Explanations in Software Engineering: The Pragmatic Point of View

Jan De Winter

Abstract This article reveals that explanatory practice in software engineering is in accordance with pragmatic explanatory pluralism, which states that explanations should at least partially be evaluated by their practical usability. More specifically, I offer a defense of the idea that several explanation-types are legitimate in software engineering, and that the appropriateness of an explanation-type depends on a) the engineer's interests, and b) the format of the explanation-seeking question he asks, with this format depending on his interests. This idea is defended by considering examples that are representative for explanatory practice in software engineering. Different kinds of technological explanation are spelled out, and the dependence of their appropriateness on interests and question-formats is extensively illustrated.

1. Introduction

Research on scientific explanation shows that there is not one kind of explanation that guarantees maximal explanatory power. Different kinds of explanation are legitimate (e.g., Pettit 1996; Weber and Van Bouwel 2002). A question that then arises, is whether one can randomly choose a kind of explanation without running the risk of choosing a deficient explanation-type ('anything goes'). If not, one can wonder what the nature is of the factors that determine which explanation-type is best. Philosophical analyses indicate that the following factors can influence the appropriateness of an explanation-type: the information looked for (Pettit, 1996), the explanation-seeking question (e.g., van Fraassen 1980; Van Bouwel and Weber 2008; De Langhe 2009), and the function the explanation should serve (e.g., Weber 1999; Weber, Van Bouwel and Vanderbeeken 2005; Weber and Vanderbeeken 2005).

This article is dedicated to the question of how explanation-types, formats of explanation-seeking questions, and interests are related to each other in software engineering. An examination of these relations contributes to our understanding of how humans acquire knowledge, and will reveal that the pragmatic explanatory pluralistic framework, which states that explanations should

at least partially be evaluated by their practical usability, can be expanded to technological explanatory practices.

2. Theoretical background

2.1. Formats of explanation-seeking questions

According to the erotetic model of explanation, which I will take for granted here, explanations can be thought of as answers to why-questions. A framework of question formats that will cover all explanation-seeking questions we will meet in the next sections, is formulated in Van Bouwel and Weber (2002), and contains the following formats (each accompanied by an example):

(P-contrast) Why does object *a* have property P, rather than property P'?[1]
e.g., Why does the software procedure *GenerateSchedule* have a very long worst-case execution time, rather than a short worst-case execution time?

(T-contrast)   Why does object *a* have property P at time t_1, but property P' at time t_2?
e.g., Why does the software procedure *GenerateSchedule* have a very long worst-case execution time, while ten minutes ago, it had a very short worst-case execution time?

(O-contrast)   Why does object *a* have property P, while object *b* has property P'?
e.g., Why does the software procedure *GenerateSchedule1* have a very long worst-case execution time, while the software procedure *GenerateSchedule2* has a very short worst-case execution time?

(plain fact) Why does object *a* have property P?
e.g., Why does the software procedure *GenerateSchedule* have a very long worst-case execution time?

Questions that fit one of the first three formats, are contrastive questions. According to Van Bouwel and Weber (2008), such questions can be motivated by (amongst others) surprise and a therapeutic or preventive need. A question is motivated by surprise if the reason for asking it, is that one wants to explain why things turn out to be different from what we expected. A question is motivated by a therapeutic or preventive need if it is motivated by a desire to know why things are different from how we want them to be, and by the need to know how the correspondence between the actual and the desired state of affairs can be achieved. Possible incentives for asking a plain fact question, are sheer intellectual curiosity, and a desire to causally connect object *a* having property P to events with which we are more familiar (Van Bouwel and Weber 2008, p. 170). Later in this article, we will meet more possible incentives for raising explanation-seeking questions.

2.2. Pragmatic explanatory pluralism

Since engineering is a task-oriented discipline (Pitt 2001), we can expect the explanatory power of explanations in this discipline to depend on pragmatic criteria. An engineer starts with a

---

[1] P and P' are mutually exclusive properties.

predetermined purpose, and when he has to choose between different options (such as which explanation-type to use), he should select the option that best serves this purpose. Therefore, it should not come as a surprise that I will defend a pragmatic point of view regarding explanation-types in software engineering. More specifically, I will defend a pragmatic explanatory pluralism that is committed to three theses:

1. There is more than one legitimate kind of explanation in software engineering.[2]
2. Which kind of explanation should be preferred by a software engineer, depends on a) his reasons for seeking an explanation (his interests) and b) the format of the explanation-seeking question he asks.
3. The format of the explanation-seeking question a software engineer asks, depends on his interests.

