Open Kosmaczewski
Thoughts on Software Engineering and More

Adrian Kosmaczewski

November 1st, 2009

Contents

Introduction

I have toyed with the idea of writing a book for years. In the meantime, I kept a blog, with my own articles about my findings in the field of software engineering. After 5 years, I realized I had published over 600 articles, and then I thought it might be a good idea to gather my preferred ones into a book.

Of course, text on the web has a particular feeling, way different from what you get on a printed book. Hyperlinks, video, animations, everything goes away, and this brings content to the front. Having content that stands the proof of being published in paper is not easy to do. I hope I've been able to create something timeless and meaningful.

This is the result of that process. I hope that these words will be as enjoyable on this medium as I wanted them to be. You can always contact me via e-mail on adrian at kosmaczewski dot net, or on Twitter as @akosma. I'd love to hear from you.

Acknowledgements

This book would not have existed without the tacit, unknowingly collaboration of many brilliant people I've met around the world, Fabrice Truillot, Damián Sanguine, Paolo Badino, Adam Jones, Rami Entin, Cédric Bompart, Cédric Ducommun, Jacob Deker, Jean-Eric Cuendet, Manuel Siggen and Patrick Valsecchi. Thanks to them for sharing their knowledge with me in different times and in different ways during the past 13 years.

And thanks to Claudia, my wife, for being there day after day, supporting and encouraging me to go always beyond. Nothing would have much sense if you were not here.

Creative Processes

It always starts with a white space, like this book.

It could happen on a sheet of paper, or lately an empty <textarea>, but the feeling is the same. You get the first flow of ideas, rushing through your head. That's why I always have a small Moleskine notebook in my pocket; you never know when these things will happen, how long they will last, and when you'll have another one. Paper has an obvious advantage on computers: you do not need electricity.

I do not "have" an idea; the idea has me. It comes, and then it goes. My task is to write it down, or let it go. I choose.

Jadis you would have started writing in the old-fashioned way; pen, pencil, paper, but the keyboard has got our attention. It's a different feeling; before, just one hand was to held responsible for my stuff; now it's every one of my fingers. Simultaneously. It has an advantage, which is to make things go faster. I can write faster, almost as fast as I think. But I need electricity, though.

I can create not only texts but also code. I can later make that code run on the computer, on mine or even on yours. I can talk to those computer, to make them do what I want. We're new wizards, in a world that would turn crazy any 16th century man. We talk to the machine, and it answers us.

Then I go back, I erase, I reorder paragraphs, I re-read, I spellcheck, I smile, sometimes I tear off the page and start from scratch (the equivalent of selecting "Edition / Select All" & hitting the delete key).

Sometimes I publish, sometimes you read me. There is a nice randomness in all of this, and I clearly enjoy it. Thanks for being there.

Software Engineering

Web Development == Software Development?

Most of what I write in my blog has to do with the craft of making good software. It is not an easy task, but it's a rewarding experience, not only from a technical point of view, but from a human perspective as well. Many of the chapters in this book are papers I wrote from 2005 to 2008, while preparing my Master's degree. This one belongs to this group, and has been one of the most popular articles in my blog for years[1]*.*

Introduction

I have been developing web applications since 1996, and I still do a fairly large amount of web development nowadays. During these years I have seen some common misconceptions and myths about web development, that ultimately have a direct impact in the usability of the system. I will outline these in this article, providing at the end my opinion on how to achieve a proper QA process.

Website Development IS Software Development

A website is a piece of software, whether the team acknowledges the fact or not; as such, all the best practices for software development apply to web design and development as well, particularly in those projects that aim to build interactive "web applications" instead of simple "brochure websites" based on some CMS, usually displaying rather static information.

[1] *http://kosmaczewski.net/2007/08/02/web-development-equal-software-development/*

Acknowledging this fact is something that many teams do not do (or want to do); in my opinion, just because JavaScript lacks a compiler it does not mean that it is not a programming language, and sooner or later, nearly all websites use JavaScript - arguably the "world's most misunderstood programming language" (Crockford). Thus, the following tasks apply even for the smallest website:

- Have a specification;
- Have a schedule;
- Have an architecture (albeit small) that separates the UI from the rest of the application; this allows the designers to change the look & feel without disturbing those working in the functionality;
- Make a prototype; preferably many of them; keep them for future reference;
- Have (human) testers;
- Automate your tests (and builds, if you use a compiled technology like Java);
- Use a bug database;
- Use version control;
- Be agile: get your client involved and have small development-testing-deployment cycles;
- Know your audience;
- Scope your target.

I will provide a small comment on the last two items in the following sections.

Know Your Audience

A very important point to know is the target audience of the website: is it a public or intranet system? What is the average age of the users? What are the legal and accessibility requirements? And last but not least, what languages should we support? These informations must come from a simple analysis of the target users, and the answers to these questions must be stated in the design document of the website project.

Knowing the target languages is fundamental, since it means choosing a target encoding for the system output; Unicode UTF-8 is a canonical choice, but for English-only websites ISO-8859-1 is more than enough. On the other side, some languages like Arabic are displayed in a "right-to-left" fashion, and this means that your application must look good and work properly in this mode too.

Finally, some clients (particularly governments and federal agencies) require providers to comply with accessibility rules, for supporting users with disabilities. The use of text-to-speech browsers, large font sizes and other media must then be taken into account during analysis, design and development of the application.

Scope Your Target

When developing web applications, one common mistake that I've seen many development teams do is to forget to set up a list of minimum target browser requirements for the application. In my experience, relying on HTML, CSS or other standards is fundamental but sadly not enough: setting a scope means creating just a short list of browsers, including screen size, browser and operating system versions, scoping the sandbox where the QA operations will be applied. This list must be part of the specification of the system being created, and must be visible and known by everyone. A sample list could be the following:

- Internet Explorer 5.5+ for Windows XP Service Pack 2
- Firefox 1.5 and all Gecko 1.8-based browsers (many different OS)
- Konqueror 3 for Linux
- Safari 2 for Mac OS 10.3
- Opera 9 (any OS)

Using the name of the layout engine also helps (like the "Gecko" reference above) since several browsers use the same layout engine, providing the same characteristics to otherwise different browsers: for example, Gecko 1.8 is used in Mozilla, Camino (for Mac OS X), Galeon, Firefox 1.5 and 2.0, and other browsers (Wolter, 2007)

It is also important to scope the minimum supported screen size:

- Minimum screen size: 1024 x 768 pixels.

Of course, if the website must support other types of devices (game consoles like the Wii or the Xbox, cell phones or PDAs), this must be specified as well, with detailed version information, and details about the supported screen sizes.

Last but not least, it is fundamental to define the "DOCTYPE" to be used in the website; this setting defines the set of valid HTML tags to be used in the page:

> Per HTML and XHTML standards, a DOCTYPE (short for "document type declaration") informs the validator which version of (X)HTML you're using, and must appear at the very top of every web page. DOCTYPEs are a key component of compliant web pages: your markup and CSS won't validate without them.

(Zeldman, 2002)

Another good practice is to add an explicit list of definitely non-supported browsers or operating systems:

- Netscape 4.x (pre-Gecko rendering engine)
- Opera (versions prior to 9)
- Internet Explorer for the Macintosh
- Mac OS versions prior to 10.3
- Linux kernels prior to 2.4

Perhaps surprisingly, these simple actions are often forgotten, which leads to confusion and lack of coherence in the QA activities of the team. Managers avoid having developers refusing to fix some incompatibility because of the lack of standards support in some browser (which always happens). On the other side, the whole team is shielded against some change in the scope; for example, the marketing team might come up with a requirement to support Opera 8, and this can have a negative impact on the schedule if it was not specified upfront, since all the CSS and JavaScript code might have to be tweaked to support the new browser.

Different Dimensions

As shown in the above paragraphs, websites are complex beasts, that can have several (often conflicting) dimensions:

- Different languages;
- “Right-to-Left” against “Left-to-Right” layouts;
- Different target browsers;
- Different target operating systems;
- Users with potential disabilities;
- Finally, the website functionality itself!

How to manage this complexity? Unfortunately, given the high fragmentation of the web market, there is not a simple answer to this problem; I think that (at least currently) the best approach is a mix of automated and manual testing procedures:

- Use standards, everywhere, all the time. No exceptions to this rule.
- Have your website tested by human beings, particularly to find spelling or grammar errors, and to verify that the websites does what it is supposed to do, in all the target browsers and operating systems.
- Use JsUnit (http://www.jsunit.net/) to unit test your JavaScript code. Run the unit tests as often as possible, usually every night.
- Use JsDoc Toolkit (http://www.jsdoctoolkit.org/) to document your code and make this documentation available to the team.
- Compress your JavaScript files using tools like Dojo ShrinkSafe (http://alex.dojotoolkit.org/shrinksafe/ to make files lighter and speed up their execution.
- Use Selenium (http://www.openqa.org/selenium/) for browser compatibility and functional testing.
- Use online validators, particularly those of the World Wide Web consortium (http://validator.w3.org/). Use standards and fix your code to eliminate warnings and errors in the validation process.

- Have your team use several browsers in their development workflow (all those that are specified as mandatory in the specs); every new feature means a unit test, a functional test, and a "smoke test" reloading the page on the browser.
- Have the developers use tools like FireBug (http://www.getfirebug.com/) and the Web Development Extension for Firefox (http://chrispederick.com/work/web-developer/).

Conclusions

Web development is a fascinating and constantly evolving activity. We are reaching a point where the technologies are getting more and more stable, secure and affordable, but I think that the web industry lacks of comprehensive and reliable integrated solutions yet.

References

Crockford, D.; "JavaScript: The World's Most Misunderstood Programming Language", [Internet] http://javascript.crockford.com/javascript.html (Accessed May 12th, 2007)

Wolter, J.; "Javascript Madness: Layout Engines", February 2007 [Internet] http://www.unixpapa.co
(Accessed May 12th, 2007)

Zeldman, J.; "Fix Your Site With the Right DOCTYPE!", [Internet] http://alistapart.com/stories/docty
(Accessed May 12th, 2007)

Design by Contract

http://kosmaczewski.net/2007/08/23/design-by-contract/

Introduction

Even if Design by Contract is a trademark (Eiffel Software, 2007) the idea behind it is the more general one of "defensive programming". As software developers, we often concentrate our efforts in the main code of the application, which is the interesting part, and that provides the realization of the use cases identified during the early phases of the project.

My opinion is that defensive programming techniques lead to more secure, stable, and less bug-prone code, and that they require less documentation, since the resulting code is more "self-documenting". Moreover, I will describe in this paper language-independent techniques, that can be used in different situations and systems, that are somehow similar to the Eiffel approach.

Design by Contract

In a previous entry in my blog, I have described the Ariane 5 rocket disaster of 1996, its cause and how the notion of defensive programming could have helped avoid it:

> Of course neither me nor my colleagues work on Eiffel (yet). But, we do create software, we do handle exceptions (do we?), and we do lose money, credibility and faith every time there's an unhanded exception in our software. These exceptions cost us a lot: that "Design by Contract (TM)" thing

> can positively help us, just by rethinking the way we build our software. Here's my idea: even if we don't use Eiffel, our good-old Algol-related languages can be used in pretty much the same way as Eiffel behaves, but of course it requires some of our own brainy CPU time. The trade off is simply a much clear interface that fits into a higher level, architectural view of the system, explicitly stating the valid ranges of execution for our code, and helping out in setting unit and integration tests.

(Kosmaczewski.net, 2005)

Eiffel's idea is coherent with the concept of "Defensive Programming". It is not a new trend in programming, but if you Google for it you will find quite a few papers describing common techniques and best practices, like the one on CodeProject.com that I provide in the references (Manderson, 2004).

The idea behind defensive programming is that you should not trust what comes from outside your code, even if it is your own code that calls other pieces of your own code; you could just as well call it "limited applied paranoia". This is important not only for stability purposes (you do not want to call methods on a null pointer) but also for security; take for example the well known "SQL injection attacks" that happen when you trust too much the inputs of your users... a little verification helps to avoid disasters.

In Eiffel, this is done using the following syntax:

```
1  method_name (parameter_name: INTEGER): INTEGER is
2  require
3          parameter_name <= some_maximum_value
4          -- more conditions, if needed...
5  do
6          -- code of the method here
7  ensure
8          -- postconditions that must always be met
9          -- no matter what happens, here
10 end
```

(Source: Kosmaczewski.net, 2005)

Of course, not all programming languages provide this kind of pre- and post-verification syntax; some ways to apply defensive programming in non-Eiffel languages, could be for example:

- Asserting for validity

- Using Aspect-Oriented Programming
- Using proper exception handling

I will give a short overview of these in the following paragraphs, comparing them with the Eiffel approach.

Asserting

"Asserting" is a common technique used in C and C++, but that can be used as well in any other language; basically it consists in using a macro or function that will raise an exception (or halt the program execution altogether) if a condition is true. Typically this is done before sending a message to a null pointer, since this mistake is a common one:

> The ANSI assert macro is typically used to identify logic errors during program development by implementing the expression argument to evaluate to false only when the program is operating incorrectly. After debugging is complete, assertion checking can be turned off without modifying the source file by defining the identifier NDEBUG. NDEBUG can be defined with a /D command-line option or with a #define directive. If NDEBUG is defined with #define, the directive must appear before Assert.h is included.

(MSDN, 2007)

The use of macros help to remove the asserts from the shipping code, that are usually only used during development and debugging. You do not want to ship code that discloses too much information about errors. . .

The situation in which asserting is useful is not uncommon in other languages; the much feared "null pointer exception" can happen in JavaScript, Ruby, C#, Java, and many other languages. In Eiffel, this would be handled in the "require" block, right before the main code execution.

Just for the record, the Apple Cocoa runtime (and in general the Objective-C language runtime) allows messages to be sent to "nil" objects (aka null pointers), without problem. However strange this might sound, this has an interesing side effect; even if the

developers forget to initialize a pointer, and send a message to it, nothing will happen; the application will not crash, and this particular feat is one of the secrets of the stability of the overall Mac system. It is not that developers make less mistakes or that some magic applies; the Cocoa runtime proactively protects users from sloppy programmers, providing a more stable environment for them: the application might not do what the user want, but at least it does not crash.

Aspect-Oriented Programming

AOP can be used in this context as well, intercepting messages before and after methods execution, to verify their conformity and their correctness. The problem with AOP is that the definition of aspects is usually done in a different code file, which makes it harder to understand and maintain; this is where the Eiffel approach is the most interesting, since the "require" and "ensure" methods are somehow part of the method's signature.

However, the advantage of AOP is that the definition of poincuts is much more flexible, and one could theoretically inject code in different situations, without having to touch the original code (and thus reducing coupling and cohesion); the AOP runtime would take care of the weaving for us. Moreover, the weaving could be changed at runtime, while "require" and "ensure" conditions in Eiffel are compiled and statically linked.

Just for the example, I will show a bit of Ruby on Rails code that shows a class that provides a "hook" for before-execution methods; these methods are called automatically by Rails before any execution:

```
1  # The administration functions allow authorized users
2  # to add, delete, list, and edit products.(...)
3  #
4  # Only logged-in administrators can use the actions here. (...)
5
6  class AdminController < ApplicationController
7
8    before_filter :authorize
9
10   # List all current products.
11   def list
12     @product_pages, @products = paginate :product, :per_page => 10
13   end
14
```

```
15    # Show details of a particular product.
16    def show
17      @product = Product.find(@params[:id])
18    end
19
20    (...)
21
22  end
```

The important part of the code above is the “before_filter :authorize” line, that tells Ruby to automatically call the “authorize” method before executing the other methods. The “authorize” method belongs the to the “ApplicationController” class, and the “before_filter” hook is defined in the “ActionController::Base” class, that’s part of the Ruby on Rails framework. Rails uses the dynamicity of Ruby to “detect” a call to a method, and intercepts it to inject the desired behavior before the execution of that method. Similar hooks exist for post processing.

It is interesting to note that such mechanisms can also be implemented in static, compiled languages using class hierarchies, where base class’ methods call virtual methods of subclasses, similar to how ASP.NET processes pages.

Proper Exception Handling

Modern object-oriented runtimes, such as Java, .NET, Ruby and Cocoa use “exceptions” to handle errors. Exceptions are objects that are “thrown” whenever some unexpected condition is met during runtime to the method callers up in the stack, until some method “catches” it (hopefully someone will). This objects carry a whole meaning about the error, and are much more explicit than the C / C++ approach of returning numeric codes (HRESULTs, anyone?). By looking an exception, maintainers can see the context of execution of the code, and find the bugs that have created (or allowed) it to happen.

Even if the idea is interesting, it is also known that Exception handling is an expensive way to handle errors; for each exception thrown, runtimes have to walk the stack and look for possible handlers. This operation is expensive (Sintes, 2001) and that is why exceptions should not be used to control program flow.

The canonical example to show this is the following: instead of writing this (pseudo) code

```
try
{
    openFile(fileName);
}
catch (FileNotFoundException e)
{
    doSomething(e);
}
```

one could write the following, functionally similar, but more “defensive” code:

```
if (fileExists(fileName))
{
    openFile(fileName);
}
else
{
    notifyProblem();
}
```

If the above code is part of a performance-sensitive system, such as a web application, and the “file not found” situation happens more often than not, then a lot of processing power could be saved by just introducing this small change in the implementation.

