Naveen Prakash
Deepika Prakash

Data Warehouse Requirements Engineering

A Decision Based Approach

Springer

Data Warehouse Requirements Engineering

Naveen Prakash · Deepika Prakash

Data Warehouse Requirements Engineering

A Decision Based Approach

Springer

Naveen Prakash
ICLC Ltd.
New Delhi
India

Deepika Prakash
Central University of Rajasthan
Kishangarh
India

ISBN 978-981-10-7018-1 ISBN 978-981-10-7019-8 (eBook)
https://doi.org/10.1007/978-981-10-7019-8

Library of Congress Control Number: 2017961755

Printed on acid-free paper

This Springer imprint is published by the registered company Springer Nature Singapore Pte Ltd. part of Springer Nature.
The registered company address is: 152 Beach Road, #21-01/04 Gateway East, Singapore 189721, Singapore

To
Our Family

Preface

That requirements engineering is part of the systems development life cycle and is about the first activity to be carried out when building systems is today considered as basic knowledge in computer science/information technology. Requirements engineering produces requirements specifications that are carried through to system design and implementation. It is assumed that systems automate specific activities that are carried out in the real world. These activities are transactions, for example reservations, cancellations, buying, selling, and the like. Thus requirements engineering produces requirements specifications of transactional systems.

So long as systems were not very complex, the preparation of a requirements specification was feasible and did not compromise on system delivery times. However, as systems became more and more complex, iterative and incremental development came to the fore. Producing a requirements specification is now frowned upon and we need to produce, in the language of Scrum, user stories for small parts of the system.

About the time requirements engineering was developing, data warehousing also became important. Data warehouse development faced the same challenges as transactional systems do, namely determination of the requirements to be met and the role of requirements engineering in the era of agile development. However, both these issues have been taken up relatively recently.

Due to this recent interest in the area, requirements engineering for data warehousing is relatively unknown. We fear that there is widespread paucity of understanding of the nature of data warehouse requirements engineering, how it differs from traditional transaction-oriented requirements engineering and what are the new issues that it raises.

Perhaps, the role of agility in data warehouse development is even more crucial than in transactional systems development. This is because of the inherent complexity of data warehouse systems, long lead times to delivery, and the huge costs involved in their development. Indeed, the notion of data marts and the bus approach to data warehouse development is an early response to these challenges.

This book is our attempt at providing exposure to the problem of data warehouse requirements engineering. We hope that the book shall contribute to a wider

awareness of the difference between requirements engineering for transactional and data warehouse systems, and of the challenges that data warehousing presents to the requirements engineering community.

The position adopted in this book is that even in the face of agile development, requirements engineering continues to be relevant. Requirements engineering today is not to produce requirements specifications of entire systems. Rather, it is done to support incremental and iterative development. In other words, rather than restrict incremental and iterative development to downstream tasks of design and implementation, we must extend it to the requirements engineering task as well. We argue that the entire data warehouse systems development life cycle should become agile.

Thus, we make requirements and requirements engineering as the fulcrum for data warehouse agile development. Just as requirements specifications *of systems* formed the basis for proceeding with systems development earlier, so also now requirements specifications *of system increments* must form the basis of incremental and iterative development.

Following this line of argument, instead of a requirements specification, we propose to develop requirements granules. It is possible to consider building a requirements granule per data mart. However, we consider a data mart as having very large granularity because it addresses an entire subject like sales, purchase etc. Therefore, the requirements granule that will be produced shall be large-grained resulting in relatively long lead times to delivery of the intended product increment. It is worth developing an approach to requirements engineering that can produce requirements granules of smaller sizes.

To reduce the sizes of requirements granules, we introduce the notion of a decision and propose to build data warehouse fragments for decisions. Thus, data warehouse requirements engineering is for discovering the decisions of interest and then determining the information relevant to each decision. A requirements granule is the collection of information relevant to a decision. If this information is available then it is possible for the decision maker to obtain it from the data warehouse fragment, evaluate it, and decide whether to take the decision or not. This implies that the size of a granule is determined by the amount of information that is associated with a decision.

