SAS®
DATA ANALYTIC DEVELOPMENT
Dimensions of Software Quality

TROY MARTIN HUGHES
Contents

Preface xi

Acknowledgments xvi

About the Author xvii

Chapter 1 Introduction 1
   Distinguishing Data Analytic Development 3
   Software Development Life Cycle (SDLC) 7
   Risk 14

Chapter 2 Quality 21
   Defining Quality 24
   Software Product Quality Model 30
   Quality in the SDLC 40

Chapter 3 Communication 49
   Return Codes 51
   System Numeric Return Codes 53
   System Alphanumeric Return Codes 70
   User-Generated Return Codes 74
   Parallel Processing Communication 79

PART I  STATIC PERFORMANCE 85

Chapter 4 Reliability 87
   Defining Reliability 90
   Paths to Failure 91
   ACL: The Reliability Triad 102
   Reliability in the SDLC 108

Chapter 5 Recoverability 123
   Defining Recoverability 125
   Recoverability toward Reliability 127
   Recoverability Matrix 131
   TEACH Recoverability Principles 132
   SPICIER Recoverability Steps 136
   Recovering with Checkpoints 148
   Recoverability in the SDLC 151
Chapter 6  Robustness  159
  Defining Robustness  162
  Robustness toward Reliability  163
  Defensive Programming  164
  Exception Handling  172
  Robustness in the SDLC  203

Chapter 7  Execution Efficiency  207
  Defining Execution Efficiency  209
  Factors Affecting Execution Efficiency  210
  False Dependencies  211
  Parallel Processing  220
  Execution Efficiency in the SDLC  232

Chapter 8  Efficiency  243
  Defining Efficiency  246
  Disambiguating Efficiency  246
  Defining Resources  249
  Efficiency in the SDLC  259

Chapter 9  Scalability  273
  Defining Scalability  276
  The Scalability Triad  276
  Resource Scalability  278
  Demand Scalability  279
  Load Scalability  290
  Scalability in the SDLC  309

Chapter 10  Portability  313
  Defining Portability  316
  Disambiguating Portability  317
  3GL versus 4GL Portability  318
  Facets of Portability  319
  Portability in the SDLC  338

Chapter 11  Security  341
  Defining Security  344
  Confidentiality  344
  Integrity  345
  Availability  365
  Security in the SDLC  379

Chapter 12  Automation  383
  Defining Automation  386
  Automation in SAS Software  387
  SAS Processing Modes  388
CHAPTER 1

Introduction
DATA ANALYTIC DEVELOPMENT

Software development in which ultimate business value is delivered not through software products but rather through subsequent, derivative data products, including data sets, databases, analyses, reports, and data-driven decisions.

Data analytic development creates and implements software as a means to an end, but the software itself is never the end. Rather, the software is designed to automate data ingestion, cleaning, transformation, analysis, presentation, and other data-centric processes. Through the results generated, subsequent data products confer information and ultimately knowledge to stakeholders. Thus, a software product in and of itself may deliver no ultimate business value, although it is necessary to produce the golden egg—the valuable data product. As a data analytic development language, Base SAS is utilized to develop SAS software products (programs created by SAS practitioners) that are compiled and run on the SAS application (SAS editor and compiler) across various SAS interfaces (e.g., SAS Display Manager, SAS Enterprise Guide, SAS University Edition) purchased from or provided by SAS, Inc.

Data analytic software is often produced in a development environment known as end-user development, in which the developers of software themselves are also the users. Within end-user development environments, software is never transferred or sold to a third party but is used and maintained by the developer or development team. For example, a financial fraud analyst may be required to produce a weekly report that details suspicious credit card transactions to validate and calibrate fraud detection algorithms. The analyst is required to develop a repeatable SAS program that can generate results to meet customer needs. However, the analyst is an end-user developer because he is responsible for both writing the software and creating weekly reports based on the data output. Note that this example represents data analytic development within an end-user development environment.

