Requirements Engineering
A Modern Approach to the Requirements Engineering Body of Knowledge

by Sheldon Linker, PhD & Michael Scott Brown, PhD
First Edition - 2021

Published in 2021.

The writing of this book was partially funded by a Maryland Open Source Textbook Initiative (M.O.S.T) grant.

*From Sheldon: Thanks to Ken Ellson (*obm*) and Betty Niimi, who taught me half of what I know about project management and requirements engineering. The rest I learned from really bad managers who let projects get completely out of hand, and thereby funded most of my consulting career.*

From Michael: Thanks to all of the requirements analysts that I have worked with over the decades. I learned about requirements engineering from the good and bad requirements analysts.

Versioning Info

Date	By	Change
15-Aug-2021	Sheldon Linker & Michael Brown	Prerelease version as converted from Google Docs
16-Aug-2021	Sheldon Linker	Additions for autonomous processes and flushing out of some "text goes here" tags, and by bringing some "add these" items into the rest of the document.
17-Aug-2021	Sheldon Linker	Addition of §6.3, and insertion of a new §5.5 (sliding the existing sections numerically down)
18-Aug-2021	Sheldon Linker	Addition to §2.6 and 10.1.7, and addition of 10.3
19-Aug-2021	Sheldon Linker	Insertion of new §2.9 and 10.4 (sliding the existing sections numerically down); flushing out of text in 4.10 and 5.2
20-Aug-2021	Sheldon Linker	Flushed out §5.3.3; added remaining discussion question slots, some with questions, and some without
21-Aug-2021	Sheldon Linker	Filled in some questions and inserted a new §9.4 (sliding the existing section numerically down); addition of Appendix A & B
22-Aug-2021	Sheldon Linker	Added Appendix C & E
29-Aug-2021	Sheldon Linker	Added a "How Things Can Go Wrong" section to most chapters.
2-Sep-2021	Sheldon Linker	Added Appendix D
12-Apr-2022	Sheldon Linker	Dropping a repeated paragraph

Contents

1. Introduction

By this point in your education, you probably have a good idea of what requirements are. If you ever wrote a program, you probably read requirements to know what the program should do. Requirements are the most difficult part of whatever software development methodology you may use. It is truly the document that bridges communication between technical people and non-technical people. This is no easy task.

We are engineers. So our opinion is that non-technical people use words incorrectly. Words like "never", "are", and "always" mean something different to non-technical people. Michael once built a system for a government agency. A field in the system was called "case number". The customer told the requirement team "Case numbers are ten digits." The system was built and it went live. Then Michael got a call from the customer saying that they couldn't put in cases. They showed him an example of a case number that started with the letter "A". He explained that the requirements stated that case numbers were 10 digits, and asked if that was a correct statement. They responded "Yes, case numbers are ten digits, unless it is a letter and nine digits." As engineers, we would claim that the first statement is wrong, because it is incomplete. The customer didn't see it that way. But it is not just the definitions of words that cause issues.

The English language is ambiguous. In fact, all human languages are ambiguous. Consider this sentence: "*Fruit flies like grapefruit."* What does that mean? There are two ways to understand that sentence. The first is that there is an animal called "fruit flies", and they like to eat grapefruit. The second is that there are objects called "fruit", and that they know how to fly. How they fly is similar to how grapefruit flies. For humans, this example is easy to figure out, because we know that fruit cannot fly, so we eliminate that option. But we can't always eliminate possibilities. Consider this sentence: "*Be careful; those chicken wings are hot.*" In this sentence, are you being warned that the wings have a high temperature, or that they are very spicy? You don't know.

Normally as the quantity of information on a topic increases, ambiguity decreases. As you are given more information, ambiguous options are eliminated. In the example above, we would say "Fruit flies like grapefruit. In fact, fruit flies can eat a grapefruit in only a few hours." But sometimes, as the amount of information increases, it becomes more difficult to understand. Consider this requirement:

> Assume that there is an invisible grid across the screen with 100 rows and 100 columns. We number the rows 1 to 100 and the columns 1 to 100. Rows and columns are of the same size. On the screen, fill in each cell of the grid that falls within a direct line between the following ordered pairs. In each ordered pair, the first number indicates the row and the second number indicates the column: (33, 1) and (33, 100); (66, 1) and (66, 100); (1, 33) and (100, 33); and (1, 66) and (100, 66).

This is a very detailed requirement. What does it do? We can all see that it draws some lines, but what is the "big picture"? Now let's rewrite the requirement a different way:

> Draw two horizontal lines and two vertical lines across the screen to make nine equal size squares, assuming that the borders of the screen count as lines.

