

Innovation in Programming Languages and Systems

By Adrian Marriott, an independent consultant at Logos Software

Modern Innovation

Contrary to popular belief, innovation per-se is not rare in the IT industry; it is ubiquitous and goes on almost unnoticed on many development projects. Many jobbing programmers, particularly those using general programming languages such as Java and C++ rather than configuration files or simple scripting languages, innovate every day. The motivation for this ‘ordinary innovation’ is commonly because there are no language features or off-the-self libraries that can meet their particular requirements. The best of this activity produces new optimal or near-optimal implementations of novel algorithms to meet unique requirements. In the worst cases, dreadful, un-maintainable code is produced, with poor runtime performance characteristics.

The innovation often discussed in the media is ‘celebrity innovation’ and is commonly misrepresented; either as ‘radical innovation’ that has happened in the past, so is proven technology and the innovator is hailed as a genius, but the actual process of innovation is surrounded by an obfuscating aura of mystery; or ‘cutting edge innovation’ which is very often corporate marketing ‘hype’ masquerading as the next radical innovation. This hype often attempts to misrepresent existing languages and techniques as ‘legacy’, to hyperbolize just how advanced it is.

“‘Legacy code’ is a term often used derogatorily to characterize code that is written in a language or style that (1) the speaker/writer consider outdated and/or (2) is competing with something sold/promoted by the speaker/writer. ‘Legacy code’ often differs from its suggested alternative by actually working and scaling.” Bjarne Stroustrup^[1]

There is no denying that radical innovation is a genuine phenomena and the point here concerns its source. Radical innovation may indeed be the inspiration of genius, but this genius does not work *en vacuo*. From the IT perspective this sort of innovation can only occur with respect to a given problem space and a given set of technologies that work within that space. Anyone, genius or otherwise, attempting to offer genuine new approaches to a given problem must be working within the problem space, and familiar with existing technologies. This context is very similar to that of the scientist presenting a radical new scientific theory^[2] as described by Karl Popper^[3], and exactly the position of the jobbing programmer. Radical innovation is only recognized in retrospect and arises when clever people attempt to solve complex problems using existing technology. Most, if not all, of the radical innovations within computing in the last 30 years have come from this sort of context.

¹ http://www.research.att.com/~bs/bs_faq.html#legacy

² The Logic of Scientific Discovery, by Karl Popper, Routledge, ISBN: 0-415-27844-9
http://www.amazon.co.uk/Logic-Scientific-Discovery-Routledge-Classics/dp/0415278449/ref=sr_1_1?ie=UTF8&s=books&qid=1201021216&sr=1-1

³ http://en.wikipedia.org/wiki/Karl_Popper

Historical Perspective

It is useful to take a historical perspective, and examine why programming languages changed in the past. Along with the question of what problems any new programming language can solve, there is the question of motivation. What are the problems with the existing programming languages that using the new language will solve?

C to C++

The C language was used to write the Unix operating system. Some of these libraries have remained essentially unchanged for over 30 years and are still in widespread use today. C was fast, available on many platforms and had full and easy access to the operating system facilities through standardized system libraries. What problems did programmers using the C language experience that were solved by moving to C++?

Classes and Objects

The reification of abstract data types (ADT) into classes and objects was a major innovation. The OO paradigm provides for a much richer design space for programmers and with the advent of C++ OO becomes widely adopted. ADT is possible in C, using structs, pointers, and careful function naming, but it is not directly supported by the C language, and is therefore harder to express.

OO provided a useful way of conceptualizing complexity; dividing complex problem spaces into sets of cooperating classes helped programmers visualize and understand these systems. The OO paradigm provides a useful way of thinking about complexity compared to purely functional or procedural approaches.

Encapsulation

Having OO supported directly by the language provided opportunities for improved degrees of encapsulation compared to C. Again, encapsulation at the level of the function, the file or the library is possible in C, but with the introduction of OO and the keywords *public*, *protected* and *private*, this is massively enriched in C++.

Now different components could be designed and implemented to fulfil particular programming 'contracts' with particular interfaces. This helped to reduce the coupling between parts of complex systems, which in turn helps to produce independently testable components.

By arranging components into hierarchies where the lower level components can be independently tested in isolation from the overall complexity of the whole system, these can be verified as 'sound' before higher level components that use them are tested. Improving testing procedures increases the reliability of the final system.

Initially with the move from C to C++ the overhead of virtual function calls were an issue, with the hardware at the time, which prompted the introduction of features like in-lining, but the development proceeded in the spirit of the C language where the aim was to keep programs as fast and as memory efficient as possible.

