

The Hardship of Software

Erik Ernst
Department of Computer Science
University of Aarhus
Aabogade 34, DK-8200 Århus, Denmark

ABSTRACT

This paper argues that being flexible (soft) or predictable (hard) is the most fundamental dimension in the design of programming languages including aspect-oriented ones, giving rise to myriads of derived trade-offs, and emerging in many different forms and variants. Based on a presentation of some of these forms and their concretizations into language design choices, we conclude by proposing an overall principle which may help making sound choices when choosing a location in the associated language design space.

1. INTRODUCTION

The so-called ‘ilities’ denote qualities of software artefacts, and they have been discussed in the AOSD community for several years [8]. There are numerous ilities to consider for those who venture into the dangerous realms of programming language design, e.g., comprehensibility, evolvability, reliability, scalability, extensibility, composability, etc. For each design decision taken, the effects on all those qualities in realized software systems or components may be radical, subtle, or even both.

However, we claim that many of these ilities can be viewed as consequences of more low-level, technical properties, e.g., as a result of interactions between the human mind and software artefacts. In particular, we propose the dimension of being flexible versus being predictable as a very fundamental dimension at the technical level, where a large number of design decisions represent a certain choice in the space which is spanned by this dimension and its associated subdimensions. We illustrate this by enumerating a number of such subdimensions and giving examples of the concrete forms they take in connection with various design situations. Based on this, we propose a principle which may be applied in language design, thus giving it a direction and intentionality which makes it easier to understand which kinds of trade-offs are being made in this process, and making it easier to arrive at a balanced design and with consciously chosen properties.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. *SPLAT’05*, March 15, 2005, Chicago, IL, USA

The paper first discusses a very direct expression of the trade-off between flexible and predictable, namely randomness versus determinism, in Sect. 2. Next, Sect. 3 discusses one important tool which may be used to control the level of randomness, in a sense, namely the choice between a dynamic and a static version of various mechanisms. Section 4 continues this development by outlining a design choice which is often at a more detailed technical level, but which has serious implications for the range of possibilities in that choice of dynamicity; we use the terms ‘direct’ and ‘compiled’ to describe this subdimension. In Sect. 5 we consider a dimension which cross-cuts all the previous ones, namely that of supporting complexity versus enforcing simplicity. From this development, we derive a principle which can help making language design choices informed and with more conscious trade-offs, in Sect. 6, and finally Sect. 7 concludes.

2. RANDOM VS. DETERMINISTIC

To outline this subdimension, imagine a computer which is capable of executing instructions in a similar way as an ordinary computer, but rather than fetching the instructions in the memory of the computer it decides on which instruction to execute based on measurements on some quantum phenomenon associated with the cosmic background radiation. Such a computer would be capable of doing anything which may be achieved by means of a computer (it isn’t even limited by decidability boundaries) and hence it is extremely flexible, but there would be no way to predict what it will do in any particular computation, and it is extremely unlikely that it will be useful for any concrete goal. Hence, there is a need to impose some restrictions on the ultimately flexible computation, such that it becomes useful.

To pinpoint the other extreme, imagine a computer which executes a program in a simple linear language, i.e., a language supporting some simple operations like standard computations on fixed size scalar values and finite length text strings, but without choice (like `if`-statements) or loops (like `while`-statements). The execution of such a program would start at the beginning and proceed with exactly the same control flow and exactly the same computations every time. This computation might be useful for a given concrete purpose, but it would be useless in practice because it is so predictable that multiple executions of the program will always produce exactly the same results as those produced by the first execution.

Taking one step from the first extreme towards the other one, consider genetic algorithms [17]. Here, small programs are constructed randomly and then executed by agents in a virtual environment where a given goal and a given goal satisfaction measurement are used to promote some of the agents and suppress others, and reuse of good agents in ‘mutation’ or ‘mating’ processes is used to direct the somewhat random construction of new agents in order to produce better and better agents iteratively.

Conversely, by adding interaction with the environment (input/output), choice (such as `if`-statements or late binding of object-oriented methods) and loops (such as `while`-statements or recursive procedures/methods), the completely predictable execution may be enhanced to a predictable function of the behavior of the environment. This model is so rich that it has proved useful in practice for decades, in the shape of mainstream procedural and object-oriented languages [13, 14, 20, 10].

Generally, computing outside special communities such as artificial intelligence and genetic algorithms support only this level of randomness as a core language feature. However, threads, parallel execution, and asynchronous communication (events) provide a potential for ‘randomness’ which enables more flexibility of software systems in the environment. Even crude simulations of multi-threaded execution such as the event loop based programming style known from graphical user interfaces enable software systems to choose what to do next based on the given events. This is ‘randomness’ because the events might as well have been generated more or less randomly as far as the software system is concerned, and at the same time it allows for tight control with the execution from the environment. It is crucial to achievement of useful results that the randomness is used as a ‘possibility generator’, and various constraints on the execution are used to select good choices during the execution among all the available ones. Event based GUI programming is an example where basically only one possibility is left open at each point. We believe that it would be valuable to explore greater degrees of freedom.

