CASSANDRA CROSSINGS

Bureau indépendant d'analyses et d'études

Will you take over a little supervision?

A technical-social study

A, approx. December 2025

Our professional experience has given us the opportunity to observe closely the circumstances of IT software production, from the initial design to sustainability related problems.

Before proposing solutions, we will review the IT development processes.

++++

When the wise man points to the moon, the fool looks at his finger...

In this story, the wise man is the customer !

He makes a wish: he asks the fool to fetch him the moon. And the moon, which he insistently points to, is software that corresponds to his expectations and needs.

The client needs an application that can perform certain tasks. The client draws up an inventory of the current situation and based on these findings, prepares specifications and requirements and submits them to the supplier.

Both agree on their respective expectations and obligations. The client undertakes to provide the developer with all the elements that the developer requests, and the IT engineer undertakes to build an application that takes into account the information provided.

Upon completion of the work, the Supplier shall make the thus created product available to the Customer and the Customer shall pay the Supplier the agreed sum.

The fool is the IT world.

Faced with client's demand, the IT world undertakes to provide quality work that respects client expectations. To do this, it will follow a procedure, a framework that has been validated by the experience of its corporation. The structure chosen is supposed to act as a safeguard to prevent possible misuses.

The IT world is no different from any other. When the need for supervision of the work arose, the IT world looked at the solutions of other professional fields and built its own solution based on the analogies that seemed most relevant at the time when these considerations were taking place.

We are going to review these solutions the IT world has implemented in recent decades and we are going to consider whether they have kept their promises, that is, whether they have helped the fool to provide the wise man with what the wise man asked for...

Situation during the 1960s

By the middle of the decade, a tipping point had been reached: the price of software exceeded the price of hardware, and the gap between them would never stop widening.

In 1968, the first **NATO Conference on Software Engineering** drew up a poor picture:

Programs were delivered late and budgets weren't kept.

But these inconveniences could have been accepted if the software had not had more serious flaws: the applications did not match what was requested. Their use was too complex and bugs were common.

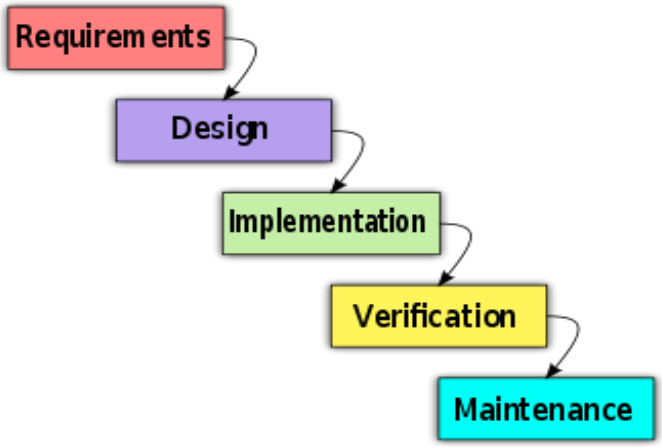
Icing on the cake (if you dare put it that way): the internal structure of the software was anarchic, maintenance was very difficult, debugging was almost impossible, therefore, scalability couldn't be guaranteed at low cost.

From that moment on, the decrease in tolerance for design and programming errors will lead to an awareness of their financial and organisational consequences. The growing pressure for better software quality will lead to the emergence of many more or less effective responses.

Situation during the 1970s

The response to these justified criticisms was the creation of a framework to structure the development carried out by the IT world.

The first structure implemented was the Waterfall method, based on methods used in civil engineering.



Each step must proceed in a logical sequence: first the client is asked to specify his expectations, then the plans are drawn, then the foundations are built, then the walls are built, then the roof is built. Once a step has been completed, it is considered completed. Hence the name 'Waterfall: once you have crossed a step, you'll never go back.

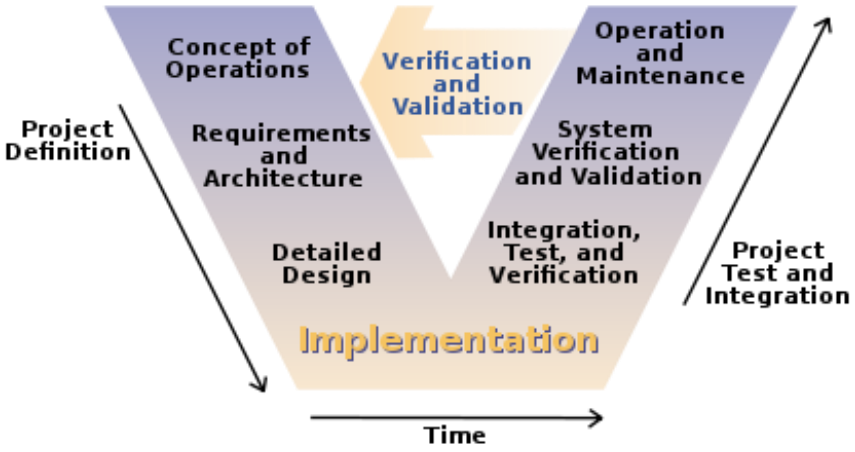
In addition to this work designed to respond more effectively to development's constraints, recommendations have emerged within the IT community: the addition of comments to the code to make it easier to understand, the versioning of the work to make it easier to return to a previous stable situation, the definition of explicit and well-documented test procedures to ensure reproducible supervision of the progress of the work. These recommendations are still valid.

