# A two-step methodology to reduce requirement defects

Robert J. Kosman

*Naval Undersea Warfare Center Division, Code 3122, 1176 Howell Street, Newport, RI 02841-1708, USA*
E-mail: kosman@a1.vsdec.nl.nuwc.navy.mil

Defects are introduced into a software product during every phase of software development. A major source of defects that is often overlooked is requirements generation. Requirement errors discovered in later phases of the software development process are the most costly to correct because all phases of software development are usually impacted. Requirement defects can be categorized into two main types: 1) specification generation errors; and 2) unwanted/unnecessary/incorrect user functionality. This experience report presents the results of incorporating a two-step methodology which combines Operational Demonstrations of the user interface and Requirement Inspections on software requirement specifications. The two-step methodology addresses and corrects both types of requirement defects. Results from this experience support the premise that cost reduction and quality improvement can be obtained using a combined Operational Demonstration and Requirement Inspection development methodology for software requirements.

## 1. Introduction

It is no mystery that software development costs are rising and budgets are shrinking. Therefore, software organizations are searching for ways to reduce development costs without sacrificing quality. Many organizations are investing in computer aided software engineering (CASE) tools with the hope that a more structured development environment and extensive traceability will produce better code in less time. CASE tools can improve software development, but training costs and learning curves are often high, which can lead to frustration.

One way to reduce software development costs is to eliminate rework caused by defects. Defects are introduced into a software product during every phase of software development. Detecting and correcting defects during later phases of software development, or after product delivery, can be extremely costly. A major source of defects that is often overlooked is requirements generation. Requirements are often written in parallel with software design and coding, or written quickly with little or no concern for quality. Requirements written in this manner often result in significant software rework, causing schedule slippage due to late detection of requirement defects and high requirement specification volatility. Requirement errors, discovered in later

© J.C. Baltzer AG, Science Publishers

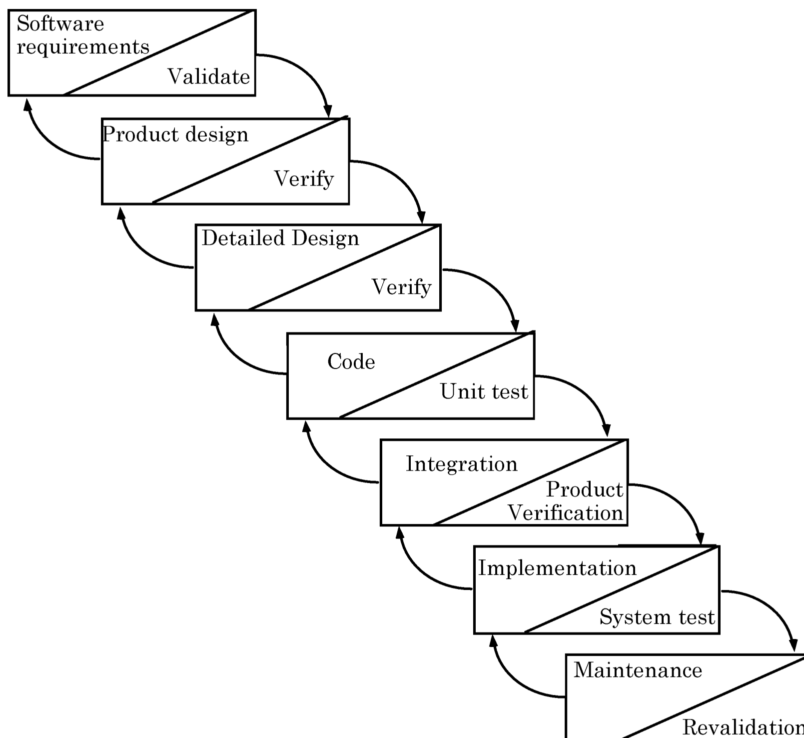


Figure 1. “Typical” software development.

phases of software development, are also the most costly to correct because all phases of development are impacted. Estimates found in literature indicate that detecting and correcting a defect in the requirements phase costs one percent of the cost of detecting and correcting the same defect during later design phases or after delivery [Beohm 1987; Pleeger 1992]. This is evident from Figs. 1 and 2. Figure 1 presents the “conventional” waterfall software development model [Royce 1987], which can be found in any software development textbook. Figure 2 presents a “realistic” waterfall software development model, which incorporates requirements rework. As reflected in Fig. 2, if the requirements specifications are late, written poorly, or full of defects, software engineers will discover these defects throughout the software development process (hopefully), but at a high cost. Even if the specifications are written well, with few defects, there is still the possibility that unnecessary, unwanted, or incorrect functionality was specified in the document. This type of requirement defect also is costly to correct because it results in wasted effort writing, designing, coding, testing, and documenting functionality that the user does not need, want, or can use.

