Specify First or Build First? Empirical Studies of Requirements Engineering Activities: A Survey *

Irwin Kwan and Daniel M. Berry

School of Computer Science
University of Waterloo,
Waterloo, Ontario N2T 3G1, Canada
{ihkwan,dberry}@uwaterloo.ca

Abstract. The importance of the requirements phase is recognized by academics and by industry. Many methods for eliciting and specifying requirements have been proposed, such as up-front specification, formal specification, and prototyping, but there are few empirical studies that study the effects of these activities on the later phases of the software life cycle. This paper surveys multiple empirical studies on different activities performed during the requirements phase and studies their impact on downstream development.

1 The Debate

We have been developing computer-based systems (CBSs) for over 50 years. For over 30 years, software engineering (SE) researchers and, later, requirements engineering (RE) researchers have been insisting that it is necessary to work out and specify all the requirements for a CBS before starting the implementation of that CBS. They argue that it is illogical and useless to start programming until it is known what to program. For even longer than 30 years, programmers and their managers have insisted on moving to implementation of a CBS as quickly as possible, often with little or no RE, because delaying the beginning of an implementation obviously delays completion of the implementation. Besides, even if requirements are completely specified, they undergo relentless changes during implementation and beyond [1][2][3]. So why bother specifying requirements before beginning implementation?

The SE and RE researchers reply that requirements do change, but mainly because the requirements details were not fully worked out and the specification had lots of errors. If all the requirements errors were to be tracked down and fixed before implementation, then the requirements would change a lot less during implementation. Moreover, it is known that the cost to fix an error found after deployment of software is about 200 times the cost to fix the same error found

* Adapted from a term paper the first author delivered to Steve Easterbrook for partial fulfillment of the requirements for CSC2106S, University of Toronto, Winter 2004.

during RE [4]. Therefore, if all the requirements errors are found during RE, it will cost a lot less to implement the specified CBS and the implementation will be finished a lot faster.

The programmers and programming managers reply that even if errors are removed from the requirements specifications, there will still be requirements changes. So, again, why bother specifying them before beginning the implementation. The requirements can be worked out as the programming proceeds in what are known as agile methods, for example, extreme programming, which *embraces* requirements changes [5].

The SE and RE researchers point out that even in extreme programming, there is pain dealing with requirements changes [6]. Requirement changes cause the need to refactor and the pain of refactoring leads to its postponement and thus to its growing difficulty and expense [7].

Regardless of where one stands in this debate, all seem to agree that failing to identify requirements is costly.

2 Cost of Failing to Do Requirements Engineering

The costs of failing to elicit and specify requirements are drastic, even to the points of wasting millions of dollars and of losing lives. Forsberg and Mooz [8] cite a number of examples of systems that failed due inadequate requirements gathering. These include (1) the 1996 Atlanta Olympics management system developed by IBM, which admitted that it did not understand the user requirements, and (2) the Geostationary Operational Environment Satellite project, in which the first satellite was five years late and cost three times what was estimated. Over half of projects described in the Standish Group's 1994 CHAOS Report [9] failed for requirements-related reasons. The 2001 Standish Group Report [10] ranked

- user involvement as number 2,
- clear business objectives as number 4,
- minimized scope as number 5, and
- firm basic requirements as number 6,

on its list of success factors. Observe that all of these factors are requirements related.

3 Process Models

Different software process models propose different ways to handle requirements elicitation and specification. These models can be categorized into plan-based models and agile models. In plan-based models, RE is done up front, and a majority of requirements are specified before implementation begins. In agile models, RE is done on the fly, and requirements specification and implementation are done at the same time. The waterfall model [11] is an example of a plan-based method, and extreme programming [5] is an example of an agile method.

Formal methods [12] tend to be plan-based because they require formal specification of software functionality. Formal specification removes ambiguity in requirements, uncovers possible error cases, and leaves open the option for automated proofs of the consistency of specifications and verification of the consistency of the specifications and the implementation.

Agile models [13], which have been introduced recently as an alternative to plan-based models, focus on writing code and releasing software in time-bounded iterations, allowing each iteration to respond to requirements changes. Early versions of the software can be used in a prototyping mode to elicit requirements from the users, so that their new requirements can be incorporated into later versions. Agile methods appeal to any practitioner who has a perception that "There's not enough time to do full RE."

4 Dearth of Empirical Research

There has been very little empirical research on the impact of RE activities on subsequent phases of a software project. The consequences of not effectively eliciting requirements are well-known, but what are the effects of performing different requirements activities? Moreover, with respect to the debate, programming managers are demanding from the SE and RE researchers proof with experimental data that up-front RE (UFRE) produces better software and is more cost effective than on the-the-fly RE (OTFRE).

This paper is an attempt to fulfill this need for proof. It surveys the few empirical studies the authors could find on the issue of UFRE vs. OTFRE. The authors know of no other similar survey.[1] Specifically, we take a closer look at the specify-first approach versus the prototype-first approach to RE.

Obtaining the desired empirical proof is difficult for the reasons mentioned in Section 5 below. These difficulties probably account for the scarcity of empirical studies dealing with the issue of UFRE vs. OTFRE.

5 Empirical Studies in Requirements Engineering

Empirical comparison of SE and RE methods is relatively uncommon, mainly because there is no completely satisfactory way to empirically compare two SE or RE methods. In order to do a statistically significant controlled experiment comparing two methods, we have to do many replications of applications of the methods to problems to which the methods are applicable. In order to be able to afford to do a sufficient number of repetitions in a reasonable time period, the problems used in an experiment must be small, i.e., a toy problem. Therefore, even if the results are statistically significant, it is not certain that the conclusions are generalizable. We know that methods that work on toy problems, of the size

[1] If we are wrong on either count, we would appreciate hearing about it. That is, if we have missed any empirical studies or there are other surveys, please let us know at one of the e-mail addresses given with our address on the first page of this article.

needed for a controlled experiment, do not necessarily scale up to industrial-sized problems.

