JOHN LLOYD

introducing system engineering to the system admin

John Lloyd is a computer systems engineer at MDA, a Canadian technology company that builds and operates systems ranging from land-title information to space robotics. He has been doing system administration and system engineering on machines ranging from PDP-11s to SGI Altix for nearly 30 years.

John.Lloyd@MDAcorporation.com

THE METHODS OF SYSTEM ENGINEERing (SE) can help solve some typical system construction and operation problems. Using a case study as a demonstration, in this article you will learn to use SE methods to build a simple database system.

A System Administrator's Story

As a system administrator, you are asked to put together a computer system to provide an Oracle database for an Internet Web site. The project leader explains to you that costs have to be low but the system has to be highly available, with better than 99.9% uptime. The Oracle software required by the Web page developers who are providing active (programmed content) Web pages to the system's end users largely dictates the capacity of the system you need to acquire.

You find some used Sun gear, consisting of an A1000 SCSI RAID system and an E450 four-way SPARC computer. The Oracle DBA (database administrator) is very pleased, since this system contains twelve 72 GB disks and has four processors. This seems to be plenty. The project manager is pleased, since you saved lots of money by buying used equipment.

Over the next year the computer system goes through several major and minor crises.

For example, one day a disk in the RAID storage array fails; you notice the yellow light and replace the disk with a spare you thoughtfully ordered with the system. No data is lost. The project manager is happy and publicly congratulates you.

A few months later the software developers phone you at home complaining that the database server has stopped functioning during a software upgrade. After some investigation you discover one of the Oracle filesystems is full. After half an hour of careful and patient probing, you discover the developers have to modify several very large tables in the database and are doing this one table at a time. It turns out this is what consumes disk space—complete copies of the original table are stored in the Oracle "rollback" space. The software upgrade has to be aborted for lack of adequate disk space. Because the half-completed upgrade process has left the database unusable, it takes all night and most of the next day to restore the database, resulting in a full day of downtime.

The next day the project manager gets very angry when he discovers what had happened the night

before. Everyone points fingers at the other groups: The DBA complains about disk space; the developers complain about the Oracle software behavior, and you complain about developers not knowing the consequences of their actions. No points for anybody. But you get budget approval for a disk upgrade.

The third episode revolves around another disk failure. This time one of the system disks fails. You have thoughtfully implemented software RAID 1 (disk mirroring). But upon replacing the failed disk you discover the system will not boot. After several hours of investigation, you discover you've made a basic error in setting up the system disk mirroring. The fix involves a complete system-disk replication and a planned one-hour shutdown has turned into a 12-hour outage. Once more, the project manager is unhappy.

At your annual performance review, the project manager points out this history. He states that as a result, the system performance was not met, since the system was down for two days, resulting in only 99.45% uptime for the year. You respond by acknowledging the system-disk problem, but you refuse to accept responsibility for the Oracle disk-space issue; it was not under your control. You point out the success of the RAID disk replacement. By your accounting, system availability was 99.86% because the system was only down for 12 hours on a Saturday, and everyone knows the customers don't use it on weekends. That's "pretty close" to 99.9%, isn't it?

System Engineering to the Rescue

System engineering is the systematic application of engineering methods to identify the issues and requirements, develop and evaluate alternative architectures and designs, implement and test the selected design, and verify the correctness of a system implementation.

This is a lengthy definition, and you may well ask, "How do all these activities help this situation?" The short answer is that SE enables successful implementation of the features that are important to the customer. For the example given here, this means identifying and meeting the 99.9% uptime requirement given all foreseeable uses of the system.

The hardware failures cited in the case study could not have been avoided. However, the systematic failure to provide disk space could have been prevented by SE processes. System engineering provides the means to design for and test the system response to identifiable situations, including software upgrades.

A Simplified SE Process

SE uses several types of design information, including figures, tables, and text. By using a specified procedure, they are developed and maintained as documents. These will require some effort to generate and maintain. The value of the SE process outweighs the effort required to maintain these documents.

The general outline of the process is as follows. First, identify the system to be built, ensuring that interfaces to outside systems are well defined. Second, define system functionality by writing requirements statements and casting them in a testable form. Third, design the system by methodically placing the list of requirements in the design. After assembly and construction, test the system by showing that it meets the stated requirements.

In terms of SE, identifying requirements is the central goal. Providing clearly defined requirements that are unlikely to change makes designing, implementing, and acceptance-testing understandable and manageable. Additionally, a clear and accurate set of requirements for a system provides stability to the designers, and to the implementers.