All requirement defects can be categorized into one of two main types. The first type is specification generation defects, consisting of inconsistencies, ambiguousness, typographical errors, equation errors, input/output mismatches, missing and incorrect data, etc. This defect type occurs when translating user requirements into

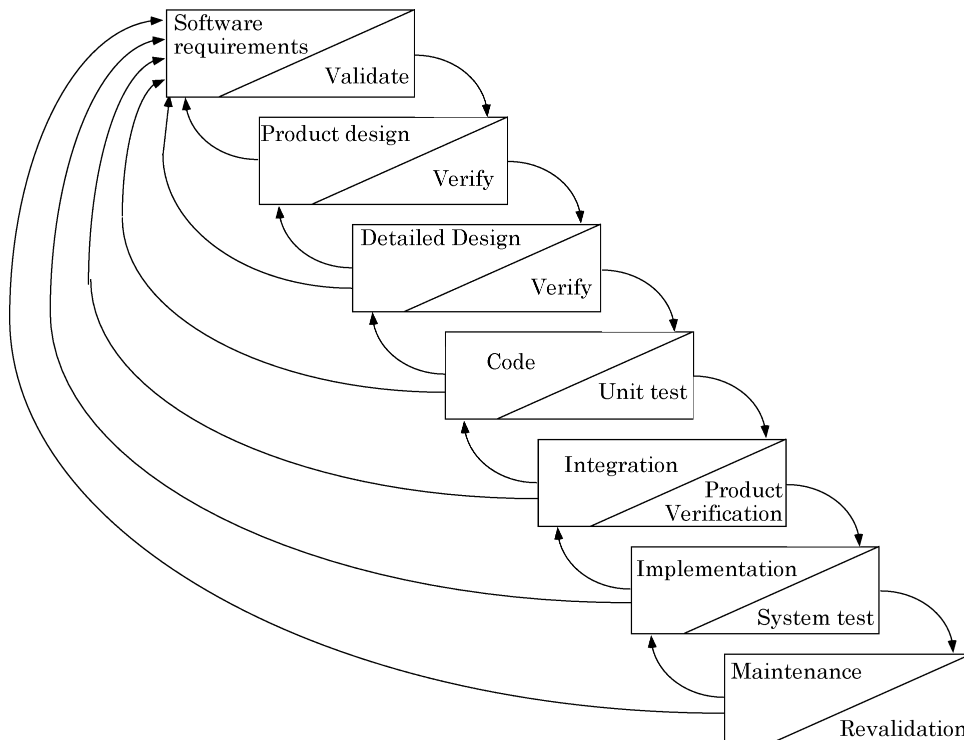


Figure 2. "Realistic" software development.

specifications. The second type is unnecessary, unwanted, or incorrect functionality written in the specification and implemented in the Graphical User Interface (GUI), often referred to as "bells and whistles," but also includes such things as omitted user requirements, unusable GUI processing, incorrect GUI processing, poor GUI designs and/or flow, etc. In order to generate quality specifications, both types of requirements defects must be addressed.

This experience report presents the results of utilizing a two-step methodology designed to address and correct these two types of requirement defects. The methodology combines Operational Demonstrations (OP-DEMOs) – a form of Software Rapid Prototyping – and Requirement Inspections (RIs) – a form of Software Inspection. OP-DEMOs ensure that users obtain operability features to satisfy their needs, while RIs ensure that developers obtain stable, quality requirement specifications to meet their needs. The two-step methodology can be applied to any system (i.e., business, mission-critical, government, military, etc.) to be developed which will have a GUI and will require some type of requirements specification to be developed. The GUI could consist of as few as one screen or as many as a hundred screens. If there is no GUI, then an OP-DEMO is not required. Having a GUI, however, is not a prerequisite of RI. RI can be performed on a requirement specification regardless of whether or not the system has a GUI. The combined use of OP-DEMOs and RIs is simply

a process to generate more accurate, usable, understandable, and stable requirement specifications in a shorter period of time. Stability in requirement specifications is especially important for mission-critical and military systems, which require specifications up-front and static before software development begins. For those systems that follow an evolutionary development approach or do not require static requirements, the two-step methodology is still applicable because it can provide a more accurate, representative software baseline and specification from which to evolve from, possibly eliminating one or more evolutions of development. The combined OP-DEMO and RI methodology can be applied to almost any software system development. All that is required is a GUI and the need to generate a requirements document.

The results in section 4 were obtained by collecting requirements defect data from two consecutive software product deliveries. The first delivery did not utilize the OP-DEMO and RI methodology, but the second did. For both deliveries, metric data were collected and analyzed. The results support continued use of the OP-DEMO and RI two-step methodology.

## 2. Software prototypes

### *2.1. Software rapid prototyping*

Before one page of any specification is written, user requirements must be identified, clarified, and stabilized. Software rapid prototyping [Hekmatpour 1987] was designed for that purpose. Poorly understood or unclear requirements are quickly developed on the target hardware in order to clarify, refine, and approve them before developing "production" code. Requirements that are understood are not prototyped. The prototype can also be used to experiment with software design alternatives. After the prototype is complete and answers have been gained, full-scale development can commence.

Properly implemented, the GUI for a software product or system can be defined and demonstrated. There are some mistakes often made with Software Rapid Prototyping that must be avoided. These mistakes, along with a solution, are presented below.

### *2.2. Prototyping mistakes*

Care must be exercised during the Software rapid prototyping phase to ensure that prototyping does not lose focus of its intended purpose, which is to investigate areas of uncertainty in the system functionality and determine the best solutions. There are several problems that an organization needs to be aware of when developing a "prototype" software system:

1. Developing production code with full system functionality.
2. Fixing prototype code in maintenance phase.

3. Evaluating the prototype with the wrong personnel.
4. Involving the end-user not until later development phases, usually during system integration or after delivery.
5. Utilizing target hardware instead of a hardware suite that supports the fastest prototype development.

Prototypes developed experiencing these problems result in inferior "production" code due to incomplete design decisions, hastily written code, and lack of end-user input.