Aspect-oriented languages such as AspectJ [7], Hyper/J [21], and AspectS [12] enhance traditional languages with a number of new concepts and constructs, but generally they are confined to the ‘deterministic function of environment behavior’ universe. We believe that there is ample room for extensions in this dimension, and also that the expressive power of generating many possibilities and choosing among them is immense. Just consider such an idea as an `if`-statement which randomly chooses between the `then` and the `else` branch, along with a kind of advice which restricts the choice to a deterministic one, thus collapsing the non-deterministic branching to an ordinary predictable `if`-statement in a non-invasive manner; or how about non-deterministically choosing which method to call at some point, or which in order to execute a set of statements, along with non-invasive, advice-like restrictions which may be used to enhance the predictability towards traditional deterministic constructs.

3. DYNAMIC VS. STATIC

Now we focus on mostly deterministic execution, controlled by the behavior of the environment. Within this universe we can outline a similar spectrum of flexibility vs. predictability which is based on the ‘time’ at which decisions are made with respect to the program elements. Dynamic mechanisms are the ones which are executed at run-time, whereas static mechanisms are executed earlier, during compilation, linking, loading, or whatever phases precede the actual execution.

At the most flexible end we find executions based on programs which may change in almost arbitrary ways at any point during the execution. A prime example of such systems would be programs in CLOS [2], where the rich support for meta-programming can be used to modify the running program while it is being executed. Image based languages/systems such as Smalltalk [9] or Self [23] provide other examples of a very high level of flexibility at run-time.

At the other end of the spectrum we find such a language as C++ where priority has been given to avoid run-time features in order to preserve the performance of C, and to some extent in order to provide programmers with support for detailed static analysis. Also at this end of the spectrum we find very different languages like Standard ML [16], where the strict static analyzability and suitability for mathematical modeling has priority, even though the performance benefits derived from static analysis are also appreciated.

This trade-off is well-known in the area of aspect-oriented language design, and apart from the liberally self-modifying approaches there are several proposals detailing such ideas as dynamically enabling or disabling advice, e.g. [4; 3, e.g.], or dynamically selecting which advice to apply, e.g. [6].

Probably the most interesting novelties in this area would be incremental: How could we make various constructs a little more dynamic in order to achieve extra flexibility, or how could we make them more a little more static in order to make them more analyzable and predictable?

4. DIRECT VS. ‘COMPILED’

When a choice has been made as to whether a given mechanism should be provided in a dynamic or a static form, one of the consequences is that the static elements may open opportunities in the direction of some kind of systematic partial evaluation of the program. By partial evaluation we mean expression of the program for the run-time environment in a form which is less general than its original (source) form, based on an analysis which exploits the given predictabilities of the run-time execution; the term partial evaluation is relevant here because the less general form is in principle reached by carrying out some of the computational work already at compile-time. For example, in the Java programming language it is possible to determine statically whether a method invocation will be made on a class type or an interface type, and in the latter case the restriction to single inheritance for classes enables certain forms of sharing of run-time structures (vtables) which are not possible with interface invocations.

We use the term ‘direct’ to characterize a direct and naïve treatment, usually with a substantial performance penalty,

and ‘compiled’ to characterize the more optimized strategies which exploit some kind of partial evaluation. Note that seemingly direct and dynamic approaches may well be very sophisticated and highly optimized by means of run-time compilation etc., but it may appear to be a direct treatment as long as the flexibility is preserved, e.g., by being able to undo and redo the compilation and optimization. In relation to predictability, however, when the entire array of flexibility has been preserved then the potential for unexpected developments is also preserved, which means that they count as ‘direct’ in this context.

In an aspect-oriented context, the static resolution of pointcut expressions as in AspectJ is a typical example of this trade-off, with [3] as an example of a development towards a more dynamic treatment. In several formalizations of aspects based on the lambda calculus as well as in some approaches based on Scheme [22], it is common to model advice application as an entirely dynamic process, i.e., for each function application the whole database of advice is searched and the applicable ones are applied. In contrast, AspectJ programs are compiled in such a way that each advice is generally only considered at a very restricted set of program points, e.g., because it is known statically that a given advice will only ever be enabled at the invocations of a method having a specific name and signature.

The overall significance of this subdimension is that compiledness restricts dynamism, and premature compiledness may give rise to a lack of flexibility which is never noticed because it is taken for granted. Conversely, a useful mechanism may remain marginal because the missing realization of some kind of compiledness potential makes it much more expensive and/or less predictable than necessary.

