and open source based product, the Nokia 770 Internet Tablet, and the www.maemo.org community web site in 2005. Internet tablets are a new kind of mobile internet devices and the community web site supports application development on these devices. These initiatives provide Nokia with an open source based software platform for handheld devices. This article discusses our experiences of using Linux and open source at Nokia.

1. The product

The Nokia 770 Internet Tablet, illustrated in Figure 1, is a handheld device optimized for internet. It can access internet services, such as email, browsing, media, internet telephony, and chat in WiFi hotspots or over a cellular network.

The Nokia 770's dimensions are 141 x 79 x 19 mm and it weights 185 g. It has a touch screen with an 800x480 pixel resolution. The product is build around the Texas Instruments OMAP 1710 platform with 128Mb Flash and 64Mb RAM memory. Additional data can be stored on a MMC card. The connectivity is provided over an 802.11g wireless network, and over a Bluetooth to a cellular network using a phone as a modem. The street price is approximately $350.

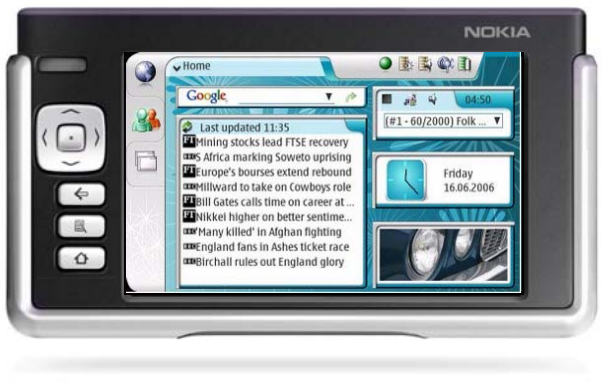

NOKIA
Home
Google
Last updated 11:35
Mining stocks lead FTSE recovery
S Africa marking Soweto uprising
Europe's bourses extend rebound
Millward to take on Cowboys role
Bill Gates calls time on career at ...
Nikkei higher on better sentime...
'Many killed' in Afghan fighting
England fans in Ashes ticket race
Birchall rules out England glory
04:50
(#1 - 60/2000) Folk ...
Friday
16.06.2006

Figure 1: Nokia 770 Internet tablet

2. The architecture

We used open source extensively in the creation of the Nokia 770. We favored components that were developed by active communities and already used by many users. Thus, instead of an embedded version of a component, we rather used a mainstream desktop component when possible.

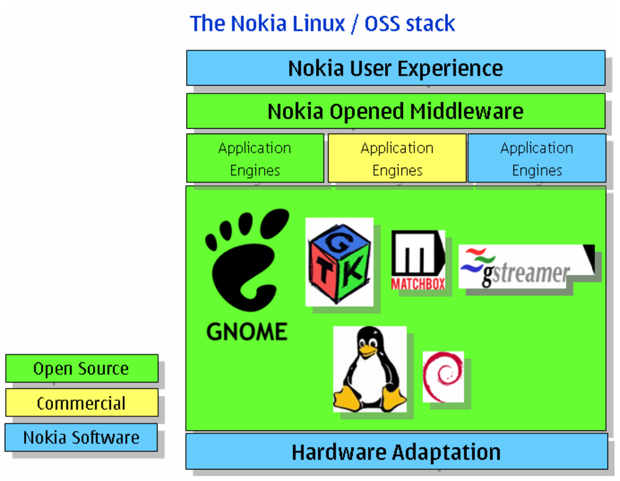


Figure 2: The Nokia 770 software architecture

Figure 2 illustrates our software architecture. The majority of the code is licensed under open source licenses. In addition, we have closed components developed by Nokia or third parties. An example of such component is the Real audio & video plug-in. We keep some limited parts of the software that are very close to our hardware close. Examples of such components are the boot loader and battery charging.

End users can flash a new firmware into Nokia 770 to get new features, improved performance, and bug fixes. Nokia can thus develop and improve software in customers' devices.

2.1 Kernel

Our software is currently based on Linux 2.6 kernel. We source the kernel directly from the kernel.org. We do not to use any commercial Linux distributions, but follow the Debian [Debian 2006] distribution closely. As an example, we use Debian package management. This provides us with a standard way of managing component dependencies.

