Received February 23, 2020, accepted March 30, 2020, date of publication April 8, 2020, date of current version April 29, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2986545

IT Project Failures, Causes and Cures

SOREN LAUESEN
Computer Science Department, IT University of Copenhagen, 2300 Copenhagen, Denmark
e-mail: slauesen@itu.dk

ABSTRACT We have known for decades that IT projects often fail. The usual explanation is that the cause is poor project management, poor cost estimation, poor requirements, etc. But how can we prevent these causes? To answer this question, it is not sufficient to know that there was poor project management, etc. Would it suffice to educate better project managers? And what would we have to teach them that they don't learn today? We have to know what actually took place in the projects, identify the root causes, and find ways to prevent them (cures). This is similar to accident investigations in aviation. This paper reports the results of five accident investigations of large government IT projects in Denmark. We identified 37 root causes and 22 potential cures. Surprisingly, only one of the causes is programming-related. Each project suffered from around 15 of the causes. Twenty-seven of the causes are not reported in the research literature on IT project failures (e.g. *surprises with system integration* and *wrong estimate of human performance*). Half of the cures are familiar to developers, but were ignored in the specific project (example: *usability test*). The rest are unfamiliar (e.g. *problem-oriented requirements* and *monitoring remaining work*).

INDEX TERMS Project management, project failure, accident prevention, software development.

I. INTRODUCTION

We have known for decades that IT projects often fail. The usual explanation is that it is caused by poor project management, poor cost estimation, poor requirements, etc. [2], [13], [22] But how can we prevent these causes? To answer this question, it is not sufficient to know that there was poor project management, etc. Would it suffice to educate better project managers? And what would we have to teach them that they don't learn today? We have to know what actually took place in the projects, identify the root causes and find ways to prevent them (cures).

Let us compare with the situation in aviation. Flight accidents are by definition *events that cause significant damage to persons and/or aircraft*. How do the investigation boards prevent accidents? They are not satisfied with knowing that it was a pilot error and that 80% of accidents are caused by pilot errors. They find out what actually happened during the flight (the time line) and what caused the accident.

Example: A plane crashes into a forest. What happened? Shortly after takeoff, one of the AI's (Attitude Indicators or artificial horizons) fails. The pilot turns a switch that makes both pilots see the same AI. However, he turns the switch in the wrong direction so that both pilots see the failing AI. This is the primary cause of the accident. Why did he do it? He had been trained with a switch version that worked as the pilot expected (this is a root cause).

The board may have to ask "why" again - maybe several times - to find more root causes, e.g. about certification. There are other causes too, e.g. that it was a dark and misty night. When the board knows the causes, they can come up with ways to prevent similar accidents in future flights.

Finding out what happened and why, is usually difficult. It may require technical investigations, interviews, flight recorder data, medical records of the pilot, weather data. The board uses whatever is necessary in the specific case. The investigation report is published.

In the IT world, we will define an accident as *a project with significant damage, e.g. large cost or schedule overrun, failing business goals, low usability, etc*. Accident investigations in IT projects are rarely published, and as a result, we know little about the root causes [2].

This paper reports the results of five accident investigations of large government IT projects in Denmark. Like most government projects, they were outsourced with fixed-price contracts. The author and project-specific co-investigators identified 37 different causes, each of which caused damage to one or more projects (see table 2). As discussed below, 27 of these causes are apparently not reported in the research

The associate editor coordinating the review of this manuscript and approving it for publication was Resul Das.

literature on IT project failures (e.g. *surprises with system integration* and *wrong estimate of human performance*). Further, we came up with 22 cures that in combination could have prevented most of the damages (see table 3). Half of the cures are familiar to developers, but were ignored in the specific project (example: *usability test*). The rest are unfamiliar (e.g. *problem-oriented requirements* and *monitoring remaining work*).

IT developers in industry say that they see the same causes, but industry practices a few more cures.

II. BACKGROUND

There is a lot of research on IT project failures. I have selected the following papers to illustrate the various methods for finding failure causes and cures over time:

Jones, 1995 [4] discusses cost overruns caused by programming troubles. He reports 10 causes, e.g. *failure to use code inspection*, *requirements creep*. He suggests 11 cures, e.g. *overtime pay, redevelop modules, complexity analysis*, and *terminate project*. There is no overlap with the causes and cures that we found. In general, we found few causes in the programming area.

Montealegre and Keil, 2000 [14] look at the Denver International Airport baggage system (1994). The system was crucial for fast transfer of passengers and their baggage from one flight to another. The system never performed and delayed opening the airport for years. The authors report the damages (huge cost and schedule overrun; the system never worked and even crashed the bags). However, the authors don't report any root cause. External consultants tested an isolated loop of the total conveyor system. This too didn't work, indicating a simple technical root cause. However, nobody reacted. Also, the authors of the paper didn't notice. Nobody made a true accident investigation.

Wallace and Keil, 2004 [22] look for correlations between risk factors and project outcome. They identified 53 risk factors. They made a survey with 507 software project managers, asking them about the importance of these factors and the project outcome in their most recent project. They cover only 7 of our causes, e.g. *lack of project goals, lack of management support*.

