Available online at www.sciencedirect.com

ScienceDirect

International Journal of Project Management 35 (2017) 13–27

International Journal of Project Management

www.elsevier.com/locate/ijproman

Excessive software development: Practices and penalties

Ofira Shmueli [a], Boaz Ronen [b,*]

[a] Ben-Gurion University of the Negev, Industrial Engineering and Management Department, Israel
[b] Tel Aviv University, Faculty of Management, Israel

Received 19 October 2015; received in revised form 25 September 2016; accepted 3 October 2016
Available online 28 October 2016

Abstract

This study focuses on the tendency to develop software excessively, above and beyond need or available development resources. The literature pays little attention to this issue, overlooking its crucial impact and penalties. Terms used in reference to excessive software development practices include over-requirement, over-specification, over-design, gold-plating, bells-and-whistles, feature creep, scope creep, requirements creep, featuritis, scope overload and over-scoping. Some of these terms share the same meaning, some overlap, some refer to the development phase, and some to the final system. Via a systematic literature search, we first demonstrate the poor state of research about excessive software development practices in the information systems and project management areas. Then, we suggest a framework consolidating the problems associated with excessive software development in three 'beyond' categories (*beyond needs*, *beyond resources*, *beyond plans*), describe and analyze their causes, consequences, boundaries and overlapping zones. Finally, we discuss the findings and present directions for future research.

Keywords: Software development; Project management; Over-requirement; Over-specification; Over-design; Gold-plating; Bells-and-whistles; Mission creep; Feature creep; Scope creep; Requirements creep; Featuritis; Scope overload; Over-scoping

1. Introduction

Over four decades ago, Brooks (1975) observed that the most difficult part of developing a software system is deciding precisely what to build. He further noted that, if done incorrectly, no other part of the development work cripples the resulting system as much or is more difficult to undo later. This observation, which has been repeatedly acknowledged over the years by academic research and practical experience, still holds today.

Catering to user, market or organizational needs[1] in software development projects has a major impact on project success. Not meeting just the right needs is one of the reasons for project failure (Charette, 2005; Keil et al., 1998), perhaps even the most crucial one (Kliem, 2000; Longstaff et al., 2000). Software scoping is a critical project process (Zwikael and Smyrk, 2011) and project success is sensitive to the defined scope (Cano and Lidón, 2011). Although wrong scope definition refers to both scoping under and over the actual needs (Bjarnason et al., 2012; Buschmann, 2009; Zwikael and Smyrk, 2011), excessively loading the scope is much more common (Bjarnason et al., 2012; Boehm, 2006; Karlsson et al., 2007), and thus stands at the focus of this work. Exceeding the right scope expands project size, which is a major risk dimension in software development projects (McFarlan, 1981; Zmud, 1980) in the sense that project risk is an increasing function of project size (Barki et al., 1993; Glass, 1998; Houston et al., 2001; Maguire, 2002). In comparison to smaller projects, large-scale projects fail three to five times more often (Charette, 2005), are much more prone to unexpected colossal events including even bringing an organization down (Flyvbjerg and Budzier, 2011), and have a 65% probability of being stopped and abandoned (Jones, 2007).

This study relates mainly to traditional plan-based software development methodologies, such as the waterfall approach. Current agile techniques claim to resolve problems associated

* Corresponding author.
E-mail addresses: ofirash@post.bgu.ac.il (O. Shmueli), boazr@post.tau.ac.il (B. Ronen).

[1] See Appendix A for explanations of the software engineering terms.

http://dx.doi.org/10.1016/j.ijproman.2016.10.002
0263-7863/00/

with plan-based methodologies but the debate on which methodology is more effective (Beck and Boehm, 2003; DeMarco and Boehm, 2002), especially in the requirements engineering (RE) context still exists (Dyba and Dingsyr, 2009; Inayat et al., 2015a). Critics of agile techniques claim that agile requirements engineering concepts lead to neglecting non-functional requirements related to performance, security, and architecture (Cao and Ramesh, 2008; Dyba and Dingsyr, 2008; Maiden and Jones, 2010). Although studies that describe requirements engineering practices in an agile context address some of these problems (Bakalova and Daneva, 2011; Lucia and Qusef, 2010), knowledge about the solutions that agile brings to RE is fragmented and whether while introducing solutions to these problems new challenges are introduced is yet to be examined (Inayat et al., 2015a). Accordingly, recent studies claim that this field is still immature and needs further research on agile RE and its real-world impact and applications (Maiden and Jones, 2010; Inayat et al., 2015a, 2015b). However, the understanding that *no size fits all* (Boehm and Lane, 2010a) and that both approaches have their merits and excel under appropriate conditions, suggests more balanced hybrid approaches that integrate both into the right mix for each specific project (Boehm and Turner, 2003, 2005; Boehm et al., 2010; Dyba and Dingsyr, 2008, 2009).

The risky practice of expanding a software project to include excessive functionality and capabilities[2] is referred to in the literature by a variety of partially overlapping terms, including: over-requirement, over-specification, over-design, gold-plating, bells-and-whistles, mission creep, feature creep, scope creep, requirements creep, featuritis, scope overload and over-scoping.

While all these terms relate overall to excessive software development practices, as elaborated upon in the next sections, there are some differences, depending, for example, on the project development phase in which each practice takes place, whether requirements[3] added under the excessive development practice can be implemented within the project constraints or not and whether the added requirements are essential, just optional or completely unnecessary. However, once extra features are introduced into a project excessively, they are seldom eliminated regardless of necessity or of how and during which project phase they are included within the scope (Dominus, 2006; Wetherbe, 1991).

Excessive software development practices are considered risky practices. They impose a variety of penalties on project outcome, with many negative consequences on project schedule, quality and costs (Bernstein, 2012; Bjarnason et al., 2010; Buschmann, 2009, 2010; Coman and Ronen, 2010; Ronen et al., 2012). While some studies refer to a change in requirements of about 25% on average (Jones, 1994; McConnell, 1996), others present an average total volume growth of 14% to 25% for software projects in various domains, with a monthly rate of change in requirements of 1% to 3.5% (Choi and Bae, 2009; Jones, 1996). Jones (1996), however, emphasizes that these numbers can be misleading since the maximum growth rate observed in many cases exceeded 100%. Coman and Ronen (2009b) ascribe over 30% of the features in financial software applications serving such organizations as banks or insurance companies to excessive software development. They claim that over 25% of the software development efforts in R&D organizations are devoted to issues and activities that do not add value (Coman and Ronen, 2010). Considering the conservative estimate of 25% superfluous scope (Battles et al., 1996; Coman and Ronen, 2010), one must wonder what the costs of excessive software development amount to in terms of budget and schedule overruns as well as damage to system quality and integrity. According to McConnell (1996), due to the multiplicative costs associated with doing work downstream, these costs probably amount to much more than 25%. Using the COCOMO II estimation model (Boehm et al., 2000a, 2000b), which considers the exponential nature of the development effort, the estimated cost increment might indeed be even higher than the scope increment, at least with respect to the development activity. Non-development project activities, such as preparing infrastructures or training users, are affected by size and content and are expanded as well due to excessive software development practices. To show that costs can be reduced by eliminating excess, Battles et al. (1996) provide an example of an electric utility which succeeded in reducing the software development budget by 30% without reducing performance by avoiding unnecessary upgrades and non-critical work. Ronen et al. (2012) refer to a cellular phone service provider that by adopting the 25/25 rule in software development, i.e., terminating 25% of the projects and eliminating 25% of the features[4] in the remaining projects, improved the project completion rate and development pace.

Although it is thus extremely important to explore the risky excessive software development practices, enhance the knowledge and awareness of them, and to recommend remedies for their mitigation, a literature search reveals only thin, spare, fragmented and scattered research on these issues. This work makes three main contributions to this challenge. First, via a systematic literature search, focused on title, abstract and keywords of articles in top-rated journals, it unravels the small amount of current relevant research in these leading journals. Second, it gathers and elaborates upon the different terms associated with excessive software development practices and provides a comprehensive picture regarding their nature, causes and consequences. Third, based on the findings and analysis presented here it proposes a research agenda for future research in several directions.

The rest of this paper is dedicated to reviewing the various excessive software development practices and to exposing the current poor state of relevant research. Section 2, first identifies the various terms that relate to excessive software development practices and then presents the findings of a systematic literature search for relevant research. Section 3 consolidates the various excessive software development practices in three 'beyond' categories, and analyzes their nature, causes, and boundaries. Finally, Section 4 discusses the findings, conclusions and implications and proposes a research agenda. A glossary of basic software engineering terms used here is

[2] See Appendix A for explanations of the software engineering terms.
[3] See Appendix A for explanations of the software engineering terms.
[4] See Appendix A for explanations of the software engineering terms.

provided in Appendix A, and a further description of our literature search is provided in Appendix B.

2. The state of research on excessive software development practices

As mentioned above, the risky practice of excessively expanding the software project is referred to in the literature by a variety of names, some of which share the same meaning or overlap: over-requirement, which refers to specifying a product or a service beyond the actual needs of the customer or the market (Shmueli et al., 2014); over-specification, which also refers to defining product or service specifications beyond the actual needs of the customer or the market (Ronen and Pass, 2008); over-design, which refers to designing and developing products or services beyond what is required by the specifications and/or the requirements of the customer or the market (Ronen and Pass, 2008); gold-plating, which refers to extra software which not only consumes extra effort but also reduces the conceptual integrity of the product (Boehm and Papaccio, 1988); bells-and-whistles, which refers to adding unnecessary features to software (Ropponen and Lyytinen, 2000); feature creep, which refers to changing features while a product is still in development (Elliott, 2007); requirements creep and mission creep, which also refer to changing requirements and mission while a product is still in development (Elliott, 2007); scope creep, which refers to the steady increase of the system's scope (Buschmann, 2009); featuritis, which refers to the tendency to trade functional coverage for quality and the tendency to deliver as many functions and as early as possible (Buschmann, 2010); scope overload, which refers to the excessive inclusion of software features within the scope of a project while consuming more resources than those available (Shmueli et al., 2015c); and over-scoping, which also refers to setting a scope that requires more resources than are available (Bjarnason et al., 2010).