In order to compare two methods on industrial-sized problems, we need to find an organization that is willing to fund doing a real problem twice, a highly unlikely occurrence. Even if such an organization could be found, the results would hardly be statistically significant because only one data point would be available. In addition, it would be hard to separate out learning effects. Moreover, it is hard to ensure that the differences in the compared methods caused the observed difference. Sackman, Erickson, and Grant's 1965 experiment [14] to show that interactive programming is more effective than batch programming failed to produce significant results because the effect of the independent variable, use of interactive vs. batch submission of the job, was drowned out by individual differences in programmers of equal experience. One experienced programmer was found to be 28 times more effective than another equally experienced programmer!

Hence, we are often left doing many introspective case studies on applications of different methods to industrial-sized examples, reporting the lessons learned in each application, and later comparing these lessons and the project histories. If an organization that carries out such case studies has been keeping project history data, it can notice when a new method yields a significantly different history, better or worse than normal.

A notable exception to this difficulty are the many empirical studies of inspection methods, e.g., by Porter, Siy, and Votta [15]. The exception occurs because an industrial-sized instance, a two-hour inspection by a half-dozen professionals of one not-too-big document, is small enough that we can afford to replicate it enough times to draw statistically significant and generalizable results.

In summary, when SE and RE methods are compared empirically, data usually come from one of three sources:

- a controlled experiment that compares two different techniques on a small problem, usually involving university students, but occasionally with professionals,
- an analysis of data gathered during or after one or more industrial software developments, or
- an introspective case study of the history of one or more industrial software developments.

Occasionally, the second and third sources are combined.

Two quantitative metrics are usually used in comparing process models: quality and productivity. Productivity is usually measured in lines of code (LOC) or thousand lines of code (KLOC) per person–month (PM) or person–day (PD).[2]

Obtaining a metric for quality is more difficult; proposed metrics include the number of defects reported in a unit of code over a time period and the evaluation of the software by an expert or a customer. Qualitative observations

[2] The usefulness and accuracy of the metric have been debated heavily, but KLOC/PM, or some multiple of it, is still the dominant metric.

are gathered often in the form of questionnaires and are useful also to understand the social implications of different requirements techniques on practitioners.

6 Up-Front Investment in the Inception Phase

The inception or study phase is the stage that occurs before software development officially begins. This phase is a part of the Unified Process [16] but has an important part in every project. The inception phase involves exploring the market, the current alternatives, and possible solutions and their feasibility and possible costs. In short, it involves exploring the requirements thoroughly.

Figure 1 contains a graph adapted from that presented by Kevin Forsberg

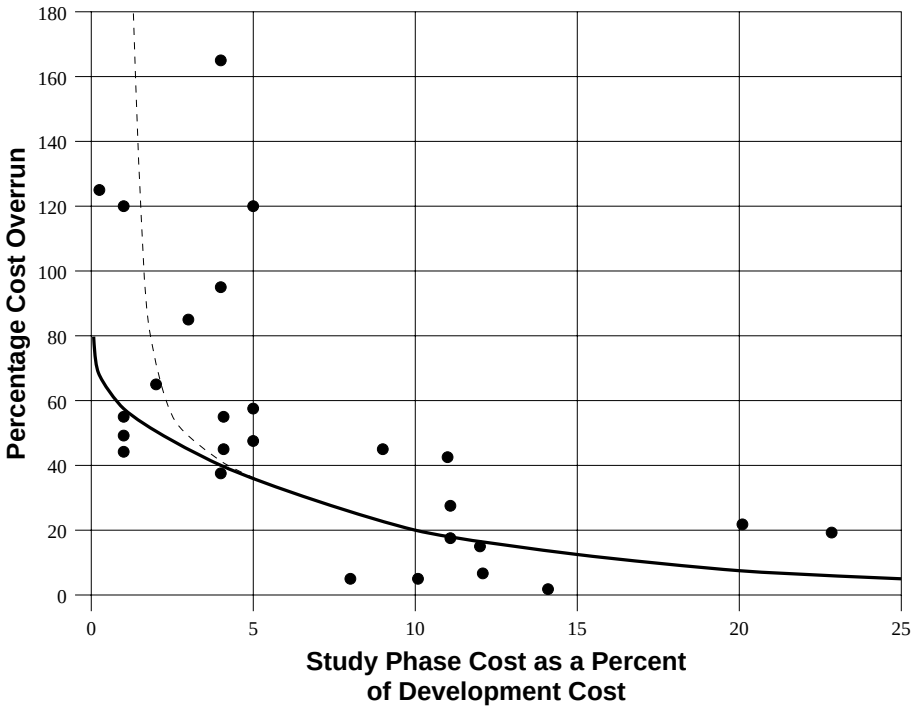

Fig. 1. Percentage Cost Overrun vs. Study Phase Cost as a Percentage of Total Development Cost in 25 NASA Projects

and Harold Mooz [8] that shows that spending more time in a project's study phase leads to a lower cost overrun. The study, performed by W. Gruhl at NASA HQ includes such projects as Hubble Space Telescope, TDRSS, Gamma Ray Obs 1978, Gamma Ray Obs 1982, SeaSat, Pioneer Venus, and Voyager. The extremes

are that if only 4% of the total development cost is spent to study the problem, the cost overrun can be as high as 165%, and if 23% of the total development cost is spent to study the problem, the cost overrun can be as low as 20%. There can be two explanations for these data.

- The more time spent studying the problem, the better is the cost estimate and thus the lower is the cost overrun.
- The more time spent studying the problem, the better is the understanding of the requirements and thus the fewer are the costly errors.

Probably, a mixture of both explanations account for the actual data.

7 Up-front Requirements Specification

Up-front requirements specification may result in the reshaping of the software development life cycle. Because the requirements are more complete, the software architecture allows functionality to be incorporated into the system with less rework. If more thought has gone into the pre-coding phases, the actual coding and integration task may end up requiring less time and effort.