System Boundaries

Identifying the system to be built provides the boundaries of the system and the information flows which the system provides or acts on. System requirements can then be stated in terms of these flows.

In our case study, the system admin implicitly identified the system as a computer and Oracle database software. The system admin did not consider the Web server software as part of system admin responsibilities. The software developers understood the system as Java code running on the Oracle database supplied by the system admin. They understood the database in abstract terms, as a data storage system.

Each group had a different view of what the complete Web server system consisted of. Neither fully understood the scope or extent of their views, or the relationship of their part to the organization's real goal of operating a Web site. As a result, the combination of computer and software did not meet management's expectations.

By taking the steps to formally list the actual interactions between these two views of the system, we can define and separate two separate subsystems and the information flows between them. This process provides a way to clearly identify where the database system begins and ends. It also provides the basis for defining the expectations for the database (the requirements).

The interactions between database software and Web application are familiar to most developers and system admins. These interactions, in addition to the usual ones required for any computer system (monitoring, backups, etc.), define the list of interfaces to the database subsystem. By taking this environment into consideration, these interactions define the scope or the extent of the database subsystem. In other words, we identify the database system by defining its interactions with external systems in its environment.

The database system environment is illustrated in Figure 1, as a context diagram. In this figure, each interface with an external entity is identified and the flows of information are shown by the arrows. The intent here is to show all substantial information flows, while excluding uninteresting flows, for example, power and cooling, details of user interfaces, and operating system installations. To supplement this figure, we normally add a table describing the key types of information exchanged on each interface. An example is given in Table 1 (facing page).

The list of interfaces and their data flows will be used as the basis for defining functions and behaviors of the system, also called "requirements." But first, let's take a more detailed look at the figure and table.

Experienced system designers might have some comments about these interfaces as I have described them. The database subsystem is responsible for backups, and opinions may vary on whether the data-model updates are a software upgrade or an information flow. From the system admin point of view, they are data items exchanged with the Web developers and are considered as data.

Interface	Item	Description
Application queries	Oracle queries and inserts	SQL statements operating on the set of application tables
Developer updates	Software updates	Replacement .jar files and .war files comprising the Java application are supplied by the software developers
	Data model updates	Scripts to convert the schema to the next version, and to undo this conversion
Offsite backup tapes	Media	Online Oracle backup tapes, in tar format, are sent offsite by courier
Sysadmin monitoring	Logs	System logs
DBA monitoring	Logs	Database alerts and trace logs

TABLE 1: EXTERNAL INTERFACES

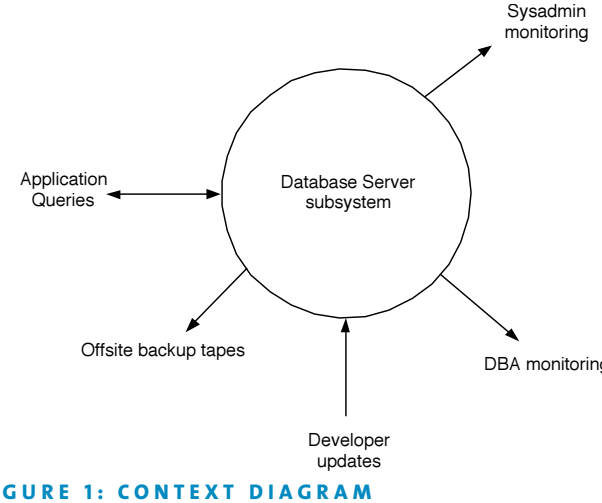

FIGURE 1: CONTEXT DIAGRAM

Here are some simple rules for composing a context diagram:

- Significant information flows, together with directions, are shown.
- Information flows with separate external entities are shown.
- Control flows are excluded.
- Heat, power, light, and space are excluded.
- Software installation, configuration, and management are excluded.

The last rule seems to be broken in my example. In this case, the database subsystem operates on data, and here the data happens to be considered as software by another subsystem.

In addition to the diagram, provide a table (see above) identifying and describing each information flow. The interfaces and flows define the operational context and the extent and scope of the system. Now any functions or information can easily be identified as part of this system, or not.

With the context and information flows between the system and significant externals identified, we can examine functional behavior of the system.

Requirements

With the boundaries of the system identified, we can now address system requirements. These are a set of statements that clearly state all of the

"real" behaviors, functions, and system characteristics; they identify the pertinent functions and capabilities of the system.