Traditional, user-focused, or software applications development contrasts sharply with data analytic development because ultimate business value is conferred through the production and delivery of software itself. For example, when Microsoft developers build a product such as Microsoft Word or Excel, the working software product denotes business value because it is distributed to and purchased by third-party users. The software development life cycle (SDLC) continues after purchase, but only insofar as maintenance activities
are performed by Microsoft, such as developing and disseminating software patches. In the following section, data analytic development is compared to and contrasted with end-user and traditional development environments.

DISTINGUISHING DATA ANALYTIC DEVELOPMENT

So why bother distinguishing data analytic development environments? Because it’s important to understand the strengths and weaknesses of respective development environments and because the software development environment can influence the relative quality and performance of software.

To be clear, data analytic development environments, end-user development environments, and traditional software development environments are not mutually exclusive. Figure 1.1 demonstrates the entanglement between these environments, demonstrating the majority of data analytic development performed within end-user development environments.

The data-analytic-end-user hybrid represents the most common type of data analytic development for several reasons. Principally, data analytic software is created not from a need for software itself but rather from a
need to solve some problem, produce some output, or make some decision. For example, the financial analyst who needs to write a report about fraud levels and fraud detection accuracy in turn authors SAS software to automate and standardize this solution. SAS practitioners building data analytic software are often required to have extensive domain expertise in the data they’re processing, analyzing, or otherwise utilizing to ensure they produce valid data products and decisions. Thus, first and foremost, the SAS practitioner is a financial analyst, although primary responsibilities can include software development, testing, operation, and maintenance.

Technical aspects and limitations of Base SAS software also encourage data analytic development to occur within end-user development environments. Because Base SAS software is compiled at execution, it remains as plain text not only in development and testing phases but also in production. This prevents the stabilization or hardening of software required for software encryption, which is necessary when software is developed for third-party users. For this reason, no market exists for companies that build and sell SAS software to third-party user bases because the underlying code would be able to be freely examined, replicated, and distributed. Moreover, without encryption, the accessibility of Base SAS code encourages SAS practitioners to explore and meddle with code, compromising its security and integrity.

The data-analytic-traditional hybrid is less common in SAS software but describes data analytic software in which the software does denote ultimate business value rather than a means to an end. This model is more common in environments in which separate development teams exist apart from analytic or software operational teams. For example, a team of SAS developers might write extract-transform-load (ETL) or analytic software that is provided to a separate team of analysts or other professionals who utilize the software and its resultant data products. The development team might maintain operations and maintenance (O&M) administrative activities for the software, including training, maintenance, and planning software end-of-life, but otherwise the team would not use or interact with the software it developed.

When data-analytic-traditional environments do exist, they typically produce software only for teams internal to their organization. Service level agreements (SLAs) sometimes exist between the development team and the team(s) they support, but the SAS software developed is typically neither sold nor purchased. Because SAS code is plain text and open to inspection, it’s uncommon for a SAS development team to sell software beyond its
INTRODUCTION

organization. SAS consultants, rather, often operate within this niche, providing targeted developmental support to organizations.

The third and final hybrid environment, the end-user-traditional model, demonstrates software developed by and for SAS practitioners that is not data-focused. Rather than processing or analyzing variable data, the SAS software might operate as a stand-alone application, driven by user inputs. For example, if a rogue SAS practitioner spent a couple weeks of work encoding and bringing to life Parker Brothers’ legendary board game Monopoly in Base SAS, the software itself would be the ultimate product. Of course, whether or not the analyst was able to retain his job thereafter would depend on whether his management perceived any business value in the venture!

Because of the tendency of data analytic development to occur within end-user development environments, traditional development is not discussed further in this text except as a comparison where strengths and weaknesses exist between traditional and other development environments. The pros and cons of end-user development are discussed in the next section.