Working directly at the kernel.org provides us with the latest technology and direct access to community work. It allows us to flexibly use our own staff, subcontractors, and open source communities all working on our own hardware.

2.2 Middleware

The big touch screen of the Nokia 770 enables rich user experience. We therefore use the GNOME [GNOME 2006] desktop environment as the basis of our application framework. GNOME has a vibrant community and Nokia is well integrated with the project. As a part of GNOME environment, we use the GTK+ [GTK 2006] multi-platform toolkit for creating graphical user interfaces.

We created a new desktop and the user interface style optimized for internet tablets, as illustrated in Figure 1. Our desktop is based on GNOME and we call it the Hildon Application Framework. It includes, among other things, a task navigator to manage application executions, a home view to embed different plug-ins and a status bar to communicate device status changes. In addition, we provide new widgets and theming modifications on top of the GTK+ to match the Nokia user interface style and various services for third party applications to integrate with the Nokia platform.

We use the Matchbox window manager. It is a lightweight window manager for X11 supporting a PDA style windowing. D-BUS is our message bus system for applications and libraries to talk to one another. We use D-BUS for system notifications between applications, for separating applications user interfaces and engines, and for launching applications from our task navigator.

2.3 Applications

Our applications typically consist of two components. Application engines provide the application functionality, and application user interfaces provide the visible look and feel of applications. This two-level architecture allows us to develop and manage functionality and visual appearance separately, similar to the MVC approach [Krasner and Pope 1988, Jaaksi 1995]. We can, for example, change the engine under the visual UI implementation if we need to.

Application engines are either from open source, such as our email engine, closed third party components, such as the handwriting recognition engine, or developed by Nokia. Various application user interfaces that provide the Nokia user experience are closed source.

3. Developing the product using Linux and open source

3.1 Selecting the core components

In the beginning of the product development, we analyzed different technologies and architectural solutions. When selecting open source components we took technical, community, roadmap, and legal and IPR aspects into account.

We analyzed the technical suitability of various components and subsystems. A selected component needed to fulfill our functional requirements and be suitable for our hardware specifications. The component also needed to be of good quality and mature enough for a consumer product.

We then analyzed the communities developing the components or subsystems. We wanted to integrate ourselves with vibrant communities to ensure that the selected subsystem would develop further over time. The more active the community, the better.

We analyzed the roadmaps and future plans for the selected technologies. We wanted to ensure that the goals of the development communities would meet those of ours. This all happened through face-to-face discussions in open source conferences, over direct email discussions with key developers, and with participation of discussion in community mailing lists and other such forums.

Finally, we analyzed the legal and IPR aspects of the components under consideration. It was important for us that the open source components used have been licensed under proper licenses and have clear copyright and licensing information attached to them. We also wanted to select components that do not lock us into one vendor, for example through a mandatory copyright donation or a dual licensing model.

It is important that certain components are licensed under an open source license, such as LGPL, that allows us to integrate also proprietary components on our platform. We need to offer the best possible user experience to our customers. In some cases, such as with audio and video, that requires commercial closed components.

For the major components and subsystems we did not have too many options to choose from. For example, the only true graphical environment alternatives were Qt and Gtk+ [Trolltech 2006, GTK 2006]. We selected Gtk+ for it is developed by a vibrant multi-polar community with no single company dominance. It is therefore easy to contribute our changes to Gtk+ on the basis of general usefulness and technical merit only. Also, Gtk+ is licensed under LGPL and that allows us to mix proprietary UI elements without a dual commercial license.

3.2 Creating software as a part of communities

We integrated tens of unmodified open source components on our platform. We also sponsored the enhancements of many existing open source components to. Finally, we developed new components from scratch and open sourced them.

We used existing open sourced components such as the gnuchess chess game engine, bzip2 data compressor, id3lib for manipulating ID3v1 and ID3v2 tags in digital audio files, and many more as such, whenever that was possible. The bigger the component or subsystem was, the better. When possible, we reused an entire subsystem and subsystem architectures, such as GNOME [GNOME 2006] and Debian [Debian 2006]. Instead of separate components we reused architectural blocks that already integrate several independent components.