Good requirements do not impose or specify a design. Good requirements describe the system behavior; they do not describe the internals of the system. The goal of requirements analysis is to provide objective descriptions of the system. Each requirement, therefore, must also be testable; we need to be able to verify the defined behaviors or attributes. (It is worth noting that writing testable requirements saves time and trouble later.)

Requirements are the most important and critical element of SE. As the basis for the system design phase, they must be complete and understandable. They form the basis for testing, where the tester must show that each requirement is met. Requirements also describe characteristics, such as system reliability or usability, that have to be built into the system design and proven by analysis, examination, or testing of the assembled system. Requirements are also the principal means of communication among developers, sysadmins, and the system users and owners. They are (or should be) mutually understood and agreed upon.

What are "good" requirements statements? Here are some guidelines:

- They can be demonstrated, measured, or verified by analysis.
- They do not contain adjectives that are subjective.
- They do not contain "and" or "or" to separate multiple requirements.
- They do not contain design descriptions or prescriptions.
- They may contain constraints such as preferred vendors or technologies.

Here are a few good examples:

- Shall run Oracle version 10GR2 (this constraint is demonstrable).
- Shall support 120 GB of disk storage (demonstrable).
- Shall perform daily full backups of Oracle data (demonstrable).
- Shall perform daily full backups of operating system (demonstrable).
- Shall support 6 Web logins per second (measurable).
- Shall be available 99.9% of a calendar year (verifiable by analyses).

These examples are clear and concise. Although a software product is specified, this is more of a constraint on the designer than a design requirement, but it is realistic. The sizing statement can be demonstrated by examining results of a command (no arithmetic is needed). The backup functions are separated into two statements, to avoid "and." The availability requirement uses calculations based on vendor-supplied failure rates, and is therefore verifiable by analysis (99.9% of a year is about 8.75 hours downtime per year).

Some examples of bad requirements statements are:

- Shall provide appropriate backup system (adjective is subjective).
- Shall prevent insecure logins ("insecure logins" undefined).
- Ought to be available 7x24x365 (not a yes/no requirement).
- Cannot allow hackers in (unachievable).
- Has to use RAID1 on all database disks (design imposition).
- Shall use Java for writing the Web app (not applicable to this subsystem).
- Should use disk-to-disk backups for Oracle data (design imposition).

These examples show some possible errors of composition or design specification. Some use undefined terms, define impossible goals, or include statements that are not applicable to this particular system.

The last example is a classic requirements error. The "obvious" function to require disk-to-disk backups is stated as a requirement. Fixing this statement requires some detective work to uncover the true purpose: Is the intended meaning to limit recovery time from backups, or is some other objective sought? The requirement should be rewritten to specify the actual need (e.g., "A database restore shall take less than 4 hours").

The benefit of a set of good requirements is simplicity. Good requirements make it easy to understand the functions and characteristics of a system. They also make these statements readily verifiable.

One recurring problem that arises when writing requirements is the risk of missing some "important" requirement. This is a constant issue for SE, and there are a number of ways to reduce this risk. One way is to establish "use cases," as is common in software engineering. These are paper exercises where the functions, inputs, and outputs of the system are considered for different scenarios. These descriptions are checked against the list of requirements. Another method is evaluating the lifecycle of each key data item or information item. We can identify the birth, life, and death (or permanent archival) of each data item and verify that requirements cover every step.

How many requirements statements are enough? If there are too many, the system may not be achievable because of conflicting requirements; too few, and the users have not communicated enough about what is needed. A short answer is: Enough to be clear on the goals of the system.

System Design

The system designer identifies useful components such as computers, networks, and software and connects them in a way that meets the system objectives. In this example, we have already committed the act of design by separating the Web site into a database subsystem and a software subsystem, and connecting them by their data flows. It's not always that simple.

The context diagram describes the database subsystem in its environment, whereas the requirements list identifies the functions and information processed by the database subsystem. The database design provides components to support each information flow in the context diagram and each function and each characteristic identified in the requirements.

Based on experience, research on vendor guidelines, recommendations of peers, or pure invention, the designer identifies the components required to complete the design and composes a figure showing their relationships, as in Figure 2 (next page). In this figure, boxes are the components, circled numbers identify internal information flows, and labeled arrows show flows identified in the context diagram.

This is what is commonly called the design of the system. It shows the computer parts, backup system, two kinds of storage, and related connections. It does not show unnecessary detail such as electrical power, system installation or configuration procedures, cost, etc.

