# Scope Determined (D) Versus Scope Determining (G) Requirements: A New Significant Categorization of Functional Requirements

Daniel Berry[1], Márcia Lucena[2], Victoria Sakhnini[1], and Abhishek Dhakla[1]

[1] Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario N2L 3G1, Canada
{dberry,vsakhnini,adhakla}@uwaterloo.ca

[2] Department of Computer Science and Applied Mathematics
Universidade Federal do Rio Grande do Norte
Natal, RN, Brazil
marciaj@dimap.ufrn.br

**Abstract. Context**: Some believe that Requirements Engineering (RE) for a computer-based system (CBS) should be done up front, producing a complete requirements specification before any of the CBS's software (SW) is written. **Problem**: A common complaint is that (1) new requirements *never* stop coming; so upfront RE goes on forever with an ever growing scope. However, data show that (2) the cost to modify written SW to include a new requirement is at least 10 times the cost of writing the SW with the requirement included from the start; so upfront RE saves development costs, particularly if the new requirement is one that was needed to prevent a failure of the implementation of a requirement already included in the scope. The scope of a CBS is the set of requirements that drive its implementation. **Hypothesis**: We believe that both (1) and (2) are correct, but each is about a different category of requirements, (1) scope determininG (G) or (2) scope determineD (D), respectively. **Results**: Re-examination of the reported data of some past case studies through the lens of these categories indicates that when a project failed, a large number of its defects were due to missing D requirements, and when a project succeeded, the project focused its RE on finding all of its D requirements. **Conclusions**: The overall aim of the research is to empirically show in future work that focusing RE for a *chosen* scope, including that of a sprint in an agile development, on finding all and only the D requirements for the scope, while deferring any G requirements to later releases or sprints, allows upfront RE (1) that does not go on forever, and (2) that discovers all requirements whose addition after implementation would be wastefully expensive, wasteful because these requirements *are* discoverable during RE if enough time is devoted to looking for them.

**Keywords:** Agile methods, Cost to repair defects, Defect tickets, Empirical studies, Exceptions and variations, Missing requirement, Requirements specification, Scope, Scope-determined requirement, Scope-determining requirement, Software development lifecycle, Sprint, Upfront requirements engineering, Waterfall methods

## 1 Introduction

The current great debate [3, 8, 13, 15, 17, 19, 22, 23, 28, 29, 36, 38] in Requirements Engineering (RE) is whether requirements for a computer-based system (CBS)

1. should be identified upfront before design and coding begin, as in the waterfall lifecycle [30], or
2. should be identified incrementally, interleaved with design and coding of requirements identified so far, as in the spiral or agile lifecycles [2, 10].

Here, "identifying requirements for a CBS up front" means "identifying requirements for the CBS in their entirety".

The argument for identifying requirements upfront is that catching and fixing a requirement defect, i.e., a missing or incorrect requirement, during coding costs 10 times the cost of catching and fixing it during upfront RE [9, 31]. Thus, developing a CBS using waterfall methods, with requirements determined for the entire CBS up front before beginning any coding, leads to the shortest overall development time [5, 6, 9, 12, 16, 27, 34].

The arguments for identifying requirements incrementally are that

- requirements never stop coming [2, 6]; if design and coding do not start until *all* requirements are identified, design and coding will *never* start, and
- many requirements change as more and more of a CBS is developed and as the world changes as a result of the CBS's being used [20, 21]; many requirements that were identified before will be thrown out; and the time spent identifying these thrown-out requirements is wasted!

Thus, we should develop CBSs using agile methods, with requirements determined for each sprint of coding only at the beginning of the sprint.

Attempts to settle the debate with empirical data have failed. Empirical studies go both ways and are overall inconclusive [3,13,19,23,28,36]. Consequently, the choice of CBS development lifecycle, upfront RE or agile, to use in a CBS development project is made on the basis of gut feelings informed by experience and a recognition that if a project does something different from what is established practice, and the project fails, the heads of the project's decision makers will roll.

The reason that data have not decided the debate is that each side is right!

A1. Requirements *do* never stop coming; and many requirements *do* change, resulting in wasted effort.

A2. There *are* a lot of requirements defects that *can* be found and fixed early if one is spending enough time doing RE, and a complete requirements specification (RS) for a CBS *dramatically reduces* the incidence of expensive-to-fix requirement defects that appear in the code for the CBS.

We believe that the two competing arguments, A1 and A2, are talking about two different kinds of requirements, respectively:

K1. One kind often cannot be identified until users are trying some version of the CBS and notice its necessity, and it is best handled incrementally so that when it is identified, it is less likely to change [2].

K2. The other kind can be identified before design and coding if enough time is devoted to RE, and it is wasteful to leave this requirement to be found only later in the lifecycle when it is more expensive to fix [6].

