Intentional Programming: Asymptotic Fun?

Charles Simonyi
Microsoft
charless@microsoft.com

This note is based on Gregor Kiczales' position paper on Aspect-Oriented Programming: "AOP, The Fun has Just Begun". I am a big fan of AOP. It has working and useful examples, and it brings into focus a key fault of traditional languages and OOP in particular, namely that modularity or refinement by itself can provide for a proper separation of concerns. It can not. Gregor describes the false hope of the traditionalists this way: "*Many people … suggest …[that such concerns] could be modularized in other ways including the use of patterns, reflection, of 'careful coding'*." I can also testify that this happens, especially the "careful coding" exhortation which is used frequently by language designers and by management to tell programmers that if the software is difficult to modify or difficult to share, the problem is with them – their coding must be obviously below par, since the language that they must use (ADA, Java, Modula, etc.) has many advertised features that support modularity.

The strange thing is that programmers actually buy into this nonsense and frequently feel guilty – programmers in this instance are a little like many end-users of beta software who feel that the system crashes are due to their abuse of the software or lack of understanding of how to use it, not a fault in the software that needs to be fixed. Such beta-testers find an enthusiastic audience in the lazy programmers who loathe fixing the problems, which completes the parallel, because language designers and management are also very conservative folk and rather ascribe the undeniable problems to someone else.

The differences between Aspect Oriented Programming (AOP) and Intentional programming (IP) are due to their different foci: AOP looks at programs and seeks "cross-cutting aspects" which localize the expression of that aspect with all the benefits that such localization affords. IP, on the other hand, looks at the specs, the stakeholders' concerns and strives to faithfully represent those concerns in an essentially XML tree/graph like form. It is understood that in such systems the source – that is the summary of the stakeholders' contributions – is viewed in any number of display projections, so the issue of syntax or notation is orthogonal to the main issue of what is being represented. It is also understood that the implementation is generated by a "compiler", "weaver" or transformer which, in case of AOP somehow has knowledge of the aspects and in case of IP has knowledge of the "intentions", that is the generic parameterized nomenclature in which the specs were expressed, using the terms of the problem domain.

Almost by definition, the stakeholders would express their contributions in terms of crosscutting concerns, whenever a concern has effects on multiple parts of the implementation. One could say that IP is an extreme form of AOP: in IP everything is an aspect and everything presents a task for the aspect weaver. Another way to say this is that IP focuses on the original expression of the problem, while AOP (in a pragmatic way) focuses on the programmatically encoded expression of the problem and attempts to improve it.

The greatest effect of this difference in focus is in the size of the vocabulary. First a definition. Traditional computer languages distinguished between two kinds of words – those that were "part of the language" and were defined in the manual and those that were defined by the user. The former were called "keywords", and sometimes "operators". The latter were called "names".

I define "vocabulary" to be similar to natural language – it is the set of distinct identities that are familiar to those who need to communicate – in this case the stakeholders and the transformation system (or, if you will, the human creators of the transformation system.) We must be careful not to get confused by homonyms and the like. It is irrelevant that a word in English might have several meanings – those who wish to communicate must be familiar with every relevant meaning – that have distinct identities. For example, the word "form" is merely an English notation for a verb and several nouns that are listed in a dictionary as form(1), form(2), and so on.

Clearly, in a traditional computer language, the vocabulary consists of the keywords. In AOP, the number of keywords in increased to reflect the aspects that the system can process. In IP, the vocabulary is expanded as necessary to precisely record the stakeholders' concerns. The distinction between keywords, operators, and names is greatly reduced. The size of the vocabulary depends on the state-of-the-art in the problem domain, hence it may be very large indeed.

Traditional languages are based on a number of assumptions which were almost self-evident in the early days of computing, but which are worth re-examination given our design needs and the capabilities of our computing systems. These assumptions are listed below. For the sake of clarity, I have to stress that I believe that these assumptions are wrong, even though they are cornerstones of our current thinking and underlie practically all software engineering research. They are also contrary to the needs of IP.

1. *Programs should be expressed in unambiguous languages with syntax and names*. I will not spend time on this, since it takes a long time arguing against it and it is not strictly necessary to win this argument. But I will point out that replacing syntax with projections and replacing (or rather augmenting) names with identities goes a long way to make the solutions for the other key problems practical. Conversely, hanging onto these old ideas that in their bases are related

to paper tape and punched cards issues, seriously retards progress in software engineering.

2. *Languages should be simple*. This has really been post-facto reasoning, insofar as language definition has been so difficult and language learning has been so unrewarding that there has been always a powerful incentive for the creators as well as the user of languages to keep the languages simple. On the definition side, the difficulties had to do with the need for precise definition (which is the next item) and the need for providing unambiguous and user-friendly syntax for constructs (which was the previous item.) For the users, the learning of new languages is a nearly meaningless chore: new ideas are few and far between and most of what is called "learning" is to re-discover common design patterns using the terms of the language to be learned – which activity is made difficult because this is not part of any documentation and because there isn't a reference meta-language in which the design patterns could be identified. In a new language, trivial things like loops may become an object of concern for somebody who has just learned the language – let's see, how does one loop N times in this idiom? The creation of an iterator may become a design problem even for an "experienced" programmer.