In the case of Eiffel, the “contract” for the above “openFile” method would not only include the file name, but also a “require” block stating that the file must exist before any processing. This way, all calls to openFile would be safe, and as a result, clients would not need to check that fact before calling the method. Less code, cheaper to maintain, and more stable.

Conclusion

Defensive programming is a mind paradigm; you can apply them in any language, be it procedural of object-oriented, and provides stronger code, resistant to changes and self-documenting.

References

Eiffel Software, “Building bug-free O-O software: An introduction to Design by Contract(TM)”, [Internet] http://archive.eiffel.com/doc/manuals/technology/contract/ (Ac-

cessed June 22th, 2007)

Kosmaczewski, Adrian, “The Exception to the Rule”, February 15th, 2005 [Internet] http://kosmaczewski.net/2005/02/13/the-exception-to-the-rule/ (Accessed June 22th, 2007)

Manderson, Rob, “Defensive Programming”, August 6th, 2004, [Internet] http://www.codeproject.com/gen/
(Accessed June 22th, 2007)

MSDN, “assert (Visual C++ Libraries)”, [Internet] http://msdn.microsoft.com/library/en-us/vclib/html/_CRT_assert.asp?frame=true (Accessed June 22th, 2007)

Sintes, Tony, “Does exception handling impair performance?” [Internet] http://www.javaworld.com/javaworl
07/04-qa-0727-try.html (Accessed June 22th, 2007)

Thomas, Dave & Heinemeier Hansson, David, “Agile Web Development with Rails”, The Pragmatic Programmers, 2005, ISBN 0-9766940-0-X, sample source code taken from http://pragmaticprogrammer.com/titles/rails1/code.html (Accessed June 22th, 2007)

Dangers of Prototyping

http://kosmaczewski.net/2008/08/15/dangers-of-prototypin/

Frederick P. Brooks Jr. has written about prototypes, saying that they are not only useful but strictly fundamental pieces of the overall software process, as in many other engineering activities. He gives the example of a pilot chemical plant, prepared to process 10´000 units per day instead of the 2 million units a day that the final plant would have to handle, in order to demonstrate the feasibility and uncover some unforeseen problems.

He summarizes his opinion in the famous phrase "plan to throw one away" [**?**, page 116], underlining the problem of change management: managing change, right from the beginning of the project, instead of ignoring or avoiding it, is particularly important in software projects, since it presents a solid mindset for all stakeholders in order to avoid scope creep, schedule and staffing problems.

One of the proposed solutions to manage change (the "only constant thing") in a software project is the use of prototypes, which consist of UI mockups, showing basic interactions between the components and screens, allowing users to see in real time how the final software will look like, and serving as part of the feasibility study before creating the final product.

Several authors have highlighted different problems related to prototypes:

> We consider three dangers to be most serious.

- Assumptions may be hidden (. . .)
- Feedback may be too expensive (. . .)
- Inappropriate human-computer interaction issues are highlighted

[**?**, page 180]

> One of the risk of creating a User Interface Prototype is accidentally raising unrealistic expectations about future progress on the project.

[**?**, page 117]

> Firstly, there are still plenty of users and managers around who think that the software is not much deeper than the user interface, and so if they see a nearly finished interface, they think the project is nearly done. (. . .) Secondly, engineers often get carried away about making a lovely user interface, and spend far too much time on it. (. . .) Lastly, and perhaps most importantly, is the fact that almost no code that is written as a 'temporary throw-away' actually gets thrown away. It usually gets used in the product in one way or another.

[**?**, page 254]

- Needs cooperation of management, developers, and users Managers may view prototyping as wasteful
- Managers and/or customers and/or marketing may view prototype as final product
- Programmers may lose discipline
- Prototype can be overworked (reason for prototype is forgotten)
- Prototyping tool may influence design
- Possibility of overpromising with prototype

[**?**, page 5]

In my personal experience, I must say that I have not worked in many teams using prototyping extensively. Many times, when I have came up with the idea of doing a prototype, my managers (and even my coworkers) would dismiss it with the following criteria:

"We don't have time - money - staff - (add your own preferred variable here)" "OK, but do the prototype in such a way that we could reuse it later" "We do not lose time

in prototypes in this company; we do real work here" "We do not have a budget for prototypes in the project" "We know exactly what the customer wants" "Oh, we've tried that once, but the customers never like what they see anyway, so why bother" "Our client does not want prototypes, period" "Our CEO does not want prototypes, period" "I do not want prototypes, period" "Our policy does not allow prototypes"

However, I must add that in just one case (a successful project I worked in three years ago), the analysts team managed to fit in the project budget the capability to create a set of user interface prototypes using Microsoft Visio. This tool (or any other diagramming tool) has many advantages over the traditional "Visual Basic"-based prototyping:

- Any user with Visio (be it analysts, developers, end users with basic Microsoft Office knowledge) can contribute to the prototypes, suggest changes and play with the mockups;
- No code is needed, which avoids developers to reuse it later;
- The output can be inserted (typically as images) in other documents, e-mails, etc.

This technique had a tremendously positive impact in the project:

- A great deal of feedback from the client was related to the mock-ups;
- The design documentation had an excellent complement of information, in the form of screenshots, coupled with the textual indication of every possible action on the user interface;
- The developers could use these indications to reduce the uncertainty and deliver a product that matched the final decisions exactly;
- Since the prototypes were not "reusable" (they were after all only drawings on Visio) the developers could not be tempted to reuse the code in the final product.

In my opinion, this approach was one of the key factors of success of the project, but not the only one: the team members and the customer agreed that the project was critical and complex, and that a prototype could help everyone to understand it better. There was a specific mindset in the whole project, at both sides of the equation, which made prototyping a good option, and a success factor. Without this mindset, I do not think that prototyping will fit in any project.

A Watch - from an OOP perspective

http://kosmaczewski.net/2008/07/13/a-watch-from-an-oop-perspective/

A watch might be one of the most common types of objects, but it remains also one of the earliest pieces of human craftmanship to show an extreme level of complexity, all contained in a small amount of space. Since the late 1700s, artisan watchmakers in Switzerland and elsewhere have shown their pride and skills creating watches called "Grande Complications", containing thousands of individual parts and performing incredible functions:

> The most complicated watch ever made, known in watch enthusiasts' circles as "The Ultimate Watch," is Patek Philippe's "Calibre 89. The incredibly precise operation of 1728 parts in this really ultimate masterpiece of watchmaking allows to perform no less than 33 (thirty-three!) complicated functions, among them a correction for the 400-year-rule, an Easter date indication, a star chart, a tourbillon, a perpetual calendar, a sidereal time indication, and, and, and ... This watch was sold in 1989 for the nice round sum of about four million Swiss francs."

(Ozdoba, 2005)

(Source: CNN.com, 2005)

More information about the "Calibre 89˝ can be found here and in the Patek Philippe Museum website.

However, the same watchmakers that made these fine pieces were also aware of the basic information that their creations were to provide: time. As such, their watches re-

mained extremely easy to use, and they set up the basic standard for analog watches, in such timeless designs that the latest Swatch models show the same basic layout and functionality.

The underlying concept is the very same used in today's object-oriented abstraction and encapsulation. Even Apple uses the idea of the watch to show this characteristic:

> All programming languages provide devices that help express abstractions. In essence, these devices are ways of grouping implementation details, hiding them, and giving them, at least to some extent, a common interface—much as a mechanical object separates its interface from its implementation, as illustrated in Figure 2-1.

(Source: Apple Developer Connection, 2006)

In this article I will provide my view about how different OOP concepts apply to a real-life object such as a watch, in all its forms.

Concepts

Object

Every existing watch could be thought of a single instance of a more generic class "Watch". This is possible since all watches share a common set of characteristics, that is, at least, the capacity to display the current time. Of course their external appearance and their whole set of characteristics may differ, but in this common aspect, they are all the same. This makes the "Watch" class a simple but extensible one.

Attributes

Every watch has a distinct set of attributes, for example:

Brand Type of wristband (metallic, leather, plastic) Waterproofness Current time Size Cadran color Analogic or Digital Number of hands Battery level Behavior

Watches have a basic behavior; they increment their "current time" attribute, by one second every second. This way, they manage to update themselves automatically,

and to provide users with the right time all the time. Watches that provide different functionalities may also have different behaviors (I had long ago a Casio watch which provided barometric pressure and temperature, both constantly updated and very handy when doing outdoor activities).

Class

A “Watch” class would provide common characteristics to all watch instances, the most basic being that of providing time information and being able to be held by a person (his owner).

Inheritance

There are lots of different types of watches, but some common subsets can be easily seen:

- Swiss vs. Japanese
- Digital vs. Analog watches
- “Grande complication” vs. simpler watches
- Battery vs. wrist kynetics- powered
- Plastic vs. metal watches
- Quartz vs. mechanical

These subsets can be used to model the right class inheritance, the one that makes more sense in an application (since there is usually not a single answer). C++ provides also multiple inheritance, which is not the case of many languages; in this case, every instance could inherit from several base classes (Japanese + digital + simpler + battery + plastic + quartz = Casio watch).

At the same time, a “Watch” could be seen as a subclass of “Clock”, which also provides time information but is usually not easy to hold in the palm of your hand (the Big Ben is a clock, but not a watch).

Abstraction

Users are completely isolated from the internal mechanisms of their watch; they usually don't know (nor they need to know) how their watches work; they just care about the current time displayed.

Modeling

When creating a new kind of watch, a designer might find inspiration on existing watches or from new functionalities that might be useful to the end user. This in turn "reshapes" an implicit Watch class, adding new subclasses, or properties to existing classes. This is something that Casio made extensively in the 80s and 90s, providing watches with extended functionalities, rather different from what was offered at the time (temperature, barometric pression, speed, even television or radio).

Messages

Whenever the user sets up the right time in a watch, or uses one of its functionalities (for example to get the current date, the temperature), she has to interact somehow with the watch; usually this is done using a set of controls (knobs or keys) located around or over the watch, and each time that the user presses or turns one of those controls, one could think of a message sent to the watch. Of course a correct protocol is needed (that is, turn or press twice to set the time to 2 o'clock) and some operations are forbidden, or even better, impossible (like setting the hour to 25, or the minutes to 345).

Encapsulation

This concept is closely related to that of "Abstraction"; users can use and interact with rather complex structures, using extremely simple commands, like buttons and knobs. This encapsulation guarantees a correct mechanism (it is difficult for users to screw up their watches), satisfies warranty requirements (since only qualified people can deal with the internal structure of the watches), and also simplifies the use of the watch, making it easy to use and providing value to the user.

Interface

Users do not need to know whether Patek Philippe watches are more complicated than Swatches, simply because their basic functionality is the same, and the difference has more to do with aesthetics (and price...) than anything else. Both provide the time in analogic form, with similar knobs and maybe even similar advanced functionalities (date, chronograph, etc).

Information hiding

The watch displays only the right amount of information to the user; the rest is kept internally, and is used by the internal mechanisms without the user even knowing about them.

Data members

Inside the mechanisms of the watch, implicit (in the case of analog watches) or explicit (in the case of digital watches) bits of information can be found; these could be either private (angles of the hands, resort tensions, controller values) or public (current time, battery level).

Member functions

Every distinct functionality or service of a watch could be thought of like a "member function" of the class Watch:

Display current time Set time to new value Display current date Start chronograph Buzz alarm at the right time Internally, the watch may have other "private" member functions, used to perform internal duties.

References

Apple Developer Connection, "Interface and Implementation", [Internet] http://developer.apple.com/docume
(Accessed October 1st, 2006)

CNN.com, “Patek Philippe — The ultimate watch”, Monday, October 31, 2005, [Internet] http://edition.cnn.com/2005/WORLD/europe/10/27/ultimate.watch/index.html (Accessed October 1st, 2006)

Ozdoba, Christoph, “Pocket Watches - Glossary”, August 13, 2005, [Internet] http://www.ozdoba.net/ (Accessed October 1st, 2006)

Patek Philippe Museum, [Internet] http://www.patekmuseum.com/ (Accessed October 1st, 2006)

Swatch website, [Internet] http://swatch.com/index_flash.php (Accessed October 1st, 2006)

Webb, Ken, “Digital Watch - Sample Xholon App”, [Internet] http://www.primordion.com/Xholon/sam (Accessed October 1st, 2006)

Null References

http://kosmaczewski.net/2008/03/07/null-references/

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.

Django Architecture Approaches

http://kosmaczewski.net/2008/04/04/django-architecture-approaches/

I've just had a very interesting conversation with my colleague Marco about different approaches to the organization of code inside a Django application.

As you might know (and if you don't I'll tell you anyway), Django's views (somehow occupying the "Controller" level in an MVC architecture) must take (at least) an HttpRequest instance as a parameter and must return an HttpResponse instance. That's how it goes in Django, this is the law. This means that you must be sure that the last instruction in your request processing code (in whichever way you've organized it) must return an HttpResponse instance, usually calling the HttpResponse() constructor (or of any of its useful subclasses), or by calling the django.shortcuts.render_to_response() function, or something similar.

This has, in my opinion, a major drawback: it might limit code reuse and it increases the coupling in the code. Everything's not lost, however.

Before you start the flame wars, let me explain, using an example coming from the Django website; this represents a basic Django view function, returning some response containing data fetched from the database:

```
1 from django.shortcuts import render_to_response, get_object_or_404
2 # ...
3 def detail(request, poll_id):
4         p = get_object_or_404(Poll, pk=poll_id)
5         return render_to_response('polls/detail.html', {'poll': p})
```

Let's say now that I want to reuse that particular data (the 'p' variable) in another view: given that the return value is always an HttpResponse instance, you are screwed;

sometimes you just need the data, to find something, or simply to render it in another format like JSON or XML (RESTful architectures, anyone?). This goes pretty much against the DRY principles, and if you don't go deeper than the Django tutorials, your whole application might feature lots of repeated code.

Even worse, you have a direct reference to a template ("polls/detail.html"), and this kind of coupling does not scale well. It can become a real problem in big projects.

There are, however, strategies to avoid this: the first, the most common, is to refactor your code and to create a "layer" of data-specific functions, which will return instances (or arrays thereof) that you can reuse here and there. Doing this in a big project already started requires a good deal of unit testing first, to ensure that your refactoring is not breaking something elsewhere, but that's another problem (because you DO unit test, right??). This approach might not scale well in complex projects, and thus you would like to organize your code in other ways.

I learnt about organizing views using callable objects instead of functions while studying the code in the Django REST Interface project. In this case, you create code like this:

```
1  class Resource(ResourceBase):
2          """ Generic resource class that can be used for resources that are not based
               on Django models. """
3          # ... snip ...
4          def __call__(self, request, *args, **kwargs):
5                  """ Redirects to one of the CRUD methods depending on the HTTP method
                       of the request. Checks whether the requested method is allowed for
                        this resource. """
6                  # Check permission
7                  if not self.authentication.is_authenticated(request):
8                          response = HttpResponse(_('Authorization Required'), mimetype=
                               self.mimetype)
9                          challenge_headers = self.authentication.challenge_headers()
10                         response._headers.update(challenge_headers)
11                         response.status_code = 401
12                         return response
13                 try:
14                         return self.dispatch(request, self, *args, **kwargs)
15                 except HttpMethodNotAllowed:
16                         response = HttpResponseNotAllowed(self.permitted_methods)
17                         response.mimetype = self.mimetype
18                         return response
```

The important bit here is the "__call__" method, which allows an instance to be called

as a function, without specifying any particular method. This makes me remember of the dreadful VB default methods but in Python it's not that bad, actually (VB is horrible by default anyway), and allows you to use a cool syntax to do complex tricks ("command pattern" way of doing things, without the method call overload). And of course, since you are using an object-oriented approach, you can use polymorphism and inheritance to organize and reuse code as much as you can (or want).

Finally, Marco told me that his team uses another cool approach: they avoid returning HttpResponse instances from the views, and instead use Python decorators to generate those. This way, you can achieve another neat separation of concerns, and you can reuse code simply and effectively.

I understand that the Python philosophy cares about explicitness, but the "easy" way of processing requests in Django leads to trouble in big applications: increased coupling, reduced DRY, more headaches. I think you should use some code-reuse strategy in your Django code, but this, of course, is more an architectural problem than a Django problem.

