Languages evolve too! Changing the Software Time Scale

Jean-Marie Favre
University of Grenoble, France
http://www-adele.imag.fr/~jmfavre

Abstract

Humans will have to live with software for a long time. As demonstrated by the Y2K problem, computer professionals used a wrong time scale when thinking about software. Large software products live much longer than expected. It took a few decades to the research community to admit that software engineering was not only about software development, but above all, about software evolution. However, most academics still consider languages as immutable artefacts. Language/software co-evolution issues are still neglected. Migration issues are however commonplace in software industry... It is therefore time to recognize that languages evolve too. Languages are integral parts of software products. Languages are software too. This paper surveys a few models of evolution taking decades and centuries as time-scales. Then programming languages evolution over the last half-century is sketched by means of a metamodel movie.

1. Introduction

Computer science is new, at least when compared with other sciences like physics, or biology. While evolution and history make sense in these domains, the short history of software engineering is characterized by a series of misunderstandings about the real nature of software. What is software exactly? While most programmers are convinced by their answers to this question (e.g. software=set of programs), the notion of software is far from clear. This notion still evolves.

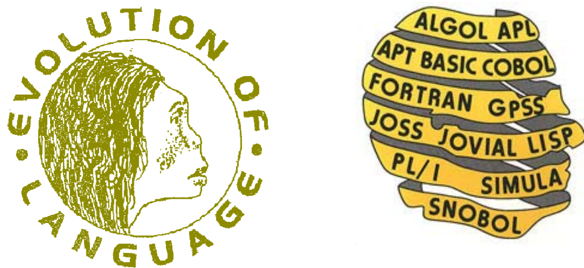


Figure 1 Quiz: how are these two logos related ?
(answer at the end of the paper).

This work has been led in the context of the RELEASE European Network of Excellence on Software Evolution.

The goal of this paper is to study the real nature of software, taking as a research hypothesis that evolution is an intrinsic part of software.

While software engineers use a narrow vision of time, many time scales are required to understand the real nature of software. This paper puts the emphasis on language evolution. It claims that this is a serious research issue, though this phenomenon is usually neglected by academic research. Various historical models are presented for software engineering.

The paper is structured as following. Section 2 attempts to give an insight about the real nature of software. The notion of software time scale is stressed. The notion of metaware is shortly introduced. Section 3 surveys various models of evolution based on a large time-scale. Section 4 deals with computer languages evolution, and in particular with programming languages evolution which is a special case. Section 5 shows how the history of languages can be described by means of a metamodel movie, giving a more precise picture of the evolution of software engineering paradigms. Finally conclusions are given.

2. The real nature of software

Most programmers don't know what software is. In fact nobody really knows. This concept is difficult to grasp. Moreover the notion of software evolves with our understanding. This section fights with 4 wrong, yet widespread, ideas:

(1) software is about seconds, days, and weeks,
(2) software is just a set of algorithms,
(3) a program can be "finished", and
(4) languages are immutable.

2.1. Misconception #1: using wrong time scales

Programmers have a quite narrow vision of what is time. Programmers do not care about history. They are not used to think about years. They live in the present time. Programmers consider two kinds of time-scale:

- (1) ***short time-scales***. Expressed in terms of mili-seconds or seconds, they are used to think about *software execution*. Application users interacting with software applications use this kind of time-scale.

- (2) ***medium time-scales***. Expressed in terms of weeks or months, they are used by programmers when they speak about *software development*.

The "start-up" economic model just enforces the false impression that software does not have to last, and that software engineering is just about now and the near future.

Besides this narrow vision, history has shown that this belief was flawed. During the last century nobody really cared about software evolution. It occured to nobody in the 60's or 70's (of the last century) that the software products they developed would last until now. Hence the huge amount of money spent to correct the Y2K bug. This expensive anecdote just revealed that software was related with time in an unexpected way. "Real"-time software uses a milli-second time scale, but does this means that other time scales are "unreal"?

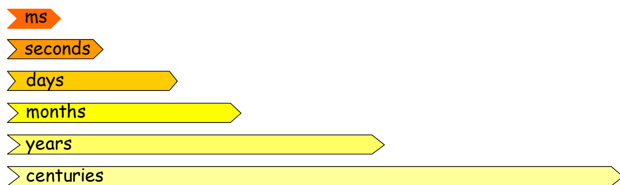


Figure 2 Which Time-scale to study software ?

Software evolution is (also) about decades, and centuries. Even millenniums, who knows? Five millenniums ago, in Mesopotamia, nobody even thought that the slow "invention" of writing would have such a radical impact on our modern societies [1]. Though unanticipated, this invention marked the shift from prehistory to history...

Nobody really understand what will be the long-term impact of the invention of software. Studying software evolution using large time-scales make sense however. In the 90's (of last century) some studies showed that the value of software on Earth was already exceeding the value of petroleum reserves. It is now reasonable to think that software will last longer than petroleum. One has to be prepared to imagine a world without petroleum. By contrast, it is quite hard to imagine a world without writing [1]. Similarly it will be harder and harder to imagine a world without software.

2.2. Misconception #2: software = algorithm

One of the first misunderstandings about software was to believe that software was only about algorithms. The distinction between programming-in-the-small and programming-in-the-large was not established until 1976, when DeRemer and Kron published their seminal paper. From then it was clear (at least for some people) that software was not only about lines of code, but should also consider higher level structures such as software architectures. In 1976 Boehm defined software as "*Computer programs and the associated documentation required to develop, operate and maintain them*". In other words software is much more than source code. Software also includes specifications, binaries, makefiles, and so on. This list clearly evolves as our understanding about the nature of software increases.