End-User Development

Many end-user developers may not even consider themselves to be software developers. I first learned SAS working in a Veterans Administration (VA) psychiatric ward, and my teachers were psychologists, psychiatrists, statisticians, and other researchers. We saw patients, recorded and entered data, wrote and maintained our own software, analyzed clinical trials data, and conducted research and published papers on a variety of psychiatric topics. We didn’t have a single “programmer” on our staff, although more than half of my coworkers were engaged in some form of data analysis in varying degrees of code complexity. However, because we were clinicians first and researchers second, the idea that we were also software developers would have seemed completely foreign to many of the staff.

In fact, this identity crisis is exactly why I use “SAS practitioners” to represent the breadth of professionals who develop software in the Base SAS language—because so many of us may feel that we are only moonlighting as software developers, despite the vast quantity of SAS software we may produce. This text represents a step toward acceptance of our roles—however great or small—as software developers.

The principal advantage of end-user development is the ability for domain experts—those who understand both the ultimate project intent and its
data—to design software. The psychiatrists didn’t need a go-between to help convey technical concepts to them, because they themselves were building the software. Neither was a business analyst required to convey the ultimate business need and intent of the software to the developers, because the developers were the psychiatrists—the domain experts. Because end-user developers possess both domain knowledge and technical savvy, they are poised to rapidly implement technical solutions that fulfill business needs without business analysts or other brokers.

To contrast, traditional software development environments often demonstrate a domain knowledge divide in which high-level project intent and requirements must be translated to software developers (who lack domain expertise) and where some technical aspects of the software must be translated to customers (who lack technical expertise in computer science or software development). Over time, stakeholders will tend to broaden their respective job roles and knowledge but, if left unmitigated, the domain knowledge divide can lead to communication breakdown, misinterpretation of software intent or requirements, and less functional or lower quality software. In these environments, business analysts and other brokers play a critical role in ensuring a smooth communication continuum among domain knowledge, project needs and objectives, and technical requirements.

Traditional software development environments do outperform end-user development in some aspects. Because developers in traditional environments operate principally as software developers, they’re more likely to be educated and trained in software engineering, computer science, systems engineering, or other technically relevant fields. They may not have domain-specific certifications or accreditations to upkeep (like clinicians or psychiatrists) so they can more easily seek out training and education opportunities specific to software development. For example, in my work in the VA hospital, when we received training, it was related to patient care, psychiatry, privacy regulations, or some other medically focused discipline. We read and authored journal articles and other publications on psychiatric topics—but never software development.

Because of this greater focus on domain-specific education, training, and knowledge, end-user developers are less likely to implement (and, in some cases, may be unaware of) established best practices in software development such as reliance on the SDLC, Agile development methodologies, and performance requirements such as those described in the International Organization for Standardization (ISO) software product quality model. Thus, end-user developers can be disadvantaged relative to traditional software developers,
both in software development best practices as well as best practices that describe the software development environment.

To overcome inherent weaknesses of end-user development, SAS practitioners operating in these environments should invest in software development learning and training opportunities commensurate with their software development responsibilities. While I survived my tenure in the VA psych ward and did produce much quality software, I would have improved my skills (and software) had I read fewer *Diagnostic and Statistical Manual of Mental Disorders* (DSM) case studies and more SAS white papers and computer science texts.

**SOFTWARE DEVELOPMENT LIFE CYCLE (SDLC)**

The SDLC describes discrete phases through which software passes from cradle to grave. In a more generic sense, the SDLC is also referenced as the *systems* development life cycle, which bears the same acronym. Numerous representations of the SDLC exist; Figure 1.2 shows a common depiction.

In many data analytic and end-user development environments, the SDLC is not in place, and software is produced using an undisciplined, laissez-faire method sometimes referred to as *cowboy coding*. Notwithstanding any weaknesses this may present, the ISO software product quality model benefits these relaxed development environments, regardless of whether the SDLC phases are formally recognized or implemented. Because the distinct phases of the SDLC are repeatedly referenced throughout this text, readers who lack experience in formalized development environments should learn the concepts associated
with each phase so they can apply them (contextually, if not in practice) to their specific environment while reading this text.

**Planning** Project needs are identified and high-level discussions occur, such as the “build-versus-buy” decision of whether to develop software, purchase a solution, or abandon the project. Subsequent discussion should define the functionality and performance of proposed software, thus specifying its intended quality.