The above excessive software development practices present extreme risks to project success (Bjarnason et al., 2010; Boehm, 1991; Buschmann, 2009, 2010; Coman and Ronen, 2010), leading to delayed project launch and resource overruns, overall excessive complexity, increased probability of defects and reliability problems, software which is larger than needed and even loss of an entire company (Battles et al., 1996; Buschmann, 2009; Buschmann, 2010; Coman and Ronen, 2009b, 2010; Westfall, 2005). Acknowledging their highly negative influence and prevalence (Buschmann, 2009, 2010; Coman and Ronen, 2010; Jones, 1996; Ronen et al., 2012), one might have expected research to explore these excessive software development practices and to reveal their nature and causes. However, all our digging and burrowing for such research indicated no such focus.

The many terms used to describe excessive software development practices and their acknowledged penalties they impose on the software project, on one hand, and the feeling of research shortage on the other hand, have motivated us in performing a systematic literature search to check the state of research on excessive software development practices. Our systematic literature search which focused on 17 leading sources confirms that there is very little research on these negative practices. A description of the literature search is presented next, together with its findings. Further elaboration on the search, the list of the searched terms and their variations is provided in Appendix B.

Since the issue we relate to is mainly relevant to the information systems (IS), software engineering (SE) and software development (SD) areas, we covered in our literature search, as listed in Table 1, eight sources (1–8) included by the Association of Information Systems (AIS) among the Senior Scholars' Basket of Journals (AIS, 2011), two top sources in the project management (PM) area (9–10), namely the International Journal of Project Management (IJPM) and the Project Management Journal (PMJ), and five sources dealing with practical issues in the software development (SD) and software engineering (SE) areas, namely three IEEE journals, Computer, IEEE Transaction on Engineering Management, and Information and Software Technology (11–15). Finally we also covered two reviews that deal with and also set the current agenda (16–17), namely Communications of the ACM and the Harvard Business Review.

In these publications, we considered all papers published up to September 2015. The search targeted the 12 terms related to excessive software development practices in various forms ("scope overload", "scope-overload and "scopeoverload"; "bells and whistles" and "whistles and bells"; etc.), as elaborated in Appendix B. To target papers that focus on an excessive software development practice and to avoid those that mention it in some marginal manner, for example as part of a risk list, we focused our search on the title, abstract and keywords.

This search yielded 20 hits which, after manual relevancy checks, came down to 12 articles, pointing to the scarcity of research on these problems. Out of the overall publications in our literature search, the proportion dealing with excessive software development practices is less than 0.001%.

Table 2 presents the acceptable yield (after manual relevancy checks, as elaborated in Appendix B) ordered by year of publication. It is noteworthy that the majority of the articles (six of the 12, namely 1, 3–7) deal with requirements creep. It is also

Table 1
Sources for literature review.

#	Journal
1	European Journal of Information Systems (EJIS)
2	Information Systems Journal (ISJ)
3	Information Systems Research (ISR)
4	Journal of AIS (JAIS)
5	Journal of Information Technology (JIT)
6	Journal of Management Information Systems (JMIS)
7	Journal of Strategic Information Systems (JSIS)
8	MIS Quarterly (MISQ)
9	International Journal of Project Management (IJPM)
10	Project Management Journal (PMJ)
11	IEEE Transactions on Software Engineering
12	IEEE Software
13	Computer
14	IEEE Transactions on Engineering Management
15	Information and Software Technology (IST)
16	Communications of the ACM
17	Harvard Business Review (HBR)

Table 2
Extracted articles — source type, method of research.

#	Article	Type of source	Term	Method of research
1	Jones, 1996	SD-IEEE	Requirements creep	Speculation/commentary
2	Boehm, 1996	SD-IEEE	Gold-plating	Speculation/commentary field study
3	Hendrix and Schneider, 2002	ACM	Requirements creep	Case study
4	Damian and Chisan, 2006	SE-IEEE	Feature creep Requirements creep	Case study
5	Lee-Kelley and Sankey, 2008	PM	Requirements creep	Case study qualitative research Semi-structured interviews
6	Chen and Yang, 2009	SD-IEEE	Scope creep Requirements creep	Case study
7	Choi and Bae, 2009	SE-IST	Requirements creep	Conceptual model; Case study
8	Buschmann, 2010	SD-IEEE	Featuritis Unnecessary features	Speculation/Commentary
9	Coman and Ronen, 2010	PM	Over-specification Over-design	Speculation/Commentary
10	Bjarnason et al., 2012	SE-IST	Over-scoping	Case study
11	Shmueli et al., 2015a	PM	Over-requirement	Laboratory experiment
12	Shmueli et al., 2015c	IS	Over-requirement Scope overload	Laboratory experiment

important to note that, out of the 12 papers, three (9, 11, 12) are products of the authors of this paper. Regarding the source type, five of the articles (1, 2, 4, 6, 8) were published in SD/SE-IEEE journals, three of the articles (5, 9, 11) were published in PM journals, two of the articles (7, 10) were published in the SE-IST journal, one (3) in the ACM journal, and one (12) in an IS journal. Referring to the research method column in Table 2, based on the taxonomy developed by Palvia et al. (2007), four papers (1, 2, 8, 9) presented opinions in a commentary, providing little or no empirical evidence, six papers (3–7, 10) applied a case-study approach, and two (11, 12) presented empirical results of a laboratory experiment. Examining the content of the 12 articles demonstrates the generality of the excessive development problems in terms of the different domains and software types they are manifested in: from commercial software to military software, from general information systems to enterprise resource planning software and whether the software project is developed in house or by an outsourced supplier. Looking at the level of analysis in these studies implies overall an organization issue, since this is the level of analysis in most studies, with only one study (5) referring to the group (the development team) and three (8, 11, 12) referring to the individual (developer or architect). Reviewing the negative consequences of the various problems shows that in general all excessive development problems expose the software project to similar penalties, negatively affecting schedule, cost and quality. However, reviewing the causes, dynamic and the attitude toward the manifestation of the various problems, points to some differences. Requirements creep, feature creep and scope creep are considered in some studies an inherent consequence of the software project dynamics on which the software team and sometimes also the clients have no control. This is because change, during the course of a software project, is considered almost inevitable, either due to external factors such as changes in regulation or due to business factors such as competitive pressures. The other problems, such as over-requirement and over-scoping however, which are listed sometimes as negative by-products of the creep problems, are treated differently. Most of their presented causes are related to human factors, such as the wish to provide the best possible solution or the desire to include front-end technology even when not needed, and thus theoretically can be avoided. However, only two papers (11, 12) were dedicated to empirical exploration of the behavioral roots of over-requirement via a laboratory experiment and only one (23) also explored a mitigation concept in an experimental environment.

The essence of these 12 excessive development practices together with the above observations have led us to consider the classification and consolidation of these practices into three major categories, relating to the project aspect which they interfere with — project needs, project plans and project resources. Accordingly, based on a wider literature view, not limited to the listed leading journals and not limited to studies that focus specifically on this issue, the next section presents these three categories and proceeds with a deeper analysis of their nature, causes, and boundaries.

3. Review and analysis of excessive software development practices

The dozen terms used to describe excessive software development practices, together with selected references, are listed respectively in the first and second columns of Table 3. Terms 1 to 5 refer to specifying, designing and developing a software system beyond the actual needs of the customer or the market, namely loading the software with unnecessary features and capabilities (Boehm and Papaccio, 1988; Markus and Keil, 1994; Ronen and Pass, 2008; Ropponen and Lyytinen, 2000). Terms 6 to 9 refer to changing and adding features and functionality once the project is under way (Elliott, 2007; Feiler, 2000) while the tenth term, which usually accompanies them, refers to the tendency to load a project with features (Buschmann, 2010). Finally, the terms 11 and 12 refer to setting a scope that includes more functionality than can be implemented within the framework of the project resources, i.e., the time, people, and budget assigned to the project. Relating to the

Table 3
Excessive software development practices.