2.3. Misconception #3: "the program is done"

Another common mistake is to believe that a program could be "finished". As pointed out by Parnas in 1994 [3], the Software Engineering discipline will make a lot of progress when programmers and managers will achieve to understand that the first delivery is nothing but a first step in a continuous evolving process. Large *successful* software products last years and decades. Hence a large time scale is required, not only when considering the history of Software Engineering, but also to represent the evolution of industrial software products. Can someone assert that (in the future) no software products will last a century? Did software engineers anticipate in the 60's that programs would reach the next century?

In fact, when long-term evolution comes into play (that is when software is successful), the real nature of software has to be considered with great care; simply because every single aspect of software could evolve sooner or later. For instance the architecture/ code co-evolution phenomenon has been clearly identified by the research community. Architectural drift is considered worth studying (e.g. [4][5]).

2.4. Misconception #4: languages are immutable

While everybody agree that software architecture evolves, most academics still consider computer languages as if they were immutable artefacts. For instance most people consider "grammar evolution" and "grammar reverse engineering" as strange combination of words. They often argue that a grammar can be seen as a mathematical entity, and as such it is eternal.

Fortunately not everybody shares this vision. Grammar engineering is an important research theme as stressed by Lämmel and his colleagues [18]. In particular grammar evolution leads to many problems in industry [6][17]. Unfortunately *language evolution* and *language/program co-evolution* phenomenons are still neglected by the software engineering research community [7]. The reader is invited to search the web using "language evolution" as a keyword and look how many hits are related to software... Try also "grammar evolution" for instance.

This lack of interest might be partly due to another misconception about the nature of software. The concepts of software and program are by no means the same. A program is *absolutely meaningless* per se. A program is *absolutely useless* in the absence of tools to interpret it. A program must neither be disconnected from the language it is written in, nor from the language-based tools it relies on (e.g. interpreters and compilers) [18].

Too often, the implications of this very simple consideration are largely underestimated [7]. For instance nowadays most professional programmers take great care in archiving source code. However, by doing so they usually miss an important point. A few years after, source code will reveal of no use if nobody remembers with which (version of the) compiler these sources were successfully compiled. Moreover, there will be no intrinsic guarantee that (this version of) the tool will still continue to run on future platforms. It is not unusual in a software project to have legacy source but no way to transform these sources into binaries. Moreover, it could be difficult to maintain a legacy program if the legacy language it was written in was left undocumented, or if the documentation is not up to date.

The reader might still misunderstand the point, because he or she might equates "language" and "programming language". The latter is just a specific (yet important) case. Software is much more than programs. Computer languages are by no means restricted to "programming" languages. The next section show that software products are based on many other kinds of "languages" including schemas, domain specific languages, formats, and so on. Programming languages just represent the visible part of the iceberg.

2.5. The (short) History of the FooFoo Company

Let us consider as an illustration the FooFoo start-up company. This company was very proud of its brand-new "FooFoo" software product. When the company started business, nobody thought about calling this piece of software "FooFoo I". In fact, nobody realized that this software would have to evolve... This software used the EJB 1.7 standard enhanced by WeWeb1.3 framework that enforced an efficient "programming model". The building process was fully automated through XML files based on Ant 1.4.1. Engineers declared that using brand-new technologies and development methods allowed them to save 30% of development time. For instance, 20% of the code were generated from UML diagrams drawn with the powerful AcmeUML3.1 tool. This tool implemented a small subset of UML1.4 diagrams, but engineers used the Acme EJB profile v1.3. To produce the code the FooFoo company used various code generators from the BaBar company: BarUML2Java code generator v3.14, BarUML2Php3.4.8 code generator v1.12 for the server side, and BarUML2Java1.5 v1.15 which generated Java 1.5.1 code with generics. All tools were integrated thanks to the XMI 1.2.1 standard from the OMG. In practice this was done through a nice XMI wrapper 2.11 sold by BarCorp. Additionally the Foo company extensively used the FooL language of its own invention (no version there because nobody cared about keeping track of the evolution of this language). This Domain Specific Language (DSL) allowed FooFoo engineers to describe powerful business rules at a very high level of abstraction.

Guess what is the end of the story. After one year of successful business the FooFoo company never reached to deliver FooFoo II. Software engineers were excellent programmers. But they really misunderstood the real nature of software. For sure, they used a configuration management system to deal with *program evolution*, but they failed to deal with *software evolution*. They missed the point. They didn't consider *language evolution* as an actual issue. They didn't anticipate that the BaBar company could crash. They also simply forgot to keep the versions of the transformation engines. They drew many UML diagrams and produced many XML files but they didn't understand that they were not actually expressed in UML, but rather in an unsupported profile. They learned what are legacy models the hard way. In addition, the FooL DSL was produced by a bright programmer, who unfortunately left this language undocumented. He moved a few months after to another company... He was right, the FooFoo company was about to collapse, not because of bugs in programs, but because of the lack of recognition of language evolution issues...

2.6. Software = appliware + metaware

To avoid the kind of problems mentioned above, mature software organisations follow very strict quality insurance procedures. These procedures guarantee, when a software is released, that all pieces of languages and tools are attached to the programs. In this way every single bit of software can be reconstructed when required.

Summing up, the following informal equation holds: "software = program + language". A much more complete discussion of this topic can be found

in [7]. This paper modelled the software space a 3D space (+1 when evolution is added). Each dimension is structured as a pyramid. For the sake of simplicity, only the meta-dimension will be considered in the remainder of this paper. This dimension is about the various levels of languages.