Ok, this has a lot less detail. But it makes it easier to understand. We clearly know that we are drawing four lines. Now we are going to write the requirement again:

> Draw a Tic-Tak-Toe board across the entire screen.

Which of these three requirements is the easiest to understand?

These exercises demonstrate some of the difficulties faced by requirements analysts. How do you make a requirement document unambiguous and easy to understand? Much of this book will focus on this.

1.1. Why Requirements Engineering and Specs are Important

The obvious answer to this question is that we cannot build anything if we don't know what it needs to do. But more broadly this answer has deep meaning to all of engineering. Without requirements we have no defects. Without requirements we do not know when we have finished. Without requirements we cannot judge quality.

1.2. Who are the Stakeholders?

You'll hear the term "stakeholders" a lot. But who are the stakeholders? A stakeholder is anyone who has a stake in the outcome of the project, whether or not they are paying for the project, and whether or not you or anyone else reports to them. For instance, if there is a defined customer, then the customer is a stakeholder. If there is a proposed class of customers, then that class, collectively, are stakeholders. Any proposed user is a stakeholder. Everyone you report to, your coworkers, and everyone who reports to you are stakeholders. The stockholders, too, are stakeholders, but in a purely economic sense. Bear in mind that you *can't* keep all of the stakeholders completely happy.

1.3. Do They Even Know They're Stakeholders?

You've no doubt heard that you'll be getting input from the stakeholders. When the stakeholders are external, there's no problem. The customers have come to you. There are only two stakeholders involved — the customer and the service provider or vendor. However, if the project is internal, then determining who stakeholders are is not as easy as you'd like. There are major stakeholders, minor stakeholders, people who are clearly not stakeholders, and pretenders to a stakehold. In many cases, people who are major stakeholders will know that they are, but some of them may think that they're minor stakeholders. Many minor stakeholders think that they're major stakeholders, or at least want to be treated as if they are. Some think that they are not stakeholders. Some people want their opinion heard, even if they have no

connection to the project. Don't take their demands seriously, but at the same time, don't let a good suggestion go unheard or unheeded. One way to deal with uncertainty in the situation is to get an organizational chart (often called an "org chart") and to see how everyone fits in.

1.4 Certification

There is a certification that some requirements engineers choose to earn. The certification is called the Certified Professional for Requirements Engineering, or CPRE. However, many requirement engineers never obtain the certification for their entire career. But some employers see value in it, and it might give people looking to get into requirement engineering the edge that they need to get their first job.

The certificate is given by the International Requirements Engineering Board (https://www.ireb.org). They have a series of exams, starting with the Foundation Level exam.

1.5. An Example

In one of Sheldon's consulting assignments, he was working for a client company that has a "code first" philosophy. No design, no plan, not even flushed-out requirements. Just one user story. They use a one-size-fits-all philosophy. They code *everything* in one specific language, and use one specific database, even when neither is the optimal way to go. Sheldon could have done the project by himself in two days. But it was a 9 man-month schedule (meaning that they planned on paying 9 months' salary to get it done). Why? Because they used layer upon layer of abstraction, and experimented and redesigned as they went, and used a 4-tier distributed architecture. This system costs $100,000 per year to run, even though it could have been wedged into a server that rents for $15 a month. The morals of this story: (1) Know what you need to do before you start; (2) If you handle a cheap job like it's an expensive job, it *will* be an expensive job; and (3) Use the right tools for the job.

1.6. How Things Can Go Wrong

Getting requirements engineering right is pretty-much required to have a project be a success, and in industry, most projects are reported to be failures in one way or another. Most projects are not 100% successful in everything they do or 100% failures, although there are some of each. There tend to be varying degrees of success. The goal of this book is to help you get to 100% success, or very close to it. We're not going to claim that there is one right way to get a project done right, or even that there is a best way. The claim is that there are a hundred ways to do any endeavor right, and a thousand ways to do it wrong.

1.7. Discussion Questions

1. Can you provide another example of ambiguity in written and/or spoken language?
2. Are all stakeholders created equally? Can you think of examples in which one stakeholder needs to have more influence on requirements?

2. The Phases of Requirements Engineering

Requirements engineering is not a monolithic task. There are phases and they often repeat in a repeated process. Here are the phases of requirements engineering. As requirements engineers gain experience they begin to develop an understanding of when a phase is completed or needs to be revisited.

2.1. The Need

Before anything else can happen, there needs to be a perceived need for something. It could be a new product, some sort of research, or something the business needs to do.

