Motivations for reusing *and* sharing code between the NoiseTube Mobile apps for Java ME and Android

Matthias Stevens – BrusSense Team, Vrije Universiteit Brussel

August 2011

After having decided to program the new *NoiseTube Mobile for Android* app in Java we were faced with two additional choices.

Reuse of ideas and/or code

First there was the choice between either reusing actual code from the Java ME app, or writing the Android app from scratch, reusing only ideas and concepts – in which case the existing app and its code would only serve as an example. While it may seem obvious to capitalise on previous efforts as much as possible (and thus reuse code as well as ideas) the latter option had its own appeal. Namely because it would have given us the opportunity to start from a clean slate without being restricted by any past design choices, some of which might have been dictated by limitations of Java ME or the devices we were using at the time. Nevertheless we chose to go with the former approach because, in the first place, we assumed this would go faster, although in retrospect this is debatable. Secondly, we realised that using the existing code as an example would anyway result in "copy-pasting" of many bits and pieces, a therefore it seemed wise to make it a conscious decision right away.

Stealing or sharing

Opting for code reuse from the onset allowed us to carefully consider a suitable approach with respect to that matter. Essentially the choice was between two code reuse strategies which we will call *stealing* and *sharing*. The stealing approach means that the Android app would be built from a new, separate codebase to which any useful pieces of existing code (methods and/or entire classes) would be copied. This copying would happen in an unstructured, ad-hoc manner and this *stolen* code would be adapted *in-place* to make it Android-compatible. The sharing approach means that the NoiseTube Mobile code would be organised in three codebases, two for the platform-specific parts of each app and a third one containing a *platform agnostic* core of classes shared by both apps. To realise this cross-platform architecture and establish the shared codebase, the existing Java ME codebase would have to be refactored to identify and extract all generic app logic – thereby separating it from Java ME-specific code.

In spite of the moral connotations of the terms, both the stealing and the sharing approach had their own appeal. After careful consideration we decided in favour of the sharing approach. However, before we discuss the details of the refactoring this entailed, it is worthwhile to elaborate on the decision process itself. As we will see, both strategies were especially contrasted by the way they would affect the short-term and long-term development of NoiseTube Mobile.

The main short-term concern was the speedy completion of an Android app which was functionally on par with the Java ME app. As one might expect this development time factor ruled in favour of stealing. At least two reasons can be thought of:

- For one, the stealing approach would have allowed us to focus our attention entirely on the development of the new app. The existing Java ME app could have been left unchanged and the only time spent looking at its code would have been in search of inspiration (the "ideas and concepts" discussed above) or parts worth stealing. The sharing approach, on the other hand, first required a far-reaching refactoring of the Java ME code. Furthermore, during the development of the Android app we had to ensure that any changes made to the shared codebase did not cause the Java ME app to break.
- The other reason relates to the complexity of the task at hand: the sharing approach required a considerable effort to design a fitting architecture for the shared classes and their interface with platform-specific ones. This work had to precede the refactoring of existing code and the writing of most new, Android-specific code. Conversely, by adding less architectural complexity, the stealing approach would have allowed more time to be spent on straightforward programming.

In the medium- and long-term the main concern was (and still is) the maintenance and further development of both or either one of the apps. Whether the reused code was stolen or shared, our limited resources (in terms of manpower) require us to set priorities. The shrunken market share of the Java ME platform and its lack of recent innovations made it quite obvious: since the completion of the Android app new development efforts are mainly focussed on that platform[1] and eventually maintenance of the Java ME app will be discontinued altogether. However, the stealing approach would have accelerated the death of the Java ME app considerably by hampering the maintainability of our software.

The maintenance problem stems from the fact that the stealing approach effectively amounts to code duplication. Consequently future improvements (bug fixes or new functionalities) involving "stolen" code would have to be applied twice in order to keep our offerings for both platforms on par. To make matters worse, the duplicated code would have been adapted to suit the new platform. With no shared classes to interface with, these adaptations would have been carried out without much regard for code-level compatibility with the existing app[2]. Therefore, improvements would not just have to be applied twice, but twice to different code, compounding the maintenance problem. In essence,

[1] In parallel with the development of prototypes for Apple's iOS platform.

[2] Even if a form of compatibility had been aimed for, there would have been no mechanism to enforce it.

by stealing code instead of sharing it, we would have allowed both NoiseTube Mobile implementations to diverge from the start. At first, this divergence would only have affected details of the code (due to platform-specific adaptations). But soon this would have turned into a divergence in terms of software quality and features as our limited resources would have required us to make choices. This would probably have meant that improvements to the Android app would not be carried over the Java ME app at all.

By following the sharing approach we reckoned it would be easier to let both apps live side by side for a period of time. Because code duplication was avoided, maintenance of the reused – or rather, shared – code only has to be performed once and affects both apps. Even as our development focus shifts towards Android the Java ME app still benefits from our efforts because a significant part of the code is shared. Of course many improvements, definitely those which add new functionalities, are not restricted to the shared code. Often they also require changes or additions to platform-specific code – for example with regard to the user interface. Still, the sharing approach created a setting in which both apps can co-evolve and in which our transition from Java ME to Android can run smoother. However, it should be noted that this involves an on-going balancing act. As new functionalities are added, the design of the shared classes has to evolve and at all times the capabilities of both platforms have to be taken into account.

While they would not necessarily have compensated the troubles of code duplication there are two aspects of the stealing approach that could have had a positive effect in the medium- or long-term. One is that the code size of the Android app – in terms of LoC[3] and number of classes – would have been lower. The reason is that all features would have been implemented in a platform-specific manner right away, without a need for intermediate layers of platform agnostic code. For the same reason the size of the Java ME codebase would not have grown bigger. When looking at the apps in isolation these smaller, less complex, codebases could have simplified some maintenance tasks and made them less prone to error. The second, related advantage of separate codebases is that they could have made it easier to benefit from platform-specific features (which have no equivalent on the other platform).

Despite these mitigating circumstances the choice between stealing or sharing code still boiled down to a comparative assessment of short-term benefits and long-term sustainability. The main advantages of the stealing approach are due to the fact that it would have represented a cleaner break from the past, thereby permitting a more focussed and straightforward development process for the Android app. But with a number of experimental measuring campaigns still on the calendar and our prior investment in a series of Java ME devices, it would have been a risk to neglect the maintenance of the Java ME app already. By contrast, perhaps the most powerful – and conclusive – argument in favour of the sharing approach was that it represented an opportunity to rethink past

[3] *Lines of Code*.

design and implementation choices, well beyond what was strictly necessary to extract platform agnostic code. In this sense, the additional design and refactoring efforts were worthwhile because they significantly improved the Java ME app, rather than just keeping it maintainable. Furthermore, the creation of a platform agnostic expression of the core NoiseTube Mobile functionalities could even benefit future porting efforts, even to platforms which do not employ Java.

These reflections also evoke thoughts on the tension between pragmatic and idealistic visions on software engineering. In that respect copying and adapting code in an ad-hoc manner might be seen as a pragmatic embrace of the benefits of code duplication. In the same light, refactoring and sharing code might be seen – somewhat cynically – as an idealistic (or even dogmatic) pursuit to avoid the ills of code duplication at all cost. That said, this idealistic pursuit is also what made the sharing approach appear more interesting, more challenging and just more fun. Although that may be a matter of taste, it was a (secondary) argument in its own right. Furthermore, in retrospect it could be said that pragmatic considerations have governed much of the prototyping and initial development of the NoiseTube platform. Therefore a more principled redesign of NoiseTube Mobile seemed indeed beneficial beyond the goal of developing a functionally equivalent Android app.