A reflection on Software Engineering in HEP

Federico Carminati, on behalf of the ALICE Collaboration
CERN-PH, 1211 Geneve 23, Switzerland
E-mail: Federico.Carminati@cern.ch

Abstract. High Energy Physics (HEP) has been making very extensive usage of computers to achieve its research goals. Fairly large program suites have been developed, maintained and used over the years and it is fair to say that, overall, HEP has been successful in software development. Yet, HEP software development has not used classical Software Engineering techniques, which have been invented and refined to help the production of large programmes. In this paper we will review the development of HEP code with its strengths and weaknesses. Using several well-known HEP software projects as examples, we will try to demonstrate that our community has used a form of Software Engineering, albeit in an informal manner. The software development techniques employed in these projects are indeed very close in many aspects to the modern tendencies of Software Engineering itself, in particular the so-called “agile technologies”. The paper will conclude with an outlook on the future of software development in HEP.

1. Introduction
Modern HEP experiments have software suites consisting of $O(10^7)$ lines of code written in fairly powerful (and complex!) languages, such as C++, Java and Perl. In general, HEP has been quite successful at developing its own software. A quote attributed, probably apocryphally, to S. Ting, 1976 Physics Nobel Prize winner, is “no experiment ever failed for its software”.

In spite of its rather honourable record, HEP has never claimed to use, nor has it used on a large scale, traditional Software Engineering methods, which have been specifically conceived in order to facilitate the production of large programs. However it is our opinion that some intuitive Software Engineering was applied without ever being formalised. Failure to recognise this led to a complicated and not very efficient relationship with Software Engineering and software engineers, that still continues today.

This paper will provide some elements of explanation of how this could happen, and what are the reasons behind this apparent contradiction. A more detailed account of this matter can be found in a recently published book [1].

2. Traditional Software Engineering in HEP
Several generations of successful HEP experiments have produced complex software systems. These have been maintained over the years, adapting to changes in the detector, the replacement of the population of users and developers, and the evolution of the basic technologies of hardware, operating systems and data storage media. Non-professional programmers with no formal Software Engineering training have successfully done all this. Was this madness and pure beginners’ luck or is there method in it?
2.1. Traditional Software Engineering

Writing large programs has always been recognised as a problematic activity. Since fifty years [2] scientists are trying to address the fact that “Software projects finish late, they run over budget and their products are unreliable” [3]. This has led to the development of an entire discipline that is known as Software Engineering.

The fundamental predicate of Software Engineering is that solid engineering principles should be applied to the production of software in order to transform it into a more predictable and efficient activity. As one can see this is more an objective than a definition, which is moreover based on a questionable assumption on the existence and field of application of “solid engineering methods”.

Unfortunately it seems to be an unavoidable fact of life that any program, once it is tried out in reality, needs to be changed to conform to its intended usage and behaviour. Confronted with change, traditional Software Engineering tried to devise a strategy consistent with the supposed similarity between the software production activity and the engineering processes that, as we have seen, constitutes its founding analogy. Change had to be avoided or, at least, disciplined and controlled via a formal process.

In very broad terms, the strategy devised by classical Software Engineering to achieve its objective of introducing predictability and efficiency in software production, has been to plan exhaustively and in detail the software production process, and to divide it into a series of clearly defined and well-controlled steps, which could be planned in advance.

One condition for the success of this strategy is to minimise the ambiguity with which information is moved from one step to the next. Communication tends to be in the form of documents which try to be exhaustive and as detailed as possible in their attempt to cover all the aspects of each phase in the process in order to minimise the need of subsequent changes. This process tends to become quite burdensome and the “overhead” with respect to the software writing activity itself may be substantial. However this could pay off if the objective, i.e. the reduction of the occurrence of change, were attained.

Unfortunately such a process tends to create a situation where the cost of change grows faster than the likeliness of change is reduced. When so many formal steps are involved and so many documents have to be modified and kept consistent between each other, that even the smallest change may be the cause of substantial expense or delays. The net result is to make change slightly less likely but much more expensive with little or no gain.

To be fair, in some application domains there is a reasonable chance to capture the essence of the problem and to reach a full specification of the process with an effective reduction of late change. But these applications tend to be few and, as we will see later on, almost none are to be found in the domain of HEP research.