7.1 Requirements Reshaping the Development Life Cycle

Berry, Daudjee, Dong, Fainchtein, Nelson, Nelson, and Ou [17] report an instance of up-front RE significantly shortening the subsequent implementation and testing. One of the authors, Lihua Ou, was implementing a WYSIWYG drawing program, `WD-Pic`, for Berry. Berry had already received prototypes from previous students and requested that Ou specify the requirements in the form of a user's manual. Based on her previous industrial experience developing commercial systems, Ou proposed the plan shown in Table 1. However, Berry, as a

Duration in Months	Development Phase
1	Preparation
2	Requirements Specification
4	Implementation
2	Testing
1	Buffer (probably more implementation and testing)
10	Total Estimated Time

Table 1. Estimated Project Time for `WD-Pic` Project [17].

stubborn customer, required Ou to spend more time on the requirements phase

Duration in Months	Development Phase
1	Preparation
4.9	Requirements Specification, 11 versions of the user's manual
0.7	Design including planning implementation strategy for maximum reuse of code and libraries
1.7	Implementation, including module testing and 3 user manual revisions.
1.7	Integration testing including 1 user manual revision and implementation changes
10	Total Actual Time

Table 2. Actual Project Time for `WD-Pic` Project [17].

to ensure that they were correct. When she spent nearly five months in requirements, she worried that the schedule had slipped beyond recovery. However, as shown in Table 2, the implementation and testing phases were much shorter than estimated, resulting in the project's surprising on-time finish.

Although the requirements phase took about 2.5 times longer than planned, the implementation and testing took much less time than expected. In addition, Ou was able to port the software from the original Sun Solaris platform to the Microsoft Windows platform, still within the original planned time, despite that the original plan was to implement on only one platform.

The main threat to validity in this case study is that the software requirements were extremely stable, and the customer was extremely aware of what his requirements were due to having used previous `WD-pic` prototypes. However, Ou did have to write eleven versions of the user's manual that was used as the specification. Some of the revisions were quite extensive, indicating large misunderstandings between her and the customer. Another threat to validity is that this project involved only one customer and only one developer.

7.2 The Luxury of Enough Time to Do Full RE

Arnis Daugulis [18] reports a significant increase in the quality of the requirements for a national power plant control system as a result of delays in start of production caused by a shortage of funds. Groups of five or six requirements analysts led by him had the luxury of time to do two revisions, i.e., three versions, of the requirements specifications. After each of the first two versions, there were not enough funds to begin to implement the specified system; rather than sit and do nothing, they decided each time to review and revise the requirements specifications one more time.

As they were doing the second version, they realized that an implementation based on the first version would have been a death-march project. When they finished the second version, its improvement over the first was clear; an implementation based on the second version would probably have succeeded.

The assumption for the first two versions was that the implementations would be from the ground up. The assumption for the third version was that the system would be built from readily available, mature off-the-shelf software that had been used in power plants elsewhere in the world. When they finished the third version of the requirements specification, its improvement over the second was clear; they felt that implementing the system using the off-the-shelf software would be quite straightforward. They got a chance to test this feeling because at that point, the funds were available.

An e-mail message from Daugulis to Berry [19] six years later indicated that the implementation had gone pretty much as expected. Daugulis said, "We launched the project, selected the supplier. Today, the system is up and running, but the project itself was not flawless. We stayed within budget, but delayed going live by almost 6 months. There were several reasons for that. As one guy from the supplier's team put it—'the technology is good but management s---s'. The spec itself was quite OK. The project proved that requirements elicited for RFP are indeed very important ...."

The total duration of the project, including the delay was about 32 months. Daugulis's team consisted of about fifteen people working over varying time periods, that is, about six full-time equivalents. The size of the vendor's team is unknown. Each version of the requirements took about one-half year, and five or six people people working occasionally, that is about 1.5 full-time equivalents. Thus, to describe this project in terms of the data reported by Forsberg and Mooz, this project had *no* cost overrun, about 18% time overrun, and RE comprised about 56% of the total project duration, well outside the x axis of the graph in Figure 1.

8 Requirements and Formal Methods

Formal methods have been shown to be viable for the specification of critical systems, for which either the cost of failure or the cost of maintenance is extraordinarily high. Formal methods are more expensive by an order of magnitude during the short term, mostly in the requirements and design phases, but many believers state that these expenses will be recovered in subsequent phases of software development. Unfortunately, there appear to be no studies that involve data analysis for an entire project using formal methods that can support or refute this claim.

8.1 Formal Methods compared to Conventional Methods in Security Gateway

Fitzgerald, Larsen, Brookes, and Green [20] performed an experiment at British Aerospace Systems and Equipment Ltd. in specifying a trusted gateway that decides whether to send communications along a secure line or an insecure line. One team, called the "conventional team", worked with conventional requirements specification and systems design methods, supported by Teamwork and

RTM CASE tools. The other team, called the "formal team", additionally used VDM-SL to describe at least the security functions in the system. To test the processes' adaptivity to changes, an additional requirement was introduced late in the system design phase.

This study unfortunately neither reports its statistical measurements nor explores the effects of the requirements specification methods on the subsequent developments.

One interesting result of the study is the number and nature of the queries about the specification posed to the customer. Approximately 40 questions were submitted by the conventional team, and approximately 60 questions were submitted formal team. The questions were passed on to a number of other experienced engineers who sorted them into categories:

- function, clarification of what the system must do;
- data, specification of the data used by the system;
- exceptions, description of what the system must do in certain situations; and
- design constraints, exploration of the boundaries of the system and of implementation restrictions.

Table 3 shows the percentage of each category question each team asked.

Question Category	Formal Team %-age	Conventional Team %-age
Function	42	60
Data	31	10
Exception Conditions	14	8
Design Constraints	14	22

Table 3. Percentage of Questions in Each Category by Teams

The formal team asked more questions dealing with data and exceptional conditions than did the conventional team. The formal team asked fewer questions about design constraints than did the conventional team. Both of these trends are seen as positive results about the requirements phase. The questions posed by the formal team can be answered and verified also internally, based on experience with and analysis of similar systems. It was thought that the conventional team left implicit a number of assumptions that the formal team made explicit.

The use of VDM-SL required about 25% more effort than using only conventional methods and resulted in a more abstract but more detailed design. There was no record of the number of faults encountered by the two groups. There was no comparison of the complexity and usability of the two specifications. The

authors do suggest that the cost overhead of using VDM-SL in addition to the conventional methods will probably be recovered in faster and more error-free design, implementation, and testing. No follow-up report has been published indicating if the additional costs were, in fact, recovered.