Situation during the 1980s

The cascading model was very rigid in terms of time and organisation. Each step had to be completed before the next could start, at least in theory. Consequently, any subsequent challenge to a decision taken in a previous step was very difficult.

Checks carried out at the very end of the process could reveal problems that were very difficult and costly to address. Each correction could also mean that a greater or lesser amount of work was lost.

The so-called 'V' model, born from the evolution of the Waterfall model, made it possible to better take these constraints into account.



The downward branch of the V represents the design and programming that starts from the abstraction of general concepts and goes down into the realisation of the distinct elements.

The ascending branch represents those tests whose level of generalisation can be paralleled with the corresponding item in the descending branch. The separate items that are programmed as one reaches the bottom of the descending branch correspond to the unit tests of those same items. If an error occurs at this point, the feedback loop to correct this item is very short, that is, very economical.

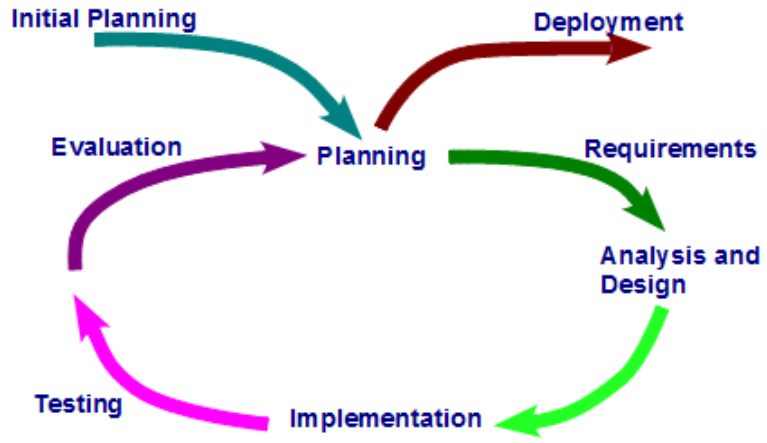
The next step in the uplink is module validation. If a module fails, the feedback loop will be longer because it will involve the correction of one or more items in the module concerned. In practice, the further the work progresses along the uplink, the lower the probability of a mistake requiring corrective feedback. But the greater the impact of any correction.

Situation during the 1990s

The development of increasingly complex software incorporating many modules has revealed a major flaw of Model V: modifying a module is certainly costly, but the problem is significantly aggravated when the correction of one module leads to a cascade of corrections in other modules of the application.

In such a situation, the increase in costs can become literally explosive. Moreover, it is only at the last stage that the client can finally assess whether the work meets his expectations.

An evolution of the V model led to the creation of the incremental spiral model. The family of current iterative processes such as Rapid Application Development, Scrum or Extreme Programing emerged from this incremental model. Most of them are grouped under the umbrella of the Agile Manifesto.



The Agile Manifesto

Agile thinking encompasses several methods that recognise common values and practices. These methods claim a pragmatic approach that strongly involves the client throughout the development process.

Agile Values

The 4 core values are:

- **The team**: according to Agile, the people who make up the project team are more important than the tools used and teamwork is strongly encouraged.
- **The app**: The correct functioning of the app is vital. It is better to have an app that works than a complete documentation.
- **Collaboration**: The client is involved in the development. They must collaborate with the team and provide explicit feedback on the suitability of the product for their needs.
- **Acceptance of change**: Product planning and structure must be flexible to allow the product to adapt to changing customer demands.

Agile Principles

These 4 values are broken down into 12 principles common to all Agile methods.

The page https://agilemanifesto.org/iso/en/principles.html describes :

We follow these principles:

I. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
II. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
III. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
IV. Business people and developers must work together daily throughout the project.
V. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
VI. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
VII. Working software is the primary measure of progress.
VIII. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
IX. Continuous attention to technical excellence and good design enhances agility.
X. Simplicity--the art of maximizing the amount of work not done--is essential.
XI. The best architectures, requirements, and designs emerge from self-organizing teams.
XII. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

All these values, all these principles are supposed to guarantee the delivery of a product that meets the wishes of the customer, since the customer was involved from start to finish, and partial deliveries and controls at each stage were supposed to ensure that the application was fine-tuned with the customer's expectations and needs.

Under these conditions, how can we explain such resounding failures as the fiasco of the software environment of the F-35 US military aircraft?

How does the Swiss Confederation explain the abandonment of programs that have swallowed up hundreds of millions of Swiss francs? (FTA INSIEME, 150 mio CHF or DDPS FIS, 700 mio CHF)

Or the ongoing sinking of the SECO (Swiss State Secretariat for Economic Affairs) tailor-made jobless management app requiring more than 6 months of corrective patches ?

Our knowledge of this world has taught us that the laws of IT development have stopped evolving after the implementation of Agile iterative environments.