The metapyramid, which is a foundation of the Model Driven Architecture (MDA) approach from the OMG [13], is represented in Figure 3. The architecture of this 4-layer pyramid raises a lot of questions [21], but in this paper, concentrating on levels M1 and M2 is enough. M0 and M3 can safely be ignored (we believe that most phenomenon identified in this paper also apply at these levels but this is another story). Figure 3 makes the distinction between metaware and appliware as introduced by[7].

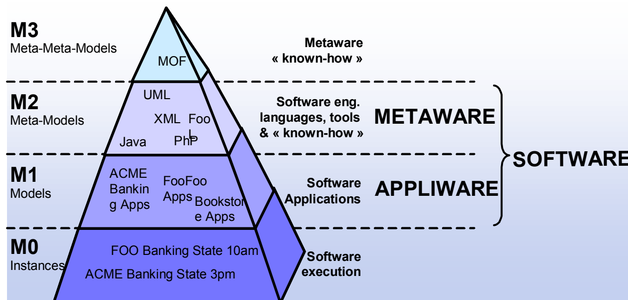


Figure 3 The meta pyramid [7][21]**.**
Software = Appliware + Metaware

Considering the "meta" dimension, software is decomposed in two distinct yet connected parts:

- **Metaware**. Simply put this is the "language" part of the software. Concrete metaware artefacts are at the M2 level in the metapyramid [21]. At this level one can found for instance the C or the Java languages, but also all concrete artefacts representing these (programming) languages. This include languages descriptions in the form of metamodels, grammars but also interpreters, compilers and other language-dependent tools. It should makes no doubt that this is software. Java parsers, java grammars and java metamodels fit at this level. These software artefacts are independent from particular software applications.
- **Appliware** (**M1**). This part of software is the traditional one. It is made of programs forming software applications. Usually only this part is considered. Appliware artefacts fit at the M1 level in the metapyramid (see Figure 3). This includes regular software artefacts such as application documents, specifications, models, programs and so one. For instance a banking application can be represented by many java programs that would fit at this level. These java programs are "conform" to the java grammar. The ConformsTo relation is one of the cornerstones of Model Driven Engineering [12][21].

Though this decomposition of software in two levels could seem strange at first sight it just reflects the very concrete fact that *a program is absolutely useless if it is disconnected from the language that allows its interpretation*. This distinction is consistent with the work of Lämmel and his colleagues, who introduced the notion of Grammarware and Grammar Engineering [18]. While Grammar Engineering is at the M2 level, Application Engineering is at the M1 level, that is where most programmers work. Though more research is required to define the clear boundaries of metaware and grammarware, we see metaware as a generalization of Grammarware. Grammarware could be seen as a particular technological space in the Model Driven Engineering terminology [12].

3. Models of Large time-scale Evolution

After having introduced the notion of metaware and its relationship with languages, let us review models of evolution based on large time scales (based on decades and centuries). This includes in particular

(1) models of the evolution of engineering disciplines,
(2) models of the evolution of Software Engineering,
(3) models of the process of technology maturation, and
(4) models of the evolution of computer languages.

Note that the last point, though important, cannot be really understood without considering the context in which computer languages where developed and evolved. Let us then start with a much more broader view.

3.1. Evolution of Engineering Disciplines

In [8], Shaw led an historical study showing that the emergence of engineering disciplines was the result of a very slow process. Such process can take centuries or millenniums. Shaw's model of evolution is decomposed in 5 steps as shown in Figure 4:

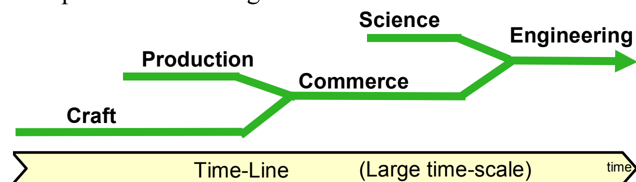


Figure 4 Emergence of Engineering Disciplines [8]

- (1) *Craft*. In this early stage, isolated craftsmen and gurus develop some artefacts by empirical means.
- (2) *Production*. This stage starts when the usefulness of the product is recognized among a larger community.
- (3) *Commerce*. It is usually a response to a crisis, such as product shortage, bad quality or wrong products. In this stage, methods and processes are established to enable the production of larger quantities of products.

- (4) *Science*. At some point, problems encountered in practice show the limit of existing theories. Academics are pushed to investigate the scientific nature of the product.
- (5) *Engineering*. The whole process results in the emergence of engineering. This step is reached when scientific knowledge is routinely applied by engineers to control the production of industrial-strength products with proven properties.

This model of evolution was applied by Shaw to various engineering disciplines. For instance, it was shown that the transition between the "Commerce" and "Engineering" phases took about two millenniums for Civil Engineering [8]. As an illustration, Romans were able to reproduce similar and robust bridges in many European places ("Commerce"), but the first bridge based on a full scientific model was built in 1850 ("Engineering"). Similarly, in Chemical Engineering the shift from "Commerce" to "Engineering" took about a century. In other words the emergence of engineering should be studied using a large time scale.

What about "Software Engineering" then? As it is well known, this term was coined in 1968 in a NATO conference organized as a reaction to the so-called "software crisis". Simply put, this conference mostly emphasized the importance of software processes through the concept of lifecycle. The use of the term "engineering" was clearly an abuse. In fact in the late 70's, the "commerce" stage just began. The focus was on the design of development methods based on accumulated experience rather than scientific foundations.