Term used for practice	Selected references for practice	Category of similar practices and commonalties
1. Over-requirement	Pass and Ronen, 2014; Shmueli et al., 2015a, 2015c	A) *Beyond Needs*: Specifying, designing and developing a software system beyond the actual needs of the customer or the market, loading the software with unnecessary features and capabilities
2. Over-specification	Abrahams, 1988; Coman and Ronen, 2010; Ronen and Pass, 2008	
3. Over-design	Coman and Ronen, 2009a, 2010	
4. Gold-plating	Boehm and Papaccio, 1988; Boehm, 1991; Kaur et al., 2013; Khanfar et al., 2008; Malhotra et al., 2012; NASA, 1992; Wheatcraft, 2011	
5. Bells-and-whistles	Ropponen and Lyytinen, 2000	
6. Mission creep	Elliott, 2007	B) *Beyond Plans*: Continuously changing and adding features and functionality beyond the planned plans, once the software development project is underway, loading the project with extra features
7. Feature creep	Elliott, 2007; McConnell, 1997; Rust et al., 2006	
8. Scope creep	Buschmann, 2009; Feiler, 2000; Lang and Fitzgerald, 2005; Murphy, 2001; Wheatcraft, 2011	
9. Requirements creep	Damian and Chisan, 2006; Elliott, 2007; Jones, 1996; Wheatcraft, 2011	
10. Featuritis	Buschmann, 2010; Elliott, 2007	
11. Scope overload	Shmueli et al., 2015c	C) *Beyond Resources*: Setting the scope of a software system beyond the available resources, including more functionality and capabilities than can be implemented within the framework of the project resources
12. Over-scoping	Bjarnason et al., 2012	

phases of the traditional software project life cycle, while loading the software project with unnecessary features (terms 1 to 5) and determining a project scope that exceeds its available resources (terms 11 and 12) can take place at any phase of the project, the rest take place during the development phase and on. The similar practices are consolidated in Categories A to C in the third column of Table 3 and Sub-sections 3.1 to 3.3 detail their commonalities, causes and penalties. Sub-section 3.4 then compares and contrasts these twelve practices, drawing the boundaries and the overlapping zones.

3.1. Beyond needs (Category A)

The tendency to specify, design and develop a software system beyond the actual needs of the customer or the market, namely loading the software with extra, completely unnecessary or nice-to-have features and capabilities (Ronen and Pass, 2008; Shmueli et al., 2015a), is common to the over-requirement (Pass and Ronen, 2014; Shmueli et al., 2015a), over-specification, over-design, gold-plating, and bells-and-whistles practices of Category A. Shmueli et al. (2015a) define over-requirement as a synonym to over-specification and gold-plating and as specifying a product or a service beyond the actual needs of the customer or the market. Ronen and Pass (2008) define over-design as designing a product or a service beyond its specification.[5] Abrahams (1988) associates the over-specification practice with the unlikely-ever-to-be-needed capabilities. Boehm and Papaccio (1988) refer to gold-plating as extra software which not only consumes extra effort, but also reduces the conceptual integrity of the product while others define gold-plating as the addition of unnecessary whistles and bells (Markus and Keil, 1994; Ropponen and Lyytinen, 2000). While the term gold-plating is often used in the academic literature (Boehm and Papaccio, 1988; Kaur et al., 2013; Schmidt et al., 2001), over-specification is the term usually used in industry (Coman and Ronen, 2010). Worth mentioning is that the term over-specification is also used to describe another phenomenon, whereby specifications of a product or a feature are too elaborate and constrain the later life cycle phases (Boehm and Hansen, 2001; Boehm and Turner, 2003; Palshikar, 2001).

Already in the early 1990s, NASA (1992) included gold-plating among the eight "don't do" warnings in the software-development context and Boehm (1991) listed it among the top ten risks in software development projects. Others repeatedly mentioned, as a major risk and a major concern in software-development projects, not only gold-plating (Bernstein, 2012; Kaur et al., 2013; Khanfar et al., 2008; Malhotra et al., 2012; Schmidt et al., 2001; Wheatcraft, 2011) but also over-specification, over-design (Belvedere et al., 2013; Pass and Ronen, 2014), and over-requirement (Pass and Ronen, 2014; Shmueli et al., 2014; Shmueli et al., 2015a). Nevertheless, the five excessive software development practices in Category A, manifested as excessive functional requirements, excessive performance requirements or excessive scalability requirements, are still prevalent (Boehm et al., 2000a, 2000b; Boehm and Hansen, 2001; Boehm and Lane, 2010b; Buschmann, 2009).

Potential damages associated with these beyond-needs practices include delayed launch, high complexity, cutting off core features due to project time constraints, devoting human and machine resources to developing functionality that is of no real value, project overruns, and even demise of an entire company (Buschmann, 2009; Coman and Ronen, 2009b, 2010; Ronen et al., 2012; Westfall, 2005). These practices may also lead to highly complex software, with increased risk of defects and reliability problems (Coman and Ronen, 2010; Westfall, 2005) and to larger systems that are difficult to manage and costly to maintain (Battles et al., 1996; Buschmann, 2010). The supplier is at risk as well since the customer will prefer another supplier in the future (Kautz, 2009).

[5] See Appendix A for explanations of the software engineering terms.

As elaborated next and presented in Table 4, the list of causes related to beyond-needs excessive practices is quite long and diverse. Professional interest or pride of developers and demands of users are considered to be the main reasons for beyond-needs practices (Ropponen and Lyytinen, 2000). Developers often ignore business requirements for the sake of introducing advanced technology (Buschmann, 2009; Schmidt et al., 2001), or pursue unauthorized features to satisfy their own interest (Coman and Ronen, 2010; McConnell, 1997). In many cases they wish to achieve the best possible solution (Rust et al., 2006; Westfall, 2005) or aim to fulfill all future needs and add just-in-case functionality (Buschmann, 2010; Coman and Ronen, 2010), especially when they do not know which features will be more important (Anton and Potts, 2003; Boehm, 1996). Users as well often opt for advanced technology, ignoring business requirements, or exhibit an all-or-nothing attitude (Cule et al., 2000), adding costly unneeded features to system requirements (Markus and Keil, 1994). Sometimes they even try to coax individual developers into incorporating their favorite features (McConnell, 1997). Lack of a time constraint or a budget constraint may also cause such practices, as work tends to fill the time available (Kemerer, 1987) and money allocated becomes money spent (Miranda and Abran, 2008). Other causes are misconceptions during the specification phase that the development effort is free (Koopman, 2010, 2011), that each added feature has the same marginal effort (Coman and Ronen, 2010) or low cost (Boehm and Papaccio, 1988). Behavioral biases such as the I-designed-it-myself effect and the planning fallacy have been demonstrated to impact over-requirement, since software professionals may perceive features as more valuable and beneficial for the software system than they really are (Shmueli et al., 2015a, 2015b, 2015c). In an outsourced software-development project, a contract type of time and material might be behind practices in Category A (Gary, 2009; Kautz, 2009). DeMarco and Lister (2003) describe a case where politics may cause some of the stakeholders who are adversaries to overload the project with excessive functionality. For off-the-shelf software or a large system developed for a variety of users, the desire to build one system that fits all (Coman and Ronen, 2010; Rust et al., 2006) or to release improved versions on a continuous basis (Coman and Ronen, 2010) can lead to Category A practices.

Table 4 summarizes the above, lists the causes and indicates the specific stakeholder, of the three main stakeholders involved in software project development (developer, user/customer, manager), reported to be affected by each cause in the inclusion of beyond-needs content.

It can be seen from the foregoing, that of the three main stakeholders involved in software projects, the developer is most likely and most motivated to include beyond-needs content, whether for his/her own professional interest or due to his/her good intentions to achieve the best solution. However, the other two stakeholders, the user, who wants as much as possible, and the management, who have to be aware of these tendencies and avoid being drawn into going along with them, are responsible as well. Reviewing the list of causes, it seems that at least six causes (1–4, 6–7, 10) are overall derived from good intentions, without counting the negative effects of developing beyond needs software. Only two causes (8, 9) can be associated with the deliberate inclusion of beyond needs content. The rest are such that stem from knowledge shortage (5, 11) or human biases (12, 13). Considering these causes, without the two of deliberate intention, strongly implies a lack of awareness regarding the penalties of beyond needs development. The main recommendation, thus, is to enhance awareness of the beyond-needs excessive development phenomenon and the penalties it imposes on project outcome. Another recommendation is, especially for the manager, to be aware of the behavioral biases that interfere with and affect decision making and objectivity of the subordinates and as well as of the manager.

Future research on beyond-needs excessive practices should address the following issues, whether as independent directions, or as sequential stages. First, the extent and volume of this phenomenon needs empirical exploration and measurement, since the current numbers reported in the literature are based on evaluations and speculations. This research direction should consider plan-based and agile development methodologies. Second, in order to set a solid base for deciding on which causes to focus on, it is important to explore the prevalence of the various causes presented here. Again, agile practices should be included. Third, and most important, remedies should be developed and empirically evaluated. Apart from one study (Shmueli et al., 2015c) that empirically demonstrates

Table 4
Causes and stakeholders of beyond needs category.

		Developer	User/customer	Manager	None
1	Professional interest	×			
2	Wishing for best possible solution	×	×		
3	Aim to fulfill all future needs	×			
4	Just-in-case functionality	×			
5	Lack of knowledge (not knowing which features will be more important)	×			
6	All-or-nothing attitude		×		
7	Lack of time/budget constraint			×	
8	Time and material contract type			×	
9	Politics — adversaries overload the project			×	
10	One system that fits all			×	
11	The misconception that development effort is free	×	×	×	
12	The I-designed-it-myself effect	×			
13	The planning fallacy	×		×	

how remedies of the planning fallacy can mitigate over-requirement and over-scoping, no other empiric research was found.

3.2. Beyond plans (Category B)

The tendency to continuously change and add features and functionality, once the software development project is under way, loading the project with extra features (Elliott, 2007; Wheatcraft, 2011), is common to the mission creep, feature creep, scope creep, requirements creep and featuritis practices of Category B. As in Category A, the boundaries between these five terms are vague in the literature, with practically no clear distinctions among them. Elliott (2007) mentions mission creep, scope creep, requirements creep and featuritis as synonyms to feature creep, defined as changes in features while a product is still in development. Similarly, according to Feiler (2000), scope creep occurs "when additional activities are added once the project is under way" (p.49), while Buschmann (2009) complains that due to "a steady increase of the system's scope" the software system "becomes a jack of all trades and master of none" (p.68). Buschmann (2010) defines featuritis as the tendency to trade functional coverage for quality, denoting it as the disease that afflicts development when software professionals seek to deliver as many functions as soon as possible while quality issues such as reliability, performance and maintainability are postponed to "when functionality is stabilized" (p.10).