3. Example

An example I consider representative of software engineering tasks, is the following. There are two conferences in a sports league, each containing three teams, as is depicted in *Scheme 1a*. Every team has to play at least one game against every other team. Teams of the same conference have to play two times against each other, while two teams of different conferences have to compete only once. *1b* presents all the games that have to be played.

Scheme 1 – Teams and games					
a	Conference 1	Conference 2			
	A	D			
	B	E			
	C	F			
b	A – B	B – C	C – D	D – E	E – F
	A – B	B – C	C – E	D – E	E – F
	A – C	B – D	C – F	D – F	
	A – C	B – E		D – F	
	A – D	B – F			
	A – E				
	A – F				

A schedule has to be generated that meets the following conditions: 1) the schedule contains no more and no less than the games in *Scheme 1b*, 2) a team cannot play more than one game per day, and 3) the games are distributed over as few days as possible (1 and 2 take priority over 3).

A software procedure should fulfill the task of generating the schedule. This software procedure should do this in such a way that the content of the schedule is as random as possible. In other words: if n is the number of schedules that meet conditions 1 to 3, then for any schedule that meets conditions 1 to 3, the chance that it is generated by the procedure should be equal to

---

[2] I consider an explanation-type to be legitimate if there is at least one interest that is best served by an explanation of that type.

1/n. The user should be able to execute the procedure by pressing a button, but only after all games on the current schedule have been finished. When the current schedule contains games that haven't been played yet, pressing the button should cause the appearance of the message "You can only generate a new schedule after finishing all games on the current schedule."

Several problems might occur while developing such a computer program, and a strategy I consider very useful in solving some of these problems, is to ask certain explanation-seeking questions. The usefulness of this strategy is made clear in the next sections, in which I give examples of interests and explanation-seeking questions that might turn up in the process of writing and debugging software, and pay attention to the nature of the explanations that answer these questions.

4. Contrastive questions

In order to come to a first explanation-seeking question, I suggest an algorithm that might allow our software procedure to randomly generate a schedule that satisfies the three conditions mentioned earlier. The first step of the algorithm is the random permutation of the set of games presented in *Scheme 1b*.[3] We can call the resulting array of games *Array1*. The second step fills a calendar with days on which games will be played. For each day *D*, the procedure runs through *Array1*, checking for each element whether it can be added to *D*. An element can only be added if two requirements are met. The first is that it is a game, and the second is that for both teams, this game should be the first they have to play on *D*. If and only if these two requirements are met, the game is added to *D* and removed from *Array1* (replaced by something that is not a game). When all elements of *Array1* have been checked, *D* is added to the calendar, and if *Array1* still contains games, a new day is created. This process is repeated until all games are part of the calendar. Then, the calendar is converted into a schedule that is accessible for users.

It is possible that one has made some mistakes in implementing this algorithm. In that case, the procedure will probably not have the anticipated result. Suppose an engineer has tried to implement the algorithm, but made a typing error in the program line that should bring about the adding of a game to day *D* if possible. When he tests his procedure, he finds out that it does not work properly. Of course, the engineer does not know what causes the perceived error (otherwise, he would have corrected the typing error before testing). In order to debug the procedure, he asks why the procedure led to the error, and not to the desired result (P-contrast question). The answer to this question is that the procedure should contain a program line that actually adds a game to the calendar if possible, while his procedure does not. This explanation is complete if the explainer assumes that the procedure would have worked properly if it contained such a program line. It has the following format:

Object *a* has property P, rather than P' because it does not have the properties $D_1, …, D_n$ (with $n \geq 1$).

It is assumed that object *a* would have property P' if it had properties $D_1, …, D_n$. I will call the combination of this assumption and the explanation format, the P-contrast explanation-type or the PCE-type.

---

[3] An algorithm that can be used to shuffle the games, is Knuth's modern version of the Fisher-Yates algorithm (Radu 2008).