8.2 Impact of Formal Methods on Development

Pfleeger and Hatton [21] analyzed the data that had been gathered during the development of Praxis's Central Control Function Display Information system (CDIS) for the UK Civil Aviation Authority. The system is approximately 200,000 lines of C code. Functional requirements were developed using ER diagrams, real-time extensions of Yourdon-Constantine-style structured analysis, and formal specification languages. VDM was used to specify critical parts of the system, and the Calculus of Concurrent Systems (CCS) was used to specify concurrency.

Pfleeger and Hatton's analysis involved the comparison of data gathered about source code developed with conventional methods to those gathered about source code developed with formal methods. They looked at the numbers of faults submitted by developers, the change requests over time, the defects uncovered by unit tests, the code audits, and the number of post-delivery changes. The studied units of code implement different functions and were written and designed by four different teams:

- *Application code.* Ten developers used VDM to refine the core specification.
- *Concurrency.* One developer used finite state machines (FSM) for concurrency.
- *Local area network.* Two developers used a mix of VDM and CSS, with some proofs to uncover faults in the design.
- *User interface code.* Four developers used pseudocode to describe the user interface.

Thus, for the purposes of the following discussion, these four CDIS system components are named "VDM", "FSM", "VDM/CCS", and "informal", respectively, after the methods used in their developments. The name "formal" is used to denote the first three collectively.

Formal methods in the project did not reduce the number of fault reports during development. The formal parts had 19.6 changes per KLOC and the informal part had 21.0 changes per KLOCC, a nonsignificant difference. However, Pfleeger and Hatton noticed trends in the numbers of fault reports over time. Code designed using VDM/CSS suffered a spike in fault reports earlier than the code designed by other methods, implying that users of VDM/CSS found and fixed faults sooner. The faults were being discovered earlier than in system testing, meaning that methods of fault finding other than system testing were involved, e.g., proof. Recall that the VDM/CSS group used also proofs, which may have helped uncover faults early in the lifecycle.

Use of formal methods led also to the development of simpler, more compact code. Pfleeger and Hatton used QAC to perform coverage analysis of the code

and discovered that each of the modules in the software was relatively simple. The formally-specified models were simpler than the informally-implemented models, but it was not possible to conclude if this phenomenon was a direct result of the application of formal methods as opposed to the indirect result of a more-thorough-than-usual analysis of specifications.

There appeared also to be fewer faults uncovered in the subsequent unit testing of formally-specified modules than in the subsequent unit testing of informally developed modules, implying that formal methods assisted in the early uncovering and repairing of faults.

The number of changes required after delivery to the customer is the final metric. This metric reveals a combination of the software's reliability and the software's suitability to the customer. The number of changes, normalized by the number of modules, was 0.12 for the formally-specified parts, while that same number was 0.27 for the informally-implemented parts. In other words, the informally-implemented parts experienced more than twice the number of changes per module after delivery than did the formally-specified parts. The whole system had a remarkably low fault rate of 0.81 faults per KLOC. Pfleeger and Hatton compare this rate to previously published significantly higher rates, thus suggesting that projects using formal methods have a lower fault rate. Pfleeger and Hatton remind the reader that they were not able to determine the way the metric was calculated; thus the metric should be taken cautiously.

In conclusion, the Pfleeger and Hatton state that there is no compelling quantitative evidence that formal methods alone provide higher quality code than informal methods, because the total number of change reports for the parts developed informally and formally do not differ. However, the data show

- that faults were found sooner in the VDM/CSS parts than in the other parts,
- that fewer faults were discovered during unit testing in the VDM/CSS parts than in the other parts.

In addition, formal methods resulted in fewer changes in the code after delivery than did informal methods.

8.3 Cost-Effective Automated Verification Using Proofs

King, Hammond, Chapman, and Pryor [22] discuss the use of formal methods and compare their effectiveness in finding faults with that of traditional testing techniques. Formal methods with proofs were applied to the Ship Helicopter Operating Limits Information System (SHOLIS) designed for the UK Ministry of Defense. Z was used for the formal specification and proof of specification consistency, and code written in SPARK, a so-called verifiable sublanguage of Ada, was subjected to flow analysis and proofs of partial correctness with respect to the Z specification.

The proofs and analyses of the Z specification and the SPARK implementation were performed by four engineers. Two engineers worked on the Z specification and one of these engineers worked also with two others on the proofs about

the SPARK code. All of the engineers were experienced with formal methods, Z, and code proofs, but only one was familiar with SPARK before development started. The authors state that it is important that practitioners on a project have either academic or industrial experience in formal methods before they can use formal methods cost effectively.

Table 4 shows the metrics that were gathered for all fault-finding activi-

Project Phase	**Faults found (%)**	**Effort (%)**
Specification	3.25	5
Z proof	16	2.5
High-level design	1.5	2
Detailed design, code, & informal test	26.25	17
Unit test	15.75	25
Integration test	1.25	1
Code correctness proof	5.25	4.5
System validation test	21.5	9.5
Acceptance test	1.25	1.5
Other (Project Management, etc.)	8	32

Table 4. Faults Found and Effort Spent During Phases of the SHOLIS Project [22]

ties in the various phases of the project. The phases listed in the table did not necessarily occur in the order listed. The Z proof phase includes specifying the requirements formally and executing the proofs. Figure 2 shows the data provided by the engineers. These data show that the most efficient, in terms of the number of faults found per day of effort (FFPD), fault finding methods were the Z proof, which verified the consistency of the specification and the system validation tests which validated the entire system's correctness. In particular,

- Z proofs found 16% of the faults found in the system and found them with an efficiency of 0.54 FFPD,
- code proofs found 5.25% of the faults found in the system and found them with an efficiency of 0.21 FFPD, and
- unit testing found 15.75% of the faults found in the system and found them with an efficiency of 0.06 FFPD.
- The faults uncovered during the system validation tests were mostly the result of errors originating in the requirements phase, rather than the result of incorrect implementation of the specification.