Code

SQLite Blessing

http://kosmaczewski.net/2008/03/09/sqlite-blessing/

I just found this in the SQLite source code, just fantastic:

```
1  /************** Begin file sqliteInt.h ****************************************/
2  /*
3  ** 2001 September 15
4  **
5  ** The author disclaims copyright to this source code.  In place of
6  ** a legal notice, here is a blessing:
7  **
8  **    May you do good and not evil.
9  **    May you find forgiveness for yourself and forgive others.
10 **    May you share freely, never taking more than you give.
11 **
12 *************************************************************************
13 ** Internal interface definitions for SQLite.
14 **
15 ** @(#) $Id: sqliteInt.h,v 1.658 2008/01/30 16:14:23 drh Exp $
16 */
17 #ifndef _SQLITEINT_H_
18 #define _SQLITEINT_H_
```

Blow your mind

http://kosmaczewski.net/2008/03/11/blow-your-mind/

Take a careful look at this:

```
1  #include <iostream>
2
3  class Gadget
4  {
5  public:
6      void sayHello() const
7      {
8          std::cout << "Gadget!" << std::endl;
9      }
10 };
11
12 class Widget
13 {
14 public:
15     void sayHello() const
16     {
17         std::cout << "Widget!" << std::endl;
18     }
19 };
20
21 template <class T>
22 class OpNewCreator
23 {
24 public:
25     T* create()
26     {
27         std::cout << "Using␣'new':␣";
28         return new T;
29     }
30 };
```

```
31
32 template <class T>
33 class MallocCreator
34 {
35 public:
36     T* create()
37     {
38         std::cout << "Using 'malloc': ";
39         void* buf = std::malloc(sizeof(T));
40         if (!buf) return 0;
41         return new(buf) T;
42     }
43 };
44
45 template <template <class Whatever> class T, class B>
46 class Creator : public T<B>
47 {
48 public:
49     void exec()
50     {
51         B* obj = this->create();
52         obj->sayHello();
53         delete obj;
54     }
55 };
56
57 typedef Creator<MallocCreator, Widget> Manager;
58
59 int main (int argc, char * const argv[])
60 {
61     Manager obj;
62     obj.exec();
63     return 0;
64 }
```

Try changing "MallocCreator" by "OpNewCreator" and "Widget" by "Gadget" in the typedef of line 57, recompile and run; of course you can provide default values, so that

```
1 ...
2 template <template <class Whatever> class T = MallocCreator, class B = Widget>
3 class Creator : public T<B>
4 ...
```

so that you just do

```
1  ...
2  typedef Creator<> Manager;
3  ...
```

I've just started reading "Modern C++ Design" by Andrei Alexandrescu and I've already my head spinning out of orbit. This is amazing (and by giving a quick look at the rest of the book, there's even more incredible stuff there)!

Templates

http://kosmaczewski.net/2008/03/13/templates/

Did you knew this is possible in C++? I didn't.

```
1  void Fun()
2  {
3      class Local
4      {
5          //... member variables ...
6          //... member functions ...
7      };
8
9      // ... code using Local ...
10 }
```

This feature is called "local classes" and is part of the standard; the limitations are that these local classes cannot have static member variables and cannot access nonstatic local variables. But, as Alexandrescu points out (page 28), you can use them in template functions:

```
1  class Interface
2  {
3  public:
4      virtual void Fun() = 0;
5      // ...
6  };
7
8  template <class T, class P>
9  Interface* makeAdapter(const T& obj, const P& arg)
10 {
11     class Local : public Interface
12     {
13     public:
```

```
14          Local(const T& obj, const P& arg)
15              : obj_(obj), arg_(arg) {}
16
17          virtual void Fun()
18          {
19              obj_.Call(arg_);
20          }
21
22      private:
23          T obj_;
24          P arg_;
25      };
26
27      return new Local(obj, arg);
28  }
```

Not only that, but partial template specialization and template-based compile-time verifications just blew me away. The second chapter of the book was realizing I just haven't had the slightest clue about all the cool things you could do with C++!

The beauty of Cocoa

(Highly geeky post ahead. You've been warned!)

I have found the very message that summarizes the beauty of Cocoa in a single word; see by yourselves, hereunder in line 47:

```
1  #import <Foundation/Foundation.h>
2
3  // The interface of a person
4  @interface Person : NSObject {
5      NSString* firstName;
6      NSString* lastName;
7      int age;
8  }
9  @end
10
11 // The implementation of the Person
12 @implementation Person
13 -(id)init {
14     if (self = [super init]) {
15         firstName = @"";
16         lastName = @"";
17         age = 0;
18     }
19     return self;
20 }
21
22 -(void)dealloc {
23     [firstName release];
24     [lastName release];
25     [super dealloc];
26 }
27
28 -(NSString*)description {
```

```
29      return [[NSString alloc]
30              initWithFormat:@"Name: %@ %@, %d years old",
31              firstName, lastName, age];
32  }
33  @end
34
35  // Some code using the Person class
36  int main (int argc, const char * argv[]) {
37      NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
38
39      NSMutableDictionary* dict = [[[NSMutableDictionary alloc] init] autorelease];
40      [dict setObject:@"Teto" forKey:@"firstName"];
41      [dict setObject:@"Rodriguez" forKey:@"lastName"];
42      [dict setObject:[[NSNumber alloc] initWithInt:34] forKey:@"age"];
43
44      Person* person = [[[Person alloc] init] autorelease];
45
46      // The beauty of Cocoa can be resumed to this very line:
47      [person setValuesForKeysWithDictionary:dict];
48
49      // Now sit back and relax:
50      NSLog([person description]);
51
52      [pool drain];
53      return 0;
54  }
```

Project Management

On the Need of Minimalist Polyglots

http://kosmaczewski.net/2008/05/12/minimalist-polyglots/

Many companies, at some point of their history, ask themselves a simple question: what programming language should I use? The answer to this question is tricky, and has big, big consequences, for every single line of code of your future products will be written, read and suffered by it. This single choice defines the level of salaries you will have to pay, the skills of programmers you will have to deal with, the relative length and performance of your systems, the availability of tools (or lack thereof), the kind of support you will get (or not), the number of operating systems your code will work in, etc.

Given the fact that Web Development equals Software Development, this discussion will be of interest to those building the smallest websites, as well as old desktop-intensive apps. It will not be a "Tell me what programming language you use, and I will tell you who you are" type of article, though it may look like one, because that is something you have to figure out all by yourself.

If you take a look at the list of programming languages, any business person would have an instant headache. There are lots of them. With the strangest names. You cannot possibly guess which one to pick from such a list, obviously. So, how do companies choose the languages they use? There are some straightforward methods that I have seen so far, in no particular order:

Looking at what other companies use (typically Google, Microsoft, Apple or Sun, but it could be 37signals too). Following the advice of the CIO, the Lead Architect or some other politically-powered person, which might or might not have read this ar-

ticle ;) Looking at what the current pool of programmers in the company know how to use. Rinse, wash, repeat. Following hype. Because there is a market plenty of available, cheap programmers that I could use for this project. Taking into account the characteristics of the languages themselves (static vs. dynamic, etc). Following the company's history of past projects (successful or not). Following what your the client suggests (or mandates). I think that it is a very bad idea to take any of the above methods in isolation, without considering other factors. Doing so is a path to self-destruction in the medium to long term, even if you succeed in the short term.

Just as a small background for people not into programming: you can safely (roughly) group programming languages in a table like this:

Static Dynamic Strongly typed Java, Objective-C, Pascal, . . . Python, Ruby, JavaScript, Objective-C, Lisp, . . . Weakly typed C++, C, . . . JavaScript, VBScript, . . . In a static language all variable type references are bound at compile time; in a dynamic language, this is done at runtime (which allows you to assign a string or an int to the same variable). In a strongly-typed language, either the compiler or the runtime enforces the operations you can and cannot do on an object (depending on its type, as you may have guessed). In a weakly-typed one, there is no such restriction, and you can perform implicit conversions from one type to the other. And then you have functional ones, but that is another problem, because there are many programming paradigms out there. And then there is the hybrid ones, which I love as Steve Yegge does:

But also there's, like, the Boo language, the io language, there's the Scala language, you know, I mean there's Nice, and Pizza, have you guys heard about these ones? I mean there's a bunch of good languages out there, right? Some of them are really good dynamically typed languages. Some of them are, you know, strongly [statically] typed. And some are hybrids, which I personally really like.

He did not include Objective-C as "hybrid", but I think it is. Anyway, so much for the theory, here is the main point of this article:

First of all, I consider programming languages (I know a few of them, and I have my personal picks) just as tools to get things done™. Nothing else. I think of them as hammers or Black & Decker screwdrivers or saws or nail guns.

Second, I believe that specialization is for insects. Getting stuck in a single programming language because of any of the reasons I have enumerated above is just stupid.

So what I want to say is: You need polyglot programmers in your team, like you

need a team of people knowledgeable in many human languages in every company doing business at global scale. I would say even more, you need not only people fluent in western languages (like English, French, Spanish and Italian, which is my combination) but also in other languages, with different paradigms behind, like Arab, Hebrew, Hindi or Chinese.

What does that mean in programming terms? You want to have programmers in your team being able to use different languages in different ways; you want to have programmers that learn new languages every so often, just for the sake of it. And most importantly, you want to get rid of programmers that not only get stuck on a single language, but that, even worse, dismiss any other way to do things. Having people that takes a negative look on the learning side of things can bring your whole company to a dead-end. This industry is plenty of integrists, and you do not want that in your company.

For example, JavaScript can be used as a procedural, object-oriented or functional programming language. Does your JavaScripters know how to write functional JavaScript? If not, that is too bad, because they will not be able to fully understand what is going on behind the scenes in Prototype or jQuery then. Of course this will not block them from using those libraries, but they might not understand how to apply some interesting patterns in their own code.

Ask more about your programmers: do they know how to do complex C++ template metaprogramming? Are they aware of some performance problems brought by garbage collectors? Do they follow the latest evolutions of the next version of JavaScript? (even if they do not like them) Which blogs do they follow? Do they know why some people hate PHP? Do they know what a continuation server is? Which programming books have they read lately?

And finally, what is even more important than having chosen a good programming language? Your methodology. Ask yourself (or your team) about these points:

Do you write unit tests? You might not have testers, which is a bad idea anyway, but that does not block you from testing code yourself (Steve Yegge): And I would say it's a pain in the butt, but I mean... it's a pain in the butt because... a static type-systems researcher will tell you that unit tests are a poor man's type system. The compiler ought to be able to predict these errors and tell you the errors, way in advance of you ever running the program.

Do you use defensive programming techniques? Do you have a wiki, a project website

or at least good documentation in your code? If not, how do your new hires learn about your product? No, reading the code is NOT a good answer. Is your chosen programming language portable? You might think that your system will never have to run on Linux or Mac, but why stuck yourself on purpose? The common factor of all the above items is that you have to manage complexity. If you write code, you are creating complex stuff, and one of the best ways to manage complexity is to create small systems; as Steve Yegge said:

Small systems are not only easier to optimize, they're possible to optimize. And I mean globally optimize.

And this is why you do not only need polyglot, but also minimalist programmers in your team. Small is beautiful. Paul Graham (of Y Combinator fame) knows that dynamic languages yield small systems:

The right tools can help us avoid this danger. A good programming language should, like oil paint, make it easy to change your mind. Dynamic typing is a win here because you don't have to commit to specific data representations up front. But the key to flexibility, I think, is to make the language very abstract. The easiest program to change is one that's very short.

And how can you get small programs? By choosing the right programming language. Which brings us to the beginning of this post! So, here go some tips for all of you looking for the right programming language:

Prefer strongly-typed, dynamic languages; there are a lot of reasons for that, particularly those exposed by Steve Yegge in his excellent presentation at Stanford: Yeah, sure, it catches a few trivial errors, but what happens is, when you go from Java to JavaScript or Python, you switch into a different mode of programming, where you look a lot more carefully at your code. And I would argue that a compiler can actually get you into a mode where you just submit this batch job to your compiler, and it comes back and says "Oh, no, you forgot a semicolon", and you're like, "Yeah, yeah, yeah." And you're not even really thinking about it anymore.

Which, unfortunately, means you're not thinking very carefully about the algorithms either. I would argue that you actually craft better code as a dynamic language programmer in part because you're forced to. But it winds up being a good thing.

Some points about his talk: Many of the caveats of dynamic languages are not (so) true anymore (like the lack of decent IDEs or the performance problems); There is a lot of research going on nowadays on the performance of programming languages, and

this means that code written in these languages will benefit from many improvements in the near future, for free; And yes, there is the hype factor I have mentioned above; the cool kids are using them: Which makes them exactly the kind of programmers companies should want to hire. Hence what, for lack of a better name, I'll call the Python paradox: if a company chooses to write its software in a comparatively esoteric language, they'll be able to hire better programmers, because they'll attract only those who cared enough to learn it. And for programmers the paradox is even more pronounced: the language to learn, if you want to get a good job, is a language that people don't learn merely to get a job.

Teach yourselves; setup internal workshops to have all your team learn how to do cool stuff with those languages you have read about in DDJ. Teach the community around you, and do not be scared of competition; have your team write papers, articles on business or technical journals, publish code as open source projects, and show that you can go beyond. Keep an open mind: continuation servers, Comet or multicore processors are the future. Is your team prepared? Software is a social process. Once you get this in your mind, and get your team working proactively, collaboratively and teaching each other, the choice of a programming language comes in second place.

Saving a Failing Project

http://kosmaczewski.net/2008/08/11/saving-a-failing-project/

In 2006 I had the opportunity to work as a "project leader" into a small failing project. Three developers were working in an ad hoc basis, creating a software application for an important client (a government office in Lausanne), without any kind of detailed formal specification, without any kind of design documentation, and with strong pressure from the management to release the application, even if not in an usable state. Needless to say, the project was also beyond budget.

I had just joined this company a couple of days ago, and the management asked me to take the project in charge. Not an easy task, particularly because it was my first experience of this kind.

The client was pushing to get the software it had paid for (it was a desktop reporting application for the Police department), and had not got any kind of preview yet. So the first thing I did is to pick up my copy of "Leading a Software Development Team" book and read chapter 2, "I'm taking over the leadership of an existing project // where do I start?" and get a thorough read:

The first thing that you should start to do is to review the situation. This involves more than just absorbing impressions; you need to organize these impressions into a framework. Try to organize your thoughts into the following areas, and in each area try to separate technical issues from personnel ones: - Where is the team now? (. . .) - Where is it supposed to be getting to? (. . .) - How does the team currently intend to continue?

(Whitehead, page 17)

Another highly pragmatic resource was Joel Spolsky, and his "Joel Test":

The neat thing about The Joel Test is that it's easy to get a quick yes or no to each question. You don't have to figure out lines-of-code-per-day or average-bugs-per-inflection-point. Give your team 1 point for each "yes" answer.(. . .) A score of 12 is perfect, 11 is tolerable, but 10 or lower and you've got serious problems. The truth is that most software organizations are running with a score of 2 or 3, and they need serious help, because companies like Microsoft run at 12 full-time.

(Spolsky, 2000)

The "Joel Test" result for this team was 2 when I joined the team (they just had source control and good tools). When I left the company, they were running at 9 (we just did not have candidates writing code during interviews, nor testers, nor hallway usability testing).

For this project I took the following decisions:

Since the priority for the client was to see results, I asked the developers to concentrate on stabilizing "visible" features, particularly on a visual report editor, that used a complex set of controls, similar to those of a drawing application, to create reports. Doing this, we could have a stabilized preview version that we showed to the client as early as one week after my arrival to the project. In agreement with the developers, we set up a daily build procedure, and I also asked them to provide a "client build" every Wednesday, that would be placed in a public directory available to the client. It turns out that the client never downloaded the binaries, but they liked to see the version numbers grow, and the binaries being delivered. Every week, Wednesday was the "public build" day, Thursday was the "bug correction" day, and Friday, Monday and Tuesday were "new features day". Small stand-up meetings every day allowed us to know what was going on. Another important concern from the developers' side was to have a quiet environment to work. They were constantly interrupted by the (quite nervous) managers to see their progress, and as such, I decided to stand in between both; I asked them to not to interrupt the developers for any reason, and to ask me for updates. I became a "proxy" between both, which reduced the tensions, and brought some peace to the developers. I created a fast project plan in our Intranet (there wasn't any, so tracking the project was next to impossible) by asking the developers about the tasks they needed to do to finish the project, with the estimated time to do them, and setting some milestones. Since the project was in a wiki page, the developers could change the time estimations in case that they felt they had made a mistake; the only condition being to notify me of these changes. Using that information, I could create a couple of reports for everyone to see, and bring more visibility to the project: I wrote

a weekly report stating the week's achievements, the status of the project (number of open bugs, new functionality available, etc). In the intranet, I set up a couple of graphs and report tables, which were automatically updated every day. I did not take any technical decisions about the project; I gave full authority on this matter to the lead developer, who in turn appreciated this trust and took spontaneously the decision of documenting and unit testing the system thoroughly doing extra hours every day. This boosted the morale of the team, and the quality of the application as well. The other two developers contributed to these tasks as well, and the rhythm of releases and their quality increased in a couple of months. It turns out that the architecture of the system was particularly well done, and as such, adding new features was a relatively simple task, once the underlying framework was done; of course, during that time no visible results were available, which made everyone nervous. Looking backwards, the only technical decision I've needed to take during this project was to use the company wiki; there I could add information pages that everyone contributed to, reducing the number of communication channels and reducing the misunderstandings between project team members. I cannot stress how much this helped; it provided a complete dashboard for everyone to refer to.

The most important problems in this project were human and customer related ones. By providing more visibility to the project, and by reducing the signal-to-noise ratio in the communication channels between developers and management, the team was able to provide the customer with a more reliable and full-featured product.