In several cases we sponsored the development of existing components, such as the Linux kernel, D-BUS, GNOME-VFS, GTK+, GStreamer, and OBEX to make them better meet our requirements. The additional work was needed especially in the areas of UI and usability, power management, performance, and memory management. Our engineers worked directly with communities participating development projects. We also hired and asked developers within the communities to enhance components based on our needs.

As an example, we sponsored the development of the D-BUS [D-BUS 2006] message bus system. We selected D-BUS because it addressed our technical requirements well at the outset. In addition, there was an active community around D-BUS, and we and the community had very similar goals. We believed that we can become a part of the community and there is no need to branch the development for our needs. We believed we can get our requirements and contributions accepted to the project.

We hired key developers from the D-BUS community to work for us. We contributed code and participated the development of D-BUS through these developers. In addition, we performed a lot of testing that helped in reaching the needed product quality. Finally, we promoted the use of D-BUS in various conferences and articles. The more developers come and use D-BUS the better for us and everybody else. Having a good interprocess communication mechanism available benefits everybody!

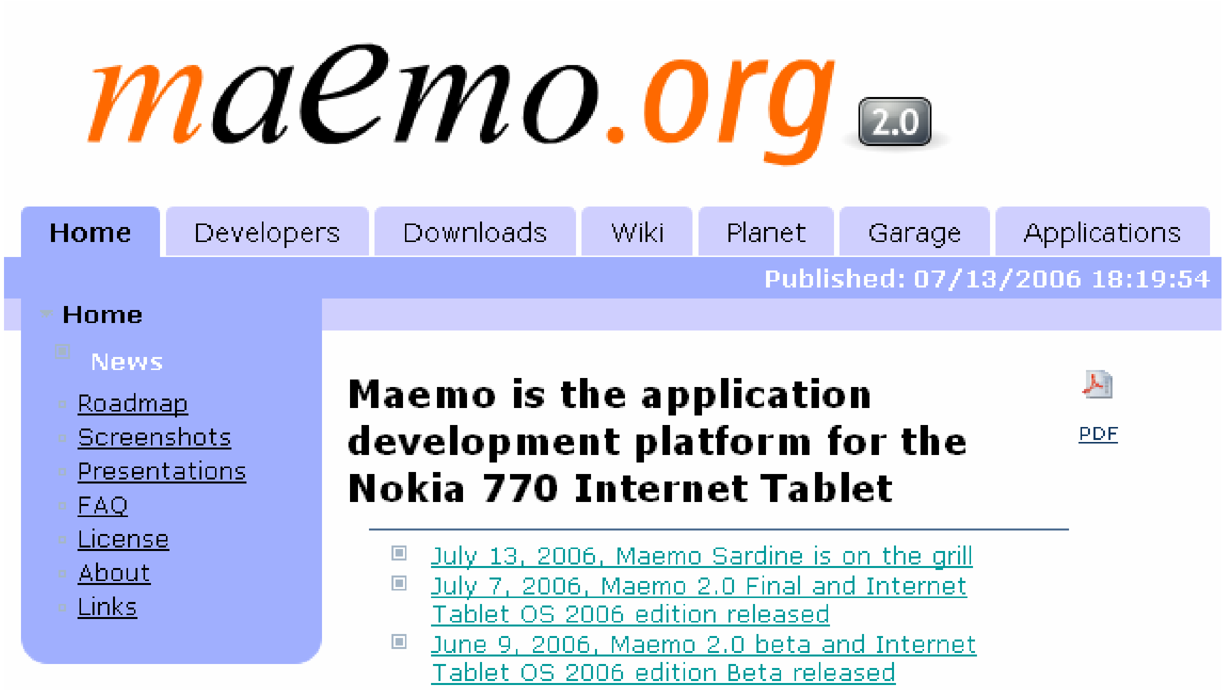
We open sourced also new components and subsystems that we had developed. A good example of such component is the Hildon application framework. We soon realized that after open sourcing our own code, we need to ensure that our developers continue to work with the open sourced components. We must be able to continue support the code in open source so that the code will meet our future needs, too. Just releasing code with no plans to develop it further won’t benefit us.