Excessive software development practices in Category B are considered a software development risk (Bartlett, 2008; Buschmann, 2010; McConnell, 1997; Murphy, 2001; Naz and Khokhar, 2009). McConnell (1996, 1997) counts feature creep as one of the most common sources of cost and schedule overruns, a major factor in project cancelations and a source of destabilization. Changes in requirements were identified by managers in a large software development company focused on security applications as the most prevalent risk associated with its projects (Shalev et al., 2014). Frequent requests for requirements changes was identified as a major cause for failing to accurately predict development time and budget (Lederer and Prasad, 1992). Loading a project with features and capabilities eventually lowers value from a user perspective, besides increasing the risks associated with project size (Rust et al., 2006).

Software type, whether enterprise system, web application or data warehouse, seems to make no difference in terms of the negative impact of excessive beyond-plans practices (Chen and Yang, 2009; Connor, 2003; Lang and Fitzgerald, 2005; Momoh et al., 2010). Scope creep has been identified as one of the reasons for the high failure rate of enterprise projects (Chen and Yang, 2009; Momoh et al., 2010). Internet developers consider scope creep and feature creep as most troubling (Lang and Fitzgerald, 2005), and data warehousing projects have been reported to miss deadlines due to scope creep (Connor, 2003). Software systems suffering from scope creep tend to be overly generic or offer a list of functions that is too long (Buschmann, 2009).

Wishing to continuously improve the developed software, a tendency for perfection, and changes in actual business conditions seem to be major causes of excessive software development practices in Category B (Elliott, 2007; Jones, 1996). Accepting the first specification change request opens the door for additional ones to creep into scope, whether at the request of users or the initiative of developers (Murphy, 2001). External factors over which one has no control, such as changes in tax laws or in commercial software due to competitive pressures, have been reported as well as causes of beyond-plans software development practices (Jones, 1996). Outsourcing is also mentioned as a cause of scope creep, since vendors usually expect to gain extra profit from incremental scope changes (Davison, 2003; Zhao and Watanabe, 2010). Human behavior is noted as causing featuritis when using flexibility to cover for uncertainty, adding extra functionality just in case, or adding functionality just because it is so simple to do (Buschmann, 2010). Table 5 summarizes the above, lists the causes and indicates the specific stakeholder, of the three main stakeholders involved in software project development, reported to be driven by each cause in introducing beyond-plans practices.

Hence, apart from external factors which can be unpredictable and represent a force majeure, beyond-plans practices are primarily manifested due to the seemingly positive desire for continuous improvement and an outsourced supplier who gains profit from such a dynamic. Thus, among the main three stakeholders involved (developer, user/customer, manager) such practices should be controlled and governed by the project management. Considering these causes, without the two external causes, the implication is low awareness regarding the penalties of beyond-plans development and/or lack of management control.

Future research on beyond-plans excessive practices should address the following issues, whether as independent directions, or as sequential stages. First, here as well, the extent of this phenomenon needs empirical exploration and measurement, since the current numbers reported in the literature are based on evaluations. Second, it is important to examine which causes are most influential, especially with regard to the external ones. Finally, remedies should be developed and empirically evaluated. An interesting exploration would be to examine how the agile philosophy of embracing the change cope with or avoid the negative consequences.

3.3. Beyond resources (Category C)

The tendency to set the scope of a software system to include more functionality than can be implemented within the limits of project resources (Bjarnason et al., 2010; Shmueli et al., 2015c) is common to the scope-overload and over-scoping practices of Category C. Developing a project with a scope overload, or over-scoping a software project, means exceeding the available resources (Shmueli et al., 2015c), and "biting off more than one can chew" (Bjarnason et al., 2010, 2012). These practices are considered a top software development risk (Buschmann, 2010; Coman and Ronen, 2009b; Elliott, 2007; Flyvbjerg and Budzier, 2011) since the more the project is over-scoped, the higher the probability of failure. Due to the need to reduce the scope when it becomes clear in later development phases that project resources cannot permit the extra scope, system requirements must be

Table 5
Causes and stakeholders of beyond plans category.

		Developer	User/customer	Manager	None (external)
1	Wishing for best possible solution	×	×		
2	Just-in-case functionality	×			
3	Wish to continuously improve the developed software			×	
4	Accepting the first specification change			×	
5	Outsourcing (provider interest)			×	
6	Changes in laws / regulation				×
7	Competitive pressures				×

changed and all work already invested in developing the extra requirements goes to waste (Bjarnason et al., 2012). Category C practices may burden the developers with too much work which, in turn, may hurt overall software quality and customer satisfaction (Bjarnason et al., 2012). These excessive, beyond-resources, software development practices involve inclusion of both required and over-required software features, whether introduced during the initiation phase of the project lifecycle or that crept in during the construction phase (Zwikael and Smyrk, 2011). Applying agile methods for requirements engineering does not prevent beyond-resources practices, though the load becomes more manageable and is perceived to result in less wasted effort due to the principle of continuous scope prioritization (Bjarnason et al., 2011, 2012).

The root causes of scoping beyond resources relate mainly to poor management of continuous requirements inflow, compounded by unclear overall project goals (Bjarnason et al., 2010, 2012). In addition, the cognitive bias of the planning fallacy impacts this tendency to load the project with extra functionality exceeding available resources (Shmueli et al., 2015b, 2015c). Due to the planning fallacy, which leads to underestimating the time needed for development and, as a result, overestimating how much can be accomplished within a given period of time, the software professional tends to load the scope with more features than can be completed on time. Table 6 summarizes the above, lists the causes and indicates the specific stakeholder, of the three main stakeholders involved in software project development, reported to be driven by each cause in including beyond-resources content.

So, beyond-resources practices mainly stem from management mistakes, whether in failing to set clear project goals and keeping to them or in failing to obtain correct estimates of time or other required resources. Aside from the obvious need for clear project goals and management control, the main recommendation is, especially for the manager, to be aware of the planning fallacy, a behavioral bias that leads to under estimation of time needed to develop software and over estimation of what can be accomplished by specific resources.

Future research on beyond-resources excessive practices should measure the extent of this phenomenon, considering plan-based and agile development methodologies. It is also important to recognize the extent of over-scoping beyond the scope set at the beginning of the project and added to the project along the way (due to one of the beyond-plans practices).

3.4. Excessive software development categories compared and contrasted

The various excessive software development practices overlap and are often confused. Buschmann (2009) describes scope creep in terms of over-requirement, where the developed system is overly generic and offer functions that do not contribute to the system's purpose. Markus and Keil (1994) associate scope creep with adding more and more costly bells-and-whistles to system requirements. Keil et al. (1998) claim that scope creep can be prevented by educating the customer on its negative impact and recommend that managers draw a line between desirable and absolutely necessary functionality, hence relating actually to over-requirement. Zwikael and Smyrk (2011) associate over-scoping with the inclusion of over-required, redundant or superfluous features within scope (Zwikael and Smyrk, 2011). This mix up is understandable since the boundaries between the various excessive development practices are indeed vague and blurred. In addition, one excessive development practice might be a consequence of another. Feature creep, for example, which involves expanding the project beyond its initial plans, may include addition of over-required features and scope-overload, meaning features beyond needs or beyond available resources. Scope overload may emerge from including in the project scope, beyond-needs, bells-and-whistles functionality and vice versa, from setting a scope beyond available resources; some extra beyond-needs functionality may be included as well.

Figs. 1 and 2 depict (in the inner circle) the actual essential needs necessary for a software project to provide using the assigned project resources (marked by a dashed line). Fig. 1

Table 6
Causes and stakeholders of beyond resources category.

		Developer	User/customer	Manager	None (external)
1	The planning fallacy	×		×	
2	Poor management of continuous requirements inflow			×	
3	Unclear overall project goals			×	

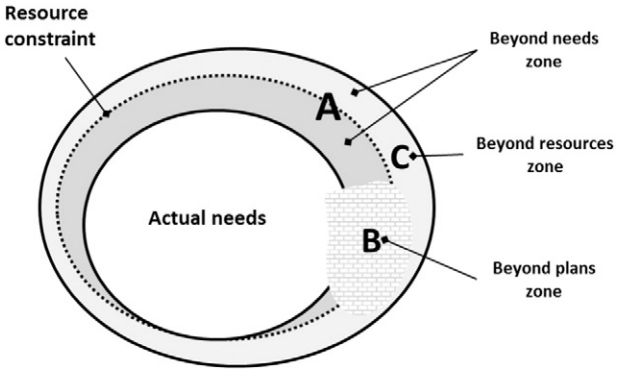

Fig. 1. The project scope and the various excessive development practices zones (actual needs are feasible within the limits of available resources).

refers to the situation in which the actual essential needs are feasible within the limits of available resources while Fig. 2 refers to the less prevalent situation where essential needs exceed available resources, dooming the whole project to failure. Beyond the actual needs is the Category A zone (the gray areas), that is, the over-requirement practice by definition of being beyond the actual needs of the customer or the market. As can be seen in both Figs. 1 and 2 this gray zone lies beyond as well as within the assigned resources, as features introduced under these practices usually exceed project resources, but can sometimes be developed within their limits.

The zone of Category C (in the lighter gray), representing requirements beyond project resources, partially covers in Fig. 1 the Category A zone, since by depicting all essential needs within the available resources means that requirements which are beyond resources are necessarily beyond needs.