It can be said that classical Software Engineering in the form of High Ceremony Processes (HCP) did not yield the expected results. In general, the inadequacy of these methods is becoming more and more evident in the e-Business era, when change can no longer be considered an accident, but it has to be regarded as the norm. Software Engineering has its crisis too. In fact many university syllabi on Software Engineering start with the Software Engineering Crisis.

One of the major dangers of the application of HCP Software Engineering to a software project comes from its tendency to adapt reality to its method rather than the opposite. The trouble with HCP Software Engineering is that the elaboration of a plan is very expensive. If reality does not follow “The Plan”, the cost of the elaboration of an alternative strategy may simply be prohibitive. It is often the case that, at this point, the entire HCP construction collapses like a house of cards. All divisions of work, established roles, documents, plans, dictionaries and the like suddenly become useless and their remnants stay as a cumbersome legacy that hinders possible correcting actions. The failure of “The Plan” leaves the project with no plan and it then enters a no-man’s-land where time-scales and costs-to-completion become unpredictable.
The effects are the more serious the earlier this happens during the lifetime of the project. And in turn, the heavier and more rigid the project, the greater the likelihood that reality will not follow at an early stage. This has come to be commonly denoted by the sentence “too much process kills the process”.

One of the keys of the problem probably lies in the deceptive ease with which software can be written, rewritten and modified. In spite of this, the intrinsic time taken by writing a large piece of code is just a small part of the cost of bringing this code into production. Another problem lies in the fact that at the core of applying “solid engineering methods” to software development is the assumption that a personal creative activity, which often software development is, can be transferred into an industrial process.

At this point, however, the above account of classical Software Engineering’s status should be tempered with some of its positive aspects. Information Technology revolution at large has changed our world in a way that was not even imaginable only twenty or thirty years ago. No method of software development could have survived this fast-paced evolution without becoming obsolete. Even if we cannot point to a single Software Engineering method that can be said to capture the essence of software development or that proposes a strategy that works, at some level, in most cases, the collective reflection on software development operated by Software Engineering has been invaluable to understand this activity which is at the heart of the Information Technology revolution and it has certainly contributed to it in many ways.

The trouble is that Software Engineering has been offering interesting forays into a very complex field whereas the software industry needs effective strategies for software development. This complex field is at the heart of our society, constantly changing and evolving and still largely unknown; but, and for very good reasons, it has produced no practical recipe that would yield a predictable result.

2.2. Classical Software Engineering in HEP

The introduction of Software Engineering methods to the industry eventually attracted the research sector’s attention as well. At CERN, for instance, every new method and formalism that appeared over the years has been tried out. Scores of computer scientists have tried to apply these methods with no noticeable impact on HEP software at large, in spite of the time and resources invested. None of the results obtained in this way have come even close to the success and robustness of CERNLIB [4] (and in particular PAW [5] and GEANT 3 [6]), developed over the years with “amateurish” methods, or of ROOT [7], the spiritual heir of CERNLIB.

As we have seen, it does not come as a surprise that classical Software Engineering has not worked as advertised for HEP. It can hardly be claimed that it has worked anywhere as advertised. However what is perhaps surprising is the fact that the software projects where traditional Software Engineering has been adopted, or at least attempted, are amongst the less successful ones of the history of HEP software.

Our interpretation of this is that, over the years, HEP had developed a methodology for software development that was both adequate to its environment and reasonably effective. What these projects did was not to introduce some methodology in the development of HEP software, but rather to replace an existing methodology, which, as we will see, was never formalised or properly described, with one, well represented in the literature, but rather inappropriate for our environment.

3. THE HEP software development environment

The problems that software designers are called upon to solve in HEP have some specific features that differentiate them from those of an industrial environment. The purpose of industry is to produce goods, products and services and bring them to market. The technologies involved are, in most of the cases, mature and well understood. Marketing and selling implies knowledge of
the customer needs and the capacity of maintaining a quality standard for all goods that are sold. The output is measurable in terms of quantity and revenue. Late discovery of flaws is expensive and may seriously harm the image of the manufacturer.