In order to solve P-contrast questions, it can sometimes be useful to ask other explanation-seeking questions. To demonstrate this, we can look at how our engineer attains a button that allows the user to generate a new schedule if and only if all games on the current schedule have been finished. Suppose a button with the label 'generate schedule' has already been inserted. Suppose further that when the last game of the regular season is finished, the value of a variable V stocked at a certain location in a database, turns from 0 into 1. When a new schedule is generated, the value of V turns from 1 into 0. Because up till now, all cases in which the value of V is equal to 1, correspond to the cases in which all games on the schedule have been finished, and all cases in which the value of V differs from 1, correspond to the cases in which the schedule still contains games that have not been played yet, the consequences of pressing the button can be determined on the basis of whether or not the value of V is equal to 1. The engineer knows this and assigns to the pressing of the button a procedure that generates a new schedule if and only if the value of V is equal to 1. While testing his program, he finds out that the pressing of the button has the desired consequences.

Later, the engineer decides to extend his program with some post-season events. The games at the post-season events should be added to the schedule, and only when all these games are over, the user should be able to generate a new schedule. During a first attempt at implementing the extension, the engineer deletes the program line that shifts the value of V from 0 into 1 when the last game of the regular season is finished, but forgets to insert a line that brings about this shift when the last game of the last post-season event is finished. He discovers that the pressing of the 'generate schedule' button never leads to a new schedule, even when there are no non-played games on the schedule remaining. An answer to the T-contrast question 'Why does the program have, at this moment t_2, the property of never generating a new schedule when the 'generate schedule' button is pressed, while at time t_1 (before extending the program), it had the property of generating a schedule if 1) the 'generate schedule' button was pressed and 2) all games on the schedule were finished?' can help him solve this problem.

The T-contrast can be explained by the fact that at t_2, a program line that leads to the shift of V's value when the last game on the schedule is finished, was present, while such a program line is absent at t_1. This explanation can be fitted into the following format (without changing its meaning):[4]

> Object *a* has property P at time t_1 but P' at time t_2 because it had properties D_1, …, D_n (with $n \geq 1$) at t_2, while these properties were absent at t_1.

In combination with the assumption that object *a* would have property P at time t_1 if it had properties D_1, …, D_n at t_1, I will call this format the TCE-type of explanation (with TCE standing for T-contrast explanation).

The explanation of the TCE-type brings the engineer closer to an answer to the question why the pressing of the button does not lead to a new schedule when it should, that is, when all games on the schedule are over (P-contrast question). The answer is that the program should contain a program line that changes the value of V into 1 when the last game on the schedule is finished,

---

[4] The explanation can be reformulated into: The program has, at this moment t_1, the property of never generating a new schedule when the 'generate schedule' button is pressed, while at time t_2, it had the property of generating a new schedule if 1) the 'generate schedule' button was pressed and 2) all games on the schedule were finished, because at t_2, it had the property of containing a program line that leads to the shift of V's value when the last game on the schedule is finished, while this property is absent at t_1.

while it doesn't (PCE). The benefit in asking the T-contrast question 'Why doesn't object a work properly at this moment t_1, while it did at time t_2?', is that it draws attention to those factors that have been changed during interval $[t_2, t_1]$. Since the program worked properly at time t_2, these factors are more likely to cause the malfunctioning, as is illustrated by the previous example. The fact that the value of V did not get changed anymore, caused the malfunctioning, and not, say, a typing error in the program line that converts the calendar into an accessible schedule. One need not pay attention to this last possibility because if this program line would not perform its function, the program would not have worked at time t_2 either, while it did. A T-contrast question can thus function as a tool to address the engineer's attention to those factors that are most likely to be part of an accurate answer to a P-contrast question.[5]

An O-contrast question can serve the same purpose. Suppose the engineer did not extend the original program with some post-season events, but started a new program, in which he reuses the program code that allows for the random generation of a schedule. The same error as with the T-contrast might arise (a button does not lead to a new schedule when it should). Given that the engineer now has two programs, one that does not allow for the generation of a schedule at the appropriate times, and one that does, he can ask the O-contrast question 'Why does the new program a have the property P of never allowing for the generation a new schedule, while the original program b has the property P' of allowing for the generation of a new schedule at the appropriate times?' This question draws the engineer's attention to those factors that a and b do not share (and makes abstraction of all other factors), which are the factors that are most likely to be part of an accurate answer to the P-contrast question 'Why does object a have property P, rather then the ideal property P'?' Thus, both T- and O-contrast questions can facilitate the debugging process.