In his almost 300-year-old work, Charles Louis de Secondat, Baron de Montesquieu, already taught us that '*If a law is bad, it must be changed.*'

What is the IT world waiting for to change its laws?

We will answer this very legitimate question in the following paragraphs and our answer will be as explosive as the increase in the costs of the offending IT sofware.

We will highlight the reasons for not improving the quality of IT applications, we will explain why budgets can only be exceeded many times. We will show why we are falling back into an IT crisis on a scale greater than that of the original crisis of 1968, but we will also give the keys to the remedies to be applied.

Crisis? What Crisis?

Understanding the crisis

(Note: this analysis dates from 2007, we repeat it unchanged since the situation that justified its writing has not fundamentally changed.)

Almost 40 years after the NATO Conference, the techniques used to develop software have failed to answer these questions, and the increasing complexity of large systems has compounded the problem.

Comparison of models

We analysed these models and determined how they responded to vulnerabilities known as "software crisis."

	Overall management	Testing	RETEX (feedback)	Requirements	Budget
Waterfall	OK	-	-	-	Failed
V-model	OK	+/-	-	-	Failed
Iterative/agile	OK	OK	+/-	+/-	Failed

The **Waterfall** model adds global control over the project lifecycle but does not solve other problems, such as the fact that testing at the end of the process only detects errors when there is insufficient budget to correct them.

Model V also meets the global control and provides an effective test procedure for small parts of the software (unit tests) but does not solve the problem of defects in an entire module. With Model V, it is possible to detect errors and correct them within budget, provided they are small in size and have little impact, which is why the test is rated (+/-). When errors are larger, budget control fails.

The **iterative model** family has a fairly good overall control and accurately tests the written code. User feedback is injected into subsequent iterations to correct defects. This feedback, which is a major asset of the iterative model family, allows for varying degrees of compliance with the user's requirements, meaning that in order to meet the requirements accurately, it may be necessary to do more iterations than expected, which can lead to budget overruns. This is why the requirements are rated (+/-) and the budget fails.

So much for the theory.

In practice, the Waterfall and V models have such glaring limitations that they have been abandoned in favour of iterative models. But even these models that are supposed to correct all the flaws fail miserably, sometimes leading to the project being abandoned altogether.

The failure of iterative models explained

The failure of the iterative models is due to 2 unrelated causes.

The first factor is a scale effect: all the members of the group that drafted the Agile Manifesto were very high-level experts, each in his field, they were used to working with small groups of followers themselves of very high competence.

This dual selection (small group and high-level experts) had a particularly perverse effect: all the recommendations made by Manifesto members could only work within working groups with the same characteristics, i.e. small groups of experts.

Unfortunately, for customers who bought applications born under the umbrella of the Agile Manifesto, their products were built by ordinary people working inside large structures. Scaling up could only lead to utter failure because none of the conditions for creating the model were present.

To be applicable to large teams of normally competent people, the Agile Manifesto would have had to be written by ordinary people in large teams. They would have lined up banalities and no one would have paid any attention to them.

The writers of the Manifesto were listened to because they were recognised and respected gurus, and it was precisely this status of gurus amongst gurus that led to the failure of the model.

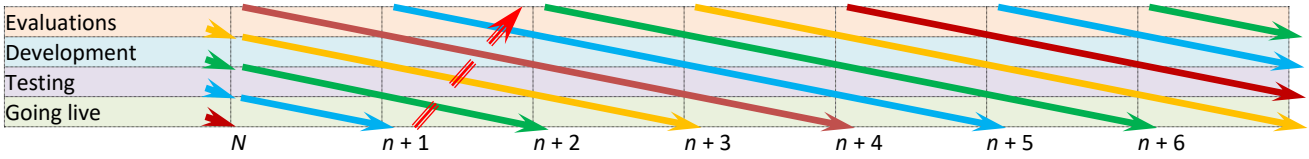
The other reason why iterative models fail to deliver applications of sufficient quality is the change in the financial paradigm.

With the Waterfall and V models, the customer always received a fully functional application, developed and tested as a finished product. This gave customers the choice of whether to purchase or not to purchase a new version depending on the new features built into the application.

For the purchaser, this meant a lot of freedom of action and substantial savings. For the manufacturer, every new release was a risky challenge. Would the new features reach an audience that wanted them? Or did the manufacturer miss a new niche that appeared too suddenly to be anticipated and that would make the product obsolete overnight?

Iterative development has completely changed the IT world.

From now on, the product would be in perpetual development: the concept of a stable version has disappeared, replaced by the sprint that was supposed to be the panacea since it made it possible to quickly integrate requests from customers.



In iteration models, the 4 major steps of software development take place in parallel. As soon as a team has completed the work for a given step, it sends its draft to the next team for the next step and begins the work for step ***n* + 1** in the draft given to it by the previous team. This is literally a development on a treadmill.
When a problem occurs; for example, if the user finds a bug in the productive application, that bug is passed to the analytics team who will have an iteration to review the bug and make a fix, which fix will be passed to the developers for integration in the next step, which fix will be tested one iteration later, etc. until the application is delivered to the client for production. Assuming one iteration per month, it will take 5-6 months for the bug to be fixed.