C++ to Java

So what problems did C++ programmers experience that are solved by the introduction of Java? These were well documented right from the earliest versions of Java.^[4]

Simplified Language and Syntax

The C++ syntax is complex, and this can slow down programmers, particularly novices. C/C++ syntax maps very closely to exactly what is happening at the assembler level. This level of control is useful for writing operating systems, but is largely unnecessary for many commercial applications. For example, the design of Java enables the syntax to dispense with the distinctions between the ‘.’ and ‘::’ and ‘→’ operators. Rarely used features such as multiple inheritance and operator overloading are dropped. This makes the language simpler to learn, and simpler to use in most contexts.

Memory Management

Common problems for C++ programs were dangling pointers, and leaked memory, where allocated memory was either mistakenly left, or more rarely, simply sacrificed because of complex shared custody issues. Java solved this by re-introducing the innovation of garbage collection from earlier languages such as Lisp.

Portability

The C language was touted as a portable way of writing assembler, but as anyone who has ported a C/C++ program to another platform will testify, porting is not a trivial issue. Java’s use of the JVM architecture massively simplifies the overhead and costs associated with supporting systems on different platforms. In many cases it is a NOP.

Java Library Facilities

C++ inherited the C language’s use of the C system libraries because of its link compatibility with C, so many of the problems solved in C could be directly reused, or solutions could be developed using the same familiar and tested library resources. However, much innovation in the C++ community is fulfilled by programmers using 3rd party libraries which might not be commonly available on every platform. There is a plethora of 3rd party C++ libraries, many of which do very similar things. This provides fantastic opportunities for genuine innovation, but it also builds in dependencies on these 3rd party libraries, and this dependency on particular libraries can be a problem in various situations.

- If you attempt to install your C++ program on new system where these 3rd party libraries are not present, your program will not operate correctly.
- New improved versions of the C++ library are released. If you depend on many 3rd party libraries your code-churn can be immense.
- With everyone using different C++ libraries to do similar things, systems are unnecessarily incompatible; there is more unnecessary detail for programmers to learn as they move from one project to another. There is a significant lead time to

⁴ See: <http://java.sun.com/docs/books/tutorial/getStarted/intro/definition.html> for a restatement of these features

get in-depth knowledge of how best to use any library, and to be able to accurately characterize runtime behaviour of programs based on this library.

- If everyone uses different libraries the rate at which users discover bugs in the libraries is reduced, because there are fewer users of each library than might otherwise be the case.

Java provides a good standard set of libraries to cover almost every common programming eventuality, and this can be relied on to be available wherever Java is installed, it is largely platform independent, and the update cycle is in-frequent and because there is a dependency on a single component, code churn is reduced.

Types of Innovation

This leaves open the problem of distinguishing genuine innovation, radical or otherwise, from hype. This is not a trivial problem nor is it academic. There is a crisis in the software industry over exactly this issue of 'technology assessment'; how is the worth of new technology best assessed? Corporations spend many millions of pounds each year attempting to understand various new technologies, and whether they represent genuine innovation over that which is currently established. Will it give them a competitive advantage, or is it essentially worthless hype? Full treatment of the software assessment question is beyond the scope of this paper but it is useful to distinguish two types of innovation here; *convergent* and *divergent* innovation.

Convergent Innovation

Convergent innovation is where the new component is provided in such a way that it integrates easily with existing technology and can therefore easily utilize the good parts thereof. Users of the new component have relatively little to learn and can still use their experience with the existing technologies to guide their development decisions. Convergent innovation does not 'throw the baby out with the bath water'.

We will use illustrate this discussion with reference to a relatively new computer technology called complex event processing (CEP)⁵. An example of convergent innovation is where someone implements a useful functional component, such as a novel way of matching events to a set of condition-action rules in CEP, packages this up as say a Java or C++ library and sells it. The direct users of this technology are technologists, i.e. programmers, and they are expected to have the prerequisite skill in Java or C++ in order to use the component. Integrating this 'innovative technology' into either existing systems, or to employ it in new systems, is relatively easy because it is supplied as a language library. In general language libraries have proved to be the best way of distributing arbitrary functionality because these libraries *must* be used in the context of a general programming language, and in the case of Java and C++ these are very mature languages, and consequently provide the simplest and most flexible way to combine components together.

Divergent Innovation

In contrast there is divergent innovation. Here the technology is packaged in such a way that it does not easily integrate with existing technologies, and requires a

⁵ See http://en.wikipedia.org/wiki/Complex_Event_Processing for more information

disproportionate intellectual investment from the direct users in order to use it effectively.

To reuse our CEP example above, this might be to bury this innovative event matching technology into a system which uses its own ‘innovative’ event processing language, when it could be simply supplied as a Java or C++ library. Hiding this useful functionality behind its own language then presents two issues: first the direct users have to learn an entirely new programming language, and second there will be a linkage problem, so integrating this functionality into programs written in existing languages becomes an issue.