Jones, 2006 [5] reports why cost estimation is often wrong, e.g. *cost estimates are demanded before requirements are fully defined, conservative estimates are overruled by management*. It is likely that some of these factors have been at play in the five projects we have studied, but we cannot see that they caused damage.

Kappelman *et al.*, 2006 [6] identify 53 project risk factors, based on literature and interviews with experienced project managers. The factors can be assessed early in the project, thus giving early warnings. The factors cover 6 of our causes, e.g. *undocumented requirements, no business case*.

Cerpa and Verner, 2009 [2] report that there are few post-mortem investigations and as a result little is learned. However, based on questionnaires to 90 software practitioners in 70 failed projects, they come up with 18 failure causes such as *project underestimated* and *staff had an unpleasant experience*. They cover 3 (maybe 5) of our causes.

Flyvbjerg and Budzier, 2011 [3] report that some projects have cost, schedule and benefits much worse than average. They claim they are able to predict such failures, but don't give details. As a cure, they suggest to *break large projects into several smaller projects*. This roughly matches one of our cures. They seem uninterested in finding other cures that could prevent the failures.

McLeod and MacDonell, 2011 [13] provide an extensive review of literature and summarize the findings as 18 factors that can influence success. They break the factors down to 56 sub-factors, e.g. *size, complexity, technology, active and meaningful user participation*. However, we cannot see that these factors relate to the causes we observed.

The papers are based on questionnaires, literature and sometimes expert knowledge, rather than accident investigations. This may be the reason they don't cover more than 9 of the 37 damage causes we found. Further, little is written about potential cures, e.g., what we can do to prevent poor project management or high complexity.

We apparently missed most of the 50+ factors reported by others. Why? We have looked at these factors, but couldn't see that they occurred in our cases. And if they did, they had no significant effect on the damages. Some factors have disappeared over time because IT specialists have learned the lessons. As an example, lack of user involvement was common earlier. We don't see this in Denmark anymore. However, we see cases with excessive user involvement.

III. THE FIVE PROJECTS AND THEIR DAMAGES

In four of the five investigated projects, the author was fortunate to work as a consultant to the National Auditors of Denmark. They act on behalf of the parliament and can get access to documents and conduct interviews that otherwise would be extremely hard to get.

The first investigation started late 2009. The National Auditors looked at the recently deployed Electronic Land Registry. They contracted with the author to help them with the IT aspects. They had selected the Land Registry because the press had focused on it, and because it caused large problems to citizens and the financial sector. There is a detailed account of the Land Registry investigation in Lauesen [9].

The other four projects were selected in the same way as something that caught national attention. The author investigated three of them in cooperation with the National Auditors, and later continued the investigation on his own. The last project (the health record system) started with an exceptionally good report written by two thesis students [16]. Lauesen continued the investigation, interviewed project staff, observed surgeons at work, found additional reports, etc.

An investigation of this kind takes around 9 months. You don't just make dozens of such investigations. You have to catch the opportunity when it occurs. As in aviation, it is important to investigate every accident in order to prevent similar accidents in the future.

TABLE 1. **Damages in the projects. We show the estimated and actual value like this: *3.5 became 7.5*. We show that they are almost the same in this way: *Around 85*.**

Damages	Land Registry	Travel card	Police case mgmt.	Debt collection	Health record
Development time, years	1.2 became 2.7	3.5 became 7.5	3.7 became 7	2.5 became 8.5	Around 3
SW cost, M $	Around 11	Around 85	25 became 50	20 became 86	Around 160
Other costs, M $	27 became 50	4 became 14 HW around 70	0 became 30	No data	Around 240
Business performance	Initially 30% of registrations delayed 40 days	Little benefit to operators	None specified	Benefit lost, 10,000 M $ debt delayed for years	Initially low productivity
User satisfaction	Initially low	Initially low	N/A	No data	Initially low
Supplier loss, M $		70	7		
Feasibility doubts			Performance	Legality	
State today	Success	Success	Closed	Phasing out	Operates

The basic facts about the first four projects below, are extracted from the National Auditor's reports. Here is a summary:

1. Electronic land registry [17]: Replace the many paper-based registration offices with a central electronic registration of ownership, mortgages, etc. for the entire nation. It was planned that 30% of the registrations had to be done with human intervention, either because they were complex or because they were selected randomly for security check. The system had to integrate with banks, the civil registration system and several other systems.
2. Electronic travel card [18]: Replace the many ticket systems with an electronic card for travels by bus, train and ferry in the entire nation. The contract included also cabling and scanners in busses, railroad stations, etc.
3. Police case management [19]: Replace the current, partly paper-based system for keeping track of offences, detection, etc., with an electronic one. The system had to integrate with many existing systems.
4. Debt collection [20]: Replace the many bailiff offices in municipalities, electricity corporations, etc. with a centralized system that automatically finds the best way to collect debts. The system had to integrate with around 20 other systems.
5. Health record system [16]: Replace the many clinical systems in 17 large hospitals with one. The system had to integrate with 20 other systems and numerous medico-technical systems.

