What Will the Software Architecture Discipline Look Like in 10 Years' Time

Adrian Kosmaczewski

2006-03-16

This is a tricky question; after all, 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)

Thus, trying to extrapolate our own craftsmanship up to 2016 is inherently tricky, but a nice thought experiment after all.

Software Architecture, today

First of all, I should say that I work primarily as Lead Software Developer and (lately) as Software Architect. My clients are businesses that need to automate some of their day-to-day tasks, to achieve better productivity and lower costs. This already narrows the type of applications I work in, to the good old three-tiered application, usually on top of a database. Lately this applications began to communicate a lot more between them, raising a (higher level) concept known as Service Oriented Architectures.

My objectives, as an architect, are the following:

- Creating strong, dynamic and knowledgeable teams
- Lowering development and maintenance costs
- Creating value by continuous innovation and research

I think that the role of the Software Architect will strengthen in the future; but it is my hope that we will become less astronauts (Joel Spolsky, 2001), that we will mix up with developers (junior and senior) and that we will keep a close eye into the real, final product that software is made of: source code. It is very

easy to fall into the trap of losing the context, and spend too much time talking about buzzwords and trends:

Now it's tagging and folksonomies and syndication, and we're all supposed to fall in line with the theory that cool new stuff like Google Maps, Wikipedia, and Del.icio.us are somehow bigger than the sum of their parts. The Long Tail! Attention Economy! Creative Commons! Peer production! Web 2.0! (Joel Spolsky, 2005)

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.

The Future

There are several aspects that, in my opinion, will be key to understand the future of my activity; I will go one by one, explaining them (or providing pointers for more information) and giving some ideas about the benefits they would bring.

1) Aspect-Oriented Programming (AOP)

Object-Oriented Programming OOP took 20 years to get mainstream (from Simula in the late 60s, to the creation of Smalltalk in the XEROX PARC in the mid 70s, to Java in 1995); I think that AOP (actually, AOP was also created by a XEROX PARC scientist) will have a similar, if no longer, incubation time; this is because of its relative complexity.

AOP aims to complement OOP orthogonally:

"Aspect orientation is a set of technologies aimed at providing better separation of crosscutting concerns." (Ivar Jacobson, 2004)

An archetypical example of crosscutting concern is logging; usually logging code is scattered throughout business application code, which ultimately makes maintenance harder; what if a bug is discovered in a third-party logging component, and hundreds or thousands of method calls must be changed? Other examples of crosscutting concerns are configuration, security, resource management or error handling. Using AOP, the business application code can be completely separated from crosscutting concerns; these are "mixed" ("weaved", using AOP terminology) during runtime following specific conditions, in specific execution points ("pointcuts", as they are called in AOP).

I think that AOP will be a key tool in the future, helping architects model better systems, helping developers concentrate in key business rules, and lowering the costs of maintenance. One big drawback of AOP nowadays is execution speed, since runtime weaving usually is a costly operation:

I tested three cases: the generation of the Trace.WriteLine call through AOP.NET, the Trace.WriteLine call manual coded before the call to the empty virtual function, and the Trace.WriteLine call without the virtual function call. The result was that using AOP.NET to cross-cut the logging was eight times slower than manually coding it. Again, these results are for cross-cutting a light-weight logging call onto an empty virtual method. For a fatter cross-cut onto a fatter method, the relative impact of the cross-cut will be much smaller. My test case is close to a worse-case scenario, so don't ditch AOP.NET or AOP in general based on this result. On the same token, don't be dismissive of the impact of doing compile-time-like things at runtime. The cost can be high. (Nick Wienholt, 2005)

2) Design Patterns

"Design Patterns" is a concept taken from the world of "real" architecture, used to describe certain conceptual elements that hold integrity and offer style to a building. In the case of software, the term Design Patterns is closely related to one of the most important software-related books written in the 90s: Design Patterns: Elements of Reusable Object-Oriented Software (Addison-Wesley Professional Computing Series, ISBN 0-201-63361-2) (http://www.amazon.com/gp/product/0201633612/103-2097252-2747802).
This book identifies almost 20 common OOP designs, useful for solving common problems found in software development. This book laid the path for other similar books, describing common networking, business or enterprise patterns.

I think that in the future, design patterns will be incorporated in programming languages (Ruby already defines Singleton, Observer and Iterator directly in its syntax - see RubyGarden in References), in visual development tools and in the overall toolset of architects and developers. In other words, the overall level of abstraction will raise, and design patterns will be used to describe these new levels.

3) Inversion of Control Frameworks

One of the most interesting design patterns is what Martin Fowler names "Inversion of Control":

Inversion of Control is a common phenomenon that you come across when extending frameworks. Indeed it's often seen as a defining characteristic of a framework. (…) One important characteristic of a framework is that the methods defined by the user to tailor the framework will often be called from within the framework itself, rather than from the user's application code. The framework often plays the role of the main program in coordinating and sequencing application activity. This inversion of control gives frameworks the power to serve as extensible skeletons. The methods supplied by the user tailor the generic algorithms defined in the framework for a particular application. (Martin Fowler, 2005)

This pattern or characteristic of is being widely exploited by a growing number of frameworks or "containers" that accelerate considerably the development of applications:

- Spring (http://www.springframework.org/ and http://www.springframework.net/)
- Hibernate and other persistence frameworks (http://www.hibernate.org)
- ASP.NET (http://www.asp.net)
- Struts (http://struts.apache.org/)
- Cocoa (http://developer.apple.com/cocoa/)
- Ruby on Rails (http://www.rubyonrails.com/)

All the packages named above provide not only pre-built functionality in their libraries, but also (and most important) overall application scaffolding, sequencing and instrumentation; they tightly define how an application will behave, they provide configuration options (so that the behavior can be changed later without modifying the source code) and also they provide standardization and portability, in the sense that usually the code that sits on top of these frameworks can be ported to other platforms in the future.