References

Joel Spolsky, "The Joel Test: 12 Steps for Better Code", Wednesday, August 9th, 2000 [Internet] http://www.joelonsoftware.com/articles/fog0000000043.html (Accessed June 8th, 2007)

Whitehead, R.; "Leading a Software Development Team - A Developer's Guide to Successfully Leading People & Projects", Addison-Wesley, 2001, ISBN 0-201-67526-9

Adding Manpower

http://kosmaczewski.net/2008/08/08/adding-manpower/

Published in 1975, "The Mythical Man-Month" is considered an all-time classic in the software engineering field. The book author, Frederick P. Brooks Jr., used his experience as the project manager of the IBM System/360 and its software, the Operating System/360, to explain a common set of problem patterns, applicable to other software projects as well.

One of the most famous citations in the book is the one regarding the consequences of adding human resources to a late project; this article will provide a couple of thoughts about this assertion, and highlight some contrariwise opinions.

The Mythical Man-Month

The second chapter of Brooks' masterpiece is named exactly as the book, "The Mythical Man-Month"; the core argument of this chapter is that the most frequent factor of project failure is schedule and time estimation. Brooks states that this is due to the fact that

Men and months are interchangeable commodities only when a task can be partitioned among many workers with no communication among them. This is true of reaping wheat or picking cotton; it is not even approximately true of systems programming. When a task cannot be partitioned because of sequential constraints, the application of more effort has no effect on the schedule. The bearing of a child takes nine months, no matter how many women are assigned.

(Brooks, pages 16 & 17)

The final phrase of the above paragraph is often used as a graphical depiction of the nature and meaning of Brooks' law. It implies the strong need for communication and integration existing in software projects; being social processes, software requires a

strong network of communication between team members, allowing them to coordinate the inherent set of interdependencies that every project has.

After an interesting analysis of common time overrun situations, Brooks ends this chapter with the following conclusion, which contains the enunciation of the law itself:

Oversimplifying otrageously, we state Brooks's Law: Adding manpower to a late software project makes it later. This is then the demythologizing of the man-month. The number of months of a project depend upon its sequential constraints. The maximum number of men depends upon the number of independent subtasks. From these two quantities one can derive schedules using fewer men and more months. (The only risk is product obsolescence.) One cannot, however, get workable schedules using more men and fewer months. More software projects have gone awry for lack of calendar time than for all other causes combined.

(Brooks, pages 25 & 26)

This "law" is known and cited throughout the industry as an example of a common pattern, observed once and again in different projects all over the world:

Fact 3: Adding people to a later project makes it later(. . .) Intuition tells us that, if a project is behind schedule, staffing should be increased to play schedule catch-up. Intuition, this fact tells us, is wrong. The problem is, as people are added to a project, time must be spent on bringing them up to speed.(. . .) Furthermore, the more people there are on a project, the more the complexity of its communication rises.

(Glass, page 16)

As a personal experience, I must say that the lecture of this book opened my eyes more than many, many other books. It is a funny read, but also an enlightening one: many anecdotes told by Brooks strangely correspond to my own experience, and this one is no exception. I have seen projects gone unfortunately late because of the simple fact of adding more people; and in one particular case, the project was cancelled altogether. These projects had several factors in common, though:

Bad documentation, or lack thereof; the only way for newcomers to the project to know what was going on was interrupting the other developers, disrupting the current operations on the project; I think that a good set of documents, describing both the high-level architecture and the low-level APIs are needed for new developers to jump in and catch up. It's maybe not enough, but a good leap forward anyway. Lack of architectural vision; projects that do not have an architect, providing vision and

technical leadership to the team, are in my opinion exposed to problems when more developers join the project. The architect can act as a proxy person, guiding new developers while they familiarize themselves with the project, isolating other developers from this task. Bad project decomposition in components; if the system to be developed is sufficiently large, and the decomposition in components is not properly done, the overlap and extended communication paths among team members might affect the whole project negatively. A good decomposition breaks down the whole project in a set of smaller ones, with the corresponding set of interfaces, which brings the whole team to work separately on different subsystems. In these, the risk of getting later for adding manpower is reduced proportionally. Bad working conditions; I positively think that open spaces are a common disease in our industry. Teams working in open spaces suffer more of noise and visual distractions, and this is more evident when new team members join the project. Criticism

However famous, Brooks' law has had a good deal of criticism as well, regarding the specific characteristics of projects that might be affected in case that new people is assigned to them. The OS/360 project, which served as the basis for Brooks' work, might not be similar to other projects, and as such, the law would not necessarily apply to them:

For Brooks' Law to be true, the amount of training effort required from existing staff must be significant. The amount of effort lost to training must exceed the productivity contributed by new staff when they eventually become productive. (. . .) "Late" chaotic projects are likely to be much later than the project manager thinks–project completion isn't three weeks away, it's six months away. Go ahead and add staff. You'll have time for them to become productive. Your project will still be later than your plan, but that's not a result of Brooks' Law. It's a result of underestimating the project in the first place.(. . .) Controlled projects are less susceptible to Brooks' Law than chaotic projects. Their better tracking allows them to know when they can safely add staff and when they can't. Their better documentation and better designs make tasks more partitionable and training less labor intensive. They can add staff later in the project with less risk to the project.

(McConnell, 1999)

Scott Berkun gives a more concrete analysis on why the law could be wrong:

It depends who the manpower is. The law assumes that all added manpower is equal, which is not true. Some teams can absorb more change than others. Some teams are more resiliant to change. There are worse things than being later. (. . .) That can be ok

if you also get higher quality There are different ways to add manpower. (. . .) The more experience everyone has with mid-stream personnel changes, the better. It depends on why the project was late to begin with. (. . .) no amount of programming staff modifications will resolve the psychiatric needs of team leaders or the dysfunctions of executives. Adding people can be combined with other management action. (. . .) if you're removing your worst, and most disruptive, programmer and adding one of your best, it can be a reasonable choice. (Berkun, 2006)

And what about open source projects? Many of these (Linux, Apache, MySQL) are potentially among the biggest software projects ever undertaken, and they don't appear to suffer o fthe problems pictured by Brooks' law:

But proponents of open source and free software development, including Linux developers, are not completely satisfied with the Law. Most famously (among geeks at any rate), Eric Raymond in his "The Cathedral and the Bazaar," declared Brooks' Law obsolete, if not simply limited, saying "if Brooks' Law were the whole picture, Linux would be impossible." Although Raymond now says that he has somewhat modified his views or was misunderstood, some still would say he is given to oversimplifying and outrageousness himself. "I don't consider Brooks' Law 'obsolete' any more than Newtonian physics is obsolete; it's just incomplete. Just as you get non-Newtonian effects at high energies and velocities, you get non-Brooksian effects when transaction costs go low enough. Under sufficiently extreme conditions, these secondary effects dominate the system — you get nuclear explosions, or Linux."

(Jones, 2000)

Conclusion

So far, the discussion seems to be open. There might be a scale factor for projects, which in turn might expose them to be affected by Brooks' law. I think that research is needed to arrive to a conclusion, even if it will be a statistical one.

Other important facts highlighted in the book are the "second system phenomenon", the productivity advantage of using high-level languages, and the importance of building a prototype - "one to throw away". I can only recommend this book to everyone interested in the field of software engineering (which I did in my own review of classic books in this blog: http://kosmaczewski.net/2005/11/20/my-bookshelf-part-iii/)

References

Berkun, S.; "Exceptions to Brooks' Law", January 11th, 2006, [Internet] http://www.scottberkun.com
to-brooks-law/ (Accessed June 8th, 2007)

Brooks Jr., F. P.; “The Mythical Man-Month - Essays on Software Engineering, Anniversary Edition”, 1995, Addison Wesley, ISBN 0-201-83595-9

Glass, R. L.; “Facts and Fallacies of Software Engineering”, Addison-Wesley, 2003, ISBN 0321117425

Jones, P.; “Brooks’ Law and open source: The more the merrier?”, IBM, May 1st, 2000, [Internet] http://www.ibm.com/developerworks/linux/library/os-merrier.html (Accessed June 8th, 2007)

McConnell, S.; “Brooks’ Law Repealed?”, IEEE Software, November/December 1999 [Internet] http://stevemcconnell.com/ieeesoftware/eic08.htm (Accessed June 8th, 2007)

Certification

http://kosmaczewski.net/2008/08/05/certification/

While several other professions have a long, established and standard procedure of certification, the title “software engineer” is applied to both self-made developers, turned into experts of some technique, or to people with PhD degrees, and a long history of both academic and professional achievements.

When in some situations it is not legally possible to use the title “software engineer” without an engineering degree of some kind (for example, in some states of the USA or some institutions like the IEEE - http://www.ieeeusa.org/policy/positions/titleengineer.html), the term “software developer” is usually applied to people in charge of designing, writing and / or maintaining software-based systems. I will use the terms developer and engineer interchangeably in this discussion, which some people might think is not correct.

The discussion about the need of a formal certification process is a relatively new one:

Professional certification in the IT industry is a relatively recent phenomenon. It was begun in the late 1980s by Novell, Inc., an upstart networking vendor from Provo, Utah, in an effort to build market share and manage support costs for its products by building the skill levels of the people who worked with those products. Novell was one of the first companies to recognize the links between education/skills and product success. They knew that they could not build an education infrastructure that would support their worldwide marketing plans with their own resources. However, they also recognized that if they did not provide for skills acquisition for their highly technical products, they could never meet their product revenue goals.

(Shore)

However, no consensus about whether or not certification is needed has been reached

yet. This article will highlight some of the problems raised by software engineering certification, which might explain the lack of consensus cited before:

The first one has to do with the inherent extension of the software engineering field: are all software developers equal? The second one has to do with the large number of available certifications: which one to choose? Which ones are "reliable" indicators of expertise, and in which fields? What is a "Software Engineer"?

In my career, I've found self-made people (I'm one of them, actually), real-estate architects, lawyers, mathematicians, economists and even geophysicists writing code for a life. What I've seen so far is that the most successful software developers are those who like doing it, no matter which profession they've followed. And the opposite is also true: many guys with a computer science degree discover, some time after they start their careers, that they definitely do not like that code thing.

One of the biggest problems with certifications is that there is not such thing as a "single kind" of software developer:

There are those who write games, and spend most of their time writing in low-level languages for game consoles, optimizing for speed and space, and creating three-dimensional worlds using as little memory as possible... There are those who write web-based applications, and spend their time creating 3-tier architectures, talking to a database, using some kind of object-oriented platform, and luckily exposing some data using XML web services, dealing with cross-browser issues, and wondering what is all that fuss about Web 2.0... There are those who write operating systems, and work for some embedded software company, or hack Linux kernel device drivers every night, or work for Microsoft or Sun or Apple, and spend most of their time discussing whether microkernels are better than monolithic architectures... There are those who have the ill fate of working as a consultant, and spend more time switching from project to project every day, or dealing more with corporate politics, rather than with code... There are those who manage projects and spend more time in their mailing list or in Microsoft Project rather than being able to code (and then complain about this in their blogs)... There are those who have a software engineering degree, but work for ZDNet writing about industry trends... There are those who turn into human resource consultants, and try to keep up to date on the new trends, but feel completely lost given what they learnt in university... There are those who do a little bit of all what I've mentioned above, and are or not really good at all of them... And finally there are those who might fit any of the characteristics above, but would have preferred not to listen their parents and rather open that scuba-diving shop in

Honolulu. Available Certifications

This diversity explains the existence of more common product-specific certifications: you can be certified to use Microsoft technologies (http://www.microsoft.com/learning/mcp/default.mspx), MySQL databases (http://www.mysql.com/certification/), Apple servers (http://www.apple.com/xserve/rai various IBM products (http://www-03.ibm.com/certify/certs/index.shtml), Java development stacks (http://www.sun.com/training/certification/java/index.xml), Cisco routers (http://forums.cisco.com/eforum/servlet/CCNP?page=main), RedHat Linux installations (https://www.redhat.com/training/certification/) or UML diagrams (http://www.omg.org/uml certification/index.htm)

However, given that technology companies have interest in having many people taking their certifications, their affordability and low-entry barriers to get them, many of these become much easy to get than they should be, and as a result, they lose credibility, and do not help IT recruiters to filter properly software developers during the selection processes. I've heard many complaints of project managers regarding these certifications, and I think it's a generalized feeling:

"Certified skills pay has not just flat lined, it's in the negative. This is big news if you're certified and you're thinking about getting recertified," said Foote. "This trend is in the fourth quarter, that pay for certifications is on the wane, while non-certified skills are growing in pay."(. . .) Certifications are losing value because employers are looking for more in their workers than the ability to pass an exam; they want business-articulate IT pros."

(Rothberg, 2006)

Bruce Schneier, a well-known security researcher, has written about security certifications as well, with a mixed feeling:

In the end, certifications are like profiling. They work , but they're sloppy. Just because someone has a particular certification doesn't mean that he has the security expertise you're looking for (in other words, there are false positives). And just because someone doesn't have a security certification doesn't mean that he doesn't have the required security expertise (false negatives). But we use them for the same reason we profile: We don't have the time, patience, or ability to test for what we're looking for explicitly.

(Schneier, 2006)

The conclusion of all of this is that the debate is pretty much still open, and that there is not a simple answer to it.

Market Fragmentation

There is an interesting anonymous comment in Schneier's website as well:

Another thought on certification is they are not all equal.

There are Vendor Certs. Microsoft's MCP/MCSE, CISCO CCNA/CCNP/CCIE Pro: The canidate is likley to know how to work on your specific platform. Con: The canidate is likely to think in only the vendor's interest.

There are Certs to assure knowledge of standard security terminlogy. ISC CISSP Pro: Can talk strategy and evaluate the nine domains to evaluate how the company is doing overall Cons: Most likely could not tell you what the nineth byte of an ip packet means or if OpenSSL is out of date on Red Hat Linux.

Topic specific, vendor neutral. SANS GIAC Pro: Vendor neutral. A lot of focus on specific skills in NIDS or Hardening Windows, Incident Handeling, etc. Con: Concentration on open source tools since they are easily available, but it does not seem to impress all employers.

(@nonymou5, in Schneier, 2006, spelling mistakes not corrected)

I think that this comment summarizes pretty well another problem with certifications: there is a great level of fragmentation in today's market. Every single important technology in the IT world requires a huge investment in time and practice in order to master it, and this translates in a huge complexity for the developers to choose the right certification. All of these without taking into account the large number of IT-related university degrees available, online or not.

Conclusion

The term "software engineer" is sufficiently vague, and the number of "certifications" sufficiently large, as to allow a single "yes or no" answer to whether professionals in the software sector should be certified or not. I personally think that I would rather avoid vendor-specific certifications as far as possible, and choose university-related or problem domain related certifications instead, to keep my career options open, and my mind free of marketing.

References

Rothberg, Deborah; "Another Nail in the IT Certification Coffin", Novembrer 3rd, 2006, [Internet] http://www.eweek.com/article2/0,1895,2051272,00.asp (Accessed June 3rd, 2007)

Schneier, Bruce; “Security Certifications”, July 20th, 2006, [Internet] http://www.schneier.com/blog/archiv
(Accessed June 3rd, 2007)

Shore, Julie; “Why Certification? The Applicability of IT Certifications to College and
University Curricula”, [Internet] http://www.developer.ibm.com/university/scholars/certification/ebusines
certification.pdf (Accessed June 3rd, 2007)

The Truth Be Told

http://kosmaczewski.net/2008/01/22/the-truth-be-told/

Reg describes 99% of all available programming jobs with incredible sincerity:

> You do a clerk's job, you settle for a clerk's working conditions and wages, but you take solace in the thought that you are somehow more than a clerk, because you have a university degree and the dental technician who cleans your teeth doesn't.
>
> Only everyone knows it's a sham, especially the hiring manager who puts "University degree required" in the job advertisement. He wants to hire a clerk, someone who will work long hours doing as they're told in a top-down, hierarchal command structure. Does that job sound like there is any Science involved? Of course not, everyone knows that, it's why the industry is trying to weed all of the Science out of a Computer Science degree.

I do not have a university degree. Heck, I started university 4 times and finished none (Physics, Economics, Marketing and even Computer Science! :) I have been refused jobs because of this (particularly at the beginning), but I have been given jobs because of this too (particularly lately). Having some experience in your CV pays off; the hard thing is to start without that "paper".

As a matter of fact, I'm doing an online Master's degree right now, which I will finish this year, and I'm actually quite happy to have started. I've been able, in the past two years, to read books and papers that I had not heard about; I could understand some underlying issues in Software Engineering, both from technical and social points of view; I can understand more, I can learn more.

But, the truth be told, I'm also doing it to have that "paper" hanging on the wall. I know it's silly, but I want to take that step too, and that's why I've changed careers so many

times. However, that is a secondary thing for me: what I am looking is something else altogether; a bit of guidance in my own learning path. I have followed an unusual way in my career, and looking backwards I'm happy to have done that (mind you, it was not at all consciously!). That's why the 6 books and the new programming language every year. It's all part of the same pattern.