The empirical studies are inconclusive because none of them distinguishes these particular different kinds of requirements.

We have identified a new binary categorization of new requirements being considered for addition to a CBS:

C1. The first category of requirement is a *scope determininG (G)* requirement, and
C2. the second category of requirement is a *scope determineD (D)* requirement.

Here, the *scope* of a CBS is the set of requirements — a.k.a. use cases or features — it implements. This categorization has been identified in the past under different names. For example, among use cases, a variation or exception of another use case is a D requirement, but a new, independent use case is a G requirement. New are the names of the categories, which are more suggestive of

- how the categorization of a requirement can be done and
- how knowledge of the categorizations of candidate requirements for a CBS can be used during RE for the CBS and during its subsequent development.

Maybe, the data will be more conclusive for each category of requirements.

This article shows empirical evidence from past case studies, all done for other purposes, that

- A1 = K1 = C1, and G requirements can be handled incrementally, and
- A2 = K2 = C2, and D requirements are best handled up front.

The article then proposes some empirical studies that validate the preliminary conclusions of this article.

In the rest of this article, Section 2 describes the categories of D and G requirements in depth. Section 3 defines the completion of a scope as the scope with all its D requirements made explicit. Section 4 gives an iterative procedure for discovering all of a scope's D requirements, thus building the scope's completion. Section 5 makes a few observations about D and G requirements in the wild. Section 6 describes the past empirical work that directly led to the research reported in this article, and Section 7 describes other related past work. Section 8 predicts future work consistent with the long term goals of this research. Section 9 concludes the article.

## 2 G and D Requirements

Suppose we have a simple calculator CBS, $C$, offering only the four operations: addition, subtraction, multiplication, and division. Then the set of requirements,

$R =$ {addition, subtraction, multiplication, division},

is a *scope* of $C$. Then, the requirement,

$r1 =$ exponentiation,

is a G requirement with respect to (w.r.t.) $R$, because exponentiation is not needed for the correct functioning of any of addition, subtraction, multiplication, and division. Adding $r1$ to $R$ makes a different calculator. That is, the addition of $r1$ is *determininG* a new scope. However, the requirement,

$r2 =$checking that the division denominator is not zero,

is a D requirement w.r.t. $R$, because this checking is needed for correct functioning of division. Adding $r2$ to $R$ does *not* make a different calculator; $r2$ is implicitly already in the calculator's scope because its division will break any time the checking fails. That is, the so-called addition of $r2$ is already *determineD* by the current scope.

More formally, suppose that $C$ is a CBS. A *scope of* $C$ is a set $R$ of $C$'s requirements. Each requirement $r$ can be classified into one of two categories w.r.t. $R$.

1. $r$ is a *scope determininG (G) requirement* w.r.t. $R$ if $r$ is not needed for correct functioning in $C$ of any element of $R$
2. $r$ is a *scope determineD (D) requirement* w.r.t. $R$ if $r$ is needed for correct functioning in $C$ of at least one element of $R$ other than $r$.

When the scope $R$ is understood, "w.r.t. $R$" can be elided.

In the rest of this article, (1) "$r$ is needed for correct functioning in $C$ of $q$", (2) "$q$ determines $r$ w.r.t $C$", and (3) "$r$ is determined w.r.t. $C$ by $q$" are synonyms. In these sentences, $r$ is a D requirement w.r.t. $R$.

## 3 Completion of Scope

That adding to a scope one of its D requirements is not considered changing the scope says that there is some notion of *the completion of a scope*, $R$, as $R$ with all its D requirements made explicit.

> The *completion* w.r.t. $C, \mathcal{C}_C(r)$, of a requirement $r$, is the set of all requirements $R$ such that each $r'$ in $R$ is determined w.r.t. C by $r$ or by an element of $R$.
>
> The *completion* w.r.t. $C, \mathcal{C}_C(R)$, of a set of requirements, $R$, is the union of the completions w.r.t. $C$ of all of $R$'s elements.

In principle, the completion of any set of requirements should be the same, no matter the order in which its elements are considered for completion; testing that this is so is part of future work.

RE for a CBS, $C$, typically starts when $C$'s customers supply to requirements analysts (RAs) an initial set of features, $F$. A feature is a requirement, and thus, $F$ is a scope, which is taken, at least initially, as defining $C$.

The distinction between a requirement and a feature is merely a social construct. A feature of a CBS is a requirement of the CBS that the users of the CBS are aware of. Generally, it is the scope of a CBS as a set of features that figures in the decision of customers and users to buy or use the CBS, i.e., what is in the scope and what is not. It is the scope of a CBS as a set of features to be implemented that the customers describe to the RAs to begin the development of the CBS.