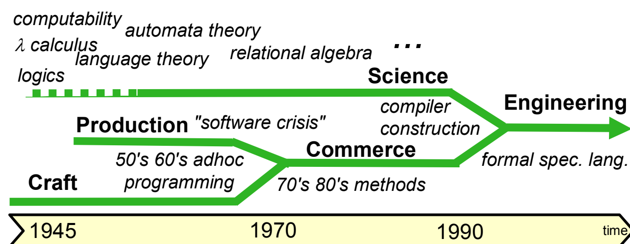


Figure 5 Emergence of "Software Engineering" (adapted from [8]**)**

As suggested in the above figure, Computer Science inherits from mathematics, logics, but also *linguistics*. The work of Saussure [2] in the end of the 19th century constituted one of the roots of modern linguistics. Then came the formal language theory and the classification of formal grammars by Chomsky in 1956 (note that Chomsky is also a linguist). Compiler construction is one of the most advanced computer technology. It directly comes from this heritage. This is indeed one of the particular examples that deserve the term "engineering" [8].

3.2. Evolution of Software Engineering

Shaw's model enables to compare the maturity of software technology with respect to other fields. It does not bring much information about Software Engineering itself. Raccoon produced various models of evolution to represent "50 years of progress in Software Engineering" [8]. The core idea of these models is that new techniques just replace old ones, over and over, leading to a series of "waves" that together form a "stream". In [9], the history of Software Engineering is modelled through 11 streams that flow concurrently. Only two streams are relevant in the context of this paper (see Figure 6 and Figure 7)

In a given "stream", each "wave" is decomposed in four successive stages: (1) *Innovation* where new ideas emerge and are tested, (2) *Growth*, when the technique becomes more popular than the previous one, (3) *Maturity*, when the limits of the technique become apparent, and (4) *Convention,* when the limitation of the technique is addressed by moving to another technique. Note that the vertical axis represent industrial interest, not academic ideas. The hardware stream is depicted below.

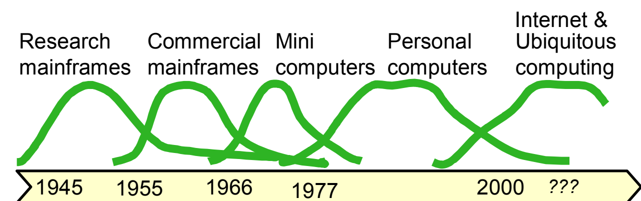


Figure 6 The Hardware Stream [9]

Note that, at the time of writing (in 1997) Raccoon was quite right in forecasting ubiquitous computing as the next important wave. The stream above is important in the context of this paper because the evolution of computer languages is largely influenced by hardware economics. For instance in the early 90's, Java (formerly called oak) was initially designed as simplified derivative of C++, for developing small, portable software for consumer electronics devices. Then, Java was released to the public in 1995 as a tailor-made language for Internet development. At the time of writing this paper, the HOLP database at the University of Murdoch references 582 computer languages for the Internet wave [11]. The evolution of hardware platforms influence not only the evolution of languages, but also the evolution of particular software products. This is true in particular for software products with lifetimes exceeding the duration of a hardware wave. Complete migration can be very difficult to achieve. On the contrary, it is quite common to wrap legacy software running on mainframes into applications with web interfaces. This is typically done through other computer languages, such as XML for interchange, Java or PhP for user interfaces, etc.

In parallel with the above stream, *software structuring paradigms* also evolve. Figure 7 shows this stream as

provided by Raccoon. The births of major programming languages have been added on the top of the figure as time references.

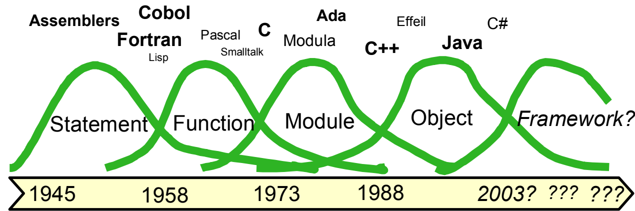


Figure 7 The Programming Paradigm Stream [9]

This figure clearly shows that, no matter the benefit brought by a paradigm, sooner or later, extensive use on larger projects and broader domains will invariably show its limits ("Maturity" step). Interestingly past waves had lasted between 10 to 15 years [9]. Note that, in 1997, Raccoon envisioned the end of the object wave around 2003. In 2002 there was a panel at the OOPSLA conference on the theme "Objects have failed".

What will be the next wave remains an open question. While "frameworks" was the bet of Raccoon in 1997, there are many other potential candidates including "components", "services", "aspects", etc. Instead of choosing one particular trend from these ones, the author of this paper would vote for "models" in the sense of Model Driven Engineering [12], especially since this approach, sometimes referred to as "Language Engineering", directly addresses the diversity of paradigms and languages. The motto "everything is an object" is being replaced by "everything is a model" [12][21][22].

3.3. Models of technology transfer processes

Raccoon's model is interesting because it shows that a large-time scale is necessary to understand the real nature of software. This is especially true for large software products because various waves will flow during their lifetimes. However, the reader might disagree when reading Figure 7. One might say "Hey, object orientation is quite old. It did start before 1988!". This is precisely what Figure 7 says. The "innovation" part of the wave starts in the 60's, with Simula in 1964, and Smalltalk in 1969. In other words, technology transfer is a quite slow process. Again a large time scale is necessary to understand software.

Raccoon's model fails however to model the interactions between basic research, applied research and industry. In [10], Raccoon's model and Shaw's model were merged. This led to the *helicoidal model*, in which software field was represented as a forest of helicoids centred on application domains. This space is regulated by attraction forces either centrifugal (leading to concreteness) or centripetal (leading to abstractness) [10]. The helicoidal model can be used both to explain interactions (1) between domains and (2) between research and industry. With respect to technology maturation processes, they should be measured using a decade-time scale [14]. The history of smalltalk is a good example with this respect. It took 1,8 decade to go from Kay's thesis to the first commercial version of smalltalk. Similarly Unix was born in early 70's, but its widespread use started only 3 decades after [14].