The notion of a decision is thus central to our approach. A decision represents the useful work that the data warehouse fragment supports and a data warehouse fragment is the implementation of a requirements fragment. The approach in this book represents a departure from the conventional notion of a data mart that is built to "analyze" a subject area. Analysis for us is not an aim in itself but taking a decision is and analysis is only in support of the decision making task.

As more and more decisions are taken up for development, there is a proliferation of requirements granules and data warehouse fragments. This results in problems of inconsistent information across the enterprise, and of proliferating costs due to multiple platforms and ETL processes. This is similar to what happens in the bus-of-data-marts approach except that a decision may be of a lower granularity than a data mart. This means that we can expect many more data warehouse

fragments than data marts and the problem of inconsistency and costs is even more severe.

Given the severity of the problem, we do not consider it advisable to wait for the problem to appear and then take corrective action by doing consolidation. It is best to take a preventive approach that minimizes fragment proliferation. Again, keeping in mind that for us requirements are the fulcrum for data warehouse development, we consolidate requirements granules even as they are defined.

This book is a summary of research in the area of data warehouse requirements engineering carried out by the authors. To be sure, this research is ongoing and we expect to produce some more interesting results in the future. However, we believe that we have reached a point where the results we have achieved form a coherent whole from which the research and industrial community can benefit.

The initial three chapters of the book form the backdrop for the last three. We devote Chap. 1 to the state of the art in transactional requirement engineering whereas Chap. 2 is for data warehouse requirements engineering. The salient issues in data warehouse requirements engineering addressed in this book are presented in Chap. 3.

Chapter 4 deals with the different types of decisions and contains techniques for their elicitation. Chapter 5 is devoted to information elicitation for decisions and the basic notion of a requirements granule is formulated here. Chapter 6 deals with agility built around the idea of the requirements granules and data warehouse fragments. The approach to data warehouse consolidation is explained here.

The book can be used in two ways. For those readers interested in a broad-brush understanding of the differences between transactional and data warehouse requirements engineering, the first three chapters would suffice. However, for those interested in deeper knowledge, the rest of the chapters would be of relevance as well.

New Delhi, India — Naveen Prakash
Kishangarh, India — Deepika Prakash
September 2017

Contents

About the Authors

Naveen Prakash started his career with the Computer Group of Bhabha Atomic Research Centre Mumbai in 1972. He obtained his doctoral degree from the Indian Institute of Technology Delhi (IIT Delhi) in 1980. He subsequently worked at the National Center for Software Development and Computing Techniques, Tata Institute of Fundamental Research (NCSDCT, TIFR) before joining the R&D group of CMC Ltd where he worked for over 10 years doing industrial R&D. In 1989, he moved to academics. He worked at the Department of Computer Science and Engineering, Indian Institute of Technology Kanpur (IIT Kanpur), and at the Delhi Institute of Technology (DIT) (now Netaji Subhas Institute of Technology (NSIT)), Delhi. During this period he provided consultancy services to Asian Development Bank and African Development Bank projects in Sri Lanka and Tanzania, respectively, as well as to the Indira Gandhi National Centre for the Arts (IGNCA) as a United Nations Development Programme (UNDP) consultant. He served as a scientific advisor to the British Council Division, New Delhi and took up the directorship of various educational institutes in India. Post-retirement, he worked on a World Bank project in Malawi.

Prof. Prakash has lectured extensively in various universities abroad. He is on the editorial board of the Requirements Engineering Journal, and of the International Journal of Information System Modeling and Design (IJISMD). He has published over 70 research papers and authored two books.

Prof. Prakash continues to be an active researcher. Besides Business Intelligence and Data Warehousing, his interests include the Internet-of-things and NoSQL database. He also lectures at the Indira Gandhi Delhi Technical University for Women (IGDTUW), Delhi and IIIT Delhi.

Deepika Prakash obtained her Ph.D. from Delhi Technological University, Delhi in the area of Data Warehouse Requirements Engineering. Currently, she is an Assistant Professor at the Department of Big Data Analytics, Central University of Rajasthan, Rajasthan.

Dr. Prakash has five years of teaching experience, as well as two years of experience in industrial R&D, building data marts for purchase, sales and inventory and in data mart integration. Her responsibilities in industry spanned the complete life cycle, from requirements engineering through conceptual modeling to extract-transform-load (ETL) activities.