In the research environment in general, and in HEP in particular, nearly all these points are overturned. Our general objective is to acquire knowledge. Not only is what we research still unknown, but means to achieve our objectives may not be available yet, and their acquisition may be part of the research activity itself. We are constantly required to innovate, in order to provide tools and methods that current technology cannot supply.

3.1. Software Engineering in HEP

The heart of any Software Engineering method is the attempt to control the impact and the cost of change. Surprisingly enough the idea that all software systems can be designed first and then implemented has seduced several people and the very fact that it is merely a mental model implying a gross simplification of reality seems to escape the majority.

In some sense HEP has always lived in a situation where change was unavoidable at any moment. As we have explained before, some requirements evolve and become clear only during the deployment of software and its use. New ideas are born, which could not have existed if physicists did not experiment with the software. The drive to buy the cheapest possible hardware and the necessity to use existing computing equipment in the collaborating institutions, as they join the experiment, impose to follow the continue diversification and evolution of the computers and operating systems. Once the detector is built, change comes from the need to experiment new ideas and to adapt to unforeseen experimental conditions and to the modifications and upgrades of the apparatus.

If change cannot be avoided, then the economy of change must be reconsidered. The axiom that late change is a disaster needs to be challenged and a methodology that does not allow the cost of change to increase exponentially with time needs to be implemented. If this approach is taken, then sometimes even deferring implementation, and thus creating the possibility of a future change, can be the winning strategy, as the time to implement a given feature may never come. In this case, starting with a minimal design and then evolving it may be much more effective than spending a large amount of time designing the system while the requirements are still unclear.

Over the years HEP has developed a self-styled method that, within the constraints and relative to its objectives, has been surprisingly successful. What computer scientists have had difficulties appreciating is that HEP is an environment where a well adapted method is applied instinctively but successfully to the projects. What is peculiar of HEP is that this methodology is transmitted via “oral” tradition and it has never been formalised. This situation is far from ideal, as knowledge must be abstracted and formalised in order to be properly communicated and to evolve.

HEP development strategy manifestly can cope with change. The Large Electron Positron (LEP)-era experiments started when the dominant computing platforms were mainframes, and the total computing needs estimated for LEP were less than the power of a modern laptop. Yet the LEP software has been ported and adapted over all the technology revolutions intervened over twenty years, and the user investment has been protected. HEP has taken advantage of every piece of inexpensive hardware that hit the market with no disruption in the production of results. The same code that was running twenty years ago on mainframes and on the first VAXes is now running on the latest CPUs with little modification. We believe it is important to analyse and understand the reasons for this undeniable success to see how these can be leveraged for the new generation of experiments.

The main feature of the HEP development method is the apparent absence of design. Indeed the most successful HEP projects started with a very precise initial idea guiding the development.
Developers and users “knew what they wanted” because they knew what they missed in the past and what they would have liked to have. This is not very precise but to any Software Engineer’s despair, it is as precise as you can make it without imposing useless constraints and without inventing details that most probably you will have to throw away, without wasting time. The objective or “plot” is clear, although sometimes difficult to express and formalise.

This initial plot, or metaphor, becomes common ownership of the community of physicist users, and guides the evolution of the system. The actual code is written usually quickly in accordance with the objectives of the project, and it is quickly delivered to the community of users, who are very active in testing it because it answers their immediate needs.

Once started in this way, the software naturally goes through development cycles. The most important features are the ones on which the developers work first, because they are the ones most in demand by the community. Somehow, in the best cases, it all happens in the right way because there is a common culture of the product, the moral of the initial plot, which is shared by the community. Users usually complain vocally, but constructively, and they insist on using the product. Developers implement the most wanted features. Advanced users provide pieces of code, following the style and the philosophy of the product, that most of the time can be just “slotted” in place and become part of the common code-base.

The community that feels collectively responsible drives the code development. Parallel developments often merge into a single product. When necessary, re-factoring or even massive redesign happens. New releases appear quite frequently, and the version under development is constantly available to guinea-pig users or to those who desperately need the new features.

Progress is guided by mutual respect, dialogue and understanding. Most of the communication happens via e-mail and it is not infrequent that some of the major developers meet only a few times in their life. Major redesigns are carried out with the responsibility to protect the investment of existing users, while paving the way for further evolution. This is not the description of a dreamy future. This has been the day-by-day story of the CERN Program Library since then. Other products, where the design-first method has been used, either have never seen the day (E. Pagiola’s GEM in the 70’s) or had difficulties entering production (e.g. GEANT 4 [8]).