When a new iteration is released on the market, it is already followed by three successors already in development, each at a different stage. Feedback from a version *n* will therefore be incorporated into version *n* + 4 or 5. Similarly, any bug fix will have to wait a similar time.

As we have already described in our study on a **Comparison on Cultures and Languages**, we are also faced with a marked discrepancy between the behaviour claimed in commercial discourse and the behaviour actually expressed in facts.

The commercial rhetoric is that the vendor works for the good of the client. In fact, not only does the vendor have no incentive to produce quality, but in fact, everything is geared towards producing defective applications.

There is no black magic behind this assertion. There is the simple observation that, in order to keep the customer trapped in his subscription, one has to make him thirsty to force him to continue to pay.

Assuming that (to its greatest misfortune) a vendor releases a perfectly debugged iteration containing all the features the customer wants, it runs the risk that a larger or smaller proportion of its customers will tell it that they want to cancel the subscription and request a sales offer for this version, wishing to stay with this version for some years to come.

Such a situation would be tragic for this supplier: instead of a cow to be milked with the regularity of a clock, he would find himself faced with a sudden, very large cash inflow, but no follow-up.

And if he doesn't find new customers quickly to replace those who have just left, he risks going out of business. In conclusion, a good app is a nightmare !

As a result, iterative models are the perfect answer to this economic nightmare. They promise an indestructible bond between supplier and customer. Unfortunately, the medal without setbacks has yet to be invented; unfortunately, iterative models have turned the customer into a suckling cow to the delight of the supplier, who no longer needs to bother producing quality.

This is a challenge because it does not reflect the reality on the ground: the need for IT applications is so vast and growing so fast that IT companies will never run out of customers. It would therefore be quite possible to produce high-quality applications without fear of economic starvation.

In fact, all indications are that the opposite could happen, i.e. a vendor with the highest quality would most likely be overwhelmed by development demands from customers who want to equip themselves with flawless applications.

In conclusion, the focus on development models since the 1968 NATO Conference has been lost, as the issue of development models was much more linked to the economic context of IT development than to achieving better quality.

Update 2025

We were recently invited to participate in a team's periodic review of a Software department and we were able to appreciate the changes in management that have taken place since we first entered this world in the last millennium.

The most surprising element was that everything was about metrics, progress charts and success percentages. Where is the place for the human being, the user-customer in this litany of self-congratulation?

Of course, this endless litany of testimonials to the achievement of objectives flatters this little world, but this hyperactive narcissism conceals a detachment from everything to do with the user's experience.

In fact, all these presentations were only intended to show how each team was aligned with the steps presented during the previous quarterly session and how the results of the day served as a stepping stone for the objectives of the next quarter.

All these management methods have hyperbolic names that betray their transatlantic origin, and all the communications are peppered with jargon that serves primarily to separate the insider from the outsider...

But the most disturbing thing is the predominance of the two-stage US way of thinking, whose limitations and dangers we have demonstrated in our **Social Retroengineering Study**.

A manager proudly described the responsiveness of his team, praising its ability to correct errors reported within a very short time frame.

A table is worth a thousand words

#	**Agile approach**	**Approach "*à la française*[1]"** (French style)
1 – 1	An error is detected in Production	The end user is not involved in the ergonomic search for solutions (in contravention of the precepts of French ergonomics, which are very different from the US precepts and which strongly require this participation)
2 – 2	The error is corrected within a few days	The activity's ergonomics cannot deploy positive effects.
3 – 3	Champagne ! ! 	The characteristics of real work are not taken into account.
| - 4		Testing procedures cannot reflect the reality of the user's work.
| - 5		The finished product does not correspond to what the user needs.
| - 6		Even if it has been developed with great professionalism, the product cannot overcome the shortcomings of the management method and fails in its task.
| - 7		Wouldn't it have been better to develop a tool that is closer to the real needs of the users? The product wouldn't have cost more to build and it wouldn't have shown any discrepancy with the users' expectations nor would it have failed in its task.
| - 8		The product would have cost less – *since it was immediately developed as the users expected* – and it wouldn't have led to a loss of productivity – *the first consequence of the error in Production* – nor would it have reduced the Return on Investment.

It is obvious that the tools exist, we just have to use them. The first trap is not the lack of a solution, but the lack of will to apply it.

It is obvious that tackling a problem that is 60 years old and that, because of its age, has managed to convince everyone that it is impossible to eradicate it, is not an easy task, but the only insoluble problems are those that are not tackled.

It is also worth noting that the IT world has brilliantly succeeded not in solving the problem, but in changing the way it is perceived, to make people believe that it has been solved.

[1] The French approach is the theoretical exercise resulting from a possible implementation of activity's ergonomics based on French ergonomics as taught two decades ago in some French technical universities as well as the use of software developed with the tools described in the specialised literature at the end of the last millennium.
We would like to point out that these tools have never been put into industrial production, they have remained laboratory curiosities.

Hope in the software crisis: the high-quality app

There is nothing to expect from development models, none of which can solve the quality crisis on its own.

The answer will only come, if there's someone who's likely to listen, from the side of the IT project management.

What does the customer ask when complaining about the software crisis?