In Fig. 2 as well, the zone of Category C (the lighter gray), partially covers the Category A zone. However, since in Fig. 2 not all essential needs lie within the available resources, Category C also partially covers some of the actual needs. This demonstrates an anomaly whereby the project scope includes unnecessary requirements while lacking some essential ones. This paradoxical situation may have several causes. For example, it might be a result of the project dynamic, where during the development phase, upon realizing that the project is about to exceed its planed schedule, due to the limited scoping freedom at that stage, core features are discarded while less essential or nice-to-have ones are kept in scope (Coman and Ronen, 2009a, 2009b, 2010). It might also be a result of the specific mix of available resources, versus the resources needed for the development of the different requirements. The underlying argument here is that although these two figures present a two-dimensional schema, in reality it is a multi-dimensional situation, since in referring to time, human and budget constraints under project resources, three dimensions are already at play. So, there might be a situation in which an essential feature exceeds budget while a nice-to-have one does not. Finally, Figs. 1 and 2 present practices in Category B (the brick texture), that is, feature creep, requirements creep, scope creep and mission creep, where extra features are added after the project has already begun, and may include required as well as beyond-needs requirements, whether within the limits of the assigned resources or beyond, hence overlapping with all other zones.

Thus, as can be graphically seen in Figs. 1 and 2, the three categories of excessive software development practices overlap. Requirements which are beyond needs, introduced by Category A practices, may lie within the limits of the available project resources but also beyond them, overlapping with Category C, and they can be introduced into the project at the initial phase or creep in later on during the development, overlapping the excessive development practices in Category B. Similarly, requirements added during development under beyond-plans practices of Category B might be unnecessary, thus overlapping Category A, or they may lie beyond the limits of the assigned resources, thus overlapping Category A. Finally, requirements introduced into the project scope by Category C practices, which are beyond available resources, may include requirements that are beyond needs, thus overlapping Category A, or requirements included after the initial phase, thus overlapping Category B. So, each of the three categories actually overlaps the other two.

Relating to the project phase or phases in which each excessive development practice introduces its burden, Fig. 3 schedules graphically all practices in a typical traditional software development life cycle schema, while Table 7 presents this information in rows and columns.

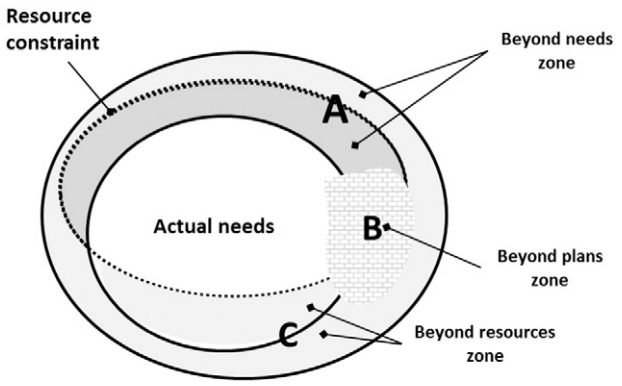

Fig. 2. The project scope and the various excessive development practices zones (actual needs exceeding available resources).

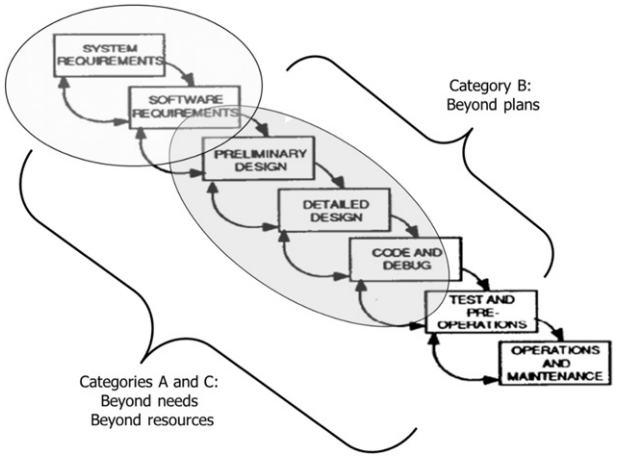

Fig. 3. Excessive software development practices along project development phases.

Table 7
Excessive software development practices along project development phases.

	System requirements	Software requirements	Preliminary and detailed design	Code and debug	Test and pre-operations	Operations and maintenance
Category A: Beyond needs	√	√	√	√	√	√
Category B: Beyond plans			√	√	√	√
Category C: Beyond resources	√	√	√	√	√	√

The excessive beyond-plans practices of changing and adding features and functionality once the project is under way occur, by definition, mainly during the development phase (specification, design and coding). Similarly, by definition, over-design practice takes place mainly during the design phases (preliminary or detailed) but also later on. However, the other beyond-needs and beyond-resources software development practices can be introduced into the project at any stage, though of course the early project phases are more prone to such faults.

4. Discussion, conclusion, limitation and future research

We focus on three excessive software development practices that are addressed in the literature by a variety of terms, which sometimes overlap and are sometimes used in a confused manner. The study addresses these issues, describes and categorizes the various terms and points out causes and penalties of excessive practices, as well as their scope and boundaries. The following sub-sections discuss some of the data and findings of this study from other aspects, such as the role and relevance of various stakeholders, and the role and relevance of different software development methodologies, in triggering or mitigating these excessive development practices. Overall, the findings and discussion of this study pave the way for the future research directions suggested next.

4.1. Literature search findings

The literature search presented in this study, demonstrates the overwhelming scarcity of relevant research in IS, PM and SD, top rated journals, identifying only 12 relevant publications. Focusing the search on topic, abstract and keywords of publications, aimed at finding those studies in which the excessive development practice is the main issue and not just mentioned marginally. This finding is surprising since the penalties and negative impact of these excessive practices have been acknowledged for decades, and, at least regarding beyond-needs practices, have been recognized as top software development risks since the late 1980s (Boehm and Papaccio, 1988; NASA, 1992). Moreover, the literature search reveals the lack of research on beyond-needs practices, with only four papers on this problematic issue, three of which were written by us (Coman and Ronen, 2010; Shmueli et al., 2015a, 2015c). As we see it, these Category A excessive development practices, where project resources are devoted to the development of software with no added value, is the most wasteful one. The excessive development practices that receive the highest attention (seven out of the 12 papers) are the beyond-plans Category B practices (Buschmann, 2010; Elliott, 2007; Feiler, 2000). However, looking at the year of publication of the studies identified by our literature search, the trend seems to be positive over time, with two publications till 1999, five in the decade 2000–2009 and also five just over the last six years.

4.2. Causes and stakeholders

The causes of excessive software development practices are diverse in nature, specifically in terms of which stakeholder is at fault, in which project phase it takes place and whether it can be avoided, just mitigated or is a force majeure. Table 8 presents a consolidated list of the various causes presented in the specific 'beyond' category sub-section, and relates to the three main stakeholders involved in software project development. The first three columns indicate the 'beyond' category that stems from each cause, marked by "A", "B" and "C" accordingly. The next three columns refer to the three main stakeholders, where "×" indicates a specifically documented relevancy, similarly to Tables 4, 5 and 6. Besides documented relevancy, Table 8 also includes, an assumed relevancy as we evaluate it, indicated by the "." sign added here. The last column ('none') is for causes driven by no direct stakeholder and representing an indication of external causes.

As can be easily seen most causes are internal ones, explicitly related to one of the stakeholders. The two external causes however, are related to the beyond-plans excessive practice category. It is also clear that some causes are reported to cause more than one type of 'beyond' category. Thus, for example 'wishing for best possible solution' is reported to cause the beyond-needs and beyond-plans categories, and 'the planning fallacy' is reported to be a cause of the beyond-needs and beyond-resources categories. Since the three beyond-needs, beyond-plans and beyond-resources excessive practices overlap and one triggers another, it is reasonable to assume a larger group of mutual causes.

Of the three main stakeholders, the user is the one with the least influence, so, the developer and, of course, the management are responsible for most causes. Comparing the last two, regarding the causes that were specifically documented as relevant ("×") or regarding all causes that were marked as relevant here ("×" and "."), it reasonably turns out that management is the stakeholder mainly responsible for excessive software development practices. Moreover, regarding causes triggered by one of the other stakeholders, such as "professional interest" of developers or the "I-designed-it-myself" manifestation, it is the

Table 8
Causes of excessive software development practices and stakeholders.

		Beyond category			Stakeholder			None (ext.)
		A	B	C				
		Needs	Plans	Resources	Developer	User/customer	Manager	
1	Professional interest	A			×			
2	Wishing for best possible solution	A	B		×	×		
3	Aim to fulfill all future needs	A			×			
4	Just-in-case functionality	A	B		×			
5	Lack of knowledge (not knowing which features will be more important)	A			×			
6	All-or-nothing attitude	A				×	.	
7	Lack of time/budget constraint	A			.		×	
8	Time and material contract type	A			.		×	
9	Politics – adversaries overload the project	A					×	
10	One system that fits all	A				.	×	
11	The misconception that development effort is free	A			×	×	×	
12	The I-designed-it-myself effect	A			×	.		
13	The planning fallacy	A		C	×		×	
14	Wish to continuously improve the developed software		B			.	×	
15	Accepting the first specification change		B				×	
16	Outsourcing (provider interest)		B		.		×	
17	Changes in laws / regulation		B					×
18	Poor management of continuous requirements inflow			C			×	
19	Unclear overall project goals			C			×	
20	Competitive pressures		B					×

role of management to govern and control the scope and project dynamics. Nevertheless, it is important for each stakeholder to acknowledge, first, the negative consequences of excessive development practices and, second, to consider how he/she can avoid them in his/her own area of influence. Worth mentioning is that some causes are a result of a human bias, which if one is aware of it may not eliminate the bias but can mitigate its impact (Stacy and MacMillian, 1995; Shmueli et al., 2015a). However, management should keep the whole picture in focus.

