Just Enough Systems Engineering

A review of the basics and how to use them

About the Author

Dwayne Phillips worked as a systems and computer engineer for 28 years for the U.S. Government. He is now a systems engineer with ITT in Herndon, Virginia. He lives with his wife of over 25 years Karen and near his three sons and (as of now) one daughter-in-law and one grandson.

Dwayne's web site is http://dwaynephillips.net and his email address is d.phillips@computer.org.

If you have any questions from this book, please ask Dwayne.

If you find any typos or other errors in this book, don't hesitate to contact Dwayne.

About the Artist

The cartoons in this book were drawn by Mr. Mark Tatro of Rotate Graphics (rotategraphics.com). Working with Mark is a pleasure. Contact him at mark@rotategraphics.com if you would like him to do art for you.

About your rights and my rights

If you know a publisher who would like to publish this material, please send them Dwayne's information.

Table of Contents

Section 1

Systems Engineering

This text begins with a one-chapter section describing systems engineering. There are many definitions of systems engineering, so I add several of my own hoping to clarify rather than confuse. My definitions concentrate on what a systems engineer does and how the systems engineer relates to other people.

1 Systems Engineering is...

My first question in a text about systems engineering is, "What is systems engineering?" This is a fair question, and one that I think we should consider.

I started my professional life writing computer programs. After a few years of programming, I took on the title of "software engineer." That sounded good and, considering I had a degree or two in engineering, seemed appropriate. I wasn't really sure what distinguished a programmer from a software engineer, but I was young and impressionable and liked the title. I was a software engineer and I had colleagues called hardware engineers. Software and hardware pretty much covered everything as what else is there? Then one day I heard the job title "systems engineer." I wasn't sure what that was, but it sounded like something better or something that would receive a higher pay than either a software or hardware engineer.

I wandered about asking people what systems engineering was. I even took a one-week class titled "systems engineering." My questions about systems engineering prompted plenty of blank stares and a few, "Systems engineering is what we are doing now. So get to work."

Blank stares and admonitions to "get to work" didn't do me much good. I kept looking for that definition that would click in my mind and spur me on to great things.

The best definition I found at the time was:

> Almost no one agrees what systems engineering is, but most engineers feel that they are doing it.
>
> (source unknown)

This wasn't much, but it was something to go on.

I continued my search and my work. Along the way I found several definitions that I studied, but didn't like much. The next few pages show and discuss those.

A Definition

Shown below is a definition that I found in an IEEE publication [IEEE]. I added the numbers inside the parentheses to help refer to my comments below. Please take a moment (or ten) to read and contemplate this definition.

> There are probably more (1) definitions of "Systems Engineering" than there are AESS members. In its simplest form (2) systems engineering is the design of the whole as opposed to the design of the parts. The vast number, complexity and diversity of elements can (3) overwhelm and degrade system performance and reliability. Embedded processing and software can be both a boon and a bane. A systems engineer analyzes and optimizes an ensemble of elements that relate to the flow of energy, mass and communications into a design that performs the desired function. "Systems engineering" is used herein to cover a very broad spectrum of processes and controls to engineer a product at the many levels required to satisfy all aspects of the original requirement. Our definition is not intended to either include or exclude systems engineering and integration as used in the computer field. In any case, systems engineering is the application of solid engineering principles to design and develop a large enterprise within cost and schedule to satisfy the needs of the ultimate user. It involves conceptualization, design, development, test, implementation, approval/certification and operation (including human factors) of a system. In essence, systems engineering is a problem-solving discipline for the (4) modern world.

Some commentary on this definition. For (1), I agree whole-heartedly. There are many definitions of systems engineering. I, however, don't agree that the authors of the quoted paragraph have to contribute yet another one.

For (2), I agree once again. This is a common definition that people utter when someone asks repeatedly for some clarity. "Systems engineering is the design of the whole system." I liked that definition the first few dozen times I heard it. My fondness waned with time, however, as I struggled to find a whole system that needed designing.

For (3), I am overwhelmed with the number, diversity, and complexity of the words in this definition.

For (4), while someone may not have used the term "systems engineering" until recently, I think that people did build large complex systems long before now. See, for example, the great pyramids, the great wall of China, and a few other large and complex systems that existed before the modern era.

Another Definition

This definition is from INCOSE, the International Council on Systems Engineering [INCOSE].

> Systems Engineering is an (1) interdisciplinary approach and means to enable the realization of (2) successful systems. It focuses on defining customer needs and required functionality early in the development cycle, documenting requirements, then proceeding with design synthesis and system validation while considering the (3) complete problem:
>
> - Operations
> - Cost and Schedule
> - Performance
> - Training and Support
> - Test
> - Disposal
> - Manufacturing
>
> Systems Engineering (4) integrates all the disciplines and specialty groups into a team effort forming a structured development process that proceeds from concept to production to operation. Systems Engineering considers both the business and the technical needs of all customers with the goal of providing a quality product that (5) meets the user needs.

Some commentary on this definition. For (1), "interdiscip..." this is one of those words that when I read it I feel as if someone is rubbing chalk across my teeth. Ouch! It is quite painful for me to keep reading.

For (2), well at least we are aiming for success. I have often seen projects that were instead aiming for full employment of all engineers in America.

For (3), once again, we are considering the complete problem. I think that is a good objective. I often, however, become lost while trying to understand how to do this in real life.

For (4), somehow a "discipline" is going to combine different groups of people into a team. I don't know how a discipline can do that. I think people who are organized, caring, and disciplined may be able to do that.

For (5), I like this part. Let's try to meet the needs of the user. That is why we try to build systems. Why do we have to wait until the end of the half-page definition to reach the good part?

A Non-Definition

The previous two definitions were typical of what I found while trying to describe systems engineering. I was about to quit my search, when I found something else. Richard Thayer, whom I had met in a seminar years earlier, wrote a paper on software systems engineering [Thayer]. Thayer explained systems engineering in terms of what a systems engineer does as opposed to what other people do.

For example, the project manager does:

- planning
- organizing
- staffing
- directing
- controlling

The systems engineer may help with some of the project planning, but he merely helps – he is not responsible for it. In a similar vein, the systems engineer may help organize and direct the project, but only as a helper.

Thayer continued his approach by contrasting the systems engineer's job against the jobs of other engineers.

The systems engineer does not design software. That is the job of the software engineer.

The systems engineer does not design circuitry. That is the job of the hardware engineer.

The systems engineer does not write computer programs. That is the job of the programmer.

The systems engineer doesn't test the system. That is the job of the tester.

I could continue on with all the important and necessary tasks that people do on a project that the systems engineer doesn't do.

So, what does the systems engineer do?

The systems engineer ensures that the product satisfies the customer[1].

I once said this while leading a seminar at a conference. One attendee objected vehemently, "It is every one's job to satisfy the customer! I want to hear you say that that is everyone's job!"

I sheepishly agreed. Yes, each person working on the project strives to contribute in a way that satisfies the customer. What I want to emphasize is that the hardware engineer designs a bridge or engine or circuit that works well. The systems engineer ensures that the hardware engineer designs the right bridge or engine or circuit. There is a difference between the two tasks.

The Ten-Pound Pamphlet

To illustrate the value of having a person responsible for satisfying a customer, allow me to tell a story of a project where we didn't have a systems engineer. I could tell many such stories as I have worked on many such projects – none of them ended well.

[1] I use the word "customer" in this definition. Builders and marketers of products differentiate groups of people with words like "client," "user," and "customer." The client provides the money for the project that builds the product. The customer represents all those who will receive the product. The user actually uses the product. These distinctions are important in some contexts. In this book, however, I shall group these and describe them with the word "customer."

The goal on this project was to deliver a career guide. The guide would show a progression from entry-level to expert in all the occupations in my organization. New employees could use this guide as an aid in their career decisions.

We did everything right on this project. Several people from each occupation gathered and discussed their careers, what they did, what they wished they knew when, and what they believed people at each level in an occupation should be able to do. We employed facilitators to help us surface and document what we knew.

We *documented* what we knew. We created page after page of tables with tiny print in rows and columns. We explained everything in painful detail.

The result was well organized. A person could take the pages, find the right table, the right row, the right column, and make little check marks next to the classes, characteristics, and capabilities. Once all the little boxes were filled with check marks, the person would have completed an excellent career.

There was one problem with this career guidebook – it filled a large, ten-pound, three-ring binder. No one ever used the career guidebook. The customers looked at it once and walked away never to open it.

We did each individual task correctly. The employee groups shared information well, the facilitators helped these groups share, the writers wrote good prose, the publisher made appealing pages, the binders were attractive and had ample capacity. No one, however, kept checking with the customer to ensure the career guidebook would be satisfying. No one did the systems engineer's task.

What the customer – the employees – wanted was a little one-page, folded pamphlet with brief summaries and pointers to other information.

For lack of a systems engineer, the career guide project was a dismal failure that wasted thousands of person hours of effort and who knows how much money.

A Little More Definition

Now that I have written the one-and-only statement that defines the subject in easy-to-understand language, allow me to supplement the definition. I will do this with a story I once heard about farmers and potatoes.

A Lazy Potato Farmer

There once were several potato farmers. One of the big tasks for the farmers was to sort their potatoes. You see, once the farmers arrived at the potato market with their potatoes, the potato buyers would buy potatoes by size. The big potatoes garnered one price per pound, the not-so-big potatoes another price per pound, and finally the small potatoes another price per pound.

Because the potato farmers wanted to be ready for the buyers at the market, they judiciously sorted their potatoes on their farms. All the picked and cleaned potatoes were laid in front of the farmer and the farm hands. They worked hard to quickly separate the potatoes into three piles – big, not-so-big, and small. They then loaded the potatoes by size into the farmer’s truck for transport into town and the waiting potato buyers. They first loaded the small potatoes, then piled the not-so-big potatoes on top of them, and finally the big potatoes on top of the pile.

All the farmers did the sorting by size, except one. He seemed to be a lazy farmer because he didn't sort his potatoes by size. He and his helpers just loaded all the potatoes into the farmer's truck and he drove to town with his unsorted potatoes.

As if by magic, when the lazy farmer arrived in town with his truckload of potatoes, the potatoes were sorted by size – just like those of the farmer's who worked so hard to sort the potatoes before loading them onto their trucks.

One day, the other farmers discovered that the one farmer was not sorting his potatoes. He was skipping a necessary step, but everything worked. Someone asked the farmer how his potatoes were sorted by size even though he and his farm hands did no sorting.

"Simple," replied the wise (not lazy) farmer, "I drive to town on a bumpy road."

The vibrations of the bumpy road caused the larger potatoes to bubble to the top and the smaller potatoes to settle to the bottom of the truck. The ride to town sorted the potatoes, so the farmer didn't have to.

This farmer was a systems engineer. He performed another important systems engineering task:

The systems engineer examines the entire system and applies a little wisdom.

At first glance, bringing potatoes to market has these steps:

(1) dig the potatoes

(2) clean the potatoes

(3) sort the potatoes

(4) load the potatoes on the truck (by size)

(5) drive to town

(6) sell the potatoes by size

Upon further examination, bringing potatoes to market has these steps (notice step 5):

(1) dig the potatoes

(2) clean the potatoes

(3) sort the potatoes

(4) load the potatoes on the truck (by size)

(5) drive to town (*wherein the vibrations of the road sorts the potatoes by size*)

(6) sell the potatoes by size

The systems engineer farmer noticed that steps (3) and (5) were redundant. He applied some wisdom and eliminated step (3). Hence, this farmer did these steps:

(1) dig the potatoes

(2) clean the potatoes

(4) load the potatoes on the truck

(5) drive to town (wherein the vibrations of the road sorts the potatoes by size)

(6) sell the potatoes by size

Therefore, while ensuring that the system (sorted potatoes brought to market) satisfies the customer (the potato buyer), the systems engineer examines the entire system and applies a little wisdom.

The not-to-do List

The systems engineer farmer didn't do one costly and time-consuming step. He understood that he and his helpers could have sorted the potatoes before loading them on the truck, but he chose not to do that. While the other farmers had a solid "to do" list, this farmer had a not-to-do list.

There are many tasks that an engineering team can perform while building a system. There are many features that a team can put into a system that they are building. Not all these tasks and features are necessary for satisfying the customer. Some tasks are just what I call "jobs programs for engineers." They are nice tasks; they add some value; they give the engineers something to do, but they are not necessary.

Many of these unnecessary tasks greatly increase the cost of the system. People make mistakes at some rate (one mistake per X hours of work). When engineers do more work on a product, they make more mistakes. Finding and correcting these mistakes is one of the most costly activities on any project. Therefore, to increase the cost of a product, add unnecessary work. To reduce time and cost, use a not-to-do list like the potato farmer.

A good systems engineer will have a not-to-do list that shows what tasks he won't do and what features he won't put into this system. Some of the tasks that I have on my not-to-do lists are:

- measure how many widgets we make during the morning hours of the day
- measure how many widgets we make during each of the four seasons of the year
- correlate broken widgets to when the widgets were made (morning summer, non-morning fall, etc.)
- gather 50 people in a one room at one time to offer suggestions of what we should measure

I have a lot of measuring tasks on my not-to-do list. One reason is that most of the systems I have built were low-volume systems, i.e. we made ten systems total. If we were making and then repairing ten million systems, the measures mentioned above might be good to put on the "to do" list.

Some of the features I have put on not-to-do lists include:

- shorten the time needed to use the system by one hour
- shorten the time needed to learn how to use the system by one hour

These items appeared on my not-to-do lists because of simple return-on-investment calculations. The customers used the system two times a year for a total of eight hours. The two features on the list would have increased the cost of building the system by $1 Million. Since the customers were paid $20 per hour, we saved $80 a year, so the $1 Million in added cost to build would be paid back in... well, you see the point.

These example tasks and features went on my not-to-do lists because they didn't make sense in the context of my projects and systems. That is one way in which the systems engineer examines the entire system, he considers the context of the project and system and applies a little wisdom.

Common Sense and Common Practice

At this point in the chapter I can hear my mother, "but this is just common sense." (My mother often chided me about common sense.) Of course you ensure that the system satisfies the customer. Of course you look at the whole thing and be as wise as you can. Who doesn't know that? Who doesn't do that?

Most people I have seen in 20-something years working on systems know these things. It is unfortunate that most people I have seen during that time don't do these things. Systems engineering is one of those things that is common sense, but not common practice.

One reason systems engineering is not commonly practiced is that people assume that someone else must be busy satisfying the customer and applying wisdom. Some manager *must* be doing these tasks.

This "the manager must be doing this" is one of the problems with systems engineering. "Real engineers" push the systems engineering task to managers – those people who aren't smart enough to be real engineers. This may be true in some cases, but in those rare projects where I saw a competent person assigned the systems engineering role, that person was challenged, worked hard, and the entire project and all the "real engineers" benefited.

Systems engineering doesn't apply to all projects. I believe that all projects *can benefit* from the practices of systems engineering. I also believe that there are projects where concentrating on systems engineering *doesn't* make economic sense.

Here are my rules of thumb on when to make common sense common practice.

(1) Don't exert much systems engineering when one or two people can carry all the technical and task details in their heads.

(2) Use systems engineering when the system and project are bigger than any two people.

I haven't worked on many projects that fall into the first category. The vast majority of projects are too complex, too difficult, and too big to carry in the heads of the two smartest people on the project. This wasn't the impression that we – especially the two smartest people – had at the time. Later, however, and often years later, we learned that we were wrong. The maintenance mess and expense sitting in our laps showed all the issues that we didn't see and didn't solve.

Closing Thoughts

I believe I now understand what a systems engineer does. I express the systems engineer's tasks as

The systems engineer ensures that the product satisfies the customer.

and

The systems engineer examines the entire system and applies a little wisdom.

I needed many years of puzzled looks to arrive at these two statements. Perhaps this book will allow you to understand systems engineering without all the years of frustrated longing; that is one of my hopes.

Now that I think I understand systems engineering, I just have to worry about when and how much to apply it. The "when" is starting to be clear in my mind:

Use systems engineering when the system and project are bigger than any two people.

If the two smartest people in the room have any doubts, use systems engineering.

The remainder of this book contains my current thoughts on "how much" systems engineering to use. I describe what I believe is enough systems engineering for the vast majority of systems that people build.

Please use the techniques herein with thought and care; thought for the system at hand and care for the people.

Suggested Exercises

1. What are some differences between the systems engineer and the customer?
2. Given the two short definitions in this chapter, have you ever been a systems engineer? What did you do?
3. Consider any system that you participated in building. Who was responsible for satisfying the customer? What did they do?
4. Consider any system that you participated in building. Who was responsible for examining the entire system and applying wisdom? What did they do?
5. Create a not-to-do list of tasks that you won't do today. Create a not-to-do list of tasks that you won't do while building your next product.
6. At this point in the text, what do you think are some of the tasks a systems engineer does?

References

[IEEE] Aerospace and Electronic Systems Magazine, IEEE, October 2000.

[INCOSE] http://www.incose.org/practice/whatissystemseng.aspx. INCOSE is the International Council On Systems Engineering.

[Thayer] "Software System Engineering: A Tutorial," Richard H. Thayer, Computer, April 2002, pp. 68-73.

Where We are Now

We are still at the beginning of this book. Now we have a few definitions and guidelines to use in systems engineering. These will serve as themes or foundations for the rest of the book.

Section 2

Systems Engineering Techniques

The first chapter defined what the systems engineer does in two statements:

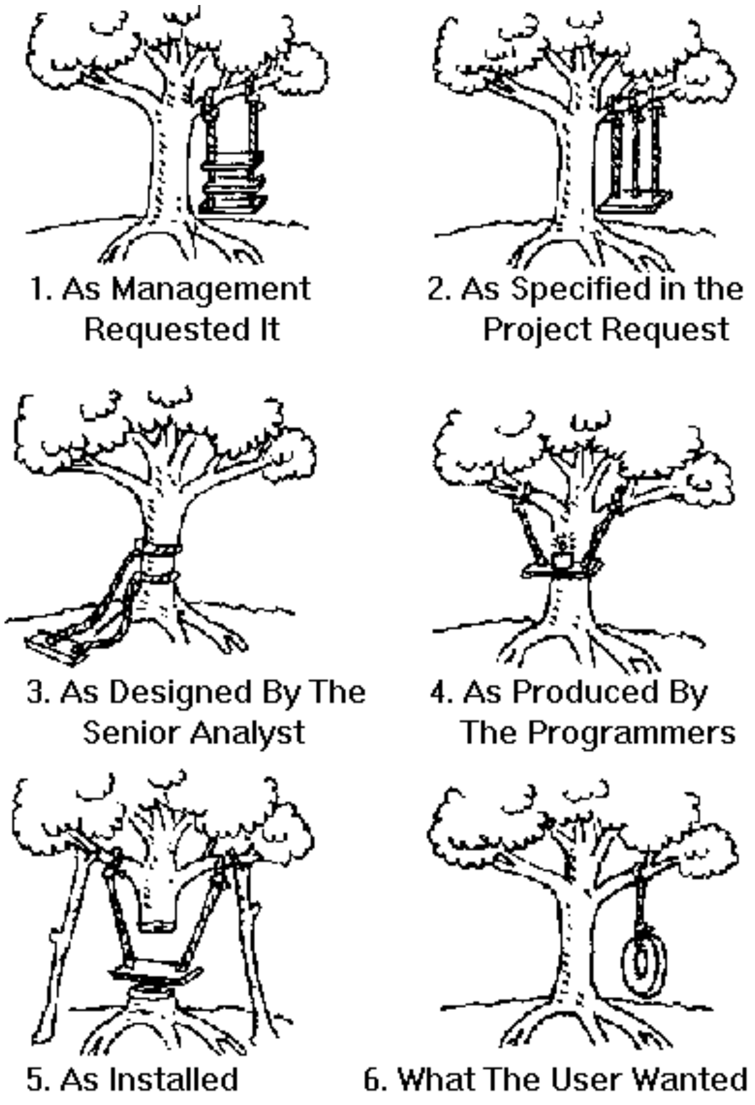
(1) The systems engineer ensures that the product satisfies the customer.

(2) The systems engineer examines the entire system and applies a little wisdom.

In this section, we look at techniques that help the systems engineer accomplish these two over-arching tasks.

2 The Requirements

They Want What?



1. As Management Requested It

2. As Specified in the Project Request

3. As Designed By The Senior Analyst

4. As Produced By The Programmers

5. As Installed

6. What The User Wanted

Early in my career, I often saw one version or another of this tire-swing cartoon. I hated it. I wondered how competent people would mess up so badly. The cartoon had to be a silly joke that wasn't based on any reality.

I was wrong. The tire-swing cartoons are based on reality. I've contributed to the lore of building systems that didn't come close to satisfying the customer. I've built systems without any systems engineering.

To satisfy the customer, the systems engineer needs to know what system to build. The systems engineer needs to understand the requirements for the system.

Great Books on Requirements

This isn't a book on requirements. I would like to present everything the systems engineer needs to know about requirements work in ten pages. I don't know how to do that and trying would only frustrate both the reader and me.

Instead, I will mention three great books on requirements. Given the time, please look at them. These books will make the task of systems engineering much easier. I write that confidently as they have greatly aided me.

"Exploring Requirements: Quality Before Design" by Donald C. Gause and Gerald M. Weinberg (Dorset House Publishing, 1989).

The authors emphasize people in this book. The systems engineer (a person) needs to talk with the users (also persons) to work with requirements. Conversations among persons, however, are fraught with trouble. Hence, the authors provide excellent background on these perils and how to work through them.

I especially like the book's chapter on context-free questions – those questions that the systems engineer can ask regardless of the context of the system.

The context-free questions about the product (the system) include:

- What problems does this product solve?
- What problems could this product create?
- What environment is this product likely to encounter?
- What kind of precision is required or desired in the product?

The authors also include questions about questions. These include:

- Am I asking too many questions?
- Do my questions seem relevant?
- Are you the right person to answer these questions?
- Are your answers official?
- May I write down your answers and give you a written copy to study and approve?
- Is there anything else I should be asking you?
- Is there anything you would like to ask me?
- May I return or call you with more questions later, in case I don't cover everything this time?

"Mastering the Requirements Process" 2nd edition by Suzanne and James Robertson (Pearson Education, 2006).

This is the best book on requirements that I have read. The Robertson's have developed a thorough method of learning, identifying, and managing requirements.

I especially appreciate the Robertson's method of identifying the requirements on cards. Each card holds the necessary information for a requirement. Cards can be excellent in some cases. In others, I suggest using a computer as the stack of cards can become unwieldy. The card concept, however, can serve the systems engineer well when entering the requirements into a database.

"Just Enough Requirements Management: Where Software Development Meets Marketing" by Alan M. Davis (Dorset House Publishing, 2005).

Davis describes his journey in the field of requirements and documenting requirements. He started 30 years ago advocating good requirements *documents*. He then advocated better *documents* then even better *documents*. Finally, in this book, Davis recommends a short *list* of requirements. The list will best address the customer's needs.

Davis provides a full description of requirements and working in the requirements industry. His emphasis comes back to the customer. He emphasizes "just enough" of everything – an attitude that partly inspired my text on systems engineering.

If you don't have a lot of time to study requirements, spend it reading Davis' book.

Learning the Requirements

The first, never-ending step in requirements work is learning the requirements. Some people call this exploration, others call it elicitation, others call it interviewing. I use the term "learning" as I feel the systems engineer needs the ability to learn from people and situations.

Learning the requirements never ends. I have built systems where I learned of new requirements on the day we delivered the system. It is easy for me to say that the customers were obstinate, immature, lazy, and a bunch of other things, but they were still the customers and the source of requirements. They may have forgotten a requirement, may have forgotten to tell me something, or may have just learned something about what the system should be. Whatever the reason, customers may have new requirements on any day of the project. The systems engineer should be ready to learn of these at all times.

Requirements come in two basic flavors: (1) functional and (2) non-functional. To obtain these two basic items fill in the blanks for these two statements:

(1) The system shall __________ (do something).

(2) The system shall be __________ (something).

The first statement tells the systems engineer what the system shall do; what functions it shall perform.

Examples include: The system shall water plants. The system shall provide electric power to appliances.

The second statement tells the systems engineer what the system shall be; what attributes it shall have.

Examples include: The system shall be large enough for 10,000 acres of tomatoes. The system shall be small and light enough for an elderly woman to use. The system shall be suitable for a portable hair dryer. The system shall be powerful enough for a drill that can punch one-inch holes in reinforced concrete.

The most important requirements for a system come from people.

The importance of people in learning requirements hasn't always been easy for me to accept. I often wanted to study some books and papers to learn about a system. That doesn't work. Instead, talk with people. Gause and Weinberg's book is an excellent source of techniques and advice on how to talk with people about what they want.

When I learn a requirement from a person, I need to record the person's name. The techniques taught in [Robertson] include a prominent place for the source of the requirement – the person's name. At some later date, the systems engineer, other engineers, and other people building the system will debate the meaning of a requirement. The only person who really knows what the requirement means is the person who provided it. We should discuss the meaning of the requirement with that person. Without the person's name, I won't be able to conclude the discussion and build a system that satisfies the customer.

When learning requirements from persons, have them tell stories. Stories supply valuable information. The most valuable is that while telling stories, people will reveal their emotions, and their emotions will reveal what is significant to them.

I recently spoke with a customer about how he performed his job (his job was interviewing people). This customer "lapsed" into telling me some of his favorite stories. Instead of yawning, I inched forward in my chair and encouraged him. In the midst of his stories, I asked if he had any video tapes of his interviews. I would love to see some of them. He stopped his stories, exhaled loudly, and groaned a long, "No! We do interviews in hotel rooms, and there isn't enough space for a video camera, and it would be too obtrusive, and I don't know how to set up a camera," and on and on about the disappointments of not having video and audio recordings of the interviews and how much they would help him review his interviews and help train new people and other helpful things.

That was it. The ability to record the audio and video of interviews was one of the most important requirements for the new interview system. No one had mentioned that before because they were sure it was impossible. We have many challenges facing us in implementing this recording system, and I don't know if we will be able to overcome them. I do, however, know that recording is significant. I learned that by listening to stories and noticing emotion in the stories.

Another technique in learning requirements from people is to ask why – five times. For example, in helping to rearrange an education center's rooms, the "why" conversation was:

Customer: I wish I had movable walls in here.

Systems Engineer: (#1 why) Why do you want movable walls?

Customer: Movable walls would let me divide this room into smaller classes and still have a room large enough for parties.

Systems Engineer: (#2 why) Why do you want smaller classes?

Customer: That would allows students at different levels to be in groups with others of their own level.

Systems Engineer: (#3 why) Why do you want students to be with others of their own level?

Customer: Then the students wouldn't be so bored in class.

Systems Engineer: (#4 why) Why do you want the students to not be bored?

Customer: Then they wouldn't complain about Mr. Smith the instructor.

Systems Engineer: (#5 why) Why do you want the students to stop complaining about Mr. Smith?

Customer: Because Mr. Smith *is* boring, and when they complain it makes me realize that I should fire Mr. Smith.

Aha! Instead of building a system with movable walls, I now see that we need a system that will remove Mr. Smith, teach Mr. Smith how to not be so boring, introduce more instructors to the facility that are not boring, or do all these things. The constant "why" questions helped me learn the real requirement. If I had taken the "movable walls" as the one and only requirement, I would have built a system that did not satisfy the customer.

Identifying the Requirements

The next, and again a never-ending, step in requirements work is to identify the requirements. I use the term "identify." Other terms that may apply include writing, documenting, storing, and recording. Each of those latter terms infers a technology or specific method of keeping the requirements. I like the term "identify" because it reminds me of a systems engineer pointing across a room and saying, "Look there! Those are the requirements we have identified (so far)."

There are several methods available to keep the identified requirements. For many years, the 1980s through the mid-1990s for me personally, systems engineers identified requirements in books or documents. The classic was the SRD (System Requirements Document). We also had a Hardware Requirements Document and a Software Requirements Document (more about flow down of requirements in a later chapter).