As a recent development, we have opened the development of our application framework and selected other middleware components at the maemo Sardine [Sardine 2006]. Open middleware development enables application developers to follow the latest changes in our code so they can test their applications against the latest changes, update them as a result of any API changes, and pilot the latest additions to our software. This open development also allows anybody to participate in the development of the middleware code and see where it’s heading at. This all is available before a stable release of the software for the end-users.

A lot of the code for our yet to be released future products is now available and developed jointly with open source developers. We believe this approach is unique and provides an access for developers to participate the creation of the future Nokia devices.

3.3 The developer platform – maemo

We utilize the open model in the creation of our products. In addition, we want others to execute their software development projects on our software and devices.



Maemo is the application development platform for the Nokia 770 Internet Tablet

- July 13, 2006, Maemo Sardine is on the grill
- July 7, 2006, Maemo 2.0 Final and Internet Tablet OS 2006 edition released
- June 9, 2006, Maemo 2.0 beta and Internet Tablet OS 2006 edition Beta released

Figure 3: The maemo developer platform

For that reason we manage the www.maemo.org web site, illustrated in Figure 3. Maemo is an open source development environment for Nokia 770 Internet Tablet, targeted to open source developers and innovation houses. It provides tools to develop and share your own applications for Nokia 770. We opened the maemo site months before the devices were commercially available to provide an early access to developers.

The key features of the maemo include a test and debug environment on x86, flashing tools, a developer root file system, the Hildon Application Framework and UI, the Scratchbox cross-compilation toolkit, developer documentation, sample applications, mailing lists, wiki and planet for discussion, announcements and support, and bug reporting system. In addition, the site provides a forum to share applications and code for the Nokia 770 internet tablets.

4. BENEFITS OF OPEN SOURCE

We can conclude that it is very beneficial to use open source in consumer product creation. Our experiences are limited, though. We developed only one single software platform and product category within one company. On the other hand, our experiences include all the phases starting from initial screening to selling the devices, participating with many other companies and open source projects, and offering upgrade software for end users.

4.1 Cost savings

The biggest cost savings came from the utilization of already available components. We utilized several free components and subsystems as such, with no modifications.

We also improved several components to better meet our requirements. Such improvement is cheaper than creating the needed functionality from scratch.

Some 2/3 of the code of the Nokia 770 is licensed under an open source license. These components made it possible for us to build the software cheaper than we could have done using closed and proprietary technologies.

4.2 Quality and flexibility

Our code comes from different sources. The majority of the code originates from open source projects. Some code, such as the application user interfaces, is developed by us. Some code, such as the browser engine, is provided as binaries to us. So we have no access to the code.

If we compare the code from open source to the code developed by us, our conclusion is that open source is of better quality. We have more bugs and problems in the Nokia developed code. This is only natural because the majority of the Nokia code is build from scratch and is thus very young. Open source code, on the other hand, has mostly been used by others already. They have fixed the most severe errors already before we started to use the code.

If we compare open source code to commercial components used on our platform, the quality difference is not that obvious. The commercial components have typically been used by others, too. That has improved their quality.

Open source is flexible when we needed to fix a problem or change functionality. We often requested bug fixes or modifications to the commercial closed components on our platform. If the vendors didn't have the capacity or will to fix the problem on time, we had few options. We could not fix problems ourselves because the companies using closed source didn't want us to access their source code. With open source components, though, we fixed bugs yourself, hired somebody else to fix them, or worked with the communities for the modifications. We thus had many options available, and in most cases we managed to fix the problems at hand. The free access to the code and to the developers improved the quality of open source originated components within the final product.

4.3 Speed and time

It is clear that using available subsystems, such as the Linux kernel or the GTK+ toolkit saves time and resources. Actually, developing an operating system and middleware was never even an option for us. In reality, we had two different options for the Nokia 770 internet tablets: either use an existing commercial and closed operating system and middleware, or then use an existing open source operating system and middleware. We used the open approach because we believed it saves time and money, and enables us to freely mold and shape our products according to the market needs.