The projects suffered various kinds of damage, such as cost overrun, insufficient business results, or low user satisfaction. Table 1 gives an overview. Four of the projects had large schedule overruns. Two had large cost overruns. All of them had insufficient business results. In two of them the supplier lost a lot of money. For two of them, it was dubious whether they could ever succeed (feasibility doubts). Two (maybe three) became a success although they had damages. Only one of them was never deployed (Police case management).

A general observation was that everybody had tried to do their best under the given circumstances, yet the project suffered significant damage.

Were all the projects failures? It depends on how we define failure. In general it is hard to define what a project failure is [13, page 8]. Inspired by accident definition in aviation, we avoid the problem by talking about significant damage. In some cases, the project has so severe damage that it is cancelled. We would call this a failure. But it may also be cancelled for other reasons, such as a better and cheaper solution appearing on the market. We would not call this a failure.

Since most projects don't die, but operate in spite of damages, we talk about accident investigations, not post-mortem investigations.

IV. IDENTIFYING WHAT HAPPENED – THE TIME LINE

How can we find out what happened in an IT project, i.e. make a timeline? Ideally we would study all documents produced on the way, interview developers and stakeholders, and study the system in actual use. In practice it is not that easy for several reasons:

1. There are so many documents that you don't know where to start, e.g. a contract consisting of 2,600 documents.
2. You cannot get access to the documents.
3. Informants don't want to be interviewed, are not allowed to, or cannot set off the time.
4. Managers often consider the system a success, although developers and stakeholders find it unsuccessful. As a result, the manager opposes any investigation.

Four of the five projects were investigated by the National Auditors of Denmark. Normally, the auditors don't investigate the IT supplier's role, only the customer's (the government organization). However, the author persuaded them to involve the suppliers too in these projects. It gave a very different picture of what happened. This experience made us involve other stakeholders too. In the Land Registry project we involved real-estate agents, lawyers, the financial sector and municipalities. We learned that informants often said they did X (e.g. made a usability test), but when we looked at the details, it was something else they did (e.g. discussed the user interface with a few users). It was obvious that we would not have identified important project events if we had used only questionnaires and management interviews.

The National Auditors look primarily at the economic and managerial aspects. So in order to identify the IT-related aspects, the author gathered additional information in these ways:

a. Contacted key IT staff and asked for an interview and discussion. Explained what the purpose was: To identify damage causes in order that others can learn from them. The purpose was not to find someone guilty. This convinced many participants to tell what happened from their point of view.
b. Promised to anonymize all sources and allow the informant to review the report.
c. Gave seminars for experienced developers and project managers, where the author explained the findings. Many participants had been involved in one of the five projects, and they told the author points he had missed, agreed to meet him later, or told him whom to contact for further information.

V. IDENTIFYING THE CAUSES

How can we identify the accident causes in the projects? The author studied each project in cooperation with people who were or had been closely involved in the project. So when I say ''we'' below, it is different we's for each project.

We studied the timeline of the project and identified items that had a major effect on the damages. They were the primary causes. Sometimes we had to ask ''why'' a couple of times to get the root cause. Next, we generalized the causes to causes that might appear in other projects too. As an example, we observed that the new health record system had this damage: *Waiting lists grew and the hospitals had to pay overtime salaries*. The cause was that doctors spent longer time on each patient visit than before. This is a specific cause that won't occur in non-medical systems. Can we define a broader cause that covers other projects?

The Land Registry system was damaged in a similar way, because the registry staff couldn't perform as fast as expected. The registrations piled up and often took weeks instead of days. As a result, citizens lost money because they couldn't get a loan as planned. These two observations became cause *F3: Wrong estimate of human performance*.

In total we identified 36 root causes. Table 2 shows all of them, the projects in which they occurred, and how they could be prevented (the cures). A full description is available in [12]. The causes are ordered according to the kind of development activity where they tend to occur:

A: Analysis, causes A1-A10.
B: Acquisition, causes B1-B3.
C: Design, causes C1-C5.
D: Programming, causes D1-D2
E: Testing, cause E1.
F: Deployment, causes F1-F3.
G: Management, causes G1-G13.

These activities occur in waterfall development as well as agile. In waterfall, the activities are long, in agile there are many short ones.

The green causes in table 2 match common IT project failure literature. Examples: *Requirements don't cover customer needs (A2)*; *wrong cost estimates (B3)*. The yellow causes are mentioned in a Danish investigation (Bonnerup, 2001 [1]). Examples: *Makes heavy demands and believes it is for free (A4)*; *excessive user involvement (G10)*. The red causes are not mentioned as causes in common project failure literature. Examples: *The customer doesn't assess the proposals (B2); designs user screens too late (C2)*.

Some of the red causes are mentioned indirectly in maturity models or systems development literature. As an example, maturity models say that the customer must assess the proposal. The assumption is that if he doesn't do so, it may cause damage. We provide evidence for this. Similarly, systems development literature advocate early prototypes. We provide evidence that not doing so can cause damage.

Some judgement is needed in order to decide whether a cause in literature matches one of ours. We have been quite liberal in this judgement. Here is an example:

Kappelman *et al.* [6] writes as cause 2: *Functional . . . requirements are not documented*.