These documents were thick, heavy, impossible to understand, and no one read them. The requirements were identified, but only the favorite ones were implemented. Customers weren't satisfied often.

In the mid-1990s the state of the art of identifying requirements employed a sophisticated relational database. The systems engineering team would type the requirements into that database. The database technology allowed for sorting, linking, printing in nice documents, and other tasks where computer technology excels.

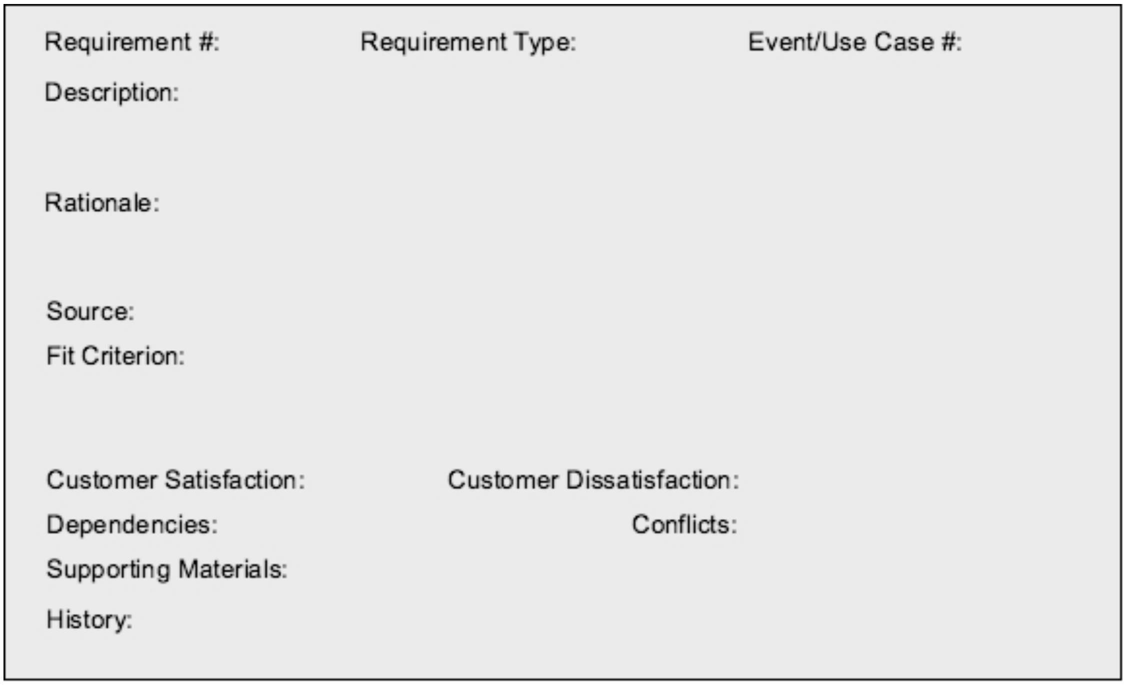
The relational databases were wonderful. They were logical, easy to interface, never forgot anything – in short, they behaved like the perfect systems engineer. We loved them so much that we soon were spending the majority of our time with them.

Well, as the previous section asserted, requirements come from people. Systems engineering teams were concentrating so much on the databases that they forgot about the people. The customers were ignored and not satisfied.

A reaction to the thick documents and relational databases are the agile methods of building systems. These methods emphasize working with people in short iterations. A favorite technique of identifying requirements is on 3x5 cards pinned to a visible wall. I personally like the cards-on-the-wall technique. It brings together people near the wall and increases the conversations among the systems engineers and customers.

The cards-on-the-wall also has its limitations. Rooms have finite wall space, so we have to remove cards to make room for more cards. Remembering what was on the wall is difficult.

I hope that we have arrived at some happy medium. The Robertson's book recommends an excellent method of using cards for initially identifying the requirements (see Figure 1). Their method uses large cards that have enough room to keep many aspects of each requirement (please see their book as I cannot do it justice in a few paragraphs). Use the cards to identify each requirement and then enter the information into a computer for later use. The card-first computer-next method keeps the systems engineer working with people and lets the computer do the mundane task of remembering.



Requirement #: Requirement Type: Event/Use Case #:

Description:

Rationale:

Source:

Fit Criterion:

Customer Satisfaction: Customer Dissatisfaction:

Dependencies: Conflicts:

Supporting Materials:

History:

Figure 1 – A Requirements Card from [Robertson]

A systems engineer should take care when identifying requirements. The reason is that the requirements will define the system and drive all the work. A common problem in building systems is that there are too many requirements. Sources of the "too many" requirements include engineers, customers, and our own lives.

I have worked with all sorts of wonderfully creative systems engineers. They have thousands of great ideas in their minds that they want to share while building this particular system for this particular customer. The frequent problem is that their wonderfully creative idea has nothing to do with this customer and this system. Their creative idea is another, extra requirement.

Customers often mention "27-sigma" cases while discussing requirements. "27-sigma" refers to something that is 27 standard deviations from the mean (the Greek letter sigma denotes the standard deviation in statistics work). I don't want to diverge into statistics. Suffice it to say that a 27-sigma case occurs about as often as a total eclipse of the sun.

Focusing large amounts of effort on a 27-sigma requirement is not wise, and recall that a primary duty in systems engineering is to apply wisdom. Concentrate on the requirements that the customer will use daily. Once (if) those are working in the system, delve into some of the rare cases that may help the customer.

It isn't surprising that engineers and customers create extra requirements. Most requirements are extras – they are not essential to the form and function of a system that will satisfy the customer. This is true of much of what we do in our lives. I have three pair of blue jeans and since I can only wear one at a time, two of them are extras. I have hundreds of books on my book shelves. Since I can only read one at time, the hundreds less one are extras.

I digress into blue jeans and books to emphasize something about identifying requirements. Identify each requirement dutifully. Pause to discuss if each requirement is essential.

A technique that has worked for me in discarding extra requirements is a requirements gateway. The gateway is a single place through which each identified requirement must pass. A group of people – systems engineers and customers – examines each requirement candidate and makes a binding decision on if that candidate will be a requirement or be discarded. The requirements gateway is also a place where the "essential-ness" of a requirement is judged.

Managing the Requirements

The final never-ending part of requirements work is managing the requirements. After learning and identifying, the systems engineer has a large, complex, intricate set of requirements. It is easy to forget some requirements, overemphasize others, and ignore some we simply don't like. Some requirements change, some are deleted, and some are added. The set of identified requirements needs close attention and diligent management.

Requirements management involves working closely and continuously with people. Not surprisingly, many of the problems with requirements management come from people trying to work with people.

Learning and identifying requirements are more technical in nature. Technical people gather information, draw diagrams, and analyze them to prepare for design. Systems engineers use all their technical skills and apply themselves whole-heartedly. In contrast, requirements management is a management problem. It involves meetings, reaching agreements face-to-face, and keeping detailed records. Systems engineers often see requirements management as a clerical job.

Regardless of how distasteful requirements management might be to most systems engineers, it must be done. So, how can we ensure that systems engineers do the requirements management work? The answer is simple in principle but difficult to do. The answer is one word – help. Provide real help to systems engineers in the requirements management area. Real help is not threats to "do it right or else." Real help is realizing that requirements management probably won't be done well and providing the systems engineer with resources and direction.

Real help can take several forms. The simplest help is to appoint a full-time requirements manager. This is a person who understands technical issues, likes working with people, and is good at managing things. The requirements manager performs all the face-to-face and detailed record-keeping tasks. This person has the personality and temperament to keep the customer happy and the technical people out of trouble.

Another form of real help is to clearly state responsibilities. Place a poster in the project area that lists who is responsible for what on the project. The responsibilities include who is the systems engineer, the requirements manager, and who represents the customer. This reduces the amount of changing directions to try to satisfy everyone who is only slightly involved with the system.

A third way to help in requirements management is in the area of the requirements gateway. This is the same gateway mentioned earlier that decides which candidate requirements become real requirements. Any proposed change to the requirements should be reviewed by the people who stand in the requirements gateway. The customers change their minds about requirements. There are many good reasons for these changes and many bad reasons for them as well. The people working the requirements gateway decide if the desired requirements change and the reason behind it are good or bad. They implement change control – a fundamental part of systems engineering. This is where the systems engineer examines the entire system and applies some wisdom. This is not an easy task and one that must not be hurried.

Change costs money. Most people with whom I have worked don't like to accept that. Any requested change needs to be discussed with the cost in mind. The cost of making the change includes the cost of identifying the change, implementing the change, testing the change, managing the changed set of requirements, and maintaining the changed system. The cost of all these tasks accumulates. Sometimes the change is necessary even given all these costs. Many times, however, requested changes are not justified. Most times unjustified changes are accepted because the systems engineers fail to show everyone all the costs. Explaining how someone's great idea is too costly to use does not increase the systems engineer's popularity. Being popular, however, is not one of the systems engineer's duties.

Closing Thoughts

I cannot overemphasize the importance of requirements to the systems engineer. The systems engineer's job begins with asking the customers what they want – the requirements – and ends with handing the customer a satisfying system – one that meets the requirements.

Requirements work can be described in three never-ending and often cycling parts:

Learning the requirements,

identifying the requirements, and

managing the requirements.

As important as the requirements are, they are less important than the people describing and working them. Listing the requirements and checking them off one at a time is a good thing for the systems engineer to do. It doesn't, however, always satisfy the customer. Sometimes being that thorough and methodical upsets the customer.

Wisely apply requirements techniques and stay with the customer – the ultimate judge of systems engineering.

Suggested Exercises

1. List three additional context-free questions you could ask.
2. Write three of the more information-yielding questions you have asked. How have they worked?
3. Consider a requirement that you are working on at this time. Try to attach a name or a face to it. What did you feel while doing that?
4. Think of a customer's story that led you to an important requirement. How would the system you are building be different if you didn't know that story?
5. Think of a requirement that you worked (like movable walls) that turned out to be a person requirement (like a boring teacher like Mr. Smith). What did you do with that requirement? Did you approach the person? Why or why not?
6. Consider a low-tech device (like 3x5 cards) that helped you with requirements. Consider a low-tech device that only brought problems with requirements. What was the difference between these two devices? Do the same exercise considering high-tech devices (like a relational database).
7. Consider an engineer or technician you know who likes to work face-to-face with people. How did that person do when working with requirements? Are that person's personality traits worth emulating?

References:

[Gause] "Exploring Requirements: Quality Before Design," Donald C. Gause and Gerald M. Weinberg, Dorset House Publishing, 1989.

[Robertson] "Mastering the Requirements Process," Suzanne and James Robertson, Pearson Education, 2006.

[Davis] "Just Enough Requirements Management: Where Software Development Meets Marketing," Alan M. Davis, Dorset House Publishing, 2005.

Where We are Now

At this point, the systems engineer has the customer's requirements. We know what the customer wants, i.e. what will satisfy him (or do we?).

Now the systems engineer uses these requirements to build a system. It is wise to first build the system on paper – with a design.

A powerful design tool is an architecture. The next chapter discusses architecture and how the systems engineer and customer can use architecture to refine the requirements and build a satisfying system.

3 Architecture

or what are these leftover parts for?

I know the feeling. Take it apart, fix it, clean it, put it back together, and there are leftover parts sitting on the counter. Why can't there be a sign inside the toaster "Little Obscure Parts Go Here"? It should be obvious that there are places for parts and parts for places.

An architecture of a system is like that. It shows where parts fit and which parts will fit in the system.

Once again, let's begin with a definition (from http://dictionary.com):

architecture –noun

1. the profession of designing buildings, open areas, communities, and other artificial constructions and environments, usually with some regard to aesthetic effect. Architecture often

includes design or selection of furnishings and decorations, supervision of construction work, and the examination, restoration, or remodeling of existing buildings.

2. the character or style of building: the architecture of Paris; Romanesque architecture.
3. the action or process of building; construction.
4. the result or product of architectural work, as a building.
5. buildings collectively.
6. a fundamental underlying design of computer hardware, software, or both.
7. the structure of anything: the architecture of a novel.

The first definition that has always come to my mind is not the first given. I think of architecture as a field of study and work. Architects design buildings and cities. Architecture is the form of those buildings and cities.

In relation to a systems engineer, I go to definition 7 – the structure. The architecture of a system is the structure, the form, the skeleton of the system.

While all systems have architectures, deliberate architectures work much better than accidental ones. Deliberate architectures, those we create on purpose with a purpose, permit the systems engineer to place specific parts into the structure of the system. The systems engineer ensures that the parts fit in the system.

When the parts fit, there are no leftover parts.

An Architecture

Figure 1 shows an architecture. This isn't very impressive (nothing like the National Cathedral), and it surely isn't very complicated (nothing like the National Cathedral). Architectures don't have to be impressive or complex. They are tools the systems engineer can use to apply wisdom, and simple tools are often easier to use.

The architecture of Figure 1 shows the outline or structure of a system. The system will have three sections – input, process, and output. This structure or outline will allow the systems engineer to place parts into the sections.

The input-process-output architecture of Figure 1 is also a type of an architecture. There are many systems that use this architecture type. Examples include:

- data processing
- word processor (Microsoft Word)
- food processor (blender)
- candy manufacturing plant (Snickers)
- hospital
- high school
- health club

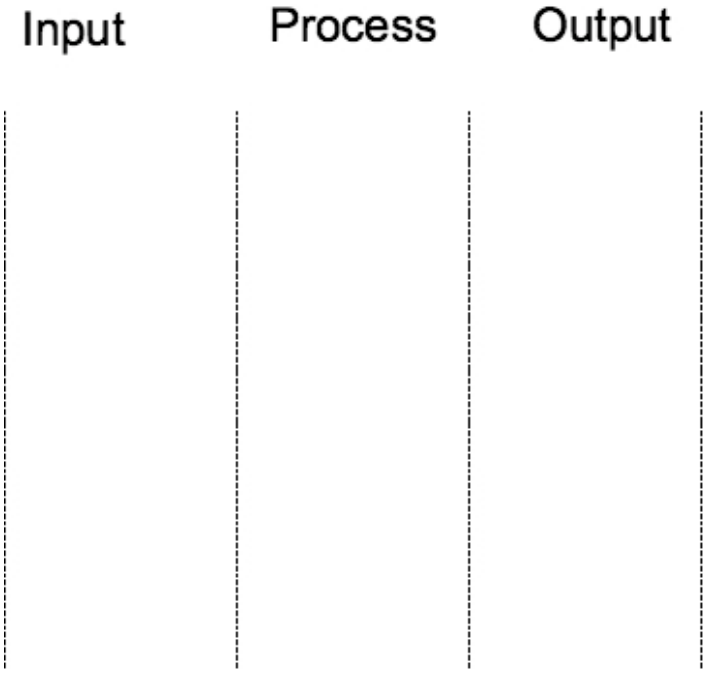


Figure 1 – The Input-Process-Output Architecture

Each of those systems has a method to input something, to process the input, and to send out the processed input.

There are many architecture types like the input-process-output architecture of Figure 1. Christopher Alexander's book [Alexander] is an excellent source of architecture types. His book presents architecture types that have been used many times in homes and communities. I recommend systems engineers have a copy of "The Timeless Way of Building." It doesn't matter if your field is data processing or candy manufacturing; the concept of architecture types and reusing architectures is a valuable one for systems engineers.

Let's narrow the focus of this architecture a little. Figure 2 shows the architecture for a system of storing food in a home. The architecture in Figure 2 comes from a customer's basic requirement: "I need a system that shall store food in my home." I have such a system in my home; my wife calls it a pantry. Yes, something as simple as a pantry has an architecture, and a systems engineer can apply wisdom to it.

The input section of the architecture holds the means by which we put food items into the pantry. The process section of the architecture relates to how the food items are stored (stacking, rotating, presenting). The output section of the architecture concerns how food items are removed from the pantry.

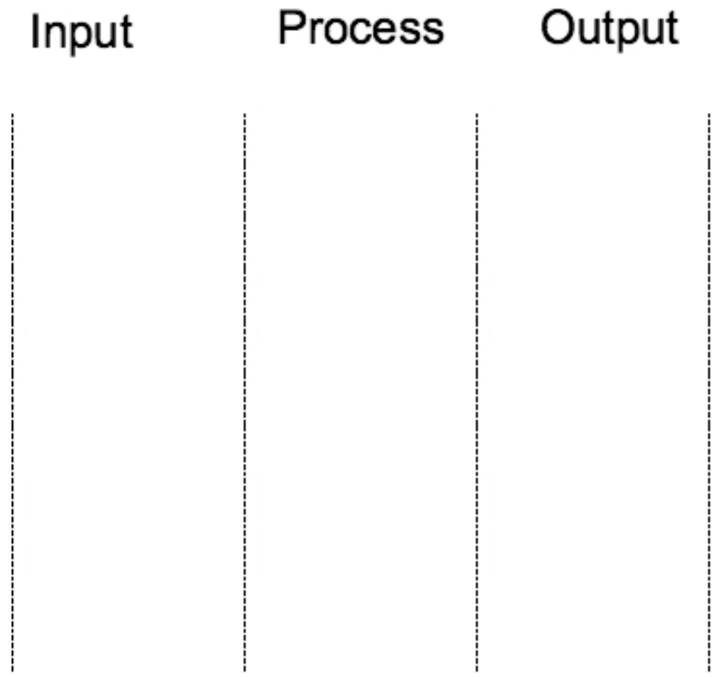


Figure 2 – The Home Food Storage Architecture

Now we have an architecture for a home food storage system. It still is neither impressive nor complicated. I hope to show below that it is useful and sometimes powerful. For now, it is simply a structure or outline of a system.

Some Properties of an Architecture

Architectures have several properties that make them useful to the systems engineer. Among these are (1) context, (2) fit, and (3) type. I'll start with context and my own definition:

Context – noun

1. the part of a text or statement that surrounds a particular word or passage
2. the circumstances in which an event occurs, a setting.

An architecture provides a setting that surrounds a system. It gives the circumstances in a simple form and helps the systems engineer answer:

- Where are we?
- What are we doing?
- What is our situation?

If we cannot answer these questions, we have no architecture, no context, and we are clueless. There is an old saying, "a text without a context is a pretext" (a pretext is a lie or false statement). I contend that a system without an architecture is a false system.

An architecture provides fit for the parts of a system. The concept of "fit" is paramount to me. I have always wanted to know where items fit in the greater scheme of things. Items could be persons (like me, a middle child), processes (why do I have to fill out this form that is exactly like the one I filled out five times already today?), emotions (does denial come before anger or after?), and physical things (why is it we need a love seat in the den?).

An architecture provides the "greater scheme of things." Items the systems engineer chooses can then fit into that scheme. Fit allows the systems engineer to design a system – to select the right item for the architecture.

Finally, an architecture allows the systems engineer to use architecture types. Systems engineering may be a relatively new field of endeavor by its current name. It is not, however, a recent practice. People have been creating and using architectures for centuries. These architectures or outlines for systems can be used and reused by the systems engineer. The systems engineer "just" has to find the type and apply it. Doing so, however, is more difficult than writing that sentence.

Using an Architecture

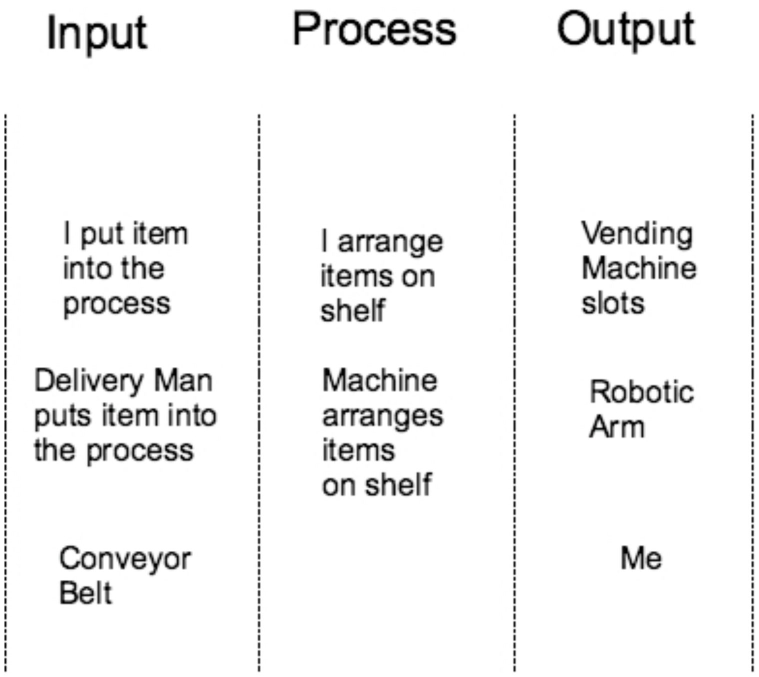
Now let's use the architecture shown in Figures 1 and 2. What the systems engineer does at this point is enter items that fit into the different sections of the architecture. Figure 3 shows an example of this. I have inserted three items that fit into the Input section, two items that fit into the Process section, and three items that fit into the Output section. These eight items show possibilities. Each item can fit in its section, but may not be a wise choice.

Consider the items in the Input section. I have shown three different methods and items that could put food into the food storage system process (commonly known as a pantry). These three items are (1) I could put the food items into the pantry by myself (this is what I do now at my house), (2) a delivery person could put the food items into the pantry (I would like to do this at my house, but those guys cost too much), and (3) I could use a conveyor belt to place the food items into the pantry (as an engineer, that sounds cool, but would my wife like it and how much would that cost?).

Notice how a person doing something can be an item in an architecture. Items in the sections of an architecture don't have to be a machine or a computer. This is an important concept for the systems engineer:

People are parts of systems, too.

Many systems engineers find themselves in all sorts of trouble by forgetting this simple concept; I have.



Input	Process	Output
I put item into the process	I arrange items on shelf	Vending Machine slots
Delivery Man puts item into the process	Machine arranges items on shelf	Robotic Arm
Conveyor Belt		Me

Figure 3 – Design Options for this Architecture

Now let's move to the Process section of the architecture of Figure 3. I have entered two possible items here. These are (1) the items sit on the shelf as shuffled by me, and (2) a machine arranges the items on the shelf. I like (2). I would love to have a machine (a) put all the beans in one part of the pantry and all the fruit in another, (b) ensure all the food items face the front so I can read the labels easily, (c) move the oldest items to the front so I consume them first, and (d) do all sorts of other sorting and presenting for me.

Finally, consider the Output section of the architecture. I have three possible items in this section that sends the food items out of the pantry and into my hands. (1) Vending machine slots could slide the items out to the bottom of the pantry where I grab them. (2) A robotic arm could reach into the pantry, grab the desired food item, and hold it out to my hand. (3) I could just reach in the pantry and grab the item myself. After all, the Process section has arranged the items in an easy-to-see, oldest-item-in-the-front manner.

Notice how each item in each section of the architecture functions properly, i.e. it fits. I could choose any of the three items in the Input section and satisfy the requirements of the section. They will all perform the function required of the Input section. Those three items are different and have their own good and bad characteristics, but they will all work properly.

Now consider how the architecture helps the systems engineer examine the entire system and apply some wisdom. The architecture with the possible items in each section allows the systems engineer to look at the many possible designs of the system. There are 18 paths through the architecture from the Input section through to the Output section (3 Inputs times 2 Process times 3 Outputs = 18 paths). Each of the 18 paths is a design for a system that may satisfy the customer.

Figure 4 shows one path through the architecture. It shows me putting the food items in the pantry, me arranging the items on the pantry, and me taking them out of the pantry. That is the system that exists in my home. That system works properly, i.e. it meets the customer's requirements.

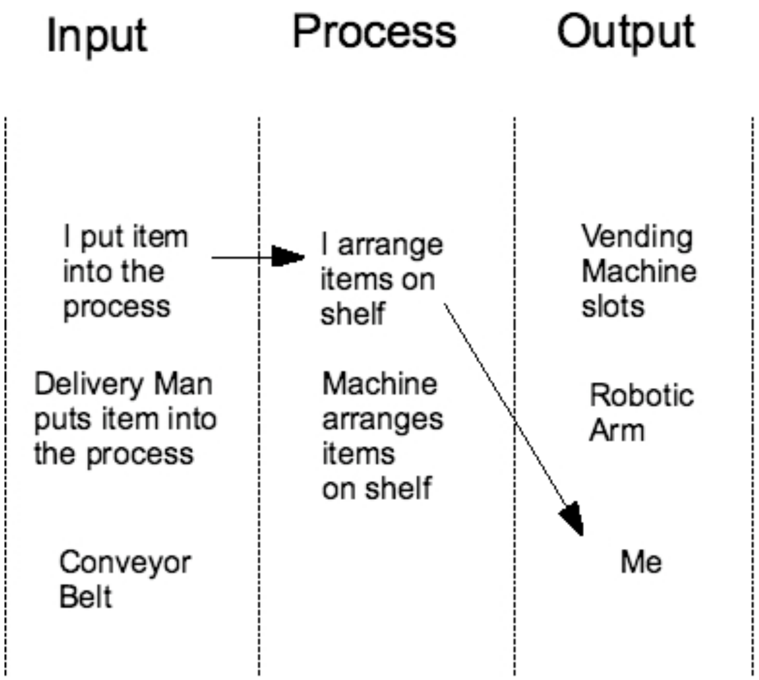

Input
Process
Output
I put item into the process
Delivery Man puts item into the process
Conveyor Belt
I arrange items on shelf
Machine arranges items on shelf
Vending Machine slots
Robotic Arm
Me

Figure 4 – One Path Through the Architecture

I could also build a system using the path through the architecture shown in Figure 5. That would use a conveyor belt to load food items, a machine to arrange the items, and a robotic arm to give the food items to me. I would love to have that system in my home. Well, I think I would love it. I haven’t considered the cost and complexity of that system.

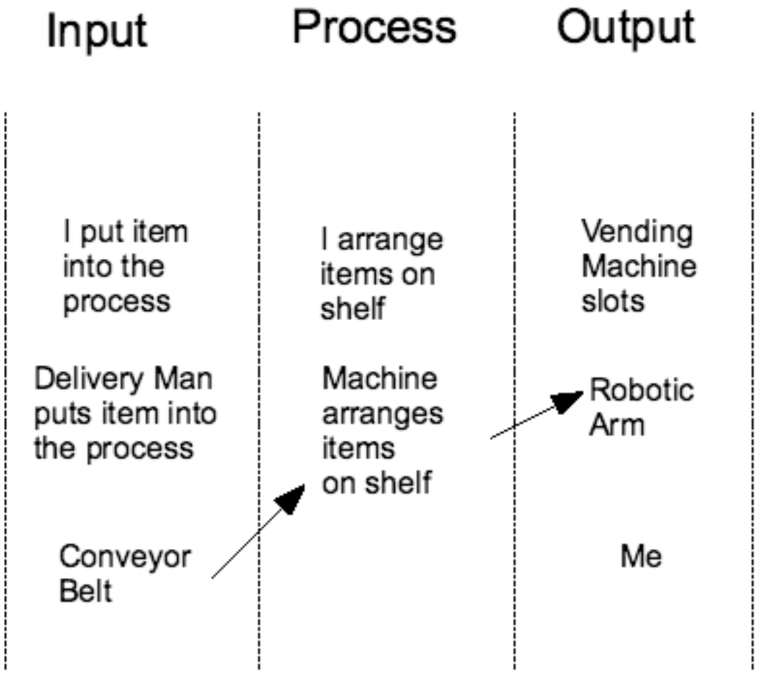

Input
Process
Output
I put item into the process
I arrange items on shelf
Vending Machine slots
Delivery Man puts item into the process
Machine arranges items on shelf
Robotic Arm
Conveyor Belt
Me

Figure 5 – A Second Path Through the Architecture

The architecture in the pantry example shows many possible ways I could build a food storage system. As I think of more possible items for each section, the number of system designs grows. This is one thing an architecture does, it shows the disadvantages of the possible. I can see all the crazy designs for a system. All those designs will satisfy the requirements, but all those designs are not wise. The architecture helps me see these poor choices and avoid them. The time I spend sketching and disregarding designs is minuscule compared to the time I would spend building, dismantling, and rebuilding food storage systems.