5. COMPLEX VS. SIMPLE

Relatively independently of whether or not randomness is exploited, whether the mechanisms are dynamic or static, or the extent to which compiledness has been used, languages may have differing support for complexity. This has consequences for the trade-off between flexibility and predictability in the following way.

Generally, the size of the set of possible executions depends on the level of sophistication of the supported language mechanisms and constructs. For instance, the extremely limited linear programming language mentioned in Sect. 2 allows for just one possible computation. Adding new features such as objects, dynamic dispatch, and recursion, the same size program allows for many more different executions. It is hard (and probably useless) to precisely quantify the number of possible executions, but there are a few categories of concepts which will definitely increase that number when added.

The basic point here is that a larger number of possible executions yields more flexibility but less predictability. In other words, the level of complexity supported by the available selection of language mechanisms—ultimately by the set of programs which can be expressed using them—this level of complexity directly influences the trade-off which is the main focus of this paper.

In general, we want to allow for a large set of possible executions, but this set should have so much internal structure and consistency that we can gain a lot of expressive power and flexibility and still preserve the property that executions are sufficiently predictable. Quite simply, we want to have the option of selecting among many useful program executions without excessively increasing the danger of selecting useless, meaningless, or even damaging ones.

First, consider value parameterization. The expressive power in procedures or methods taking arguments is obviously much greater than the expressive power in parameterless procedures and methods. Of course, parameter passing may be simulated by means of explicit stack management or similar techniques in context of any given Turing-complete language, but the amount of useful work being done in a program of a given surface complexity will intuitively definitely be greater when parameters are available. Similarly, in a typed language, type parameterization of classes and methods enables more expressive and flexible programs than parameterless typing, all other things equal.

In an aspect oriented context, value parameterization of pointcuts and advice is a well-known mechanism which brings new ideas to the already mature topic of parameter passing mechanisms for procedures, functions, or methods. There have also been proposals about type parameterizing aspects [19], but this topic is certainly a non-trivial challenge because aspect instances are created implicitly so it may be hard to determine what actual type arguments to use at creation time.

Next, consider context sensitivity. Basically, object-orientation could be said to be about enabling the dynamic choice of an appropriate behavior for an operation (which is then known as a method) in context of a given stateful environment (which is then known as an object), by means of a dynamic association of the method with the object. In other words, we automatically get the right behavior because we go to the object and ask *it* for a suitable implementation of the given operation. Context sensitivity is also the crucial concept behind virtual classes/types and family polymorphism [15, 5], which is a powerful tool for managing mutually dependent types safely, but in a polymorphic manner.

In the aspect-oriented world, Object Teams [11] and Dynamically Composable Collaborations [18] provide examples of context sensitivity (by being a member of an object team or a collaboration) being applied to objects which are used in an aspect-instance-like manner.

It is likely new and useful concepts and mechanisms will be derived from both parameterization and parameter passing mechanisms, as well as context sensitivity and context construction and selection, together with yet more abstraction mechanisms. In any case, this part of the trade-off provides flexibility by supporting complexity, and provides predictability by controlling complexity, and this trade-off may be made both in language design when the level of sophistication in the provided mechanisms is chosen, and in system development when the usage of the available mechanisms is chosen.

6. PRINCIPLE

From the discussion so far, it should be evident that the trade-off between flexibility and predictability has a large number of down-to-earth technical interpretations and implications. We believe that this trade-off is in a sense at the primitive level, because it may be associated with the direct, technical choices made in the design of programming languages and similar devices. Many other ilities may be considered as derived in the sense that knowledge about the flexibility/predictability trade-offs provide useful input towards a characterization of those other ilities. For example, comprehensibility is an aspect of the reaction by human developers when confronted with software artifacts, and this is greatly influenced by the predictability of the behavior of the software, which is again related to all the subdimensions mentioned in the previous sections. For another example, evolvability is essentially a kind of flexibility where special emphasis is given to the ability of some software artifacts to be easily changed in ways which fit into the usage context dynamics, and again complexity may hamper evolution even though it may be possible in principle. From this discussion we derive the following principle:

When designing language mechanisms, and when applying language mechanisms during system development, consciously choose a suitable trade-off between flexibility and predictability. If possible, provide a large amount of flexibility at a basic level, and then use declarative restrictions to improve on the predictability of complex entities.

As an example of this approach, consider the notion of Open Modules [1] which equip modules with explicitly exported pointcuts and prevents pointcuts from matching locations in modules that don't export them. The exported pointcuts are declarative in the sense that they just have a name and a parameter list, whereas the actual implementation of the pointcut may be altered without affecting clients of the module. At the same time, the constraints on pointcuts ensure predictability in the sense that there cannot be unforeseen pointcuts from external clients which destroy the maintenance of the invariants of the module. All in all an improved trade-off between flexibility and predictability in context of modern software development, giving rise to improvements in many other ilities including comprehensibility, evolvability, and scalability.