Thus, proof was more efficient than unit testing in finding faults. The verification and validation team used the Z specification to generate about 300 pages of test cases. The proof generation for the Z specification identified 50 specification faults, while test-case generation identified only 4 faults. The actual code

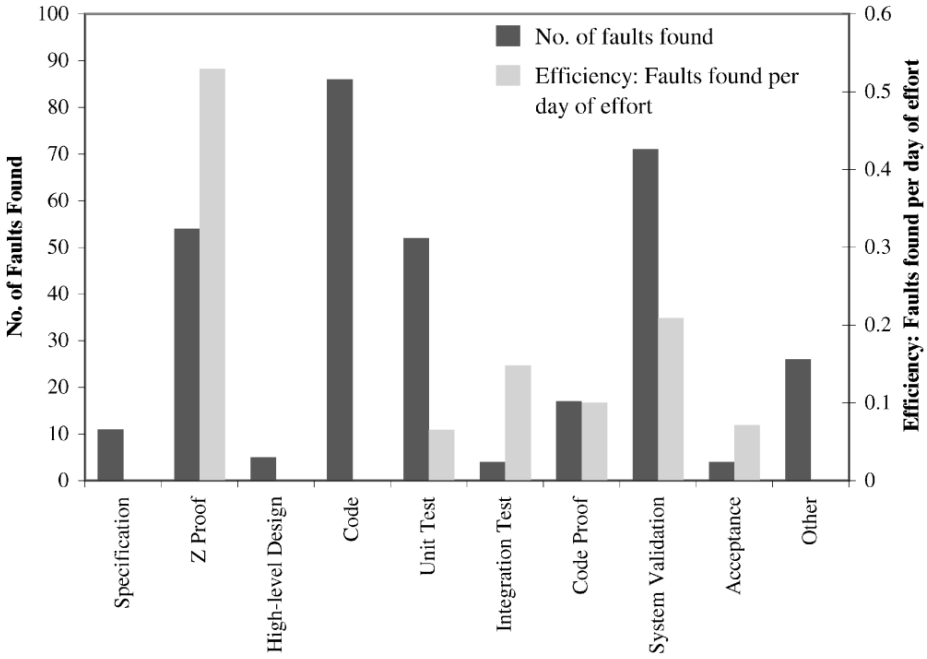

Fig. 2. Efficiency of Validation Methods in the SHOLIS Project[22]

proof revealed some faults that probably would have been difficult to locate using traditional techniques, and similarly, the actual testing revealed some faults that probably would not have been discovered with the proofs.

Proofs are more effective than testing at finding faults, but are limited in their abilities and must be supplemented by traditional verification and validation techniques. The study reveals also the relative inefficiency of unit testing, a phenomenon observed also by Pfleeger and Hatton [21]. Although there has been some empirical work on test-first programming [23], there is no conclusive evidence of its effectiveness.

9 Effects of a Requirements Process Change on Software Developers

The Australian Center for Unisys Software (ACUS) worked on improving its requirements process under the guidance of Damian, Chisan, Vaidyanathasamy, and Pal [24][25]. ACUS develops product-line software and software for stakeholders that are distributed over several continents. The process changes were initiated in August 2001, and continued over a period of 18 months. The changes included

- involvement of people from different functional teams, such as development, testing, and product information, in requirements analysis,
- drawing of context diagrams during requirements specification and high-level design,
- using a structured requirements specification document,
- using on-line sentence templates and their supporting software to record and analyze requirements, and
- improvement of their change management.

The data acquired about this project were from interviews, questionnaires, and document reviews. Numerical data about change requests were collected.

In almost all cases, the developers reported that they felt more productive, more confident in their work, and more involved in the decision-making process. Developers were less confused about their tasks and felt that communication between customers, engineers, and management had improved. Some expressed concerns about the new process involving more effort. The accuracy of cost estimation was greatly improved over past performance.

Although change requests were reported for this project, there is no mention of project size. Thus, comparisons between this project and others cannot be made. There were two changes during the requirements phase, 37 during design, 31 during coding, and 7 during integration testing. The changes resulted in a total of 158 changes to the documentation, 50 of which were in the requirements document. Most of the changes to the requirements document occurred during design and coding, suggesting that developers were able to trace back features to the requirements document. In addition, the low number of changes during

integration suggests that having produced a requirements specification helped reduce the problems that would otherwise have shown up late in the lifecycle.

In general, the changes during the requirements phase were welcomed by most of the staff and resulted in perceptions of less wasted effort, better communication, and more informed decisions.

10 Requirements and Prototyping

Prototyping is not a new concept. It has been advocated for many years. Brooks in the Mythical Man-Month said, "Plan to throw the first one away; you will anyhow." [26]. Prototyping is a large part of Boehm's spiral model, and it helps identify possible risks and alternative designs [27].

There are two kinds of prototyping: rapid, throwaway prototyping and evolutionary prototyping [28]. Rapid, throwaway prototyping is designing a prototype, usually for the purpose of requirements elicitation, and then throwing it away once the requirements have stabilized. Evolutionary prototyping is using the initial prototype as a base from which the production-quality system is constructed.

The recent trend has been toward lightweight, agile processes. Agile processes use short, iterative development cycles, usually on the order of a few weeks, in which requirements, design, implementation, and testing are all performed to create a next version. In essence, an agile process relies heavily on the concept of prototyping.

10.1 Comparison of Specifying and Prototyping

Boehm, Gray, and Seewaldt [29] performed a controlled experiment comparing specification and prototyping methods using students in a graduate-level University course.

The authors asked students to implement an interactive COCOMO model. A large portion of this project involved the design of a user interface. There were four groups that executed specification methods, with three groups consisting of three members each, and one group consisting of two members. There were three groups that executed prototyping methods, with one group consisting of three members, and two groups consisting of two members. Each group assigned to specify was required to present a requirements specification during week 3 and a design specification during week 6. Each group assigned to prototype had to demonstrate a prototype during week 5. All artifacts were reviewed by the course leaders, and feedback was provided to the groups. This experiment was not limited to RE activities; each specifying group had to write both a requirements document and a design document, while each prototyping group had to do neither. Of course, each group had to deliver running code.