The RAs flesh $F$ into its completion, generally requirement by requirement. Because completion adds to a scope only requirements determined by the scope, this fleshing out is not seen as changing the scope of $C$. Therefore, $F$, $F$'s completion, and every scope generated during the fleshing out are considered as describing *the scope of* $C$, $\mathcal{S}(C) = S$. The goal of this fleshing out is to make $S$ explicit, that is to actually contain specifications of all the elements of the completion of $F$, and to serve as a written specification of $C$.

There will be an iterative procedure for completion:

Initially $S = F$. Each iteration considers a candidate new requirement, $r$ to add to $S$, $r$ being identified by any of a variety of elicitation means.

— If $r$ is D w.r.t. $S$, then $S \cup \{r\}$ becomes $S$ for the next iteration.

— If $r$ is G w.r.t. $S$, then, *unless it is explicitly decided to expand the scope with* $r$, $S$ is unchanged for the next iteration, and $r$ is added to the backlog list.

The iteration is complete when $S = \mathcal{C}_C(F)$.

If in any iteration, it is decided to expand the scope of $C$ with the new $r$, then the iteration starts over with $S \cup \{r\}$ as the initial scope.

To allow the iterative procedure to be used not only for upfront RE but also for each sprint of an agile method, the procedure is allowed to start with *any* scope, *any set of requirements*, not just $F$, which is intended to be for the whole of $C$.

## 4 Iterative Procedure for Finding All D Requirements of a Scope

RE for a scope, $R$, of a CBS is done when all of the D requirements of $R$ have been found and included in $R$'s RS, which specifies $R$'s completion [6]. Thus, a procedure[1] for finding all D requirements for a scope is a procedure for carrying out upfront RE for the scope.

1. Pick a scope for the CBS consisting of some initial set $R$ of requirements for the CBS.
2. The analysts must ask, using any of a variety of techniques, such as fault-tree analysis (FTA), failure model and effect criticality analysis (FMECA), hazard analysis (HAZOP), exceptions analysis, etc. [1, 11, 32, 33, 35, 37, 39], what additional requirements are implied by each requirement in $R$, and by any combination of them.
3. The result is a set of D requirements, $R'$.
4. If $R'$ is not empty, repeat Steps 2 through 4 for the new scope, $R \cup R'$.
5. Done, with $R$ being the completion of the original scope.

That is, RE for any scope continues until an iteration, despite all efforts, yields no new requirements, and thus until all requirements determined by any requirement in the scope are found [6]. Any truly *new* requirement, independent of the scope, discovered along the way is put into the backlog list for consideration in a new scope to be developed in the future. Thus, RE for one particular *scope* of a CBS does *not* go on forever.

The reason that the procedure is iterative is that sometimes an exception has its own exceptions, and Step 2 has to be done to a requirement generated in a previous instance of Step 2.

[1] This is a *procedure* and not an *algorithm*. It is *not* guaranteed to find all D requirements of a scope. Thus, elicitation skill and luck are still needed. The procedure improves requirements elicitation by focusing on finding requirements essential to the scope.

## 5 Observations and Implications

The ability to categorize a requirement as either D or G allows focusing the precious RE effort for any version of a CBS on finding for its scope *all* and *only* those requirements, the scope's D requirements, that are necessary to have a complete RS for the version before its implementation begins. The procedure is to chose a scope, i.e., a set of G requirements, for your CBS. Focus all RE effort on finding all D requirements implied by the requirements in the chosen scope, while ignoring all other G requirements, i.e., those that are orthogonal to the requirements in the chosen scope. While this procedure sounds like the upfront RE in a waterfall method, it can be the initial steps in an agile method sprint for the chosen scope. The test cases that serve as the means to verify the correctness of the code for the sprint can be generated from the requirements that emerge from the procedure, if it is not desired to produce an actual RS.

Once the distinction between D and G requirements is understood, it becomes clear that the addition of a D requirement to the scope currently being implemented is *not* scope creep, because the D requirement was already in the scope even if it were not written in the scope's RS. Only the addition of a G requirement is true scope creep.

Another way to understand a missing D requirement is that it is a case of requirements and requirements documentation debt [4] that may not even reflect a conscious decision to incur the debt.

Still another view arises from use-case-based methods, which distinguish two kinds of use cases, (1) *main use cases* or *basic use cases* and (2) *variation* and *exception use cases*. In retrospect, these kinds of use cases are nothing more than (1) G use cases and (2) D use cases, respectively. When use cases are classified in this way, it becomes clear that all of the D use cases of a G use case have to be considered together and be implemented together with the G use case.

## 6 Antecedent Work

Some papers that author Berry coauthored in the past show data that are consistent with and even actively support the claims made in this article. Each paper was written before the ideas reported in this article crystalized; it is actually another piece of slowly accumulated evidence leading to these ideas. Nevertheless, since the data were gathered with no notion of D and G requirements, there is no chance that researcher bias towards supporting this article's claims influenced the data gathering or the original conclusions drawn from the data. In these studies, in each challenged or failed project, a large number, if not a majority, of its defects were from missing D requirements. In the one highly successful project, its RE focused on finding *all* D requirements of its scope.