While these frameworks arguably reduce the creativity of the developers (and usually this is a complaint) or could impact the overall performance of the final application (something that is less of a problem every new release), the benefits that these packages bring greatly outcome the drawbacks, mostly regarding the shorter delivery times and the lower maintenance costs.

4) Dynamic and strongly-typed programming languages

This topic could likely be the one to raise more religious wars in the next 10 years; all developers have their preferred programming languages, and usually they tend to defend them against all odds, whenever possible, touting their advantages.

While the benefits of strong-typing in enterprise applications are clear (compile-time bug detection, ease of maintenance, clear syntax), the value dynamic languages can provide is yet to be seen, Java and C# (both statically typed) being the most widely used languages in these environments. Nevertheless, Ruby on Rails (http://www.rubyonrails.com) already provides an enterprise-class framework based in the Ruby language, and it is slowly gaining acceptance thanks to: speed of development, ease of maintenance, strong-typing and readability; the basic tradeoff between dynamic and static languages are the number of instructions needed to perform a task; the more dynamic the language, the less lines of code are needed:

Vastly reduced code footprint : We have read our Fred Brooks and respect the fact that there has never been a "Silver Bullet", and there is unlikely to ever be such. Rails is not a Silver Bullet. However, widely reported results place productivity increases over modern Java methodologies (e.g., J2EE , Struts, etc.) in the 6-fold to 10-fold range (with many of these claims coming from long-time Java luminaries). (Rick Bradley, 2005)

I think that dynamic languages like Ruby will be more accepted in the future for enterprise development. A big drawback against wide acceptance today is the lack of proper editors (Statically typed languages benefit from IDEs like Eclipse and Visual Studio.NET, which provide syntax help during development time; this is harder, if not impossible, to do with dynamic languages such as Ruby).

5) Model-Driven Architectures (MDA) and code-generation techniques

Code-generation techniques are mainstream today. Several different commercial and open-source packages allow to generate source code, usually from a design built in some kind of modeling tool; lately one of the methods that has gained the biggest momentum is MDA (http://www.omg.org/mda/), which stands for Model Driven Architecture. MDA uses standard UML to define and design application from different perspectives, and defines a basic workflow that ultimately is translated into source code.

MDA enables, for example, to create the skeleton of the same application in several different platforms (hardware or software), for example, Java, .NET and Cocoa, or the Web and the Desktop, simultaneously. This way, one can achieve knowledge reuse at architectural level, bringing more productivity and adding more value.

I think that MDA will grow in acceptance and support (it already has a lot of support), and will ultimately become a useful standard in 10 years' time.

Conclusion

Another factor that I could have added in the list above is Open Source Software (OSS), but I think that this is a huge one that ultimately surrounds all the others without distinction, and gives them more strength and value (both social and economically speaking). OSS is a subject in itself, that would have made this article longer that it already is!

It will be interesting to re-read this article in 2016 and see how wrong I am now; I think that in Computer Science, 10 years are worth 10'000 in archaeological times: animal species appear and disappear, ice ages come and go, volcanoes explode and deserts replace forests. The whole landscape changes continuously, but I strongly think that the core of our activity, the code, will remain the center of our attention. Only time will tell if in 10 years' time we will still use Emacs to type our code, or not.

References

Bruce A. Tate, "Beyond Java", ISBN 0-596-10094-9, O'Reilly, 2005

Ivar Jacobson, Pan-Wei Ng, "Aspect-Oriented Software Development with Use Cases", ISBN 0-321-26888-1, Addison-Wesley, 2005.

Joel Spolsky, April 21st, 2001 [Internet], "Don't Let Architecture Astronauts Scare You", http://www.joelonsoftware.com/articles/fog0000000018.html (Accessed January 27th, 2006)

Joel Spolsky, October 21st, 2005 [Internet], "Architecture Astronauts are Back", http://www.joelonsoftware.com/items/2005/10/21.html (Accessed January 27th, 2006)

Martin Fowler, June 26th 2004, "Inversion of Control" [Internet] http://martinfowler.com/bliki/InversionOfCon (Accessed January 27th, 2006)

Nick Wienholt, "AOP Performance Numbers for .NET" [Internet], http://ablog.apress.com/?p=454 (Accessed January 27th, 2006)

Rick Bradley, "Evaluation: moving fromm Java to Ruby on Rails for the Center-Net rewrite" [Internet], http://rewrite.rickbradley.com/pages/moving_to_rails/ (Accessed January 27th, 2006)

RubyGarden, "Example Design Patterns In Ruby" [Internet], http://www.rubygarden.org/ruby?ExampleDesig (Accessed January 27th, 2006)

Salon.com, 2000 [Internet], "Why Bill Gates still doesn't get the Net", http://archive.salon.com/21st/books/ (Accessed January 27th, 2006)