Of course, this has worked out for me, and your mileage may vary. I know excellent developers both with and without degrees, and also horrible professionals with and without them. Some people need a physical teacher in front of them, while I prefer to learn alone.

A degree, like any decision that you take in life, should mean something to us, and in that sense, I find Reg's arguments enlightening.

Challenges for Software Engineers

http://kosmaczewski.net/2008/08/03/challenges-for-software-engineers/

Software Engineering is the youngest of all the professions, being born around 50 years ago, but since then it has been continually improved. Practicers have fiercely debated upon it through the years, given the extremely fast pace of the innovations in the field, and the extremely difficult and inherently dynamic nature of software. Many trends have appeared and vanished, and many others will come.

In this article I will provide a short overview of two kinds of challenges that I consider that software engineers will have to confront in the next 20 years: the human and the technical.

The Human Factor

A quick look at the agenda of the 29th Int. Conference on Software Engineering (held in Minneapolis last year, from the 20th to the 26th May 2007) shows the key themes considered by the software engineering research community as the major challenges today:

“Improving Software Practice through Education: Challenges and Future Trends” “Research Collaborations between Industry and Academia” “Model-driven Development of Complex Systems: A Research Roadmap” “Source Code Analysis: A Road Map” “Software Reliability Engineering: A Roadmap” “Global Software Engineering: The Future of Socio-technical Coordination” “Collaboration in Software Engineering: A Roadmap”

"Self-Managed Systems: An Architectural Challenge" "Software Project Economics: A Road Map" (Source: ICSE 2007)

Mixed up with technical concerns, some presentations highlighted core problems that appears in the current state of software engineering: communication, collaboration and human issues.

The core substance of software deserves more eyes and more minds, thinking ways to describe not only the big picture (something that you can do with fancy diagrams) but also to give solutions to the problems that developers find daily while building systems up. Software is a process, but not any kind of process: a human one, maybe the most intangible of all processes; and as such, it is filled with all human brightnesses and failures.

(Myself, in 2006)

I have the deep, strong conviction that software development cannot and must not be separated from the human-side problems of forming, keeping and training teams, enhancing the internal and external communications, improving and enhancing the individual creativity as well as the ways of reaching team consensus. As a powerful example, the seminal Peopleware book by DeMarco and Lister showed that many of the most successful software companies have been those that excelled in creating human-centric environments:

In 1982, (Mitchell Kapor) founded Lotus Development Corporation, for which he is most noted. While there, he revolutionized corporate workplace culture by making diversity and inclusivity top priorities in his goal for creating an environment that attracted and retained employees. There were many "firsts" for Lotus, including being the first company to sponsor an AIDS Walk event in the mid-80's and refusing to do business with South Africa due to Apartheid.

(Sterling-Hoffman)

Thanks to a sharp hiring process, a series of innovations in their flagship spreadsheet product, and a progressive corporate culture, Lotus dominated the software landscape of the 80s. Today, Google follows very closely Lotus' steps (Google, 2007a), and their brilliant results in the last few years seem to confirm this trend. Google for example allows their employees to use 20% of their time in their own projects (Google, 2007b). This is resulting in an incredible amount of code, used internally and also released as open-source projects:

Google is a fantastic company to work for. I could cite numerous reasons why. Take the concept of "20 percent time." Google engineers are encouraged to spend 20 percent of their time pursuing projects they're passionate about. I started one such exciting project some time back, and I'm pleased to announce that Google is releasing the fruits of this project as an open source contribution to the Macintosh community. That project is MacFUSE, a Mac OS X version of the popular FUSE (File System in User Space) mechanism, which was created for Linux and subsequently ported to FreeBSD.

(Google Mac Blog, 2007)

The empowerment of both the individual and the team (the emphasis is important here) is key for a successful software project.

Parallelization

Herb Sutter has put it very clearly: technically speaking, since the beginning of the decade, there is no way for getting more processing power without jumping to multicore architectures:

The key question is: When will it end? After all, Moore's Law predicts exponential growth, and clearly exponential growth can't continue forever before we reach hard physical limits; light isn't getting any faster.(. . .) If you're a software developer, chances are that you have already been riding the "free lunch" wave of desktop computer performance.(. . .) Right enough, in the past. But dead wrong for the foreseeable future.

(Sutter, 2005)

The problem is that more cores do not necessarily mean more computing power, because the jump done by chip manufacturers has not (yet) been completely followed by the software community. Of course there is the concept of "threads", and multithreaded applications can benefit of performance boosts when running on multicore hardware platforms; however, a number of myths have to be debunked, as the common "2 x 3GHz = 6GHz" (as explained by Sutter here: http://www.ddj.com/showArticle.jhtml?documentID=c and even more importantly, creating multithreaded applications is not easy. At all.

A couple of months ago, in the "Questions and Answers" of LinkedIn.com I answered an interesting question about parallelization; the following excerpt of my answer pretty

much summarizes my opinions about the current state of multithreading, as well as some challenges that are raised for the future:

The problem is simply that the "streamline" programming languages languages do not provide good ways to code multithreaded applications. (...) Not at all. The problem is real, since multithreading applications are extremely complicated to think of, let alone develop properly. A line of code in a high-level language could mean several hundred instructions in a processor; and depending on the sharing algorithm used at the CPU level, each one of these instructions might be executed separately, sharing resources with other processes. So what happens when? (...) What I mean is that the fact that the JVM and the CLR support threads does not make good .NET or Java developers good multithreading developers by default. It's a different mindset; who is accessing your resources? (...) I think that as long as programming languages do not take multitasking and multithreading as base features (and not as mere library or API add-ons) we will continue struggling with single-threaded applications that collide with each other.

(Myself, this time on LinkedIn Answers, 2007)

I think that the challenge of parallelization is not only an extremely tough one, requiring what Thomas Kuhn calls a "paradigm shift", but also an extremely huge business opportunity; after all, while the top of the Chinese ideogram for "Crisis" means "Danger", the bottom part means "Opportunity" (Mary R. Bast, 1999).

Very Large Systems

I also think that software systems will invariably get bigger and bigger. And given the historically high risk of failure of software projects, the dependency on software of the modern society, the pervasiveness of the Internet, the low prices of connectivity and the overall globalization, it is more important than ever to get ready for those challenges.

In July 2006, the well known Software Engineering Institute of the Carnegie Mellon University published an impressive report (freely downloadable) called "Ultra-Large-Scale (ULS) Systems: The Software Challenge of the Future":

The study brought together experts in software and other fields to answer a question posed by the U.S. Army Office of the Assistant Secretary of the U.S. Army (Acquisition, Logistics & Technology): "Given the issues with today's software engineering,

how can we build the systems of the future that are likely to have billions of lines of code?" Increased code size brings with it increased scale in many dimensions, posing challenges that strain current software foundations. The report details a broad, multi-disciplinary research agenda for developing the ultra-large-scale systems of the future.

(SEI, CMU, 2006)

The 150-page long report gives an extremely detailed vision of the challenges raised by complex systems, in the following areas:

Design Monitoring Human interaction Computational Engineering Deployment Legal issues The report provides interesting conclusions, highlighting the methodologies and techniques that will required to tackle these systems efficiently, among them the role of the W3C, the forthcoming trends of grid computing and parallelization, the Model-Driven Architecture (MDA) initiative of the OMG, and finally the development of larger Service-Oriented Architectures (SOA) platforms, such as .NET or J2EE (page 41 of the report).

The report also places a strong emphasis in the concept of socio-technical ecosystems and I think it's worth a read by everyone interested in software engineering.

Conclusion

Given its youth, we have yet to see the most important developments in software engineering. However, it is extremely difficult to predict the future in this industry: Bill Gates himself published a book in 1995, "The Road Ahead", where he only slightly talks about the World Wide Web:

"The Road Ahead" appeared in December 1995, just as Gates was unveiling Microsoft's master plan to "embrace and extend" the Internet. Yet the book's first edition, with its clunky accompanying CD-ROM, mentioned the Web a mere seven times in nearly 300 pages. Though later editions tried to correct this gaffe, "The Road Ahead" remains a landmark of bad techno-punditry — and a time-capsule illustration of just how easily captains of industry can miss a tidal wave that's about to engulf them.

(Salon.com, 2000)

In any case, I think that there are important challenges in our industry: the need for better human management, the jump to multicore architectures and multiprocessing,

and the ever-growing size of software projects. These three elements will without any doubt change the shape of the industry in the years to come, and raise new challenges in turn.

References

Adrian Kosmaczewski on LinkedIn Answers, "For the software architects out there, do you feel there is an impending paradigm shift in the software development model, towards "parallel computing" models?", January 2007, [Internet] http://www.linkedin.com/answers?vi (Accessed June 3rd, 2007)

Adrian Kosmaczewski on Kosmaczewski.net, "What will the Software Architecture discipline look like in 10 years' time?", March 16th, 2006 [Internet] http://kosmaczewski.net/2006/03/ architecture-future/ (Accessed June 3rd, 2007)

Bast, Mary R; "Crisis: Danger & Opportunity", 1999 [Internet], http://www.breakoutofthebox.com/cr (Accessed June 3rd, 2007)

DeMarco, Tom & Lister, Timothy, "Peopleware - Productive Projects and Teams, 2nd Edition", 1999, Dorset House Publishing, ISBN 0-932633-43-9

Google, "Top 10 Reasons to Work at Google", 2007a [Internet] http://www.google.com/jobs/reasons.h (Accessed June 3rd, 2007)

Google, "What's it like to work in Engineering, Operations, & IT?", 2007b, [Internet] http://www.google.com/support/jobs/bin/static.py?page=about.html (Accessed June 3rd, 2007)

Google Mac Blog, "Taming Mac OS X File Systems", January 11th, 2007, [Internet] http://googlemac.blogspot.com/2007/01/taming-mac-os-x-file-systems.html (Accessed June 3rd, 2007)

ICSE, "Future of Software Engineering", 2007, [Internet] http://web4.cs.ucl.ac.uk/icse07/index.php (Accessed June 3rd, 2007)

Software Engineering Institute, Carnegie-Mellon University, "Ultra-Large-Scale (ULS) Systems - The Report", July 2006, [Internet] http://www.sei.cmu.edu/uls/ (Accessed June 3rd, 2007)

Salon.com, 2000 [Internet], "Why Bill Gates still doesn't get the Net", [Internet] http://archive.salon.c (Accessed June 3rd, 2007)

Sutter, Herb; "A Fundamental Turn Toward Concurrency in Software", 2005, [Internet] http://www.ddj.com/dept/architect/184405990 (Accessed June 3rd, 2007)

Sterling-Hoffman, “Opening Doors To Higher Education”, [Internet] http://www.sterlinghoffman.com/newsle
(Accessed June 3rd, 2007)

Wikipedia, “Thomas Kuhn” [Internet], http://en.wikipedia.org/wiki/Thomas_Kuhn (Accessed June 3rd, 2007)

Books

Best books of 2007

http://kosmaczewski.net/2008/01/23/best-books-of-2007/

I have several mantras in my life. One of them is to learn a new programming language every year. Another one is to read at least 6 technology-related books every year.

I've already talked about Erlang (and boy that was the most read article ever in the whole life of this blog! More than 1600 visits just for it!) so now it's time to discuss the greatest books I've read in 2007 (ordered by preference, from more to less):

Transcending CSS by Andy Clarke Founders at Work by Jessica Livingston The Old New Thing by Raymond Chen iWoz by Steve Wozniak Best Software Writing by Joel Spolsky Eric Sink on the Business of Software by Eric Sink

Transcending CSS

My big winner for this year. It's hard not to recommend this book enough, not only to those that work in the web design industry, but also to those that design, simply put. It's a beautiful book. It's a pleasure to read. It's a surprise, a bliss and a revelation, all in one.

Andy Clarke, a well-known designer in the UK, who is also now a member of the CSS committee at the W3C, tells us that HTML can and should be semantically correct, and that you can do a lot using the standard tags already available. Stop using those <DIVs> everywhere! You can give your page a meaning to begin with, and then apply a style on top of it. You can find inspiration in everyday life, and you can make all of this a truly cross-browser experience without much effort. Forget about your HTML editor; use Notepad or TextEdit or gedit and discover a new world.

This book has a deeper meaning and raison d'être, too; the web technologies are getting to a point of maturity as of yet unseen. We can go beyond what we've seen so far, just sticking to standards, making meaningful designs, and caring about the user.

Absolutely enlightening.

Founders at Work

This book was one of the most hyped ones in 2007. Everyone wrote about it, starting with Paul Graham, Guy Kawasaki or Joel Spolsky. And even me!

Frankly, the book is really worth every bit of the hype that surrounds it. I love computer history books (I already own a few of them) and this one is, together with "Dealers of Lightning" the one I liked the most. The stories of how Lotus, Apple, VisiCalc, Firefox, PayPal or the BlackBerry appeared and grew are simply fascinating.

The book might have had more impact and hype in the entrepreneurial world, but I prefer to see it as a landmark history book (too). I've started working in the 90s, during the dot-com boom, and saw many similar patterns as those described in the book; incredible market value evaluations, products strongly marketed but born dead, and incredible stories of successes that nobody would have thought to be possible. The personal computer revolution of the 70s and the 80s has many similarities with what happened in the web revolution, and also with what happens now during the Web 2.0 hype. These are tremendous waves, that redefine the entire industry. And I think, we're doomed to relive these again and again.

The New Old Thing

I've already written about this book, albeit in French :) So I'll translate what I've said so far there:

I've just finished reading The Old New Thing. The author, Raymond Chen, worked in the Windows development team since 1995 (at least) and explains the reasons behind some decisions taken during the design of different versions of Windows, since 1985 to Vista. This book is a compilation of some of the best articles in his blog.

And frankly, it's hard to believe.

Windows Vista still has APIs used to run DOS 1.0 applications, just for the pleasure of "ad infinitum" backwards compatibility. The names of the Win32 methods are

completely cryptic, impossible to remember, but Chen justifies each and every one of these oddities by different historic reasons. The registry contains informations used to change the internal behavior of the memory manager, so that Lotus 1-2-3 version 2 for Windows (1990) could work flawlessly under Vista.

I ask myself how could M$ allow such a book to be published! It makes me wish to never, ever develop software for Windows ever again, in any programming language. I strongly recommend this book, particularly if you have technical knowledge about the Linux kernel! The descriptions of the internal workings of Windows are impressive, with a level of detail never seen before.

iWoz

It is hard to argue the fact that Steve Wozniak has invented the personal computer as we know it today. If you had any doubts (even after reading his interview in "Founders at Work"), this book will wipe them away completely.

It is written by Woz himself. Wait, did I say written? This is a told story, that almost becomes a legend at the end of the book. Woz is not modest about his feat; but he does not brag about it either. He talks about his parents, his marriages, his children, Steve Jobs, the Apple I and the Apple][, with sincerity, humor and ingenuity.

You do not need to be a fan of Apple to enjoy this book; you just need to use a computer, remember that your parents didn't, and ask yourself, how did all of this began?

Best Software Writing

This book holds the "I" numeral, but the second version has not yet been published at the time of this writing. This book is interesting in many ways; first of all, it is part of an overall tendency to write blog-based books. Joel Spolsky, Eric Sink, Raymond Chen and others are part of this trend; popular blog posts that become excellent books when put together. This book is a compilation of what Joel found the most interesting during 2004, published in agreement with the respective authors.

This book is also interesting for another reason: I consider 2004 to be a pivotal year in our industry. Subversion was released that year, as were Rails and Firefox and many

other popular packages. Not only that, but the whole Web 2.0 trend can be seen as a rising to the public eye in that precise moment. The book does not explicitly show these trends, but there is an overall feeling on all the best writing for that year, that something was going on. The dot-com boom was finally behind, and new things could happen again.

Eric Sink on the Business of Software

Finally, a complete hands-on resource, useful to those seeking to start a software business (yeah, like me :) The author is Eric Sink, founder of SourceGear, maker of Vault, a popular version control system for Windows marketed as a drop-in replacement for SourceSafe. He talks about all the aspects of running a software company: finance, technology choices, tradeoffs, human resources, everything.

Even if the book is primarily targeted to the US market (which makes some stuff useless in other parts of the world, particularly the legal stuff) I think it is worth a read, and again, Sink's a great writer and the book is clear and concise.

Best books of 2008

You might remember my beloved mantras: learning a new programming language and reading at least 6 relevant books every year. Following the 2007 edition, here's the list of the 8 books I have enjoyed most in 2008, ordered by a purely subjective and absolutely irrational decreasing preference. I strongly recommend all of them!

Winner: Geekonomics: The Real Cost of Insecure Software by David Rice

Runner-up: The No Asshole Rule: Building a Civilized Workplace and Surviving One That Isn't by Robert I. Sutton, PhD

And 6 more:

- Software Estimation: Demystifying the Black Art by Steve McConnell
- Modern C++ Design: Generic Programming and Design Patterns Applied by Andrei Alexandrescu
- It's Not How Good You Are, It's How Good You Want to Be by Paul Arden
- RESTful Web Services by Leonard Richardson and Sam Ruby
- JavaScript: The Good Parts by Douglas Crockford
- Programming Collective Intelligence: Building Smart Web 2.0 Applications by Toby Segaran

Geekonomics: The Real Cost of Insecure Software

My big winner for 2008. If you don't care about software quality (yet), or if you think that machines aren't already in charge or our world, this book is for you. This has

been one of the most hyped releases of 2008, and indeed, it can send a chill down your spine. Geekonomics is a lively catalog of software glitches and disasters, showing users as "crash test dummies", describing a whole industry built upon "end user license agreements", removing liabilities as no other industry has ever been capable of. Even the world of free and open source software is described with strong criticism.

