Our man at Nokia

Amateur astronomer and KDE hacker **Graham Morrison** meets amateur astronomer and Qt hacker Benoit Schillings.

Check it out, KDE fans: we're not biased against you! Benoit Schillings is the Chief Technologist at Qt Software. It's up to him what features should and shouldn't be make it into *Qt*, and hence, KDE. We questioned him about *Qt*, why it's so darn hard to program in and why the Greenphone could never make or receive a single a phone call.

Linux Format: What is your role at *Qt* software exactly?
Benoit Schillings: My role is chief technologist, which is always interesting because my job is to be right about what the technology will be like in the future – that's probably the best way I can describe it. You figure out what is going to be the best way in two years to get the partners, device manufacturers, and so on happy with our solution. So we're spending a lot of time with customers, spending quite a lot of time at events like this, spending time with developers to get a picture of where we're going to be in two years' time.

LXF: And has that role changed much with the acquisition by Nokia?
BS: It's a bit early to say exactly how the role is going to evolve. But I do find that Nokia is a company that really thinks in the same kind of horizon – you have short-term, medium-term and long-term horizons, and I do find that first of all they know a lot of things and there are lots of very interesting brain-teasers. What the role of *Qt* will be in the context of Nokia is a very interesting question, so I'm having a blast.

LXF: So presumably it's increasingly embedded?
BS: Not especially you know. I think that trying to split the market between embedded and non-embedded is probably a dangerous path. First of all because you see that capability or architecture becoming more similar, partly because people want to be able to take their skills and experience and apply themselves across a range of devices.

People want to go and learn *Qt* and apply it to embedded, to mobiles, to PCs, which is something that is quite interesting for developers. They do not need to use as big a part of their brain in order to be able to learn all those different skills. The other aspect is that I think we get a bit

stuck adding devices in our life if they do not integrate properly with what we already have. I mean, who wants another system that has to be backed up? Who wants another set of configuration tools? So I think that more and more for an end user to have more devices in their life you need to get those devices to have a lot of commonality, and that's the place where the cross-platform framework has a very big role to play. Everybody complains about converging but nobody does anything about it, and I think what is needed is a cross-platform framework combined with standards.

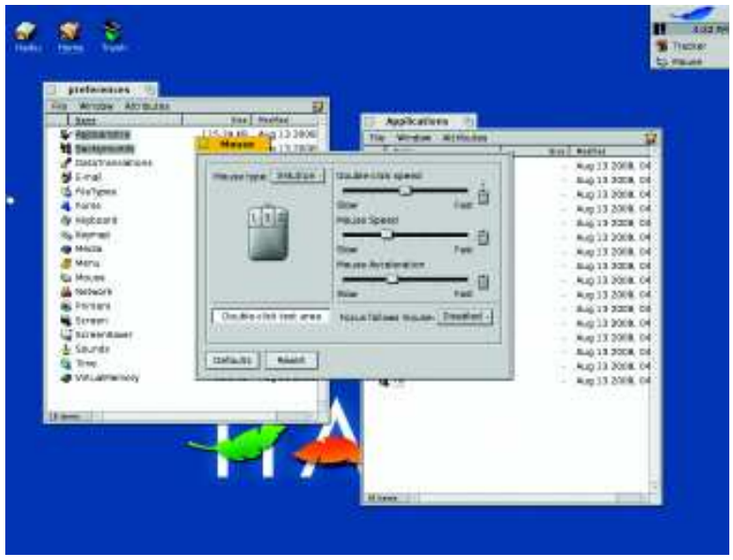
LXF: It must be particularly challenging for a feature -rich toolkit such as *Qt* though, maintaining performance on an embedded system that you don't have to worry about necessarily on the desktop.
BS: Yes, we always have to be careful. It's very easy to go overboard. If you look at embedded devices there are always certain things that go with the embedded device: the screen is much smaller, and if you look at graphical operations quite a lot of them are proportional to the size of the screen, so there are some aspects that mitigate the difference. The other aspect is that, yes, it is always possible with any framework to do things that will not run properly on any device. I think that what you want to provide is something that makes experimentation easy.

One example if you look at rendering in *Qt*, you can decide to turn anti-aliasing on or off. That's just one very simple example. If you have a higher-end system you may decide that you want to get smoother font rendering at the cost of more CPU cycles, and when you move to a low-power system there are a number of such options that you can decide to disable or enable. If you use images or bitmap graphics you can bitmap to use a low-resolution version of you application if you need to use less power. So I think the abstraction of *Qt* makes it easy to grow and be able to modify what your application does given the capability of the device.

LXF: I did have a Greenphone you know. Which I couldn't make calls or receive calls on.
BS: Really? You didn't modify the code so that it would work? That was the whole point of the Greenphone!

LXF: I did write some user-level apps for it, and I did a tutorial on coding for it. It was on the *Qt* server, the *Qt* side of things, but I couldn't get involved in the kernel or anything too technical like that.
BS: That's the difficulty. Just putting code in open source doesn't ensure that people will really be able to contribute. There needs to be some sort of evolution in the way that we think about open source, so people can make contributions without investing their whole lives into understanding a piece of code.

LXF: That was one of my questions actually. After so long, why hasn't programming got any easier?
BS: I think there's something inherently difficult in programming, which is that when you use most programming languages, if you make one mistake the application dies. It's a single-point-of-failure type process, which is not very forgiving. Programming languages have not evolved much into helping you avoid this single point of failure.



› As well as a stint developing Macintosh software, Benoit Schillings was one of the first full-time developers of BeOS, today kept alive by the Haiku project.

LXF: The logic exists in your head, and transcribing that into a program is difficult and can be verbose. I find that, especially with *Qt*, when you're continually abstracting classes, at least when you don't want to spend all day doing it and maybe only do a bit at the weekend, it's very difficult to stay on top of it. It's fine once you've learned the whole kit and perhaps you're getting paid to do it.
BS: I wouldn't say I'm a great *Qt* programmer because that's not where I spend most of my time, but I think that what I find in *Qt* compared with other platforms I've used in the past is that there are different levels of abstraction that you can use to program in *Qt*.

I think what we tried to do with *Qt* is we tried to get an initial approach that was simple enough but at the same time has some depth, and we find that the style and how people implement their application can vary quite a bit, depending on their mental patterns I guess, and their experience or the time they have to spare.

> **"My job is to be right about what technology will be like in the future."**

LXF: How would you recommend someone start with *Qt*?
BS: I think that the best way to learn *Qt* is to go, take some of the existing applications and just mess around. We do have a number of tutorials, but I always take the view – and this is true for learning a programming language or a programming framework – that the best thing is to take some existing application and mess around with it, so you can see how things have been done in the past.

Very often when I'm stuck on a programming problem with *Qt* I must admit that instead of going to the documentation the first thing I do is to look for a piece of code that performs the function that I need, and I can see: "OK, that's how it's done". So I think looking at concrete examples is often the best way to learn a system. In the case of *Qt* there's so much code available on the net that between what we provide with the *Qt* environment and and what there is on Google, you can find so many examples, and that's a very good way to learn. LXF