### 6.1 Lihua Ou's Master's Thesis

Ou's Master's thesis, under Berry's supervision, is a case study of using a user's manual (UM) for a CBS as its RS with upfront RE leading to what was to be a complete RS [5, 27]. That is, Ou was not to begin even design of the CBS until she had finished writing the UM to her customer's satisfaction. Her customer, Berry, who was also her supervisor, had experience with prototypes of the CBS she was to develop and had very clear ideas about what he wanted. So, he forced her to revise the UM, yet again, whenever there was something in the UM that he did not like or could not understand fully. Ou ended up producing eleven versions of the UM before beginning to design the implementation. There were three more revisions necessitated by discoveries of new or changed requirements during implementation. All the major revisions of the manual that affected the CBS's architecture were among the eleven pre-implementation versions, and each of the three revisions that occurred during implementation were relatively minor, focused on specific exceptions to one use case, and had no effect on the CBS's architecture.

Ou had built other CBSs in industrial jobs, mainly in commerce. In these jobs, she had followed the traditional waterfall model, with its traditional heavy-weight SRS. Based on this industrial experience and her study of previous prototypes, Ou planned a 10-month project schedule:

| **Duration in months** | **Step** |
|---|---|
| 1 | Preparation |
| 2 | Requirements specification |
| 4 | Implementation |
| 2 | Testing |
| 1 | Buffer (probably more implementation and testing) |
| 10 | Total planned |

In this schedule, 1 month was allocated to studying the problem, 2 months were allocated to the RE to produce the UM/RS, and 7 months were allocated to the design, implementation, testing, and debugging.

In the actual project, Ou spent a lot more time than planned in RE:

| **Duration in months** | **Step** |
|---|---|
| 1.0 | Preparation |
| 4.9 | Writing of user's manual ≡ requirements specification, 11 versions |
| 0.7 | Design including planning implementation strategy for maximum reuse of pic code and JAVA library |
| 1.7 | Implementation including module testing and 3 manual revisions |
| 1.7 | Integration testing including 1 manual revision and implementation changes |
| 10.0 | Total actual |

The 11 iterations of the UM/RS required nearly 5 months, a nearly 3 month slippage, and Ou thought she was hopelessly behind and would be at least 3 months late. However, the implementation went so smoothly, with almost none of the usual surprises, that Ou ended up finishing by the end of 10th month, on time, with not only the implementation on a Solaris machine that her supervisor required, but also a copy for her own use on a Windows machine. The UM/RS answered nearly every question that Ou, the implementer had about the CBS. There were *no* major new requirements discovered. The 3 UM/RS revisions that occurred during implementation dealt with poor responses to some input errors for a few use cases. These revisions necessitated redesigning only the poor responses, very local updates to the UM/RS, and very local updates to the code being written. So, they slowed the implementation down for only a few hours.

Particularly helpful was that Ou and Berry had worked out all the exceptional and variant cases of every use case that Berry required and had described them in the UM. Thus, Ou had to flesh out only very few exceptional and variant cases and did not have to do any subconscious RE during implementation.

As Ou explained [27],

> We didn't save time during the requirement phase by writing the user's manual instead of a requirement specification. In fact, I would say that we lost time.
>
> Everything got paid back in the later design, implementation, and testing phases. . . . Implementation went much faster than expected. . . .
>
> Compared to projects that I did before, the requirement phase in this case study was no easier than that in a normal life cycle. I expected that writing a user's manual would have been easier than writing a formal requirement specification. However, design, implementation, and testing went much better than expected. The project finished on time and with the customer's satisfaction. While in my past experience, of about 5 years, usually the early requirements and design phases went much more smoothly than in this project. However, in the past projects, always requirements and design problems were discovered during implementation and testing. In this project, there were much fewer problems discovered during implementation and testing, allowing them to go very quickly.

Using the vocabulary of this article, the first version of the UM/RS identified all the G requirements. The UM/RS was complete, because the customer Berry had learned from previous prototypes, i.e., class projects and masters' theses, what he really wanted. In retrospect, each revision thereafter focused on exceptions and variations of an existing requirement, i.e., D requirements. RE continued until it had squeezed out every last drop of D requirements. So, there were almost no requirements, of any kind, to discover during implementation. Only a few obscure D requirements with very local effect were discovered during coding. Thus, coding went *very* fast, and the implemented CBS worked right — in the double senses of "building the right system" and "building the system right" — almost the first time.