Prototypes are supposed to be developed fast, by definition. Consequently, if an organization tries to quickly develop a prototype and then deliver it as a production system – the first major problem with software rapid prototyping – it will be poorly designed and full of software defects. Also, because the system is full of defects and poorly designed, changes required during development will take longer to make and cause many integration problems. Furthermore, many defects will have to be corrected causing even more development problems. The bottom line is that prototype defects and poor design cause a "thrashing" effect – defects require rework, but the rework results in more defects, which require rework which causes more defects, etc. What was initially supposed to be a fast prototype turns into a long, "bug" fixing development.

The second major problem is the final result of the first major problem. Many of the defects identifed but not fixed during development, probably because of schedule constraints, must now be fixed in the maintenance phase. Unfortunately, defects corrected during maintenance are up to 100 times more costly to correct than those found in earlier phases. Delivering prototype code as production code does not reduce software development costs but actually increases software costs.

The third major problem is not utilizing the proper personnel during the prototyping phase. The main purpose of a prototype is to determine what requirements the user wants. If the user community is not involved during the prototype evaluation, how can this determination be made? Yet, this mistake is often made. The user community, i.e., operating guidelines, training, manuals developers, and actual users, are usually not involved until it is too late to make changes in the product, which is the fourth major problem identified above. Involving the user community during the prototyping phase can identify required functionality, eliminate unnecessary functionality, correct erroneous processing, and simplify the GUI before any specification is written or production code is developed. The wrong concept of prototyping results in unnecessary/unwanted/incorrect functionality being designed, coded, tested, and documented, which increases software development costs.

Finally, the last major problem often made with prototyping is to always use the target hardware as the prototype platform, which is usually done because prototype code will become "production" code. Prototypes are supposed to be developed fast. If the target hardware does not have tools to support fast GUI development, then a different hardware suite should be utilized, provided the same GUI concepts are

supported by both hardware suites. By using the appropriate tools, a software prototype should be developed and evaluated in a matter of weeks, vice months.

### *2.3. Operational demonstration*

A prototype is a working model, by definition. However, when "software" and "rapid" are combined with "prototype," many organizations lose sight of the real purpose of the prototype. They think that Software Rapid Prototyping is a way to develop production code quickly, which is the wrong concept of software prototyping. OP-DEMO is intended to counter the problems incurred with Software Rapid Prototyping. OP-DEMO is very similar to Software Rapid Prototyping except the name has been changed to reflect its intended purpose. Because the intended audience for prototyping is suppose to be the user community, and a prototype is really a demonstration and evaluation tool, a better name for Software Rapid Prototyping is Operational Demonstration – OP-DEMO. Also, the OP-DEMO is not thrown away as is done with conventional prototypes [Davis 1992], but is used as a working model for the developers and users to update, as necessary, to reflect new user ideas and concepts. An OP-DEMO is a visualization of all user requirements and a demonstration of the GUI, not just the unclear or misunderstood user requirements. Screen flows, button pushes, screen content, color usage, data content, processing flows, etc., are built using GUI builder tools and then demonstrated. Software Rapid Prototypes are built as quickly as possible, but usually on the target hardware suite. OP-DEMOs, however, are built on whatever hardware suite has the most suitable GUI builder tools, so that evaluating and refining the GUI can be performed within two to three weeks. The OP-DEMO does not have to actually process any real data, however, actual processing could be integrated into the OP-DEMO if algorithms need to be evaluated. This type of OP-DEMO is more functional and takes a little more time to develop, but sometimes is necessary to determine user requirements. Finally, OP-DEMO code is always viewed as non-deliverable.

OP-DEMOs and software prototyping share many benefits:

- Eliminating unnecessary and unwanted functionality.
- Clarifying required processing.
- Identifying priorities of processing to be implemented.
- Stabilizing the GUI quickly.
- Involving the user community during requirements phase.
- Promoting a team oriented development approach.
- Identifying areas of confusion or uncertainty early, so that developers can plan for possible changes during development.

Outputs from the OP-DEMO phase used as inputs to the Requirement Inspection phase of requirements development are:

- A working, stable GUI model for the system.

- An understood set of user functionality to be developed.
- Areas of uncertainty identified.
- Organizational agreement of system functionality.

## 3. Requirement inspection

Once the GUI has been demonstrated and all organizations agree on the functionality to be implemented, the translation of user requirements to software requirement specifications must be executed. The requirements generation phase is easier and proceeds quicker after required functionality has been identified and prioritized. During the requirements generation, all system and/or software requirements are documented in some type of specification. Specifications describe not only the user interface and processing, but also algorithms, inputs, outputs, error conditions, error handling, reliability, survivability, redundancy, memory usage, hardware limitations, or any other requirement of the system. Usually, the more mission-critical the system, the more formal the specifications. For example, for mission-critical military systems, system requirements are specified in a high level document called a Prime Item Development Specification (PIDS) and low-level software requirements are specified in a Software Requirements Specification (SRS), both of which are written before software development begins. This formal requirements development is intended to capture and document all the requirements to be developed. Non-mission-critical systems usually have less formal requirement specifications, such as User Requirement Documents (URDs), but these still describe what requirements the system needs to satisfy. RI can be applied to both types of requirement specifications.

After software requirements are written, specifications must be reviewed for correctness. RIs will ensure that specifications reflect user interface requirements, are understandable, are stable, and virtually free of defects.