Geekonomics sometimes feels like a desperate plea to apply Deming's Total Quality Management principles in the world of software.

The book is interesting in several fronts. David Rice pledges for the creation of standards of quality, as well as tightening the requirement for certification of practitioners. His views are based on the United States, describing the legal framework of "tort law", the economic foundations of "incentives" for industries, and using comparisons with other economic sectors, particularly with the automotive industry. The book is academic yet entertaining, frightening but instructive, but sometimes falling on the side of sensationalism, in my opinion. I could even say that some of Rice's proposals are downright impossible to achieve, given the particular characteristics of the software industry, and the situation of the legal system in other parts of the world.

In any case, even if I do not particularly agree with all of his views, David Rice is right to emphasize his point strongly, giving concrete proposals, and opening the debate: the book would not have had the same impact if it had been written mildly.

Geekonomics was enlightening: it made me think about the quality of my own work in a completely different way (and prompted me to talk about it at Lausanne's 2008 Barcamp). I think that taking a time of introspection, and reading your own code with different eyes is required to realize that we are, as software developers, responsible for much of what is going on in the world. And this is the major contribution of this book.

The No Asshole Rule: Building a Civilized Workplace and Surviving One That Isn't

Some of my former colleagues will chuckle when they see this entry, because I've flown from a previous job a couple of years ago because of a total, complete, utter, outright and unmitigated asshole. A complete control freak, self-proclaimed genius, irresponsible jackass, who was the reason why 5 people (including me) left the place in less than 3 months.

(Guys, feel free to leave comments below ;)

More seriously, in the software industry it's common to see that great developers are, more often than not, shy or introverted. This fact, coupled with the Swiss' legendary attitude of eternal conciliation and conflict avoidance, has the side effect of creating troubled workplaces all over the country. No wonder this book made the headlines last month in Switzerland. This is a huge problem which is only being revealed right now.

In any case, this book is, together with Peopleware, a gem of teambuilding and a real bag of tricks to avoid turnover and burnouts in your company. An absolute read to everyone involved in the creation of workplaces, in every possible industry.

Software Estimation: Demystifying the Black Art

This book should be a mandatory read for every software developer, in every company, all over the world. It is a true gem, and a book that will become a timeless classic.

How many times have you been asked to estimate a project? To provide a deadline? To approve an estimation done by someone else? This book starts by enumerating the problems related to estimation, including those related to the definition of the word "estimation", the politics and the economics surrounding and defining what is accepted and required, and the common traps where all of us have stumbled upon at least once. Then it provides a catalog of common estimation methods, from the most obvious to the most complex ones, including a description of their relative drawbacks and benefits.

McConnell is the author of Code Complete (which I'm re-reading these days) and Rapid Development (which I plan to read soon). This guy knows what he's talking about, and he provides all the data required to support his claims. I cannot stress this more: if you are a project manager, a developer, a tester, an architect, or deal with software projects in some uncanny way, you have to read this book.

Modern C++ Design: Generic Programming and Design Patterns Applied

I bought this book during the writing of my Master's degree dissertation project. I had heard about it before, but since my project involved a lot of C++ template metaprogramming, I thought that the best idea was to check with the real experts. And boy, I'm happy I did.

This book goes beyond your common programming grounds, whichever your favorite language is. I personally enjoy writing C++ a lot, but that's me, and your mileage may vary. Doing multiple inheritance mixed with template metaprogramming, however, stretches your mind into the realm of what you previously considered esoteric and impossible, and that's precisely the kind of readings that take you a long way forward.

If you are looking for a new way to write your old C++ code, or if you are asking yourself how different are C++ templates from Java or C# generics, read this book. You'll find a lot of interesting stuff in it, with explanations related to an existing library (written by the author) called Loki, which provides the implementation of several patterns explained in the book.

It's Not How Good You Are, It's How Good You Want to Be

This little book (less than 130 pages long) was written by Paul Arden, former creative director of Saatchi & Saatchi, a recognized advertising agency (whose "early growth was also helped by a policy of settling the invoices from small suppliers as late as possible, while promptly paying large, high-profile companies", dixit their Wikipedia article. No comments.)

Paul Arden provides extremely interesting and pragmatic views on creativity, people management, client relationship management and other stuff; if you happen to work in a web or advertising agency, you should read this book. However, as a software developer, I think that many of his views are still applicable to our own activity, particularly when, like me, you're beginning your own independent path. It is not always obvious how to deal with situations when you're "in charge", and Arden's recommendations do help.

RESTful Web Services

For years I thought that SOAP was the way to go. And then some Rails activists started talking about RESTful this, RESTful that, and well, it tickled my curiosity. It all started with Roy Fielding's doctoral dissertation about network architectures, followed by all the buzz of Web 2.0 APIs, which in the case of most startups took the form of RESTful APIs, found to be much easier to document, use and maintain than their XML-RPC or SOAP counterparts.

I read this book because of a requirement of a project where I worked at the beginning of the year, and this led to several blog posts and even a project, that are still today quite popular. I've been using the RESTful approach for other projects, involving .NET, the iPhone and Cocoa on the Mac, and I would have never thought that doing network-based programming could be this fun. Looking backwards, attempts like SOAP and XML-RPC look clunky, adding layers of unneeded complexity for most projects, and creating a higher barrier of entry for new users of an API.

JavaScript: The Good Parts

Douglas Crockford is head of web development at Yahoo!, and probably the person in the world that knows most about JavaScript. I've been reading his articles since I started working in my Propano project, and then watching his videos as soon as they became available.

His views of JavaScript as the World's Most Misunderstood Programming Language helped me see this (now I think) beautiful language under a new light. And of course, I could not miss reading his short (170 pages) but extremely detailed book. I thoroughly enjoyed it and strongly recommend its reading to anyone dealing with JavaScript in any way.

Programming Collective Intelligence

I love programming books focusing on solutions rather than on specific features of some language; they provide insight on how to solve problems, showing the mathematical background required to do it. This is one of them.

This book can be considered a practical handbook on statistics programming, using Python for the code examples, but providing all the required insight and theory required to understand the logic beneath every code bit. The focus on Web 2.0 applications is clear, and this book has helped more developers than the author might ever know. Social sites are the realm of networks, large numbers, big datasets and complex relationships, and the techniques described in this book can help developers get out more information about their databases. Ever wondered how LinkedIn finds out someone you might know? This book has the answer for you.

Apple

Pastrami Sandwich

http://kosmaczewski.net/2008/06/10/pastrami-sandwich/

I find many similarities between an event like WWDC and a similar one I've attended at Redmond long ago; both are big (huge!) events, with thousands of (men) engineers from around the world (and very few women), with a keynote by the founder, lots of events every morning and afternoon, and merchandasing stuff all over the way. And of course, in both cases you get food boxes for lunch.

However, there is one basic difference between both events. Apple not only has interesting technologies to show up, even bleeding edge ones, more often than not on the open and public domain (many of which I can not write about, and boy they are going to make a difference!), but even better than that, it has a vision.

And passion. Cocoa developers are among the most passionate I've ever met, and you just can't find that in a Microsoft event. You can feel that in the (conditioned) air of the Moscone center, almost touch it. New projects everywhere. People discussing about their ideas. Lots of collaboration, openness and willingness to go further. Microsoft's stuff is, well, boring at best; dull and gray. Enterprise IT is no fun, believe me, but there's no reason to try to look at it in a different way. And faithful to its own way, Apple is precisely doing that, right now; and what's about to come will reshape the industry forever.

As Steve Jobs announced yesterday, the iPhone is now ready for the enterprise, given that the majority of companies here and there use Exchange as the basic means of communication. Now the iPhone is ready to access all that information, and I while I can't tell you more, this is the just the first step. iPhones will be an important player in the enterprise landscape in the future months.

On the other hand, well, Microsoft is asleep. The sales of Macs are booming, and more and more shops are shifting to it. The iPhone is sold out, the WWDC is sold out, a

new wave of iPhones is about to hit the street, and there's much, much more in store for the future.

Personally, I think that the center of gravity of the consumer software industry has shifted (pay attention to the tense of the verb I used). There is one and only clear leader, and that's Apple. Microsoft should concentrate its effors in what it does best (I think), which is enterprise products, and stop delivering crappy operating systems every 5 years. Any advantage they had 10 years ago has just vanished, and that's mainly Microsoft's own fault. Ballmer should have resigned long ago. But again, as Joel said, they have enough cash to continue screwing things up for a long time.

In the meantime, I'm finishing my pastrami sandwich, sitting right beside the Mission conference room, waiting for the next session. I'm actually here in San Francisco and I can barely believe it.

The Dirty Little Secret of iPhone Development

http://kosmaczewski.net/2008/12/23/dirty-little-secret/

This is happening right now, at a web agency near you.

The dot-com boom of the 90's spawned a brand new generation of coders and software developers, including me, by the way. While before that time the term of "software developer" might have been reserved to system programmers fluent in C, COBOL, C++ or other languages, right now the vast majority of developers I know spend their time writing web applications, either public or in a private intranet, in J2EE, ASP.NET, Rails, PHP, you name it.

I have said before that writing web applications should be taken as seriously as writing desktop systems. Call me names if you want, but I'm a big fan of Joel's Test.

However, after all this years, after the Chaos reports, after Peopleware, after the Mythical Man Month, people still treat quality as an afterthought. And also complain about how much software sucks, how expensive it is, and how late it arrives, by the way. Now that the iPhone SDK is widely available, that the App Store is selling more apps that we could have had imagined 6 months ago, many web agencies want to jump to native iPhone development contracts, which are hype and nice and pricey and whatnot. Which is only going to make things worse.

The dirty little secret in this story is this: iPhone development looks more like developing applications for a desktop operating system, and less, much less than web development. And I'm frightened to see some small shops (and even bigger ones), who never attained a real level of professionalism or quality in their software tasks, starting projects and realizing, later, when they are over budget and behind schedule, that

this kind of applications requires a different mindset.

Let's repeat something that you should know by now: web apps are easy to maintain. To begin with, you just have one instance running of it, and as Paul Graham said, you can write them in any language you want, you can release bug fixes with your application running, monitor performance issues live, change its appearance quite easily and for everyone at once, etc. This is probably the single biggest advantage of web apps. With AJAX we even got the possibility of going beyond in terms of user interface, responsiveness, perceived speed, you name it, and today the number of famous web apps is outstanding.

However, web apps run into this terrific (and terrifying) sandbox called the browser. They (normally) cannot access your file system, they cannot open other applications and interact with them - well, with some custom URL schemes like lastfm: mailto: or skype: you might have the ability to do some things, but certainly not much more than activating the application in some limited way as a help for the user. You cannot access the user's hardware directly - well, again, you can access the webcam through Flash, or you can trigger the window.print() method to have the native print dialog pop up, but that's more or less the maximum you can do.

In the iPhone, there is a similar situation. You choose to create a native iPhone application over a web one when you require one or many of these things in your application:

GPS geographical data; Accelerometer information; Photo camera or library; 3D graphics; Complex animations; Address Book entries; Sound recording and playback; etc. . . ! I've talked about the dichotomy between iPhone native vs. web apps in my speech during the iPhone conference where I used this graphic, which might help those having to decide whether they should do a native or a web application:

The tradeoff, basically, consists in knowing that having the ability to access these features, means that you are giving up your capacity to roll out quick upgrades to your code. You have to depend on Apple's own review process timings for that, which, as far as I've seen so far, cannot be predicted. And finally, you depend on the user's final will to update your application!

Similarly, you also lose the ability to create these applications using your common, well-known, garbage-collected programming language, but you'll have to deal with Objective-C exclusively, together with its weird syntax, possible memory leaks, eventual out-of-memory notifications, lazy loading techniques, compiler warnings and er-

rors, and seeing the MVC design pattern even in your wildest dreams at night. Regarding the syntax, I admit that I love it, but it took me a while to get used to it, when I started playing with it back in 2003.

Given these conditions, when you start an iPhone project you will have (let me be very clear about this, so I'll repeat) you will have to dust out your good old desktop application programming techniques (or learn if you don't have them), read Code Complete again (which I'm doing right now) and take all the advice that you can from C and C++ programmers who have been working in this kind of stuff for the past 20 years.

I've even done my Master's degree project in C++ because of this: the mindset required for iPhone apps is not the same as for your day-to-day web application, and is closer to that of a desktop application. I've been doing web apps for 12 years now, and desktop apps for 5 years (mostly in C#, Objective-C and C++); that's my ground for stating this. You require a spec, you require tests, you require testers, you require daily builds, you require bug databases, you require quiet working conditions.

You require at least an 11 in Joel's Test to know that you're doing fine, but not only that: you need to know about pointers, you need to know C, you need to know that your code runs in an fragile environment with EXC_BAD ACCESS (SIGBUS) and low-memory warnings and stuff like that happening when you expect it the least.

Unfortunately, from what I've seen so far (at least here in the Lake Geneva region) many companies are spending much more than they intended to for rolling out their iPhone apps; these are real quotes from people I've dealt with in the past 6 months:

"Oh, we do not write specs, we prefer to modify the code as we have ideas!"; "Oh, we do not write tests, we do not have the time for that!"; (in general) "Oh, we do not work like that here. We do what the client asks and that's all."; "We will not follow Apple's iPhone Human Design Guidelines at all; we have our own. Users must double-click on buttons, so they feel more comfortable while using our application"; "We want our [PUT YOUR FAVORITE UIKIT CLASS NAME HERE] to have [SOME IMPOSSIBLE FEATURE WHICH DEFIES GRAVITY]. Easy, right?" "We have to remove this XYZ feature, right now!", which means rewriting half of the code and leads to this: "What? So much? For such a small change?" "Why have you spent all that time 'fixing memory leaks'? I won't pay for that. Please concentrate in the new features we've asked. By the way, what are memory leaks?" "Here's the styles document you asked, with the colors and fonts for the application" and the guy provides me with... a CSS stylesheet. Which references a font not available on the iPhone, by the way, and with colors in hex format. "We want an animation at startup, like the Chanel application" (everyone wants to release

the next Chanel iPhone app these days) “here’s an [animated GIF / Flash movie / PowerPoint slides] with the animation for you.” “Our team has slightly modified the code when you weren’t there, but it does not work any more, could you fix it please?” And of course, the all-time winners: the support tickets stating that “everything is slow” without further indication, or that “the application hangs”, without any details in it.

OK, I confess, I’m a bit of an extremist here; many of the quotes above are valid in some contexts. But not those of the small-to-medium sized iPhone projects I’ve been involved in lately, with the urgency of releasing the code as fast as possible, just for the sake of being there, in the App Store, right now.

There’s a long road ahead. The problem, again, is not the technology itself, but the people involved in these projects (me, for example :). There’s a bit of what Joel mentioned a while ago, about the perils of Java schools, which actually only has to do with developers, but there’s also the issue about teaching the clients the limits of the platform, and about creating a strong software engineering body of knowledge in companies which usually did not need it previously.

We’ve been doing web apps for so long that we’ve forgotten how to sit down, take a deep breath, fix that goddamn memory leak, and realize that, as Brooks and Joel said, good software takes time.

10 iPhone Memory Management Tips

http://kosmaczewski.net/2009/01/28/10-iphone-memory-management-tips/

Memory management in the iPhone is a hot topic. And since tonight I'm talking about it on tonight's monthly meetup of the French-speaking Swiss iPhone Developers group, I might as well share some tips here from my own experience.

I won't go dive through the basics; I think that Scott Stevenson did a great job in his "Learn Objective-C" tutorial at CocoaDevCentral, from where the image below comes. I'm just going to highlight some iPhone-specific issues here and there, and provide some hints on how to solve them.