The client's complaint is clear:

- The software is inadequate, it does not meet the needs.
- They have too many errors, they are difficult to maintain.
- They are too expensive, deadlines and quotations are not kept

And what did the IT world understand of this demand?

The IT world must have been surprised to be so attacked, because unless one knowingly does something wrong, everyone is convinced that one is doing their best. Therefore, everyone takes it for granted that their work is good.

The first answer, the Waterfall/Cascade model, was a good answer. The validation of the work carried out at the end of each step resulted in an improvement in average quality.

However, the major issue of cost control remained, since an error detected during final testing could jeopardise the viability of the product. The problem of unsuitability remained: the customer could only control the suitability of the product when it was finished, i.e. when the development budget was exhausted.

Model V, which was subsequently adopted, partially addressed the problem of late detection of defects during the final testing process. The unit tests, followed by the modular tests, ensured that the application was free of its most obvious defects and easier to maintain, as well as some cost control; however, the issue of unsuitability remained unresolved, as it was only upon delivery of the finished product that the customer had the opportunity to discover defects or misunderstanding of its requirements.

The adoption of incremental and iterative models has made it possible to correct some of these problems. In these models, a module is delivered to the client as soon as it is completed and tested. The client then receives the application to be validated in bricks and pieces, and if one of them presents a problem, the correction can be undertaken in a very short time.

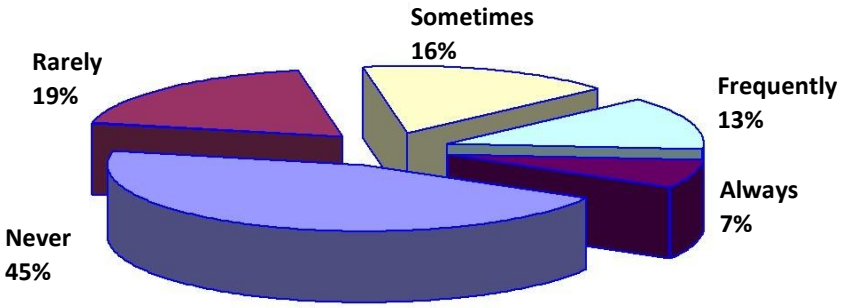
The problem of incapacity was finally being addressed: if a deficiency was identified, it could be rectified within the framework of still available funding.

Current situation

Despite all the promises made by the promoters of these various methods, the situation in the software world has hardly improved since the NATO Conference in 1968. Projects continue to be poorly managed, budgets and deadlines consistently missed, user satisfaction continues to be poor, and quality continues to be perfectible...

How can such a harsh observation be established when the methods used are supposed to guarantee software development free of such excesses?

The CHAOS Report 2002, published by the Standish Group, published a very interesting statistic:



Pie chart shows the usage rate of features by the users of an app.

As can be seen, almost half of the features developed (and paid for by the client) are in vain since they are never used. The bundle of rarely or never used features accounts for two thirds, while the bundle of frequently or always used features accounts for 20% of the total.

The question of what features have never been used is a very important issue. The Standish Group study deals with *ad hoc* software developed for professional use. That is, any functionality present in such software has been explicitly requested and defined in the specification that served as the basis for the development of the final product.

There is therefore a glaring contradiction in the fact that a certain functionality is required, analysed, developed, paid for but not used.

Frequency of use and rejection

An initial answer is that features that are seldom used are poorly mastered and require constant attention when using them. As a result, the high workload involved in implementing such features reduces productivity.

The corollary of this is that the user will develop a certain aversion to features that are difficult to use and have a high error rate.

Therefore, the user will develop avoidance strategies so that he doesn't have to use them. And since he doesn't use them, he has no opportunity to learn how to master them.

We are in a vicious circle:

- If the functionality is difficult to learn, the user will avoid using it.
- If he doesn't use it, he has no way of learning to master its subtleties.
- As a result, the functionality has been developed in vain and the money invested in the functionality has been wasted. Software containing such functionality is too expensive for its use, the return on investment will be poor, and both user and funding provider satisfaction will be low.

This last observation is exactly what was made in 1968, when the software crisis was first denounced, and all the answers given since then have all missed the mark, regardless of the real or supposed merits touted by their promoters.

Cause of the failure of development methods

The methods used to date have all failed to varying degrees because they have not taken into account the fundamental characteristic of IT products: they are all unique.

Contrary to what happens in industry, there is no software prototype. The programmed object is exactly the same as the one that will later be used by the end user. It is not a copy in the industrial sense of the term that sees two separate objects coming out of the same mould.

Depending on the hazards of the injected material, the service life and resistance to stresses of an industrial product are not the same. The first one may fail after 2 days, while the next one will offer several years of good use to the customer.

Copies of the same software installed in two different computers are, on the other hand, one and the same object. If the same manipulations are carried out in parallel, the two clones will fail identically and simultaneously.

As such, the creation of software is much closer to an artistic activity, such as sculpture, than to industrial production. The failure to take this into account is at the root of most of the known problems.

The misunderstanding is reinforced by industry usage: when industry speaks of a prototype, it is something that comes out of design offices and has nothing in common with the finished product. For example, the prototype has been machined piece by piece with machine tools or made with a polymer resin 3D printer while the finished product will be injected into a mould.

