Null References

Adrian Kosmaczewski

2008-03-07

There's an interesting discussion going on these days on Ruby blogs about, basically, how to avoid one of the most common, annoying, easy-to-create bugs in any programming language: calling a method on a null reference (or pointer, depending on your language).

This single issue happens all the time, in garbage-collected and non-managed languages, static and dynamic, weakly and strongly typed; you have a handler variable "pointing" to an object, and before calling any methods on it, you'd better be sure that the object is there; you end up using assertions, "if" statements (and all of its variants), boilerplate code all over the place, when everything you want to do is to call that damn method. It's frustrating, time-consuming and oh so common that we just try to not to think about it anymore.

In Objective-C there is an easy solution to this problem: you can safely send messages to (which is roughly equivalent to "call methods on") nil, but of course, not everyone likes that. I think that this single feature is responsible for a big deal of "user perceived stability" in the whole Cocoa runtime; it exchanges a what could be a potentially fatal, low-level and unrecoverable error (leading to a complete application crash) into a purely functional one; "look, I've clicked here and nothing happens!". The application does not crash anymore, it just does not do what it should, because the object that should have received the message is not there. The user has a smoother experience, and this means a lot in the long term.

The beauty here, shared by Objective-C and Ruby (and as far as I understand, Smalltalk and other languages), is that messages and method implementations are decoupled; you can forward messages from one object to another, creating chains of responsibility; you can log messages before you execute them (doing some aspect-oriented stuff without all the marketing fuss); you can change the implementation (or even remove it and place it somewhere else altogether) without breaking your clients. It brings a whole lot of power, with the small overhead of having a runtime process dispatching methods, which is, yes, takes slightly longer than a virtual method call, and (of course) even longer than compile-time bound method call.

I think that dynamic languages have a definitive advantage in this field, and this

is why I prefer them in environments with requirements evolving constantly, where clients more often than not request new features and where you must reduce maintenance costs; not having your app crash in your face is a good sign of software, and languages that allow you to deliver them are fundamental.