To begin with, some important background information:

The iPhone 3G has 128 MB of RAM, but at least half of it might be used by the OS; this might leave as little as 40 MB to your application... but remember: you will get memory warnings even if you only use 3 MB; The iPhone does not use garbage collection, even if it uses Objective-C 2.0 (which can use garbage collection on Leopard, nevertheless); The basic memory management rule is: for every [alloc | retain | copy] you have to have a [release] somewhere; The Objective-C runtime does not allow objects to be instantiated on the stack, but only on the heap; this means that you don't have "automatic objects", nor things like auto_ptr objects to help you manage memory; You can use autorelease objects; but watch out! Since they are not released until their pool is released, they can become de facto memory leaks for you...; The iPhone does not have a swap file, so forget about virtual memory. When there is no more memory, there is no more memory. Having said this, here's my list of tips:

Respond to Memory Warnings Avoid Using Autoreleased Objects Use Lazy Loading

and Reuse Avoid UIImage's imageNamed: Build Custom Table Cells and Reuse Them Properly Override Setters Properly Beware of Delegation Use Instruments Use a Static Analysis Tool Use NSZombieEnabled Respond to Memory Warnings

Whatever you do in your code, please do not forget to respond to memory warnings! I can't stress this much. I have seen application crashes just because the handler methods were not present on the controllers, which means that, even if you do not have anything to clear in your controller, at least do this:

view plainprint? ·····10·····20·····30·····40·····50·····60·····70·····80·····90·····100····· - (void)didReceiveMemoryWarning { [super didReceiveMemoryWarning]; } And you might as well respond to them on your application delegate, as follows:

view plainprint? ·····10·····20·····30·····40·····50·····60·····70·····80·····90·····100····· - (void)applicationDidReceiveMemoryWarning:(UIApplication *)application { [[ImageCache sharedImageCache] removeAllImagesInMemory]; } For the description of the ImageCache class, continue reading ;)

Or finally, as an NSNotification:

NSNotificationCenter *center = [NSNotificationCenter defaultCenter]; [center addObserver:self selector:@selector(whatever:) name:UIApplicationDidReceiveMemoryWarningNotification object:nil]; Avoid Using Autoreleased Objects

Autoreleasing objects is easy and useful, but on the iPhone you should be careful with it. By default there is an NSAutoreleasePool instance created for you at the beginning of the main() function, but this pool is not cleared up until your application quits! This means that during runtime, your autoreleased objects are de facto memory leaks, since they are retained until the application quits. (please see the comments below; I have experienced better performance when avoiding autoreleased objects, but my understanding of pools is misleading :)

I started getting a better performance from my iPhone apps when I stopped using some methods creating autoreleased objects, for example:

view plainprint? ·····10·····20·····30·····40·····50·····60·····70·····80·····90·····100····· // Instead of NSString *string = [NSString stringWithFormat:@"value = %d", intVariable]; // use NSString *string = [[NSString alloc] initWithFormat:@"value = %d", intVariable]; ... [string release]; In version 2.0 of the iPhone OS there was also the problem that some "convenience methods" did not work at all; I'm sure you've experienced your application crashing when using NSDictionary's dictionaryWithObjects:forKeys:

and then finding out that a replacing that with initWithObjects:forKeys: made your application run just fine. The NDA did not help at the time!

This does not mean that you can't use autoreleased objects; you should use them, for example, when you have factory methods returning objects not owned neither by the factory nor by the client calling it. You can also use autorelease pools in loops, when you need to allocate lots of small objects, but remember to release the pool right afterwards:

view plainprint? ·····10·····20·····30·····40·····50·····60·····70·····80·····90·····100·····110···· NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init]; for (id item in array) { id anotherItem = [item createSomeAutoreleasedObject]; [anotherItem doSomethingWithIt]; } [pool release]; Remember to always release the pool in the same context where it was created. CocoaDev has an interesting discussion about using NSAutoreleasePools in loops.

Oh, and please, never release an autoreleased object on the iPhone: your application will crash almost instantly.

Use Lazy Loading and Reuse

If your application consists of several different controllers embedded into each other, defer their instantiation until the last possible moment; this means in practical terms that your init method is minimalistic, and that you do more stuff when you need it; the example below is a typical list + detail layout, using a UITableViewController subclass inside a UINavigationController:

view plainprint? ·····10·····20·····30·····40·····50·····60·····70·····80·····90·····100·····110···· @interface UITableViewControllerSubclass { @private NSMutableArray *items; DetailController *detailController; UINavigationController *navigationController; } @end view plainprint? ·····10·····20·····30·····40·····50·····60·····70·····80·····90·····100·····110·····120 @implementation UITableViewControllerSubclass #pragma mark - #pragma mark Constructors and destructors - (id)init { if (self = [self initWithStyle:UITableViewStylePlain]) { // only basic stuff items = [[NSMutableArray] alloc] initWithCapacity:20]; navigationController = [[UINavigationController alloc] initWithRootViewController:self]; } return self; } - (void)dealloc { [items release]; [detailController release]; [navigationController release]; [super dealloc]; } // ... #pragma mark - #pragma mark UITableViewDelegate methods - (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath *)indexPath { Item *item = [items objectAtIndex:indexPath.row]; if (detailController == nil) { detailController = [[DetailController alloc] init]; } detail-

Controller.item = item; [self.navigationController pushViewController:detailController animated:YES]; } // ... @end As you can see in the example above, not only we are creating a DetailController instance only when needed (that is, when the user taps on an item in the UITableView), but we're reusing it every time the user taps on another cell of the table; this has another benefit: a reduction of object allocation and instantiation, which also helps increasing performance a little bit.

You could also use UIViewController's viewWillAppear: and viewDidDisappear: methods to perform some kind of lazy loading initialization and release, and if you really need to go further, you could use the UITabBarControllerDelegate's tabBarController:didSelectViewController: method to load and unload parts of your application from memory, as you need it.

Avoid UIImage's imageNamed:

Alex Curylo has written an absolutely great article about the problems with UIImage's imageNamed: static method. It seems (and in my tests this appears to be true) that the iPhone OS (versions 2.0 and 2.1 at least) uses an internal cache for images loaded from disk using imageNamed:, and that in cases of low memory this cache is not cleared up completely (this seems to be corrected with version 2.2, though, but I cannot confirm).

Since I have projects that must run on version 2.0 of the iPhone OS, I have created a UIImage category with the following method:

view plainprint? ·····10·····20·····30·····40·····50·····60·····70·····80·····90·····100·····
@implementation UIImage (AKLoadingExtension) + (UIImage *)newImageFromResource:(NSString *)filename { NSString *imageFile = [[NSString alloc] initWithFormat:@"%@/%@", [[NSBundle mainBundle] resourcePath], filename]; UIImage *image = nil; image = [[UIImage alloc] initWithContentsOfFile:imageFile]; [imageFile release]; return image; } @end
The name of the method includes the word "new", to comply with Objective-C's naming guidelines, since the object we're returning to the caller is not autoreleased and has a retain count of 1. The caller is then owner of the UIImage and responsible to release it.

Once I have this UIImage instance, I place it in an image cache with this interface:

view plainprint? ·····10·····20·····30·····40·····50·····60·····70·····80·····90·····100·····
#import <Foundation/Foundation.h> @interface ImageCache : NSObject { @private NSMutableArray *keyArray; NSMutableDictionary *memoryCache; NSFileManager *fileManager; } + (ImageCache *)sharedImageCache; - (UIImage *)imageForKey:(NSString

*)key; - (BOOL)hasImageWithKey:(NSString *)key; - (void)storeImage:(UIImage *)image withKey:(NSString *)key; - (BOOL)imageExistsInMemory:(NSString *)key; - (BOOL)imageExistsInDisk:(NSStri *)key; - (NSUInteger)countImagesInMemory; - (NSUInteger)countImagesInDisk; - (void)removeImageWithKey: *)key; - (void)removeAllImages; - (void)removeAllImagesInMemory; - (void)removeOldImages; @end Basically, ImageCache can be configured to have a fixed size in memory, and the images that were added first are removed first. It loads the images from the disk as required, keeping a copy in memory, and as suggested by Alex, you can remove them from memory in case of a warning:

view plainprint? ·····10·····20·····30·····40·····50·····60·····70·····80·····90·····100·····110···· - (void)applicationDidReceiveMemoryWarning:(UIApplication *)application { [[ImageCache sharedImageCache] removeAllImagesInMemory]; } The complete source code of this ImageCache class, together with some unit tests (thanks to the Google Toolkit for Mac), is available on the Projects section of this blog for you to download and play with.

Build Custom Table Cells and Reuse Them Properly

Remember to always use static NSString identifiers for your cells, which helps the UITableView class to reuse them and reduce memory consumption:

view plainprint? ·····10·····20·····30·····40·····50·····60·····70·····80·····90·····100·····110···· - (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath { Item *item = [items objectAtIndex:indexPath.row]; static NSString *identifier = @"ItemCell"; ItemCell *cell = (ItemCell *)[tableView dequeueReusableCellWithIdentifier:identifier]; if (cell == nil) { cell = [[[ItemCell alloc] initWithIdentifier:identifier] autorelease]; } cell.item = item; return cell; } I also avoid using NIBs when working with table cells, for performance reasons. I prefer to draw the cells through my own subclasses of UITableViewCell, themselves using overridden setters for their properties, which takes me to the next point.

Override Setters Properly

As I said above, I tend to create my own subclasses of UITableViewCell, providing a simple property through which I change the model class holding the data that the cell is supposed to show. This has the effect of changing all the values of the fields and labels to the values corresponding to those of the model instance.

To do that, I override the setters as follows; for the following class definition. . .

view plainprint? ·····10·····20·····30·····40·····50·····60·····70·····80·····90·····100·····110···· @interface SomeClass { @private NSArray *items; NSString *name; id<SomeProtocol>

delegate; } @property (nonatomic, retain) NSArray *items; @property (nonatomic, copy) NSString *name; @property (nonatomic, assign) id<SomeProtocol> delegate; @end . . . I use the following implementation:

view plainprint? ·····10·····20·····30·····40·····50·····60·····70·····80·····90·····100····· @implementation SomeClass @synthesize items; @synthesize name; @synthesize delegate; - (void)dealloc { [items release]; [name release]; delegate = nil; } #pragma mark - #pragma mark Overridden setters - (void)setItems:(NSArray *)obj { if (obj == items) { return; // thanks Marco! (see comment #3 below) } [items release]; items = nil; items = [obj retain]; if (items != nil) { // create the internal structure of the cell // if not present, and change the widget values } } - (void)setName:(NSString *)obj { [name release]; name = nil; name = [obj copy]; // I always copy NSStrings! if (name != nil) { // create the internal structure of the cell // if not present, and change the widget values } } - (void)setDelegate:(id<SomeProtocol>)obj { // do not retain! This is an "assign" property delegate = obj; if (delegate != nil) { // create the internal structure of the cell // if not present, and change the widget values } } @end Overriding setters properly is important because you might be introducing memory leaks if done wrong. I usually copy all my NSString properties too, as a rule of thumb.

Beware of Delegation

If your code is delegate of some other object which you are about to release, remember to set its delegate property to nil before releasing it; otherwise, the object might “think” that its delegate is still there, and will send a message to an invalid pointer. To see what I’m talking about, consider this code:

view plainprint? ·····10·····20·····30·····40·····50·····60·····70·····80·····90·····100····· @interface SomeClass <WidgetDelegate> { @private Widget *widget; } @end view plainprint? ·····10·····20·····30·····40·····50·····60·····70·····80·····90·····100·····110····· @implementation SomeClass - (id)init { if (id = [super init]) { widget = [[Widget alloc] init]; widget.delegate = self; } return self; } - (void)dealloc { // widget might be retained by someone else! widget.delegate = nil; [widget release]; [super dealloc]; } #pragma mark - #pragma mark WidgetDelegate methods - (void)widget:(Widget *)obj callsItsDelegate:(BOOL)value { // and here something happens... } @end SomeClass is delegate of Widget. Widget instances might be retained by someone else, which means that even after the release message in the dealloc method, widget might still be alive and call its delegate; if this variable is not nil, widget will send a message to a non-existent object, which will surely crash your application.

Use Instruments

The “Leaks” instrument is your friend, and you should use it after you write the first line of code. Typically, I launch it every time before doing a checkin of some new code. In Xcode, select “Run / Start with Performance Tool / Leaks” and you’re done. You can use it in the simulator or on your device.

Use a Static Analysis Tool

Use the LLVM/Clang Static Analyzer tool. This amazing tool will catch naming errors (regarding the Objective-C naming conventions) and some hidden memory leaks, which are particularly nasty when using CoreFoundation libraries (Address Book, sound, CoreGraphics, etc). You can add it to your daily build script, it’s very easy to use.

But you must use it. Enough said.

Use NSZombieEnabled

Lou Franco has posted an excellent article about how to use NSZombieEnabled in your development cycle. The idea is to be able to find which messages are being sent to invalid pointers, referencing objects which have been released somewhere in your code. Always remember who’s the owner of your objects, and check for existence elsewhere!

And You?

How about you? What are your tips or best practices you usually use for your iPhone apps? Feel free to share them in the form below.

Update, 2009-01-29: I am overwhelmed with the response and traffic that this post has gotten so far! Yesterday evening I had the pleasure of discussing this subject with the guys of the iPhone Developers Facebook group, and I got interesting remarks from Marco Scheurer from Sen:te (including a comment below), which I’ve added to this post today.

Oh, and by the way, I’ve uploaded the slides here! They have a Creative Commons license, so feel free to use them if you find them useful.

iPhone SDK: Une Nouvelle Ere Démarre

Il y a de moments clés dans l'histoire de la technologie. Hier soir, vers 18h (heure suisse), il s'est produit l'un de ces moments. Apple a dévoilé un SDK (Software Development Kit) pour l'iPhone, et le monde du développement logiciel mobile ne sera plus jamais le même. Voici pourquoi.

Après de multiples rumeurs, Apple a finalement devoilé hier soir un SDK (Software Development Kit) pour son téléphone fétiche, l'iPhone. Cette nouvelle, bien que apparemment sans intérêt pour l'utilisateur moyen, aura des effets sans précédents, autant pour chacun de nous, communs mortels utilisateurs de téléphonie mobile, comme pour l'industrie toute entière.

En effet, bien que ce n'est pas la première fois qu'un fabriquant de téléphones mobiles offre un tel produit, l'engouement autour de l'iPhone tout comme ses charactéristiques techniques font que la nouvelle prenne une toute autre ampleur.

Pour ceux qui ne le sauraient pas, un SDK est un ensemble d'outils, qui permet aux développeurs de logiciels de pouvoir créer des applications autour d'une plate-forme. Par exemple, J2EE ou Ruby on Rails peuvent être considérés comme des SDKs, spécifiquement conçus pour créer des sites web, qui stockent leurs informations dans un système de base de données structuré. De la même façon, chaque système d'exploitation comme Windows, Mac OS X, Linux, mais aussi Solaris, QNX, BSD et n'importe quel autre, est généralement fourni avec un SDK (usuellement gratuit) pour que les développeurs puissent augmenter les capacités de la plate-forme, en l'étendant dans des façons nouvelles et inconnues par son fabriquant.

Dans les cas des téléphones portables, aucun des SDK existants (la plupart basés sur

le langage Java) n'ont eu un succès fulgurant, bien que le hardware utilisé pour le monde mobile devient chaque jour plus puissant, et bien que l'expérience utilisateur de la plupart de téléphones laisse vraiment à désirer. Quiconque aura attendu 2 minutes pour que son jeu puisse être utilisé sur son portable (le temps pour que le logo "Java" disparaisse), ou quiconque aura vu son SMS disparaître lorsque le téléphone redémarre inopinément, saura de quoi je parle.

Voici donc Apple; une société dont l'iPhone est la deuxième intervention sérieuse dans le monde de l'informatique mobile, le premier essai, le Newton, s'étant soldé sur un échec commercial (mais un succès conceptuel, comme Palm l'a prouvé quelques années plus tard). Cette fois-ci, la société fondée par Steve Jobs en 1976 compte bien changer les règles du jeu, et le SDK annoncé hier soir est une partie fondamentale de cette stratégie.

D'un point de vue technique, on peut dire sans se tromper que l'iPhone et un Mac de poche. Le système d'exploitation de l'iPhone (ou "iPhone OS") est une version miniature du Mac OS X, le software qui gère le fonctionnement de n'importe quel Mac sur le marché. Mac OS X compte une panoplie complète de librairies et de fonctionnalités prêtes à l'emploi, la plupart d'entre elles développées et testées continuellement depuis la fin des années 80 (à l'époque de l'ordinateur NeXT). Tout cela est maintenant à disposition des développeurs dans l'iPhone OS.

Mais ce n'est pas seulement un compilateur qu'Apple fournit avec son SDK; c'est aussi une suite d'utilitaires qui permet de créer le code et de le corriger (Xcode), de créer graphiquement des interfaces utilisateurs avec le moindre effort (Interface Builder), de voir son exécution et de paramétriser ses performances (Instruments) et de livrer les applications aux utilisateurs (App Store).

Finalement, Apple s'est inspiré de Cocoa, la librairie et runtime graphique utilisé dans le Mac, en ajoutant les contrôles nécessaires pour créer des applications iPhone, qui gèrent correctement les actions de l'utilisateur, lorsqu'il promène ses doigts sur le "touch screen" de l'appareil. Cette nouvelle version de Cocoa, "Cocoa Touch" utilise tout le pouvoir d'Objective-C, le langage de programmation orienté objet, vraie "lingua franca" du développement sur Mac.

Objective-C est un langage unique en son genre: c'est probablement le seul langage de programmation dynamique et compilé à la fois; il offre toute la puissance et vitesse du langage C, avec la beauté et la grâce de la programmation objet, tout en fournissant un environnement qui se prête au "développement rapide" d'applications comme aux plus hautes performances.