### 6.2 Consulting at Company X

At the invitation of a company X, Berry *et al* conducted an interview-and-focus-group study of X's RE process [6]. The main finding was that in an attempt to follow each project's schedule exactly, the RE phase of the project was being stopped on the scheduled date and whatever about the requirements for the project was understood and specified by that date became the RS that the developers implemented the project's CBS. Not surprisingly, the RS was incomplete, leaving out many details that developers needed to know before they could write the code that they were to write. Typically, a developer faced with an incomplete RS simply invented the needed requirement, based on his or her understanding of what is needed and often influenced by what made his or her job easier. Each developer making such decisions independently made for chaos and incorrect software.

Each developer-initiated change to the RS was perceived negatively as requirements creep by all other developers. Adding to the chaos was that, to avoid the animosity that contributing to requirements creep caused, developers stopped reporting changes they made to the requirements in their code [7]. As a result, the RS grew more and more incorrect, i.e., not matching the actual code, and assumed interfaces between modules could not be relied on.

Berry *et al* determined that almost all cases of so-called requirements creep were requirements that were there all along in some of the project members' minds, but were just not expressed in the RS because RE was terminated on the due date before it had finished to produce a coherent complete RS. They called this kind of creep "avoidable creep" and the true creep "unavoidable creep". In retrospect, the avoidable creep requirements were D requirements and the unavoidable creep requirements were G requirements.

As they said in 2010 [6],

> One reason cited for not being willing to spend more time on RE is that there is no apparent end in sight for continuous RE, especially once the need for an iterative approach is identified to allow X's software to keep up with the ever-changing market.
>
> On the other hand, it is recognized that a lot more can be done than is currently done, especially to discover those missing requirement and requirements defects that are eventually found during coding of the RS and that existed at the time RE was terminated. These late-discovered requirements are thought to be true creep, but are really avoidable creep.
>
> The idea is to recognize that there are two kinds of analyses going on during RE:
>
> 1. one to determine the scope, i.e., feature set, of the system to be built, and
> 2. one to determine the details of requirements within any given scope.
>
> There is no end possible for the first kind of RE. One can always add more features to any scope and one can always find variations of any scope that achieve the same functionality. However, deciding whether any scope is right requires building that scope and letting users have a go at it. So, it is necessary to choose a scope based on whatever information is currently available and to resist temptations to modify it or add to it.
>
> However, there is no excuse to proceeding to implementation until all of the requirement defects of the chosen scope have been discovered by thorough RE and until the RS is such that a programmer can code the software without having to ask questions or to make requirements decisions. Proceeding before these details have been worked out creates a situation in which RD [requirements determination] is done by the wrong people, too many times, redundantly, inconsistently, and taking more time than needed and in which correcting defects is done at a significantly higher cost than needed.

They had observed in 2010 what this article is claiming now.

### 6.3 Daniel Isaacs's Master's Thesis

Isaacs and Berry [16] describe a case study of RE practices conducted by Isaacs at his place of employment for his master's degree at the University of Waterloo under Berry's supervision. The case study was conducted at the Ontario office, O, of a Canadian company, X (different from the company X of Section 6.2).

The typical CBS development project at O followed a so-called agile lifecycle. However, that agility was very lightweight and in name only. O did not carefully follow all steps, e.g., continuous customer presence, that ensure wide distribution of requirements knowledge in the absence of an RS. As in many places, O's "Agile" was a fancy name for doing the old-fashioned seat-of-the-pants lifecycle, with no upfront RE, no documentation at any time, skipping everything that is perceived to waste time, and with a lot of scrambling near and after the deadline, and finally, a lot of extra work to fix the inevitable mistakes.

X acquired another Canadian company, Y, whose main product is PY, primarily to incorporate PY's functionality into X's own products. Shortly after the acquisition, O began a project to build PX, which was to duplicate and extend PY's functionality. The main challenge X faced when it started the started the project was its lack of knowledge of PY's domain. PY's developers and other stakeholders, such as end users, were geographically distant from the PX project team. Also, all PY developers, who had domain knowledge about PY, quit rather than become X employees.

X's senior management communicated to the PX developers in O that their job was to replicate the functionality of PY exactly, i.e., no more and no less functionality than PY had. X could not just use PY's code, because PY's functionality had to be migrated to a different platform, in order to incorporate the functionality into O's suite of software. Consequently, the project manager at O communicated PX's requirements as a one-sentence RS:

> Mimic this Webpage.

while pointing to the Webpage implemented by PY.

PY's functionality was not defined or documented anywhere. The acquisition had failed to obtain sufficient information for a smooth development. As a result, the developers in the project did not fully understand what was required to build PY. The implementation of PX ended up relying heavily on each developer's own interpretation, a serious problem since each developer's interpretation was different from those of the others.