We use widely known components and architectures. Such familiarity speeds up our development. Developers, even new-comers, know the technologies in use. Packaging technologies, such as the Debian packages, or interprocess communication technologies, such a D-BUS, provides the backbone for our integration. Such software architecture is understood by all developers.

We believe that we saved time in integrating open source components to our architecture compared to using closed ones. Familiar subsystems, architectures, and free access to the source code made integration fast.

4.4 Software licensing

Software licensing is often a complicated and time consuming process. It requires a lot of negotiations between the licenser and the licensee. Based on our experience, an average in-licensing process for a software component takes 6 – 12 months. It is only then when you know if you can really use the component in your project or whether you need to find other solutions.

Licensing is simpler with open source. The licensor has already done the most of the due diligence work in advance. In most cases, the licensor keeps the copyright, doesn't give any exclusivity, and gives the licenser full right to use the component any way needed. At the financial side, licenser has decided that the code is free, but the licenser may offer support or engineering services for money. Nothing prevents the licensor to go elsewhere for help, if so needed.

All the source code is available for the licensor to study and evaluate. The licensor can assess the community, companies, and available hackers supporting the technology in question. Also, the licensor can talk to others without worrying about trade secrets.

Open source simplifies and accelerates software licensing, and reduces technology and quality risks. Instead of negotiation for months, the technical work can start immediately.

4.5 Available developers

We often analyzed the code of a developer or a subcontractor before hiring. If a developer or a subcontractor has submitted code to an open source project, its quality is easy to verify. Also, it is important to study the level of involvement the candidates –both individuals and companies- have on the communities and components in question.

Linux and open source have created a common vocabulary, architectural reference, and common tools for software development. Almost all developers graduating from universities are familiar with Linux and open source, and many know GNOME, Debian and other such technologies. This all simplifies the recruiting process as well as new-comer introduction. There is no need to arrange long training session for new developers on proprietary tools and architectures.

4.6 Road mapping and future

A roadmap of an open sourced component or subsystem is typically discussed openly, and is open for contributions. Some may claim that that utilizing open source would leave a company with no control over the used technology. This is not true. We influence the development of relevant technologies through the community work. Actually, things are better with open source than with closed components.

In the open source, decisions are done openly. We join technical discussions and roadmapping process. We contribute our ideas, submit our code, and discuss requirements to influence the direction of a component or technology. We are not only a licensor but also a licensee at the same time!

The choices are more limited with closed source commercial components. Companies developing closed source components typically decide themselves about the future of their technology. They may choose to reveal parts of they plans. They may choose to take external input into account. But, unlike in the open source, you cannot participate yourself and contribute in an open fashion.

4.7 Open source and confidentiality

We worked intensively with communities already before we announced the Nokia 770 Internet Tablet. Open source approach requires openness and information sharing during development. A high publicity launch, on the other hand, is the way to introduce consumer products to the public and you do not want to reveal the products before the launch date. There is thus a potential conflict between the open source openness and product launch secrecy.

The credentials, work, and history of open source hackers are open for everybody to see. The hackers typically want to work with interesting things also in the future. Therefore, they don't want to become famous for jeopardizing somebody else's project and misusing their trust. Thus, openness and open source can actually be much stronger bond than any NDA or monetary sanction one can put on an individual or a company.

Based on our experiences, we can combine open communication and product confidentiality. We had no information leakage prior to the commercial product announcements, although we had had tens of developers working on the software with us. For some of the developers, we had told very detailed information about the forthcoming product. Developing products in open source and yet maintain the confidentiality of the product plans and roadmaps was possible for us.

5. CHALLENGES

The benefits of using open source are clear. However, we also experienced some challenges when utilizing open source in product development.

5.1 From hacking to stabilizing

In the early stages of the development, we worked closely with communities, individual hackers, and hacker companies. We made core technical decisions and outlined the software architecture. A lot of ideas were discussed, a lot of trials were made, and a lot of hacking was taking place.