A key phrase in the previous paragraph is "I can see." An architecture helps us see what we can do. Seeing the possibilities helps people to disagree, and disagreement is a good thing when designing a system. If everyone quickly agrees on a design, we haven't designed yet. We haven't examined the different paths through the architecture, discussed the good and bad attributes of each path, and debated our preferences.

For example, a vending machine Output would be good for food storage, unless I keep eggs in it. Sliding raw eggs out onto the floor is quick, but dirty.

"But you don't store raw eggs in a pantry, they go in the refrigerator."

"But this is a food storage system, and eggs are food, so we cannot limit this system to a room-temperature pantry."

"But I assumed it was separate from the fridge."

"But I want cold storage in my system."

This conversation surfaces assumptions held by different people; the architecture enables the conversation and allows us to learn. This is true for the customer as well as for the systems engineer. The customer – back when he confidently told us his requirements – may not have considered all these "buts." The architecture spurred more thinking and learning. The systems engineer will probably need to revise the requirements and the architecture.

Putting Numbers into the Architecture

The architecture showed the outline of a system and the many different designs for the system. How can the systems engineer choose one design? One method is using a Technical Performance Measure or TPM.

A Technical Performance Measure is a number that assigns value to a design. Consider the system architecture back in Figure 2. We can assign a Technical Performance Measure to each section of the outline according to the amount of time for that section to perform its function.

How long will we allow the Input section to place the food items into the pantry? Five minutes a week? Five hours a week?

How long will we allow the Process section of the architecture to arrange the food items in the pantry, five or ten minutes a week?

How long will we permit the Output section of the architecture to dispense a requested food item? Five seconds? Ten seconds?

The Technical Performance Measures cause us to examine the desired performance of the system that fits in the architecture. Is the time to operate the important measure for this system? How about the weight of the system? How about the cost of the system? How many people will the system serve? The systems engineer should examine these different aspects of the system.

The Technical Performance Measure comes from asking the customer, "Which of your non-functional requirements will determine your satisfaction with the system?"

Recall that the non-functional requirement is of the form, "The system shall be __________." In this case (time), the non-functional requirement is, "The system shall be fast." The non-functional requirement leads to the determining factor(s) in choosing a path through the architecture.

Let's use the time as an example TPM. The systems engineer estimates a time needed for each option in Figure 3. We can assign the amount of time required for each option. For example, the option "I put item into the process" requires one hour per week to perform its function. The option "Delivery man puts item into the process" requires two hours per week. The option "Conveyor belt" requires 15 minutes per week. We can assign times for all the options in both the Process and Output sections of the architecture.

We can build a spreadsheet that calculates the time required for each of the 18 possible paths through the architecture (the 18 designs). The spreadsheet will show the design that optimizes the TPM of time. The systems engineer has used the architecture, the options in each part of the architecture, and the Technical Performance Measure to design a system. He examined the entire system and applied some wisdom.

Systems in real life are rarely this simple. The systems engineer often has to consider more than one Technical Performance Measure. As discussed above, we can add three more Technical Performance Measures: the weight of the system, the cost of the system, and the number of people per day the system will serve. The systems engineer can assign a value in these three areas of consideration to each of the options in each section of the architecture. The systems engineer then creates spreadsheets that calculate a Technical Performance Measure for each of the 18 paths for each of these three additional areas of consideration.

The systems engineer now may have a design that is best for all four areas of consideration. Sometimes one design is best is all four areas, but that is rare. What usually happens is that two or three of the designs are better in different Technical Performance Measures. Once again, conversations occur between the systems engineer and the other interested parties. These conversations decide which design is the best compromise among all the Technical Performance Measures.

Another Example Architecture

Let's now consider a more complex system – a system for purchasing items for an office. Figure 6 shows the architecture for this system. There are four sections in this architecture: Request, Approve, Purchase, and Deliver.

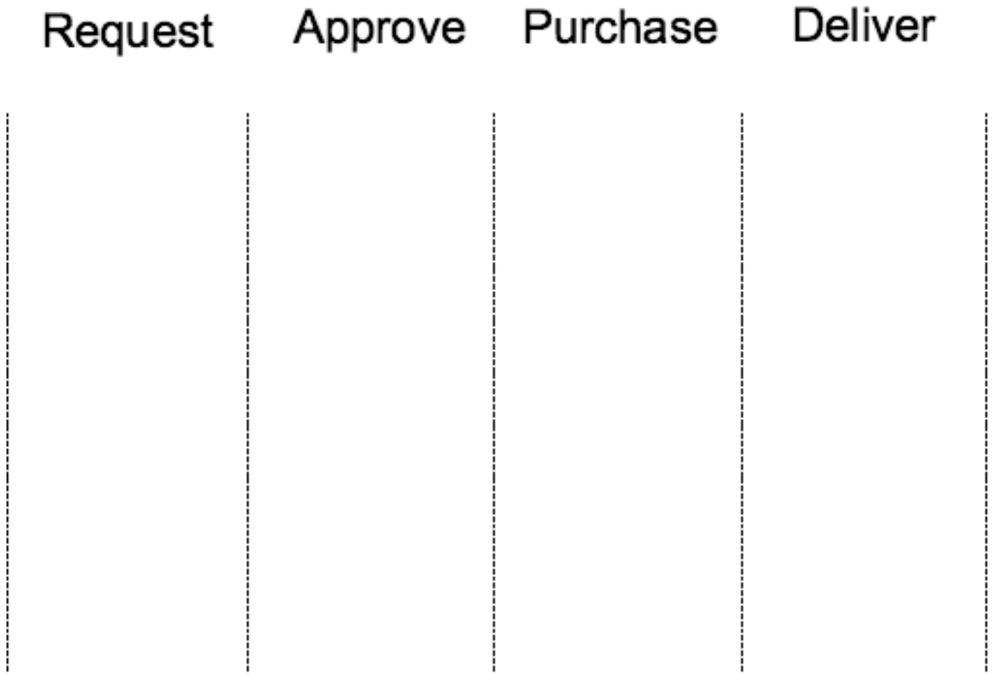
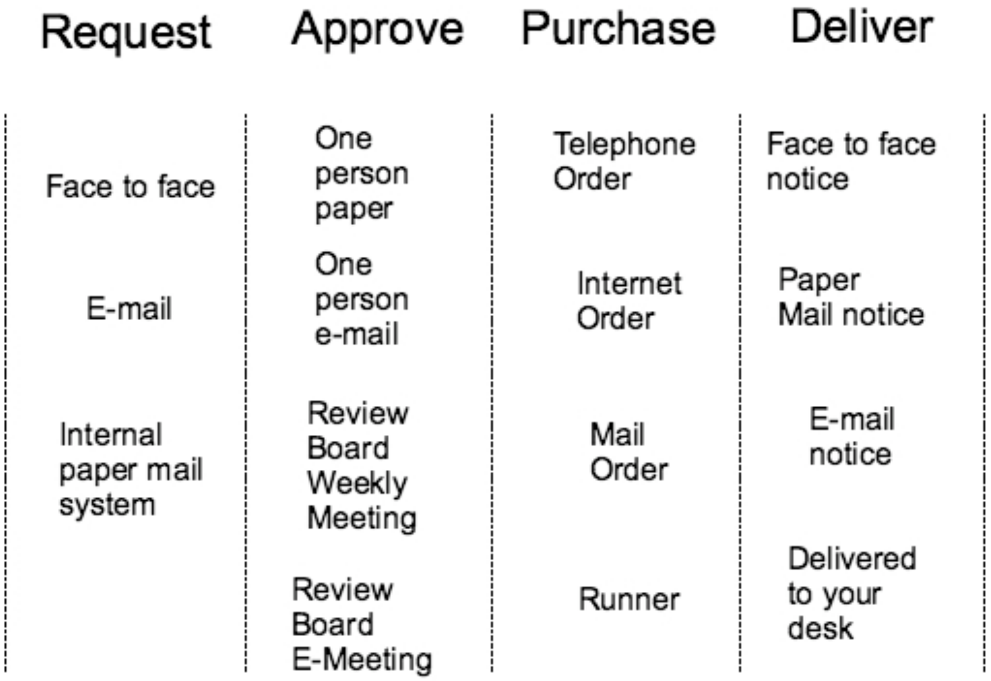


Figure 6 – An Architecture for Purchasing Items

Figure 7 shows the result of placing options into each section of the architecture of Figure 6. There are 192 different paths through Figure 7, and each will satisfy the basic function required of the system.

Let's consider a Technical Performance Measure of time required to obtain an item needed in the office. The systems engineer estimates the time required of each of the options in each of the sections. The systems engineer uses a spreadsheet to calculate the times required by each of the 192 paths through the architecture. Once again, the path through the architecture that uses the least time is that chosen as the design for the system.



Request	Approve	Purchase	Deliver
Face to face	One person paper	Telephone Order	Face to face notice
E-mail	One person e-mail	Internet Order	Paper Mail notice
Internal paper mail system	Review Board Weekly Meeting	Mail Order	E-mail notice
	Review Board E-Meeting	Runner	Delivered to your desk

Figure 7 – Design Options for this Architecture

This is all simple on paper, but far more complicated in real life. The tools described here, however, will serve the systems engineer well.

The architecture shown in Figures 6 and 7 reveal another aspect of Technical Performance Measures – the technical budget. For this example, the systems engineer can create a *budget* for the Technical Performance Measure. Consider setting a time requirement on the total system of five days, i.e. a requested item must be delivered to the requester in five days or less. Five days is the total budget for this Technical Performance Measure.

The systems engineer allocates a portion of the total budget to each section in the architecture. Given there are five days in the budget, a simple method is to allocate each section equal time or 1.2 days. That may make sense, but it probably doesn't. The Request section of the architecture commences after the requesting person knows what he wants. The Request should only take one hour, so the systems engineer allocates one hour of the budget to Request. This leaves four days and seven hours of budget to be allocated to the remaining three sections of the architecture.

After allocating the Technical Performance Measure budget, the systems engineer uses a spreadsheet to examine the 192 paths through the architecture. It is likely that a section exceeds its budgeted part of the Technical Performance Measure. That could mean a failed system, but it is likely that another section doesn't use all its budget. The systems engineer can take budgeted time from one section and give it to another. So long as the total system doesn't exceed the total budget, the systems engineer is able to move allocations of the budget among the sections of the architecture.

A basic Technical Performance Measure and budget allows the systems engineer to examine the entire system and apply some wisdom.

Closing Thoughts

The material in this chapter presents a straightforward process. The systems engineer:

- Draws an architecture
- Puts options in each section
- Creates a Technical Performance Measure
- Assigns values of the Technical Performance Measure to each option in each section
- Uses a spreadsheet to pick the best path through the architecture

Real life working with real people on real systems is rarely so straightforward. Consider the Technical Performance Measure of time for the system of Figure 7. The shortest time to acquire and deliver a requested item is the best. In the Purchase section, it is obvious that using a "Runner," a person who goes to the store to buy the item, is the most time consuming option. That could take a purchasing person hours to do while Purchasing over the Internet would only take minutes.

But, is the important Technical Performance Measure the time required of the purchasing person or the time required to put the item in the requesting person's hands? If a purchasing person went to the local store, the requesting person is likely to have the item in his hands today. Ordering over the Internet is quick for the purchasing person, but it will take several shipping days before the item arrives. Which is more desired?

Also consider the time wasted when the ordered and shipped item is not what the requester wanted. It may be exactly what was requested, but not be what was wanted. People requesting things out of catalogs are often disappointed. Suppose instead that the purchasing person goes to the local store, calls the requesting person on a cell phone, and uses the cell phone's camera to show the requesting person what is in the store.

The requesting person may say, "Oh! Look over to the left. That thing way over there, that is what I really want!"

The systems engineer can now see that some paths through the architecture score poorly with the Technical Performance Measure. They seem to please the customer in some cases, but fail miserably in others.

The course of action is to offer flexibility in the architecture. Choose one path that will serve in 80% of the system's usage. Have another path available for 10% of the usage, and another path for the final 10% of the usage. This is more complicated, but it is more likely to satisfy the customer.

The architecture, the paths through the architecture, the Technical Performance Measure, and the spreadsheet are all tools. They are good tools, but they are merely tools. The systems engineer uses the tools and a lot of wisdom to find a design. That design, however, may not satisfy the customer – the first job of the systems engineer. The design can spur further conversation with the customer to help the customer better understand what he wants.

The wise systems engineer will

(1) use tools

(2) have a conversation

(3) repeat steps (1) and (2) until he and the customer are confident of their understanding of the system requirements.

Suggested Exercises

1. Look around the house. Find a system, draw its architecture as in Figure 2. Fill in some possible items as in Figure 3.
2. Consider a food pantry. Draw an architecture that is different from Figure 2.
3. Think of an architecture type, i.e. an architecture that will work in several different situations. Draw that architecture type. Where could you use it? Think of three usable situations.
4. By hand, trace ten different paths through the architecture of Figure 3. Which path do you like best? Why? Which path do you like least? Why? Which path would be expensive? Which path would be difficult to maintain?
5. In exercise 4, expense and maintainability are two Technical Performance Measures. List three more Technical Performance Measures for the system.
6. For the architectures of Figures 6 and 7, use as a Technical Performance Measure the cost to build the system. Assume a total Technical Performance Measure budget of $10,000. Allocate this $10,000 across the four sections of the architecture. Defend your budget allocation.
7. Starting with Figure 7, what conversations would you like to have with the customer to learn what would satisfy him?

Reference:

[Alexander] "The Timeless Way of Building," Christopher Alexander, Oxford University Press, 1979.

Where We are Now

The previous chapter showed how the systems engineer can use the architecture. The systems engineer draws the architecture outline, denotes the options for items in each section, runs the spreadsheet, chooses the path through the architecture, and there you have it – a designed system.

I did leave out (at least) one step in all that – the dotted lines between the sections. These dotted lines are the interfaces, the things that allow the items in the different sections to connect to one another. Without the interfaces, those items are just interesting things sitting next to one another.

4 Interfaces

The "Goes Inta" and "Goes Outa"

This guy has an obvious problem. The ends of the two electric cords won't connect. He is using an even number of cords instead of an odd number – or it is the other way around? Either way, the system won't work.

Interfaces match the goes inta's and the goes outa's. Interfaces – though tedious – ensure the system connects.

When using an architecture, the systems engineer placed many items into each section. Those items didn't just come from thin air. There must be some source of requirements for each item. There is – and that source of requirements is called an interface.

Once again, let's start with a definition (from http://dictionary.com).

in·ter·face

–noun

1. a surface regarded as the common boundary of two bodies, spaces, or phases.

2. the facts, problems, considerations, theories, practices, etc., shared by two or more disciplines, procedures, or fields of study: the interface between chemistry and physics.

3. a common boundary or interconnection between systems, equipment, concepts, or human beings.

4. communication or interaction: Interface between the parent company and its subsidiaries has never been better.

5. a thing or circumstance that enables separate and sometimes incompatible elements to coordinate effectively: The organization serves as an interface between the state government and the public.

6. Computers.

a. equipment or programs designed to communicate information from one system of computing devices or programs to another.

b. any arrangement for such communication.

In our current context, I like definitions 3., 4., and 5. The interfaces used by the systems engineer (3.) provide a common boundary or separate the sections of the architecture. The interfaces (4.) provide for communication between sections of the architecture. Finally, the interfaces (5.) permit the sections of the architecture to function as a complete system with a new function that differs from the functions provided by each section (the system is greater than the sum of the sections).

Once the systems engineer has an architecture for the system – the sections – he defines what is between those sections. These are the goes inta's and the goes outa's of the sections. If each item satisfies the interface of a section, I can use the item. The item fits within the architecture. This gives me choice, and choice makes it much easier to satisfy the customer.

Defining and maintaining the definition of the interfaces in a system are two key tasks for the systems engineer. They are another part of looking at the entire system and applying some wisdom.

Example Interfaces

Well-defined interfaces are commonplace. First consider the electrical outlets in your home and workplace. Figure 1 shows hand sketches of common electrical plug and outlet interfaces. (The National Electrical Manufacturers Association or NEMA created and maintains these interfaces.) These interface the electric utility system to electrical appliances. The top row of Figure 1 shows the 125-Volt interfaces found in the U.S. while the bottom row shows the 250-Volt interfaces.

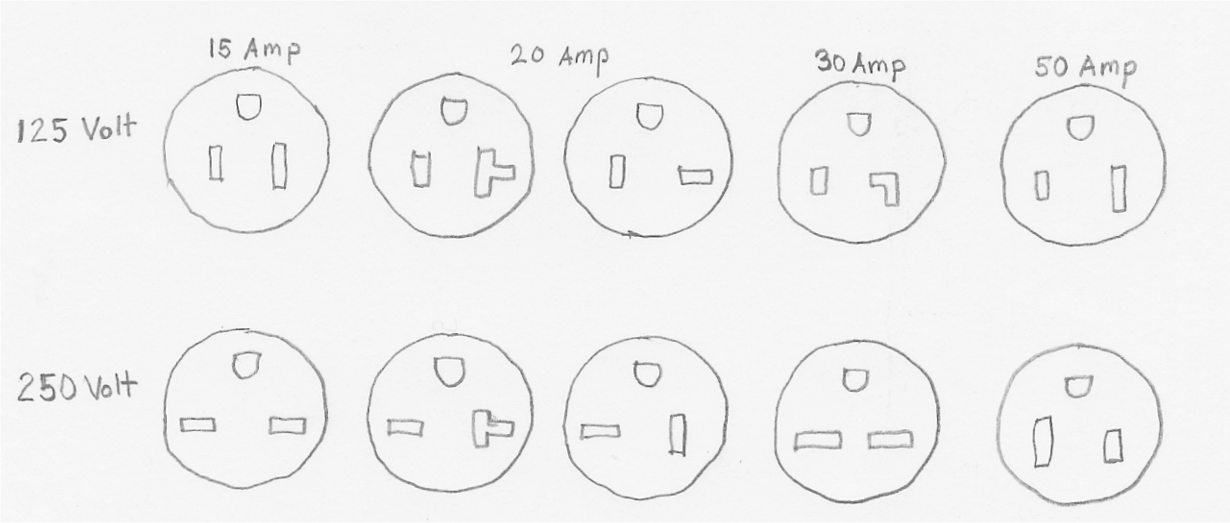


Figure 1 – Standard Electrical Interfaces

These standard electrical interfaces provide safety. Look at the 125-Volt, 15-Amp interface shown in the upper left of Figure 1. This is the most common plug-outlet interface in America. Now look at the two 125-Volt, 20-Amp interfaces in the top row of Figure 1. They use one vertical and one horizontal slot while the 15-Amp interface uses two vertical slots. If I have an appliance that draws 20 Amps of current, my appliance will have a plug adhering to the 20-Amp interface. I cannot insert the 20-Amp plug into a 15-Amp outlet. The physical interface will not allow that. If I could insert an 20-Amp plug into a 15-Amp outlet, my appliance would draw more current than the wiring in the house could safely support, and I could have a fire.

Now consider Figure 2. This shows one of the standard interfaces for 250-Volt systems that have a “twist-lock” interface. You use this interface by inserting the plug into the outlet and then twisting it. The twist locks the plug into the outlet so that it will not come unplugged unless you reverse the twist and then retract it. The twist-lock interface is required for an electrical appliance, usually a large construction power tool, that would become dangerous if its power were disconnected unexpectedly. The twist-lock interface provides extra safety for systems that need it.

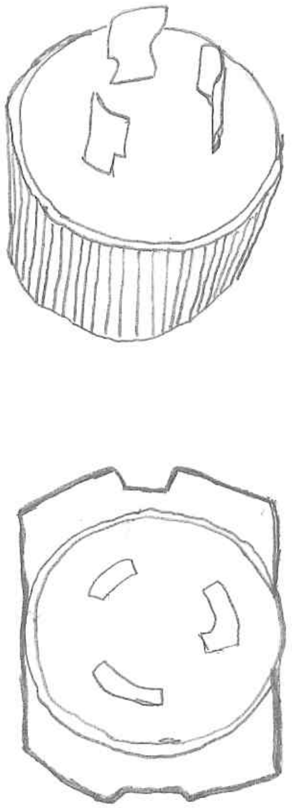


Figure 2 – A 250-volt Twist-Lock Interface

These electrical interfaces sometimes bring aggravation. Have you ever had a device with a three-prong plug and walked up to the outlet that only accepts two-prong plugs? This is what happens when I visit my in-laws. Their older home only has two-prong electric outlets (the old standard interface). My laptop computer uses a three-prong plug (the newer standard interface). I have to cheat and use a three-prong to two-prong adapter. That is safe for a laptop computer, but not so safe for a power tool or a hairdryer.

Another common interface defines outdoor faucets found on homes and businesses. This interface is set by the National Pipe Thread Standard (NPTS). The interface defines the diameter of the pipe, the number of threads per inch, and whether the threads are straight or tapered. The interface allows me to connect a garden hose to the faucet and a large number of items to the garden hose. I can wash the car, fertilize the lawn, and play on a slip n' slide because of this standard interface.

Indoor water connections use a different interface than outdoor ones. The indoor connections have more threads per inch. That provides a more water-tight seal and prevents leakage. The indoor interface costs more money, but that expense is justified by preventing water damage indoors.

Consider another attribute of the outdoor faucet interface. What if we were to send acid through the faucet instead of water? That would warrant a change in the interface.

In addition to technical, electrical, and mechanical interfaces there are also social interfaces. I visit my doctor a few times a year to check my blood pressure and adjust my medicine. I don’t knock on the front door of my doctor’s house when I think it is time for a check up. There is a defined interface between the two of us. I call the office for an appointment, and a receptionist matches the doctor’s and my availability. When I arrive at the doctor’s office I enter a waiting room. I inform the receptionist that I am there are for an appointment. The receptionist sits in a separate room and speaks through a window joining the two rooms. I complete paperwork while waiting (and waiting and waiting) to see the doctor.

When the appointment is concluded I encounter several more well-defined interfaces. There is the interface between the doctor and the accounting firm that the doctor employs. There is an interface between my health insurance provider and me. There is an interface between my doctor's accounting firm and my health insurance provider. The list of interfaces goes on and on.

Interfaces serve me in many ways. First, they reduce unnecessary thinking. Things just seem to fit together and work. Someone else like NEMA and NPTS did all the hard, detailed work for me years ago.

The interfaces also help me to raise my level of thinking. For example, when vacuuming my carpet I concentrate on cleaning the carpet. I don’t concern myself with the windings on the electric motor in the vacuum cleaner, if the motor is too small for the electric supply, if the input voltage meets the requirements of the motor, and if and if and if.

Levels of the Interface

The systems engineer will describe the interfaces between sections in an architecture. This can be a daunting task.

One way to make it easier is to apply some structure to the interface. For example, in 1977, the International Organization for Standardization (ISO), began to develop its Open Systems Interconnection (OSI) networking suite. They created a seven-layer model for describing communication interfaces among electronic systems. That model aids systems engineers in creating interfaces among those types of systems.

The OSI seven-layer model is a good one, but too complex for the current discussion. Instead, I propose a three-layer model:

3. semantic
2. descriptive
1. physical

The physical layer of the interface defines the physical pieces that I can hold in my hand. In the examples given above, Figures 1 and 2 showed the physical devices used in the electrical interfaces in my home. The water faucet with a defined diameter and threads is another physical interface.

The descriptive layer describes the material that passes through the physical layer of the interface. In the home electrical system, the material is electricity. I use 125 Volts and 15 Amps in the vast majority of my interactions with the electric system. "125 Volts" and "15 Amps" are the contents of the descriptive layer. In the water faucet example, the descriptive layer tells me that I am moving water (not acid) at a certain rate of gallons per second and at a pressure of so many pounds per square inch.

The semantic layer pertains to the messages that we may be passing through the interface and how we interpret those messages. The semantic layer doesn't apply to the electricity and water faucet examples. That is the nature of models – often the model doesn't apply to a particular situation. Use what makes sense and don't use what doesn't make sense.

The semantic layer does apply to the social interface with the doctor's office. When I talk with the receptionist we talk about appointments. We don't discuss religion or politics. In a similar vein, the interface between my health insurer and the doctor's office passes messages about types of treatment, costs, and compensation.

Defining an Interface

As a systems engineer, I often encounter situations where there is no standard interface defined for the systems I am building. I have to define the interface myself.

The usual means for defining an interface is via an Interface Control Document or ICD. The ICD was always a paper document with predefined sections (yes, an interface standard for an interface control document). In recent years, systems engineers have used web pages and even videos to describe interfaces.

One way to lessen the work involved in defining an interface is to use part of an existing interface. This is especially true for the physical layer of the interface. If I can use the NEMA-defined interface for an electrical outlet or the defined interface for an outdoor faucet, I will save lots of time and money.

Sometimes, I cannot use any existing interfaces and I have to create an entirely new one. I try to work with the levels of the interface discussed earlier:

3. semantic
2. descriptive
1. physical

Let's discuss an interface between a soccer coach for a kid's soccer league and the parents of the players. This is an example of a social interface. Trust me I have experience here, this interface should be defined as it will make life and the soccer season much easier.

Physical Layer: The coach and parents will communicate via e-mail. They will also use cell phones for emergencies only. Another part of the physical level of the interface is that at the beginning of the season the coach will give each parent a piece of paper.

The final part of the physical level is when communication will occur. The interface defines the hours of the day and days of the week that the coach and parents will check their e-mail. It also includes the times that emergency phone calls are accepted.

Descriptive Layer: This doesn't seem to apply well to this interface, so let's skip it.

Semantic Layer: This level of the interface defines the topics that are to be communicated via e-mail, emergency phone calls, and the piece of paper. The paper provides the schedule for games, practices, and parties for the entire season. This is the baseline schedule. Everyone should expect changes due to weather and other circumstances, and those changes will be communicated via e-mail. There are some topics that are permitted in one-to-many e-mails, i.e. those e-mails sent to the entire group. Some topics are permitted only in one-on-one e-mails between the coach and one parent. I find that it worked well for people to understand what information was "public" (John is out of town this weekend and will miss the game) and what was "private" (John's mother and I are in divorce proceedings so you won't see us together at the games any more).

The Semantic Layer in this example also describes what constitutes an emergency – cell phone call instead of e-mail. In my experience, this greatly reduces the time on the phone.

Another Couple of Interfaces

The example interface above was fairly simple. Let's delve into systems that are more complicated – systems with several sections like in the previous chapter on architecture.

Figures 3 and 4 show two such systems. They are both systems to water plants. Hence, they have the same architecture with four sections: Source, Transport, Dispenser, and Destination. A glance at the items in each section shows that we are considering two very different systems and situations. Figure 3 concerns a person with three houseplants to water. Figure 4 is where we have 10,000 acres of tomatoes to water. Since the systems are different, the interfaces will be different.

The following pages discuss these interfaces in detail. This is tedious. The reader may opt to skim this material. Interfaces, though a great help to me as a systems engineer, are often boring to discuss.

Notice in the two figures how I have labeled the dotted lines – the interfaces – with "ICD 0" through "ICD 3." This is a systems engineering convention that provides a convenient shorthand. Systems engineers commonly speak in terms of ICD-number.

First consider the architecture in Figure 3.

ICD 0: This interface describes how water enters the system. It separates the source of water (mother nature) from the public drinking water system.

Physical Layer: Let's assume that this particular public drinking water system obtains its water from a river. The physical interface is the river.

Descriptive Layer: The river provides a million gallons of water a day that is not potable. The public drinking water system – the first part of the system in Figure 3 – will transform the river water into potable water.