We consider this roughly the same as cause A2: *Requirements don't cover customer needs*.

Each project was hit by 13 or more causes, which together caused the observed damages (multiple causes are also reported by Cerpa and Verner [2]). Most of the causes occurred in two or more projects. You can read a detailed account of each project in *Damages and damage causes* [12]. It also gives details of the causes, how the causes generated damage, and the potential cures.

In some cases, we have observed a cause that caused damage in other projects, but not in this one. In table 2 it is marked with (x). An example is A2: *Requirements don't cover customer needs*. This was the case in all five projects, but it caused damage in only the first four. In project five, the health record system, the customer selected one of three widely used COTS solutions. The selected system could do everything needed in a hospital, so the requirements were unnecessary. Fifteen other causes generated the observed damage.

We are confident that the causes we report actually occurred. But the opposite may not be true. A project may suffer from a cause we haven't reported, because we haven't observed it or heard about it. After the first release of the damage report [12], the author got reports of several causes he didn't know.

A. BAD PROGRAMMING?

Only one of the damage causes relate to programming and test (D1: *Surprises with system integration*). The rest relate

TABLE 2. Green causes: Covered by common project-failure literature. Yellow: Covered by a Danish investigation, 2001 [1]. Red: Not mentioned in project-failure literature. Guidelines may mention the cure, but not the cause.

		Where found					Cure type				
	Damage causes and cures by project	Land Registry	Travel card	Police case mgmt.	Debt collection	Healh record	Familiar - ignored	Unfamiliar	Misinformed	Cure unknown	
	Number of observations:	18	13	18	17	15	23	17	3	5	
ID	**Analysis**										**Cure**
🟥 A1	Doesn't identify user needs and win-win	x	x	x	x	x	x				Study as-is and plan to-be
🟩 A2	Requirements don't cover customer needs	x	x	x	x	(x)		x			Problem oriented requirements (SL-07)
🟩 A3	Describes solution in detail. No freedom to supplier			x	x	(x)		x			SL-07
🟨 A4	Makes heavy demands and believes it is for free	x	(x)			(x)		x			Scope management, Open target requirements (SL-07)
🟩 A5	Oversells technology, e.g. SOA, web-based	x		x	X			x	x		Check technology (get second opinion . . .)
🟥 A6	Multi-vendor strategy - supplier independent ! ?				x			x	x		Third-party can integrate, exit strategy, flex SOA
🟥 A7	Wants everything at once, e.g. cover all types of debt	x		x	x	X	x				Pilot test, Deploy part-by-part
🟥 A8	Doesn't plan the new work processes		x	x		x	x	x			Plan to-be, SL-07 reqs (sections B1 and C)
🟥 A9	No feasible solution, e.g. data missing, performance dubious.			x	x		x				Ask experts, Early proof of concept (POC)
🟩 A10	Surprising rule complexity.		(x)		x	x	x				Deploy part-by-part, Expert spec of complex cases
	Acquisition										
🟨 B1	Supplier too optimistic - you must lie to win		x				x				POC
🟥 B2	Wrong selection criteria		x			x		x		x	SL-07 (B4-6)
🟩 B3	Wrong cost estimates		x	x	x		x	x			Function points (based on SL-07 reqs), Cost checklist
	Design										
🟥 C1	Doesn't ensure usability, even when they know how	x				x	x				Early prototypes & usability tests
🟥 C2	Designs user screens too late	x	x				x				Early prototypes & usability tests
🟥 C3	Accepts the solution description without understanding it		x				x				Early prototypes & usability tests
🟥 C4	Cannot see how far the supplier is	x	x				x	x			Early prototypes, Monitor remaining work hours
🟨 C5	My way without considering the supplier's way			x				x			SL-07 reqs
	Programming										
🟥 D1	Supplier accepts expensive reqms interpretation		x	x			x	x			Replan, SL-07 reqs
🟥 D2	Surprises with system integration	x		x	x	x	x				POC, Pilot test
	Test										
🟥 E1	Deploys the system with insufficient testing	x			x	x	x				Pilot test, Ask expert developers
	Deployment										
🟨 F1	Deploys system with insufficient support/training	x		x		x	x				Usability tests, Pilot test, Deploy part-by-part
🟥 F2	The system is not used as intended			x	x	x		x			Observe at pilot test, Follow-up study
🟥 F3	Wrong estimate of human performance	x				x		x			Check at POC, Check at pilot test
	Management										
🟩 G1	No business goals - or forgets them on the way		(x)	x	x	x	x				Monitor business case regularly
🟩 G2	Doesn't reschedule. Assumes the rest can be compressed	x				x	x				Replan at surprises
🟩 G3	The project grows without anybody noticing			x				x			Scope mgmt, Monitor remaining work hours
🟩 G4	Risk assessment downplays the danger	x	x		x		x				Correct risk management, plan what-if
🟥 G5	Financial incentive disappears and parties fight		x	x			x	x			Monitor remaining work hours, SL-07 K2, Payment plan
🟥 G6	Cashes benefit before it is harvested, e.g. sacks too early	x			x	x	x			x	Replan at surprises + ?
🟩 G7	Lack of management involvement			x	x			x			Give mgmt IT-insight, SL-07 contract 3.4
🟨 G8	Excessive management or expert involvement	x								x	?
🟥 G9	Too large committees/work groups without competencies		x			x	x				Small task force with authority, SL-07 contract 3.4
🟨 G10	Excessive user involvement			(x)		(x)	x		x		Small task force with authority
🟥 G11	Believes law blocks sound approaches, e.g. talking to suppliers	x			x			x		x	Use constructive lawyers. Make them part of the team + ?
🟩 G12	Insufficient staffing	x					x				Monitor remaining work hours, SL-07 contract
🟩 G13	Doesn't find the root cause			x						x	Ask expert developers, Give mgmt IT-insight