4. Models of Computer Languages Evolution

The previous section shows that the dynamics of software planet should be studied with a large time scale. Figure 7 models the evolution of major structuring paradigms (statement, function, module, object,...) but it does not give any idea of the complexity of programming languages evolution. Obviously many programming languages share the same paradigms though they differ in terms of syntax, applications domains, etc.

4.1. History of the computer language landscape

Figure 9 (see next page) is based on the HOPL database [11], featuring 7957 languages at the time of writing this paper. This graph shows the genealogy of computer languages. Nodes represent languages, colour (see on the web) the geographical origin of the language, and links "derived-from" relationships. Actually, this is a simplified view. As pointed by Pigott, nothing less than a 9-tuple can show the complexity of computer language inheritance.

By contrast, Lévénez focused his attention on 50 major programming languages and produced a human readable time-line [15] in the form of a 0,3 x 2m printing. The full diagram has been squeezed here to fit in the time scale below (Figure 8). Note that Lévénez' model starts with Fortran and does not take into account assemblers in the "Statement" stage.

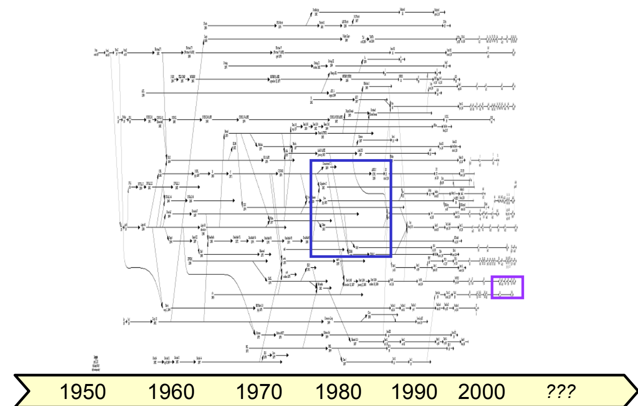


Figure 8 Evolution of 50 programming languages
(from [15], resized using a linear scale)

In the above figure each line stands for the history of a given language. Links between parallel lines showing derivation relationships.

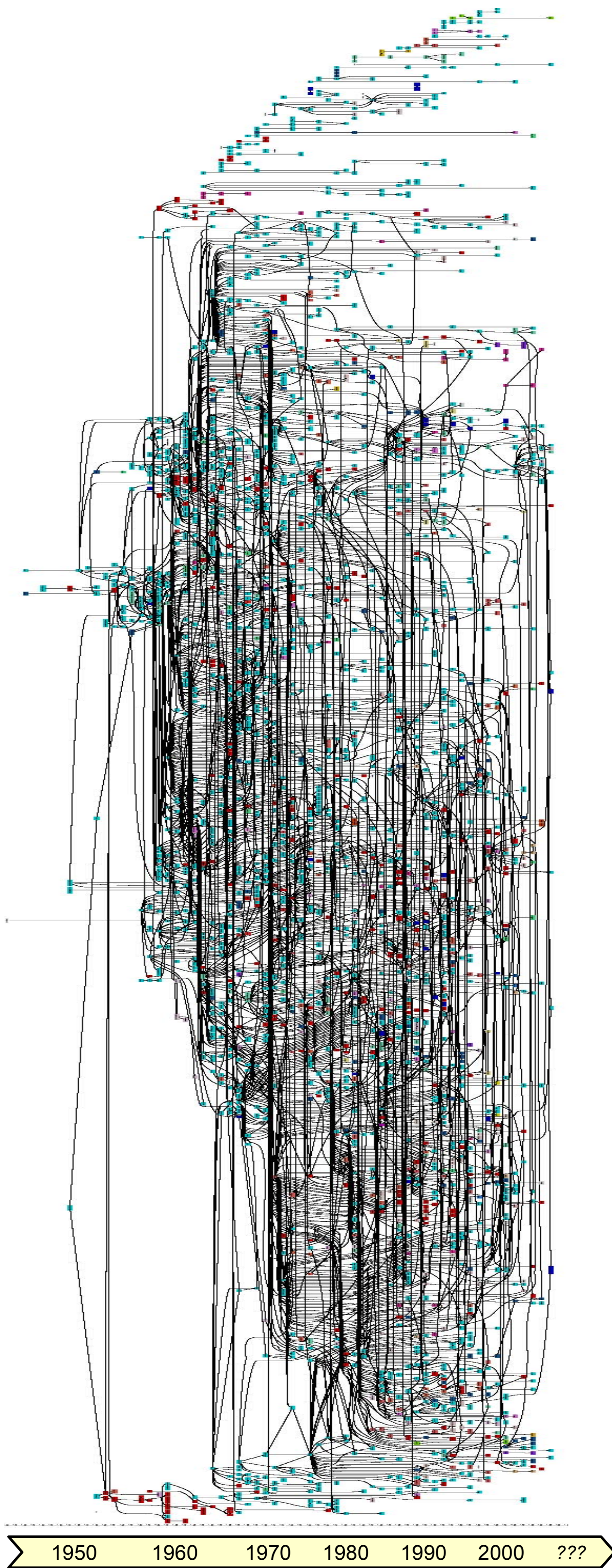


Figure 9 Genealogy of 7957 computer languages [11]

Figure 10 (see below) zooms on the birth of the C++ language and shows it relationships with other C derivatives. Once created, languages live their own lives.