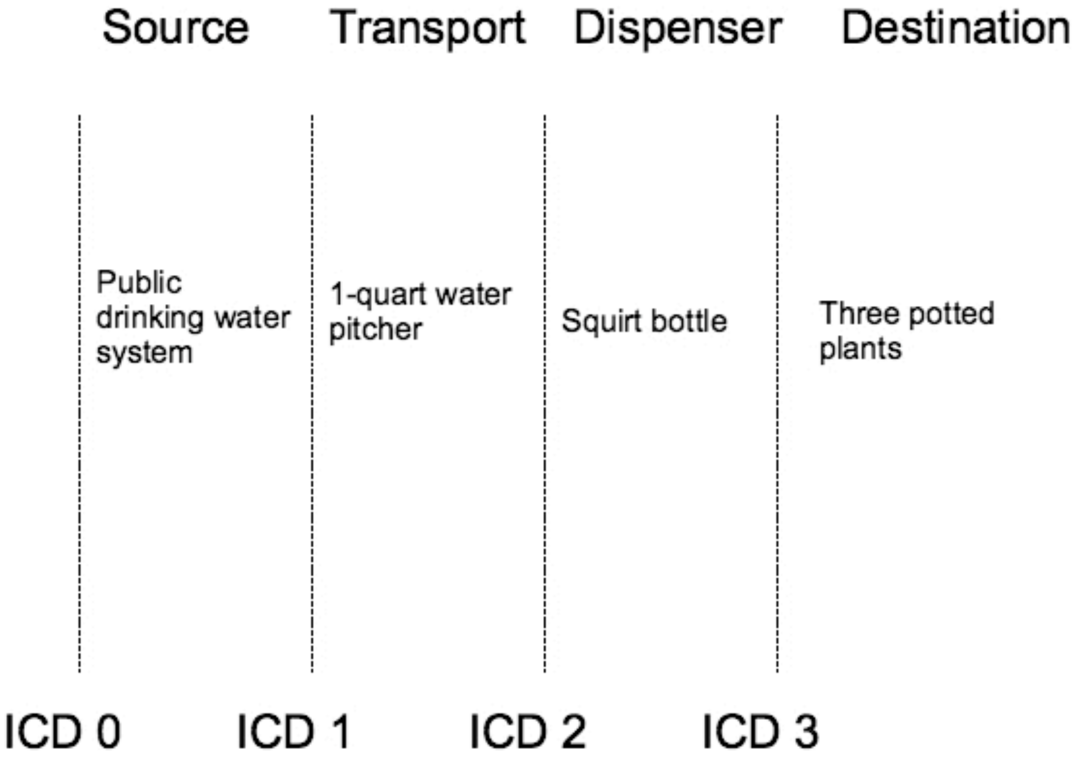


Figure 3 – A System to Water Three House Plants

Semantic Layer: This doesn't make much sense here, so let's not worry about it.

ICD 1: This interface separates the public drinking water system from the one-quart water pitcher. The water pitcher is easy to put in the sink under the faucet and fill with water.

Physical Layer: The kitchen faucet is the physical interface between the water system and the pitcher. This is pretty simple, but there could be other physical interfaces we could use like the outdoor faucet or even the main water line that enters the home.

Descriptive Layer: The kitchen faucet can deliver one gallon of water a minute. The water is clean and potable.

Semantic Layer: This doesn't make much sense here, so let's not worry about it.

ICD 2: This interface separates the one-quart water pitcher from the squirt bottle. A quart pitcher of water is too heavy to carry from the sink to the plants. Plus, it is difficult to pour a little water out of the pitcher into the small pot holding the plant. A squirt bottle is much easier to use. How can we move the water from the pitcher to the squirt bottle? The physical layer describes that.

Physical Layer: I will use a small funnel that fits into the top of the squirt bottle. This allows me to pour water from the pitcher into the bottle.

Descriptive Layer: The funnel is large enough to move one quart of water every two minutes. The water is clean and potable.

Semantic Layer: This doesn't make much sense here, so let's not worry about it.

ICD 3: This interface separates the squirt bottle from the three houseplants. I will simply squirt the water through the air onto the plants.

Physical Layer: The interface here is just the open air. This level doesn't make much sense, so let's not worry about it.

Descriptive Layer: This too doesn't make much sense here, so let's not worry about it.

Semantic Layer: Once again, this doesn't make much sense here, so let's not worry about it. The ICD 3 sentences above are sufficient for this interface.

Now consider the architecture in Figure 4.

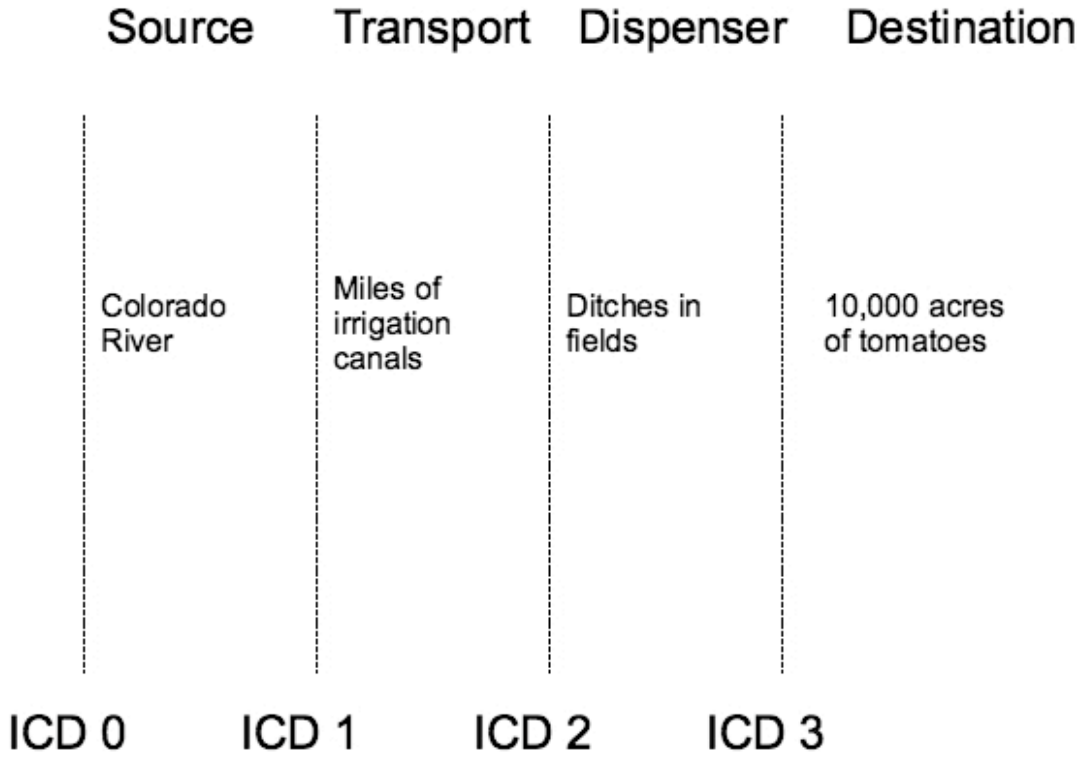


Figure 4 – A System to Water 10,000 Acres of Plants

ICD 0: This interfaces separates the source of water for the Colorado River from the river itself. The river is fed by streams and mountain run-off.

Physical Layer: The physical interface is miles of streams and mountains.

Descriptive Layer: The water feeds in at a rate ranging from a trickle to a million gallons a day. The water is fairly clean in its pristine, natural state.

Semantic Layer: This doesn't make much sense here, so let's not worry about it.

ICD 1: Large farms, like the one we are watering in this system, use miles of irrigation canals to move water. This interface separates the Colorado River from these canals.

Physical Layer: The interface between the river and the canals comprises dozens of large sluice gates.

Descriptive Layer: These sluice gates have a combined capacity of 100,000 gallons of water a day.

Semantic Layer: The material moving through the interface is water, and there isn't much of a semantic interface with water. These gates, however, are not merely "open" or "closed." They operate on a computer-controlled system that regulates the amount of water from the river going to the fields. A semi-automated system monitors the amount of moisture in the fields, the weather forecasts, and the rules of thumb that the farm manager has gathered from years of experience. The controlling computers pass electrical signals to the sluice gates to pass variable amounts of water from the river to the irrigation canals. This is a complex and expensive interface.

ICD 2: This interface separates the large irrigation canals from the smaller ditches in the fields.

Physical Layer: The interface between the canals and the ditches is similar to interface between the river and the canals. In this case the sluice gates (hundreds of them) are smaller and simpler.

Descriptive Layer: These smaller sluice gates have a combined capacity of 1,000 gallons of water a day.

Semantic Layer: The semantics of the smaller sluice gates are simpler than the large sluice gates. These gates are opened variable amounts by farm workers under the guidance of the farm manager. The messages passed between the manager and workers seem simple ("a little more, that's it"), but like most words passed between people, they can be confusing.

ICD 3: This final interface is between the water-carrying ditches and the 10,000 acres of plants.

Physical Layer: This interface comprises the individual rows of tomatoes. This seems obvious, so let's not worry too much about it.

Descriptive Layer: The rows have a capacity of 100 gallons of water a day.

Semantic Layer: This doesn't make much sense here, so let's not worry about it.

Architecture and Interface: A So-So Episode

I illustrate the value of using both architecture and interfaces with an episode I experienced at work. A group of engineers had built a set of devices whose function resembled that shown in Figure 5. The devices had Sensors known as "Switched" Sensors because they could be turned on and off.

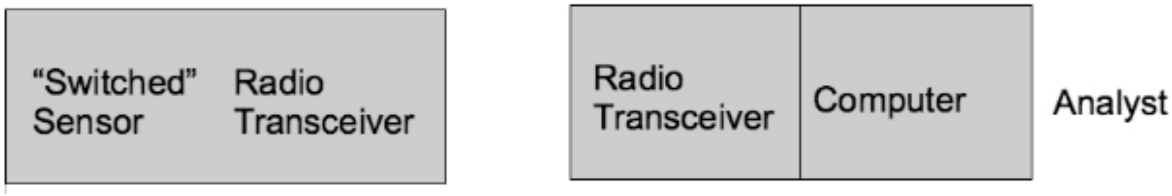


Figure 5 – A Typical Device

This set of devices used a number of different digital sensors including:

- still camera
- video camera
- fluid level
- thermometer
- pressure gauge

The output of the Sensor was sent through a pair of Radio Transceivers (devices with a transmitter and receiver) to a Computer that displayed the sensed information to an Analyst – a person. The Transceivers could send information back from the Analyst to the Switched Sensor. This allowed switching the Sensor on and off.

What differentiated these devices was the Sensor (camera or thermometer) and the data rate needed for the Radio Transceivers. Consider using a digital camera that takes a one-megabyte image once a day. This permits 24 hours to send the megabyte of data from the Sensor across to the Computer and Analyst – a data rate of about 12 bytes per second. That is a pretty slow data rate, one that may also work for a thermometer, pressure gauge, and fluid level. If, however, the camera took one picture every second, the data rate would be 1,000,000 bytes per second. If the Sensor were a video camera operating at 30 frames per second, the data rate would be 30,000,000 bytes per second. So, with these few examples, the data rate could range from 12 bytes per second to 30 million bytes per second – quite a difference.

The engineers in this episode had a successful set of devices because the customers used every device built and were satisfied. The engineers failed in this episode because they lacked both an architecture and defined interfaces. The problems that ensued are described later.

After a few hours of discussions between several systems engineers and the engineers in the episode, we drew the architecture with interfaces shown in Figure 6.

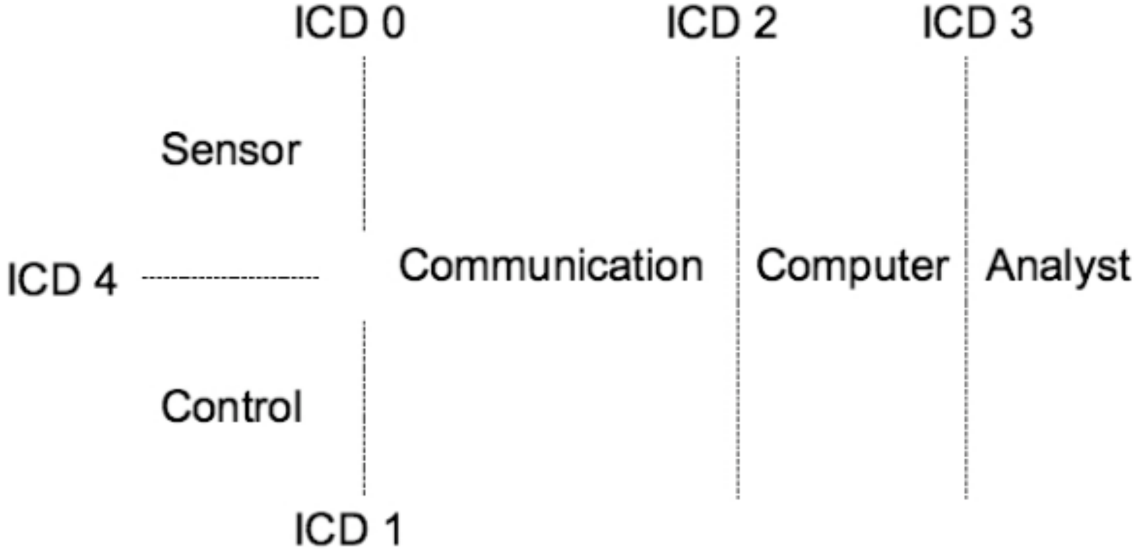


Figure 6 – An Architecture with Interfaces Corresponding to Figure 5

The architecture in Figure 6 first separates the Sensor section from the Communication section. Back in Figure 5, the engineers built each Sensor and Radio Transceiver as one part. Separating the Sensor and the Communication allows swapping a type of Communication to fit the data rate needed.

Replacing the Radio Transceivers with a Communication section allows many more options. The first option is that there are several types of communication links that work via methods other than radio. The simplest concept is using wires instead of radio to connect the two transceivers. Wires have advantages in different situations. The other options involve the data rate. Communication devices with lower data rates can be made smaller and use less power than devices with higher data rates.

Given the above, it is simple to have three different data rates (100, 1,000,000, and 50,000,000 bytes per second) and two different means of connection – radio and wired. This gives six different types of Communication items.

A third change between Figures 5 and 6 is creating a separate Control section that can switch the Sensor on and off. The Control section has defined interfaces with both the Communication and Sensor sections. These changes permit using more complex control mechanisms. In addition to on and off, the Control section could vary the number, timing, and amount of data sensed as well as other attributes.

Any item that fits in a section, i.e. it meets the requirements of the section and the interfaces, can be used in the section. This allows a family of items. The example of Figure 6 can have a family of Sensors, a family of Communication items, and a family of Control items. Defined interfaces permit mixing and matching items in the architecture depending on the situation. This brings advantages such as:

- reusable items
- simpler items
- easier maintenance
- greater flexibility
- reduced logistics expenses
- systems that better match the customers' situations

These advantages were all absent when the engineers were building devices such as shown in Figure 5. Those devices had disadvantages such as:

- limited use
- greater complexity
- difficult maintenance
- longer time required to deliver
- higher cost

Let's expand on the expense as an example. The engineers estimated the number of each type of device the customer needed. The problem was that the estimate was too high for some types and too low for others. The items they under estimated required ordering a second time. This lost the cost savings available in single, large orders. The items they over estimated sat on the shelf unused. The unused items had the effect of raising the cost per item of those used.

Using an architecture and interfaces, the engineers would have ordered families of items. These families could be assembled into systems as desired by the customers. The flexibility would have meant greater efficiency in the use of the parts with a cost savings.

I titled this section a “So-So Episode” because of the result. The engineers didn't adopt the architecture and interface methods recommended by the systems engineers. They just didn't want to change the way they built devices. The organization had an excess of funds that year, so their managers weren't excited about cost savings. The customer just wanted something that worked and wasn't interested in the mumbo jumbo engineering philosophy chat.

I do claim some success in this episode. This was the first time that several of the younger engineers had heard of these systems engineering concepts. They did see the value of architecture and interfaces. The episode met the minimum criteria for introducing a new concept: they were more interested in systems engineering than before.

Closing Thoughts

The subject of interfaces and ICDs can be tedious. I hope you have been able to read through this chapter. I've presented a mechanical process that isn't exciting:

- Label each interface ICD 0 through ICD n on the architecture drawing

- Outline a piece of paper with the ICDs and the three layers in each ICD
- Write the ICD details on that piece of paper

Interfaces, like architecture in the previous chapter, provide for a conversation between the systems engineer and the customer. This conversation is their greatest value. Please understand that the customer isn't interested in ICDs – he is only interested in a satisfying system. The ICDs, however, can help the customer think about what will be satisfying. Take care in how you, the systems engineer, present the ICDs to the customer for a conversation.

Please notice all the people in the systems and interfaces examples of this chapter. The principles of systems engineering apply to computers, machines, and tools, but they also apply to groups of people. Take great care with this thought. People aren't cogs in machines, and don't think of them as such.

Suggested Exercises

1. While drafting this chapter, I looked around my home for examples of interfaces. Some examples included:

- TV remote control
- Light bulbs (the screw threads work, they even work with those new fluorescent lights).
- Doorknob and lock mechanism fits into the door and doorframe.
- Plates in a kitchen cabinet. The plates seem to be a standard size as do the cabinets, and they interface properly.

Define the physical, descriptive, and semantic layers of these interfaces.

2. I'm in a library while editing this chapter, and there are over-sized books, i.e. books that don't fit in the standard-size bookcases. Which part of the book-to-book-case interface control document has been violated? Look around where you are right now and find something like over-sized books that doesn't fit. For what you have found, what part of the interface control document has been violated?

3. Consider the system in Figure 3. As the systems engineer, how would you present the ICDs to the customer to enable a conversation? What would be the advantages and disadvantages of:

- Giving the paper ICDs to the customer
- Reading the ICDs to the customer
- Demonstrating the ICDs to the customer
- Some other method not listed above

Where We are Now

At this point, the systems engineer has built the system on paper – so the systems engineer is done, right? Well, let's do one more fundamental technique – tracing.

Tracing allows the systems engineer to step through the system on paper: requirements then architecture then interfaces. Errors are easier to see, changes are easier to accommodate, and tests are easier to perform because of tracing.

Tracing makes the systems engineer much wiser when trying to examine the entire system.

5 Tracing

Making systems engineering a little easier

I loved to draw as a kid. Sometimes I would trace a picture instead of drawing it freehand. Tracing made some really hard drawings much easier. Place a piece of that special paper on top of the picture and slowly trace. Something so simple was such a big help.

The systems engineer can also use tracing. The tracing helps the system engineer examine the entire system and apply a little wisdom.

Classic Tracing

This section will illustrate the classic method of tracing. The concept is simple: draw a sketch that allows the systems engineer to trace a line from one artifact to the next.

Figure 1 shows the basic structure of a tracing diagram. In the upper left are the system's requirements. The system's requirements flow down and right to the architecture level of the system. Here the requirements are allocated to each section of an architecture and the interfaces between the architecture sections. These flow to the item(s) in the section. Flowing down further are how these requirements are fulfilled in the detailed design of an item. Finally, while building or buying the item, the requirements are met in the physical item.

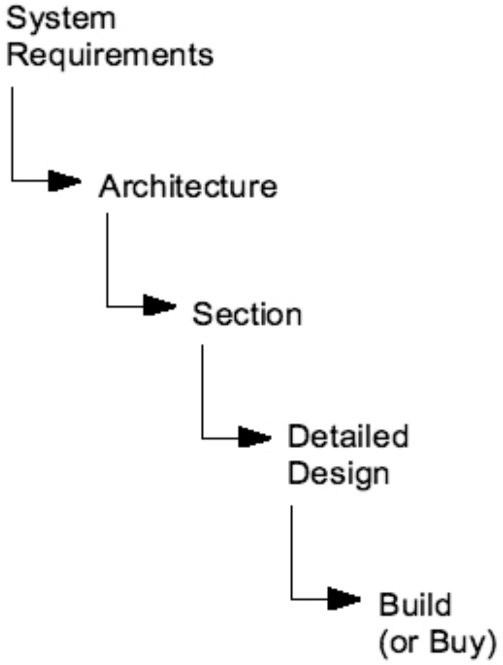


Figure 1 – A Basic Tracing Diagram

Figure 2 shows an architecture copied from the Interfaces chapter. This architecture has four sections: Source, Transport, Dispenser, and Destination. Each section is present because it has a part in satisfying the customer's requirements. The Interfaces between these sections (represented by ICD 0 through ICD 3) also fulfill part of the customer's requirements.

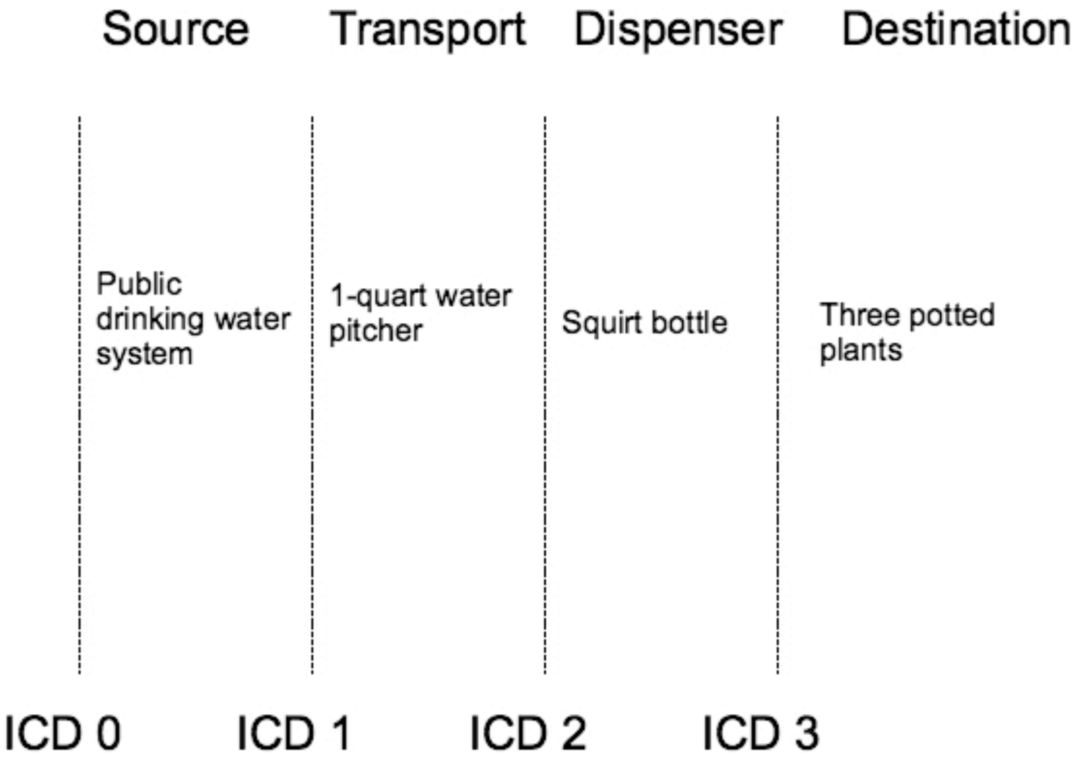


Figure 2 – An Architecture Diagram from the Interfaces Chapter

Figure 3 shows an example of flow-down and tracing. This is an overwhelming figure at first glance. Let's walk through it piece by piece and limit the discussion to four simple requirements of a system to water house plants. These requirements are:

- R1: Water three potted plants.
- R2: Water once a week.
- R3: Plants are daisies, tomatoes, and radishes.
- R4: The customer carries several items while watering the plants.

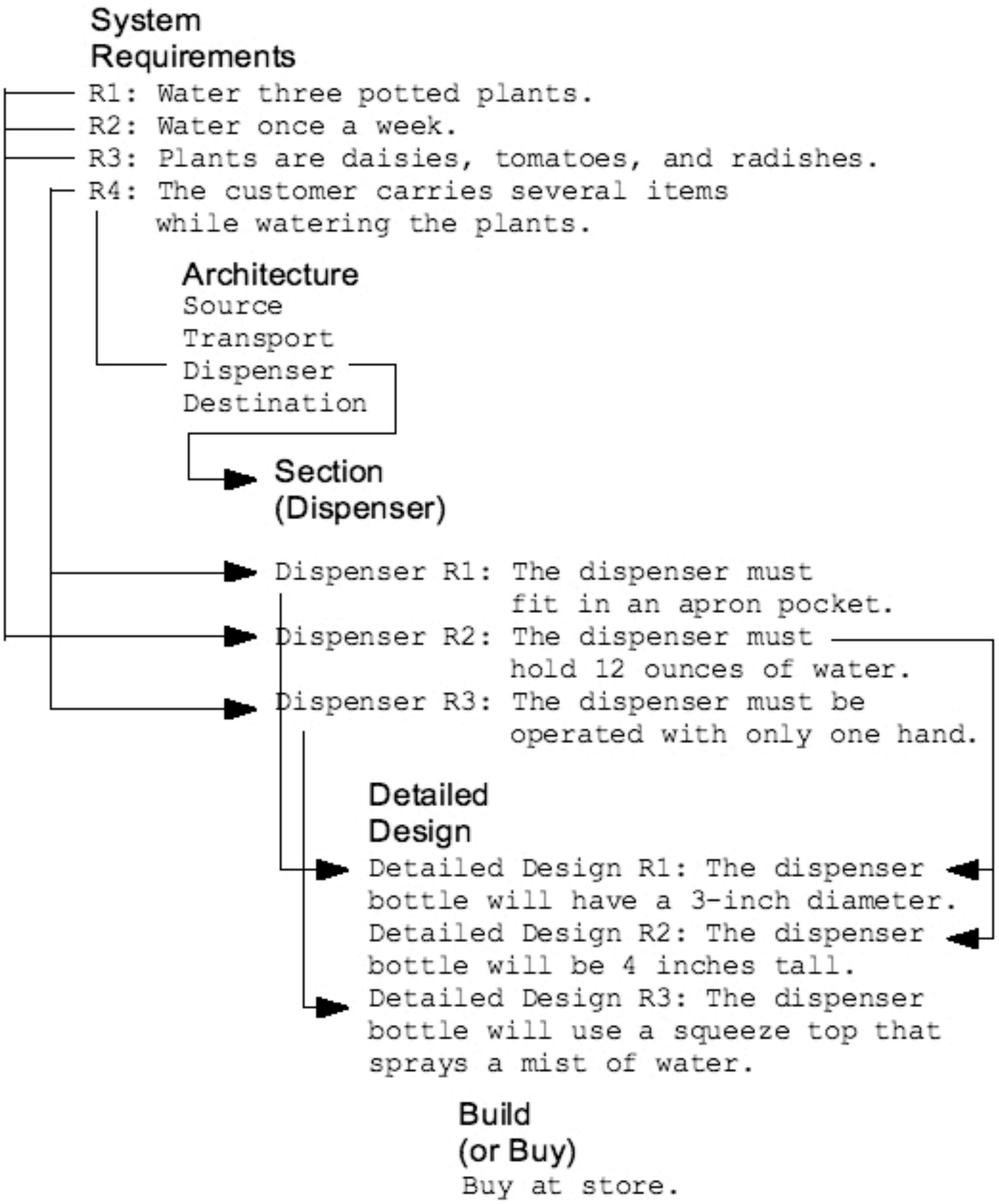

System Requirements
R1: Water three potted plants.
R2: Water once a week.
R3: Plants are daisies, tomatoes, and radishes.
R4: The customer carries several items while watering the plants.
Architecture
Source
Transport
Dispenser
Destination
Section (Dispenser)
Dispenser R1: The dispenser must fit in an apron pocket.
Dispenser R2: The dispenser must hold 12 ounces of water.
Dispenser R3: The dispenser must be operated with only one hand.
Detailed Design
Detailed Design R1: The dispenser bottle will have a 3-inch diameter.
Detailed Design R2: The dispenser bottle will be 4 inches tall.
Detailed Design R3: The dispenser bottle will use a squeeze top that sprays a mist of water.
Build (or Buy)
Buy at store.