Bref, l'iPhone OS ouvre la porte à une nouvelle génération d'applications mobiles: vitesse native, support pour "multithreading", rapidité de création, facilité de maintenance et de déploiement, et accès natif aux multiples capacités de l'iPhone (caméra, carnet d'adresses, navigateur web intégré, système de géolocalisation, et j'en passe). Je vous invite à voir la vidéo de la présentation d'hier pour voir les capacités de l'outil; avancez jusqu'a la minute 40, et regardez ce qu'on peut faire avec.

L'iPhone SDK est disponible gratuitement (il fait 2 GB!) chez http://developer.apple.com (il est juste nécéssaire de créer un compte ADC - Apple Developer Connection -, ce qui est gratuit et ne prend que quelques secondes). Dans la version disponible actuellement, seul l'Interface Builder fait défaut, mais les autres outils sont présents et prêts à l'emploi. La version définitive sera offerte dès le mois de juin prochain.

Entre temps, pour les développeurs qui voudraient apprendre Cocoa, je vous recommande trois livres:

"Programming in Objective-C" par Stephen Kochan (ISBN 978-0672325861) Learning Cocoa with Objective-C, Second Edition, par James Duncan Davidson (ISBN 978-0596003012) Cocoa Programming for Mac OS X, par Aaron Hillegass (ISBN 978-0321213143) Et quatre websites:

iPhone Dev Center Cocoa Dev Central CocoaDev Theocacao, by Scott Stevenson Je reste aussi à votre disposition pour toute question à propos de Cocoa, Objective-C et des technologies Mac, ayant utilisé Cocoa (avec un énorme plaisir!) depuis 2002.

Happy coding!

iPhone SDK 3.0: A New Beginning

Last year I blogged about the upcoming SDK 2.0 for the iPhone 3G, and boy did it change my life. For those who haven't followed closely everything that happened in this blog lately, there's been this (that's me in the WWDC keynote main room at the Moscone center) and then that (yours truly talking at the first ever European iPhone conference). All of this has been the result of going to San Francisco last June. That particular trip changed everything; I never thought that a simple plane ticket could generate this much.

The iPhone has literally changed my professional life. But it was only the beginning. Last Tuesday, Apple announced the iPhone SDK 3.0, and I'll expose here some thoughts about what's coming next.

To put it bluntly, I think that the iPhone SDK 3.0 is a jump to the realm of the desktop software platform. By that I mean that the next generation of iPhone applications will look more like small versions of more complex desktop-like systems, rather than mobile applications. Pasteboard, local Bluetooth networking, Undo support and Core Data are just some of the elements that will take the iPhone platform to the level of a small desktop platform (and no, I'm not breaking any NDA here, I'm just enumerating some of the "1000 new APIs" announced by Apple last week).

Much has been written about the App Store review process, about the lack of X, Y or Z features on the SDK, about the various problems and limitations of the platform; however, constraints are liberating. All of the limitations of the iPhone SDK have allowed lots of developers to come up with creative ideas to overcome them, and to create applications with a seriously distinctive taste to them; we wouldn't have had Ocarina or Shazam or the Facebook iPhone app otherwise.

Have there been a similar breakthrough in the Google Android platform? Let's be very clear about this: the answer is no. I've personally seen the Android SDK in action, and even if the Android platform seems promising, Apple's own SDK is years light ahead. As an advocate of open source solutions, I am sorry to say this, but Android is not a direct competitor of the iPhone SDK, at least not right now.

And not only that, but there's one important factor that's generally overseen: the strict approval factor for getting into the App Store has generated one of the most secure and stable software platforms ever delivered to the public. Have you heard about iPhone viruses? You haven't, right? Well, I've heard about Windows Mobile ones, if you ask me. And you can buy antivirus software for Blackberry and Android, by the way.

Heck, the first iPhone "virus" was the 1.1.3 firmware, it was released by Apple... and it screwed jailbroken iPhones. Big deal.

The platform might not be perfect, but hey, I can't think of working on any other right now.

And then, there's the programming language choice; as a geek and developer I can't think of a nicer choice than Objective-C's dynamicity and speed to deliver great software. Java-based platforms are limited by, well, Java itself. Want speed? Use straight C. Want object-orientation? Use Objective-C. Want compatibility and reuse? Mix C++ with your Objective-C code. Let's go back to the basics: the iPhone is delivering what no other mobile platform has dared doing before.

And, oh, by the way, don't rant about not having background processes (or any other feature, for that matter); I really don't think that the lack of support for them will hinder the development of this platform. Rather the opposite, we'll see a new generation of applications popping up in June, when the OS 3.0 will be released to the public. I'm more confortable with a platform with defined boundaries, rather than with a behemoth of unknown and undefined behaviour. I'll be glad to use the notification service.

The iPhone SDK might be limited, but it delivers what it promises. I have yet to see a platform from another vendor (or even from the open source realm) to be as coherent and as well defined from the very start.

The real power of the iPhone remains yet to be seen. This platform is beyond anything that we've seen so far, and all the elements for success are already present in the beta downloads.

To summarize: the iPhone is the next desktop platform.

In a more personal note, since last February I'm in charge of the software and business development of electronmobile, a new endeavour spinning off electronlibre, targeting businesses with the aim of providing a way to leverage the immense power of mobile applications in the fields of marketing, entertainment and social networking. Not only that, but we're going to deliver custom, brand new experiences to the iPhone SDK, with applications beyond anything you've seen before.

The iPhone is just a first step, but boy, what a jump. And Electronmobile will set the pace in the Swiss (and global) landscape to take things further. Our objective is to create a center of excellence in mobile software development, and the prospects so far leave me speechless.

Finally, what about Google Android, Symbian or Blackberry? Of course, they are interesting alternatives. The iPhone is by no means the right answer for everyone, but there's no doubt to me that Apple is setting the standard, here and now. Apple is leading the way, and iPhone shows what mobile smartphones should have been (and done) from the very beginning.

Please stay tuned for more goodies. You ain't see nothin' yet.

Other Stuff

Pourquoi pas?

http://kosmaczewski.net/2008/05/21/pourquoi-pas/

Pourquoi ne peut-on pas avoir des conférences comme celle-ci, avec des mini-events comme celui-ci en parallèle en Romandie? Ou encore comme celle-ci? Ou bien comme cette autre! Ou celle-là!

Pourquoi pas? Est-ce que le marché est trop petit? Y a-t-il un manque d'intérêt général? Je pense que le problème est important, et qu'une grande partie de l'élan technologique local est étouffé par ce manque d'une grande conférence technique chez nous.

Après tout, le web a été inventé à Meyrin.

Je constate avec amertume que la Suisse est à la traîne dans ce domaine. En Europe il faut se déplacer vers Londres (parfois Paris ou Barcelone, parfois Prague ou Copenhague) pour pouvoir assister à des événements techniques du type "world-class", avec des speakers qui font vraiment la différence à niveau mondial. Supposez pendant un instant que Reg Braithwaite, Adrian Holovaty, Leah Culver, David Heinemeier Hansson, John Resig, Tim O'Reilly, Christophe Porteneuve, Steve Yegge, Andy Clarke et Avi Bryant venaient à Palexpo pour une série de conférences et d'expositions: n'iriez-vous pas les écouter? (Vous ne connaissez pas tous ces noms? Je vous invite a suivre les liens et a voir ce qu'ils ont à dire et voir ce qu'ils font. Vous me comprendrez alors.)

Et pourtant! Des sociétés internationales, ce n'est pas ce qui manque. Des salles de conférence? Il y en a aussi, surtout à Genève. Chaque année les "TechDays" de Microsoft font le plein. J'ai vu Apple remplir une salle de conférence aux Pâquis. Des sociétés spécialisées dans le web (ou le développement logiciel en général)? Il y en a des centaines, avec des noms comme Google et Yahoo! présents dans les environs. Des écoles spécialisées? Il n'en manque pas.

Conclusion: le marché n'est pas trop petit, et il n'y a pas non plus un manque d'intérêt.

Alors?

Pour tout vous dire, je me rappelle de Telecom 95 avec une certaine nostalgie. (Du coup, je me souviens de me connecter, depuis les terminaux dans les couloirs de Telecom 95, via Telnet vers les ordis de l'Uni de Genève pour vérifier mon e-mail, ou faire un "talk" avec un ami qui était du côté du Quai Ansermet, et de me retourner et de voir derrière moi une bonne dizaine de personnes, me regardant comme si je faisais partie d'un film de science fiction. Il était 1995, donc.)

Au delà de ces souvenirs, Telecom était une vraie conférence technologique, de pointe, aux portes de la Suisse, avec les grandes et petites sociétés à portée de main. Mais les organisateurs ont senti que ce n'était pas ici que les choses intéressantes se passaient, laissant la Suisse sans une énorme vitrine dans le monde. Depuis, tout ce qui concerne le software, le web, le design, l'innovation, à l'air de se passer ailleurs.

Certes, il y a Lift, mais qui est plutôt centrée sur la connexion entre technologie et société. Comprenez-moi bien: Lift est formidable; mais je rêve d'une conférence sur la technique, sur les tendances, sur le web, où l'on explique "live" le pourquoi du succès de certaines sociétés, où les laboratoires montrent leurs nouveaux langages de programmation, où les blogs se remplissent de nouveautés, où Twitter explose 1000 fois par jour, où l'on apprend que certaines idées reçues à propos du métier d'informaticien sont carrément fausses, et qu'on peut faire mieux.

En voilà un rêve.

pequeños lugares donde todos se conocen

http://kosmaczewski.net/2006/04/02/pequenos-lugares-donde-todos-se-conocen/

steven paul jobs nacio el 24 de febrero de 1955 en san francisco, california, de un doctor en ciencias politicas sirio (abdulfattah “john” jandali) y de una maestra de escuela (joanne carole schieble) que, no pudiendo o no queriendo tenerlo en su regazo, lo entregaron en adopcion a paul y clara jobs, abogados de la ciudad de mountain view, en el condado de santa clara, california.

joanne decidio hacerlo, segun steve declarara años mas tarde, porque queria asegurarse de que su hijo se recibiria en la universidad.

los padres de steve se casaron mas tarde y dieron nacimiento a la hermana menor de steve; los dos hermanos se conocerian de adultos.

steve, vegetariano y budista, nunca termino la universidad; es mas, fue expulsado a pesar de sus altas notas.

hace exactamente 30 años, el 1ro de abril de 1976, steve jobs fundaba apple computer junto a sus amigos steve wozniak y ronald wayne.

ronald wayne vendio su parte de apple computer a los dos steve dos semanas mas tarde, por 800 dolares.

la computadora “apple I”, hecha a mano por wozniak y jobs en el garage de la casa de paul y clara jobs, costaba 666.66 dolares; unos 200 ejemplares fueron construidos.

en 1977 steve jobs tuvo una hija con chris brennan, pero el no reconocio la niña sino años mas tarde. su nombre es lisa brennan-jobs.

en diciembre de 1980 apple entra en bolsa; su valuacion bursatil es la mas grande desde que ford entro en bolsa en 1956; apple vale 1´700 millones de dolares; de sus 1000 empleados, 40 son millonarios ipso facto.

la primera computadora de apple con mouse, menues y ventanas no fue el macintosh, sino otra, mucho mas cara y menos exitosa, llamada “lisa”.

la hermana de steve es hoy dia una escritora de cierto renombre, que escribio un par de bestsellers; su nombre es mona simpson, aunque no se sabe si es su nombre verdadero o su apodo artistico.

en 1985, steve es echado de apple; con parte de la plata que tenia, le compra a george lucas una parte de lucasfilm; la empresa se llama pixar, su costo es de 10 millones de dolares.

en la serie “los simpsons”, mona simpson es el nombre de la madre de homer simpson. la hija de homer simpson se llama lisa y es vegetariana y budista.

uno de los primeros dibujantes de “los simpsons”, brad bird, trabaja actualmente en pixar, y fue el director de “the incredibles”, exito de pixar en 2005.

mona simpson publico en 1995 una novela, “a regular guy” (”un tipo normal”) donde describe a un tipo que no termino la universidad, que se hace millonario en silicon valley, lanzando una empresa en el zotano de sus padres, mientras que tiene que manejar su relacion con una hija no reconocida.

en diciembre de 1996 steve jobs vuelve a apple; en 2006, steve vende pixar a disney por 7´400 millones dolares.

la verdadera mona simpson, hermana de steve jobs, vive en santa monica, california, con su marido, richard appel.

lisa brennan-jobs es tambien escritora.

richard appel, el marido de mona simpson, hermana de steve jobs, trabaja desde 1989 como productor ejecutivo y escritor de guiones de series de dibujos animados.

entre ellas, richard appel fue productor y guionista de “los simpsons”.

Quotes

A small compilation of quotes I've put below the header of this blog during the past few years:

- No vemos las cosas como son. Las vemos como somos. (Hilario Ascasubi) - We don't see things as they are; we see them as we are.
- Don't be afraid to try something new. An amateur built the ark. Professionals built the Titanic. (unknown)
- Compatibility means deliberately repeating other people's mistakes. (David Wheeler)
- Cuando uno se compromete con lo que piensa, y encima piensa cosas que cuestionan lo incuestionable, es de esperar que haya alguna dificultad. (hernún) - When you stick to your ideas, and on top of that you think about questioning what's out of question, it's likely there'd be some problems.
- The definition of insanity is doing the same thing over and over and expecting different results. (Benjamin Franklin)
- Most people are fools, most authority is malignant, God does not exist, and everything is wrong. (Ted Nelson)
- sin incertidumbre no hay novedad, sin novedad posible no hay más que repetición y, por lo tanto, negación del otro como un ser libre: el ser libre es un ser incierto. (adrian mancuso) - without uncertainty there's no novelty, without novelty there's only repetition and, therefore, negation of the other as a free being: being free is being unpredictable.

Olé, olé, olé (Epilogue)

I just stumbled into this amazing TED talk by Elizabeth Gilbert via James Duncan Davidson (@duncan in Twitter) and I want to share it with you with some very personal thoughts below.

I've been fortunate enough to earn a pretty decent living doing basically what I consider a hobby for the past 13 years, which is typing code on a computer and see if it works. Which most of the time doesn't, but that's part of the game.

I strongly believe in what Elizabeth says in this talk, and I have believed in this for years. I deeply believe that we, software developers, software engineers, both self-taught and those coming out of college, are just creators, just as Elizabeth describes them. Simple creators, being able to provide new ways to information to be shown, to flow, to entertain, to move. Simple channels through which ideas are transformed into tools, behaviours, images and sound.

There's been a huge debate in this matter. Knuth named his masterpiece "The Art of Computer Programming", and the single choice of this title has sparkled a longlasting debate in the software community, one that this essay is unfortunately going to feed, too.

Interestingly enough, Knuth developed his own typesetting system for his book, TEX, which is named after the Greek word meaning "art or craft". His work not only had to be the most important book ever written on programming, but also, it had to be beautiful.

It had to be an object of art.

And I think that programming itself is art. And I think that programmers are artists. And this is maybe the single reason why so much has been written about programmer productivity, why software project management is so hard, why discussions around

programming languages distort into trolls and heated arguments, and why you feel this anger against this words on my blog and you call me names.

This is why writing opinionated software is key to success, that's why the best software companies take time into creating great software development environments that stand out, that's why Peopleware is so important, even 20 years after being published for the first time.

It is all about letting the flow of art come through the person whose hands are on the keyboard. It's all about letting this happen. It is not us who write, it is the writing that comes to us.

Software is art, and as such, it needs time, patience, iterations, silence, passion, coffee, naps, pizza, books, compilers, laughs, Nintendo Wiis and unit testing suites.

The creation of good software is embodied in the creative process itself. The best engineers I know suffer from this process as much as they enjoy it, using an iterative process of trial and error which, even after all these years, still applies. The best software developers release sometimes early, sometimes late, sometimes with quality, sometimes not, but they release. They refactor. They document. They teach others about all of this. They always think that they can do better.

They always think, as Elizabeth says in her talk, that their current work is the worst in the history of programming: "Not just bad, but the worst". They suffer about it. But they release, and they fight against the fear of being critisized because of their choice of programming languages, operating systems, tools, processes, insufficient testing or design patterns.

Real artists ship. The making of this industry is full of examples of why software is an art: the first Macintosh, Smalltalk, NeXTstep, the Internet, Erlang, Apache, Ruby on Rails, UNIX & C, Lisp; just glimpses of wisdom, brilliantly crafted, that struck as obvious yet incredible, and which prompt a huge crowd to cheer up and applause.

Anyway, I'm not as famous or well-known as Elizabeth or Duncan. I have not yet done anything such as Ant or Tomcat (originally written by Duncan, by the way), even if I release software and projects with an increasingly high rate lately, and with many projects in the pipeline these days. I hope that my best successes are still ahead of me. And I hope you'll enjoy them one day, too.

Do I think that a little "genie" is besides me? Yes I do. And given the extremely rational background of most engineers out there, stating such an argument will raise more chuckles than anything else. Heck, who cares.

If you ask me, there is something magic out there.

PS: there's this quote attributed to Jorge Luis Borges which says that "publishing is a way to stop editing"... and I thought about it just after publishing this article. I don't know if he really said that, but in any case I agree. The difference being that, in our case, we refer to publishing as "releasing". But the feeling is the same.