After the experiment was complete, the finished projects were evaluated by the course leaders on a Likert scale from 1 to 7 on functionality, robustness, ease of use, and ease of learning. In addition, data were gathered on the sizes of the projects, expressed as delivered source instructions (DSI), and on the effort

spent on the various phases of the project. Finally, at the end of the project, each student was asked to fill out a questionnaire that asks, among other questions, which product he or she would most want to maintain. Each student wrote also a free-form essay commenting about his or her experiences in the project and what he or she would do differently if he or she were to do the project again.

The independent variables were "group size" and "group type", i.e., "specifying" or "prototyping". The dependent variables were "quality", according to the evaluation criteria; "productivity", measured in DSI/person–hour (PH); "effort", measured as total PHs; and "maintainability", based on student and instructor evaluations.

The results of the experiment show that

- prototyping groups used 45% less effort than specifying groups to develop products with 40% less code than those of the specifying groups,
- prototyping group products were rated lower than specifying group products in functionality and robustness, but higher on ease of use, and ease of learning, and
- specifying groups produced more coherent designs and more easily integrated software than did prototyping groups.

The mean effort of specifying groups was 584 PHs, compared to 325 PHs, a significant difference between the two groups. However, the programming productivity of specifying groups was 6.3 DSI/PH, compared to 5.8 DSI/PH for the prototyping groups, not a significant difference between the two groups.

Prototyping groups appeared to have a better understanding of the effort required to implement certain features. A specification group would look at a proposed new feature and say enthusiastically, "Sure, that's just another sentence in the spec.", while a prototyping group would say only, "We'll put that in if we have time."

Paradoxically, most students expressed a preference to maintain the prototyped products, although these same students indicated that the specified products were better designed and more robust.

Prototyping also led to a flattening out of effort. While the specifying groups experienced the deadline effect and put in extra effort, primarily in programming, close to the deadlines, this effect was not as pronounced in the prototyping groups. In prototyping groups, the effort surge occurred toward the end of the project in producing documentation. Prototyping groups put more effort overall than specifying groups into integration and testing, implying that specifying led to more planning and structure in the program's design than did prototyping.

The study, overall, suggests that prototyping has some strong benefits in the software development life cycle. It was easier for prototyping groups to adapt to changes and reevaluate the risk to the project. In addition, because the programming problem was primarily the design of a user interface, prototyping may have had an advantage over specifying, because a prototype is more concrete and visual and provides stronger feedback than a specification document. On the other hand, specification results in more robust programs, faster integration, and more efficient testing.

10.2 Prototyping For Better Customer Acceptance

MacCormack, Verganti, and Iansiti [30] compared data from 17 Internet software development companies over a two-year period, presumably from 1995 until 1997, to validate the usefulness of prototyping and agility in the development process. At the time of the study, they chose the volatile and unpredictable market of Internet software, and argue that a model that overlaps the different development phases results in higher quality software, especially when requirements are unstable due to early feedback from the customer.

The authors propose that there are three important milestones in a project's life,

- the first customer prototype, which serves as a proof of concept and can be used to elicit user and customer feedback;
- the first integration; and
- the first customer beta test, in which end-users become heavily involved with the testing effort.

The authors' hypothesis is that the earlier a prototype is released, the better is the quality of the product derived from the prototype.

The authors used a questionnaire and interviews to collect information about 29 products released by the 17 Internet software development companies. The questionnaire requests data which are used as independent variables. These data include:

- the percentage of product functionality that has been developed when each of the three milestones has been reached;
- the relative amount of resources spent on the lifecycle phases: project management, architectural design, implementation, and testing; note that "RE" is not listed as a phase;
- the generational experience of each of the developers on the project. *Generational experience* means the amount of experience in product generations, which differentiates a developer who has worked on only one project for many years from a developer who has worked on many complete projects over the same number of years.

Product quality was evaluated using a Delphi method. Because the products vary, they cannot be directly compared. Thus, a panel of fourteen industry observers from magazines and Internet sites compared each product to similar existing products.

The analysis uses regression to indicate the relationship between the independent variables and product quality. Higher architecture design effort usually leads to higher product quality. Also earlier market feedback usually leads to higher product quality. A variance analysis shows that (1) architecture design effort, (2) market feedback, and (3) generational experience account for 54.2% of the variance in product quality.

This study shows the value of prototyping as a requirements elicitation technique and argues that a product that is released earlier as a prototype ends up

with a higher level of customer acceptance than do other products. The study emphasizes high-quality architectural design, arguing that such design allows more parallelization of development and allows new requirements to be implemented easily in the system.

Threats to the validity of this study include the use of a questionnaire to acquire data. The study relies on companies to report, after the fact, when their prototypes were finished. The metric "percentage of functionality" is not standardized, is not easily measured, and relies on the memory and perspectives of each who fills out a questionnaire.

10.3 Requirements Specification Not Always Productive

MacCormack, Kemerer, Cusumano, and Crandall compare different elements from 29 projects submitted to them for analysis between 2000 and 2001 [31]. Also these data were acquired using a questionnaire.

The variables are "percentage of completeness of requirements specification document", "percentage of completeness of design specification documents", "whether formal design reviews and formal code reviews were present", "whether iterative development and early prototypes were employed", "whether daily builds were performed", and "whether regression and unit tests were run regularly".

The authors compare programmer productivity, expressed as LOC per PD, and product quality, expressed as the number of customer-reported defects per month per million LOC averaged over the first 12 months after product deployment.

The presence of a requirements specification increased productivity, and there is a weak indication that a requirements specification reduced the defect rate.

Also design reviews led to fewer defects. Unfortunately, there is no mention of any relation between requirements specification reviews and the number of defects.

Early prototyping was found to lead to both increased productivity and a decreased defect rate. This observation contradicts the conclusion, by Boehm, Gray, and Seewaldt [29], that code productivity for prototyping was not significantly different from code productivity for specifying.

The authors then factor the effects into a multivariate model to build regression models for both productivity and quality. Unfortunately. they do not describe their statistical procedure in enough detail for others to comment on their methods. Their multivariate model removes the effects of requirements specifications on productivity, suggesting that other factors, e.g., prototyping, may compensate for incomplete requirements specification. Prototyping does contribute to a lower defect rate and to increased productivity.