**Design** Function and performance, as they relate to technical implementation, are discussed. Whereas planning is needs-focused, design and later phases are solutions- and software-focused. In relation to quality, specific, measurable performance requirements should be created and, if formalized software testing is implemented, a test plan with test cases should be created.

**Development** Software is built to meet project needs and requirements, including accompanying documentation and other artifacts.

**Testing** Software is tested (against a test plan using test cases and test data, if these artifacts exist) and modified until it meets requirements.

**Acceptance** Software is validated to meet requirements and formally accepted by stakeholders as meeting the intended functional and performance objectives.

**Operation** Software is used for some intended duration. Where software maintenance is required, this occurs simultaneously with operation although these discrete activities may be performed by different individuals or teams.

**Maintenance** While software is in operation, maintenance or modification may be required. Types of maintenance are discussed in Chapter 13, “Maintainability,” and may be performed by users (in end-user development), by the original developers, or by a separate O&M team that supports software maintenance once development has concluded.

**End of Life** Software is phased out and replaced at some point; however, this should be an intentional decision by stakeholders rather than a retreat from software that, due to poor quality, no longer meets functional or performance requirements.

Although the SDLC is often depicted and conceptualized as containing discrete phases, significant interaction can occur between phases. For example, during the design phase, a developer may take a couple of days to do some
development work to test a theory to determine whether it will present a viable solution for the software project. Or, during testing, when significant vulnerabilities or defects are discovered, developers may need to overhaul software, including redesign and redevelopment. Thus, while SDLC phases are intended to represent the focus and majority of the work occurring at that time, their intent is not to exclude other activities that would naturally occur.

SDLC Roles

Roles such as customer, software developer, tester, and user are uniquely described in software development literature. While some cross-functional development teams do delineate responsibilities by role, in other environments, roles and responsibilities are combined. An extreme example of role combination is common in end-user development environments in which developers write, test, and use their own software—bestowing them with developer, tester, user, and possibly customer credentials. SAS end-user developers often have primary responsibilities in their respective business domain as researchers, analysts, scientists, and other professionals, but develop software to further these endeavors.

A stakeholder represents the “individual or organization having a right, share, claim, or interest in a system or in its possession of characteristics that meet their needs and expectations.” While the following distinct stakeholders are referenced throughout the text, readers should interpret and translate these definitions to their specific environments, in which multiple roles may be coalesced into a single individual and in which some roles may be absent:

**Sponsor** “The individual or group that provides the financial resources, in cash or in kind, for the project.” Sponsors are rarely discussed in this text but, as software funders, often dictate software quality requirements.

**Customer** “The entity or entities for whom the requirements are to be satisfied in the system being defined and developed.” The customer can be the product owner (in Agile or Scrum environments), project manager, sponsor, or other authority figure delegating requirements. This contrasts with some software development literature, especially Agile-related, in which the term customer often represents the software end user.

**SAS Practitioner/Developer** These are the folks in the trenches writing SAS code. I use the terms practitioner and developer interchangeably, but intentionally chose SAS practitioner because it embodies the panoply
of diverse professionals who use the SAS application to write SAS software to support their domain-specific work.

**Software Tester** Testers perform a quality assurance function to determine if software meets needs, requirements, and other technical specifications. A tester may be the developer who authored the code, a separate developer (as in software peer review), or an individual or quality assurance team whose sole responsibility is to test software.

**User** “The individual or organization that will use the project’s product.”

In end-user development environments, users constitute the SAS practitioners who wrote the software, while in other environments, users may be analysts or other stakeholders who operate SAS software but who are not responsible for software development, testing, or maintenance activities.

**Waterfall Software Development**

Waterfall software development methodologies employ a *stop-gate* or *phase-gate* approach to software development in which discrete phases are performed in sequence. For example, Figure 1.3 demonstrates that planning concludes before design commences, and all design concludes before development commences. This approach is commonly referred to as *big design up front*.