As a researcher, she has authored a number of papers in international forums and has delivered invited lectures at a number of Institutes throughout India. Her current research interests include Business Intelligence, Health Analytics, and the Internet-of-Things.

Chapter 1
Requirements Engineering for Transactional Systems

Transactional systems have been the forte of Information Systems/Software Engineering. These systems deal with automating the functionality of systems, to provide value to the users. Initially, up to the end of the decade of the 1960s, transactional systems were simple, single-function systems. Thus, we had payroll systems that accounts people would use to compute the salary of employees and print out salary. Information Systems/Software Engineering technology graduated to multi-functional systems that looked at the computerization of relatively larger chunks of the business. Thus, it now became possible to deal with the accounts department, the human resource department, customer interface, etc. Technology to deal with such systems stabilized in the period 1960–1980. Subsequently, attention shifted to even more complex systems, the computerization of the entire enterprise, and to inter-organization information systems.

The demand for engineering of ever more complex systems led to the "software crisis", a term widely used in the 1990s to describe the difficulties that industry of that time faced. A number of studies were carried out and some of the problems highlighted were systems failure/rejection by clients, inability to deliver complex and large software well.

The Standish Group's "Chaos" reports [1] presented software industry's record in delivering large-sized systems using traditional development methods. The Group conducted a survey of 8380 projects carried out by 365 major American companies. The results showed that projects worth up to even $750,000 exceeded budget and time. Further, they failed to deliver the promised features more than 55% of the time. As the size of the applications grew, the success rate fell to 25% for efforts over $3 million and down to zero for projects over $10 million.

Bell labs and IBM [2] found that 80% of all defects in software products lie in the requirements phase. Boehm and Papaccio [3] said that correcting requirements errors is 5 times more expensive when carried out during the design phase; the cost of correction is 10 times during implementation phase; the cost rises to 20 times for corrections done during testing and it becomes an astronomical 200 times after the system has been delivered. Evidently, such corrections result in expensive products

N. Prakash and D. Prakash, *Data Warehouse Requirements Engineering*,
https://doi.org/10.1007/978-981-10-7019-8_1

and/or total rejection of software. The Standish group [4] reported that one of the reasons for project failure is "incomplete requirements". Clearly, the effect of poorly engineered requirements ranges from outright systems rejection by the customer to major reworking of the developed system.

The *Software Hall of Shame* [5] surveyed around 30 large software development projects that failed between 1992 and 2005 to try to identify the causes of this failure. It was found that failures arise because either projects go beyond actual needs or because of expansion in the scope of the original project. This implied that requirements changed over the course of product development and this change was difficult to handle.

The foregoing suggested that new methods of software development were needed that delivered on time, on budget, met their requirements, and were also capable of handling changing requirements. The response was twofold:

- An emphasis on incremental and iterative product development rather than one-shot development of the entire product. Small, carefully selected product parts were developed and integrated with other parts as and when these latter became available. As we shall see this took the form of agile software development.
- The birth of the discipline of requirements engineering in which the earlier informal methods were replaced by model-driven methods. This led to the systematization of the requirements engineering process, computer-based management of requirements, guidance in the requirements engineering task, and so on.

We discuss these two responses in the rest of this chapter.

1.1 Transactional System Development Life Cycle

The System Development Life cycle, SDLC, for transactional systems (TSDLC) starts from gathering system/software requirements and ends with the deployment of the system. One of the earliest models of TSDLC is the waterfall model. The waterfall model has six sequential phases. Each phase has different actors participating in it. Output of one phase forms the input to the next phase. This output is documented and used by the actors of the next phase. The size of documentation produced is very large and time-consuming.

Since the model is heavy on documentation, the model is sometimes referred to as document driven. Table 1.1 shows the actors and document produced against each phase of the life cycle.