RIs are Fagan's inspection process [Fagan 1976] applied to requirement specifications. Inspection is a highly structured, rigorous process performed by a trained team of reviewers, whose main purpose is to detect errors in the product. Inspection can drastically reduce the number and severity of defects passed from one development phase to the next, and ultimately to the user. A product, such as specifications, detailed designs, or code, is distributed to team members at an overview meeting and then team members individually prepare for the Inspection meeting. Defect checklists, which contain examples of common errors and how to classify errors, are distributed to maximize error detection efficiency. During the Inspection meeting, the product is paraphrased ("read") by a Reader. Paraphrasing rates should be slow enough so that no defects are missed. All defects are documented and categorized. After the Inspection meeting, the product's author corrects the defects. The Inspection process is repeated as often as necessary until the product meets some predefined exit criteria, at which time the next phase of the product's development can commence. Properly applied, inspections can detect up to 85% of all defects in a product [Fagan 1976; Kolkhorst and Macina 1988; Russell 1991].

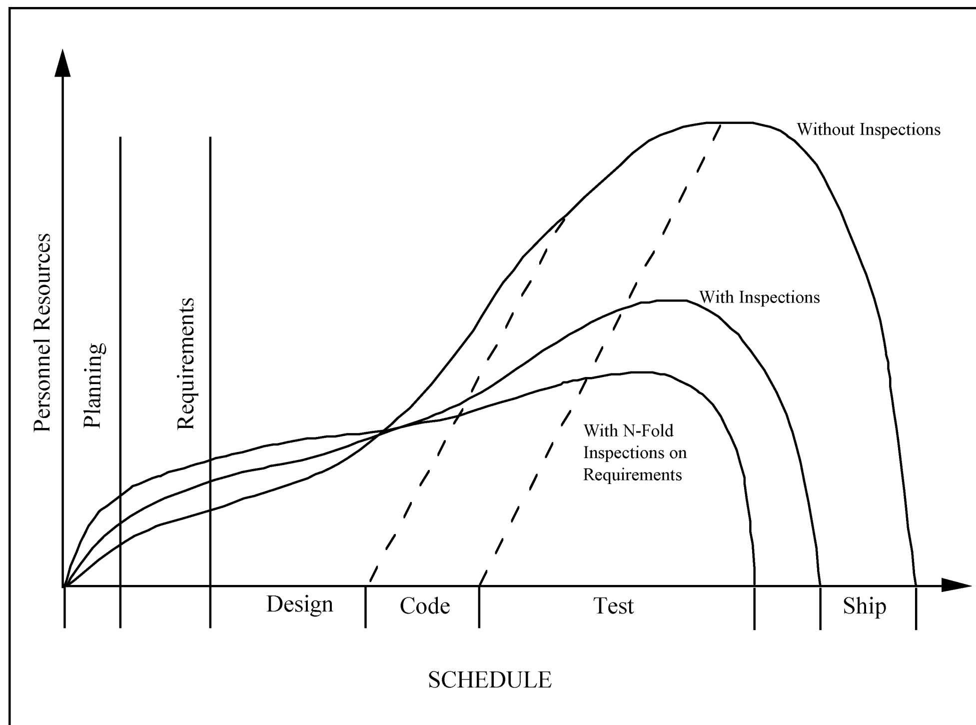


Figure 3. Impact of RIs on development (modified from [Fagan 1986]).

If the proper personnel are involved in the RI, at least an 85% defect detection efficiency should be attained. The best RI team consists of a software engineer responsible for developing the system, a systems engineer responsible for developing requirements (author of the requirements specification), an operating guidelines/training person responsible for writing user and training documentation, a software tester responsible for performing independent verification and validation (IV&V), and an end-user. To further improve defect detection efficiency, multiple inspections teams could be utilized. This technique, studied by Martin and Tsai [1990] and referred to as $N$-fold inspection, utilizes $N$ independent teams to inspect the same product. Its primary use is to detect defects that might not be found by a single inspection team, which would further reduce development costs. The requirement defect detection efficiency documented by Martin and Tsai [1990] was 84% – which is comparable to defect detection efficiencies documented for design and code inspections [Fagan 1976; Kolkhorst and Macina 1988; Russell 1991]. The impact of $N$-fold requirement inspections on software development is depicted in Fig. 3. The benefits of RI are:

- Ensures user requirements are accurately specified.
- Ensures developer's requirements are accurately specified.
- Corrects defects in a single pass instead of in an iterative method.
- Detects and corrects defects before any software is written.

- Identifies areas of improvement for future specification generation by using metric data.
- Promotes low cost per defect detection ratio.
- Produces less rework in later development stages, which reduces development costs.

Combining OP-DEMOs and RIs is very simple. The OP-DEMO phase begins only after the need for a system has been identified. For example, the United States (US) Navy determines that it needs a system to detect enemy torpedoes. The Navy would then perform some investigations into its feasability and gather some high-level requirements on system content and format. If the system is feasable and funding is available, the Navy may decide to fund development of the torpedo detection system. At this point, OP-DEMOs would be used to demonstrate the initial system content and formats and then used to refine these controls and formats based on demonstration feedback. After a certain period of time, a stable GUI would be developed which would identify all required operator processing, in a consistent, simple manner (hopefully). OP-DEMOs would also identify areas of operator processing that are controversial or volatile, which would alert developers to potential problem areas during development. This is important because knowing about problem areas well in advance allows for better management of the system, resulting in lower development risk. After the GUI has been stabilized, writing requirement specification(s) would be performed. All requirements would be specified, not just those dealing with the GUI. When the specifications are ready to be reviwed, $N$-fold RIs would be performed. The number of RI teams would be determined based on funding, schedule constraints, importance of finding requirement errors, and management decisions. All defects should be corrected before beginning preliminary software design.