To use this paper's vocabulary, the scope for the development of PY consisted of one G requirement, "Mimic this Webpage". Because the developers had access to the PX Webpage, they felt that they could answer any question as it came up by just seeing what the PX Webpage did in the question's situation. As a result, the developers plunged directly into implementation without fleshing out any other requirement, which in this case would be a D requirement, since it was already embodied in the Webpage, in the current scope, but not made explicit in any RS.

The developers delivered a first version of PX after 24 months, 6 months later than planned. Most of the developers admitted that the product that they delivered was of poor quality. The low quality was confirmed by the Quality Assurance (QA) Team. They compared the behavior of PX, the newly created product, against the behavior of PY, the original product. If the new system was missing some functionality, or a bug was found, the QA team opened a ticket in a bug-tracking application. By the end of the third month after delivery, the QA team had logged 681 tickets, a large number, even for O.

For the purposes of the case study, Isaacs tried to determine the origins of all 681 tickets. After reviewing only the first 100 tickets, he gave up, confident of saturation. According to Isaacs's assessment, 37 of the 100 were from missing requirements, and the remaining 63 were bugs introduced during programming of known requirements. Note that Isaacs was an employee of X and worked on the PX project. Thus, he had as much knowledge of the domain as anyone else on the project team, and his classifications of the nature of a defect can be accepted as as accurate as is possible.

In this paper's vocabulary, it is safe to say that of the first 100 tickets, 37 of the defects were from missing D requirements, because the meaning of "an input causes a defect" is that PX did not behave like PY for the input, and that behavior was already embodied in the Webpage, in the current scope, but not made explicit in the RS. The remaining 63 defects were true implementation defects and were not related to any missing requirements. They were the signs of low quality programming. If they are forced to be considered as arising from either a D or a G requirement, then they would need to be considered as arising from a D requirement; they are mistakes in implementing a requirement that is known to be in the current scope.

### 6.4 Chantelle Gellert's Observation

Chantelle Gellert reports a case study of the RE practices of X (different from the companies X of Sections 6.2 and 6.3), a small Ontario software production company [14]. She reports that when a project manager (PM) assigns the implementation of a scope of requirements to a cooperative student, the student's mentor intercepts the assignment and fleshes out the requirements in the assignment with additional detail. Without this detail, the cooperative student, being a temporary employee for three months at a time, does not have enough sustained experience with X to be able to write the code correctly. When the PM assigns the same to a regular employee, the employee is expected to do the fleshing alone.

In this paper's vocabulary, the mentors are probably fleshing out the G requirements of the PM's scope with D requirements so that the cooperative student receives a complete RS for the scope. They are doing so because they recognize that the cooperative student does not yet have enough domain knowledge to quickly and reliably find all the D requirements arising from the G requirements of the PM's scope

## 7 Related Work

Some of the relevant literature is cited in Sections 1–5.

The NaPiRE effort [25] has developed a survey instrument by which participants can identify what pains them about RE, whether they be artifacts, processes, or whatever. The effort has spawned a number of studies of software development organizations in various different places, including Austria, Brazil, and Germany.

The typical report about a survey [26][2] lists the top 5 or 10 pains. Among the top pains that involve RSs are:

- implicit requirements not made explicit [40]: **D**
- incomplete and/or hidden requirements [40]: **D #1**
- inconsistent requirements [24]: **D**
- missing completeness check [18]: **D**

[2] The data from which the list of pains is obtained are in the papers listed at the cited website [26].

- moving targets (changing goals, business processes and or requirements) [24, 40]: **G**
- underspecified requirements that are too abstract and allow for various interpretations [24, 40]: **D #2**
- volatile customer's business domain [40]: **G**

The two of these that seem to be consistently listed in the top 2 of the pains that involve requirements specifications are marked "**#1**" and "**#2**", respectively. The other top pains involve RE processes and communication among stakeholders.

Some of these RS-centered pains, the ones marked "**D**", appear to involve incomplete, missing, or wrong D requirements, because they appear to be about requirements that are already present in the development at hand. Other RS-centered pains, the ones marked "**G**", appear to involve missing G requirements, because they appear to be about true requirements creep. The pains involving D requirements appear to be more frequent than the pains involving G requirements.

With regard to RE in agile projects, Wagner et al observe [40]:

> Furthermore, also in agile projects, it seems to be problematic to rush too quickly through defining what needs to be done ("Not enough time spent defining to the level of detail required").
>
> ...
>
> In many agile RE approaches, requirements are not meant to be complete but a cause for discussions with the customer. Hence, incomplete requirements are to be expected. When does this become a problem? It is a problem if the effect is "Rework or delivery that does not fully meet the customer's need." or "customer dissatisfaction (delivery that does not meet customer expectations)". It is caused by "Hidden requirements that are obvious to the customer" ... Hence, the role of the on-site customer or product owner is a central one that needs to be filled with a person being able to understand the customers and elicit all important requirements.