7. CONCLUSION

We have argued that the dichotomy of flexibility versus predictability is a particularly fundamental and primitive dimension of interest when considering ilities in general, and we have presented a multi-step break down of this dichotomy into randomness vs. determinism, dynamic vs. static mechanisms, direct vs. 'compiled' techniques, and finally complexity as a cross-cutting dimension. These very general considerations have been connected with more specific aspect-oriented considerations at each step, giving rise to some ideas about possible enhancements and generalizations. Finally, based on this discussion we have proposed a principle which may be used to choose a good location in the programming language design space.

8. REFERENCES

- [1] Jonathan Aldrich. Open modules: Reconciling extensibility and information hiding. In *Proceedings of SPLAT'2004*, 2004. Available at <http://aosd.net/workshops/splat/2004/index.html>.
- [2] B. Bobrow, D. DeMichiel, R. Gabriel, S. Keene, G. Kiczales, and D. Moon. *Common Lisp Object System Specification*. Document 88-002R. X3J13, June 1988.
- [3] Christoph Bockisch, Michael Haupt, Mira Mezini, and Klaus Ostermann. Virtual machine support for dynamic join points. In Gail C. Murphy and Karl J. Lieberherr, editors, *Proceedings of AOSD'2004*, pages 83–92, Lancaster, 2004. ACM.
- [4] Jonas Bonér. Aspectwerkz – dynamic AOP for Java. Invited talk at AOSD'2004, 2004. Available at http://aspectwerkz.codehaus.org/papers_talks.html.
- [5] Erik Ernst. Family polymorphism. In Jørgen Lindskov Knudsen, editor, *Proceedings ECOOP 2001*, LNCS 2072, pages 303–326, Heidelberg, Germany, 2001. Springer-Verlag.
- [6] Erik Ernst and David H. Lorenz. Aspects and polymorphism in AspectJ. In *Proceedings of AOSD'2003*, pages 150–157. ACM, 2003.
- [7] Gregor Kiczales et al. Aspectj home page. <http://aspectj.org/>.
- [8] Robert E. Filman. Achieving ilities, December 06 1999.
- [9] Adele Goldberg and David Robson. *Smalltalk-80: The Language*. Addison-Wesley, Reading, MA, USA, 1989.
- [10] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification – Third Edition*. The Java Series. Addison-Wesley, third. edition, 2004.
- [11] Stephan Herrmann. Object teams: Improving modularity for crosscutting collaborations, August 23 2002.
- [12] Robert Hirschfeld. AspectS – Aspect-Oriented programming with Squeak. In M. Aksit, M. Mezini, and R. Unland, editors, *Proceedings of NODe 2002*, LNCS 2591, pages 216–232. Springer-Verlag, 2003.
- [13] K. Jensen and N. Wirth. *Pascal User Manual and Report*. Springer-Verlag, 1978.
- [14] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1988.
- [15] Ole Lehrmann Madsen and Birger Møller-Pedersen. Virtual classes: A powerful mechanism in object-oriented programming. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices*, volume 24, 10, pages 397–406, October 1989.
- [16] R. Milner, M. Tofte, R. W. Harper, and D. MacQueen. *The Definition of Standard ML*. MIT Press, 1997.

- [17] Melanie Mitchell. *An Introduction to Genetic Algorithms*. Complex Adaptive Systems. MIT-Press, Cambridge, 1996.
- [18] Klaus Ostermann. Dynamically composable collaborations with delegation layers. *Lecture Notes in Computer Science*, 2374:89–110, 2002.
- [19] Raul Silaghi and Alfred Strohmeier. Better generative programming with generic aspects. In *Second International Workshop on Generative Techniques in the Context of MDA, held at the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA, Anaheim, CA, USA, October 26-30, 2003*. Also available as Technical Report IC/2003/80, EPFL (Swiss Federal Institute of Technology in Lausanne), December 2003.
- [20] Bjarne Stroustrup. *The C++ Programming Language (Second Edition)*. Addison-Wesley, Reading, MA, USA, 1991.
- [21] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 1999 International Conference on Software Engineering (ICSE'99)*, pages 107–119, Los Angeles, May 1999. Association for Computing Machinery.
- [22] David B. Tucker and Shriram Krishnamurthi. Pointcuts and advice in higher-order languages. In *AOSD*, pages 158–167, 2003.
- [23] David Ungar and Randall B. Smith. Self: The power of simplicity. In *Proceedings OOPSLA'87*, pages 227–242, Orlando, FL, October 1987.