![Figure 1.3 Waterfall Development Methodology](image)
INTRODUCTION

(BDUF), because the end-state of software is expected to be fully imagined and prescribed in the initial design documentation, with emphasis on rigid adherence to this design.

For years, Waterfall methodologies have been anecdotally referred to as “traditional” software development. Since the rise of Agile software development methodologies in the early 2000s, however, an entire generation of software developers now exists who (fortunately) have never had to experience rigid Waterfall development, so the “traditional” nomenclature is outmoded. Waterfall development methodologies are often criticized because they force customers to predict all business needs up front and eschew flexibility of these initial designs; software products may be delivered on time, but weeks or months after customer needs or objectives have shifted to follow new business opportunities. Thus, the software produced may meet the original needs and requirements, but often fails to meet all current needs and requirements.

Despite the predominant panning of Waterfall development methodologies within contemporary software development literature, a benefit of Waterfall is its clear focus on SDLC phases, even if they are rigidly enforced. For example, because development follows planning and design, software developers only write software after careful consideration of business needs and identification of a way ahead to achieve those objectives. Further, because all software is developed before testing, the testing phase comprehensively validates function and performance against requirements. Thus, despite its rigidity, the phase-gate approach encourages quality controls between discrete phases of the SDLC.

Agile Software Development

Agile software development methodologies contrast with Waterfall methodologies in that Agile methodologies emphasize responsible flexibility through rapid, incremental, iterative design and development. Agile methodologies follow the Manifesto for Agile Software Development (AKA the Agile Manifesto) and include Scrum, Lean, Extreme Programming (XP), Crystal, Scaled Agile Framework (SAFe), Kanban, and others.

The Agile Manifesto was cultivated by a group of 17 software development gurus who met in Snowbird, Utah, in 2001 to elicit and define a body of knowledge that prescribes best practices for software development.
Manifesto for Agile Software Development

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

**Individuals and interactions** over processes and tools
**Working software** over comprehensive documentation
**Customer collaboration** over contract negotiation
**Responding to change** over following a plan

That is, while there is value in the items on the right, we value the items on the left more.\(^5\)

In Agile development environments, software is produced through iterative development in which the entire SDLC occurs within a time-boxed iteration, typically from two to eight weeks. Within that iteration, software design, development, testing, validation, and production occur so that working software is released to the customer at the end of the iteration. At that point, customers prioritize additional functionality or performance to be included in future development iterations. Customers benefit because they can pursue new opportunities and business value during software development, rather than be forced to continue funding or leading software projects whose value decreases over an extended SDLC due to shifting business needs, opportunities, risks, and priorities. Figure 1.4 demonstrates Agile software development in which software is developed in a series of two-week iterations.

Agile is sometimes conceptualized as a series of miniature SDLC life cycles and, while this does describe the iterative nature of Agile development, it fails to fully capture Agile principles and processes. For example, because Agile development releases software iteratively, maintenance issues from previous iterations may bubble up to the surface during a current iteration, forcing developers (or their customers) to choose between performing necessary maintenance or releasing new functionality or performance as scheduled. Thus, a weakness ascribed to Agile is the inherent competition that exists between new development and maintenance activities, which is discussed in the “Maintenance in Agile Environments” section in Chapter 13,
“Maintainability.” This competition contrasts with Waterfall environments, in which software maintenance is performed primarily once software is in production and development tasks have largely concluded.

Despite this potential weakness, Agile has been lauded as a best practice in software development for more than a decade and has defined software development in the 21st century. Its prominence within traditional applications development environments, however, has not been mirrored within data analytic development environments. This is due in part to the predominance of end-user developers who support data analytic development and who are likely more focused on domain-specific best practices rather than software development methodologies and best practices.

Another weakness is found in the body of Agile literature itself, which often depicts an idealized “developer” archetype whose responsibilities seem focused narrowly on the development of releasable code rather than the creation of data products or participation in other activities that confer business value. In these software-centric Agile descriptions, common activities in data analytic development environments (such as data analysis or report writing) are often absent or only offhandedly referenced. Despite this myopia within Agile literature, Agile methodologies, principles, and techniques are wholly applicable to and advisable for data analytic development.