   The goal of simplicity is to save effort in some dimension. But there is no a priori argument that a simple language saves total effort, just like a simple toolbox may not help with a repair job, or a simple airplane is not necessarily a reliable airplane. More likely, a simple language simply displaces effort into the "user realm", where a large number of arbitrary names and arbitrary design patterns create an incredible obstacle to software sharing, to program understanding, and to mechanical reasoning about the program. But these horrendous costs are thought to be natural, and can be somehow attributed to the programmers, to the problem or the problem domain, not to the language.

3. *Language semantics should be precisely defined*. At some level this should be unobjectionable – but if we look at what was meant operationally by this we can see that it is a snare. The point is that we should reserve precision for expressing the stakeholders' concerns – and here precision means something different than exactness since, as we shall see, stakeholders often do not have exact concerns. They have general concerns. Being more "precise" than the intention of the stakeholder is in fact an act of "imprecision", so simply demanding "precision" can be frequently an oxymoron.

   We should have a large distinct vocabulary in which to express the concerns precisely. There is no need to be too precise about the implementation of these concerns, in fact such precision is detrimental to the value of the program. A few analogies might help: "I have a date with partner P to see "Magic Flute"" is less precise, but more intentional than a set program, such as "At time T1: Goto apartment at address A(P); ring bell; say hello; goto theater O1 /* Magic Flute playing */; present tickets K1, K2; wait until T2; etc. etc." Note the comment: the

> real payload is usually contained in comments in conventional programs.
>
> From mechanical engineering: we might say: “the piston is to fit the cylinder” instead of saying that “piston.diameter = 4.999”.
>
> In both examples the second version is arguably more precise, but clearly less useful for *all* purposes except maybe from the point of view of the processor that executes the implementation. After all, to implement the date, some specific actions will have to take place, for example the bell may have to be pressed (but not necessarily.) Similarly, at the end of the day the piston will have to have some diameter (but not necessarily 4.999, whatever that is – a real measurement would have units and tolerances also specified.) Effectively, we have been optimizing our software efforts for the benefit of the transformers, the compilers, instead of our human selves.

We can see that the intentional expression already subsumes “cross cutting” aspects, insofar as they are inherent in the intention. The idea of a “ticket” no longer shows up in some “present ticket” statement but can be an integral part of the “performance” of which “Magic Flute” is an instance. The connection between the cylinder and the piston is not some aspect that needs to be discovered, but an integral part of the model (and as Gregor says, the *model is the program*). If we wanted to change, parameterize, prove theorems, quote, adapt, handle error conditions, improve implementation, tune; we can do all of those easier with the intentional version, and in fact that is why real life is so resilient, for example if my partner works late and goes directly to the opera where we meet. In that case, the conventional program would have to change completely, but the intentional expression remains the same. Of course the transformer would have to “know” about the alternate implementation.

Which brings us to a key issue: how did the transformers learn about the intentions – the concepts of “dates”, “operas”, “tickets”, “fitting” and so on? Well, there is no free lunch. Intentional programming does not save total initial effort – it merely re-factors it. The specs, the comments, and some of the code are expressed as IP source, while the implementation details are expressed as transformations. Arguably, writing transformations is harder than writing code although this is not clearly so and the cost differential is probably a wash. One way to look at this is that writing a transformation that creates implementation X is like writing ‘printf(“X”)’. The benefits start to accrue when the intentional code is changed, shared, or reasoned about, or if the implementation is improved – because the transformation can be re-executed mechanically. We can also see how the cross-cutting concerns are expressed: they appear as parts of the transformer. Indeed, the transformer may be as “messy” as the traditional program would have been, except that it can be re-executed with new parameters, or the transformations can be changed to change the implementation – while the parameters are kept constant.

The re-factoring of the program into an intentional specification and a transformer with the implementation information also means that the skills of the people are factored and greater specialization is enabled.

Today we are facing a crisis of personnel. First, we do not have enough software engineers. Second, the software engineers are not sufficiently skilled in various problem domains, such as human factors, or flight dynamics. But this crisis is created by our insistence that programmers write programs. We have sufficient number of people skilled in human factors or flight dynamics – they just can not program real-time Ada. Conversely, if programmers did not have to worry about the human factors or flight dynamics, there would be much fewer of them needed.

The most bizarre manifestation of this waste is the notion of "HTML programmers". These unfortunate individuals are human transformers of very simple layout intentions into the machine code of the web, namely Html. They are not unlike the "computers", as defined in pre-WW 2 dictionaries which were human clerks performing computation whether by hand or by some mechanical help. Of course practically all programmers today spend some of their time doing transformations – the transformation from an implicit or explicit spec or understanding of the model into an executable implementation.