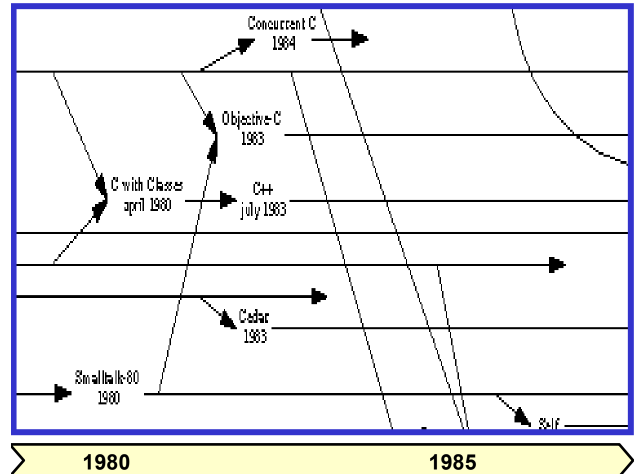


Figure 10 Derivation of languages (zoom)

4.2. Language versionning

As a first sight, there is no versionning information in Figure 10 (in fact "80" behind Smalltalk provide such a hint). Figure 11 zooms on another part showing clearly that since each language evolve by it own, it makes a lot of sense to treat it as regular evolving software. One again languages are software too. This means that languages are versionned too. The figure below zooms on the history of 4 languages from september 2003 to december 2004: Self, PhP, Perl and OCaml. It should be clear for programmers using PhP 4.3.5 that this language evolves! Interestingly this model shows on the top-right corner branch versionning to support concurrent development of the PhP language.

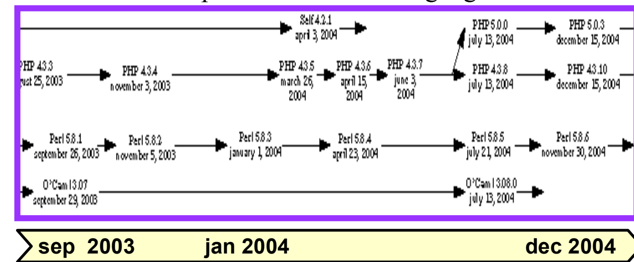


Figure 11 Languages and Versionning (Zoom)

4.3. Language evolution rates

PhP is born with the internet wave and is still rather young. It has celebrated its 10th anniversary this year. Just like other internet technologies this language is moving (too?) fast. Hopefully for its users it may reach sometimes a certain kind of maturity and stability.

What about older languages and elders? Fortan is the oldest but still robust and alive. Fortran 2003 was released in 2004 and it celebrated its 50th anniversary. Half a

century! Cobol is little younger. It is 45 year-old. As such it deserves respect (see Lämmel's "Don't bash Cobol" [17]) especially since a large part of software on the planet relies on this mature language. According to Gartner group 75% of all business data is processed in COBOL. Since (successful) languages lifetime are measured in terms of decades they are naturally subject to paradigm waves. Just like Ada, COBOL naturally incorporated Object Orientation. This led to OO COBOL in 1997 and COBOL 2002 ISO/ANSI. According to Gartner, in 2005 about 15% of all new applications on Earth will be developed in COBOL.

4.4. Language Families

Academics don't like COBOL. According to some of them Lisp is the greatest programming language on Earth [16]. Lisp was born just after COBOL. That's all they share. Academics enthusiasm for Lisp can in part be explained by Lisp' simplicity and versatility. Its very rudimentary syntax made it an excellent candidate for extension and variation, leading to easy language prototyping.

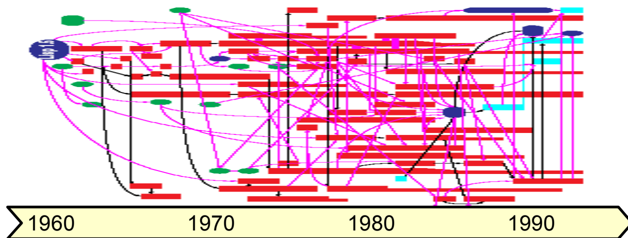


Figure 12 History of the Lisp Language Family [16]

In fact Lisp is a *language family* rather than a language. Again we see here that techniques usually applied to software applications also apply to languages. Figure 12 depicts the life of this big family as portrayed by Gabriel in [16]. This family is quite international. Some Lisp variants were born in America, a few others in Europe. Birthplaces are on the vertical dimensions. Note that not all variants are still alive. Lines show the lifefime of each variants.

5. A meta-model based history

The models presented above show that using a large time scale is necessary to understand the evolution of the software planet. However, these models are either very complex (e.g. Figure 9) or very simple (e.g. Figure 7). Metamodels offer a solution to push the analysis of evolution further. Simply put a metamodel is a model of a language [21]. Hence modelling the evolution of computer languages can be done by modelling the evolution of corresponding metamodels. This idea is briefly sketched in this section.

5.1. Zooomm: The International Zoo of Metamodels

Instead of collecting just language names and historical information such as birthplace, category, etc., the author of this paper has collected over the years a quite large collection of metamodels and metaware artefacts (level M2 of the metapyramid [7][21]).