Each of these manufacturing methods creates objects that may behave completely differently. For example, the response to stresses cannot be compared: machining tends to create microcracks that can act as fracture initiators; 3D printing behaves rather isotropically, while casting induces tensions within the part depending on casting speed and solidification conditions.

In the IT world, a prototype should be called a sketch, because everything in it will end up in the finished product; just as the finished statue will contain all the defects in the sketch, *quick and dirty* development can never result in anything other than a *quick and dirty* final product.

Any component implemented in the sktech will remain integrated throughout the development of the product to the distribution stage; and any imperfection left in the sketch will be a potential flaw.

This method is all the more dangerous because it is based on the illusion that the code is constantly being reviewed to track down errors. In reality, dirty code that withstands testing will be incorporated into the finished product "*as is*" because the criterion for introducing a code into the finished product is not the cleanliness of its programming, but its ability to pass the tests.

The Agile Manifesto

This Achilles' heel demonstrates that all the methods used in the Agile Manifesto can only produce low-quality applications. This flaw in iterative or incremental methods is reinforced by the financial contingencies of the economic environment. If software programing really copied the industry, it would prototype the product and rewrite the entire code at the time of writing the final version for commercialisation.

If only for financial reasons, such a method is unrealistic, although in theory it is capable of guaranteeing a code free of almost all its errors.

The reason for the failure of Agile methods is that they assume that any code will be reviewed at least once during the development process. As a result, there is no quality pressure on any stakeholder, since everyone is allowed to believe that the burden of detecting potential errors rests on a subsequent audit implicitly integrated in the implementation of the Agile process.

The definition of the framework is crucial because only an explicit framework based on clearly defined procedures can guarantee satisfactory quality. If quality is based on implicit procedures, it becomes dependent on humans who interpret these procedures according to their idiosyncrasies.

As a result, the reliability of the system can no longer be guaranteed, since the slightest human failure automatically leads to the failure of the entire development process.

Redefining the system

Therefore, the system must be made independent of the people and be based only on the procedures that will be implemented by the people. The individual disappears behind the procedure that becomes the medium of quality.

Contrary to appearances, such a system is very favourable to people who see their level of stress greatly reduced. The stability of the system is enhanced by the fact that employees can devote themselves to their work with complete peace of mind.

Since all current methods have failed to achieve satisfactory quality due to their failure to take into account the human factor, the following chapters will examine which tools are best suited to meet these challenges.

Rousseau and the managers

There seems to be a *priori* that defines staff as systematically working correctly; mistakes are unfortunate events beyond the control of both management and executors.

The employee is intuitively perceived by his or her superiors as a person who unknowingly commits misconduct out of his or her own free will, but who is educable because of his or her good will.

The logical corollary of this is that nothing is done to prevent errors in product development.

This industrial rousseauism can be expressed as: '*The employee is born good, but the constraints of work pervert him. Appropriate education will save his soul and improve his productivity*.'

In contrast to this molasses of good feelings, behaviouralism has a position that could be described as cynical or disillusioned: '*The employee always works at his best but is incapable of achieving perfection. There is no method of sublimating his original imperfection. Only a change in his environment can transcend his defects to allow the desired qualities to emerge*.'

Of course, HR specialists and the hierarchy will defend themselves by arguing that they have many very effective strategies to improve quality. But the analysis of the results shows that these practices are as much a matter of sports training as military drill and are more or less attributable to autosuggestion.

Their very relative and ad hoc successes, coupled with a flagrant inability to stabilise the lessons of these successes over the long term, testify to the weakness of these approaches.

If computer scientists were sheep

We propose to clarify the problems associated with the hierarchical structure of IT development by comparing a company to a herd of sheep.

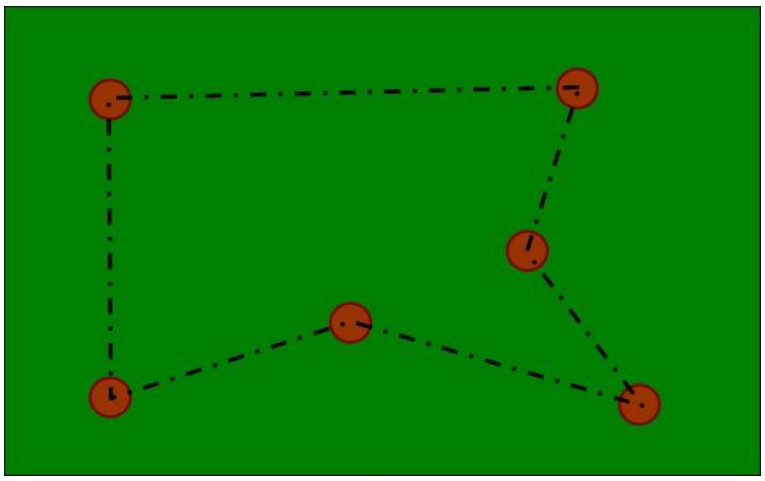
In this demonstration, the shepherd is the mirror of the management of the company. The hierarchical framework, from top to low level management, is symbolised by the shepherd's dogs, and the productive forces are represented by the sheep that make up the herd.