It is unfortunate that the statistical methods for the author's multivariate model is not explained. It is possible that the authors may have not taken into account the interactions between different processes in their model. For example, if each of performing unit testing and completing a requirements specification

leads to reduced defect rates, it does not make sense to study the processes separately due to their possible interactions.

11 Discussion

There are too few studies in this paper to be able to make a general statement about the effects of RE on software development, but some trends can be observed.

The presence of a requirements specification appears to improve the quality of code. A specification reduces confusion and allows developers to make more informed decisions, as reported by Damian [25]. Following the production of a complete requirements specification, higher developer productivity and higher resulting code quality were observed by Pfleeger and Hatton. [21], MacCormack, Kemerer, Cusumano, and Crandall [31], and Boehm, Gray, and Seewaldt [29]. However, MacCormack, Kemerer, Cusumano, and Crandall later minimize the effects of up-front requirements specification on developer productivity when they present their multivariate statistical model.

Specification tends to reduce effort required in implementation, implementation, and testing. King, Hammond, Chapman, and Pryor [22] report proofs as an effective tool to complement testing. Boehm, Gray, and Seewaldt [29] observe reduced effort required by students to integrate and test code. Berry, Daudjee, Dong, Fainchtein, Nelson, Nelson, and Ou [17] observe a remarkable change in the shape of the development life cycle, resulting in more being done in less time than estimated. Pfleeger and Hatton [21] observe a shifting of defect discovery to earlier in the software development lifecycle when formal methods with proofs are employed.

Using formal methods or analyzing requirements in detail appears to result in simpler code and in the discovery of fewer defects after delivery, but not necessarily reduced defect discovery during development [21]. This observation suggests that developers make the same number of mistakes overall no matter what methods are used. However, with formal methods or detailed analyses of requirements, the defects are discovered and repaired earlier, repairs are cheaper, and future releases have fewer defects. Formal methods are also an effective method of finding faults in requirements specifications so as to prevent costly defect repair later in the lifecycle.

The relative ineffectiveness of unit testing for finding defects in projects using formal methods was observed by King, Hammond, Chapman, and Pryor [22] and Pfleeger and Hatton [21].

Prototyping results in more usable software and appears to be an adequate replacement for up-front requirements specification. Releasing a prototype earlier to the customer leads to more end-user satisfaction [29][30][31].

There appears to be truth to the folkloric statement that the software being built determines the building process. In an established, well-understood domain, such as telephony, it makes sense to use formal methods, based on the known models. Formal methods are generally employed when the cost of either change

or failure is high, but there is some testimony on how using formal methods has resulted in cost-savings or has resulted in delivery of code with very few defects with no or only small cost overruns [12][32].

Agile methods appear to be more appropriate for business information systems for which the requirements are not as defined and the domain is more volatile. Adopting a prototype-first method allows the developers to elicit feedback from the customer and better scope the project. It appears to be especially important to prototype user interfaces because requirements specification techniques have a difficult time describing something as fuzzy as human–computer interaction.

The research presented in this paper cannot be generalized to requirements specification, formal methods, or agile methods. There are too few studies to make a broad conclusion, and many of these studies cannot be compared directly. A standard benchmark to compare the effects of different activities on software development is required before a more complete analysis can be made.

11.1 The Manager's Dilemma

Even if proof exists that UFRE is better, the perceived risk of UFRE and past experience works to cause managers to abandon UFRE, to begin implementing, to hope that OTFRE will work *this* time, and ultimately to never know if UFRE would have worked better.

The experience of most managers is that RE accounts for only about 10% of the lifecycle up until deployment. Suppose that a manager has estimated realistically[3] that, given the resources available, ten months suffices to develop a particular CBS. Suppose she has decided to try UFRE for the first time but subconsciously sees RE being completed in only one month. Good RE naturally requires longer and begins to spill into the third month. The manager begins to panic, because her gut says, "We've spent three months on RE and we are not done yet! My experience tells me that 10 months is not enough, that we will need at *least* 30 months (multiplying RE time by 10), probably 40! Oh no! We'll never make the 10-month deadline. We have to get moving." So she stops the UFRE and begins implementing with what requirements are already understood, counting on OTFRE to fill in on the rest of the details. Due to the general competence of her team and their heroic overtime efforts, the implementation is finished in only 15 months, a slip of only 50%, which is sometimes considered not bad. However, along the way an endless stream of faults kept popping up, and everyone working on the project swears never to undergo this experience again.

The irony is that had she waited another month or two until the RE were finished and a good specification had been written, the project would have made the 10-month deadline, because much fewer requirement errors would have been discovered during implementation, and the implementation and testing would

[3] That is, the estimate is not impossibly optimistic, failing even if everyone is super competent and is making a super-heroic effort.

have gone very quickly. The effect of the two-order of magnitude reduction in cost to fix an error discovered during RE is extremely powerful in reducing the time and cost of implementation dramatically.

The double irony of all this is that the manager who has gone through this experience simply cannot believe that the programming would have been finished by the end of the 10 months if she had let RE continue the additional month or two that it needed, simply because believing this alternate history requires suspending all beliefs formed from the bad experience, which gave rise to all the errors that caused the project to require 15 months of sustained overtime work.

11.2 Future Work

In the RE field, there is still a debate as to the effectiveness and the cost of formal methods. It has generally been established that formal methods cost more in the requirements phase, but there are still few data about the effects formal methods have on downstream development. A study that gathers comparative statistics on the costs of using formal methods in all stages of the software development cycle is needed.

Much of the empirical evidence appears to be in favor of prototyping, but issues that have not been addressed in the empirical studies include software maintenance, developer training, and adaptation to new requirements. These are all related because they require some kind of knowledge transfer or tolerance to change.

No research has been performed studying development processes that incorporate both specifications and prototyping. Specifications and prototyping are not mutually exclusive activities, and may be able to complement each other. With the development of CASE tools, it should be possible for a group both to write a formal specification and to generate an executable prototype based on this specification that can be used to elicit more accurate requirements from users.

Finally, more work is required in empirical SE, especially in establishing a benchmark that can be used to measure process models.