Something very important should accompany this design figure. Each component of the design needs to be matched to one or more requirement statements (see Table 2, next page). This is a very significant (and some-

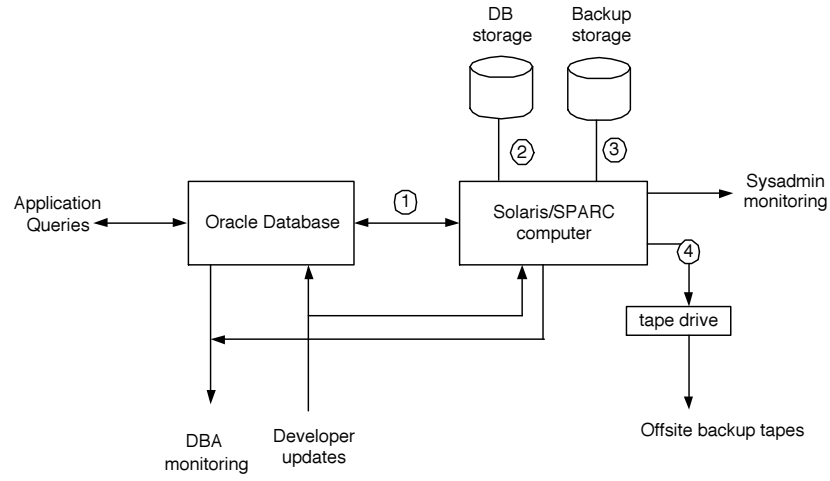

FIGURE 2: SYSTEM DESIGN

Category	Requirement	Allocated Component
Oracle	Shall run Oracle 10GR2	Oracle database
	Shall perform daily full backups of Oracle	Oracle database, Solaris/SPARC computer, DB storage, backup storage
Availability	Shall be available 99.9%	Solaris/SPARC computer, DB storage

TABLE 2: REQUIREMENTS ALLOCATION TO DESIGN

times troublesome) step. Matching components to requirements guarantees two things:

- Each requirement is covered by at least one component of the design.
- All design components are necessary.

By matching a requirement to one or more components, the designer ensures that the requirement can be satisfied by the functions of those components.

If the design contains components not related to official requirements, then there is something funny going on. Possibly, requirements are missing, the designer has made a mistake, or something in the process has broken. This can be a very useful test of the process and a clear quality-assurance check on both the design and the requirements. Getting to this point (clear requirements and consistent and clear design) is a major milestone in the process and has clearly significant benefits for the project.

For the design to be valid and useful, it should follow some rules. First, the design must match the context diagram. Each data flow in the context diagram must match the corresponding flow in the design. The design should maintain a fairly consistent level of abstraction, providing information flows between components rather than reflecting physical connectivity. (This can be a subjective measure, and hard to achieve in some cases.)

As shown in our design, we have identified internal information flows, and so we should identify what goes on in each flow. Some are very simple. In the figure, number 4 is a SCSI or Fibre Channel interconnect, whereas number 1 will reflect all the information flows between a complex database product and the underlying operating system. You should be able to identify each element and describe its functions. The flows of information between each element should be described as well. This combination of (1) a figure showing relationships between components and (2) information flows defines the design.

Interface	Item	Description
1. Oracle-to-Solaris	Data storage for tablespaces	Filesystems for storing tables
	Control-file locations	Distinct filesystems on separate devices to hold copies of Oracle Control-files
	Operating system interface	Process creation, memory mapping, I/O and other standard OS features
2. DB store	Disk I/O	SCSI-compatible disk storage
3. Backup store	Database backups	SCSI-compatible disk storage
4. Tape interface	System backups	SCSI-compatible tape drive

TABLE 3: INTERNAL INTERFACES

Even this figure and accompanying tables, of course, are not enough to fully characterize the design. Performance and scalability are not described in this figure; an analysis of system availability is needed, and so on. In fact, a significant effort is normally made to analyze the characteristics of a design. If an analysis shows noncompliance to requirements, the designer will necessarily have to modify the design or even try to renegotiate requirements.

Readers familiar with software systems design will recognize a pattern here. Requirement statements are formally linked to each element of the system. The links are traced out in both directions, and these are thoroughly checked.

The design must be proven by assembling the components and by checking each requirement. This is the actual work of constructing the system.

Integration and Testing

Once the design is completed (and this means that the requirements set is well defined and that results of various design analyses are satisfactory), the system is assembled and tested.