In what follows we will see some of the most important aspects of the HEP software development cycle in greater detail.

3.2. The fine art of releasing

One of the most delicate points of the development of a software project in HEP is the process of delivering (releasing) new versions of a product (including the first one!) to the users. HEP has developed a rather sophisticated procedure for doing this, that in many ways is central to its own development process and that is instructive to examine in more depth.

The reason that the release process is of paramount importance in the HEP software development cycle is that, as we have seen before, requirements are not all and not completely defined at the beginning of the development. Therefore, as the understanding of the requirements becomes clearer and new requirements are added, new versions of the software, sometimes with substantial changes, have to be brought to the users. The acceptance of change in the development process which characterise the HEP development strategy, as opposed to the traditional HCP Software Engineering, implies that any product will go through several releases during its life-cycle.

The first release has special importance. Usually a product is developed when there is a need for it, therefore the user community has expectations and a genuine need to use it in its day-to-day work. Delaying the release too much may force the user community into using some alternative product and then becoming entrenched in it. Early release is important: there is a window of opportunity that should not be missed due to a late release. Also, as changes will
certainly be requested, early release when the code base is still limited will make substantial changes of direction easier. Late releases tend to make developers less receptive to requests for major changes in architecture and direction, and we have often seen such “original sins” having long-haul consequences on the life-cycle of the software product. There is a danger in early releases as well. If the product is too immature or offers too little functionality with respect to the intended use, it risks disappointing the users who therefore will try it only once.

Subsequent releases follow a similar logic with the distinctive feature of being rather frequent. “Release often, release early” is the punch line, and the necessity of this follows directly from the dynamics of the development that we have explained above. As requirements “trickle” in during the project development, frequent releases are necessary to make sure that the production version of the product satisfies a large fraction of users. Also, in the absence of a “master plan”, constant “user verification” of the changes is needed in order to keep corrections small and not disruptive.

Failure to follow this simple recipe has usually been the symptom of a malfunctioning project. Moreover, this is a self-catalysing process. The longer the release time, the larger the number of changes. In such a situation it quickly becomes impossible for the users to simply “replace” the working version with the new one, because the impact of a large number of changes cannot be easily estimated, and substantial backward incompatibilities may have crept into the system. This tends to require extra checking and tests, and may lead to the necessity to expose the system to the users in parallel with the current version for a long time. This makes administration heavier and user interaction more complicated. In a fixed-size developer team, the necessary resources are subtracted from development. Moreover, if the differences between the current and the new version are significant, few users will have the time or willingness to test the new version, with a reduction of valid feedback. This in turn slows down the rate at which the new version improves to meet user needs, which creates a vicious circle.

When this situation arises, there are essentially five major handles on which to act in order to break the stalemate and recover, already described by.

Money. Sometimes buying better and faster machines, rewarding developers with cool laptops or installing fancy junk food vending machine in the developers’ corridor might help. In most cases however the time derivative of the improvements that these measures yield is smaller (if positive!) than the degradation of the situation due to slipping release dates. Given the resource-constrained nature of most HEP experiments, depending essentially on public funding and usually chronically understaffed and underfunded, this option is not readily available to ailing HEP software projects.

Time. If the release date cannot be met, then delaying it is the most natural reaction. However very often it is the worst option too, because as time passes, new requirements arise which, because the release is delayed anyway, and because it would look very bad if they were not introduced by the time finally the code is released tend to be introduced. Unfortunately this delays the release even further and it is often the starting point of the vicious cycle. We have seen this happen very frequently in HEP, where even mission-critical products have seen substantial delay in their release. The peculiarity of HEP in this respect is that timing is governed by the experimental apparatus (accelerator and detector). By the time the RAW data start flowing, users will make use of whatever product is there, or will force the project to deliver the product. In the best case this will yield a working product with a reduced scope, in the worst, a partially working product which will address the user requirements only partially.