Soon, we froze our requirements to get things focused. The requirements were allocated to individual engineering teams. The teams continued the work in a close collaboration with external parties to add features, and enhance components to meet our needs

We have an internal milestone when we expect all software functionality to be implemented. We predict when the software is ready for shipping to synchronize marketing activities, and reserve a factory production line. At this milestone the system testing can run all test cases. All features are implemented, but the system is still unstable and buggy. From that on, all effort is put into bug fixing and stabilization of the system

At this milestone, the whole organization moves from hacking and development to integration and stabilizing. While hacking and development happens around independent components within teams, the integration and stabilizing happens around the entire software stack and between teams. We shift from a component view to a system view of software development.

Based on our experience this step is more radical with open source than with closed proprietary code. The open source culture is very much for trials, hacking, innovation and other creative aspects of software development. Meeting deadlines, dropping your latest crazy idea, and making compromises to gain stability are not what many open source communities or developers naturally do. In addition to us, the Linux project, Debian, and others seem to have difficulties making a final good quality release on time [Glance, 2004], [Brockmeier 2005]. In that sense our distro is no different that others'.

In the current projects we try to tackle this challenge by making the move from the hacking to stabilizing more apparent and strict. It seems we must make the change very explicit in our process, and enforce it even more than in some conventional product development projects.

5.2 Architecture management

Open source requires that you manage your architecture not only form the conventional 4+1 point of view [Kruchten, 1995] but also from the legal and IPR point of view. As some components are available only as closed source components, we need to mix open source and proprietary code. This mixing calls for proper architecture management to manage different licensing rules within the product code.

We manage the legal and IPR status of each software component. It is not enough to manage the architecture in terms of the development time API compatibility or run time performance, but we also need to manage the mix of various open source and closed source licensing rules. This is a complex task because not all licensing terms are compatible or clear. This is an additional job that we need to do when developing products based on open source.

5.3 Additional investments

Using open source code effectively requires community participation. It is sometimes possible to use a component as such. Such participation is almost free of charge. In many cases, though, we work with communities to enhance components and develop them further. Such participation requires extra resources.

We have open sourced individual components and participated some development with no clear benefit for us. We have either been left alone to develop the component, or our needs have not been taken into account when developing a component further. In these cases the joint open source development didn't happen or it didn't benefit us. We therefore now observe individual projects and try to identify when the open source investment pays off and when it doesn't.

6. SUMMARY

We have created a consumer device, the Nokia 770 Internet Tablet, utilizing open source software. Our experiences demonstrate that open source technologies and development model suit very well for such devices. We created the product in shorter time and with lesser resources that we have managed to develop other products utilizing proprietary software. In essence, open source offers time and cost savings in a form of readily available components and subsystems, available developers, and effective development model.

Open source doesn't solve all the problems, though. As a device manufacturer, we alone are responsible for the quality of the end product. We must therefore utilize all quality and software engineering mechanisms to achieve the needed quality. One cannot skip specification, integration, testing, and documentation, for example. In addition, open source introduces certain new requirements, such as community interaction and legal and IPR management.

Open source doesn't make software development free or easy. It provides effective tools for product creation. Combining these new tools, such as community involvement, and utilization of open components, with more traditional software and product engineering practices is a good mix.

References

GNOME 2006: http://www.gnome.org/

Debian 2006: http://www.debian.org/

Krasner and Pope 1988: G.E. Krasner, S.T. Pope. 'A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80'. *Joop*, August/September, 1988.

Jaaksi, 1995: A. Jaaksi (1995). 'Implementing Interactive Applications in C++', in *Software Practice & Experience*. Volume 25, No. 3, March 1995, pp 271-289.

Trolltech 2006: http://www.trolltech.com/

GTK 2006: http://www.gtk.org/

D-BUS 2006: http://www.freedesktop.org/wiki/Software/dbus

Sardine 2006: http://repository.maemo.org/sardine/.

Glance, 2004: http://www.firstmonday.org/issues/issue9_4/glance/index.html

Joe 'Zonker' Brockmeier, 2005, http://lwn.net/Articles/127031/

Kruchten 1995 : Kruchten, P.B. The 4+1 View Model of Architecture. In IEEE Software, November, 1995: 42-50