The process starts with identifying what needs to be built. There are usually several stakeholders of a system. Each stakeholder sits down with the requirements engineer and details what s/he specifically expects from the system. These needs are referred to as requirements. A more formal definition of the term requirements is available in the subsequent sections of this chapter. These requirements as given

Table 1.1 The different phases of TSDLC

TSDLC phase	Actor	Document
Requirements engineering	Stakeholder, Requirements engineer	System requirements specification
System and software design	System analysts	High-level and low-level design documents
Implementation	Development team	Code
Verification	Tester	Test case document
Maintenance	Project manager, Stakeholder	User manuals

by the stakeholder are documented as a System Requirements Specification (SRS) document.

Once the SRS is produced, the actors of the system design phase, system analyst, convert the requirements into high-level design and low-level design. The former describes the software architecture. The latter discusses the data structure to be used, the interfaces and other procedural details. Here, two documents are produced, the high-level and low-level design document.

The design documents are made available to the implementation team for the development activity to start. Apart from development, unit testing is also a feature of this phase. In the Verification phase, functional and non-functional testing is performed and a detailed test case document is produced. Often test cases are designed with involvement of the stakeholders. Thus, apart from the testers, stakeholders are also actors of this phase. Finally, the software is deployed and support is provided for maintenance of the product.

Notice, each phase is explored fully before moving on to the next phase. Also notice, there is no feedback path to go to a previous already completed phase. Consider the following scenario. The product is in the implementation phase and the developers realize that an artifact has been poorly conceptualized. In other words, there is a need to rework a part of the conceptual model for development to proceed. However, there is no provision in the model to go back to the system design phase once the product is in the development phase.

This model also implies that

(a) Requirements once specified do not change. However, this is rarely the case. A feedback path is required in the event of changing requirements. This ensures that changes are incorporated in the current software release rather than waiting for the next release to adopt the changed requirement.
(b) "All" requirements can be elicited from the stakeholders. The requirements engineering phase ends with a sign-off from the stakeholder. However, as already brought out, studies have shown that it is not possible to elicit all the requirements upfront from the stakeholders. Stakeholders are often unable to envision changes that could arise 12–24 months down the line and generally mention requirements as of the day of the interview with the requirements engineer.

Being sequential in nature, a working model of the product is released only at the end of the life cycle. This leads to two problems. One that feedback can be got from the stakeholder only after the entire product is developed and delivered. Even a slightly negative feedback means that the entire system has to be redeveloped; considerable time and effort in delivering the product is wasted.

The second problem is that these systems suffer from long lead time for product delivery. This is because the entire requirements specification is made before the system is taken up for design and implementation.

An alternate method to system development is to adopt an agile development model. The aim of this model is to provide an **iterative** and **incremental** development framework for delivery of a product. An iteration is defined by clear deliverables which are identified by the stakeholder. Deliverables are pieces of the product *usable* by the stakeholder. Several iterations are performed to deliver the final product making the development process incremental. Also, iterations are time boxed with time allocated to each iteration remaining almost the same till the final product is delivered.

One of the popular approaches to agile development is Scrum. In Scrum, iterations are referred to as sprints. There are two actors, product owner and developer. The product owner is the stakeholder of the waterfall model. The requirements are elicited in the form of user stories. A user story is defined as a single sentence that identifies a need. User stories have three parts, "Who" identifies the stakeholder, "What" identifies the action, and "Why" identifies the reason behind the action. A good user story is one that is actionable, meaning that the developer is able to use it to deliver the need at the end of the sprint.

Wake [6] introduced the INVEST test as a measure of how good a user story is. A good user story must meet the following criteria: Independent, Not too specific, Valuable, Estimable, Small, and Testable. One major issue in building stories is that of determining when the story is "small". Small is defined as that piece of work that can be delivered in a sprint. User stories as elicited from the product owner may not fit in a sprint. Scrum uses the epic–theme–user story decomposition approach to deal with this. Epics are stories identified by the product owner in the first conversation. They require several sprints to deliver. In order to decompose the epic, further interaction with the product owner is performed to yield themes. However, a theme by itself may take several sprints, but a lesser number than for its epic, to deliver. Therefore, a theme is further decomposed into user stories of the right size.