It would be futile to search for any correspondence between the usual reputation of these avatars and the actual behaviour of the individuals thus described. This study is not a social satire, in the sense that the fables of Aesop or Jean de la Fontaine were.

The subject of the study is not the animal or the human psyche, but the relationships between the actors of this demonstration.

The shepherd and his sheep

As explained in the preamble, the IT provider is like a shepherd who has to move his sheep through his client's land. There are areas where sheep will be allowed to graze; and there are prohibited or dangerous areas. The illustration below shows such an area.



The client ordered the shepherd to only let the grass graze between the trees.

The right supplier, the right shepherd, will have to get his employees, his sheep, to provide the customer with the right product, respectively, to graze the permitted areas without touching the rest. To do this, the shepherd is helped by his dogs, just as the entrepreneur has managers, i.e. a supposedly effective hierarchy.

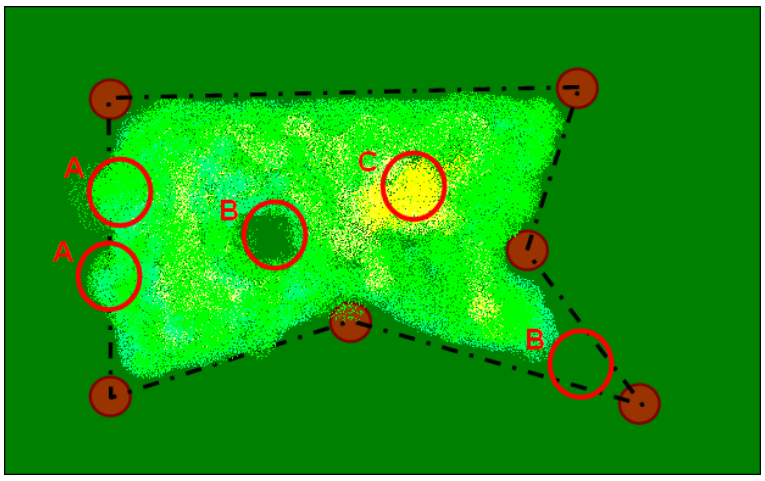
The shepherd will use long, modulated whistles to get his dogs to guide the herd to stay on the agreed ground, just as management issues instructions and service orders to ensure that employees work properly.

But behind this idyllic vision lies a tremendous amount of stress: the management is putting a lot of energy into keeping the project within its framework, in the same way that the dogs run around to keep the sheep within the permitted perimeter.

The sheep, on the other hand, are stressed by the whistling and barking that serve as reminders to order; this is also the case for people involved in the development of a product who have to justify the progress of their work at each team session.

When the job is done

After the sheep have passed, this is what the land in question might look like:

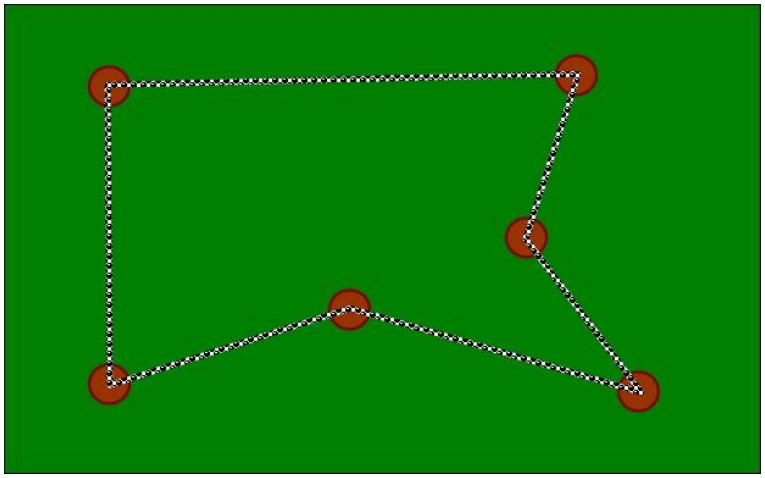


A closer look at this picture reveals several problems:

- In **A**, the sheep have exceeded the permitted limit. In the case of software, this can result in a failure to protect data integrity, or a backdoor that opens up a security breach.
- In **B**, the sheep did not graze a part that should have been grazed. In the case of software, this may result in a missing function, although properly specified in the specifications given to the supplier.
- In **C**, the sheep have overgrazed the grass, which will lead to regrowth problems; in other words, there is a decrease in the value of the land. In the case of software, this is equivalent to over-specification on the part of analysts or redundant functionality. The consequence of this over-specification is that the customer will have to pay a disproportionately high price for the function in question. As a result, the final product will have its development costs exceed the optimum, thus worsening the return on investment.