Figure 3 – A Tracing Diagram Including the Requirements in Each Level

Let's limit the discussion to the Dispenser section and how Requirements R1, R2, R3, and R4 affect the Dispenser section. Conversations with the customer reveal several requirements for the Dispenser section of Figure 3. These are elaborations of requirements R1, R2, R3, and R4. These elaborations or "flowed-down" requirements are:

- Dispenser R1: The dispenser must fit in an apron pocket.

- Dispenser R2: The dispenser must hold 12 ounces of water.
- Dispenser R3: The dispenser must be operated with only one hand.

The "Dispenser R1" requirement flows down from requirement R4. The customer cannot be carrying the dispenser while carrying other items. Hence, the dispenser must fit in an apron pocket.

The "Dispenser R2" requirement flows down from requirements R1, R2, and R3. Watering these three plants once a week requires 12 ounces of water. (My apologies to readers who know a lot more about watering house plants than me. I am using arbitrary numbers in this example.)

The "Dispenser R3" requirement also flows down from requirement R4 as two-hand operation won't work for the customer, e.g. the customer may be speaking on the phone while watering plants.

Now the systems engineer flows down the three Section-level requirements to the Detailed Design level of Figure 3. The systems engineer elaborates the requirements in terms that make sense for this level. Part of the requirements become:

- Detailed Design R1: The dispenser bottle will have a 3-inch diameter.
- Detailed Design R2: The dispenser bottle will be 4 inches tall.
- Detailed Design R3: The dispenser bottle will use a squeeze top that sprays a mist of water.

The "Detailed Design R1" requirement flows from the "Dispenser R1" requirement. A 3-inch-diameter bottle will fit in an apron pocket.

The "Detailed Design R1" and "Detailed Design R2" requirements combine to meet the "Dispenser R2" requirement as a 28-cubic-inch bottle will hold 15 ounces of water. That is more than enough.

The "Detailed Design R3" requirement flows down from the "Dispenser R3" requirement as it allows for one-handed operation.

At the Build level of Figure 3, the systems engineer decides to go to the store and buy such a squirting bottle. The size and mechanism required by the Detailed Design level are easily met at a hardware store.

Notice the principles from Figure 3.

(1) Each thing in the figure flows from some other thing. Most of the things flow down from something above them in the figure. The System Requirements in the upper left corner flow from the systems engineer's conversations with the customer.

(2) Each thing in the figure flows to something more detailed (except for the thing in the lower right which is the actual item in the system).

Also notice that Figure 3 does not show everything related to the system. It only shows how three requirements flow down to one section of the architecture, and how those flow to the actual water dispenser. The entire tracing diagram for this simple system – water three houseplants – would cover a wall. While the tracing diagram provides a good visual display of flow-down, it quickly becomes unusable. In practice, I recommend other tools (see later) to keep track of the flows and enable tracing.

Examining a wall-sized diagram would be overwhelming. That is a characteristic of tracing in systems engineering: it quickly becomes just too much work. Nevertheless, I believe it is worth the effort as the following sections illustrate.

Tracing and Changes

A fundamental concept in an engineered system is that everything is connected. Hence, a change in one place in a system leads to changes in other places. Ensuring that all the changes throughout the system are correct is a basic and daunting challenge facing the systems engineer.

Tracing permits changes to the system. The systems engineer works with customers who learn. The learning customer wants changes to the system while the system is being built. Hence, change is inevitable and the systems engineer is best served with a method that accommodates change. Let's walk through some change scenarios.

A Change Scenario

First, consider a change in requirement R1 from watering three potted plants to watering six potted plants. The basic requirements shown earlier become:

- R1(new): Water *six* potted plants
- R2: Water once a week
- R3: Plants are daisies, tomatoes, and radishes
- R4: The customer carries several items while watering the plants.

Combining R1 (new) with R2, and R3, the systems engineer can trace these down to the Section level of Figure 3, i.e. "Dispenser R2: The dispenser must hold *12 ounces* of water." The Dispenser requirements become:

- Dispenser R1: The dispenser must fit in an apron pocket.
- Dispenser R2 (new): The dispenser must hold *24 ounces* of water.
- Dispenser R3: The dispenser must be operated with only one hand.

The systems engineer continues to trace down through Figure 3 to the Detailed Design level. The dispenser still must fit in an apron pocket, so it still must be no larger than 3" in diameter. To double the water-carrying capacity, the dispenser now will be 8" tall instead of 4" tall. The requirements at the Detailed Design level become:

- Detailed Design R1: The dispenser bottle will have a 3-inch diameter.
- Detailed Design R2 (new): The dispenser bottle will be *8 inches* tall.
- Detailed Design R3: The dispenser bottle will use a squeeze top that sprays a mist of water.

Now the systems engineer and the customer need to talk about the implications of changing from three to six potted plants. Can the customer carry a bottle that is 8" tall and holds 24 ounces of water in an apron pocket? Can the customer use this larger bottle with one hand? If the answers are "yes," the systems engineer is done. Simply alter the figure with the changed requirements and go to the store and buy a bigger bottle. If the answers to the questions are "no," the systems engineer and customer have more work to do.

Tracing enables this essential conversation between the systems engineer and the customer. The trace through the diagram shows the systems engineer what will have to change to satisfy the customer's new desires. The trace through the diagram enables the systems engineer to examine the entire system and apply a little wisdom.

Another Change Scenario

A second example: the customer tells the systems engineer that he has a new apron which has a bigger pocket than the old apron. This change takes the systems engineer to the Section level of Figure 3. The requirements here are:

- Dispenser R1: The dispenser must fit in *an apron pocket.*
- Dispenser R2: The dispenser must hold 12 ounces of water.
- Dispenser R3: The dispenser must be operated with only one hand.

The next level down – the "Detailed Design" level – has the requirements that flow down from these. The requirement "Dispenser R1" traces down to requirement "Detailed Design R1."

- Detailed Design R1: The dispenser bottle will have a 3-inch diameter.
- Detailed Design R2: The dispenser bottle will be 4 inches tall.
- Detailed Design R3: The dispenser bottle will use a squeeze top that sprays a mist of water.

Since the apron pocket will be larger than before, the 3-inch diameter dispenser will still fit in it. The systems engineer will not have to make any changes to Figure 3. That is good news because the customer's new apron doesn't require changes to the system. Again, the figure enables the systems engineer to examine the entire system and apply some wisdom.

Yet Another Change Scenario

Now consider a different change to the apron pocket. In this change the size of the pocket decreases. The systems engineer traces through the diagram just as in the second example and can see that a 3-inch diameter bottle is in the detailed design.

The systems engineer and customer need to determine if the 3-inch diameter dispenser will fit in the new, smaller pocket. If it does, the systems engineer and customer are finished as no changes are needed. If it doesn't, the systems engineer and customer have more work to do. They have to determine what diameter dispenser will fit in the new pocket. How will a smaller diameter affect the height of the dispenser? Will a taller dispenser satisfy the customer? What else might the systems engineer and customer have to do?

Again, Figure 3 and tracing enable the systems engineer to see how a change in something like the size of an apron pocket affects the rest of the system. The systems engineer can see the entire system and apply wisdom.

One Final Change Scenario

Consider a fourth change: The 3” diameter and 4” tall dispenser is not available at the store. The systems engineer cannot simply buy the dispenser as designed. The systems engineer has to determine what size dispensers are available at the store.

One that has 2” diameter and is 9” tall would have the required 28 cubic inches of capacity. That model, however, is not available.

The systems engineer can find a dispenser that has a 2” diameter and is 5” tall. That has 16 cubic inches of capacity but fails to meet the 28 cubic inches required by:

- Detailed Design R1: The dispenser bottle will have a 3-inch diameter.
- Detailed Design R2: The dispenser bottle will be 4 inches tall.

The systems engineer needs to talk with the customer. The customer never required that all watering be made in one trip. Would the customer be satisfied with having to refill the dispenser after watering two of the three plants?

If the answer is “yes,” the systems engineer is almost finished. Simply change Detailed Design R1 and R2 to meet the actual dispenser that is available at the store.

If the answer is “no,” the systems engineer and customer have more work to do. Maybe they can find the desired size dispenser at another supplier. Maybe they can modify a larger dispenser by cutting it to size. Maybe they have to build their own dispenser.

These necessary discussions only occur because the systems engineer can see the affect of changes that the customer requests. Tracing through Figure 3 allows the systems engineer to see the entire system and apply some wisdom.

Tracing and Testing

Testing is an important aspect of systems engineering. The systems engineer’s goal is to satisfy the customer. Testing validates that the system works to our understanding of the customer's requirements.

Several questions the systems engineer asks about testing include:

- “What do we test?”
- “How do we test?”

Tracing and drawings like Figure 3 help the systems engineer ensure that the right things are tested in the right manner.

Figure 4 shows the relation between tracing and testing. The left half of Figure 4 is what was shown in Figure 1. The right half adds testing. There is a set of tests that correspond to each level in the tracing diagram.

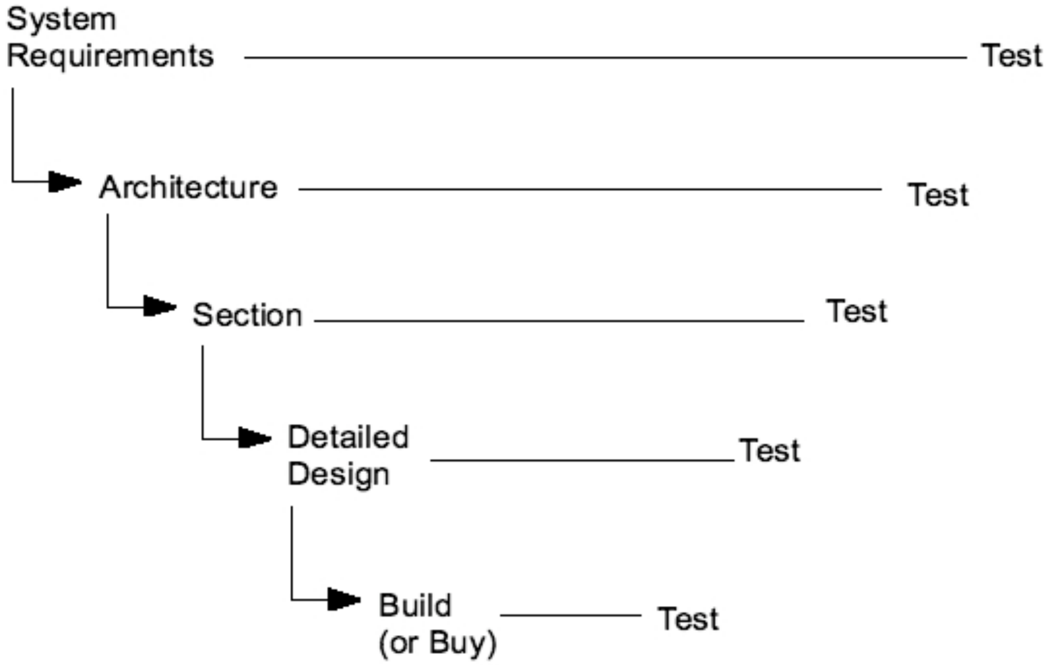


Figure 4 – Adding Testing to the Tracing Diagram of Figure 1

For each of the five levels in Figure 4, the systems engineer asks, "Why do we perform this test?"

The answer is, "To ensure that this thing is what it is supposed to be."

The example in Figure 5 helps illustrate this concept. The left half in Figure 5 is Figure 3. The right half of Figure 5 adds the five corresponding test levels.

Let's trace through the levels of testing of Figure 5. At the bottom level, the testing validates that a squirt bottle is present. This isn't much of a "test," but it ensures that we have something that we can test at the higher levels. Don't skip this "we have something" test. I have personal (tragic) experience with builders starting extensive and expensive tests without anything in their hands to test.

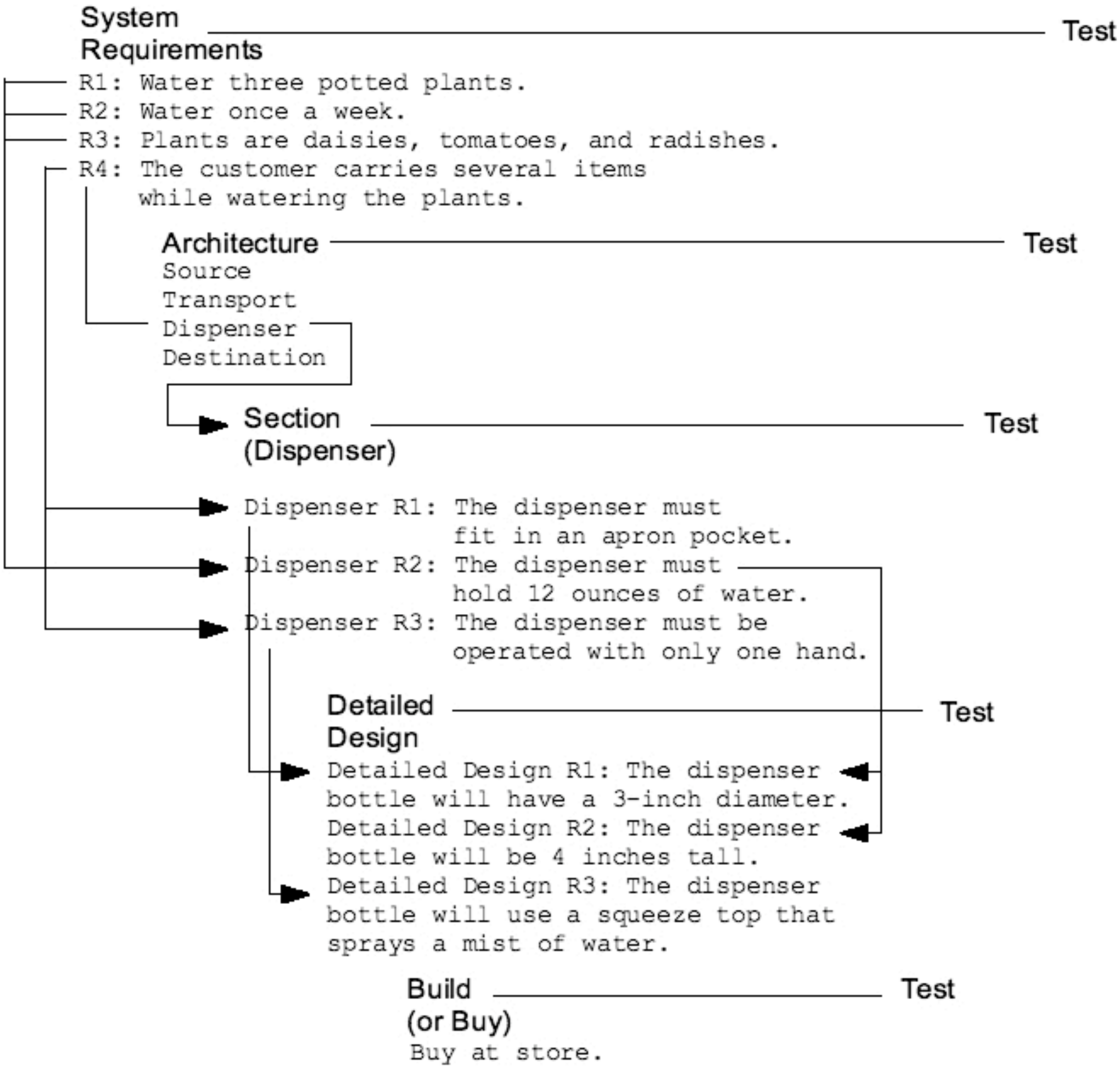

System Requirements
Test
R1: Water three potted plants.
R2: Water once a week.
R3: Plants are daisies, tomatoes, and radishes.
R4: The customer carries several items while watering the plants.
Architecture
Test
Source
Transport
Dispenser
Destination
Section (Dispenser)
Test
Dispenser R1: The dispenser must fit in an apron pocket.
Dispenser R2: The dispenser must hold 12 ounces of water.
Dispenser R3: The dispenser must be operated with only one hand.
Detailed Design
Test
Detailed Design R1: The dispenser bottle will have a 3-inch diameter.
Detailed Design R2: The dispenser bottle will be 4 inches tall.
Detailed Design R3: The dispenser bottle will use a squeeze top that sprays a mist of water.
Build (or Buy)
Test
Buy at store.

Figure 5 – Adding Testing to the Tracing Diagram of Figure 4

Next up in Figure 5 is the test corresponding to the Detailed Design level. At this point, the testers measure the size of the dispenser to ensure that it is 3 inches in diameter and 4 inches tall. The testers also ensure that the dispenser has a squeeze handle that can be operated with only one hand.

Continuing up in Figure 5 is the test corresponding to the Section or Dispenser level. The testers try to put the dispenser in the customer's apron pocket. They also try to pour 12 ounces of water into the dispenser. Finally, they ensure that a person like the customer can hold the dispenser in one hand.

The process continues up to the level of Figure 5 corresponding to the Architecture. The testers verify that the needed squirt bottle fits in the Dispenser section of the architecture and that it fits with the ICDs there.

Testing finally moves up to the System level. Here the testers (after running the tests by themselves) ask the customer to test the item in the Dispenser section of the architecture, i.e. the squirt bottle. The testers give the customer the squirt bottle containing water and stand back. They observe the customer watering his three houseplants while wearing his apron, carrying other items, and using one hand to water the plants.

If at the end of testing, the customer is not satisfied and wants more, tracing helps the systems engineer scan through the diagram and locate where changes are needed. The systems engineer can then work through the changes using steps like the change scenarios given earlier.

The interplay between tracing and testing may seem obvious. Of course the squirt bottle will work. What could be simpler? This is a simple concept. It falls back to the common sense argument. It is common sense that you would test to validate that the final system does what is supposed to do. Also recall that common sense does not always lead to common practice. I have seen many systems “finished” that do not satisfy even the most basic requirements and desires of the customer.

Use the relationship between tracing and testing. Don't assume that it is too obvious and simple.

Tools for Tracing

The two figures that illustrate tracing and testing (Figures 3 and 5) are messy, complicated, and hard to follow. All that and they only show the tracing for one part of a simple system. While they illustrate tracing and how tracing helps the systems engineer, they are impractical in many situations.

There are other tools the systems engineer can use to trace through a system. The first employs the same figures on a larger scale. Use a computer and a plotter to draw a large figure – one that is four feet wide by eight feet tall (taller if your ceilings are taller). Such a large figure would permit putting the trace of an entire, complex system in a picture that is readable and usable.

Another form of a large picture is to use a wall, cards, and string. Write the words shown in the figures on 3x5 or 5x7 cards. Tack or tape the cards to the wall in the cascading form of Figure 3. Don’t draw lines on the walls! Instead, make lines that connect the cards and words with string or yarn. This requires lots of space on large, blank walls, but provides a picture of the system trace in a usable format.

A table or a matrix of text provides all the tracing information of the figures without drawing. There are several methods employing this basic idea, and they all are much easier when used on a computer. The simplest is to use a word processor program, create a large table, and enter the System Requirements in the first column, the Architecture sections in the next column, and so on. This is a little easier if the page is turned sideways (landscape instead of portrait format).

The systems engineer can make this even easier by using a spreadsheet program. Use the spreadsheet's built-in rows and columns to ease some of the mechanics of the table. Various spreadsheet programs also provide the ability to link a cell to another cell far away in the table. That makes jumping from place to place while tracing easier.

There are software products available that perform the requirements management and tracing described in this chapter. A leading product is DOORS from Telelogic (see http://telelogic.com). I have worked on several projects that used DOORS. It works quite well. It is, however, more costly than a word processor or spreadsheet and, because of its many capabilities, it requires substantial time to learn how to use. There are other commercial software products that perform the same functions as DOORS. An Internet search on "requirements management software" points to them.

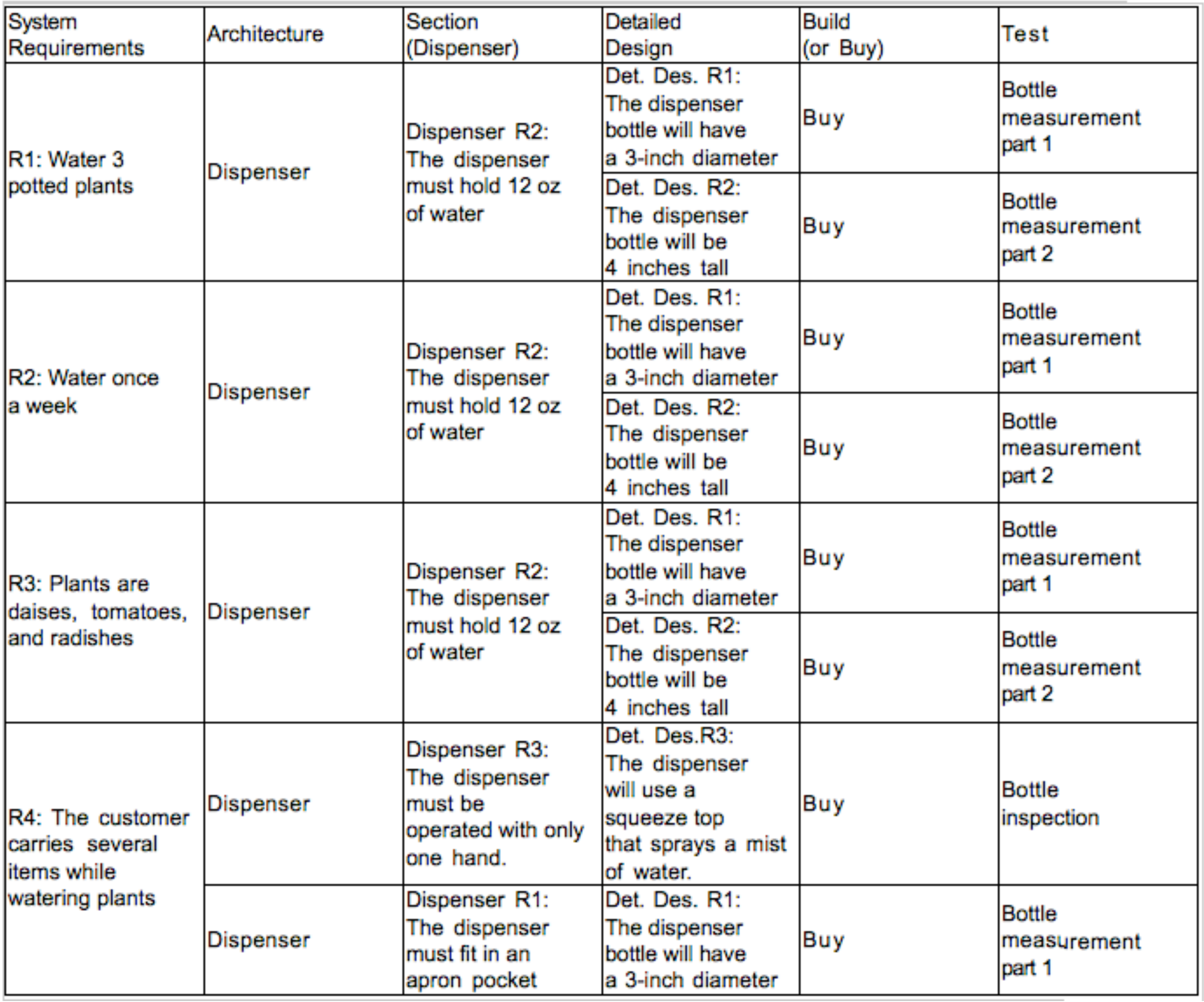
Also in the class of software products for tracing is the Open Source Requirements Management Tool or OSRMT. This is an open-source software product (see http://sourceforge.net and http://osrmt.com) that comes free of charge. I haven't used it personally, but it is rated highly on the Internet, and since it is open-source, you can download and test it without charge.

The different tools described above share a common characteristic: they allow the systems engineer to look at a requirement or design description anywhere in a system and trace backwards and forwards. Each thing in the system has a parent(s) or child(ren).

A big drawing works well for me personally as I like to literally see the traces up and down. Tables of text of varying specialization also work as long as there is a place in the table indicating where to go up and down in the trace.

A specific type of tracing table is known as the Requirements Verification Test Matrix or RVTM. Figure 6 shows the RVTM of the example of Figures 3 and 5. A key part of the RVTM is the column to the far right. This column points to the tests where the various items are validated. It corresponds to the tests that climb up the right side of Figure 5.

The RVTM is a powerful tool for the systems engineer. A blank cell in the RVTM indicates that the tracing is broken, i.e. something doesn't have a parent or child – something in the system doesn't belong in the system. Building an RVTM is difficult. Tracing through a system without an RVTM is really difficult. The software tools mentioned above are all appropriate for building an RVTM.



System Requirements	Architecture	Section (Dispenser)	Detailed Design	Build (or Buy)	Test
R1: Water 3 potted plants	Dispenser	Dispenser R2: The dispenser must hold 12 oz of water	Det. Des. R1: The dispenser bottle will have a 3-inch diameter	Buy	Bottle measurement part 1
			Det. Des. R2: The dispenser bottle will be 4 inches tall	Buy	Bottle measurement part 2
R2: Water once a week	Dispenser	Dispenser R2: The dispenser must hold 12 oz of water	Det. Des. R1: The dispenser bottle will have a 3-inch diameter	Buy	Bottle measurement part 1
			Det. Des. R2: The dispenser bottle will be 4 inches tall	Buy	Bottle measurement part 2
R3: Plants are daises, tomatoes, and radishes	Dispenser	Dispenser R2: The dispenser must hold 12 oz of water	Det. Des. R1: The dispenser bottle will have a 3-inch diameter	Buy	Bottle measurement part 1
			Det. Des. R2: The dispenser bottle will be 4 inches tall	Buy	Bottle measurement part 2
R4: The customer carries several items while watering plants	Dispenser	Dispenser R3: The dispenser must be operated with only one hand.	Det. Des.R3: The dispenser will use a squeeze top that sprays a mist of water.	Buy	Bottle inspection
	Dispenser	Dispenser R1: The dispenser must fit in an apron pocket	Det. Des. R1: The dispenser bottle will have a 3-inch diameter	Buy	Bottle measurement part 1

Figure 6 – An RVTM Corresponding to the Tracing Diagram in Figure 5

Fit

As mentioned several times in this book, I like the concept of fit. I have always felt that what I am doing at work fits in some grand plan. I have been disappointed many times, but on a few joyous occasions I learned that my part did fit into something much bigger than myself and what I could do alone.

My fondness for fit is one of the reasons that tracing and the figures in this chapter appeal to me so much as a systems engineer. When tracing, everything in the diagrams and tables has a place to fit in the actual system that satisfies the customer. Each thing comes from something and it leads to something.

The tracing diagram or table is like a jigsaw puzzle representing what will become a system. A hole stands out like a missing jigsaw piece. The puzzle allows the systems engineer to examine the entire system. It can make the systems engineer seem so much wiser than he is.

Closing Thoughts

At this point, some readers may be squirming in their chairs. The classic tracing described in this chapter is a super-organized method. It fulfills the cliché of "a place for everything and everything in it place." Some of us love that cliché and strive to live by it. Some of us hate it. Neatness and organization impose rigidity and smother our creativity. On top of that, the tracing diagrams show the classic waterfall process of building systems that bureaucracies force on creative people. And oh, the paperwork this creates!

These criticisms are valid. Classic systems engineering, flow-down, tracing, and RVTMs are systematic and organized methods. They can become burdensome bureaucracies generating mountains of useless paper.

Tracing, however, is being used in agile, iterative, and evolutionary methods. In some of these methods, users write stories of how they will use a system on a card (a story card). These cards are tacked to the wall where everyone can see them, and the builders grab the cards they think they can implement in a short time frame (one or two weeks).