For those interested in exploring Agile methodologies, dozens of excellent resources exist, although these typically describe traditional software applications development. For an introduction to Agile methodologies to support data analytic development, I demonstrate the successful application of Agile to

**RISK**

From a software development perspective, basic risks include functional and performance failure in software. For example, a nonmalicious threat (like big data) exploits a software vulnerability (like an underlying error that limits efficient scaling when big data are encountered), causing risk (inefficient performance or functional failure) to business value. These terms are defined in the text box “Threat, Vulnerability, and Risk.” While the *Project Management Body of Knowledge* (PMBOK) and other sources define positive risk as opportunity, only negative risks are discussed within this text.

### THREAT, VULNERABILITY, AND RISK

**Threat**  “A risk that would have a negative effect on one or more project objectives.”

“A state of the system or system environment which can lead to adverse effect in one or more given risk dimensions.”

**Vulnerability**  “Weakness in an information system, or cryptographic system, or components (e.g., system security procedures, hardware design, internal controls) that could be exploited.”

**Risk**  “An uncertain event or condition that, if it occurs, has a positive or negative effect on one or more project objectives.”  “The combination of the probability of an abnormal event or failure and the consequence(s) of that event or failure to a system’s components, operators, users, or environment.”

**Failure**

Software failure is typically caused by threats that exploit vulnerabilities, but neither all threats nor all vulnerabilities will lead to failure. Errors (human mistakes) may lie dormant in code as vulnerabilities that may or may not be known to developers. Unknown vulnerabilities include coding mistakes (*defects*) that have not yet resulted in failure, while known vulnerabilities include coding mistakes (*latent defects*) that are identified yet unresolved. The “Paths to Failure” section in Chapter 4, “Reliability,” further defines and distinguishes these terms.
For example, the following SAS code is often represented in literature as a method to determine the number of observations in a data set:

```
proc sql noprint;
    select count(*)
    into :obstot
    from temp;
quit;
```

The code is effective for data sets that have fewer than 100 million observations but, as this threshold is crossed, the &OBSTOT changes from numeric to scientific notation. For example, a data set having 10 million observations is represented as 10000000, while 100 million observations is represented as 1E8. To the SAS practitioner running this code to view the number of observations in the log, this discrepancy causes no problems. However, if a subsequent procedure attempts to evaluate or compare &OBSTOT, runtime errors can occur if the evaluated number is in scientific notation. This confusion is noted in the following output:

```
%let obstot=1E8;
%if &obstot<5000000 %then %put LESS THAN 5 MILLION;
%else %put GREATER THAN 5 MILLION;
```

```
LESS THAN 5 MILLION
```

Obviously 100 million is not less than 5 million but, because of two underlying errors, a vulnerability in the code exists. The vulnerability can be easily eliminated by correcting either of the two errors. The first error can be eliminated by changing the assignment of &OBSTOT to include a format that will accommodate larger numbers, as demonstrated with the FORMAT statement. The second error can be eliminated by enclosing the numeric comparison inside the %SYSEVALF macro function, which interprets 1E8 as a number rather than text. Both solutions are demonstrated and either correction in isolation eliminates the vulnerability and prevents the failure.

```
proc sql noprint;
    select count(*) format=15.0
    into :obstot
    from temp;
quit;

%if %sysevalf(&obstot<5000000) %then %put LESS THAN 5 MILLION;
%else %put GREATER THAN 5 MILLION;
```

```
GREATER THAN 5 MILLION
```
Because the failure occurs only as the number of observations increases, this can be described as a scalability error. The SAS practitioner failed to imagine (and test) what would occur if a large data set were encountered. But if the 100 million observation threshold is never crossed, the code will continue to execute without failure despite still containing errors. This error type is discussed further in the “SAS Application Thresholds” section in Chapter 9, “Scalability.”