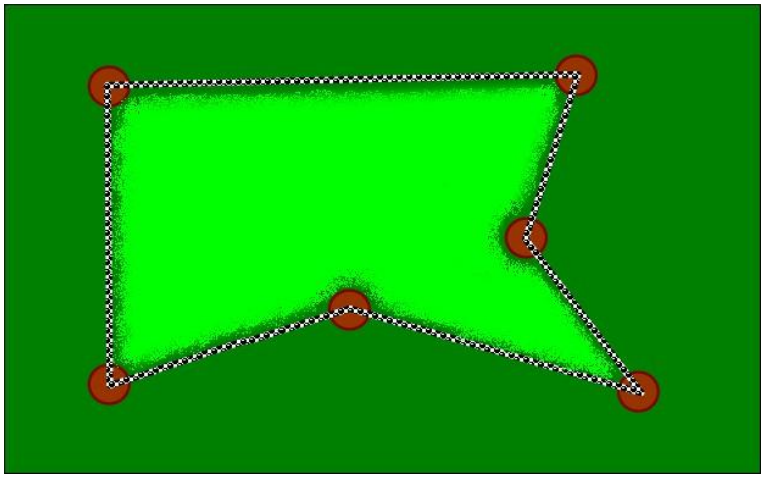
A simple solution: the fence !

The shepherd is not helpless in dealing with this problem: instead of being stressed about not being able to effectively control the behaviour of the sheep or having the herd under the stress of a possible wandering outside the permitted area, it is much simpler to demarcate this area with the help of a fence.



While the Rousseauist shepherd hopes that someday his sheep will be sufficiently educated to determine for themselves which parts are to be grazed or not, the behaviouralist shepherd knows that he cannot trust his sheep at all, and will never, at any time, imagine trying that possibility.

He puts a fence, because he knows that this is the only really effective way to get his sheep to graze only where it is allowed, with or without the help of dogs.



Productive forces, be they sheep or employees, will see their stress greatly reduced because they will no longer be subjected to excessive pressure from management.

On the other hand, an event will appear in the world of employees, which has no equivalent in the pastoral world: the sheep fence will be translated into a virtual framework that will induce employees to take responsibility.

There is a trade-off between an external stress, the management pressure, and an internal stress, the compliance with specifications. External stress can be experienced as a negative factor suffered by the employee; on the other hand, internal stress can be experienced as a positive factor, because the employee knows that he is in control of this constraint.

Faced with this obligation, the employee has the choice of voluntary submission. In this case, the coercion will no longer be perceived as a burden, but as a motivation.

There is nothing new behind this presentation: scientific studies on voluntary submission date back almost half a century, and the resulting recommendations have already proven their power in the field of marketing.

This behavioural proposition is similar to a Fabry-Perot interferometer. Even a very high-quality laser cannot emit perfectly coherent radiation. It is similar to an employee who does his best but is unable to exceed a certain quality limit.

To increase the precision of the wavelength of a laser beam, an interferometer formed by a thin plate covered with semi-reflecting surfaces is added to the emitter.

This plate generates destructive interference which cancels out wavelengths other than those in the desired range. Only the wavelength range coming into resonance with the thickness of the optical plate will be able to pass through the optical device and the beam coming out of the interferometer is cleaned of undesired frequencies and therefore of better quality.

This is exactly what we propose in this paper: to develop an approach to software development that draws on the teachings of behavioural psychology to eliminate human intrinsic weaknesses, with the aim of achieving near-perfect quality.

A picture is worth a thousand words

We have shown that an environment designed for the masses and not for hypothetical gurus has a much better chance of success. The above example is only an illustration of what is desirable, but above all it shows that it is possible to do better if the situation is approached correctly and the goal is set correctly.

Thank you for your attention.

Mid-April 2026 Addendum

A newcomer full of promise?

The media have recently been full of praise for the feats of an AI engine that has 'discovered' thousands of flaws in software that have been around for decades and are reputed to be solid.

There is no rocket science behind these announcements, the engine has done what it was designed for : search for inconsistencies and, like any software, it has done it very quickly and very reliably. But it has not been able to go beyond its intrinsic limitations: it is a software, it was born a software and it will remain a software.

Searching for an inconsistency is one thing, correcting it is another and the software is not (yet) capable of that.

As the journalists who have covered this subject have just revealed, there are two practical consequences of this success:

- The first is that it gives the companies that have access to this tool the possibility of tracking down the flaws and correcting them as quickly as possible. This is a 'White Hat' strategy (the good hacker with noble motives.)
- But the corollary is just as valid: the company that owns the engine has carefully locked down access to it to prevent it from being used by 'Black Hats' (the bad hackers with the darkest motives.)

What has been done can be copied and if access to the only instance currently at work is temporarily locked, this will only be an additional motivation to program another engine that will replicate these functionalities.

We should therefore see a tsunami of cybercrimes in the near future.

Is there any hope?

Hope is what dies last; as long as we are not dead, there is always hope !

This hope is simply what we have just described above: with the help of adapted supervision tools, it is possible to create software without defects simply by supporting the effort of engineers and architects in a way that prevents potential flaws.

We have already created a demonstrator that has fully validated this reality. Not only has the work been greatly simplified, but the development time has been halved and the costs have been reduced by a third.

However, we must emphasise a crucial point: as much as it is possible for an AI to review an existing code to detect flaws, it is conceptually impossible to load an AI with the task of developing a flawless software. Creation *ex nihilo* is (and will remain for a long time) beyond the reach of an automatic tool, only a human can possess the abstraction capabilities necessary to succeed in this task.