4.3. Software development methodologies

This study focuses primarily on traditional plan-based software development methods, such as the waterfall approach. Recent agile software development methodologies have aimed at resolving traditional waterfall pitfalls. However, they are considered weak in the area of requirement engineering, which is the focus of this article, and research regarding their evaluation is still sparse (Cao and Ramesh, 2008; Dyba and Dingsyr, 2008, 2009; Maiden and Jones, 2010). While some principles of the agile manifesto (www.agilemanifesto.org) and techniques, like the continuous scope prioritization and the guidance for simplicity ("the art of maximizing the amount of work not done"), are sensible for reducing excessive software development, others, like welcoming changes in requirements even late in development, can definitely triggers excessive software development.

Moreover, considering the generality of some of the causes described here, especially those derived from human aspirations or cognitive biases, such as the professional interests of developers or biased stakeholder evaluation, excessive development practices may be manifested in agile development projects as well. It is reasonable that due to the I-designed-it-myself effect, under agile development too, attachment to a specified feature will influence one's judgment, leading to the tendency to perceive the feature as more important than it really is and affecting its prioritization. It is also reasonable that due to the planning fallacy, while assessing features planned for a current agile iteration, time underestimation and benefit overestimation might influence the iteration planning, in the direction of including more scope than can be included with iteration resources. Indeed, regarding over-scoping for example, Bjarnason et al. (2012) observed that applying agile methods does not necessarily prevent over-scoping, although due to the continuous scope prioritization principle, the load is more manageable and perceived to result in less wasted effort.

4.4. Conclusions and limitations

Excessive software development practices that over-load software projects with extra, excessive functionality and capabilities have long been considered major software development risks (Boehm, 1991; Brooks, 1975; Jones, 1994). Despite the various overlapping and sometimes inconsistent terminology used in the literature to describe these problematic practices, their negative consequences and prevalence are well documented and acknowledged. However, as evident from a literature search of top-rated IS, PM and SD publications, research of these excessive development practices, their causes and mitigation is spare, and the studies which are completely devoted to these problems are limited in number. Indeed, we can say that the main finding of this literature search is the enormous scarcity of research on excessive software development practices. In particular, there is lack of research on beyond-needs development practices. This guides us to the conclusion that excessive software development in general and the specific problematic practice of specifying, designing and

developing a software system beyond the actual needs are neglected phenomena that are yet to be explored.

The list of documented causes of excessive software development practices reveals their heterogeneous nature. It also clarifies that no stakeholder is blameless, even though the issue is always the responsibility of managers. Most causes are rooted in human nature and behavior and it thus seems essential that the negative consequences of excessive development practices should be constantly kept in mind by all stakeholders, throughout the different project phases. Managers obviously should ensure the implementation of good old requirements engineering practices, such as having and utilizing a change control board, assessing the impact of each requested change on cost and schedule, and stabilizing requirements and specifications as early as possible (Khanfar et al., 2008). However, the prevalence of excessive software development practices despite the acknowledgment of these good managerial tools implies that other directions and remedies besides technical ones should be considered. Managers should particularly recognize the existence of the human biases that affect them and their employees, and should seek ways to mitigate them. For example, hiring an uninvolved consultant to get an outsider resource evaluation has the potential of mitigating the impact of the planning fallacy (Shmueli et al., 2015c).

The move away from plan-based development methodologies toward agile ones aims, inter alia, to improve requirement definition. Yet, our literature review did not reveal any evidence of this and recent studies indeed identify a research gap here that needs to be bridged (Maiden and Jones, 2010; Inayat et al., 2015a, 2015b). Moreover, it seems that agile project development does not eliminate the persistent excessive development practices, as specifically noted in one of the papers dealing with scope overload (Bjarnason et al., 2012). So, the influence of agile development on the extent and mitigation of excessive software development practices is yet to be explored.

This paper touches on the mechanisms that drive problematic excessive development. Yet further research is needed, especially on beyond-needs development. Acquiring knowledge of their scale, causes and roots will enhance the ability to develop effective remedies for the mitigation of excessive software development and improvement of the software development process.

The main limitation of this study is in referring to software development as a uniform practice while actually there are different software development domains with different characteristics and dynamics. Although excessive development practices are shown to be relevant within various software development contexts, the scope and consequences might differ for the different software domains. Being an initial research, this study looks at software development as a whole and leaves the differentiation to future research.

4.5. Contribution, implications and proposed research agenda

This study, via a systematic focused literature search of specific top-rated journals, confirms our notion of a problematic phenomenon that is yet to be explored, and then provides a broad view of the various excessive software development practices, and organizes and categorizes the various relevant terms. Recognizing the poor state of research on these issues, and the analysis of the knowledge regarding each practice, we propose here a research agenda for future exploration of excessive software development practices. This agenda refers to several research avenues that can be explored independently but together compose a broad picture of excessive software development practices, their extent, their manifestation, their root causes, and remedies. First, the current prevalence and extent of the various practices should be addressed in a systematic manner, measuring their scale and impact. While current numbers of around 30% beyond needs and beyond plans software are based on conservative assessments, with a suspicion of much higher numbers, a systematic measure will clarify the picture and may confirm speculations. Addressing different domains will also provide a differentiation regarding the state of excessive software development practices in each domain. In addition, considering both plan-based and agile development environments will provide a comparison between the two methods and will examine the promises of agile development for overcoming these excessive practices. Second, the causes of excessive software development practices should be explored for their prevalence, dynamics and the stakeholder that most influences them. This will clarify the picture and identify the root causes to focus on. Earlier in this section we presented a long list of 20 different causes together with the excessive-development practice and the specific stakeholder that are documented to be relevant to each cause. However, we added above some assumptions for a richer table, where the same cause affects more stakeholders than documented. Also, keeping in mind the relations and the overlapping zones between the various excessive development practices, it is reasonable to assume that the same cause may lead to several practices, or at least to additional practices besides the documented one. This should be examined to complete the table of causes with relevant practices and stakeholders. Again, agile practices should be included, perhaps adding a third dimension of development methodology, to the same table. Third, and most important, remedies should be developed or found, relating to the complete table of causes and their classifications. Perhaps other research contexts such as behavioral economics should be considered, borrowing remedies proven to be effective in mitigating behavioral effects for empirical evaluation of their influence on causes with a behavioral component. Such a direction seems promising since it reflects a new research direction which previous research, until recently (Shmueli et al., 2015a, 2015c), did not explore regarding excessive development practices. Previous research focused mainly on techniques and methodologies but it seems to have failed to improve the situation. Moreover, the behavioral characteristics of many of the causes suggest a path to explore.

Notwithstanding its limitation, this study contributes to both research and practice, providing immediate implications to both. First, based on its findings and analysis it proposes a research agenda to follow. It delineates several research directions which can be addressed systematically or sporadically; either way will enhance the little knowledge on this extremely important issue. Its contribution and implications for practice are in focusing on these neglected negative practices, pointing to the extent to which

they have been evaluated and the penalties each of them impose on the software project. Providing a consolidated picture of the causes and various stakeholders that play a role in introducing excessive-development practices highlights their broad nature and scope. This should alert management to the need to increase awareness of the issue throughout the development group hierarchy and the need for a tight management control on project scope along all stages of the software project lifecycle.

Conflict of interest

There is no conflict of interest.

Appendix A. Software engineering terms

The following explanations taken from the fields of requirement engineering and information systems refer to the specific terms used in this paper. For each of the terms, a simple clarification (a) is presented, followed, for some terms, by a formal definition (b).

1. Needs
   a. The benefits to the user if implemented within the software system.
   b. Definition: "As the solution space is explored and solution classes are characterized, stakeholder needs are developed and transformed into stakeholder requirements (from a user perspective)" (SEBoK — System Engineering Body of Knowledge).
2. Requirement
   a. The translation of a user need (or some other needs) into an unambiguous, clear, unique, consistent statement. An unambiguous, clear, unique, consistent statement of **what** the user needs (directly or indirectly, like infrastructure and performance aspects).
   b. Definition: "A statement that identifies a system, product or process characteristic or constraint, which is unambiguous, clear, unique, consistent, stand-alone (not grouped), and verifiable, and is deemed necessary for stakeholder acceptability" (INCOSE, Systems Engineering Handbook: A Guide for System Life Cycle Processes and Activities. Version 3.2.1, 2010).
3. Specification
   a. The translation and elaboration of the requirements into unambiguous, clear, unique, consistent guidance, algorithms and instructions describing **how** the target software will fulfill the user needs.
4. Features
   a. An intentional distinguishing characteristic of a piece of software, especially in terms of functionality, but also in terms of performance, or redundancy.
5. Capabilities
   a. The set of outcomes that the software system provides or enables the user to gain through its use.
   b. Definition: "An outcome or effect which can be achieved through use of features of a system of interest and which contributes to a desired benefit or goal" (SEBoK — System Engineering Body of Knowledge).