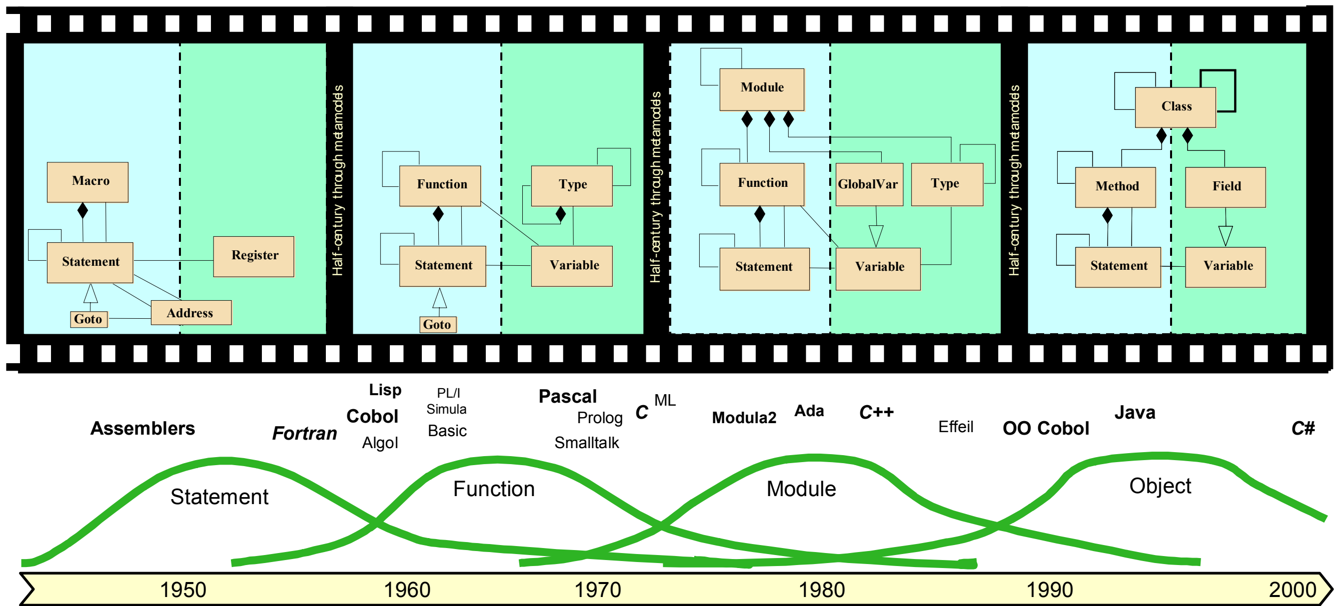
In 2005, this first collection led to the construction of the "International Zoo of Metamodels, Schemas, and Grammars". This zoo, referred as *Zooomm* (Zoo Of MetaModels), gathers the M2 artefacts from all over the world thanks to an international effort. Metamodels are too often considered as strange beasts, and the aims of this zoo is to show to the public that metamodels and other metaware artefacts are harmless, just the contrary. The notion of computer language is quite broad and many different kinds of metamodels live on the software engineering planet. At the time being the visitor could easily and safely visit the programming language zone were he or she will be able to observe metamodels of programming languages such as Cobol, Modula, C, Smalltalk, C++, JavaScript, etc.

Many metamodels in Zooomm are expressed as UML class diagrams defining the concepts of the languages. Zooomm also includes a grammar zone, since grammars are just other M2 artefacts. However we found grammars not suited for useful comparison. For instance comparing languages grammars only reveal syntactic proximity. By contrast, metamodels can be abstracted to go from collection of concrete metamodels representing particular programming languages to conceptual metamodels representing paradigms. This idea is used in the next section to show the evolution of programming languages.

5.2. Half-Century through Metamodels Movie

Each paradigm is based indeed on a small set of concepts. These concepts constitute the basic elements of the universe of discourse. They are naturally reflected in corresponding programming languages, although renamings, refinements and variations obviously take place. Nevertheless, thanks to metamodels we were able to build an historical model of the evolution programming language paradigms over the second half of the XXth century. Due to the lack of space, the method is only sketched here. From a concrete point of view the history is represented as a dynamic model in the form of a movie coined "Half century through metamodels".

Four snapshots, separated by a little more than one decade, are presented on the next page. This should be seen indeed as a film strip who suffered dramatic cuts. The full movie is made of much more images because the evolution of paradigms results from a slow incremental change process, rather than by a series of "revolutions".



The selected snapshots correspond to a particular state in each paradigm wave. As the reader might have noticed, the stability of the elements in the movie is rather surprising, only a few concepts are moving around.

The left part of each snapshot is devoted to algorithm while the right part is for data structure (Programming-in-the-small is about merging algorithms and data structures). Note that the early notion of address in assemblers (snapshot #1) was used both to store programs and data. A clear separation was then introduced between structured statements and data structures (see snapshot #2). Snapshot #3 shows the apparition of modules, which were considered at the beginning mostly as a way to structure program code. Then, the object wave in snapshot #4 re-unified both visions: a class is a consistent unit both for algorithms and data structures.

Obviously, this description provides a huge simplification of programming paradigms evolution. A more complete treatment would require much more space. We practised the movie show in classrooms using a blackboard a chalk and an eraser. This proved to be very good medium. Depending on the audience and the level of details selected the show can last from 15 minutes to 5 hours. Although 14 minutes is a quite drastic time compression, (up to 1 decade per 5 minutes), the power of metamodels enables to keep a very informative content. Obviously this assumes that students are fluent with basic metamodeling concepts and have knowledge about software evolution.

5.3. Language Evolution Patterns

During the show, emphasis can be put on recurrent *evolution patterns* such as the discovery of bad-smells, the definition of metrics, the design of analysis, restructuring and migration techniques, etc.

All these points can be illustrated on the metamodel basis. For instance in the "statement" wave goto statements were considered as bad-smells (see snapshot 1). Though this led to structured programming (see snapshot 2) elimination of goto statement was very progressive, and this bad smell have been quite stubborn. Similarly "global variables" were the bad smell of modular wave. This contributed to the move to the object wave which brought new fresh air, at least when the wave first arrived. This wave reached the "Maturity" phase a few years ago [9]. Object-oriented software products also suffer from bad smells [20].

6. Conclusion

Understanding the real nature of software requires to consider many different time-scales. While "real"-time software uses small time scale, large time scales are real as well!