Now in our example this new CEP language is by definition immature, because it is innovative technology, so as pragmatists we can expect many bugs and functional omissions; everything from basic language features, such as incorrect or non-existent handling of type promotion, to more serious drawbacks such as lack of access to extensive library components – peripheral technology – provided as standard in Java, or with 3rd party libraries such as STL and Boost with C++. Most important of all this event processing language is unlikely to provide the rich encapsulation features of a mature OO language such as Java and C++, features which have been proven to help control complexity, because writing a language parser with well defined and useful semantics, that produces an optimized runtime executable is time consuming and expensive^[6]. As the historical testament of programming languages such as C++^[7] and Java proves, developing a good general purpose programming language is not a trivial undertaking.

In addition to the problems of an immature new language, much of the hard-won Java and C++ experience of the direct user is obviated. All the experience whereby the programmer or systems architect has gained in-depth knowledge of designing, implementing and deploying systems based on these languages is thrown away. All the knowledge of how the C++ optimizer operates and interacts with various language features, all the details of how the Java garbage collector can compromise performance and the techniques to avoid this, all these detailed patterns, rules-of-thumb and idioms that are characteristic of any development/deployment environment and knowledge of which serves to guide the developer through all stages of the project to successful deployment are largely unusable, and similar knowledge will have to be learnt again in the context of the new language.

Learning this detailed knowledge is not easy, and essentially requires using the language in an attempt to solve problems and ‘learning from your mistakes’. It is widely believed that the only practical way to learn new technologies in the current IT environment is ‘on-the-job’, but this entails making a whole raft of new mistakes in this new language, all of which takes time, increases the effort required to meet project deadlines, and increases the risk of project failure.

⁶ This is true even when parser generators such as YACC or ANTLR are employed (see: <http://en.wikipedia.org/wiki/Yacc> and <http://en.wikipedia.org/wiki/ANTLR> respectively)

⁷ The Design and Evolution of C++, by Bjarne Stroustrup, Addison-Wesley, ISBN 0-201-54330-3, <http://www.research.att.com/~bs/dne.html>

Also to make matters worse, by the time this experience has been gained, another divergent innovation appears in the market and the next project is implemented using yet more half-baked divergent innovations. In this way, in some organizations it is possible that every project deployment is implemented on partial and incomplete ‘innovative’ technologies, implemented by people with little or no real experience of that technology. I have not even addressed issues of on-going system maintenance. This is not a happy scenario!

Zero Programming Innovation

The next ‘logical’ step is to attempt to remove the programming problem introduced by this divergent innovation, with yet more unnecessary technology, and one approach involves attempting to remove the requirement for any programming whatsoever. The idea here is to introduce two classes of direct users: ‘cone-heads’ that can write programs, and the ‘power user’ who for whatever reason use the technology without programming. At this point there is a conceptual shift in thinking. The aim of the technology moves from better enabling qualified developers to be more productive in languages already well understood, to attempting to make non-programmers able to write programs without any of the training necessary to actually write programs! An argument by analogy might clarify this shift; it’s like deciding that rather than developing technology to better support qualified surgeons in the operating theatre, we develop technology to allow the medically untrained public perform certain types of surgery.

Of course to make this process feasible to any degree, a huge number of protective safe-guards must then be built into the technology to stop the untrained from introducing errors, and these same safe-guards, unless very carefully designed, also limit qualified programmers and actually impede development compared to using components released as simple language libraries in the first place.

This ‘cushioning of the untrained’ from the consequences of their incompetence typically comes at a very high price, and in two basic flavours. First there is the *Configuration Approach*, where the direct user is expected to set the values of a set of parameters which control various aspects of the system, and the *GUI Approach*, which is just a more aesthetic version of the former, where the original functionality is buried in a sea of menus, and graphical widgets, along with all the protective safe-guard code and peripheral technology which must be catered for. It is also often necessary for the designers to predict many different usage contexts and explicitly cater for these with horrible hacks and add-on technology which massively add to the overall complexity with consequent implications for performance, reliability and scalability – not to mention cost.

To return to our CEP example, the GUI might be used to set up the condition-action rule base which determines how detected events are processed. This might involve using an innovative GUI based ‘point-and-click’ interface to laboriously enter this information rather than writing program code in an editor. Although this certainly stops the untrained from writing invalid language expressions and getting ‘compiler’ errors, it seriously limits the productivity of programmers who could write this same code using established languages and compilers much more easily, were a language library provided.