Note that the developers have to think of asking the customer what to do about an exception. If the developers do not bother to search for exceptions and other D requirements, then the customer who is present is not different from a customer who is not present.

## 8 Future Work and Long Term Goals

The long-term goal of our future research is to answer the research question (RQ):

**RQ:** What is the effect on

1. the development lifecycle of a CBS and
2. the quality of the developed CBS

of an RE that focuses on identifying and specifying upfront, all and only the D requirements in the CBS's scope?

A possible answer to the RQ is expressed as falsifiable, testable hypotheses that will be the subject of future research.

As typically done, an agile development discovers *all* requirements the same way: each sprint defines a scope that includes some new requirements, deferring others to later sprint. As typically done, a waterfall development tries to discover *all* requirements up front before its implementation starts.

The cost observations lead to the testable hypotheses:

**H1:** Regardless of development model,

1. the quality of a CBS, by any measure, is negatively correlated with and
2. the cost of developing the CBS is positively correlated with

the number of D requirements missing from the CBS's scope.

**H2:** Let $S$ be a scope that is missing some D requirements $D'$. Regardless of development model, a development from $S \cup D'$ produces a CBS

1. with better quality and
2. with lower cost

than does a development from $S$.

Some past empirical studies need to be redone taking into account G and D requirements to see if they produce more conclusive results.

Support for these hypotheses recommends modifying agile methods so that each sprint, with a scope, $S$, begins with upfront RE that continues as long as necessary to identify all D requirements for $S$. This modified agile method is agile globally, but within each sprint, it is a waterfall for the scope of the sprint. This modified agile method should produce better CBSs more quickly and with lower cost than do unmodified agile methods.

## 9 Conclusions

This article has identified a new significant categorization of functional requirements, D and G requirements and has offered past case studies showing that focusing a project's RE on finding all of its scope's D requirements has led to higher than expected project success. If future work shows this observation to be true in general, then each sprint of an agile method should include full upfront RE for its scope.

## Acknowledgements

The authors thank Luiz Márcio Cysneiros, Sarah Gregory, Irit Hadar, Andrea Herrmann, Mike Panis, and Anna Zamansky for their comments on previous drafts or in oral presentations of this work.

## References

1. Abdulkhaleq, A., Wagner, S., Leveson, N.: A comprehensive safety engineering approach for software-intensive systems based on STPA. Procedia Engineering **128**, 2–11 (2015)
2. Agile Alliance: Principles: The Agile Alliance (2001), http://www.agilealliance.org/
3. Balaji, S., Sundararajan Murugaiyan, M.: WATEERFALLVs [sic] V-MODEL Vs AGILE: A COMPARATIVE STUDY ON SDLC. JITBM **2**(1) (2012)

4. Barbosa, L., Freire, S., *et al*: Organizing the TD management landscape for requirements and requirements documentation debt. In: Proc. WER (2022), http://wer.inf.puc-rio.br/WERpapers/artigos/artigos_WER22/WER_2022_Camera_ready_paper_28.pdf
5. Berry, D., Daudjee, K., *et al*: User's manual as a requirements specification: Case studies. REJ **9**(1), 67–82 (2004)
6. Berry, D.M., Czarnecki, K., *et al*: Requirements determination is unstoppable: An experience report. In: Proc. RE. pp. 311–316 (2010)
7. Berry, D.M., Czarnecki, K., *et al*: The problem of the lack of benefit of a document to its producer (PotLoBoaDtiP). In: Proc. SWSTE. pp. 37–42 (2016)
8. Berry, D.M., Damian, D., *et al*: To do or not to do: If the requirements engineering payoff is so good, why aren't more companies doing it? In: Proc. RE. p. 447 (2005)
9. Boehm, B.W.: Software Engineering Economics. Prentice-Hall, Englewood Cliffs, NJ, USA (1981)
10. Boehm, B.W.: A spiral model of software development and enhancement. SIGSOFT Softw. Eng. Notes **11**(4), 14–24 (1986)
11. Dony, C., Knudsen, J.L., *et al*: Advanced Topics in Exception Handling Techniques. Springer, Berlin, DE (2006)
12. Ellis, K., Berry, D.M.: Quantifying the impact of requirements definition and management process maturity on project outcome in business application development. REJ **18**(3), 223–249 (2013)
13. Gaborov, M., Karuović, D., *et al*: Comparative analysis of agile and traditional methodologies in IT project management. jATES **11**(4), 1–24 (2021)
14. Gellert, C.: Requirements Engineering and Management Effects on Downstream Developer Performance in a Small Business Findings from a Case Study in a CMMI/CMM Context. Master's thesis, Univ. Waterloo, Canada (2021), http://hdl.handle.net/10012/17777
15. Greenspan, S.J.: Extreme RE: What if there is no time for requirements engineering? In: Proc. RE. pp. 282–284 (2001)
16. Isaacs, D., Berry, D.M.: Developers want requirements, but their project manager doesn't; and a possibly transcendent hawthorne effect. In: Proc. EmpiRE (2011)
17. Jiang, L., Eberlein, A.: An analysis of the history of classical software development and agile development. In: Proc. IEEE SMC. pp. 3733–3738 (2009)
18. Kalinowski, M., Curty, P., *et al*: Supporting defect causal analysis in practice with cross-company data on causes of requirements engineering problems. In: Proc. ICSE SEIP. pp. 223–232 (2016)
19. Kasauli, R., Knauss, E., *et al*: Requirements engineering challenges and practices in large-scale agile system development. JSS **172**, 110851 (2021)
20. Lehman, M.M.: Programs, life cycles, and laws of software evolution. Proc. of the IEEE **68**(9), 1060–1076 (1980)
21. Lehman, M.M.: Laws of software evolution revisited. In: European Workshop on Software Process Technology. pp. 108–124. Springer-Verlag (1996)
22. Lucia, A., Qusef, A.: Requirements engineering in Agile software development. Journal of Emerging Technologies in Web Intelligence **2**(3), 212–220 (2003)
23. Malm, T.: Requirements Engineering in Agile Projects – Comparing a Sample to Requirements-Engineering Literature. Master's thesis, Faculty of Social Sciences, Business and Economics, Åbo Akademi University, Turku, Finland (2020), https://www.doria.fi/bitstream/handle/10024/177487/malm_tobias.pdf
24. Mendez, D., Tießler, M., *et al*: On evidence-based risk management in requirements engineering. In: SWQD: Methods and Tools for Better Software and Systems. pp. 39–59 (2018)
25. Mendez, D., Wagner, S., *et al*: Naming the pain in requirements engineering: Contemporary problems, causes, and effects in practice. EMSE **22**, 2298–2338 (2017)