The combined use of OP-DEMOs and RIs has many advantages over using only one or the other when generating software requirements. The main benefits of the two-step methodology are:

- Software requirements generation costs are lower and quality is higher.
- Metric data collection provides feedback on how to further improve the process.
- Software developers have confidence in requirements and can concentrate on design, code, and test phases.
- Requirement defects uncovered during development and computer testing are less severe and result in reduced rework effort.
- Team oriented approach to requirements development promotes better coordination and communications among organizational groups.
- User community involvement allows for parallel, not incremental, development of user documentation.
- Positive impact on the software development process.

## 4. Case study

The remainder of this experience paper presents results from two consecutive software deliveries for a US Navy software system. The system was not mission-critical but still had to follow strict development guidelines established by the US Navy. The first delivery did not utilize the two-step requirement defect reduction methodology, but the second delivery did. Metric data were collected during Inspection meetings and computer testing for both deliveries. The following metric data were collected.

1. Defect severity – major or minor. A defect is a condition that decreases the product's maintainability or causes rework during any phase of development. A major defect is one in which a program trouble report (PTR) – a "formal" defect tracking vehicle – would have been generated if the defect was not found until IV&V testing or after product delivery. The severity of a defect does not determine whether a PTR would be generated, but only the priority of the PTR. An example of a major defect is an incorrect algorithm implementation. A minor defect would not result in a PTR being generated. An example of a minor defect is missing global variables in a file's prolog.

2. Phase Defect Detected – the basic phases of software development were: Requirements Generation, Design, Code and Unit Test, Computer Test, and Post Delivery. When a defect was detected, the phase in which it was found was recorded.

3. Defect Type – each phase of development had a unique classification sheet for defects found during inspections and computer testing. Requirement defects found during any phase were always classified using the same criteria. See Fig. 4 for the Requirements Defect Form utilized for all requirement defects. Similar forms were used to collect defect type information for all software defects detected during each phase of development.

### *4.1. First delivery*

The first delivery was an extensive upgrade to an existing scientific tool developed for use by the US Navy. The tool uses acoustic models and databases, interfaces with several surface ship sensors, and has an extensive GUI. Approximately 33% of the baseline system was impacted by the upgrade.

Software requirements generated for the first delivery did not follow any formal process. A preliminary set of software requirements, approximately 75% complete, was generated and given to the software developers to implement. The user community was not involved in the development of these requirements nor were any formal reviews performed on these requirements. Software developers found and reported requirement defects back to the specification developers during development, who corrected them and distributed change pages. Usually once every two weeks, specification developers and software developers met to discuss discrepancies and clarify processing to be implemented. This process was used during every phase of software development.

REQUIREMENTS DEFECT SUMMARY

ECP ____________ Document ____________
DATE ____________ No. Pages ____________

| TEAM1<br>Name/Role/Prep | TEAM2<br>Name/Role/Prep | TEAM3<br>Name/Role/Prep |
|---|---|---|
| /MOD/ | /MOD/ | /MOD/ |
| /READ/ | /READ/ | /READ/ |
| /AUTH/ | /AUTH/ | /AUTH/ |
| /PEER/ | /PEER/ | /PEER/ |
| /PEER/ | /PEER/ | /PEER/ |

MTG START/END ________ MTG START/END ________ MTG START/END ________
DATE ____________ DATE ____________ DATE ____________

| ABBR CODE | DEFECT TYPE | MAJOR T1 | MAJOR T2 | MAJOR T3 | MAJOR DUPS | MINOR T1 | MINOR T2 | MINOR T3 | MINOR DUPS | TOTALS |
|---|---|---|---|---|---|---|---|---|---|---|
| EQ<br>RE<br>REO | EQUATIONS<br>- Errors<br>- Errors of Omission | | | | | | | | | |
| FG<br>RE<br>REO | FIGURES<br>- Errors<br>- Errors of Omission | | | | | | | | | |
| TB<br>RE<br>REO | TABLES<br>- Errors<br>- Errors of Omission | | | | | | | | | |
| TX<br>AM<br>RE<br>REO | TEXT<br>- Ambiguous<br>- Errors<br>- Errors of Omission | | | | | | | | | |
| | TOTALS | | | | | | | | | |

| TEAM1 | TEAM2 | TEAM3 |
|---|---|---|
| Prep ________ | Prep ________ | Prep ________ |
| Inspection ________ | Inspection ________ | Inspection ________ |
| Inspect Rate ________ | Inspect Rate ________ | Inspect Rate ________ |
| Collation ________ | Collation ________ | Collation ________ |
| Effort/Defect ________ | Effort/Defect ________ | Effort/Defect ________ |

RE-INSPECTION REQUIRED (Y/N) ____________

Figure 4. Requirements Defect Form.