People. It is well known by now that adding people to a project which is late usually makes it later [9]. Nevertheless also this seems to be the most common corrective action taken when a project is in crisis. In HEP this action can be particularly damaging because of the financial and sociological constraints of most HEP experiments and project. The professionals that are most readily available for their “low cost” and because usually research institutions “by mandate”
encourage their employment, are young students and fellows. Irrespective of their quality, which is often very high, these people need to be trained and therefore for some time are more a hindrance than a help. The point is that the addition of people to a project requires an investment from the members of the project to integrate the new resources, and in time of crisis these resources may not be available to be mobilised.

**Quality.** It can be very tempting to control a project via the quality of the code delivered. However writing low quality code is only marginally faster than writing good code. On the other hand, debugging low quality code can be much more time and resource consuming. Of course aiming at the highest possible quality (whatever this means) can also be a reason for failure. The definition of the “right” quality for a given code, and of the means of implementing it is one of the hardest tasks for a project manager, and we will come back to this point later on.

**Scope.** This is perhaps the least used parameter in software projects, and this is probably a legacy of the Software Engineering HCPs where the scope of the software is deeply embedded into the original planning. When properly employed, this is a very effective handle to control the evolution of a software project. Whenever a feature risks delaying a release, it should be seriously considered whether to re-schedule it for a subsequent release. In fact even *anticipating* a release when in a difficult situation could be a good decision, as this allows developers to focus on the really problematic items while reducing user pressure. The establishment of an *a priori* release schedule, irrespective of requirements and development times, has often been proposed as an effective way of managing a software project.

As we have seen above, the release strategy is central to the success of a software project. Most of the distinctive features of the HEP release process are common to modern Software Engineering and in particular to the Open Source community. Here, once again, we are faced with one more case of “independent discovery”. Again, we can only regret that more self-consciousness of our own procedures was not present, which could have led to a constructive dialogue with modern Software Engineering experts and Open Source project managers.

### 3.3. Testing and documentation

It is common knowledge that the importance of testing and documentation cannot be over-emphasised in a software project. However both come at a price, and in a resource-strapped project these may well be in competition with the writing of the actual code and meeting the project deadlines. HEP software projects have developed specific strategies to cope with these problems.

Developing an exhaustive test suite represents a substantial amount of work, often comparable to the effort invested in writing the code itself. Moreover, the relation between the effort invested in a testing suite and the number of bugs that effectively “escape” it is not linear. While a relatively simple testing suite is invaluable at detecting a large percentage of the flaws of a new piece of code, the reward in terms of problems detected versus effort deployed in the improvement of the tests decreases rapidly.

In this HEP, the nimble release cycle effectively turns the user community into a large and exhaustive non-regression testing system. Given that the difference between releases is usually small, users can move quickly from one release to the next with their production code and therefore submit the newly released code to the most stringent test of them all, the full spectrum of real use cases. Of course this technique only works if some conditions are met.

The quality of the release must be indeed rather high. A new release that immediately breaks most user applications would simply block the development process. This means that there is a non-regression testing procedure that, however minimal, weeds out all the “trivial” bugs. Problems must be addressed quickly and professionally. The physics user community expects that programs may fail as a “fair” flip-side of seeing their requirements quickly addressed by the developers. However they also expect problems to be addressed quickly, either by workarounds or
by the issue of a patched release. The code of the program must be easily available for inspection. This had often the positive effect that expert users may report not only the problem, but also a thorough diagnosis of its causes and, in several cases, a proposal for fixing it, if not the directly the corrected code!

As a matter of fact, this can also help in writing the test suite itself. It is often the case that some (or all) of the code used to test the package comes directly from the users. Usually the code is chosen so that it provides exhaustive checks and can be run quickly. Because of the general Open Source policy of the user community, there is usually a large choice for this kind of code which makes very good test cases.

Documentation poses another problem, which is different in nature but not less serious. It is clear from what has been said that writing the documentation beforehand is not an option for a project where requirements, and hence features and functionality constantly evolve.

The lack of documentation of HEP codes is often taken as a fact, however a closer examination of the major programs shows that this is not necessarily the case. ROOT and so on all have extensive and well-written user manuals. Moreover the code of packages is usually rather well commented in-line. The fact is that this documentation is usually created only when the program reaches a certain maturity and therefore stability.