Under IP, domain experts write models/specs/programs in domain terms. Programmers write transformers that transform the domain terms into machine implementations. There is no repetitive work – or rather the repetitive work is done mechanically. When specs change – requirements change, parameters change, or the specs are in error, the transformation is re-executed. When implementation is changed, the work is done once to the transformer, and then the specs are re-transformed mechanically.

The bifurcation of domain expert and programmer can repeat in a fractal fashion: some domain experts might provide transformations for domain implementation (for example the different kinds of date scenarios or the different forms of fitting machine parts.) Other domain experts might be "end users" who actually go on dates or actually fit parts together. Yet others could create powerful abstractions for the writing of transformers. So a programmer would be just another domain expert, someone who focuses on the application of abstract computer science: sorting, searching, numerical algorithms, real time concerns, machine architecture exploitation, graphics algorithms, etc.

Miscellaneous FAQ

By reductio ad absurdum, why couldn't the program simply consist of a single node embodying the totality of the stakeholders' concerns? For example, to develop a spreadsheet, one would say "Excel", which would be absurd – it would just move the Excel development work into the transformer (under the "printf").

> There are two answers. The less interesting one is that such a primitive intention is not entirely useless, especially if it can be parameterized. But more interestingly, stakeholders typically have fairly specific ideas of what would satisfy them – this is not unlike how one talks to an architect about a new family house to be built. One does not simply say "house" to the architect – one needs

> many hours of discussions and many pages of correspondence – some of which very appropriately is called the owner's "program" for the house. The size of the vocabulary properly depends on the scope of the stakeholders' concerns. One can see that an airport authority would probably have a more complicated discussion with an architect than someone interested in erecting a trailer home.

What are multiple views?

For example Gregor Kiczales writes:

> *A central idea in all science and engineering [is] the idea of working with a system through multiple perspectives or views.*

This is important and needs to be clarified. A program will have three manifestations: lets call them content, notation, and implementation. At any moment content is unique, but there may be several notations and implementations mechanically derived from the content. "Multiple perspectives" is a valid concept for the content – meaning that programs should record relevant organization and design choices in many different aspects – hence AOP – and it is also a valid concept for notations. However these are different ideas – the former is more basic – and less obvious – while the latter, the notations, are simpler. A perfectly workable system can be developed with just one notation – for example LISP or XML – but for practical purposes, multiple notations will be a must to enable the human programmer to interact with (to read and to edit) the program contents.

The best example of this distinction can be found in CAD systems where both of these principles are readily and naturally accepted. Systems of extreme complexity – to wit a Boeing 777 airliner – are successfully created, edited, proved partially correct, and realized in an implementation – using an Integrated Development Environment. In this case the IDE is called CATIA – and the best part is that there is no CATIA "language", and certainly there are no time-consuming arguments about the "syntax" of the CATIA "language". Engineers simply use the system to encode those aspects of the artifact (the airplane) that is important to them and which they can make use of. The issue of notation (how an aspect looks on the display for interaction) is not of primary importance and it has never been a problem. For example, let's take fluid temperature within a pipe. If this is important – for example it would be used to test compatibility of materials, or heat loads in compartments – the data can be added to the database without any problems. If the temperature has a gradient, that can be added also – it just means that there would be more properties added to the description of the pipe. What is the syntax for this new statement? To input it, a GUI property sheet may be used. To display it, a range of colors might indicate the approximate temperature of the pipes. What if the colors are already used to show pressure? Perhaps a display mode could switch between the displays of the temperatures and the pressures – neither projection is complete, neither precise, but it could be still convenient and useful.

Note that the "syntax" for the input and for the display can be different – to specify a temperature seems to be different from viewing it. The input method and the viewing method can change – improve globally or adapt to local conditions - without disturbing the "contents". This is in complete contrast with text-based computer languages, where syntax is held in high reverence and is integral to the integrity of the sources.

Why not separate concerns by hiding them in modules?

I once had a very fruitful discussion with Prof. Parnas on this. Parnas started by claiming that one could separate any concern into its own module and then, by definition, we should have perfect separation of concerns. This is undeniable, but we have to ask: what will these modules contain – that is how are the concerns encoded? Indeed, at the limit, the modules must contain a direct identification, in data form, of the concern represented, possibly with parameters that would have to be in their own modules and so on. Effectively, the program would be a massive tree of nodes, each node a module, or a reference to a module – in other words an Intentional Program Tree. To set the program into motion, one has to add some sort of interpreter or compiler or transformer to this mess, but that is not properly part of the problem expression but a resource shared among programs. The point was that when the all concerns are separated, "modules" become so small that they are more like operator instances or operand instances in traditional languages and most information will encoded in the relationships between the nodes, rather than in the nodes (modules) themselves. So in this extreme sense, there is no conflict between Intentional program expression and traditional modularity.

Similarly, there is no conflict and there is a more natural kinship between AOP and Intentional Programming (IP).

So why aren't modules used this way? The above argument of course does not describe anything practical. The notation would be terrible and also it presupposes the existence of a transformer. Providing for the flexible notation and for the transformational phase is what makes IP a capital-intensive proposition.