The preferential answers to O-contrast questions are O-contrast explanations (OCE). An explanation of the OCE-type has the following format:

Object a has property P, while object b has property P' because b has properties $D_1, \ldots, D_n$ (with $n \geq 1$) which object a does not have.

An explanation that fits this format, is of the OCE-type if it is assumed that object a would have property P if it had properties $D_1, \ldots, D_n$.

5. Plain facts

Not only contrastive questions can help one to solve P-contrast questions. To illustrate this, we can return to an earlier stage in the development of the computer program, that is, before our engineer assigned a procedure to the button with the label 'generate schedule', and after he corrected the typing error in the program line that should bring about the adding of a game to the calendar if possible. At this stage, the engineer is still testing the (alleged) implementation of the algorithm I described at the beginning of section 4. While doing this, he faces a problem: in two tests, his generator generates the two schedules presented in *Scheme 2*. Because *Schedule a*

---

[5] The fact that a T-contrast explanation helps the explainer to answer a P-contrast question, does not mean that a P-contrast explanation is in itself insufficient to answer such a question. The P-contrast explanation is a complete answer to the P-contrast question, and this answer should not contain any additional information. The T-contrast explanation is the preferential and complete answer in another context, that is, when a T-contrast question is under consideration.

contains less days then *Schedule b*, the engineer knows that his procedure cannot guarantee that a generated schedule meets the third condition (the games are distributed over as few days as possible).

Scheme 2 – Schedules								
Schedule a								
Day 1	Day 2	Day 3	Day 4	Day 5	Day 6	Day 7		
E – F C – D A – B	C – E D – F A – B	A – E C – F B – D	A – D E – F B – C	D – F A – C B – E	A – C D – E B – F	B – C D – F A – E		
Schedule b								
Day 1	Day 2	Day 3	Day 4	Day 5	Day 6	Day 7	Day 8	Day 9
B – C D – E A – F	B – E C – D	A – C E – F B – D	A – B D – E C – F	A – D E – F B – C	A – C D – F	A – E D – F	B – F C – E	A – B

The P-contrast question arises: 'Why does the software procedure, when executed several times, occasionally result in schedules with more than seven days, instead of always resulting in schedules with exactly seven days?'[6] To know where to search for an answer to this question, the engineer wonders what causes the malfunctioning: a problem with the algorithm, or a problem with the conversion of the algorithm into a software procedure. If the algorithm is problematic, he should develop a new algorithm, if not, he should correct the implementation error(s). So in order to know how to continue the debugging process, the engineer needs to know whether his algorithm is problematic. This he can find out by trying to explain the shortcomings of *Schedule b* by clarifying how the execution of the algorithm could have caused them. If such an explanation is possible, the algorithm is problematic, if not, there's something wrong with its implementation.

One of the shortcomings of *Schedule b* is that the second day contains only two games.[7] The engineer then asks the plain fact question 'Why does the second day of *Schedule b* contain only two games?', and tries to answer it by an explanation that clarifies how the execution of the algorithm led to this plain fact (explanation by algorithm or EA). The format of such an explanation is:

Object *a* has property P because:
At step n, event e_n occurred
(Step n + 1 was the next step because condition $c_{(n+1)}$ was satisfied)
At step n + 1, event $e_{(n+1)}$ occurred

---

[6] Seven days is the minimum of days in a schedule, since each team has to play seven games (see *Scheme 1b*), and a team cannot play more than one game per day (second condition). The fact that a schedule containing seven days is possible, is shown by *Schedule a*.
[7] Each day should contain three games. We know this because 21 games (see *Scheme 1b*) should be distributed over seven days (see previous note), and one day cannot contain more than three games. If there would be more than three games (each between two teams) on a day, there should be more than six teams, since one team cannot play more than one game per day (second condition), and this is not the case (see *Scheme 1a*).