One particular need for empirical studies of RE is a way to measure quickly the effectiveness of an RE method or to compare two RE methods. Strictly speaking one has to wait for years to see the entire history of what happened after completing the requirements specifications yielded by the methods. He needs to see what happened during the subsequent development, to see the bug reports and maintenance history, etc. However, then he will not be able to report conclusions in a timely fashion. Peter Merrick has a neat metric that he used for measure the predictive ability of a requirements pattern language [33], namely a count of change requests during RE and during subsequent phases. This metric is based on the idea that fewer change requests is a symptom of a better requirements specification. This metric, which can be be gathered from the very beginning of a project, seems to be a useful way to measure RE effectiveness.

References

1. Jackson, M.A.: Problems and requirements. In: Proceedings of the Second IEEE International Symposium on Requirements Engineering, York, UK, IEEE Computer Society Press (1995) 2–8
2. Lehman, M.M.: Programs, life cycles, and laws of software evolution. Proceedings of the IEEE **68** (1980) 1060–1076
3. Berry, D.M.: Software and house requirements engineering: Lessons learned in combating requirements creep. Requirements Engineering Journal **3** (1998) 242–244
4. Boehm, B.W.: Software Engineering Economics. Prentice-Hall, Englewood Cliffs, NJ (1981)
5. Beck, K.: Extreme Programming Explained: Embrace Change. Addison-Wesley Pub Co. (1999)
6. Berry, D.M.: The inevitable pain of software development: Why there is no silver bullet. In: Radical Innovations of Software and Systems Engineering in the Future, Proceedings 2002 Monterey Conference, Selected Papers. Volume 2941 of LNCS., Berlin, Springer (2004)
7. Elssamadisy, A., Schalliol, G.: Recognizing and responding to "bad smells" in extreme programming. In: Proceedings of the Twenty-Fourth International Conference on Software Engineering (ICSE2002), Orlando, FL (2001)
8. Forsberg, K., Mooz, H.: System engineering overview. In Thayer, R.H., Dorfman, M., eds.: Software Requirements Engineering. IEEE Computer Society Press, Los Alamitos, California (1997) 44–72
9. Anonymous: The chaos report. Technical report, The Standish Group (1994) http://www.standishgroup.com/sample_research/index.php.
10. Anonymous: Extreme chaos. Technical report, The Standish Group (2001) http://www.standishgroup.com/sample_research/index.php.
11. Royce, W.: Managing the development of large software systems: Concepts and techniques. In: Proceedings of WesCon. (1970)
12. Wing, J.: A specifier's introduction to formal methods. IEEE Computer **23** (1990) 8–24
13. Anonymous: Principles: The agile alliance. Technical report, The Agile Alliance (2001) http://www.agilealliance.org/.
14. Sackman, H., Erickson, W., Grant, E.: Exploratory experimental studies comparing online and offline programming performance. Communications of the ACM **11** (1968) 3–11
15. Porter, A.A., Siy, H.P., Votta, L.G.: A survey of software inspection. Advances in Computers **42** (1996) 40–76
16. Larman, C.: Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process. Second edn. Prentice-Hall, Upper Saddle River, New Jersey (2002)
17. Berry, D.M., Daudjee, K., Dong, J., Fainchtein, I., Nelson, M.A., Nelson, T., Ou, L.: User's manual as a requirements specification: Case studies. Requirements Engineering Journal **9** (2004) 67–82
18. Daugulis, A.: Time aspects in requirements engineering: or 'every cloud has a silver lining'. Requirements Engineering Journal **5** (1998) 137–143
19. Daugulis, A.: Private communication by e-mail. (2004)
20. Fitzgerald, J.S., Larsen, P.G., Brookes, T., Green, M.: Developing a security-critical system using formal and conventional methods. In Hinchey, M.G., Bowen, J.P., eds.: Applications of Formal Methods. Prentice Hall, London (1995) 333–356

21. Pfleeger, S.L., Hatton, L.: Investigating the influence of formal methods. IEEE Computer **30** (1997)
22. King, S., Hammond, J., Chapman, R., Pryor, A.: Is proof more cost-effective than testing? IEEE Transactions on Software Engineering **26** (2000) 675–686
23. Müller, M.M., Hagner, O.: Experiment about test-first programming. IEE Proceedings on Software **149** (2002) 131–136
24. Damian, D., Zowghi, D., Vaidyanathasamy, L., Pal, Y.: An industrial experience in process improvement: An early assessment at the australian center for unisys software. In: Proceedings of the 2002 International Symposium on Empirical Software Engineering (ISESE'02), Nara, Japan (2002) 111–126
25. Damian, D., Chisan, J., Vaidyanathasamy, L., Pal, Y.: An industrial case study on the impact of requirements engineering on downstream development. In: Proceedings 2003 International Symposium on Empirical Software Engineering (ISESE'03). (2003)
26. Frederick P. Brooks, J.: The Mythical Man-Month: Essays on Software Engineering. Addison-Wesley Longman Publishing Co., Inc., Boston, MA (1978)
27. Boehm, B.W.: A spiral model of software development and enhancement. In Thayer, R., ed.: Software Engineering Project Management. IEEE Computer Society Press, Los Alamitos, California (1987) 128–142
28. Davis, A.M., Bersoff, E.H., Comer, E.R.: A strategy for comparing alternative software development life cycle models. IEEE Trans. Software Engineering **14** (1988) 1453–1461
29. Boehm, B.W., Gray, T.E., Seewaldt, T.: Prototyping versus specifying: A multiproject experiment. IEEE Trans. on Software Engineering **10** (1984)
30. MacCormack, A., Verganti, R., Iansiti, M.: Developing products on 'internet time': The anatomy of a flexible development process. Management Science **47** (2001) 133–150
31. MacCormack, A., Kemerer, C.F., Cusumano, M., Crandall, B.: Trade-offs between productivity and quality in selecting software development practices. IEEE Software **20** (2003) 78–85
32. Faulk, S., Finneran, L., Jr., J.K., Shah, S., Sutton, J.: Experience applying the CoRE method to the Lockheed C-130J. In: Proceedings Ninth Annual Conference on Computer Assurance (COMPASS '94), Gaithersburg, MD (1994) 3–8
33. Merrick, P., Barrow, P.: Testing the predictive ability of a requirements pattern language. Requirements Engineering Journal (2004) to appear