Developers often intentionally introduce vulnerabilities into software. For example, a developer familiar with the previous software vulnerability (exploited by the threat of big data) might choose to ignore the error in software designed to process data sets of 10,000 or fewer observations. Because the risk is negligible, it can be accepted, and the software can be released as is—with the vulnerability. In other cases, threats may pose higher risks, yet the risks are still accepted because the cost to eliminate or mitigate them outweighs the benefit.

Unexploited vulnerabilities don’t diminish software reliability because no failure occurs. For example, the previous latent defect is never exploited because big data are never encountered. However, vulnerabilities do increase the risk of software failures; therefore, developers should be aware of specific risks to software. In this example, the risk posed is failure caused by the accidental processing of big data within the SQL procedure. When vulnerabilities are exploited and runtime errors or other failures occur, software reliability is diminished. The risk register, introduced in the next section, enables SAS practitioners to record known vulnerabilities, expected risks, and proposed solutions to best measure and manage risk level for software products.

**Risk Register**

A risk register is a “record of information about identified risks.”¹¹ Risk is an inherent reality of all software applications, so risk registers (sometimes referred to as defect databases) document risks, threats, vulnerabilities, and related information throughout the SDLC. Developers and other stakeholders should decide which performance requirements to incorporate in software, but likely will not include all performance requirements in all software. While vulnerabilities will exist in software, it’s important they be identified, investigated, and documented sufficiently to demonstrate the specific risks they pose to software operation.
A risk register qualitatively and quantitatively records known vulnerabilities and associated threats and risks to software function or performance, and can include the following elements:

- Description of vulnerability
- Location of vulnerability
- Threat(s) to vulnerability
- Risk if vulnerability is exploited
- Severity of risk
- Probability of risk
- Likelihood of discovery
- Cost to eliminate or mitigate risk
- Recommended resolution

Some risk registers, as demonstrated, are organized at the defect level while others are organized at the threat or risk level. Vulnerability-level risk registers are common in software development because while many threats lie outside the control of developers, programmatic solutions can often be implemented to eliminate or mitigate specific vulnerabilities. Moreover, general threats like “big data” can exploit numerous, unrelated vulnerabilities within a single software product.

Table 1.1 depicts a simplified risk register for two of the errors mentioned in the code. The risk severity, risk probability, likelihood of risk discovery, and cost to implement solution are demonstrated on a scale of 1 to 5, in which 5 is more severe, more likely to occur, less easy to discover, and more costly to repair.

The first and second risks describe separate vulnerabilities, each exploited by the threat of data sets containing 100 million or more observations. Despite the high severity (5) if the threat is encountered, the likelihood is low (1) because these file sizes have never been encountered in this environment.

<table>
<thead>
<tr>
<th>Num</th>
<th>Vulnerability</th>
<th>Location</th>
<th>Risk</th>
<th>Risk Severity</th>
<th>Risk Probability</th>
<th>Risk Discovery</th>
<th>Risk Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>%SYSEVALF should be used in evaluation</td>
<td>less-than operator</td>
<td>scientific notation won’t be interpreted correctly</td>
<td>5</td>
<td>1</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>no format statement</td>
<td>SELECT statement of PROC SQL</td>
<td>scientific notation won’t be interpreted correctly</td>
<td>5</td>
<td>1</td>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>
If these two factors alone were considered, a development team might choose to accept the risk and release the software with the vulnerabilities, given their unlikelihood of occurrence. However, because the likelihood of discovery is low (5)—as no warning or runtime error would be produced if the threat were encountered—and because the cost to implement a remedy (modifying one line of code) is low (1), the development team might instead decide to modify the code, thus eliminating the risk rather than accepting it.

Not depicted, the recommended solution describes the path chosen to manage the risk—often distilled as avoidance, transfer, acceptance, or mitigation, and described in the following section. The recommended solution may contain a technical description of how the risk is being managed. For example, if a risk is being eliminated, the resolution might describe programmatically how the associated threat is being eliminated or controlled or how the associated vulnerability is being eliminated so it can’t be exploited by the threat.