26. NaPiRE: Napire data and publications (Viewed 1 November 2022), http://napire.org/#/data
27. Ou, L.: WD-pic, a New Paradigm for Picture Drawing Programs and its Development as a Case Study of the Use of its User's Manual as its Specification. Master's thesis, Univ. Waterloo, Canada (2002), https://cs.uwaterloo.ca/~dberry/FTP_SITE/tech.reports/LihuaOuThesis.pdf
28. Rasheed, A., Zafar, B., *et al*: Requirement engineering challenges in agile software development. Mathematical Problems in Engineering **2021** (2021)
29. Rogers, G.: How Agile can requirements engineers really be? RE Magazine (2014), https://re-magazine.ireb.org/articles/requirements-engineers
30. Royce, W.W.: Managing the development of large software systems: Concepts and techniques. In: WesCon (1970)
31. Schach, S.R.: Classical and Object-Oriented Software Engineering With UML and Java. McGraw-Hill, 4th edn. (1998)
32. Shui, A., Mustafiz, S., *et al*: Exceptional use cases. In: Briand, L., Williams, C. (eds.) Model Driven Engineering Languages and Systems. pp. 568–583. Springer, Berlin, DE (2005)
33. Shui, A., Mustafiz, S., Kienzle, J.: Exception-aware requirements elicitation with use cases. In: Dony, C., *et al* (eds.) Advanced Topics in Exception Handling Techniques. pp. 221–242. Springer, Berlin, DE (2006)
34. So, J., Berry, D.M.: Experiences of requirements engineering for two consecutive versions of a product at VLSC. In: Proc. RE. pp. 216–221 (2006)
35. Society of Automotive Engineers: Fault/failure analysis procedure. Tech. Rep. ARP 926A, Superseding, SAE, Warrendale, PA, USA (1979), https://www.sae.org/standards/content/arp926/
36. Thesing, T., Feldmann, C., Burchardt, M.: Agile versus waterfall project management: Decision model for selecting the appropriate approach to a project. Procedia Computer Science **181**(01), 746–756 (2021)
37. Troyan, J.E., LeVine, L.Y.: Ethylene oxide explosion at Doe Run. Loss Prevention **2**, 125–136 (1968)
38. Van Cauwenberghe, P.: Chapter 18: Refactoring or up-front design? (2002), http://wwww.agilecoach.net/html/refactoring_or_upfront.pdf
39. Vesely, W.E., Goldberg, F.F., *et al*: Fault tree handbook. Tech. Rep. ADA354973, Nuclear Regulatory Commission, Washington, DC, USA (1981), https://apps.dtic.mil/sti/pdfs/ADA354973.pdf
40. Wagner, S., Mendez, D., *et al*: Requirements engineering practice and problems in Agile projects: Results from an international survey. In: Proc. CibSE. pp. 85–98 (2017)