Appropriate Levels of Abstraction

The software engineering community often aspires to "higher levels of abstraction". Many believe that is the solution to defeat complexity and to increase our productivity. Assembler languages evolved to 3rd generation languages (3GL) like C, C++ and Java. 4GL languages like PowerBuilder and Visual Basic are even higher level and more productive. Some people have discussed 5GLs (although the definition of that varies widely). And now we discuss 6GLs (also here). Platforms, frameworks and design patterns all allow us to work at “higher levels of abstractions”. Rules engines, business process designers and service oriented architecture (SOA) are some current examples with a similar claim. UML, domain specific languages (DSLs), model driven development (MDD) and Model Driven Architecture (MDA) also “raises the level of abstraction”.

But “raising the levels of abstraction” does not describe what should be the real aspiration in software development. Taken to the extreme, at a very high level of abstraction, we could simply draw a box and say that is our system. Obviously a box is at such a high level of abstraction that it is completely useless! More importantly, higher levels of abstraction are not always what we need. For example, higher level tools like the 4GLs are often rejected because they do not allow enough control at lower levels of abstraction.

Instead of aspiring to higher levels of abstraction, we should instead seek to work at the **appropriate level of abstraction** for the problem at hand. The appropriate level is sometimes very high and sometimes very low. It varies for different situations even in the same software project. Just as other engineering disciplines require different tools for different situations, software development also requires tools and languages that support our work at multiple levels of abstraction.

It is not only for programming itself that we benefit from working at the appropriate level of abstraction. Other participants in software projects, like subject matter experts, most certainly work at a level of abstraction different from the programmer. An earlier blog essay described how subject matter experts can participate more effectively in software projects. They can participate at their own comfort level, their own preferred level of abstraction, while the result can be processed by a computer.

But how can we continuously work at the “appropriate level of abstraction” for each problem, especially as the problems change throughout the same project? Let us explore some thoughts on this.

Where does a level of abstraction come from? Earlier blog essays (here and here) pointed out that in choosing syntax and semantics for a programming language, the language designer in effect chooses the **degrees of freedom** the programmer will be able to use. A language with fewer degrees of freedom generally operates at a higher level of abstraction. It tends to be easier to use because it is less complex and has fewer chances for errors. It is, however, less flexible in solving more complex problems. On the other hand, a language with more degrees of freedom tends to have the exact opposite characteristics. It operates at a lower level of abstraction, tends to be more difficult to use, is more complex, has more chances for errors but is more flexible in solving complex problems.

What is the optimum balance for degrees of freedom? The optimum balance is when the degree of freedom matches exactly the problem we want to solve. Obviously the optimum balance varies from problem to problem and domain to domain. Each problem defines what degree of freedom is optimal for that problem. For example, when choosing how to

Back to our question: “how can we work at the appropriate level of abstraction at all times”? In other words how can we both eat the cake and have it too?

Mapping between levels of abstraction is one solution. Compilers are a perfect example of this. Input is a program at one level of abstraction, for example Java, and output is the same program at a lower level of abstraction, Java byte code in this case.

Compilers, however, only allow us to generate a lower level of abstraction. What if you actually want to work and tweak things at a lower level of abstraction? Generative Programming often requires this, at least in some cases. A typical example is editing a UML model and editing C++ code. Roundtrip engineering, as applied in many UML tools, is an approach to this. It allows you to do a change at one level of abstraction, and have that change reflected at the other level of abstraction. For example, a change in the model causes an update of the code. Correspondingly, a change in the code leads to a change in the model.

Unambiguous bi-directional mapping is required to make round trip engineering work. Since by definition each level of abstraction is different from each other, extra annotations are often required to map correctly and completely. To allow any change at more than one level one needs to be able to map each level of abstraction to elements in the other levels. Since that leads to redundant information across models and code, all types of synchronization issues can creep in. You need to be really careful or have really good tool support to avoid messing up.

Another solution to find the optimum balance for levels of abstraction is to separate generated code from code written by the programmer. Generated code should not be touched by the programmer, and must be kept separate from code the programmer can edit. Some tools use this approach, but it is awkward in many situations, for example during debugging or code refactoring.

So to be able to work at the appropriate level of abstraction for each problem, we have to be able to mix multiple levels of abstraction in an effective way, while minimizing or removing the overhead. Each programming language gives us a fixed level of abstraction that we can not typically change. Generative programming can be used to get from one level to another. But to be able to navigate, generate, map, edit, and display in multiple views, always at the appropriate level of abstraction and always semantically connected, we need a new way to both (re)define and integrate different levels of abstraction.

Intentional Software is trying to escape from the premise that we have to choose only one level and forever be stuck at that level. The right approach is an approach where the appropriate level of abstraction can be chosen based on the problem and the domain and levels can be unambiguously mixed without overhead.