3.4. The people
The scientists participating in this project work for the different laboratories and universities which collaborate at the construction and exploitation of the detector. The important point to appreciate here is that the different groups are geographically spread in different institutes, countries and continents. Modern HEP experiments are composed of one or two thousand physicists coming from one hundred institutions located in tens of countries in different continents. The “project” relationship is rather informal, the different institutes being bound by a Memorandum of Understanding that cannot be legally enforced. An experimental collaboration itself has only a moral but not a legal status. The real link between the physicist participating to an experiment is the common goal of building and operating a detector for fundamental research.

In this situation the management of the Offline project has to proceed by consensus, and all decisions and agreements have essentially only a moral value. Not only most of the people participating to the Offline project from different institutes have no hierarchical relationship between them, but the project itself is defined and mandated by an entity (the experiment) that has no legal existence. This introduces a vast heterogeneity of relationships among individuals.

3.5. Why does it work at all?
We believe that HEP software development works out because there is a method behind it, well adapted to the environment. As we said the method is not formalised, and therefore it suffers from all the limitations of a tradition compared to a methodology. A traditional computer scientist may claim that there is no design. We maintain that design is done continuously, even if implicitly. The open discussion with all the users is the design activity that influences the evolution of the product. The new code is put into production very quickly: most of the time the users have direct access to the development repository. This in turn means that testing is continuous. The functionality of the product is defined, again implicitly but exhaustively, by the use made by the user community.

A common criticism is that the direction of the project is not clear. In reality the direction of the project is continuously redefined on the basis of the customer feedback: the project “goes where it needs to go”, independently from, and most of the time beyond, what was decided at the beginning.
Critics claim that the code developed in this way is of low quality, difficult to maintain and to modify, not modular etc. The quality of the code developed in HEP varies largely. However most of the HEP products have served their purpose and a few products have been used by generations of experiments over tens of years and vastly changing technologies.

The downside of all this is the lack of formalisation of the process. Given the absence of a description and of manuals or other teaching materials on the software development practise, it takes time to train programmers in the HEP style of work, and people who do not have a natural inclination for it have bigger difficulties. The best developers are those who have a feeling for the needs of the user community, and there seems to be no way to learn or transmit this without finding an equally talented disciple.

A formalisation of this process would reduce these drawbacks. A clear, and possibly consensual, description of the best practises of HEP software development in the language and formalism of Software Engineering could help young researchers to become active more quickly and with less coaching. In spite of all evident advantages, this step has not been taken formally. We believe that this happened mainly because software developers in HEP were themselves unaware to be doing Software Engineering, and also because such an activity would have required an investment of resources, particularly from the main actors of these developments, and resources are notoriously in short supply in HEP software projects.

4. Modern software-engineering trends
It may now be instructive to give a quick overview of the modern Software Engineering trends to compare them with what has been said above about HEP software developments. As we have seen, HEP software developers are able to produce results under the following conditions:

- Short time-to-market.
- Changing requirements.
- Changing user community.
- Changing hardware and Operating System base.
- Changing developer teams.

These features have always been characteristic of HEP, but they are now also becoming commonplace in the Information Technology industry, particularly in the area of e-Business. These are the features that modern Software Engineering tries to capture and satisfy.

Modern Software Engineering is trying to respond to the relative inadequacy of classical methods and to the rapid evolution of software development conditions. The response to High Ceremony Processes are the so-called “Agile Methodologies” [10]. These methodologies tend to adapt to change rather than predict it, and they focus on people interaction rather than on process definition. The whole process is kept as simple as possible, in order to react quickly to change. This allows the introduction of an incremental and interactive development based on short iterations of the order of weeks, in constant dialogue with the customers and users. The work is steered by coding and testing rather than by analysis and design. These technologies are uncovering better ways of developing software by valuing:

- Individual and interactions, more than processes and tools.
- Working software more than huge documentation.
- Customer collaboration more than contract negotiation.
- Responding to change over following a plan.
The value of the last items is not ignored, but the former are chosen every time there is a conflict. In analogy with High Ceremony Processes, agile technologies are also known as Low Ceremony Processes.

These new development methods, involving Low Ceremony processes, are becoming commonplace in modern Software Engineering. One of the most successful, eXtreme Programming (XP) [11], has been recently described and represents one of the first Software Engineering methodologies that tries to reduce the cost of change, reducing initial design and centring the method on the process of change itself. XP is built around four simple principles:

(i) Communication. A project needs continuous communication, with the customer and among developers. Design and code must be understandable and up to date.