Table 1: Software Time Scales

	Time scale
Processor time	milli second
User time	second
Programmer time	Days and weeks
Project manager time	Months and years
Software product life time	Decades
Language lifetime	Decades
Technology Maturation processes	Decades
Engineering disciplines	Centuries

Without considering the whole range of time scales it is almost impossible to have a global vision of what software evolution is about. It is time to recognize that languages evolve too, and that languages are software too. Whatever the paradigm chosen when a project start, this paradigm will be superseded by a better one during the lifetime of the project. This rule applies only to successful software products, the other ones will die before.

This paper just made it clear that computer languages evolve, just like other software. In fact, this is metaware evolution. We believe that it is time to start to study the metaware/appliware co-evolution phenomenon [7]. While scientists are not predictors, they can still study the history of software in the last half-century in order to exhibit recurring (co-)evolution patterns. Establishing new laws of software evolution is the basis on understanding what is software. Evolution is treated as a very serious topics in biology. Language evolution is an active branch of research in linguistics[1]. Why should this be different in computer science?

7. Acknowledgments

This work has been led in the context of the RELEASE European Network of Excellence on Software of Evolution. This network is funded by the European Science Foundation (ESF). We also would like to thank the various groups of students for their feedback about the "Half-century through Metamodels" shows performed at University of Grenoble and at the ENSIMAG. Most of the material in this paper has been elaborated over the last decade in the context of courses on (metamodel) software evolution. Special thanks go to Eric Lévénez and Diarmuid Pigott for the web resource they provided.

1. The logo on the left of figure 1 on the first page is the logo of a well established international conference on language evolution. Obviously language here refers to natural languages. Phenomenons such as evolution in time and space have been clearly established by linguists. There are more than 3500 languages in use on Earth, excluding dialects. Many computer sciences concepts including syntax and semantics come from linguistics. However it seems that when the theory of formal languages was designed, these mathematical objects had got a status of immutable and eternal objects. Nevertheless according to Saussure only a fool can believe that a language will not evolve [8]. In fifty years only two conferences, namely HOPL I, and HOPL II, were devoted in computer science to the history of computer languages. The call for papers for HOPL III has just been released. It will be held in 2007. These conferences however focus only on individual programming language evolution. They do not consider on the evolution of language as a general phenomenon. The logo on the right of figure I is the logo of the HOPL I conference. Further research includes the integration of result from linguistics.

8. References

[1] G. Jean, "Writing: The Story of Alphabets and Scripts", Harry N. Abrams Editor, Mars 1992

[2] F. de Saussure, "Cours de linguistique générale", posthumous publication, 1916

[3] D.L. Parnas, "Software Aging,", Proc of ICSE 1994

[4] A. van Deursen, "Software architecture recovery and modelling", ACM SIGAPP Applied Computing Review, Volume 10 Issue 1, April 2002

[5] R. Wuyts, "A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation", Ph.D. Thesis, Vrije Universiteit, 2001

[6] R. Lämmel, C. Verhoef, "Semi-automatic Grammar Recovery", Software Practice & Experience, Vol 31, No 15, December 2001

[7] J.M. Favre, "Meta-models and Models Co-Evolution in the 3D Software Space", Workshop on Evolution of Large Industrial Software Applications, ELISA 2003, paper available at www-adele.imag.fr/~jmfavre

[8] M. Shaw; "Prospects for an Engineering Discipline of Software", in IEEE Software, November 1990, pp. 15-24

[9] L.B.S. Raccoon, "Fifty Years of Progress in Software Engineering" ACM SIGSOFT Software Engineering Notes, Volume 22, Issue 1, January 1997

[10] J.M. Favre, "Une Approche pour la maintenance et la Ré-ingénierie globale des logiciels", PhD, University of Grenoble, 1996, http://www-adele.imag.fr/~jmfavre

[11] D. Pigott, "An interactive historical roster of computer languages", hopl.murdoch.edu.au, last visited: March 2005

[12] PlanetMDE, Model Driven Engineering on PlanetMDE, http://planetmde.org

[13] OMG, Model Driven Architecture, http://omg.org/mda

[14] S. Redwine, W.E. Riddle, "Software Technology Maturation", ICSE 1985

[15] E. Lévénez, "Computer Languages History", available at http://www.levenez.com/lang/

[16] G.L. Steele, R.P. Gabriel, "The Evolution of Lisp", ACM SIGPLAN Notices, vol. 28, no. 3 March 1993.

[17] R. Lammel, "Cobol as a Research Theme", available from http://homepages.cwi.nl/~ralf/

[18] P. Klint, R. Lämmel, C. Verhoef, "Towards an engineering discipline for Grammarware", www.cs.vu.nl/grammarware

[19] R.P. Gabriel. "The end of history and the last programming language". J. of Object Oriented Programming, 20(7), 2002

[20] M. Fowler, "Refactoring: Improving the Design of Existing Code", Addison-Wesley, June 1999

[21] J.M. Favre, "Foundations of the Meta-pyramids: Languages and Metamodels - Episode II, Story of Thotus the Baboon", Dagstuhl Seminar 04101 on "Language Engineering for Model-Driven Software Development". DROPS, available on-line, 2005

[22] J. Bézivin, "In the search of a Basic Principle for Model-Driven Engineering" Novatica Journal, March-April 2004

[23] J.M. Favre "Cacophony: Metamodel Driven Architecture Recovery", WCRE 2004

[24] Zooomm, "Zooomm: The International Zoo of Metamodels, Schemas, and Grammars", http://zooomm.org

[25] J.L. Dessalles, "Aux origines du langage - Une histoire naturelle de la parole", Hermes Editor, May 2000