Appendix B. Description of the literature search

The search targeted the 12 terms relating to excessive software development practices, in various forms: Over requirement/Over-requirement; Overspecification/Over specification/Over-specification; Overdesign/Over-design/Over design; Gold plating/Gold-plating; Bells and whistles/Whistles and bells; Mission creep/Mission-creep; Feature creep/Feature-creep; Scope creep/Scope-creep; Requirements creep/Requirements-creep; Featuritis; Scope overload/Scope-overload/Scopeoverload; and Over scoping/Over-scoping/Overscoping. To make the search more comprehensive, we added to the list of terms, other phrases and expressions which imply and may also be associated with excessive software development practices: Over doing/Over-doing/Over do/Over-do; Over killing/Over-killing/Over kill/Over-kill; Unnecessary features; Unnecessary requirements; and nice-to-have features. All terms and phrases, except for bells and whistles and nice-to-have features, were searched for enclosed by quotation marks for their exact appearance in the text. Bells and whistles and nice to have were openly searched to capture any relevant appearance of the phrases (e.g. "the bells and the whistles" (Meyer, 1999)).

The search, which yielded 20 hits, came down to 12 articles, indicating the scarcity of research on these problems. This downsizing occurred after manually scanning the content of each hit for relevancy and excluding articles that did not relate in their abstract to software development or mentioned the searched term under a different meaning. To judge the relevance of these 12 papers beyond the abstract alone (Brereton et al., 2007), we read each article and confirmed its acceptability. The total number of searched articles was over 50,000. Thus, the proportion of publications dealing with the excessive software development practices (n = 12) out of the total in our literature search is less than 0.001%.

Table 2 in this paper presents the acceptable yield and various aspects regarding their distribution are discussed. The research method column in Table 2 is based on the taxonomy developed by Palvia et al. (2007) and also used by several other researchers (Avison et al., 2008; Fleischmann et al., 2014): 1) Speculation/commentary; 2) Frameworks and conceptual model; 3) Library research; 4) Literature analysis; 5) Case study; 6) Survey; 7) Field study; 8) Field experiment; 9) Laboratory experiment; 10) Mathematical model; 11) Qualitative research; 12) Interview; 13) Secondary data; and 14) Content analysis.

References

Abrahams, P., 1988. President's letter. Commun. ACM 31, 480–481.
AIS, 2011. Senior Scholars' basket of journals. available at http://start.aisnet.org/?SeniorScholarBasket (accessed September 9 2015).
Anton, A.I., Potts, C., 2003. Functional paleontology: the evolution of user-visible system services. IEEE Trans. Softw. Eng. 29 (2), 151–166.
Avison, D.E., Dwivedi, Y.K., Fitzgerald, G., Powell, P., 2008. The beginnings of a new era: time to reflect on 17 years of the ISJ. Inf. Syst. J. 18 (1), 5.

Bakalova, Z., Daneva, M., 2011. Agile requirements prioritization: what happens in practice and what is described in literature. Lecture Notes in Computer Science Ed. Springer-Verlag, Berlin Heidelberg, pp. 181–195.
Barki, H., Rivard, S., Talbot, J., 1993. Toward an assessment of software development risk. J. Manag. Inf. Syst. 10 (2), 203–225.
Bartlett, R., 2008. 17 Project Pitfalls and how to Avoid them. Iseries News, pp. 38–44.
Battles, B.E., Mark, D., Ryan, C., 1996. An open letter to CEOs: how otherwise good managers spend too much on information technology. McKinsey Q. 1996 (3), 116–127.
Beck, K., Boehm, B., 2003. Agility through discipline: a debate. Computer 36 (6), 44–46.
Belvedere, V., Grando, A., Ronen, B., 2013. Cognitive biases and overdesign: an investigation on the unconscious mistakes of industrial designers and on their effects on product offering. In: Giannoccaro, I. (Ed.), Behavioral Issues in Operations Management. Springer-Verlag, London, pp. 125–140.
Bernstein, L., 2012. Things I learned from taming software development. ACM Sigsoft Softw. Eng. Notes 37 (6), 5–6.
Bjarnason, E., Wnuk, K., Regnell, B., 2010. Overscoping: reasons and consequences — a case study on decision making in software product management. Proceedings of the 4th International Workshop on Software Product Management (IWSPM), September 27, Sydney, NSW, Australia.
Bjarnason, E., Wnuk, K., Regnell, B., 2011. A case study on benefits and side-effects of agile practices in large-scale requirements engineering. Proceedings of the 1st Agile Requirements Engineering Workshop (AREW), July 26, Lancaster, United Kingdom.
Bjarnason, E., Wnuk, K., Regnell, B., 2012. Are you biting off more than you can chew? A case study on causes and effects of overscoping in large-scale software engineering. Inf. Softw. Technol. 54 (10), 1107–1124.
Boehm, B., 1991. Software risk management: principles and practices. IEEE Softw. 8 (1), 32–41.
Boehm, B., 1996. Anchoring the software process. IEEE Softw. 13 (4), 73–82.
Boehm, B., 2006. A view of 20th and 21st century software engineering. Proceedings of the 28th International Conference on Software Engineering, Shanghai, China.
Boehm, B., Hansen, W., 2001. The spiral model as a tool for evolutionary acquisition. J. Def. Softw. Eng. 14 (5), 4–11.
Boehm, B., Lane, J.A., 2010a. New processes for new horizons: the incremental commitment model. 32nd ACM/IEEE International Conference on Software Engineering (ICSE), Cape Town, South Africa, May 1–8.
Boehm, B., Lane, J.A., 2010b. Evidence-based software processes. Proceedings of the International Conference on Software Process (ICSP), July 8–9, Paderborn, Germany.
Boehm, B., Papaccio, P., 1988. Understanding and controlling software costs. IEEE Trans. Softw. Eng. 14 (10), 1462–1477.
Boehm, B., Turner, R., 2003. Using risk to balance agile and plan-driven methods. IEEE Comput. 36 (6), 57–66.
Boehm, B., Turner, R., 2005. Management challenges to implementing agile processes in traditional development organizations. IEEE Softw. 22 (5), 30–39.
Boehm, B., Abts, C., Brown, A., Chulani, S., Clark, B., Horowitz, E., Madachy, R., Reifer, D., Steece, B., 2000a. Software Cost Estimation with Cocomo II. Prentice-Hall, Upper Saddle River, New Jersey.
Boehm, B., Port, D., Al-Said, M., 2000b. Avoiding the software model-clash spiderweb. Computer 33 (11), 120–122.
Boehm, B., Lane, J., Koolmanojwong, S., Turner, R., 2010. Architected agile solutions for software-reliant systems. Agile Software Development. Springer, Berlin-Heidelberg, pp. 165–184.
Brereton, P., Kitchenham, B.A., Budgen, D., Turner, M., Khalil, M., 2007. Lessons from applying the systematic literature review process within the software engineering domain. J. Syst. Softw. 80 (4), 571–583.
Brooks, F.P., 1975. The Mythical Man-Month. Adison-Wesley Publishing Company, Reading, Massachusetts.
Buschmann, F., 2009. Learning from failure, part 1: scoping and requirements woes. IEEE Softw. 26 (6), 68–69.
Buschmann, F., 2010. Learning from failure, part 2: featuritis, performitis, and other diseases. IEEE Softw. 27 (1), 10–11.
Cano, J.L., Lidón, I., 2011. Guided reflection on project definition. Int. J. Proj. Manag. 29 (5), 525–536.
Cao, L., Ramesh, I., 2008. Agile requirements engineering practices: an empirical study. IEEE Softw. 25 (1), 60–67.
Charette, R.N., 2005. Why software fails. IEEE Spectr. 42 (9), 42–49.
Chen, L., Yang, S., 2009. Managing ERP implementation aailure: a project management perspective. IEEE Trans. Eng. Manag. 56 (1), 157–170.
Choi, K., Bae, D., 2009. Dynamic project performance estimation by combining static estimation models with system dynamics. Inf. Softw. Technol. 51 (1), 162–172.
Coman, A., Ronen, B., 2009a. Focused SWOT: diagnosing critical strengths and weaknesses. Int. J. Prod. Res. 47 (20), 5677–5689.
Coman, A., Ronen, B., 2009b. Overdosed management: how excess of excellence begets failure. Hum. Syst. Manag. 28 (3), 93–99.
Coman, A., Ronen, B., 2010. Icarus' predicament: managing the pathologies of overspecification and overdesign. Int. J. Proj. Manag. 28 (3), 237–244.
Connor, 2003. Report: data warehouse failures commonplace. Netw. World 20 (3), 24.
Cule, P., Schmidt, R., Lyytinen, K., Keil, M., 2000. Strategies for heading off IS project failure. Inf. Syst. Manag. 17 (2), 65–73.
Damian, D., Chisan, J., 2006. An empirical study of the complex relationships between requirements engineering processes and other processes that lead to payoffs in productivity, quality, and risk management. IEEE Trans. Softw. Eng. 32 (7), 433–453.
Davison, D., 2003. Top 10 risks of offshore outsourcing. META Group Report.
DeMarco, T., Boehm, B., 2002. The agile methods fray. IEEE Comput. 35 (6), 90–92.
DeMarco, T., Lister, T., 2003. Risk management during requirements. IEEE Softw. 20 (5), 99–101.
Dominus, M., 2006. Creeping featurism and the ratchet effect. available at http://blog.plover.com/prog/featurism.html (accessed September 9 2015).
Dyba, T., Dingsyr, T., 2008. Empirical studies of agile software development: a systematic review. Inf. Softw. Technol. 50 (2008), 833–859.
Dyba, T., Dingsyr, T., 2009. What do we know about agile software development? IEEE Softw. 26 (5), 6–9.
Elliott, B., 2007. Anything is possible: managing feature creep in an innovation rich environment. Proceedings of the IEEE International Engineering Management Conference, July 29–Aug 1, Piscataway, New Jersey.
Feiler, E., 2000. Evaluating accounting software consultants. CPA J. 70 (6), 46–51.
Fleischmann, M., Amirpur, M., Benlian, A., Hess, T., 2014. Cognitive biases in information systems research: a scientometric analysis. Proceedings of the 22nd European Conference on Information Systems (ECIS), June 9–11, Tel Aviv, Israel.
Flyvbjerg, B., Budzier, A., 2011. Why your IT project may be riskier than you think. Harv. Bus. Rev. 89 (9), 23–25.
Gary, M., 2009. Poor industry consultancy threatens data center constructions. available at http://www.businesscomputingworld.co.uk/poor-industry-consultancy-threatens-data-centre-constructions/ (accessed September 9 2015).
Glass, R., 1998. Editor's corner: software runaways – some surprising findings. J. Syst. Softw. 41 (2), 75–77.
Hendrix, T.D., Schneider, M.P., 2002. NASA's TReK project: a case study in using the spiral model of software development. Commun. ACM 45 (4), 152–159.
Houston, D.X., Mackulak, G.T., Collofello, J.S., 2001. Stochastic simulation of risk factor potential effects for software development risk management. J. Syst. Softw. 59 (3), 247–257.
Inayat, I., Moraes, L., Daneva, M., Salim, S.S., 2015a. A reflection on agile requirements engineering: Solutions brought and challenges posed. Proceedings of the 16th International Conference on Agile Software Development, May 25–29, Helsinki, Finland.
Inayat, I., Salima, S.S., Marczakb, S., Danevac, M., Shamshirbandd, S., 2015b. A systematic literature review on agile requirements engineering practices and challenges. Comput. Hum. Behav. 51 (Part B), 915–929.
Jones, C., 1994. Assessment and Control of Software Risks. Yourdon Press, Upper Saddle River, New Jersey.
Jones, C., 1996. Strategies for managing requirements creep. Computer 29 (6), 92–94.
Jones, C., 2007. Estimating Software Costs: Bringing Realism to Estimating. McGraw-Hill, New York.