to the early activities, to deployment and to management. This is a surprise since the belief in the general public is that IT disasters are about bad programming. Actually, the suppliers may have had many programming troubles, but if they have, they don't tell about them and they are not visible as damage.

Another possible explanation is that the projects had few programming troubles because programming is rather straightforward in all of them. Development tools and programmer education help doing things right. Further, programming was mostly done in an agile way with close interaction between developer and customer.

B. TOO MUCH CAN BE HARMFUL

Two well-known damage causes are lack of management involvement and lack of user involvement. We and Bonnerup [1] observed these too, but also the opposite: *G8: Excessive management or expert involvement*, and *G10: Excessive user involvement*.

As an example, the Land Registry project was managed by an ambitious registry judge, who was a driving force throughout the entire project. However, he also designed the user screens in cooperation with a graphical designer. They didn't check usability with real users, because the assumption was that since the expert had designed it, it was good. This was excessive management involvement as well as excessive expert involvement. The result was a very cumbersome system that even lawyers and real-estate agents found difficult to use.

C. DO DEVELOPERS KNOW THE CAUSES?

The author has presented the list of causes at many seminars with seasoned developers and project managers. They always said that they recognized the causes when they saw them, yet many of the causes were a surprise. Developers hadn't thought of them before.

VI. CURES - PREVENTING THE CAUSES

When we had identified a substantial number of causes, the author reviewed the list with experts from several areas, e.g. programming, requirements, UX, function points, and project management. For each cause, we identified potential cures. In a few cases we couldn't point to an existing cure and tried to invent one. Next, during many of the seminars, participants reviewed the causes and cures, and provided their own experiences. Quite often, participants suggested that *agile would cure most of the causes*. However, when we went into detail with "how", it became clear that the term agile was too broad to make sense as a cure.

Further, if agile was based on user stories as requirements, they would cover what corresponds to use cases (or tasks, see below). These are usually around 30% of the full requirements. They don't cover data needs, system integration, business goals, exit strategy, etc. If agile meant programming small parts in close interaction with the user, we couldn't see it would cure any of the observed damage causes.

In summary, the list of cures is based on expert experience and opinion. The "grounded theory" is that the cures can reduce the causes and consequently the damages. We found 22 cures that together could prevent most of the causes. Table 3 has a column for each cure with indications of the causes it would prevent or reduce. The cures are ordered according to how many causes they could cure. Full descriptions of the cures are available in [12]. We distinguish between two types of cures: *familiar cures*, known by most developers, and *unfamiliar cures*.

Table 3 shows the unfamiliar cures in red. Examples: *Problem-oriented requirements (CA1), monitor remaining work (CC2)*. The other cures are familiar, but were ignored in one or more projects. Examples: *Early prototypes (CC1), pilot test (CE1)*. Half of the cures are familiar.

In table 2, the cures and cure types are mentioned for each damage cause. The table also mentions cure type *misinformed*. It means that the project used a method recommended by government or consultants that actually made things worse. Examples: *Service-oriented architecture (A5)*; *multi-vendor strategy (A6)*.

Table 2 and 3 also indicate *cure unknown*. It means that we at present don't have a suggestion, or that we have a suggestion that only partly cures the problem. Example: How can we prevent *excessive management* or *excessive expert involvement (G8)*?

According to table 3, the cure with the highest potential hit-rate is *problem-oriented requirements*, also called *SL-07 requirements*. Traditional IEEE 830 requirements describe what the system shall do. Problem-oriented requirements describe what the system will be used for and which problems it should remedy. These requirements leave it to the supplier to suggest a solution, be innovative and use what he has already.

User stories also try to avoid specifying what the system shall do, but often they are solutions in disguise. A typical user story is: *As a doctor, I want to see a list of the patient's diagnoses*. There is little difference from a traditional requirement: *The system shall be able to show a list of the patient's diagnoses*. In both cases, we cannot see the context of use.

Problem-oriented requirements are estimated to cure or reduce 9 of the 37 damage causes. As an example, they improve cost estimation because you can compute function points based on them.

An example of a familiar cure that often is ignored, is *early prototypes of the user interface*, combined with *thinking-aloud tests* and *iterative improvement of the prototype (CC1)* [15]. This can be done in a systematic way once you have problem-oriented requirements [8]. It should be done before any programming is made. Otherwise it is too expensive to make larger changes to the user interface. Reference [21] shows that the technique reduced the programming effort in an industrial project and made the product vastly more successful. The technique is estimated to cure 5 of the damage causes.