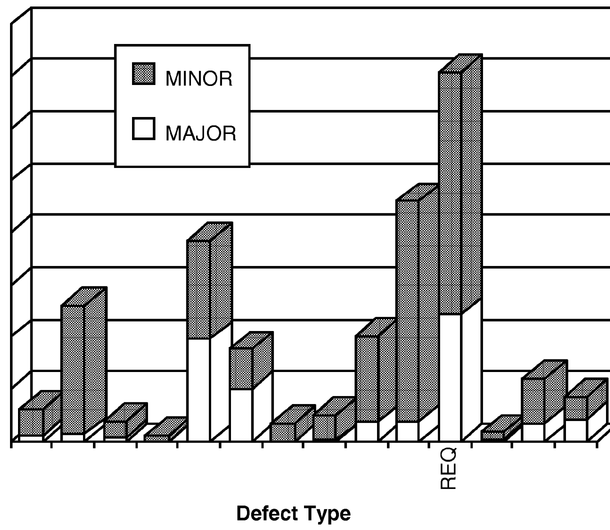


Figure 5. Defect types – delivery one.

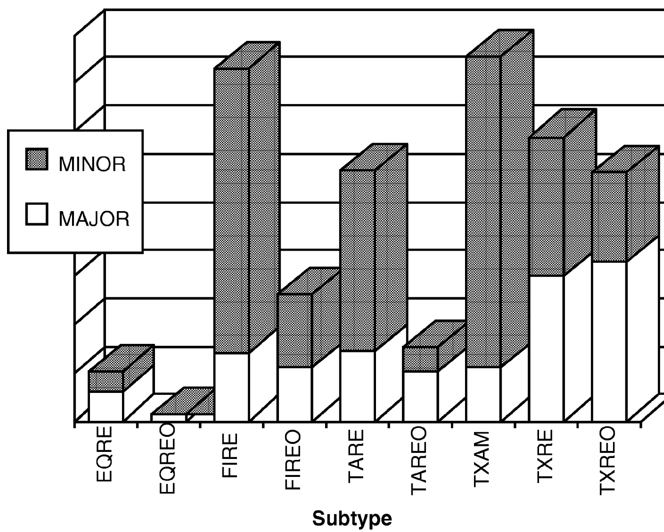


Figure 6. Requirement defects by subtype – delivery one.

This requirements generation process resulted in many requirement change pages being distributed. Each change page package had to be reviewed, and its impact on software and documentation determined. The development process evolved into the waterfall model depicted in Fig. 2. Figure 5 presents the total defects (major and minor) detected during development and testing, sorted by type. Only the most prevalent defect type – REQ – is labeled. The remaining columns in Fig. 5 all pertain to software defect types, such as logic, storage, documentation, conditional/branching, etc. Most of the problems experienced with the first product delivery revolved around

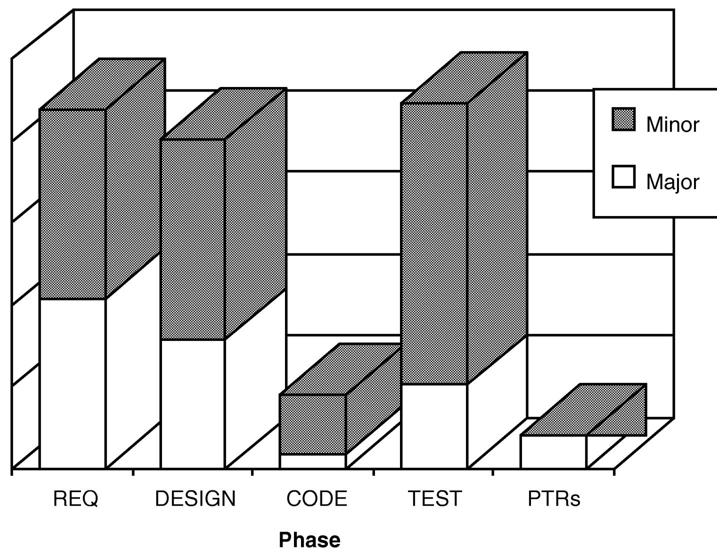


Figure 7. Requirement defects by phase – delivery one.

requirements. To better understand what modification to the requirement generation process was needed, requirement defects were analyzed. Figure 6 presents a breakdown of the total (major and minor) software requirement defects. Requirement defects were categorized into one of four types: Equation Errors (EQ), Table Errors (TB), Figure Errors (FG), or Text Errors (TX). Each type was further broken into one of two subtypes: error of omission (REO) or incorrect specification (RE). An additional subtype – ambiguousness (AM) – was added for text errors to distinguish between incorrect and ambiguous specifications. Thus, an incorrect equation would be categorized as EQRE and a missing figure would be categorized as FIREO. The high percentage of "missing" and "incorrect" textual requirements caused much rework during development. Figure 7 presents a breakdown of requirement defects by development phase, which shows that requirement errors did not drop off after the requirements phase, but instead, continued to be a problem. Detailed analyses of metric data collected during the first delivery indicated that the best part of the process to improve was the requirements generation process. The number and severity of requirement defects detected during computer test phase were mostly GUI related (FGRE, FGREO). When the user community was involved during this phase of development, they identified many deficiencies with the GUI and associated processing, which required rework and specification change pages. The remaining requirement defects (TB, TX, EQ) may have been found if a more "formal" review was performed on the specifications before designing and coding began. The magnitude of requirement defects had a "negative ripple effect" on software development. Software engineers became frustrated and productivity decreased. The two-step methodology was adopted for the second delivery to address and correct these requirement defect problems.