The concepts of tracing work in such methods. The story card is the System Requirement. It flows down to a broad idea that flows down to more detailed ideas that flow down to a product. There is a way to test each product, idea, and story card. This is tracing using a few different words and mechanisms. The figures and tables shown in this chapter express the classic concept of tracing. Use the concepts and modify them to fit in your situation and culture.

If a line in the tracing diagram leads nowhere, something is missing or the source of that line is an unnecessary extra. The hole in the diagram tells me as the systems engineer to learn which is the case. If there is a blank cell in the RVTM or other tracing table, I know the same – either something is missing or there is something extra and needless.

Further Exercises

1. Find a common object from your job (e.g. a pencil, a T-Square, a shoe). Draw the Tracing diagram for that object (like Figure 3).
2. Change one of the system requirements for the diagram in 1. Trace that change through the Tracing diagram.
3. Think of one test for each level in the Tracing diagram.
4. Look at your original Tracing diagram. What conversations would you like to have with your customer based on that diagram?
5. What did you learn about the common object that you used for 1.?
6. What changes in the object would require no changes to the Tracing diagram?
7. What changes in the object would require major changes to the Tracing diagram?

Where We are Now

At this point, the systems engineer has the basic techniques needed to build a system on paper. The systems engineer's job is finished.

I caution anyone anxious to put down the text and launch out into systems building projects. Armed with only the preceding material, I jumped into systems engineering projects and I failed miserably.

What I didn't have were the ideas given in the following chapters – the fundamentals of working with people everyday. Those fundamentals truly helped me to be wise.

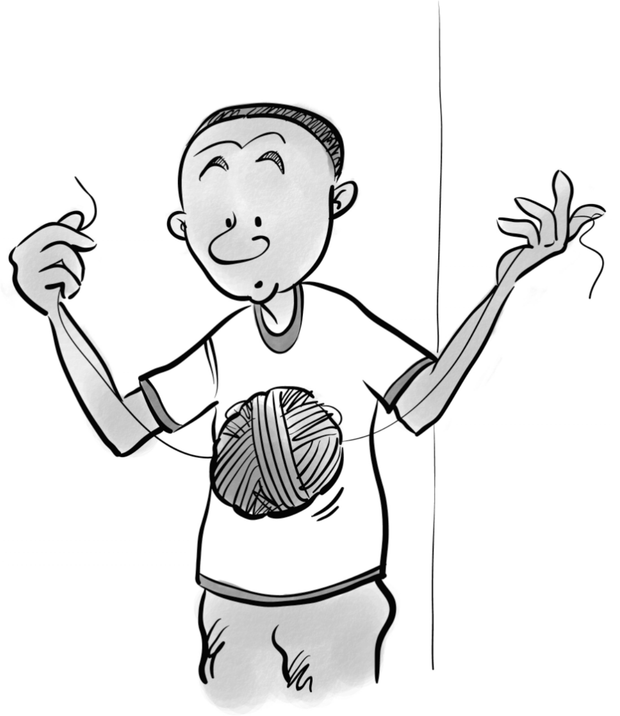
Section 3

Other Things

The previous chapters discussed techniques that the systems engineer should employ to build a system on paper. The following two chapters discuss other things that the systems engineer should do. They differ from the previous techniques in that these are things that the systems engineer does all day everyday. They are not limited to a specific part of the project or any specific order.

6 Questions

Then what happens?



How did this guy get into this predicament? There is some way to go from one end of the string through the ball to the other end. That end-to-end path, however, is, uh, problematic? One way to roll the string into an easier-to-use ball is to keep in mind a fundamental systems engineering question: "then what happens?"

This, and other questions, is important to the systems engineer delivering a ball of string that satisfies the customer. To do so, the systems engineer needs to know what the customer wants. So fundamentally, the systems engineer questions the customers to learn what they want.

But there is more. The systems engineer is also involved in the design (creating the architecture) of the system, building the system, and even explaining the system. The systems engineer needs information from designers, builders, managers, financial backers, and others. That information also comes through questions.

Here is where much of systems engineering collapses.

Systems engineers are born as engineers. Many of us entered engineering because we like to work with problems – dissecting and solving them. Asking questions means working with people; that is different, unfamiliar, uncomfortable – something we often avoid.

Questions

I am a quiet, stay-to-myself person. When a gathering of people ends, I go someplace alone to recharge. A personal puzzle for me is that in spite of this leave-me-alone temperament, I have always liked to ask questions. I have asked many different questions over the years practicing systems engineering. With some help from friends and colleagues, I have reduced my list of questions to something more manageable. The following pages describe and discuss those questions.

The Primary Question

I have what I consider to be the primary question in systems engineering. I learned it from a colleague who learned it from his two-year-old daughter. The question is:

Then what happens?

His daughter frequently asked him this question as they walked about their farm in rural Virginia. A typical exchange would be:

Daughter: What are we doing?

Father: Putting feed in the troughs for the cows.

Daughter: Then what happens?

Father: The cows eat the feed.

Daughter: Then what happens?

Father: The cows digest the feed in their stomachs.

Daughter: Then what happens?

Father: The female cows produce milk in their bodies.

Daughter: Then what happens?

Father: We squeeze the milk out of the cows.

Daughter: Then what happens?

Such is the nature of an inquisitive little girl. What she was doing, in her own precious manner, was asking her father to describe a system. She was modeling what a systems engineer does. The systems engineer learns about the system being built by helping the customer experience the system in their description.

There are two situations in which I ask "then what happens?" The first is like the little girl and her father. I ask, "then what happens?" about the system we are making. The second is asking, "then what happens?" about a process.

First is an example of asking about a system. Refer back to the chapter on interfaces. We defined the interfaces for a system used to water houseplants. The information shown in the architecture and defined in the interfaces came from asking "then what happens?" The exchange could have been:

Customer: I take water from the faucet.

Systems Engineer: Then what happens?

Customer: I usually fill a large ice tea pitcher with water.

Systems Engineer: Then what happens?

Customer: Well, my ice tea pitcher holds a couple of gallons, and I can't carry that around the house. It's too heavy for my hands. My arthritis bothers me part of the year.

Systems Engineer: Then what happens?

Customer: I put the water in a bottle in my housecoat.

Systems Engineer: Then what happens?

Customer: I walk around the house checking my plants and watering the ones that are too dry.

Systems Engineer: Then what happens?

Customer: I water them with the bottle.

Systems Engineer: Then what happens?

Customer: I put my bottle back on the kitchen counter next to the ice tea pitcher for the next day.

Systems Engineer: Then what happens?

Customer: Oh, I usually take a nap after watering my plants.

This question and answer session allows the systems engineer to learn about the system for watering plants. Notice that the customer didn't say, "I need a squirt bottle." The bottle mentioned was small enough to fit in the pocket of the housecoat. The customer implied a squirt capability in the conversation.

Here is an example of asking about a process. One example process is writing this book.

Author: I have an idea for a book on systems engineering.

Systems Engineer: Then what happens?

Author: I sketch the idea in a mind map on a piece of paper.

Systems Engineer: Then what happens?

Author: I type a one-page outline for the book.

Systems Engineer: Then what happens?

Author: Little by little, I expand the book outline until it is five or six pages long.

Systems Engineer: Then what happens?

Author: I start an outline page for each chapter.

Systems Engineer: Then what happens?

Author: I spread the chapter outlines on the bed.

Systems Engineer: Then what happens?

Author: When ideas come to me, I scribble them on the appropriate chapter outline page.

Systems Engineer: Then what happens?

Author: When I am ready, I draft a chapter.

Systems Engineer: Then what happens?

Author: I print that chapter and set it aside for a week.

Systems Engineer: Then what happens?

Author: I go on to drafting another chapter.

Systems Engineer: Then what happens?

Author: I go back to the previous chapter draft, read it, edit it with a pencil, and put the edits into the word processor.

Systems Engineer: Then what happens?

Author: I keep doing this until I have drafted and edited all the chapters.

Systems Engineer: Then what happens?

Author: This goes through to the (hopeful) publication of the book.

Another example is the systems engineering process.

Systems Engineer #1: I start by meeting the customer.

Systems Engineer #2: Then what happens?

Systems Engineer #1: We go through many of the questions in this Questions chapter of the book.

Systems Engineer #2: Then what happens?

Systems Engineer #1: I use some of the thinking tools in the Thinking chapter of this book.

Systems Engineer #2: Then what happens?

Systems Engineer #1: I try sketching a few architectures.

Systems Engineer #2: Then what happens?

Systems Engineer #1: I discuss those with the customer.

The "then what happens?" question helps the systems engineer examine the entire system and apply a little wisdom. This question steps the systems engineer through the system with the customer. The customer reveals parts of the system that the systems engineer would not know otherwise.

As a logical, rational, and caring systems engineer, I usually think that the customer will be happy to go through this question-and-answer description of the system. It does work well, and it is for the customer's benefit. I'm usually wrong. The customer hasn't stepped through this exhaustive and pains-causing interchange of "then what happens?" "then what happens?" "then what happens?"

The customer sometimes screams, "You want to know what happens? I yell at you and tell you to stop asking me 'then what happens?'"

The "then what happens?" question and answer session should be repeated. People wear out during sessions and reach the point of not being able to provide any more information about the system. They do provide plenty of information about their state of mind ("I'm tired of you asking me the same question. Go away."). The systems engineer should be patient with the user. This is a pains-causing experience, and I have never been able to gain the concept of a system in one session.

Give the customer a break. Give yourself a break. Come back the next day or the next week and step through the question again.

I have found that customers often describe a system differently the second time. This different description enables me to understand the system better and better satisfy the customer.

A Second Question

A second question that I ask as a systems engineer is

Are we done?

This question is a check for agreement. I ask it to learn if I have understood what the customer has been saying.

There are several variations to "are we done?" These are:

- Is this everything you want?
- Have I built something that satisfies you?
- Is this it?

The "is this it?" question is part of an exercise known as "that's not it." I have found the "that's not it" exercise to be of great value. A typical example occurred for me several years ago when I was leading a 50-person team in a major effort to change a multi-hundred million-dollar architecture. We had met several times, listened to customers and one another, made lots of notes, and nodded our heads in agreement many times. I was almost certain about what we wanted to do and what we wanted as a new architecture. I confidently stood at the white board, drew the architecture, and turned to the group for affirmation. Everyone shook their heads, "No."

"Great," I said as I drew a big X across my architecture and proclaimed, "That's not it! Let's keep a record of this and not build it."

My drawing was a great success and my idea for an architecture was terribly wrong. We knew the answers to "are we done?" "Is this everything you want?" and "Is this it?" were all an emphatic

NO. We had learned much in little time and were ready to continue with “then what happens?” questions.

Although the “are we done?” and related questions may seem dubious, they are of great value. This is contrary to what I’ve been writing so far. The systems engineer steps through a systematic process to learn and identify a system that satisfies the customer. Now we ask, “is this it?” Surely this *is* it. Surely we *are* done. We are, after all, adults and professionals. How could we possibly have it wrong?

While we are adults, we are adult humans. Humans make mistakes; humans misunderstand one another. We do well when we can hold something in our hands. We struggle often when discussing concepts, and the systems engineer and customer spend much of their time working with concepts.

Expect misunderstandings; expect getting it all wrong; expect proclamations of “that’s not it.” I allow time for this as I have rarely “got it right” the first time.

I append two words to the “are we done?” questions. The two words are “for now.” The questions above become:

- Are we done *for now*?
- Is this everything you want *for now*?
- Have I built something that satisfies you *for now*?
- Is this it *for now*?

I add these two words to my questions after I receive a “yes” reply to the questions.

The exchange is:

> Systems Engineer: Is this it?
>
> Customer: Yes!
>
> Systems Engineer: Is this it for now?
>
> Customer: What do you mean?
>
> Systems Engineer: Is it likely that you will want something more or different in the future?

The answer to the last explanatory question will be “yes” even if the customer says “no.” If I provide a system that satisfies the customer, he will use it. After using the system, the customer will learn of a system that will be more satisfying.

I ask the customer the “for now?” question as a signal that I expect him to desire changes in the future. I acknowledge that the system lacks something that the customer wants but cannot request now. The customer will learn what is lacking now and will be able to describe it later. We will continue to work together.

Two More Questions

I recommend two more questions for the systems engineer. These are:

If you had that, what would that do for you?

How is that a problem for you?

These are variations on the “then what happens?” question. The systems engineer repeatedly asks one of these questions until the customer replies, “I want to stop now.” When the customer says this, the systems engineer stops.

Here is an example.

Customer: I want a digital camera.

Systems Engineer: If you had a digital camera, what would that do for you?

Customer: I could take pictures of my houseplants.

Systems Engineer: If you could take pictures of your house plants, what would that do for you?

Customer: I could e-mail the pictures to my sister-in-law.

Systems Engineer: If you could e-mail the pictures to your sister-in-law, what would that do for you?

Customer: She could tell me if my plants looked too wet.

Systems Engineer: If she could tell me if your plants looked too wet, what would that do for you?

Customer: I would know if I should water my plants.

Systems Engineer: If you knew if you should water your plants, what would that do for you?

Customer: My plants wouldn’t die.

Systems Engineer: If your plants wouldn’t die, what would that do for you?

Customer: Long pause, perplexed expression, no answer.

The systems engineer recognizes this as the signal to stop.

The customer doesn’t require a digital camera. The customer wants (1) his plants to be healthy, (2) to know if his plants need water, and (3) an occasional conversation with his sister-in-law about plants. A little thought and the systems engineer can provide a system to satisfy the customer.

Notice how the customer's original requirement was far from the real requirement. This is often the case. Something has caused the customer to leap from his situation to a solution – a camera. The pains-taking and often pains-causing question-and-answer session helped both the customer and systems engineer find the requirements.

Here is a similar example using the second question.

Systems Engineer: Please tell me about your situation.

Customer: My sister-in-law doesn’t read her e-mail often.

Systems Engineer: How is your sister-in-law not reading her e-mail often a problem for you?

Customer: She doesn't answer my questions soon enough.

Systems Engineer: How is her not answering your questions soon enough a problem for you?

Customer: I need answers quickly.

Systems Engineer: How is your need for quick answers a problem for you?

Customer: My plants are tender and frail.

Systems Engineer: How are tender and frail plants a problem for you?

Customer: My plants will die if I don't get advice from my sister-in-law quickly.

Systems Engineer: How is this (your plants dying) a problem for you?

Customer: Long pause, perplexed expression, no answer.

The systems engineer recognizes this as the signal to stop.

Once again, this repeated question reveals that the customer is concerned about plants dying. The sister-in-law has been the source of information to keep the plants alive. That source of information, however, is sometimes unreliable. The customer desires a system that will make the plant-saving information reliable and readily available.

Questions about Questions

I have four questions that I recommend the systems engineer to consider when asking questions. These are:

(1)What does the systems engineer ask?

(2)When does the systems engineer ask?

(3)Who does the systems engineer ask?

(4)Why does the systems engineer ask?

What Does the Systems Engineer Ask?

I ask questions that provide information. I want to know what the customer does and what will satisfy the customer. I try to avoid asking the customer to justify his profession, his request for my time, and his existence. Justifications usually put people on the defensive. Their breathing shortens, their muscles tense, they either squirm nervously in the chair or freeze. Such limits thinking and communication – two actions that I want the customer to do.

Second, I ask questions to help me understand the customer – not to start a debate or argument. I make requests such as, "I don't understand what you said yesterday about the flow of water through flood gates. Please help me with this."

I try to avoid, "Your statement yesterday about the flow of water through flood gates disagrees with the state commission's report last year. Defend your position."

After understanding the customer, I go to thinking. One aid to thinking at this point is to ask myself, “What three things must be true for the customer to believe what he believes.” (See the following chapter for more about the Rule of Three.)

Customers disagree with one another about their beliefs. Customers disagree with their managers, financial backers, and noted people in their profession. There is something – some fact, feeling, perspective – behind the different belief. What is that something? How is that something influencing this customer? What can I learn from this something?

When Does the Systems Engineer Ask?

The answer to this section’s title is obvious – the systems engineer asks at just the right moment.

The first “right moment” is what educators call the “teachable moment.” This teachable moment is the time when a puzzle is in front of the customer and they are interested in finding the answer. They are struggling to realize something important to them and the system. As the systems engineer, I think of it as the “learn-able” moment. This is the moment when the systems engineer will learn something that is paramount to the customer. In one of the example conversations earlier in this chapter, it is the moment when the systems engineer learns that the customer doesn’t want a digital camera, but instead wants to keep his houseplants alive.

The second “right moment” is when the topic has energy. At this moment people are excited about the subject; they want to discuss it. Let them discuss it. Use people's energy and excitement to gather information.

Another “right moment” to ask the customer a question is when he has the answer. If the customer is in his sunroom watering his plants, ask him about his plants. If the customer is in the store buying plant nutrients, ask him about the plant nutrients he uses. These examples seem obvious, but I have seen people asking customers about fieldwork while they were sitting behind a desk and vice versa.

A key to asking at the right moment is to observe the customer while asking questions. Observing the customer can tell the systems engineer when we have a teachable (learn-able) moment, when the topic has energy, and when the customer is more likely to have the answer to the question.

Observe physical changes in the customer during the conversation. Changes that may indicate the right moment include:

(1)The customer changes from speaking softly and slowly to loudly and quickly.

(2)The customer changes from leaning back in their chair to sitting on the edge of their chair.

(3)The customer changes from arms folded across the chest to making gestures in the air.

(4)The customer changes from eyebrows down and eye lids half closed to raised eyebrows and eyes wide open.

And the change that I notice most at the right moment:

(5)The customer stops breathing (for a moment).

I hope these examples illustrate there are some moments to ask questions that are better than others. Information does not have a schedule. Customers don't become excited, lucid, and articulate at 9 AM on the second Tuesday of the month because that is the time for the monthly meeting. As a systems engineer, plan for information to be unavailable when you expect it and to be available when you don't expect it.

Who Does the Systems Engineer Ask?

The answer to this section's title is not so obvious. The systems engineer asks the right person – the person who is directly involved in the situation. One main reason for asking the person most involved is that the systems engineer will obtain more accurate information. The person who waters the plants everyday is more likely to have information about watering plants. This makes sense. Who would ask the wrong person? I have seen many systems engineers ask the wrong person.

I advise the systems engineer not to go to Mr. Smith and ask, "Mr. Smith, what is Mr. Jones' most needed gardening tool?"

In doing so, the systems engineer is creating a triangle among Mr. Smith, Mr. Jones, and the systems engineer. Information can bounce around inside this triangle and confuse everyone. In addition, the triangle can create animosity. For example,

> Systems Engineer: Mr. Jones, Mr. Smith told me that the lawn mower is your most needed gardening tool.
>
> Mr. Jones: Oh really? So Smith doesn't think I mow my lawn good enough for him. Well, his most needed gardening tool is a paintbrush. Have you seen his house? And what he really needs is a small nose because he had better keep his big nose out of my business!

I have created triangles like the above with items far more costly and serious than lawn mowers. The result was far more costly and serious than a "nosey" neighbor.

A second answer to this section's title question is that I ask anybody related to the system. Often, the systems engineer will ask questions of a person with a title like President of the Anytown Gardening Society and Regional Manager of ACME Gardening Supply Stores. Ask these persons and also ask members of the Gardening Society, gardeners hired by members of the Gardening Society, and checkout clerks and floor workers at the Gardening Supply Stores. Each person has a different perspective of the system. Some persons will have the key piece of information that enables a customer-satisfying system. The systems engineer does not know who that person is before asking questions. Hence, ask all of them.

The systems engineer can ask anybody about the system without creating triangles. Ask persons about the system as it applies to them. Don't ask a person to answer for another person. Don't ask a person to comment on another person's answer. Don't talk or ask about a person who is not present.

Why Does the Systems Engineer Ask?

The primary answer to this section's question is obvious. The systems engineer asks to learn information that enables building a satisfying system.

That answers the general question. There is, however, another question. Does the systems engineer ask a specific question because he wants the answer to that question or because he wants other information? For example, when the systems engineer asks, "Do you want the system to be red, yellow, or green?" he may want the customer to answer "red," "yellow," or "green," but he may want something else. He may want the customer to think about color and other attributes of appearance of the system. I believe that broad thinking by the customer is better than specific thinking, and a question that leads to broad thinking is beneficial.

This leads to a second and probably more important reason why the systems engineer asks questions. The systems engineer wants persons associated with the system to (1) think, (2) be engaged, and (3) own the result. This is why I recommend the oft-irritating question "then what happens?" I want the customer and the other persons to think through the system.

Those irritating questions also engage the customer. The customer feels some unease answering the question over and over. That uneasy emotion stays with them, and they remember that conversation because of the associated emotions.

Persons who think, engage, and have emotions associated with a conversation own the result. They participated, and their knowledge and emotions are part of the resulting system. They will work to have the system succeed. It is not just something that someone else pushed on them.

Questions to Avoid

Now that I have written several thousand words about the value and necessity of questions, allow me to state a warning:

It is easy to upset people when asking questions.

In my younger days, I upset many people while asking questions. I didn't understand why persons were so irritated with me. All I was doing was trying to learn from them so we could build a satisfying system. Why were they so angry with me?

There are key words to avoid when asking questions. The first is "why."

- Why do you do this?
- Why do you do that?
- Why are you mad about these "why" questions?

The word "why" puts people on the defensive. They have to justify themselves. As stated in the previous section, ask questions that lead to information instead of justifications.

Despite its drawbacks, the "why" question is a natural one for me. I'm dying to know the reason behind someone's use of a system – especially when it doesn't make sense to me. Not asking "why, why, why?" is hard for me to avoid. I have found an alternative to "why" and I describe it in the next section on non-question questions.

The second word to avoid is "you."

- Are you listening to me?
- Are you alright?
- Do you often have difficulty with questions?
- Why are you turning red? (has both "you" and "why")

The word “you” points at the other person.

“You asked me to come here.” (Hint – you must need me to help you do your job. You must not be very smart.)

“You need a new system.” (Hint – you aren’t able to do your job with your current system. You must not know how to use it properly.)

The word “you” often comes across as blaming.

“You want a system that is easier to use.” (Hint – if you would just learn how to use the current system, we wouldn’t have to waste time on a new one.)

Instead of “you,” try to use words such as “we,” “us,” and “our.” These shift the emphasis to the working relationship between the systems engineer and the customer. Examples that counter the above “you” statements include:

- How is our conversation? – NOT Are you listening to me?
- How are we doing? – NOT Are you alright?
- We need a new system. – NOT You need a new system.
- We can create a system that is easier for all of us – NOT You want a system that is easier to use.

Another word to avoid is “problem.” “Problem” questions are similar to “why” questions in that they put people on the defensive. They defend or justify themselves instead of thinking about the system. Systems engineers often tell people that they have problems and that ruins the search for information. Example information killers include:

- What is the problem with the system?
- How is that a problem?
- What is the difficulty here? (synonyms of “problem” are also to be avoided)
- What are the issues here? (another synonym)

Instead of using the word “problem,” seek words that imply less fault on anyone’s part. Some words to use instead are “situation,” “our place,” and “where we are.”

- What is our situation here?
- Please describe our work here.

You've probably noticed that I've contradicted myself. Earlier in this chapter I recommended asking “How is that a problem for you?” and now I recommend not using the word “problem.” The earlier question has been a powerful information-producing one for me. Use it with care and rephrase it if necessary to “How does that bring you extra work?” or “How does that change your day?”

A final word to avoid is “help.” If the systems engineer can help a customer, that implies that the systems engineer is superior to the customer. For example, How may I help you? (Hint – you obviously need a superior person like me around to do your job.)

Men especially dislike being helped. That implies that you have a problem, and I can fix it for you. Men solve problems; we don't have problems, and we don't need help.

Instead of "helping" the customer, try to speak in terms of working with the customer. For example, "How can we work together on this?"

I cannot provide a complete list of words that do not offend or imply fault. Some persons will have a negative reaction to the word "situation" while being fine with "issue." In general, notice the person's reaction to questions and statements. If the customer reacts to a word, ask him about that word. Explain that you are not implying fault and find a word that works for that customer.

In addition to avoiding particular words, try to avoid labels and interpretations. Stay with facts in what you say. This may sound simple, but labels and interpretations are so frequently used in our culture today that it is easy to slip into them.

Fact: This system has 5 subsystems spread across 20 cities operated by 200 people.

Label and Interpretation: You have a large, complex system.

The words "large" and "complex" are my labels based on the interpretations of the facts. Calling the system "yours" is also an interpretation. The customer may hear criticism in these labels and interpretations. Here are several more examples:

Fact: I may have to work 80 hours on this.

Label and Interpretation: This looks difficult.

Fact: This session lasted 10 minutes. The average session lasts 20 minutes.

Label and Interpretation: You're easy to work with.

Fact: I need the approval of people in three different organizations.

Label and Interpretation: This could be tricky politically.

A final item to avoid in questions is using questions as hints. For example,

Question: Why do you use a cell phone? Why don't you use a regular phone?

Hint: Use regular phones and you won't need a new system.

If the systems engineer wants the customer to use a regular phone, say, "Using a regular phone instead of a cell phone will cut the system cost in half and also cut in half the training time for users. Is this something we can try?"

Non-Question Questions

I end this chapter on questions with a method that has worked best for me. It is a non-question question. I use an "I" statement followed by a request. For example,

"I don't understand how that will contribute to the system. Please help me understand."

Starting with "I" helps me to avoid a couple of potential hazards. First, I am not using the blaming "you" statement. I am also avoiding the justification-seeking "why" question.

The "I" statement is an honest one about the subject and me. I am expressing what I know about the subject and my relationship to the subject.

The request is an attempt to gain information from the customer. The request always begins with "please" (a smile also helps) and proceeds to the request. In the request, I avoid "why," "you," "problem," "help," and other words that put the customer into the defensive mode.

Here are some examples of rephrasing harmful questions.

Harmful: You didn't hear my question.

Not so Harmful: I am not sure I stated my question well. Please repeat it back to me.

Harmful: How can I help you with your problem?

Not so harmful: I want to work on the current situation. Please tell me how we can work together.

Harmful: Why did you do that?

Not so Harmful: I don't understand. Please describe what just happened.

Concluding Thoughts

I'm a seeking person. I always wanted to know about things – how they worked, why they worked, how they fit into the world. People – what they do, the systems they use, the system they are apart of – fascinate me. Where did they originate? How did they move from their origin to this place on this day?

I satisfied my seeking by asking questions. As a kid, I asked my fair share of questions. I wasn't, however, as outspoken in my curiosity as the farmer's daughter and her "then what happens?" sessions. I was quieter – so I spent much of my time wondering about the puzzles I saw. I invented answers to my unspoken questions.

Then I dabbled in systems engineering. Most of my invented answers were wrong – no surprise in retrospect. Also no surprise was that my answers led to customers dissatisfied with systems.

I learned a little and started asking my questions out loud. The pages of this chapter that describe how *not* to ask questions give an idea of what I did during my first years of asking questions. In addition to systems that didn't satisfy customers, the customers were angry with me.

I learned much about how to ask questions. The pages of this chapter that describe how to ask questions are the result of those lessons. I am grateful to many people who helped me through those lessons.

In conclusion, I recommend

1. Ask a few questions with the best intentions.
2. Observe the customer's reactions to the questions. Do your questions enable or inhibit the flow of information?

3. Alter your questions per your observations.
4. Return a few days later and ask a few more questions.
5. Repeat the above.

Suggested Exercises

1. Write three “then what happens” exchanges that might occur between you and your customer.
2. Think of three projects that didn't go as well as you wanted. How might the “are we done?” question have helped?
3. Repeat exercise 1. using the “if you had that, what would that do for you?” question.
4. Repeat exercise 1. using the “how is that a problem for you?” question.
5. Describe three important “learn-able moments” you have experienced.
6. List three words that you have used in questions that put a customer on the defensive and stopped the exchange of information.
7. For the three words of exercise 6., what questions could you have used instead?