**Risk Management**

*Risk management* describes the “coordinated activities to direct and control an organization with regard to risk.”12 ISO further defines a *risk management framework* as a “set of components that provide the foundations and organizational arrangements for designing, implementing, monitoring, reviewing and improving risk management throughout the organization.” The risk to operational software discussed throughout this text is software failure—functional- or performance-related; risk management accepts failure as an inevitable reality and strives to overcome it sufficiently to deliver an acceptable level of risk to stakeholders. Failure does not necessarily denote runtime errors but describes software that fails to meet functional or performance requirements.

The risk register typically includes not only theoretical vulnerabilities within software that have never been exploited but also risks that have materialized through actual software failure. These latter failures will also be demonstrated in the failure log (introduced in the “Failure Log” section in Chapter 4, “Reliability”) if it is utilized. Risk resolution doesn’t imply that risk is being eliminated; in many cases, once vulnerabilities are identified, a decision is made to accept the risk and to leave the code unchanged. Typical risk resolution patterns include:

**Risk Avoidance** “Informed decision not to be involved in, or to withdraw from, an activity in order not to be exposed to a particular risk.”13 For example, if your software is broken, stop using it.

**Risk Sharing** “Form of risk treatment involving the agreed distribution of risk with other parties.”14 For example, some SLAs state that if the power
fails (causing production software to fail), the responsibility is transferred to the organization, and is not borne by the O&M team, developers, or users. *PMBOK and other sources use the alternative term “risk transfer.”*  

**Risk Retention** “Acceptance of the potential benefit of gain, or burden of loss, from a particular risk.”  

Stakeholders retain or accept a risk when they evaluate and decide it’s prudent not to resolve the risk. *PMBOK and other sources use the alternative term “risk acceptance.”*  

**Risk Mitigation** “A course of action taken to reduce the probability of and potential loss from a risk factor.”  

For example, a programmatic solution may be implemented to lessen or eliminate a vulnerability in software, such as by removing a defect.

A risk register is important to production software because it provides a common framework for internal stakeholders to gauge the relative strength or weakness of software. In some cases, separate quantitative priorities are established in a single register; therefore, a developer might prioritize fixing one element of code while a customer might prioritize fixing a different element, allowing stakeholders effectively to vote for how to best improve software. By evaluating the aggregate risk of software, all stakeholders can gain a clearer understanding of its quality. And, because the risk register is meant to be a living, flexible document, if aggregate software risk becomes too high over time, risks that were once accepted can later be mitigated or eliminated to restore risk to an acceptable level.

**WHAT’S NEXT?**

In the next chapter, technical and nontechnical definitions of quality are discussed, including how personal perspective can influence the interpretation of quality. The ISO software product quality model is introduced and its benefit to SAS practitioners and development teams demonstrated. The roles of functional and performance (i.e., non-functional) requirements are demonstrated, and the importance of clear, measurable technical requirements is highlighted.

**NOTES**


12. Id.
13. Id.
14. Id.
17. Project Management Institute, 2013.
About the Author

Troy Martin Hughes has been a SAS practitioner for more than 15 years, has managed SAS projects in support of federal, state, and local government initiatives, and is a SAS Certified Advanced Programmer, SAS Certified Base Programmer, SAS Certified Clinical Trials Programmer, and SAS Professional V8. He has an MBA in information systems management and additional credentials, including: PMP, PMI-ACP, PMI-PBA, PMI-RMP, CISSP, CSSLP, CSM, CSD, CSPO, CSP, and ITIL v3 Foundation. He has been a frequent presenter and invited speaker at SAS user conferences, including SAS Global Forum, WUSS, MWSUG, SCSUG, SESUG, and PharmaSUG. Troy is a U.S. Navy veteran with two tours of duty in Afghanistan and, in his spare time, a volunteer firefighter and EMT.
Gain Greater Insight into Your SAS® Software with SAS Books.

Discover all that you need on your journey to knowledge and empowerment.

support.sas.com/store/books
for additional books and resources.