### 4.2. Second delivery

The second delivery also was a major upgrade to the baseline system. It consisted of three enhancements and one PTR cleanup package. One enhancement was similar in functionality to an enhancement added in the first delivery. The other two enhancements were not similar to any functionality previously added or modified. Approximately 40% of the system was impacted by this delivery, which was slightly more than the previous delivery. Each enhancement and PTR cleanup package was treated independently, and was developed using separate, incremental builds. The PTR package, build A, was developed first, followed by the functionality similar to the previous delivery, build B. The other two enhancements, identified as builds C and D, were developed after builds A and B. A schedule of five months was levied on the requirements generation phase.

The two-step methodology was utilized to develop requirement specifications for all four builds. Three different levels of OP-DEMOs were utilized during the requirements generation phase. Build A, the PTR cleanup package, did not add any new screens or flows, but only corrected documentation and implementation problems from the first delivery. Only one existing screen was impacted, and because the change was minor, a "paper" OP-DEMO was utilized. The redlined GUI screen was evaluated by the user community and developers in a meeting. Build C utilized a semi-functional OP-DEMO, with two new algorithms being evaluated on the target hardware to determine the best one to use. End-users evaluated the algorithms, using real data, and selected an algorithm and associatted GUI to implement. OP-DEMOs for builds B and D were developed on non-target hardware, using GUI builder tools. RIs were performed on requirements generated for builds B, C, and D. An "informal" review meeting was conducted on build A requirements. The total number of requirements change pages developed, as well as hours spent developing OP-DEMOs, generating the requirements, and inspecting the requirements, were collected and are summarized in Table 1. Some observations from Table 1 are as follows:

- Requirements for build B were much easier to develop than those for build C because the functionality was similar to enhancements implemented in the first delivery. Consequently, extensive reuse of requirements was utilized. Build D

Table 1
Requirement specification effort – second delivery.

| Build | Total Req Phase Hrs | Total OP-DEMO Hrs | Total RI Hrs | Total Req Gen Hrs | Total Spec pages |
|---|---|---|---|---|---|
| A | 296.0 | 12.0 | 25.0* | 259.0 | 79 |
| B | 801.5 | 312.5 | 109.5 | 379.5 | 293 |
| C | 1474.0 | 633.5 | 109.0 | 732.0 | 152 |
| D | 575.5 | 123.0 | 59.0 | 394.04 | 113 |
| Totals | 3147.0 | 1081.0 | 303.5 | 1764.5 | 637 |

* An "informal" review meeting was conducted on Build A requirements.

requirements took approximately half as long to write as build C because metric data feedback was helping to optimize the specification generation process.

- The level of effort expended on an OP-DEMO directly reflected the amount of system functionality actually modeled. Build C spent more hours than any other build because algorithms were demonstrated as well.
- Total hours spent for these four builds may appear high, but in reality, were considerably less than the hours spent generating requirements for the first delivery. Also, the requirements phase was completed within the five month schedule, and when the requirements phase was complete, the requirement specifications were complete, as can be seen from Fig. 8. Figure 8 compares when requirement defects were found during development for the two deliveries. The two curves are normalized to the number of "original" requirement specification pages. The curves in Fig. 8 indicate that requirements for the second delivery were stable, quality specifications. This is further supported by the fact that less than 50 requirement change pages, correcting mostly minor defects, were generated for the second delivery.
- Figure 8 also illustrates the impact of combining the two requirement defect detection techniques. An OP-DEMO is a visual defect detection tool and thus, should help to reduce visual requirement defects. This assumption is reflected in Fig. 8. A comparison of requirement defects found during the Test phase (computer based testing) demonstrates a significant decrease in requirement defects between the two deliveries. RI on the other hand, is a formal defect detection process for "readable" products and thus, should help to reduce documentation requirement defects. This assumption also is reflected in Fig. 8. A comparison of requirement defects found during the Design and Code phases also demonstrate a significant decrease in requirement defects between the two deliveries.

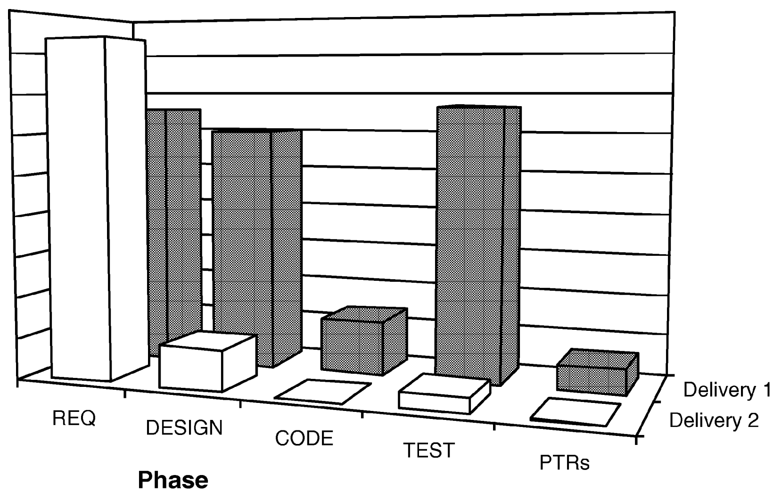


Figure 8. Total requirement defects by phase.

Table 2
RI defect detection efficiency and cost

| Build | Total Req defects found in Req Phase | Total Req defects found after Req Phase | RI defect detection efficiency | Defects/Hr |
|---|---|---|---|---|
| A* | 3 | 20 | 13% | 0.12 |
| B | 188 | 44 | 81% | 1.72 |
| C | 148 | 23 | 87% | 1.35 |
| D | 78 | 0 | 100% | 1.32 |

* An “informal” review meeting was conducted on Build A requirements.