The technique is well known in UX communities, but UX people complain that they rarely get a chance to use it. Usually they are called in at the end of the project to make the user screens look nice and colorful.

Other cures are obvious once you have identified the cause. One example is F3: *Wrong estimate of human performance* (i.e. how fast the users can work). Once you see the risk,

TABLE 3. Red cures: Unfamiliar techniques. A cause may need more than one cure.

		Cures																						
	Damage causes versus cures Highest hitrates first	CA1 Problem-oriented reqmts (SL-07)	CE1 Pilot test / deployment test	CA3 Scope mgmt & function points	CC1 Early prototypes & test	CG7 Ask expert developers	CA8 Early proof-of-concept (POC)	CC2 Monitor remaining work	CG1 Replan	CA2 Study as is, plan to-be	CF1 Deploy part-by-part	CG2 Small task force with authority	CG8 Problem-oriented contract (SL-07)	CA4 Expert involvement	CG6 Give mgmt IT-insight	CA5 Check technology	CA6 Central database & flex SOA	CA7 Third-party integrate & exit reqs	CB1 Cost checklist	CF2 Follow-up study	CG3 Monitor business case	CG4 Correct risk management	CG5 Constructive lawyers	Cure (partially) unknown	**Total**
ID	**Analysis**	9	6	5	5	5	4	4	4	3	3	3	3	2	2	1	1	1	1	1	1	1	1	4	70
A1	Doesn't identify user needs and win-win	x								x															2
A2	Requirements don't cover customer needs	x																							1
A3	Describes solution in detail. No freedom to supplier	x																							1
A4	Makes heavy demands and believes it is for free	x		x								x													3
A5	Oversells technology, e.g. SOA, web-based					x										x									2
A6	Multi-vendor strategy - supplier independent ! ?																x	x							2
A7	Wants everything at once, e.g. cover all types of debt		x								x														2
A8	Doesn't plan the new work processes	x								x															2
A9	No feasible solution, e.g. data missing, performance dubious.					x	x							x											3
A10	Surprising rule complexity.										x			x											2
	Acquisition																								
B1	Supplier too optimistic - you must lie to win						x																		1
B2	Wrong selection criteria	x																							1
B3	Wrong cost estimates	x		x															x						3
	Design																								
C1	Doesn't ensure usability, even when they know how				x																				1
C2	Designs user screens too late				x																				1
C3	Accepts the solution description without understanding it				x																				1
C4	Cannot see how far the supplier is			x	x	x		x																	4
C5	My way without considering the supplier's way	x																							1
	Programming																								
D1	Supplier accepts expensive reqms interpretation	x							x																2
D2	Surprises with system integration		x				x																	x	2
	Test																								
E1	Deploys the system with insufficient testing		x			x																			2
	Deployment																								
F1	Deploys system with insufficient support/training		x		x					x	x														4
F2	The system is not used as intended		x																	x					2
F3	Wrong estimate of human performance		x				x																		2
	Management																								
G1	No business goals - or forgets them on the way																				x				1
G2	Doesn't reschedule. Assumes the rest can be compressed								x																1
G3	The project grows without anybody noticing			x				x																	2
G4	Risk assessment downplays the danger																					x			1
G5	Financial incentive disappears and parties fight			x				x	x				x												4
G6	Cashes benefit before it is harvested, e.g. sacks too early								x															x	1
G7	Lack of management involvement												x		x										2
G8	Excessive management or expert involvement																							x	0
G9	Too large committees/work groups without											x													1
G10	Excessive user involvement											x													1
G11	Believes law blocks sound approaches, e.g. talking to suppliers																						x	x	0
G12	Insufficient staffing							x					x												2
G13	Doesn't find the root cause					x									x										2

the cure is to measure the human performance early in the project (POC) and as part of a pilot test.

We have observed several cases where developers claimed they used a certain cure, but it didn't help. As a result, they rejected the cure. Closer examination revealed that they didn't understand the cure, but did something else.

In a few cases, we don't know any cure, e.g. how to avoid excessive management involvement.

VII. PROBLEM-ORIENTED REQUIREMENTS AND SL-07

Since *Problem-oriented requirements* can prevent many problems, but are not widely known, we will explain a bit.

SL-07 is an exemplary requirements specification for a health record system. The author wrote it in 2007 on request from the Danish Government as part of their new standard contract, K-02. SL-07 has gradually been improved since then, and now shows realistic examples of all kinds of requirements, including user-task support, data needs, system integration, security, response time, usability, business goals, early-proof-of-concept, exit strategy, development process, etc. All the requirements are problem-oriented: the customer doesn't write what the system shall do, but what he wants to use the system for and the problems he wants to eliminate. The supplier writes the solution he proposes.

The free guide booklet [11] shows the requirements and explains why they are written as they are. The method uses *task descriptions* rather than use cases or user stories. Here is an abbreviated example of requirements written as a task description. The supplier's proposed solution is in red.

Task C10. Perform clinical session

Users: Doctors and nurses.