(ii) Simplicity. Do the simplest thing that can possibly work. Later, a simple design will be easily extended.

(iii) Feedback. Take advantage of continuous feedback from customers on a working system, incrementally developed. Programming is test-based.

(iv) Courage. The result of the other three values is that development can be aggressive. Every time a possible improvement in the system is detected, the system can be re-factored mercilessly.

XP is based on small, very interacting teams of people working in pairs. Testing is practised from the start, and every piece of code has its frequently run test suite. System integration is performed daily under continuous test. The development is use-case driven, with specific techniques to estimate time and cost of the project. Programs are continuously re-factored to follow the evolution of user requirements. Written documentation besides code is kept to a minimum until the program is released. The main achievement of XP is perhaps to have recast the traditional paradigm of Software Engineering (waterfall and spiral) into a much shorter timescale that is better adapted to present day software production in industry, which is very close to the normal practise in research.

Here again the similarities with HEP are very strong, even if the conditions in which XP has been developed are somewhat different from the environment of a typical HEP experiment. Evolving design, accent on most wanted features first, very short time-to-market, general availability of unreleased code for testing, i.e. constant Quality Assurance verification by the user community are all XP features that are commonplace in HEP development. There is however an important difference: HEP systems are built by distributed teams and not by small concentrated teams of programmers working in pairs as is prescribed by XP. However this needs not be a major obstacle. As with classical Software Engineering, agile methods are also models that need to be adapted to the specific software development situation. In this case the adaptation of agile technologies to HEP could give new insights into both fields.

It is very encouraging that modern Software Engineering trends seem to explain and justify what we have been doing for so many years, often feeling guilty about it. We finally start to understand why things worked in spite of the curses of generations of software pundits. This understanding, if properly exploited, may help us in doing things better, formalising and improving our process.

Now that a new culture develops and agile methodologies begin to be recognised as a perfectly honourable way of working, we are feel some pride of having been precursors on this way, when the road was still full of obstacles.

5. Conclusions
HEP has developed and successfully deployed software for many years without using classical Software Engineering. Market conditions are now more similar to the HEP environment, and
modern Software Engineering is confirming some HEP traditions and rituals that have proven successful even though they were never codified or otherwise justified.

These developments may be important for HEP software as they may allow us to formalise our software development strategy. This could improve our planning capability for distributed software development and it may help to transmit and evolve our software practises, reducing the lead-time for developers to be productive in our field. This is particularly important now that it is difficult to find long-term positions in research, and most of the developments are due to young scientists who, after a stay in research, will look for further employment in the industry.

Our environment adds complexity to the one foreseen by agile methods, as we work in large and distributed teams, with little or no hierarchy. Therefore the challenge is now to move agile technologies into the realm of distributed development. The authors believe that this could be a worthy goal for Software Engineers working in HEP, and an occasion to collaborate more closely with advanced Computer Science and Industry.

References
[1] R. Brun, F. Carminati, G. Galli Carminati, (Eds.), From the Web to the Grid and Beyond, Computing Paradigms Driven by High-Energy Physics The Frontiers Collection, Springer, 2012
[2] In Proc. Symp. On Advanced Computer Programs for Digital Computers, sponsored by ONR. Republished in Annals of the History of Computing, Oct. 1983, pp. 350-361. Reprinted at ICSE’87, Monterey, California, USA, March 30-April 2, 1987, June 1956.
[3] In P.Naur and B.Randell, editors, Software Engineering: Reports of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 October, 1968.
[4] http://cernlib.web.cern.ch/cernlib.
[5] http://paw.web.cern.ch/paw.
[6] http://wwwasdoc.web.cern.ch/wwwasdoc/geantold/.
[7] http://root.cern.ch.
[8] http://geant4.web.cern.ch/geant4.
[9] F.Brooks, The Mythical Man-Month, Addison-Wesley 1975, 1995, ISBN 0-201-83595-9 (1995 ed.)
[10] http://agilemanifesto.org.
[11] Kent Beck. Extreme Programming Explained. Addison-Wesley, Boston, 2000.