Table 3
Comparison of RI defect detection efficiency

| Application type | Inspections used | % of total defects found |
|---|---|---|
| Business application [Fagan 1976] | Design and code | 82 |
| Flight control [Kolkhorst and Macina 1988] | Design and code | 85 |
| Telecommunications [Russell 1991] | Code | 80 |
| Build A* – scientific | Review meeting | 13 |
| Build B – scientific | 2-fold requirement | 81 |
| Build C – scientific | 3-fold requirement | 87 |
| Build D – scientific | 2-fold requirement | 100 |

* An “informal” review meeting was conducted on Build A requirements.

Table 4
$N$-fold RI

| Build | Number of RI teams | % of duplicate Req defects found |
|---|---|---|
| B | 2 | 14 |
| C | 3 | 21 |
| D | 2 | 40 |

The combined OP-DEMO and RI defect detection methodology found almost all of the requirement defects during the Requirement phase instead of during later phases.

Tables 2, 3, and 4 present additional RI metric data collected during the second delivery. Some important observations can be made:

- RIs are considerably more efficient – >80% versus 13% – than “informal” reviews. This observation agrees with information found in literature [Fagan 1976; Bisant and Lyle 1989].
- The efficiency of RIs detecting requirement errors is comparable with Design and Code Inspections detecting software errors, as shown in Table 3.
- The efficiency of RIs improved as the teams gained more experience. Not only did the quality of RIs increase, but the quality of specifications as well.
- RI is a cost-effective defect detection technique, provided multiple RI teams are utilized. The relatively low percentage of duplicate errors found by multiple

teams indicates that many requirement errors would have been missed during the requirements generation phase if $N$-fold RIs were not conducted. Real-time feedback of metric data during RIs enabled reviewers to identify trends in specifications, which accounted for the increasing percentages of duplicate defect detections. A 40% duplicate defect detection value is not high enough to justify using a single RI team because more than half of the requirement defects would be missed.

## 5. Conclusions

The impact of requirement defects on software development is often underestimated. Results from the first delivery indicate that requirement defects can lead to significant rework, resulting in schedule slippage and increased development costs. Metric data collected during the first delivery's development identified where process improvement was required. Combining OP-DEMOs and RIs proved to be an effective requirement defect reduction methodology. Metric data collected during the second delivery's development support continued use of the two-step methodology.

Based on the experiences of instituting the two-step methodology for the second delivery and the corresponding metric data obtained, the following lessons were learned:

- OP-DEMOs eliminate defects that would normally be found during computer based testing, which reduces development costs.
- A working GUI model for the entire system can better model user requirements rather than only modeling unclear or uncertain requirements. Interaction between understood and unclear processing is better evaluated if both are demonstrated. GUI builder tools can inexpensively extend the scope of OP-DEMOs such that the complete GUI can be evaluated.
- RIs eliminate defects that would normally be found during later phases of development, which further reduces development costs. At least two RI teams should be utilized to obtain an acceptable requirement defect detection efficiency of approximately 80%.
- The combined OP-DEMO and RI two-step methodology had a "positive ripple effect" on software development. The software schedule for the second delivery was met and software was developed within budget. The cost of requirements generation was significantly reduced and the quality of requirement specifications was significantly increased.
- Not all requirement defects were found with the two-step methodology, but the probability of requirements defects causing schedule slippage or significant rework was greatly reduced, and the probability of end-user acceptance of the product was greatly increased.

- The two-step methodology promoted a team oriented approach to software development. A team approach encouraged communication and cooperation among groups responsible for developing various products, which was invaluable when software defects were uncovered that required redesigns or rework.
- The two-step methodology also promoted a user-oriented product development process, which resulted in a more operable ("user-friendly") product being developed.
- Process improvement is very important if development costs are to be reduced. Metric data collection for both deliveries made it possible to identify a process area to improve, and then to evaluate the impact of a process change.

## References


Beohm, B. (1987), "Industrial software metrics top-10 list," *IEEE Software 3*, 9, 84–85.

Bisant, B. and J. Lyle (1989), "A Two-Person Inspection Method to Improve Programming Productivity," *IEEE Transactions on Software Engineering 15*, 10, 1294–1304.

Davis, A. (1992), "Operational Prototyping: A New Development Approach," *IEEE Software 9*, 5, 70–78.

Fagan, M. (1976), "Design and Code Inspections to Reduce Errors in Program Development," *IBM Systems Journal 15*, 3, 182–211.

Fagan, M. (1986), "Advances in Software Inspections," *IEEE Transactions on Software Engineering SE-12*, 7, 744–751.

Hekmatpour, S. (1987), "Experience with evolutionary prototyping in a large software system," *ACM SIGSOFT Software Engineering Notes 12*, 38–41.

Kolkhorst, B. and A. Macina (1988), "Developing Error-Free Software," *IEEE AES Magazine*, 25–31.

Martin, J. and W. Tsai (1990), "N-Fold Inspection: A Requirements Analysis Technique," *Communications of the ACM 33*, 2, 225–232.

Pleeger, S. (1992), "Measuring software reliability," *IEEE Spectrum*, 56–60.

Royce, W. (1987), "Managing the development of large software systems: concepts and techniques," In *Proc. 9th Conf. Software Engineering*, pp. 328–338.

Russell, G. (1991), "Experience with Inspection in Ultralarge-Scale Developments," *IEEE Software 8*, 1, 25–31.