To illustrate this situation further we can envision other drawbacks; the first concerns *extensibility*, the second concerns *integration*. If these untrained users need to use some arcane mathematics function not supplied or accessible from this GUI what options are there? By definition they are untrained so are unlikely to be able to program this from first principles themselves, and even if they could they cannot use the GUI to implement this functionality because a GUI is *fundamentally not extensible*.^[8]

The problem of integration can be aptly demonstrated by considering how to get the events into the CEP system from outside. This requires add-on technology to interface between the ‘innovative’ system and other system processes comprising its operational context, and which are both senders and receivers of these events. This add-on technology must convert these events from/to the format used by the surrounding systems, say Java or C++, to/from a format suitable for consumption by our CEP component. This typically involves writing the add-on component in the language of the existing technology, in this case Java or C++, and then converting the event data into a format which is the ‘lowest common denominator’ – usually some form of ASCII, and maybe even XML – and then providing ways of reading this string format into the ‘innovative’ CEP system. Generally speaking all this effort compromises performance, increases complexity, decreases reliability, increases cost and is, from a technical perspective, utterly pointless.

Conclusion

By way of summary then, genuine innovation occurs on a daily basis within the IT industry and mostly where mature, extensible general purpose programming languages, such as Java and C++, are employed. This is because using a general purpose programming language allows the developer to write genuinely new code using novel algorithms and data structures and fully exploit the knowledge of a particular context of use. Such innovation is much harder or virtually impossible without an extensible general purpose programming language.

Radical innovation, the sort that changes the entire paradigm of the IT industry is infrequent, maybe requires genius, but almost certainly requires the innovator to work solving complex problems within the real world using existing technologies, to provide opportunities of finding the flaws within these technologies and coming up with better alternatives that not only solve all the problems that the existing technologies solve, but also enable superior solutions to some sub-set of these. If a new technology solves some programming problems more easily than existing technology, but fails to solve already solved problems, or introduces more complex problems than it solves, in what sense is it an improvement?

An analogy with scientific theories is useful here. If by introducing relativity theory Einstein could explain why light curves around massive objects, something Newtonian mechanics cannot adequately account for, and yet could not explain why apples fall downwards, which is explained by the Newtonian system, would his ‘relativity theory’ represent an advance in science? To be accepted, new scientific

⁸ Some GUIs can be extended in simple ways, such as tear-off menus and tool-bar configuration, but this is a very limited form of extensibility compared to using a general programming language.

theories must explain all the phenomena explained by existing theories, they must do this more elegantly and they must make predictions which differentiate them from other competing theories, such that subsequent experimental observations can falsify the existing theories ^[2]. A similar criterion will help to identify radical innovation within computing and differentiate it from mere hype.

The use of innovative proprietary programming languages is a case in point, especially ones specialised for particular tasks; these commonly present problems. These languages are often immature, do not have the standard of optimizing compiler available in mature languages such as C++ or Java, so performance can be several orders of magnitude slower, and when used ‘in anger’ to solve real world problems the programmer quickly encounters situations that require venturing ‘off piste’, situations not envisaged by the language designers, where the non-general, limited nature of the language or the lack of access to features typically supplied in good 3rd party libraries, can stop progress dead. The problem cannot be solved within the language, at least not without intensive hacks.

In short proprietary, non-general programming languages, however innovative, which claim to improve productivity, are largely a myth, propagated by people whose main aim is to lock users into expensive software. This is an example where the ‘innovation’ may superficially better solve a limited set of problems within its specialist domain, but fails to adequately address problems outside its domain, *which need to be solved by users of the specialist language working in the real world nevertheless*, and which are properly accommodated by existing general programming languages.

In general one simple way to ensure that any innovative technology can solve all the problems already solved by existing technology is to *distribute this innovative component as a programming language library*. This does not exclude the possibility of providing additional GUI based ease-of-use tools, but these tools should be written as components that use the main language library. In fact, given the current IT environment I propose a simple rule-of-thumb that any innovation not distributed as a Java or C++ library constitutes a significant risk to any project that attempts to use it, particularly in the long term, and this should count strongly against preferring it to other candidates which are packaged as such. The great innovator, the physicist Richard P. Feynman in 1985^[9] understood exactly this problem...

“..there is a great deal of work to try to develop smarter machines, machines which have a better relationship with humans so that input and output can be made with less effort than the complex programming that’s necessary today. Another problem is the standardization of programming languages. There are too many languages today, and it would be a good idea to choose just one.” Richard P. Feynman ^[10]

Perhaps one language really is too few, but it’s hard to see how anyone could seriously entertain the idea that ‘what we need is another computing language’, and

⁹ http://en.wikipedia.org/wiki/Richard_Feynman

¹⁰ Computing Machines in the Future, by Richard P. Feynman, Nishina Memorial Lecture (1985), Nishina Foundation and Gakushuin

with the plethora of so called 'innovation' today, prospective innovators could do everyone a favour and attempt to *converge* on to a smaller set of underlying languages and technologies. The obvious language candidates here are Java and C++. This will technically benefit everyone in the IT industry, most businesses, and increase the chances of genuine innovation.