Start: Contact with the patient or start of a patient conference.

End: When nothing more can be done about the patient right now.

Subtasks:	Proposed solution:
1. Identify the patient	The system can read an electronic tag.
2. Assess the state of the patient. See diagnoses, medications, lab-results . . .	The system shows an overview of diagnoses, medications . . . in a Gantt-like diagram. See . . .
2p. **Problem**. Today clinicians have to log in and out of several systems to see all relevant data.	The system integrates with LabSys . . . and shows the data immediately.
3. Provide services. . .	

At first sight, this looks like a use case, but there are profound differences. A task covers an observable period of time where one user carries out the task without essential interruptions. This is the period to support efficiently. Neither use cases nor user stories use this principle. The left-hand side describes what the user wants to achieve. A good solution takes a large share of this.

It is possible to write problems that the customer has today, without knowing whether a solution exists. Experience shows that suppliers often have a solution the customer couldn't imagine – or they come up with one. See Kuhail & Lauesen for a systematic comparison of use cases and tasks [7].

In this case, the left-hand side is much the same today as it will be with the new system. We are just supporting the present task in a better way. In other cases, the new tasks will be radically different from today. This was for instance the case in the land registry project and the travel card project. In order to make adequate requirements, we have to invent (design) the new user tasks. This would help us cure A8: *Doesn't plan the new work processes* and F1: *Deploys the system with insufficient support/training*.

Traditional requirements are long lists of tiny use cases, user stories or functions the system must provide. However, we cannot see the context in which this functionality will be used. Nor can we see which problems users and other stakeholders have today and expect the system to remedy. As a result, the customer can get a system that meets all requirements, yet doesn't cover his needs. The system is cumbersome to use, doesn't remedy the present problems, and doesn't meet the business goals. With problem-oriented requirements, we can assess how good the solution is.

Problem-oriented requirements are around five times faster to write than traditional requirements and five times shorter, because you don't have to imagine what the system shall do. In particular, you don't have to imagine solutions to your problems.

SL-07 has been used successfully in more than 100 projects. Requirements for the *Y-Foundation* [10] is an example. It is a complex system comprising applicant website, case management, payment management and integration with bank and government sites. The paper shows how it worked in practice, how disputes about *bug* versus *change-request* were resolved, test cases, time spent, and why we were 9 months late.

At its presentation at REFSQ 2018 it was recognized as the first and only publication and discussion of a complex, real-life requirements specification.

VIII. THE SILVER BULLET OR AN EVER-GROWING LIST OF CAUSES?

The 37 causes and 22 cures are of course no magic numbers. They are just what we found in these five projects. As we analyze more projects, more causes will be found, and clever experts will come up with more cures.

When the author started this project, he had a list of ten causes – a nice number. When we had analyzed the first three projects, the author hesitantly accepted that there were 33 causes. Next, the author was lucky to get information from many sources about project four, the debt collection system. It turned out to have two new causes. Project 5, the health record system, revealed just one more cause. Studying [14], the Denver airport case, revealed a cause that was observed in the Police case too, but not recorded: *Doesn't find the root cause*. Now it is cause G13.

So maybe the number won't grow that fast. However, the large number of causes and their varied nature explain why no silver bullet can kill them all.

This study has looked only at large government projects. Will we find different causes and cures in industry? At the seminars the author has given, many industry developers have been present. They too recognize the damage causes and cures, but they explain that industry has two cures that are not practiced in government projects: (1) When serious problems turn up, the project is re-planned. (2) It is no shame to close a project when it businesswise is a healthy idea.

Do small projects have similar damage causes? Probably yes. The Y-project above is an example. It was a 200,000$ non-government project and the only damage was a 9-month delay. The root-causes were: *Surprises with system integration* (D2), *oversells technology* (A5), *financial incentives disappear* (G5).

Each large project had around 15 damage causes, while the Y-Foundation had only 3. We don't know whether it was because it was small or because it used problem-oriented requirements.

IX. CONCLUSION

Accident investigations in 5 damaged ("failing") IT projects have identified 37 damage causes and 22 potential cures. Each project suffered from around 15 of the causes. The same damage causes and cures apply across projects. As new projects are investigated, the lists of causes and cures will probably grow, but slowly.

Some of the cures are well-proven, others are expert suggestions yet to be tried. Even well-proven cures such as *risk management, early prototypes with usability tests* and *problem-oriented requirements* often fail because developers misunderstand them or don't perform them correctly.

A project can use the findings in this way:

1. Apply the cures throughout the project to prevent the damage causes – and thus the damage.
2. During and after the project, identify any new damage causes to prevent in the future and any new cure tried or suggested. Record the results of using the cures.

REFERENCES

[1] *Bonnerup/The Technology Advisory Board: Experiences From Government IT Projects—How to do it Better*, (in Danish), Teknologirådet, Hvidovre, Denmark, Mar. 2001.

[2] N. Cerpa and J. M. Verner, "Why did your project fail?" *Commun. ACM*, vol. 52, no. 12, pp. 130–134, Dec. 2009.