(Step n + 2 was the next step because condition $c_{(n+2)}$ was satisfied)
At step n + 2, event $e_{(n+2)}$ occurred
...
(Step n + m was the next step because condition $c_{(n+m)}$ was satisfied)
At step n + m, event $e_{(n+m)}$ occurred
(with n ≥ 1; and m ≥ 0)

An explanation by algorithm presumes that the execution of a certain algorithm caused the fact that object *a* has property P, and explains this fact by referring to events that 1) were instructed by the algorithm, and 2) are relevant to object *a*'s acquiring property P. The reason why the sentences that explain why a certain step was next, are bracketed, is that they are only necessary in cases in which a condition, specified by the algorithm, had to be satisfied for this step to be next.

An explanation by algorithm of the fact that the second day of *Schedule b* contains only two games, might go as follows. After the second game was added to the second day of the calendar, the focus moved to the next game of *Array1*. Because this game could not complete the second day without leading to a schedule that doesn't meet the three aforementioned conditions,[8] it wasn't added to the second day. Then, the focus moved to the next game of *Array1*. This game could not complete the second day either, so that again, the focus moved to the next game without the second day being extended with a third game. This process continued until all games of *Array1* had been checked, after which the composition of the third day started. Thus, no third game was added to the second day before the process of constituting this day ended. Similar explanations can elucidate why days 6 to 9 contain less than three games.

Because the shortcomings can be explained by the execution of the original algorithm, our engineer does not have to control whether his software procedure is a correct implementation of the original algorithm, and can refine his P-contrast question into 'Why does the algorithm, when executed several times, occasionally result in schedules with more than seven days, instead of always resulting in schedules with exactly seven days?'[9] An answer to this question is that the algorithm permits that games that together exclude the possibility of a third or a second game being added to one of the days that do not yet contain three days, are added to the definitive calendar. For example, the first two games of the second day of *Schedule b* exclude together with the third game of the first day, that a third game is added to the second day.

An algorithm that avoids this mistake, is the following. First, the interconference games are randomly shuffled. Next, for each game of the resulting array, it is checked whether it can be added to the first day. If so, the game will be added to the first day, if not, it will be the first game of a day that does not contain games yet. This step results in a calendar consisting of seven days, with the first day containing three interconference games, and the next six days containing each only one (interconference) game. Then, days 2 – 7 are completed by those inconference games that can be added to them without leading to a schedule that does not meet the three aforementioned conditions. Next, for each day, the three games on it are randomly shuffled, and so are the days of the calendar. This calendar is converted into a schedule that is accessible for

---

[8] All games other than A – F could not complete the second day because they contain teams that already had a game to play on the second day. A – F could not complete the second day either, because it was added to the first day, and interconference games should appear only once on the schedule.
[9] Notice that by answering this question, one also answers the original, non-refined *P-contrast* question.

users. When this algorithm is implemented correctly, one obtains a software procedure that satisfies the requirements stated in section 3.

Answering plain fact questions can also serve other purposes than satisfying the desire to know whether a certain algorithm caused an error. One such purpose is related to the development of a new program by using components of an outdated program. If our engineer prefers creating a new program a over updating an old program b, and he wants to reuse those components of b that together have the function of generating a new schedule, he has to know which part of the original program b fulfills this function. The corresponding knowledge-seeking question can be reformulated into the plain fact explanation-seeking question 'Why does the original program b have the property P' of allowing for the generation of a new schedule?'

The answer the engineer is looking for, is a set of reusable software components that are non-redundant parts of the set of all factors that bring about the explanandum. The reason why he wants his answer to contain only reusable software components, and not, say, physical laws, non-reusable software, etc., is that he wants to obtain software that he can use while writing the new program. The reason why the software components should not be redundant, is that the engineer does not want to reuse more than needed.