Where We are Now

The systems engineer works with two items: (1) the customer and (2) information from the customer.

The information has two of its own aspects: (1) obtaining information and (2) using information.

The previous chapter on questions discussed how to obtain information. The succeeding chapter discusses how to use the information – by thinking.

7 Thinking

The obvious choice for a cartoon for this chapter would be a drawing of Rodin's The Thinker. Too obvious. Somewhere I got the idea of Dobie Gillis standing in front of the statue of The Thinker in Central City's Park on the old TV show "The Many Loves of Dobie Gillis." That show *is old* as I only saw it a few times in reruns. In each episode, Dobie (played by Dwayne Hickman – us Dwayne's have to stick together) would stand in front of The Thinker, face the camera, and talk to the audience at home. This was Dobie's thinking time.

The systems engineer needs to find his Thinker statue in the park – a time and place to think.

Creating a system that satisfies the customer is the primary responsibility of the systems engineer. That task requires thought. Hence, the systems engineer needs to do a lot of thinking. In addition, the systems engineer monitors the thinking of the people involved with the system.

On the following pages I describe some of the thinking techniques that have helped me. I don't cover every technique I have experienced. I hope that you find some of these useful. What is more important is that I hope these spur you to think about your thinking.

Decisions, Decisions

The systems engineer has to make decisions. Decide which is the best path through the architecture drawing. Decide what type of interface we should use. Decide who best knows the requirements for the system we are considering. These are a few of the fundamental decisions facing the systems engineer.

The first item to recognize is that decisions just don't happen by themselves. As much as the systems engineer uses tools in systems engineering (like architectures, interfaces, and requirements), these are just tools. Tools don't make decisions. The tools present information to help better examine the situation. People have to decide and people have to live with the consequences of their decisions. If at any time in providing a system I observe a decision but cannot connect people to that decision, I should bring that to everyone's attention. If we cannot find the "deciders," we should decide that question again.

Decisions are not permanent. I can change my mind and decide to go another way at a future date. Changing my decision is not an admission of failure. It is acknowledging that something has changed in what I know. I have learned something and I can make a better decision.

To help in changing decisions, I recommend documenting decisions. Record each decision on one page of paper (or one unit of storage in a computer tool). For each decision, record:

(1) The situation

(2) The alternatives

(3) The decision

(4) Who decided

(5) The reasons behind the decision

I feel that (5) is particularly important. I make decisions based on the situation as I know it today. If I learn something pertinent to this situation, I can go back to the decision's page, look at what I did and why I did it, and consider if my new knowledge will cause me to change my mind. Situations change. For example, the relative prices of items in an architecture fluctuate. These price changes can make a discarded alternative more attractive than a chosen one. That is a great time to change my decision.

One alternative to any decision should be "I won't decide today." Write that alternative on your decision page. Choosing "I won't decide today" is not being indecisive – that is far from it. Choosing this is stating that "information for this decision is not yet available" or "we don't have to decide this yet. We can wait a while." Rushed decisions are often worse than delayed decisions. As the systems engineer, determine when a decision is needed.

While discussing decisions, let me pass along one decision-making tool: the coin flip. I don't recommend doing something if the coin is heads and something else if the coin is tails. I do recommend flipping the coin when forced to choose between two items. Tell yourself, "Heads means item A and tails means item B." Now flip the coin. While the coin is in the air, you will usually feel in your heart which item you want. Catch the coin, don't look at it, and go with what tugged at your heart while the coin was in the air.

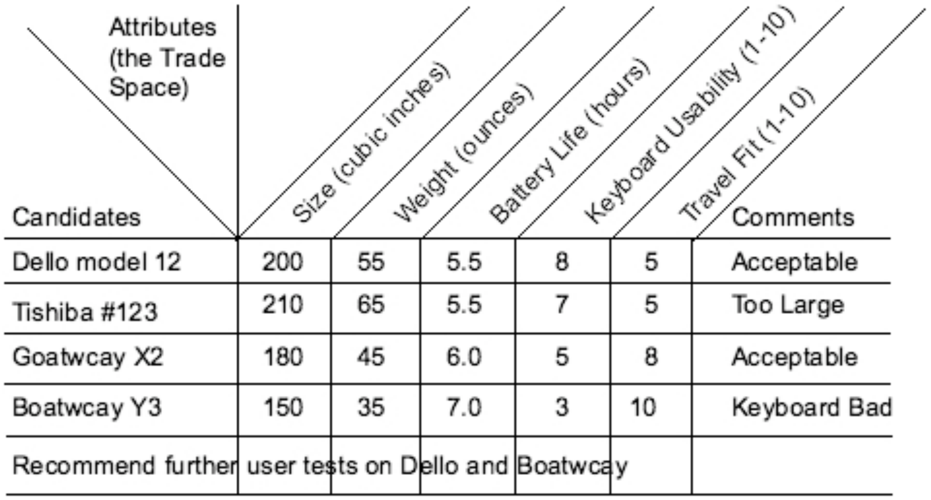
Trade Studies – A Decision-Making Tool

The systems engineer decides among items in an architecture. A thinking tool for these decisions is the trade study. In a trade study, the systems engineer examines possible alternatives for an item and assigns a score or value to each alternative. At the end of a trade study, the systems engineer can recommend an item to put into an architecture.

The systems engineer should recommend the best alternative for the item. “Best” is the key term here. Best is determined by what is in the trade space with the trade space being those attributes that the customer considers important. Those attributes are expressed in the customer’s requirements.

A simple tool to express the information found during a trade study is a table. The table lists the alternative items or candidates down one side and the attributes or the trade space across the top. Inside the table are values that represent an alternative’s scores for each attribute. Also in the table are recommendations for which item to choose or any further study that should occur.

Let’s illustrate a trade study by considering a laptop computer. Figure 1 shows a trade study table for this example. Down the left side are listed the candidates for the computer. Across the top are the attributes or the trade space for each candidate. For this example, the customer’s requirements indicate that the laptop computer is desired to be small, lightweight, have a long battery life, have a good keyboard for typing, and is handy when traveling.



Attributes (the Trade Space) Candidates	Size (cubic inches)	Weight (ounces)	Battery Life (hours)	Keyboard Usability (1-10)	Travel Fit (1-10)	Comments
Dello model 12	200	55	5.5	8	5	Acceptable
Tishiba #123	210	65	5.5	7	5	Too Large
Goatwcay X2	180	45	6.0	5	8	Acceptable
Boatwcay Y3	150	35	7.0	3	10	Keyboard Bad
Recommend further user tests on Dello and Boatwcay						

Figure 1 – A Table Showing the Results of a Trade Study

The items listed across the top of the table represent the trade space. These are the items that are important to the customer, the things the customer wants the systems engineer to consider, i.e. the customer's requirements. Take care when learning the requirements as they are to be used here and in other important aspects of systems engineering.

The numbers inside the table represent what was learned during the trade study. Some of the numbers are measured (Size and Weight). One number (Battery Life) was measured, but the measurement depends on what tests the systems engineer used while measuring (watching a movie or just typing). Several of the numbers are subjective (Keyboard Usability and Travel Fit).

To the far right of the table are comments that summarize the trade study. The “winner” is rarely obvious as the winner is based upon the scores in the middle of the table, and these scores are mostly based upon someone’s opinion.

The trade study provides information; the systems engineer provides the decision.

I urge the systems engineer to do as much testing and as little reading as possible during trade studies. Many trade studies are conducted by reading a manufacturer's spec sheet. I can read to learn about the size and weight of a laptop computer. Often, however, manufacturers issue spec sheets months before a product is available. Seeing the item is much better than reading about it. When I see the item, I know it is real and not just a marketing brochure.

Better than seeing the item is touching it. I find that many systems engineers omit this step. We try to be intellectuals – thinkers – and sometimes stress that aspect so much that we neglect to actually put our hands on an item that is before us.

Finally, test the item. I have experienced many cases in which testing showed that a spec sheet was incorrect. Sometimes the simplest items like how much a laptop computer weighs differs from its spec sheet. Scores in the trade study table based on testing are far more reliable than those gathered via other means.

Thinking Tools

The remainder of this chapter will describe tools that have helped me think. Some are commonplace but not commonly used much anymore (pencil and paper). Others are used more often today than before, but often misused (peer reviews). Consider each and remember that tools are just tools – something to use in the right situation. The systems engineer has to decide which tool is right for which situation.

The Rule of Three

The Rule of Three is a meta-tool or a tool that helps with tools. When using any thinking tool think of three examples, three reasons, three of whatever you are considering. If I cannot think of three, I probably haven't thought enough.

For example, when I feel that I have a design for a system, i.e. I've drawn an architecture and selected a path through it, I should think of three things wrong with that design. At first, that sounds strange, but surely there is something wrong with my design, and if there is something wrong with it there are probably three things wrong with it. Finding three things wrong with my design may cause me to alter the design. Those three wrongs may cause me to stay with it as it is. There may be 33 things wrong with my next best design, so having a design with only three things wrong is great.

True to be False

This tool is for use when a group of people has reached consensus on a decision. When a group has agreed on X, the group should ask, "What must be true for X to be false?" (Be the wrong decision.) This question helps the group find the faults in its thinking.

Adding in the Rule of Three, the group should ask, "What three things must be true for X to be false?"

Consider the laptop computer example. The group decides to use a Dello laptop computer because it is light enough to carry on trips yet still has a large enough keyboard for comfortable and fast typing.

The question is, "What three things must be true for the Dello to be a bad choice?"

Some answers include:

(1) The user could be so large physically that the Dello keyboard is too small.

(2) The user could have physical ailments that prove the Dello to be too heavy.

(3) The user may never move the Dello from their desk and not care about ease of carrying.

This "true to be false" question and these three answers can help me to reconsider the requirements I used in the trade study. I could talk with the customer more and learn the likelihood that these conditions exist. I may change my decision, or may conclude that the Dello was the correct choice.

The Difference that Makes a Difference

There are many attributes in the trade space. These attributes help the systems engineer see the deciding difference among the different items. I have often found there to be many differences among the different items. I have also found that these many differences only confuse me.

What I am seeking is the difference that makes a difference – the one or two differences that will clearly separate the items under consideration. This is the information to record on the decision-documenting page. This is the point that explains architecture and item choices to customers, managers, and financiers.

Applying the Rule of Three, find the three points that clearly separate the items under consideration. Another use of the Rule of Three is, find three reasons why there is only one point that clearly separates the items under consideration.

Paper

Paper is an excellent thinking tool that is being neglected more and more each day. When I mention paper, I include white boards, chalkboards, flip charts, and other physical writing surfaces.

These physical surfaces force me to write with my hands. I have yet to find a substitute for physically writing my thoughts and seeing them in front of me. Thoughts run through my mind, these thoughts direct my hand, and when I see the thoughts on paper I think about them again.

I love to use computers for almost everything these days; most systems engineers do. Nevertheless, I also pause and scribble physically.

Consider the systems engineering architecture figures. I recommend drawing these on a white board, gathering a group of people around the board, and working through the different paths physically as a group. Once all the work and thinking are finished, have someone enter the results into a computer.

Think first on paper; compute second.

Applying the Rule of Three, find three different ways other than the computer to express thoughts.

Something Wrong

No one is perfect, and no technique is perfect. Every systems engineer I have met has made mistakes. Every tool I have used has enabled me to make mistakes. If there is nothing wrong with the decision a systems engineer has made, the systems engineer hasn't thought about the decision enough.

Applying the Rule of Three, what are three things wrong with this decision?

Choice

One concept I hope to have expressed in this book is that there are many choices available in systems engineering. Given all these choices, I still hear systems engineers and customers say, "Well, we had no choice here. We were forced to do this." Such "no choice" statements should squeal as a siren to a systems engineer.

I once worked with a customer who had his mind set on the type of system that he needed. Only one system would do. We proposed a completely different type of system, but the customer repeated, "We cannot use that other system. We have no choice."

The customer was saying, "We haven't thought of a way to use that other system."

Now we had something to work with. We had to think of ways the customer could use the system we were proposing.

Interpret the "we have no choice" statement as "we haven't yet thought about this."

Applying the Rule of Three, think of three new ways that you have never used the system. Think of three ways you could use this other system.

Time

I am busy in my job. Most of the customers I have worked with have also been quite busy in their jobs. We rush from one thing to another every day.

Who has time to think?

As the person responsible for thought, the systems engineer must make the time to think. I recommend that the systems engineer set aside one quarter of the day for thinking. Most people regard this advice as pure folly. I have had jobs where such thinking time seemed to be folly. There was no way I could set aside time. I had no choice over my daily activities. Oops. There was the "no choice" statement. What happened was that I had not thought of ways to create time for thinking.

Applying the Rule of Three, find three activities in the day that can be eliminated and replaced by thinking time.

Picking Up Trash

I recommend that the systems engineer have some period of time each day to think alone. No interruptions, no conversations, and no one else interacting with you. Just some time (a part of the quarter time mentioned above) when the systems engineer is alone in thought.

Several years ago I had a busy job where I struggled to find time alone to think. I found time to think by acting as the soft drink boy. Our office had a small refrigerator that held soft drinks for employees. Someone had to restock the refrigerator and put the warm soft drinks in the back and the cold ones in the front. I did the restocking and shuffling task a couple of times each day. That required about 30 minutes.

Everyone in the office knew that the refrigerator needed to be restocked. Almost everyone in the office ignored this menial task. They were a bit embarrassed to see me doing it, so embarrassed that they didn't talk to me while I restocked the refrigerator. Ah, they wouldn't talk to me. I had uninterrupted time to think.

I told this story to a friend who immediately identified with it. Years before he worked as a professor in a university. He too struggled to find time alone to think. His solution was to walk about the campus and pick up litter. No one interrupted him. He would pick up litter until he had enough quiet thinking time and return to his busy, noisy office.

Picking up trash and restocking the refrigerator are tasks that everyone knows need to be done, most are embarrassed that they don't do them, and they will leave you alone while you do them.

Applying the Rule of Three, find three menial tasks that people are embarrassed because they don't do them. Do these tasks to create solitary thinking time.

Reviews

Reviews are a special time, place, and gathering of people for thinking. A review is a meeting where people state what they believe to be true about a system and a project. People listen, talk, think, and ensure that they are all talking about the same thing. The tire swing example shown in the chapter on requirements is a classic case of why systems engineers hold reviews.

There are many types of reviews. The book "Handbook of Walkthroughs, Inspections, and Technical Reviews" [Freedman] describes several types and how to conduct them. The book "Software Inspection" [Gilb] discusses reviews that are peculiar to the software field. I like both of these books and refer to them often when holding a review.

A part of any review is asking questions. There are questions that help make visible what other people are thinking. There are also ways to ask questions that elicit thinking. It is unfortunate, but there are many more ways to ask questions that elicit fighting, arguing, and no thinking. Please take care and read the previous chapter about Questions.

A basic request at a review is, "I want to know where you are in this project. Please tell me." That request establishes the current context or state of a project.

The next request points to the rest of the project. It is, "I want to know how we go from here to the end of the project. Please tell me."

These two requests, stated in this manner, help to establish the current and future state of a project. As simple and basic as that sounds, I have been in several large projects costing tens of millions of dollars where people could not state those items.

There are special reviews known as milestones. These occur at points in a project where it is imperative that everyone has a common understanding of the system being provided. In the strict systems engineering sense, there are several milestone reviews whose names have become part of the slang of the field. These include:

(1) Preliminary Design Review or PDR

(2) Critical Design Review or CDR

These terms are so commonplace that people speak of a project in these terms. I often hear, "We are just past PDR," or "we are about to CDR the project."

At PDR, people meet and state what they believe the requirements for the system to be. At most PDRs, people agree on 80% of the requirements, but learn that there are some misunderstandings that need to be clarified.

At CDR, the people who are to build the system state their basic design and how that design will meet the requirements agreed to at PDR. The design engineers show their design. The systems engineers show how the design meets the requirements.

One concept of a milestone is that the project cannot progress until everyone agrees on these matters. In the case of the PDR and CDR, the project halts until everyone agrees on the requirements of the system and how the design of the system will meet those requirements respectively.

Milestones can be helpful in providing a system because they help focus thinking. People understand that future work will not occur until after the milestone is passed. This "we will meet, think, and decide to keep working or stop" occasion spurs concentration. In my experience, people want to pass milestones, so they concentrate more, practice their thoughts, and practice how they express their thoughts.

Applying the Rule of Three, think of three things you use now that you consider to be reviews. Also, think of three points in time that would be good for reviews.

There is one type of systems engineering review that I disdain and discourage. That is the DDR or Dinner Design Review. The DDR occurs after a long day of engineering meetings. People go to dinner and discuss the system just a little bit more. Eight hours of discussion wasn't enough, so now, with the aid of fatigue, too much food, and a little alcohol, some people want to make crucial decisions.

DDRs are an excellent place for poor thinking, so please avoid them.

Applying the Rule of Three, think of three "reviews" that you now use that result in poor thinking.

Time Out

An anti-review is the time out. People stop and think during a time out. I call the time out an anti-review because reviews are driven by events. "We are ready to discuss requirements, trade studies, or designs, so let's hold a review." Times out are driven only by time. I recommend the systems engineer have the job of calling the time out at regular time intervals.

People think and have conversations at time outs. Thought provoking questions include:

(1) What are the top items we should be thinking about now?

(2) What are the things we are thinking about now?

(3) What puzzles you now?

(4) What do you hope will happen soon?

(5) Who do you want to walk in the door now and help us?

(6) When do you think something big (good or bad) will happen?

(7) Where would you most want to be now?

(8) How are we thinking?

Apply the Rule of Three: What are three other questions you would think about during a time out? What are three things besides events (milestones) and time (time out) that would spur a thinking meeting?

Closing Thoughts

Providing systems that satisfy a customer is a difficult endeavor. I've often wanted to build systems that were "no brain'rs," but I haven't encountered any. It seems that good systems engineering requires brain'rs (thinking).

I have many systems engineering failures in my past. In hindsight, the failures occurred when I was busy with something. That busy-ness took away thinking.

As the systems engineer, think about your thinking. More important, think about how everyone else is thinking. The other people involved are present because of their brains, i.e. their ability to think. Is it worth my thoughts to ensure that they are contributing their thoughts? The answer to that question is truly a "no brain'r."

Suggested Exercises

1. Describe a past project where you spent *too much* time using a computer and not enough time thinking. What would indicate that in future projects?
2. Describe a past project where you were too decisive, i.e. you made decisions early, refused to reconsider them, and suffered because of it.
3. Describe a past project where you were too indecisive, i.e. you delayed decisions too long or changed your decisions too often and suffered because of it.
4. Consider a decision you are making on a current project. Draw a Trade Study table for this decision.
5. Think of three "thinking tools" you use that are not mentioned in this chapter.
6. Think of three ways you could use each of the thinking tools described in this chapter.

References:

[Freedman] "Handbook of Walkthroughs, Inspections, and Technical Reviews," Daniel P. Freedman, Gerald M. Weinberg, Dorset House Publishing, 1990.

[Gilb] "Software Inspection," Tom Gilb, Dorothy Graham, Addison-Wesley, 1993.

Where We are Now

The systems engineer now has all the basics to examine an entire system and apply wisdom on the way to providing satisfying systems. I have covered the techniques as a process and added a couple of other techniques to use everyday.

Systems engineering is simple, but not easy. Just follow the steps wisely and don't make any mistakes. Well, okay you will make some mistakes. Keep thinking, hold reviews, and call time out regularly. These things will help you catch and correct your mistakes sooner.

Section 4

General Systems Thinking

The final section of this book comprises one chapter on the subject of General Systems Thinking. I have not seen a book on systems engineering that discusses this topic. I include it because of its importance to systems engineering and systems engineers. Studying this topic has helped me immensely as a systems engineer. What may seem strange is that it has helped me most in working with the people involved in systems – the customers and the other engineers. I hope you gain something from this section. I wish that you study the topic further on your own. The benefits have far outweighed the effort for me.

8 General Systems Thinking

I began my professional career as a technician. I moved to being a programmer and from there to being a software engineer. I later delved into systems engineering. My quest for higher levels of abstraction led me to a field called General Systems Thinking. I didn't know what General Systems Thinking was, but I had heard it mentioned a few times in seminars and books.

My study of the topic has led me to this chapter. I included this chapter on General Systems Thinking in this book on systems engineering because:

(1) Systems engineering is a subset of General Systems Thinking. Hence, a background material for a systems engineer.

(2) The principles have helped me as a systems engineer.

What is General Systems Thinking?

The generally accepted father of General Systems Thinking is the late Ludwig Von Bertalanffy. See [Davidson] for Bertalanffy's story and [Bertalanffy] for some descriptions of General Systems Thinking.

I will explain General Systems Thinking with an example. In the 1980s, the University of Nebraska was a national powerhouse in college football having won several national championships the decade before. Nebraska played football in the Midwest during late fall and early winter. The weather was often brutal. Nebraska played a style of football optimized for this weather and recruited players who fit this style. Nebraska dominated play in the Big Eight conference.

At the same time, Florida State University (FSU) was a rising football power. FSU, however, didn't command the same respect in college football as Nebraska. To schedule top teams and gain notoriety, FSU played many games on the road – all over the country. FSU couldn't optimize its style of play and players for the warm Florida weather.

Eventually, Nebraska and FSU would meet in major bowl games in January in places like Arizona and Florida. Nebraska, its team optimized for harsh Midwest weather, faltered in warm weather. FSU, its team built to adapt to different types of weather, won most of these contests.

This illustrates several principles from General Systems Thinking:

1. A system (in this case a football team) that is optimized for one situation, does not adapt well to other situations.
2. A system (another football team) that is not optimized for one situation retains the ability to adapt to other situations.

These two principles apply to systems in many fields far from college football. For example, the Kodak corporation was specialized to be the best chemical-based photography company in the world. This specialization didn't allow them to adapt to digital photography quickly. Another example is the retailer Wal-Mart. They did not specialize in selling one type of product. Wal-Mart sells whatever the consumer wishes to buy. This allows Wal-Mart to adapt to changing markets and consumers' tastes.

Let's move from football and business to biology to consider an engineered strain of corn. This corn is optimized for one type of climate and growing conditions. If those conditions change, the entire strain of corn will die because it cannot adapt.

Now consider computer architecture (my formal background). In the past 30 years, various researchers have set out to build custom processors optimized for small problem sets. With few exceptions, the general-purpose processors from chipmakers like Intel provided greater performance and became available before the specialized processors. The chip-making technology moved quickly. Trying to make a specialized processor in a rapidly changing environment violated the General Systems Thinking principles.

The above two principles apply to college football, business, biology, and computer architecture as well as other fields. That is the contribution of General Systems Thinking: finding principles from specific fields that apply to many fields.

A study of General Systems Thinking will show many similar examples. Founders of the field came from biology, mathematics, economics, psychology, and computing. Practitioners come from these any many more specialized fields. My personal mentor in General Systems Thinking is Gerald M. “Jerry” Weinberg. The two principles given above are from one of Weinberg's books, and I quote Weinberg in many places in this and other chapters.

The influence of Weinberg is similar to that of other General Systems Thinkers. Weinberg is able to provide insight and instruction to people who work in vastly different fields. His lessons apply to medical doctors, computer programmers, government bureaucrats (like me), economists, biologists, and others.

I link General Systems Thinking and systems engineering with a few definitions. Here are two.

> **Systems Analysis** – Generally synonymous with Operations Research. The interdisciplinary search for more efficient ways of using existing talent and technology to improve a system.
>
> **Systems Engineering** – Similar to and sometimes indistinguishable from Systems Analysis. But this approach is more likely to consider the need for a system's fundamental redesign and replacement.
>
> [Checkland]

My twist on these definitions is that

> *Systems engineering is the interdisciplinary search for more efficient ways of using existing talent and technology to replace a system.*

I emphasize two words to tie General Systems Thinking to systems engineering. The first is "interdisciplinary." Let's apply principles from many fields (biology to economics to game theory) to the situation at hand to produce a system that satisfies a customer.

I recommend the interdisciplinary approach from personal experiences (emphasis added):

> What has worked best when dealing with *unmastered complexity* is a combination of:
>
> 1. Learning from analogous situations outside the present situation.
> 2. Learning how people think and combining that thinking with facts and preconceptions to determine action.
>
> [Weinberg 1988]

Working as a systems engineer has been "dealing with unmastered complexity."

The second word from my twist on the previous definitions is "replace" as in replace a system. I have never provided an original system to a customer. Every system has replaced an existing one. Often, the customer was doing everything by hand – like the potato farmers in the first chapter who sorted their potatoes by hand. Doing everything by hand is a system – one without machines, but still a system.

Flashbacks

Previous chapters in this book have already discussed some of the principle themes of General Systems Thinking.

Conversations

The systems engineer has conversations with other people. These other people include the customers who will use the system and the other engineers building the system. Kenneth E. Boulding ([Boulding] and many other places) wrote and taught about the value of the conversation.

> This is the characteristic that distinguishes man from the lower organisms – the art of conversation or discourse.
>
> [Boulding]

Boulding described knowledge using the word "image" and conversation or discourse as delivering "messages" that modified the image or our knowledge. Boulding believed that the ability of people to grow the knowledge in conversation is one of man's great attributes.

The systems engineer works with knowledge – knowledge from the customer and from other engineers. Much of the knowledge comes not through texts or specifications, but via conversation. As I stated earlier in the text, this is where much of systems engineering breaks down – the inability or unwillingness to engage in conversation for the exchange of knowledge.

Trade Offs

The systems engineer uses trade offs when deciding which items go into the sections of an architecture. Weinberg provides an excellent statement of General System Thinking's view of trade offs:

> Because trade offs are universal, we can often use the trade off concept to reason backwards from effects to causes. The reasoning process goes like this:
>
> 1. We are doing X instead of Y.
> 2. We must be getting a benefit from X – what is it?

3. We must be losing something from not doing Y – what is it?

> [Weinberg 1988]

There are no perfect trade offs where choice X is better than choice Y in all respects. If I find myself stating that X is better than Y in all areas, I am not considering *all* the areas; I am missing something and should think again. Perhaps X is better than Y in the important areas, and Y is only better than X in a few areas that don't matter in this situation. If that is true, good – choose X. I, however, have experienced this in only a few cases. Going back through the process has always shown me attributes that I wasn't considering.

Thinking

The systems engineer is responsible for the thinking in providing a system.

> The fallacy of typological thinking, as with all illusions, lies not in the explanations themselves, but in the failure to consider other explanations.
>
> [Weinberg and Weinberg 1988]

The problem with thinking lies not in the thinking, but rather when the systems engineer *stops* thinking. Problems ensue when the systems engineer stops considering other alternatives.

This begs the question: can I ever stop thinking? Part of the answer is that I *will* stop thinking. I don't think much while I sleep and I tend to spend about one third of my life sleeping. (I have met many systems thinkers who claim to think and solve problems while sleeping. I do not doubt their word, and while I envy them, I have never had this good fortune.) A better phrase of the question is, *should* I ever stop thinking?

The answer that keeps returning to me is that as a systems engineer I should always be thinking; I should always keep considering other alternatives. The "Rule of Three" given in the chapter on thinking is one tool to help me continue thinking. Another form of that rule is:

> Think of three reasons (questions, problems, concepts, alternatives) for anything. After that, apply the Rule of Three repeatedly.

Questions

I've been a better systems engineer by asking questions of others and myself.

> We have a curious capacity for giving ourselves examinations. We know how to write the questions that we have answers for.
>
> [Boulding]

Boulding's statement causes me to consider several things. When I am struggling to understand something or find a course of action, I start asking myself questions. If I can find the right question, I soon have the answer and can move forward from my predicament.

My questions, however, are limited by my imagination. Boulding says that I only ask questions for which I have answers. My knowledge of the system limits the questions I can ask myself. I should turn to other people who have questions that I don't. Perhaps my questions of them will cause them to ask their own questions and provide additional answers.

Asking questions that cause other people to think isn't easy. What I know of this I learned from the best "question asker" I have ever met – Jerry Weinberg.