Most system admins have experience assembling computer systems such as this one. For this small example, it is fairly straightforward to assemble the components, install the software, check vendor Web sites for the latest bug fixes and patches, and complete the patching and configuration of the system. Many might object that formal testing of the system is not really required; they know what they are doing.

The point is that testing applies to the complete system, including interaction with the Web application, not just the database server. Merely testing the database system does not uncover potential systemic errors. Covering all identified requirements makes the complete system much more trustworthy, because it has been tested. Testing should be applied in a controlled, reproducible process to whatever degree of formality is needed by the paying customer, even if it is your own organization.

Preparing tests and performing testing provides three benefits:

- It confirms that requirements are met.
- It provides reproducible tests for future system-integrity checking.
- It confirms valid requirements.

The first two points should be clear. The last point requires some explanation.

The SE process as I've described it includes many references to the requirements set. This dependency is obvious and essential—meeting requirements is the goal. However, a good requirements set has to be valid, in the sense that the requirements must accurately describe the intent of the customer. We all know that customer intentions can change, owing to market conditions, technical advances, or simply improved understanding of the problem being addressed. We can use system testing, or even just the process of writing system tests, as a way to help discover true requirements based on customer intentions. The tests need to be approved or reviewed by the customer: This is their chance to confirm their validity.

Finding the Systemic Problems

Here lies the solution to the two system issues that lay undetected in the Web services system described in the case study. First, the system-disk recovery process was simply not tested; therefore no one discovered it was broken. A formal test with simulated disk failure would have uncovered the problem and resulted in a revised design.

Second, the process of upgrading the Web application software and implementing a data model change (see Table 1) would also have been tested. This test might not have uncovered a storage problem unless the disk usage was measured during the change. It's unlikely anyone would include such a measurement in a test procedure, that is, not unless there is another, real requirement statement:

> Requirement: The system shall support 1 year of online data.

Is this a real requirement? Yes, it is measurable and does not depend on the application or the particulars of implementation. And it is verifiable by adding a year's worth of simulated data to the database and running tests. A suitable test case (perform a data model change with one year of data) would have uncovered the second systemic fault before it was discovered operationally. The operational problem would have been avoided.

Other potential problems can be discovered, such as meeting the Web login rate (6 per second) when the database contains a year of online data and database backups are in progress. Systematic test planning can help uncover this scenario.

Conclusion

We used a case study of a typical database implementation to introduce systematic review and testing of system requirements. Our systematic process is called "system engineering" and meets the goal of providing an understandable, usable process to generate and deploy computer systems that correctly meet the customer's defined goals.

There is much more that can be said: I've omitted some key topics from this description, including the necessity of revising the requirements, designs, and test plans as new insights are gained. I have not described in detail how to document interfaces. The politics of uncovering and documenting requirements from sometimes uncooperative customers has not been described. Also, I have not included a description of document configuration management, which is essential to maintaining control of the SE process.

The thoughtful reader will recognize that the same methods described here for designing a database system can be used for each component of the

database system in turn, resulting in a hierarchy of context diagrams, designs, component context diagrams, component designs, and so on in greater detail. This is a key benefit of SE; once learned and applied, it can be used repeatedly over and over throughout large system implementations.

RESOURCES

The definitive text on SE is *Systems Engineering and Analysis* (4th ed.) by B.S. Blanchard and W.J. Fabrycky (Englewood Cliffs, N.J.: Prentice-Hall, 2005). This is not, however, an easy introduction to the topic. It describes SE for very large systems and includes the application of the SE process to the production of the systems (factories) as well as the final product and long-term support of systems in the field. Earlier editions are useful too.

The U.S. Defense Systems Management College's *Systems Engineering Fundamentals* (Fort Belvoir, Virginia: DSMC, 2001) is shorter than Blanchard and Fabrycky's book, but it requires careful reading. It describes SE in 40 pages (of a total of 200), with the rest of the volume describing management of the SE processes. It uses more technical (and somewhat different) jargon and references some U.S. DoD standards. It is freely available on the Web.

INCOSE (International Council on Systems Engineering) publishes a single-volume handbook, available to members. See www.incose.org.

International standards on SE include ISO 15288, EIA/IS 731, and IEEE 1498. These are often costly to obtain, and a textbook (or three) is more likely to be useful. Freely available standards include ECSS (www.ecss.nl), MIL-SPEC 498, and DOD-STD 2167A, although the latter two are more software-oriented.