In most (if not all) cases in which the question of why program b has property P', is motivated by the desire to pass on property P' to a new program by reusing parts of b, the optimal answer will be a Mackie-like explanation (ME). According to Mackie, "what is typically called a cause, is an inus condition, or an individual instance of an inus condition …" (Mackie, 1974, p. 64), with an inus condition being an 'insufficient but non-redundant part of an unnecessary but sufficient condition' for the effect to occur. For example, if $(D_1 \wedge D_2) \vee (D_3 \wedge D_4)$ is necessary for object b to have property P', while both $D_1 \wedge D_2$ and $D_3 \wedge D_4$ are sufficient for b to have property P', then D_1, D_2, D_3 and D_4 are possible causes of the fact that object b has property P'. An explanation of the ME-type consists of indicating some of the factors that are insufficient but non-redundant parts of a conjunction that is not only unnecessary and sufficient for object b to have property P', but also true. The format of this kind of explanation is:

Object b has property P' because of factors D_1, …, D_n (with $n \geq 1$).

We can now indicate why in most cases in which the question 'Why does program b have property P'?' is motivated by the desire to pass on property P' to a new program a by reusing parts of b, the optimal answer will be of the ME-type. We already saw why the explanation should only refer to reusable software components that are not redundant. These software components will not be sufficient for the program to work in a certain way, because there will always be some non-software-related background conditions (e.g., certain physical laws being active) that have to be met for any computer program to work. The software components are part of a sufficient condition for the occurrence of the explanandum, because, if otherwise, the explanandum would not occur (while it does). In most cases, this sufficient condition will not be necessary to cause the explained effect, since this effect could also be caused by using other algorithms, or other implementations of the same algorithm. The software components that constitute the explanation can thus be considered to be insufficient but non-redundant parts of an unnecessary but sufficient condition for the original program to have the property one wants to pass on to the new program. That the unnecessary but sufficient condition should be true, is evident, since the engineer wants his explanation to point at factors that actually (and not just possibly) caused the fact that program b has property P'.

6. Summary

Based on the examples I offered in sections 4 and 5, we can construct *Scheme 3*. Each row connects three elements with each other: an interest, a format of an explanation-seeking question, and an explanation-type. The scheme demonstrates that several explanation-types are legitimate, since the third column, that indicates which kind of explanation has most explanatory power in a certain case, contains different explanation-types.

Scheme 3 – Interests, question-formats and explanation-types		
Interest[a]	Format of explanation-seeking question	Kind of explanation
Debugging	P-contrast	PCE
Answering P-contrast question	T-contrast	TCE
Answering P-contrast question	O-contrast	OCE
Knowing whether the algorithm caused the error (in order to refine P-contrast question)	Plain fact	EA
Knowing which parts of a computer program *b* with property P' should be reused to pass on P' to program *a*	Plain fact	ME

[a] Other interests can be added to each row. To the first row can be added 'designing a computer program that satisfies certain demands', to the second 'designing a computer program that satisfies certain demands', 'debugging', and so on. To keep the scheme clear however, I have added only those interests I consider to be the most proximate cause of the explanation-seeking question.

The examples show that the format of the explanation-seeking question that an engineer asks, depends on his reasons for asking that question (his interests). For instance, if an engineer wants to debug a software procedure, this will naturally lead him to asking a P-contrast question. The format of this question influences the appropriateness of different explanation-types. For instance, the chance that the best answer to a P-contrast question is of the PCE kind, is much (if not infinitely) larger than the chance that it is of, say, the EA kind, while the EA kind of explanation has more chance of being preferential when the question asked is a plain fact question.

Further, the examples show that which kind of explanation will have most explanatory power, depends on the engineer's interests. The fact that an EA kind of explanation is to be preferred in one case, while an ME kind of explanation is to be preferred in another, can be explained by the different interests in both cases. The EA style will best satisfy the desire of knowing whether an algorithm caused an error, while the ME style will best satisfy the desire of knowing which parts of a computer program *b* with property P' should be reused to pass on P' to program *a*.

7. Further research

In order to avoid misunderstandings, I want to call attention to a restriction of the survey of explanation-types I offered. Some technological explanation-types have been presented in Kroes

(1998), and de Ridder (2007). Both Kroes and de Ridder study technological explanations to get a grip on the relation between an artifact's function and its physical structure. While Kroes reflects on explaining the function of an artifact by referring to a with structural-physical concepts described structure (including not only the design of the artifact, but also the relevant physical phenomena, and the actions necessary for the artifact to perform its function), de Ridder concentrates on explaining the behavior of artifacts. De Ridder offers two explanatory strategies: top-down (TD) and bottom-up (BU). Because the corresponding explanation-types are a bit more complicated than the explanation-type proposed by Kroes, I will not fully explicate them here. It suffices to say that both kinds of explanation connect, though in a quite different way, an artifact's overall behavior to physical components of the artifact. De Ridder states that "[t]he appropriateness of a TD or BU explanation depends on context and the specific explanatory question being asked" (de Ridder 2007, p. 234).