> A meta-question is a question that directly or indirectly produces a question for an answer.
>
> [Weinberg 1988]

Weinberg loves to ask meta-questions. Perhaps because they cause the other person to respond with a question – a question that will give the other person the answer they are seeking.

Weinberg once taught college classes. His favorite test questions were:

1. Write a question that would be an appropriate question for an examination in this course at this time.
2. Answer the question you wrote in part 1.

[Weinberg 1988]

Weinberg's approach to writing practice (from one of his seminars on writing) was a paraphrase of this:

1. Decide what would be the best writing exercise for you.
2. Do 1.
3. Do 1. and 2. for the rest of your life.

I extrapolate these questions to systems engineering as:

1. What kind of system would satisfy your customer?
2. Design the system from 1.

and

1. If you were an excellent systems engineer, what questions would you ask the customer and your fellow systems engineers?
2. Ask the questions from 1.

General System Thinking Principles for the Systems Engineer

This section contains some concepts from General Systems Thinking that have helped me as a systems engineer. Let's begin by highlighting some principles that can enable the systems engineer with a definition of systems thinking.

> **Systems thinking**: An epistemology (theory) which, when applied to human activity is based upon the four basic ideas: (1) Emergence, (2) Hierarchy, (3) Communication, and (4) Control as characteristics of systems.
>
> [Checkland]

Emergence is often overlooked by systems engineers (those who seek to arrange a system) and managers (those who seek to arrange a system of people).

> When applied to natural or designed systems the crucial characteristic is the emergent properties of the whole.
>
> [Checkland]

Emergence is about the relationships that emerge among the parts of a system when the parts combine. I illustrate emergence to systems engineers with a simple exercise. I give separate teams some parts and ask them to build something. Teams have built everything from decorative jewelry to robots. The parts I gave to each team were the same. The systems were vastly different because of how the teams related the parts. The different relationships emerged.

Consider a table, chair, and the floor as parts of a system. I always see the table *next-to* the chair with both the table and chair *upright-on* the floor. The relationships are *next-to* and *upright-on*. Now consider the table *upright-on* the floor and the chair *upright-on-top-of* the table. That is a very different system comprising the same parts. The difference between the systems is the relationships that emerged.

Now consider a group of people as a system. Some groups of people become highly efficient, effective, and fun teams, i.e. they jell. Other groups of people self-destruct in that they accomplish nothing except creating ill feelings towards one another. The difference is the same as the differences with the chair, table, and floor mentioned previously – the relationships that emerge.

The relationships among the chair, table, and floor are arranged by the systems engineer. The relationships among the people in a group are arranged by the group's manager. Sometimes the manager allows the people to arrange themselves and emerge their own relationships. Self-arranging is a type of arranging that I have seen work well many times.

Hierarchy describes how systems comprise parts where many of the parts are systems in themselves. While writing this, I am sitting in a coffee shop with a Christmas tree in front of me. The Christmas tree is a system made of the tree, a string of lights, and glass ornaments. The tree – a part of the Christmas tree system – is itself a system comprising a base, trunk, and branches. The string of lights – a part of the Christmas tree system – is also a system in itself comprising wire, light fixtures, and light bulbs. The light bulbs are also a system made of glass bulbs, a filament, and an electrical interface.

Each level of the hierarchy of systems has its own emergent properties. Consider the tree in the Christmas tree hierarchy. The base is *at-the-bottom-of* the trunk. The branches *extend-outward-from* the trunk. This makes sense, that is a tree, but suppose the base was *parallel-to-and-beside* the trunk and the branches were also *parallel-to-and-beside* the trunk. That describes a chopped up tree sitting on the curb waiting for the garbage truck – not the same system as the Christmas tree.

The same properties of hierarchy and emergence at each level of the hierarchy hold for groups of people. Large teams comprise smaller teams that comprise smaller teams. Relationships among teams and individual people emerge at each level of the hierarchy. In addition, individual people are able to reach out and emerge relationships with other individuals on remote teams. For example, I am in team A and my cousin is in team Z in another city. We have no relationship in the system diagram, but in reality we have a strong relationship – we are cousins. How can a manager manage that? Should a manager even try? Thousands of such informal yet strong relationships emerge in large teams of people. At times I wonder how people ever accomplish anything given all the possible relationships that can exist. That, however, is a marvelous characteristic of people. We are able to function and thrive in the face of almost unimaginable complexity.

Communication is the transfer of information that reduces uncertainty. Moving information that doesn't reduce uncertainty isn't communication. It is merely wasted effort.

The communication in a system moves through the interfaces of the system (refer back to the chapter on Interfaces). The interfaces define how communication occurs and what types of information is moved. Sometimes the communication is simple shouts between field workers controlling irrigation gates. Sometimes it is computer-to-computer transfers of databases regarding restaurant and supplier inventories.

Consider the three systems mentioned above: the table, chair, and floor, the Christmas tree, and the group of people. The table, chair, and floor don't communicate; they don't need to. The parts of the Christmas tree communicate in a sense by sending the electricity through wires to the lights. That is an important communication when considering the function of the tree, but it isn't sophisticated. The communication among the people in the group is different. Arranging a group of people into teams in a hierarchy reduces the amount of communication necessary for the functioning of the group. It doesn't, however, prohibit communication between my cousin and me working on different teams in different cities.

Communication in open systems – those containing people – is far more complex than in closed systems – a table, chair, and floor. A manager cannot predict the communication in an open system of hundreds of people. An astute manager can understand the communication inside the group. An astute manager can also observe the communication inside the group and use it for the good of the group.

Control steers a system in a desired direction under changing circumstances. In one sense, control allows the systems engineer to identify everything that is part of a system. Observe what changes when control is applied to a system. Everything that changes is part of the system. Careful observation has often surprised me. I noticed things that weren't supposed to be part of the system move with the system. Hence, those things were in fact part of the system. I just didn't know it yet.

There are hundreds of volumes of books available on control and control theory. I consider the basics to be:

1. What is the system doing?
2. What is the system supposed to be doing?
3. Move the system from 1. closer to 2.

Point 1. comes from asking questions (refer to the chapter on Questions) and observation. Point 2. comes from the customer's requirements. Point 3., the control action, comes through communication to and inside the system. This action intends to move the system closer to 2. Once the action is complete, the control cycle returns to point 1.

The systems concept comes together in terms of control. The systems engineer cannot control a system unless communication exists inside the system. This communication doesn't exist unless the systems engineer understands the hierarchy of the system. The systems engineer cannot understand the hierarchy without understanding the relationships in the system that emerge when parts become systems.

False Precision

> There's no sense being precise about something when you don't even know what you're talking about.
>
> John Von Neumann quoted from [Weinberg 1988]

Earlier in this book I described techniques for using a spreadsheet to score alternatives when designing a system. That is part of the trade off and decision-making process. That method can lead to disaster, with the danger coming from false precision.

As a systems engineer, I cannot judge the value of an alternative to five decimal places. When, however, I put a few numbers in a spreadsheet, the computer calculates results to many decimal places. That is a great sense of false security.

I keep a slide rule on my desk at work (for those of you who attended high school in the 1980s and later, please look up "slide rule" on that new-fangled Internet thing). I use it to help me estimate things without having a false sense of precision. The slide rule provides about three significant digits of precision. So, 0.8 divided by 0.3 is about 2.7 or so. That is good enough precision. Especially since most of the time the 0.8 and the 0.3 are imprecise value judgments.

General Systems Thinking advises me that I really don't know what I am talking about most of the time. I shouldn't fool myself into thinking otherwise.

Purity and Learning

> *Purity* is the enemy of *learning*.
>
> [Weinberg&Weinberg 1988]

Systems engineering can be messy; that can be great, but it may not seem so at the time. The systems engineer should be aware of the desire for a neat, orderly, step-by-step approach that pops out the correct answer on time.

There are two reasons I have for not trying to be perfect. First, it is a waste of time. Systems engineers are human, and humans are not perfect. Second, if by some chance I am able to make things perfect, I won't learn much.

For many years, I tried to use a perfect step-by-step approach to solving problems and engineering systems. I typically used the best approach I had, but found myself jumping ahead a few steps, retreating a few steps, jumping, retreating, circling, and somehow producing a system that satisfied the customer.

I often had to wonder, "Was I cheating? Was there something wrong with the step-by-step approach? Was it bologna that some theorist created and never really used?"

My conclusion has been, "No, I was learning. I had a mess, I tried to clean up the mess."

I present a *pure*, step-by-step approach to systems engineering in this book. Please don't be disappointed if you don't experience that approach.

I believe that you will frequently be working with your customer and jump to a "quick sketch" system design that satisfies them. When finished with a "quick sketch" or a "jump ahead" or a "let's not waste time with the steps" system, use the step-by-step method to check the answer.

The quick sketch solution may be right. Sometimes I have been right with a quick sketch. Sometimes I was mostly right, or I was in the right direction, but I did forget a little here and there. Checking my answer with the step-by-step method improved my answer. Sometimes checking my answer saved me from catastrophe.

General Systems Thinking tells me that I won't learn much in a neat and clean situation. My customer won't either. Learning usually leads to better systems, so relish a mess.

The Aspirin Illusion

> **Aspirin Illusion**: The suppression of pain instead of the eradication of the disease for which the pain is a warning.
>
> [Weinberg&Weinberg 1988]

I sometimes take painkillers like aspirin to suppress a headache. I rarely think about what caused my headache (stressful situations, stressful people, lack of sleep, reading too much). Thinking about eradicating those things would be more work, but would yield a longer-lasting solution.

The aspirin illusion brings to the systems engineer one of the largest challenges. I love to solve *the* problem that a customer brings to me. Sometimes that specific problem is only the pain the customer feels; it is not the cause of the pain.

I recommend the systems engineer delve deeper into the situation the customer brings. Working the deeper, hidden situation will bring greater satisfaction to the customer. Be aware, however, that attempting to show the customer that he has a deeper, hidden problem may be hazardous to the health of the systems engineer. Proceed with caution.

Replacing Systems

Much of what the systems engineer does is replace the system a customer is using. I point to two "Laws of Economic Behavior" (emphasis added).

> **First Revised Law of Economic Behavior**: We will do today what we did yesterday unless there are *good reasons* for doing otherwise.
>
> **Second Revised Law of Economic Behavior**: The good reasons which are necessary if we do not do today what we did yesterday are derived mainly from dissatisfaction with what we did yesterday or with what happened to us yesterday.
>
> [Boulding]

The systems engineer is called when the customer is no longer satisfied with the system he used yesterday. There are *good reasons* for this dissatisfaction. The systems engineer should understand those reasons as they will drive the direction of the new system.

First, the systems engineer needs to ask for those reasons. The customer, in my experience, doesn't readily express them. Most of this reticence is that the customer is embarrassed about past failures by either himself, his colleagues, or his predecessors.

Second, the systems engineer needs to ask for the real reasons for the change. I have often heard customers explain at great length why they wanted a new system. The more complex the explanation, the more likely the customer is providing a polite or political reason. They are hiding the real reasons and thereby withholding information that will determine the success or failure of a future system.

I encourage the systems engineer to explain Boulding's two laws and how understanding the *good reasons* for change are crucial to changing in the right direction.

Progressive Complexity and Its Affects

I have been building systems for over 25 years. The systems I build today are far more complex than those I built when I started. That may be more of a condemnation than a commendation. Building progressively complex systems brings consequences.

> (1) Progressive Centralization: The tendency, as a system becomes more complex, for certain parts to become dominant.
>
> (2) Progressive Differentiation: The tendency, as a system becomes more complex, for parts to be specialized.
>
> (3) Progressive Integration: The tendency, as a system becomes more complex, for the parts to become increasingly dependent on the whole.
>
> [Checkland]

From Progressive Centralization, the systems engineer should be careful of a part or segment of a system becoming dominant. The dominant part will determine the performance, reliability, usability, maintainability (and all the other -ility's you can name) of the system. If that part falters, all the “-ility's” in the system decrease. The extreme case of Progressive Centralization is called "single-point failure." This is when the failure of a single part causes the entire system to fail.

From Progressive Differentiation, specialized parts are troublesome. They are made one at a time by engineers acting as artists. Specialized parts are expensive to build, expensive to use, expensive to maintain (all those “-ility's” again), and expensive in many other ways.

From Progressive Integration, the parts in a complex system depend on all other parts to function. When one part breaks, the system collapses soon after as the rest of the parts fail. A complex system is less dependable than a simple system.

From all three of these, there are potential problems when a system becomes complex. Take care in building systems that are more and more complex. As a systems engineer, I can take great pride in my ability to build complex systems. If I can build ever increasing complexity, I must be smarter. That is false pride. Being able to solve a complex problem with a simple system is better.

Also, be careful not to confuse complex situations with complex systems. Simple systems can satisfy customers in complex situations; that is good. Complex systems can satisfy customers in simple situations; that is bad. Complex systems can satisfy customers in complex situations; that may be necessary, but it brings the potential problems from the three principles stated above.

Strive for simplicity in systems.

Equifinality

> **Equifinality:** A fundamental characteristic of open systems, by which the same goal is reached from different starting points and in different ways.
>
> [Checkland]

I am sitting here in Fairfax County, Virginia in my 49th year of life. There are many people in the same geographic and chronographic state. No two of us, however, started in the same place. We are equifinal in that we started at different points, have moved through life in different ways, but have arrived at the same final state. We are people, and equifinality is a characteristic of systems containing people.

A computer is not equifinal. Consider several computers that each takes in a number, adds 3 to that number, and displays the sum. For each computer to display the number 10, they must each take in the number 7. They can only reach the same final state – displaying a 10 – if they start with the same initial state – inputting a 7.

Computers (machines) are not equifinal. They are just machines that lack intelligence. Systems built from machines are this way as well; they are not equifinal. Equifinality usually applies only to people and other living things.

Equifinality is one of several General Systems Thinking principles that separates living things from machines. Living things and machines are different; a great mistake people often make is to treat the two the same.

Open and Closed Systems

Equifinality is a characteristic of open systems. As stated about an open system:

> ...it maintains its structure in the midst of a through-put of chemical material. It is not only a homeostatic control system, it is a self-maintaining system capable of metabolism and digestion, that is, the intake of substances which it uses in part to maintain or to extend its own structure.
>
> [Boulding]

Warm-bloodied animals have a homeostatic control system. We are able to maintain a constant body temperature in the face of changing temperatures in the environment. Maintaining this constant body temperature takes energy. Hence, we have to eat and drink to provide ourselves with the ability to maintain control of our temperature.

People are open systems. Computers are closed systems. While computers and their programs can be complex, they are simple when compared to humans and other open systems.

A person, for example, can maintain a steady state. My body weight remains relatively constant. To the untrained, it may appear that my constant weight just happens that way without any effort. That is far from the truth. Complex processes in my body are constantly at work consuming, converting, and burning fuel so that I maintain a constant weight. I'm an engineer, not a biologist. I don't understand how any of this works, but I know that it does.

The steady state of an open system is deceptive. Consider a group that had 100 people last year and has 100 this year. The group had 15 people leave and 15 new people arrive during the year.

Much happens in the group when people flow through it. The 15% turnover in this case is disruptive to the group, but people can manage that. What, however, would life in the group be like if 30 people left and were replaced? What if it were 50? What if it were 80? Life inside the group would be chaos, but seen from the outside, the group is at steady state.

Failure to understand the concept of steady state in an open system can be disastrous. Suppose a new person arrives to manage the group of 100 people. The manager sees the size of the group at steady state and feels comfortable in tinkering with the group's culture and organization. The manager reasons that a few changes can't hurt much because this is a stable group. The manager's changes amplify the chaos in the group, 75 people leave the first month, and chaos turns to catastrophe.

It seems that no one would be as simple-minded as this manager. The manager would surely understand the group's state and act wisely. I, however, have seen such actions from managers several times. They see a simple case of steady state and assume that everything about the group is in steady state.

Take care when working with open systems. There is more happening than what a steady state implies. Simple management or steering actions don't bring simple results. Adding 2 and 2 can bring 13.

The Toilet Example

Consider the basic household toilet mechanism in Figure 1. This is a closed system. It comprises the tank, the float, the regulation system, the water intake, the water output, the flush ball, and a few other parts. This is all nice and neat.

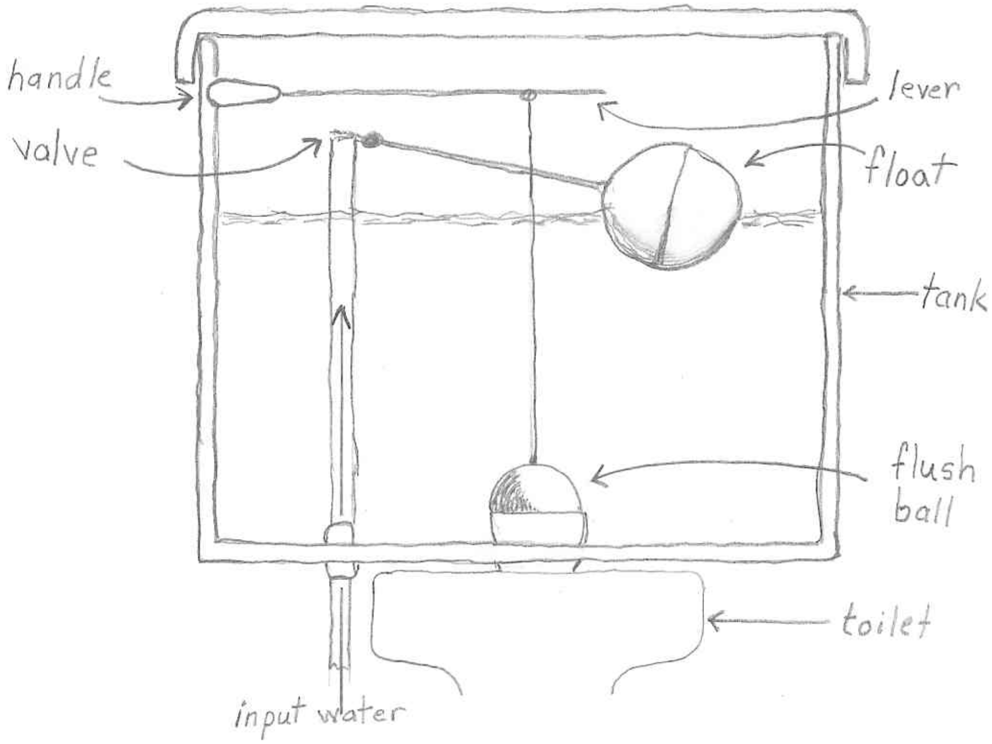


Figure 1 - A Toilet Mechanism

You push the handle. The flush ball rises, the tank empties, the flush ball falls, the valve opens, and the water flows into the tank until the rising float closes the valve. Each toilet begins this cycle with a full tank and ends with a full tank. The toilets are not equifinal. They reach the same final state only from the same initial state. They are machines.

This would be true, except that I started the previous paragraph with, "you push the handle."

That means the system has a person in it. The person makes this an open system. What is the person thinking? Is the person an adult? Is the person a child who has just realized he is controlling a hydrodynamic system and wants to play? Is their a drought in the area and people are encouraged to conserve water? Is the toilet's flush ball faulty (sticky, misshapen) causing leaks and requiring the person to intervene? All these extra things – many of them quite complex in nature – come when a person arrives and the toilet becomes an open system.

The simple toilet now becomes an impossibly complex problem for the systems engineer to work. The systems engineer must first identify the customer we are trying to satisfy. Is he:

- The child playing with the handle?
- The parent paying the water bill?

- The utility company trying to judge the water needs of the community?
- The other engineers trying to design the toilet's control system?

These people are all involved with the system. Satisfying all of them is not possible. The systems engineer will need to trade off the needs of each person against those of everyone else.

Another fundamental problem for the systems engineer is examining the entire system and applying wisdom. What is the entire system? If the system comprises only the mechanical items in Figure 1, the job is much easier – a basic closed system. If, as explained above, the system includes a person with their hand on the handle, the systems engineer faces a complex open system. Providing a system that covers 100% of the possible situations included in the open system is almost impossible. It is not wise to attempt. At the other extreme, covering 10% of the possible situations is feasible, but wouldn't satisfy the customer.

I recommend the systems engineer take advantage of the open system. A person is present, and one of the fundamental tenants of General Systems Thinking is that people are wonderfully capable. People are able to function in complex, changing, and unprecedented situations. Build a mechanical, closed subsystem (Figure 1) that works 50% to 75% of the time. Let the person – the open subsystem of the system – take care of everything else.

Observing

As noted in the chapters on Questions and Thinking, observation is a key activity of the systems engineer. I reinforce this notion with:

> Of all the things we'd like to teach systems analysts, the art of observation seems to be the most elusive.
>
> [Weinberg 1988]

Observing or noticing is one thing I have done well most of my life. My studies in General Systems Thinking have improved this skill. At times I don't understand how other people don't observe better than they do. I am sure they wonder about me regarding other things.

Learning to observe can be helped with a "Black Box."

> In our daily lives we are confronted at every turn with systems whose internal mechanisms are not fully open to inspection, and which must be treated by the methods appropriate to the Black Box.
>
> William Ross Ashby from [Weinberg 1988]

A black box is a system whose internal workings are hidden. The toilet mechanism from Figure 1 is a black box in my bathroom until I remove the lid. Then I can see inside and observe how everything is supposed to work.

Weinberg [Weinberg 2001] discusses at length a black box machine that he uses in exercises to give learners a chance to practice observing.

I have created my own version of a black box for exercises using a computer. The observer presses a key and watches what appears on the screen. I use simple things in my computer black box such as taking the input key, adding five to it, and displaying the sum. Given such a simple concept it seems that people would be able to tell me what happens inside the hidden computer program, but I have yet to receive a complete answer.

Observing people is like observing a black box. I don't know what is happening *inside* the other person. I can sometimes guess, sometimes guess pretty well, but I never know. People can be deceptive. Sometimes deception is their goal. Sometimes, however, they are trying to display on the outside what they are experiencing on the inside, but their efforts and my observations yield misunderstandings.

An advantage to observing people is that I can ask what is happening on the inside. That can help. A disadvantage to observing a person is that asking starts a conversation that itself is fraught with misunderstanding.

Observing is more than seeing. Seeing is my preference for gathering information. I often use the expressions "Oh, I see" "Do you see what I mean?" and "I get the picture." Don't let my tendency toward these and other phrases mislead you. Listen, smell, taste, and touch when observing. These other senses can provide the key piece of information to the systems engineer.

The systems engineer should observe three things: the customer, the system, and the environment. Observe the customer while asking questions, while he thinks, while he uses his current system, and while he tries the system you are providing. Does the customer's voice tremble? Does the customer's hands shake? What is the customer's posture? Is the customer perspiring in an air-conditioned room? Are there tears in the customer's eyes?

Observe the system that the customer currently uses. There is a system there even though it may be 100% people. What happens in the system? Who is in the system? What role does each item in the system play? What appears to be the most important part of the system? The least important? The most ignored? The most strained?

Observe the environment surrounding the customer and the system. What is the temperature, humidity, and atmospheric pressure? What time of day, week, month, and year does the system operate? What are the emotions in the environment?

When observing, try to sense changes as these are strong clues to what is really happening inside the black box of the customer and the system. Consider my computer black box; I press an "a" and an "f" appears on the screen. What is the change "a" to "f"? What does that mean?

Watch the changes in the customer. Consider the person who is leaning back in a chair with legs out stretched and crossed and hands behind their head. I ask a question, and they lean forward, plant both feet flat on the floor shoulder width apart, and place their hands firmly on their knees. That change in posture means something. Sure that seems obvious, but what about when a small tear appears in the corner of the customer's eye? That is a tiny physical change indicating something major is happening inside the customer.

Closing Thoughts

One conclusion I have drawn from studying General Systems Thinking is:

Be alert when your systems contain people

People are not the same as machines. Healthy people are not programmable like computers; they do not respond to button pushes with machine-like predictability. Programmable people, if you ever encounter any, are suffering a malady.

The systems discussed as examples in this book all contained people. They are all open systems fraught with and blessed by the improbable situations that come with people. Please pay attention to the actions of people in the systems. When I have, I have been around much better systems.

I hope that this chapter spurs some interest in General Systems Thinking. Please consult a few of the references given below.

Suggested Exercises

1. Consider several of your more enlightening conversations. What messages did you send and receive? How did these messages alter the image, i.e. the knowledge?
2. Describe a trade off that you are now considering on a system. Write specifically what benefit each option provides and what loss each option incurs.
3. Write some of the better questions you or other people have asked you. Extrapolate these questions to General Systems Thinking and systems engineering.
4. Describe some of the better and worse side effects that emerged from systems that you built or used.
5. Consider a few cases where you were in a mess while building a system. What were some of your bigger lessons learned about systems engineering? What were some of your bigger lessons learned about yourself?
6. Describe a situation in which a customer did not convey their dissatisfaction with what they did yesterday, i.e. their old system. How many systems did you have to build to satisfy the customer? How could you have better learned of the customer's dissatisfaction?
7. Describe a problem you have experienced in building complex systems. How could those problems been prevented by building a simpler system?
8. Describe some unexpected situations caused by people in systems.

References with Notes

I end this chapter a little differently. The books listed below are not all used in the chapter. I include a few others that have influenced my thoughts on General Systems Thinking. I also include a note with each reference to help describe the book.

[Bertalanffy] "General Systems Theory - Foundations, Development, Applications," Ludwig von Bertalanffy, George Braziller, Inc., 1968.

A treatise on the field written by its creator.

[Boulding] "The Image - Knowledge in Life and Society," Kenneth E. Boulding, The University of Michigan Press, 1956.

A wonderfully philosophic book. I had to read this one slowly and I enjoyed it immensely.

[Checkland] "Systems Thinking, Systems Practice," Peter Checkland, John Wiley and Sons, 1981.

A good discussion of the field highlighting its core principles.

[Davidson] "Uncommon Sense, The Life and Thought of Ludwig von Bertalanffy, Father of General Systems Theory," Mark Davidson, J.P. Tarcher, Inc., 1983.

If you read only one book on the subject, read this one. It covers both the principles of the field as well as gives a biography of its founder.

[Weinberg 2001] “An Introduction to General Systems Thinking,” Gerald M. Weinberg, Dorset House Publishing, 2001.

This is a second release of 1975's “An Introduction to General Systems Thinking” published by John Wiley and Sons. The 1975 printing was the first volume in a series on General Systems Thinking.

[Weinberg and Weinberg] “General Principles of Systems Design,” Daniela Weinberg and Gerald M. Weinberg, Dorset House Publishing, 1988.

This is a second release of 1979's “On the Design of Stable Systems” published by John Wiley and Sons. The 1979 printing was the second volume in Weinberg's series on General Systems Thinking.

[Weinberg 1988] "Rethinking Systems Analysis and Design," Gerald M. Weinberg, Dorset House Publishing, 1988.

Weinberg reconsiders what he wrote a decade earlier in light of further experience. Some of his concepts are repeated, some amplified, and some altered.

Where We are Now

We are now at the end of the book.

For me, systems engineering is at one end or another of a spectrum. It is the simplest thing I know – understand the situation and work to satisfy the customer. It is also the largest and most difficult thing to grasp – understand the situation and work to satisfy the customer.

To this day, at the end of writing all these words, I sometimes struggle when someone asks, "What is systems engineering?"

I stammer, "Well, you know, you're engineering an entire system, not just parts of it. You know, that end-to-end kind of thing."

On good days, I pause, and say, "A customer comes to you with a situation, a system they want to replace. Examine the entire situation wisely and work until the customer is satisfied."

That philosophic answer is correct for me, but not satisfactorily precise for many. For those I add,

- Understand the requirements
- Sketch an architecture
- Define and maintain the interfaces
- Be able to trace up and down through the levels of the design
- Keep all this information readily available

Never forget the people in the system. They have wonderful capabilities and their presence makes it all worth the effort.