[3] B. Flyvbjerg and A. Budzier, "Why your IT project may be riskier than you think," Harvard Business Review, Sep. 2011.

[4] C. Jones, "Patterns of large software systems: Failure and success," *Computer*, vol. 28, no. 3, pp. 86–87, Mar. 1995.

[5] C. Jones, "Social and technical reasons for software project failures," *Crosstalk*, pp. 4–9, Jun. 2006.

[6] L. Kappelman, R. McKeeman, and L. Zhang, "Early project warning signs of IT project failure: The dominant dozen," *Inf. Syst. Manage.*, vol. 23, pp. 31–36, Fall 2006.

[7] S. Lauesen and M. A. Kuhail, "Task descriptions versus use cases," *Requirements Eng.*, vol. 17, no. 1, pp. 3–18, Mar. 2012, doi: 10.1007/s00766-011-0140-1.

[8] S. Lauesen, *User Interface Design—A Software Engineering Perspective*. Reading, MA, USA: Addison-Wesley, 2005.

[9] S. Lauesen, "Why the electronic land registry failed," in *Proc. REFSQ*. New York, NY, USA: Springer-Verlag, 2012.

[10] S. Lauesen, *Problem-Oriented Requirements in Practice—A Case Study* (Lecture Notes in Computer Science), vol. 10753, E. Kamsties, Ed. Cham, Switzerland: Springer, 2018, pp. 3–19, doi: 10.1007/978-3-319-77243-1_1.

[11] S. Lauesen. (2019). *Problem-Oriented Requirements SL-07—Guide and Contract—Version 7*. [Online]. Available: http://www.itu.dk/people/slauesen/SorenReqs.html

[12] S. Lauesen. (2020). *Damages and Damage Causes in Large Government IT Projects, Version 11*. pp. 1–59. [Online]. Available: http://www.itu.dk/people/slauesen/SorenDamages.html

[13] L. McLeod and S. G. MacDonell, "Factors that affect software systems development project outcomes: A survey of research," *ACM Comput. Surv.*, vol. 43, no. 4, pp. 1–56, Oct. 2011.

[14] R. Montealegre and M. Keil, "De-escalating information technology projects: Lessons from the Denver international airport," *MIS Quart.*, vol. 24, no. 3, pp. 417–447, Sep. 2000.

[15] J. Nielsen, "The usability engineering life cycle," *Computer*, vol. 25, no. 3, pp. 12–22, Mar. 1992.

[16] O. Metcalf-Rinaldo, S. M. Jensen. (Jul. 2017). *Learnings from the implementation of Epic*, pp. 1–77. [Online]. Available: http://www.itu.dk/people/slauesen/SorenDamages.html

[17] *Beretning Til Statsrevisorerne om Det Digitale Tinglysningsprojekt*, (in Danish), Rigsrevisionen, Copenhagen, Denmark, Aug. 2010, pp. 1–74.

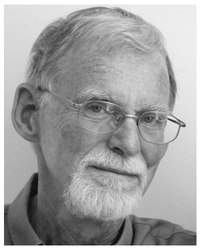
[18] *Beretning Til Statsrevisorerne om Rejsekortprojektet*, (in Danish), Report to the State Auditors on the Travel Card project, Rigsrevisionen, Copenhagen, Denmark, Jun. 2011, pp. 1–47.

[19] *Beretning Til Statsrevisorerne om Politiets It-System POLSAG* (Report to the State Auditors on the police IT system POLSAG), (in Danish), Rigsrevisionen, Copenhagen, Denmark, Mar. 2013, pp. 1–56.

[20] *Beretning til Statsrevisorerne om SKATs Systemmodernisering* (Report to the State Auditors on Tax's System Renovation), (in Danish), Rigsrevisionen, Copenhagen, Denmark, Jan. 2015, pp. 1–34.

[21] O. Vinter and S. Lauesen, "Preventing requirement defects: An experiment in process improvement," *Requirements Eng. J.*, vol. 6, pp. 37–50, 2001.

[22] L. Wallace and M. Keil, "Software project risks and their effect on outcomes," *Commun. ACM*, vol. 47, no. 4, pp. 68–73, Apr. 2004.



SOREN LAUESEN received the M.Sc. degree in mathematics & physics from the University of Copenhagen, Denmark, in 1965, and the B.Com. degree from the Copenhagen Business School, Denmark, in 1979. From 1962–1973, he worked as a developer/department manager at Regnecentralen, Denmark (Danish computer manufacturer). From 1969–1972, he was a part-time associate professor at the University of Copenhagen, and co-founder of the first computer science education in Denmark. From 1973–1976, he was co-founder of the software development department at Brown Boveri, Copenhagen (now ABB). From 1976–1979, he was a visiting professor at the University of Copenhagen, and department manager for the last two years. From 1979–1985, he was co-founder of the software development center at NCR, Copenhagen. From 1985–1999, he was a professor at Copenhagen Business School, and co-founder of the combination education in business and computer science. During this time, he also served as Head of Department from 1992 to 1996. In 1999, he became a professor at the IT University of Copenhagen, where he served for twenty years. Since September of 2019, he has been Professor Emeritus at the IT University of Copenhagen.