When comparing agile development model with the waterfall model, there are two major differences as follows:

1. In Scrum, sprints do not wait for the full requirements specification to be produced. Further, the requirements behind a user story are also not fully specified but follow the 80–20 principle. 80% of the requirements need to be clarified before proceeding with a sprint and the balance 20% are discovered during the sprint. Thus, while in waterfall model, stakeholder involvement in the requirements engineering phase ends with a sign-off from the stakeholder, in Scrum the stakeholder is involved during the entire life cycle. In fact, iterations proceed with the feedback of the stakeholder.

2. At the end of one iteration, a working sub-product is delivered to the stakeholder. This could either be an enhancement or a new artifact. This is unlike the waterfall model where the entire product is delivered at the end of the life cycle.

1.2 Transactional Requirements Engineering

Let us start with some basic definitions that tell us what requirements are and what requirements engineering does.

Requirements

A requirement has been defined in a number of ways. Some definitions are as follows.

Definition 1: A requirement as defined in [7] is "*(1) a condition or capability needed by a user to solve a problem or achieve an objective, (2) A condition or capability that must be met or possessed by a system or system components to satisfy a contract, standard, specification or other formally imposed documents, (3) A document representation of a condition as in (1) or in (2)*".

According to this definition, requirements arise from user, general organization, standards, government bodies, etc. These requirements are then documented.

A requirement is considered as a specific property of a product by Robertson, and Kotonya as shown in Definition 2 and Definition 3 below.

Definition 2: "Something that the product must do or a quality that the product must have" [8].

Definition 3: "A description of how the system shall behave, and information about the application domain, constraints on operations, a system property etc." [9].

Definition 4: "Requirements are high level abstractions of the services the system shall provide and the constraints imposed on the system".

Requirements have been classified as functional requirements, FR, and non-functional requirements, NFR. Functional requirements are "statements about what a system should do, how it should behave, what it should contain, or what components it should have" and non-functional requirements are "statements of quality, performance and environment issues with which the system should conform" [10]. Non-functional requirements are global qualities of a software system, such as flexibility, maintainability, etc. [11].

Requirements Engineering

Requirements engineering, RE, is the process of obtaining and modeling requirements. Indeed, a number of definitions of RE exist in literature.

Definition 1: Requirements engineering (RE) is defined [7] as "*the systemic process of developing requirements through an iterative cooperative process of analyzing*

the problem, documenting the resulting observations in a variety of representation formats and checking the accuracy of understanding gained".

The process is cooperative because different stakeholders have different needs and therefore varying viewpoints. RE must take into account conflicting views and interests of users and stakeholders. Capturing different viewpoints allows conflicts to surface at an early stage in the requirements process. Further, the resulting requirements are the ones that are agreeable to both customers and developers.

Definition 2 Zave [12]: Requirements engineering deals with the real-world goals for functions and constraints of the software system. It makes a precise specification of software behavior and its evolution over time.

This definition incorporates "real-world goals" in its definition. In other words, this definition hopes to capture requirements that answer the "why" of software systems. Here, the author is referring to "functional requirements". Further, the definition also gives emphasis to "precise requirements". Thus quality of requirements captured is also important.

Definition 3 van Lamsweerde [13]: RE deals with the identification of goals to be achieved by the system to be developed, the operationalization of such goals into services and constraints.

Definition 4 Nuseibeh and Easterbrook [14]: RE aims to discover the purpose behind the system to be built, by identifying stakeholders and their needs, and their documentation.

Here, the emphasis is on *identifying stakeholders* and capturing the requirements of the stakeholders.

1.3 Requirements Engineering (RE) as a Process

Evidently, requirements engineering can be viewed as a process with an input and an output. Stakeholders are the problem owners. They can be users, designers, system analysts, business analysts, technical authors, and customers. In the RE process, requirements are elicited from these sources. Output of the process is generally a set of agreed requirements, system specifications, and system models. The first two of these are in the form of use cases, goals, agents, or NFRs. System models can be object models, goal models, domain descriptions, behavioral models, problem frames, etc.

There are three fundamental concerns of RE, namely, understanding the problem, describing the problem, and attaining an agreement on the nature of the problem. The process involves several actors for the various activities. We visualize the entire process as shown in Fig. 1.1. There are four stages each with specific actors, marked with green in the figure. A requirements engineer is central in the entire process.

Let us elaborate the components of the figure.