Karlsson, L., Dahlstedt, A.G., Regnell, B., Dag, J.N., Persson, A., 2007. Requirements engineering challenges in market-driven software development — an interview study with practitioners. Inf. Softw. Technol. 49 (6), 588–604.
Kaur, K., Jyoti, B., Rani, R., 2013. Analysis of gold plating: a software development risk. Int. J. Comput. Sci. Commun. Eng. 2 (1), 51–54.
Kautz, K., 2009. The impact of pricing and opportunistic behavior on information systems development. J. Inf. Technol. Theory Appl. 10 (3), 24–41.
Keil, M., Cule, P.E., Lyytinen, K., Schmidt, R.C., 1998. A framework for identifying software project risks. Commun. ACM 41 (11), 76–83.
Kemerer, C.F., 1987. An empirical validation of software cost estimation models. Commun. ACM 30, 416–429.
Khanfar, K., Elzamly, A., Al-Ahmad, W., El-Qawasmeh, E., Alsamara, K., Abuleil, S., 2008. Managing software project risks with the chi-square technique. Int. Manag. Rev. 4 (2), 18–29.
Kliem, R.L., 2000. Risk management for business process reengineering projects. Inf. Syst. Manag. 17 (4), 71–73.
Koopman, P., 2010. Risk areas in embedded software industry projects. Proceedings of the Workshop on Embedded Systems Education (WESE), October 24, Scottsdale, Arizona.
Koopman, P., 2011. Avoiding the top 43 embedded software risks. Proceedings of the Embeded Systems Conference, May 2, Carnegie Mellon University.
Lang, M., Fitzgerald, B., 2005. Hypermedia systems development practices: a survey. IEEE Softw. 22 (2), 68.
Lederer, A.L., Prasad, J., 1992. Nine management guidelines for better cost estimating. Commun. ACM 35 (2), 50–59.
Lee-Kelley, L., Sankey, T., 2008. Global virtual teams for value creation and project success: a case study. Int. J. Proj. Manag. 26 (1), 51–62.
Longstaff, T.A., Chittister, C., Pethia, R., Haimes, Y.Y., 2000. Are we forgetting the risks of information technology? Computer 33 (12), 43–51.
Lucia, A., Qusef, A., 2010. Requirements engineering in agile software development. J. Emerg. Technol. Web Intell. 2 (3), 308–313.
Maguire, S., 2002. Identifying risks during information system development: managing the process. Inf. Manag. Comput. Secur. 10 (2/3), 126–134.
Maiden, N., Jones, S., 2010. Agile requirements — can we have our cake and eat it too? IEEE Softw. 27 (3), 87–88.
Malhotra, N., Bhardwaj, M., Kaur, R., 2012. Estimating the effects of gold plating using fuzzy cognitive maps. Int. J. Comput. Sci. Inf. Technol. 3 (4), 4806–4808.
Markus, M.L., Keil, M., 1994. If we build it, they will come: designing information systems that people want to use. Sloan Manage. Rev. 35 (4), 11–25.
McConnell, S., 1996. Avoiding classic mistakes. IEEE Softw. 13 (5), 111–112.
McConnell, S., 1997. Achieving leaner software. IEEE Softw. 14 (6), 127–128.
McFarlan, F.W., 1981. Portfolio approach to information systems. Harv. Bus. Rev. 59 (5), 142–150.
Meyer, B., 1999. Every little bit counts: toward more reliable software. IEEE Comput. 32 (11), 131–133.
Miranda, E., Abran, A., 2008. Protecting software development projects against underestimation. Proj. Manag. J. 39 (3), 75–85.
Momoh, A., Roy, R., Shehab, E., 2010. Challenges in enterprise resource planning implementation: state-of-the-art. Bus. Process. Manag. J. 16 (4), 537.
Murphy, L., 2001. Using software project "should-cost" models. Trans. AACE Int. 4, 1–4.3.
NASA, 1992. Recommended Approach to Software Development. Goddard Space Flight Center, Greenbelt, Maryland.
Naz, H., Khokhar, M.N., 2009. Critical requirements engineering issues and their solution. Proceedings of the International Conference on Computer Modeling and Simulation (ICCMS), February 20–22, Macau.
Palshikar, G.K., 2001. Applying formal specifications to real-world software development. IEEE Softw. 18 (6), 89–97.
Palvia, P., Pinjani, P., Sibley, E.H., 2007. A profile of information systems research published in information and management. Inf. Manag. 44 (1), 1–11.
Pass, S., Ronen, B., 2014. Reducing the software value gap. Commun. ACM 57 (5), 80–87.
Ronen, B., Pass, S., 2008. Focused Operations Management: Achieving More with Existing Resources. John Wiley & Sons, Hoboken, New Jersey.
Ronen, B., Lechler, T.G., Stohr, E.A., 2012. The 25/25 rule: achieving more by doing less. Int. J. Prod. Res. 50 (24), 7126–7133.
Ropponen, J., Lyytinen, K., 2000. Components of software development risk: how to address them? A project manager survey. IEEE Trans. Softw. Eng. 26 (2), 98–112.
Rust, R.T., Thompson, D.V., Hamilton, R.W., 2006. Defeating feature fatigue. Harv. Bus. Rev. 84 (2), 98–107.
Schmidt, R., Lyytinen, K., Keil, M., Cule, P., 2001. Identifying software project risks: an international Delphi study. J. Manag. Inf. Syst. 17 (4), 5–36.
Shalev, E., Keil, M., Lee, J.S., Ganzach, Y., 2014. Optimism bias in managing IT project risks: a construal level theory perspective. Proceedings of the 22nd European Conference on Information Systems (ECIS), June 9–11, Tel Aviv, Israel.
Shmueli, O., Pliskin, N., Fink, L., 2014. Behavioural effects in software development: an experimental investigation. Proceedings of the 22nd European Conference on Information Systems (ECIS), June 9–11, Tel Aviv, Israel.
Shmueli, O., Pliskin, N., Fink, L., 2015a. Explaining over-requirement in software development projects: an experimental investigation of behavioral effects. Int. J. Proj. Manag. 33 (2), 380–394.
Shmueli, O., Pliskin, N., Fink, L., 2015b. A position paper proposing behavioral solutions to challenges in software development projects. Proceedings of the Advanced Information Systems Engineering (CAiSE) Workshops, June 8–9, Stockholm, Sweden.
Shmueli, O., Pliskin, N., Fink, L., 2015c. Can the outside-view approach improve planning decisions in software development projects? Inf. Syst. J. (Available online at: http://onlinelibrary.wiley.com/doi/10.1111/isj.12091/full).
Stacy, W., Macmillian, J., 1995. Cognitive bias in software engineering. Commun. ACM 38 (6), 57–63.
Westfall, L., 2005. The what, why, who, when and how of software requirements. Proceedings of the ASQ World Conference on Quality and Improvement, May 16–18, Seattle, Washington.
Wetherbe, J.C., 1991. Executive information requirements: getting it right. MIS Q. 15 (1), 51–65.
Wheatcraft, L.S., 2011. Triple your chances of project success risk and requirements. INCOSE International Symposium, Denver, CO, June 23.
Zhao, W., Watanabe, C., 2010. Risk management in software outsourcing — a portfolio analysis of India's case based on software export market constitution. J. Serv. Res. 10 (1), 143.
Zmud, R., 1980. Management of large software efforts. MIS Q. 4 (2), 45–55.
Zwikael, O., Smyrk, J., 2011. An engineering approach for project scoping. Proceedings of the IEEE 18th International Conference on Industrial Engineering and Engineering Management, September 3–5, Changchun, China.