However, even though the explanation-types proposed by Kroes and de Ridder may serve their philosophical functions very well, I do not think they have a significant role to play in software engineering practice (because a software developer can do his job perfectly well without knowing anything about the physical basis of software), which is why I did not mention them in the previous sections. This does not mean that I definitely reject the idea that explanations of these types are most suited to fulfill certain functions in software development. Nor do I want to exclude the possibility that other kinds of explanations that have not been mentioned in the previous sections (e.g., teleological and causal explanation of product information as described in Taura and Kubota 1999, Hempel's covering law model, etc.), are useful tools for software engineers. My omission of spelling out all explanation-types that can be legitimate in software engineering, is justified because it was beyond the scope of this article to offer more explanation-types than necessary to prove my point, that is, that software engineering is a pragmatic explanatory pluralistic discipline in the way outlined above. I do not consider this restriction to be problematic, for one can easily accept explanation-types that were not included in my analysis, without discarding my version of pragmatic explanatory pluralism.

My version of pragmatic explanatory pluralism can be generalized to the idea that several epistemic guidelines are legitimate, and that the appropriateness of a guideline depends on a) the interests of the person that seeks knowledge, and b) the format of the knowledge-seeking question he asks, with this format depending on his interests. Further research should reveal whether this more general pragmatic idea is accurate with respect to epistemic practices in disciplines ranging from software engineering and other technological endeavors to the natural sciences.

References

De Langhe, R. (2009). Trading off explanatory virtues. In Weber, E., Libert, T., Marage, P., and Vanpaemel, G. (eds.), *Logic, Philosophy and History of Science in Belgium. Proceedings of the Young Researchers Days 2008*. Brussels: Koninklijke Vlaamse Academie van België.

De Ridder, J. (2007). *Reconstructing Design, Explaining Artifacts*. Doctoral thesis, Delft University of Technology, Delft.

Kroes, P. (1998). Technological Explanations: The Relation between Structure and Function of Technological Objects. *Techné, 3*(3), 18-34.

Mackie, J. (1974). *The Cement of the Universe: A Study of Causation*. Oxford: Clarendon Press.

Pettit, P. (1996). *The Common Mind*. New York: Oxford University Press.

Pitt, J. C. (2001). What Engineers Know. *Techné, 5*(3), 17-30.

Radu, S. (2008). *The Fisher-Yates shuffle algorithm*. Paper presented at the Mini Conference on Computing Algorithms, Bryn Mawr.

Taura, T., and Kubota, A. (1999). A Study on Engineering History Base. *Research in Engineering Design, 11*(1), 45-54.

Van Bouwel, J., and Weber, E. (2002). Remote Causes, Bad Explanations. *Journal for the Theory of Social Behaviour, 32*(4), 437-449.

Van Bouwel, J., and Weber, E. (2008). A Pragmatist Defense of Non-relativistic Explanatory Pluralism in History and Social Science. *History & Theory, 47*(2), 168-182.

Van Fraassen, B. C. (1980). *The Scientific Image*. Oxford: Clarendon Press.

Weber, E. (1999). Unification: What Is It, How Do We Reach it and Why Do We Want it? *Synthese, 118*(3), 479-499.

Weber, E., and Van Bouwel, J. (2002). Symposium on Explanations and Social Ontology 3: Can We Dispense with Structural Explanations of Social Facts? *Economics and Philosophy, 18*(2), 261-277.

Weber, E., Van Bouwel, J., and Vanderbeeken, R. (2005). Forms of Causal Explanation. *Foundations of Science, 10*(4), 437-454.

Weber, E., and Vanderbeeken, R. (2005). The Functions of Intentional Explanations of Actions. *Behavior and Philosophy, 33*, 1-16.