2.2. Elicitation

Elicitation is extracting the need from the stakeholders. This may sound easy. It might sound obvious, but it is not. Elicitation is one of the hardest tasks in requirement engineering. Good requirements engineers must be clever. There are a number of obstacles that hinder elicitation.

Many stakeholders are not technical people and do not have the communication skills to explain what the system needs to do. Others have never defined formal business rules, so creating systems to implement those rules cannot even begin until they are defined. In some cases stakeholders have hidden agendas and may want a system built for selfish reasons that adversely impacts the organization as a whole. Oftentimes, stakeholders know something that you don't. Be sure to glean that knowledge when you can. Sometimes, stakeholders believe something to be true, but it's not. Test whatever theories that you can. And sometimes stakeholders' opinions conflict. That's when you really need to get things sorted out.

All of these obstacles make elicitation very difficult.

There is a perception that automation eliminates jobs. Studies often prove this is not true. Automation lowers costs, which eventually drives up revenue and often adds to the labor force. But the perception that automation eliminates jobs makes some stakeholders want to sabotage the project. Very often, automation frees people up from the rote-work to do their real jobs.

2.3. Analysis

The elicitation step gives you the raw data, but not the "cooked" or processed data. After the elicitation stage, you're going to have a lot of data. It won't necessarily be complete. There will be a lot of duplicate data, and likely there will even be conflicts. Some of the items elicited will be worth the effort, and some will not be. Some may not even be feasible. Yet others may have already been implemented.

In the analysis phase, your job is to translate a jumble of ideas into a coherent story that makes sense to the prospective users, and which makes business sense. It's generally in this phase that the requirements specification is first written.

2.4. Negotiation

It is especially true in large projects that not all of the stakeholders agree on the requirements. Too often it is up to the Requirements Analyst to resolve this conflict through negotiation. While effective negotiation is an entirely other body of knowledge, here are some basic tactics that can be employed.

It is better to build a system that is flexible than inflexible.

Stress to stakeholders the importance of sticking to the requirements that they agree upon. This often minimizes the importance of requirements that they do not agree upon which increases the chance that one side will give in on a disagreement.

In projects where it is likely that there will be requirements disagreements, organizations often define voting power. When disagreements arise that cannot be resolved through negotiation, the decision is left to a vote. On some projects, there could be multiple stakeholders, but one person has all of the decision-making power. Often this is the person that is writing the check. To avoid issues in the middle of the project, it is best to define these powers early.

Oftentimes stakeholders need the system to meet some objective. They believe that the objective can only be met through one specific requirement. Often in requirements disputes, asking each stakeholder why that requirement is important can clarify the root objective. Identifying this root objective can lead to other methods to meet the objective.

2.5. Documentation

Although we started the requirements documentation in the analysis step, we can't really say that the documentation is complete until after the negotiation step. Documentation might be in the form of a requirements specification, a collection of users stories, "cards" on a KanBan board, a user manual, or a host of other forms — whatever is appropriate for your project and customers, be they internal or external. Various forms of documentation will be discussed in more detail in the following pages.

2.6. Validation

Once the documentation is complete, you're *almost* ready to go on to the next step, which will be the development stage (often starting with internal design), the next "sprint", the "backlog", or maybe even the "further refinements backlog". But before we can say we're done, there should usually be some form of validation. Usually this is done in proofing the documentation, but more importantly in going through whatever documentation was sent us, including requests, meeting transcripts or recordings, and whatever else we have to make sure that we have the truth (we don't want any conflicts between what we agreed to and what we documented), the whole truth (we want to make sure we haven't left anything out), and nothing but the truth (we want to make sure we're not just making some requirements up, and we want to make sure that whatever requirements we're listing are within the planned scope and budget of the project.). Lastly, we

want to check that what we're planning is within the planned schedule and budget. If not, plans, schedule, or budget have to change. See the Engineering Triangle section for more on that.

In general, the person who wrote the requirements document should not be the one checking it. The requirements document, like everything else in the project, should go through a formal quality assurance (QA) process when possible.

2.7. Use

These requirements documents would be worthless if they were not used. That's why we're doing them in the first place. The requirements document is going to be used in four places:

First, they'll be used as a final gate or go-ahead (or "go/no-go") point. Given that we're going to build *this*, are we really going to proceed? This is the first place a project will succeed or fail based on the requirements. If the requirements are valid, and the work looks like it will fulfill need, ability, budget, and calendar, the project will proceed. Having a well-written document is important here, in that one grammatical error or typo can be the first chink in the armor that leads to lack of confidence in the project. So be careful there.

Second, the document will be used by the developers, and this is the key point. The document must be sufficient to code from. You don't want the developers coming back to you and asking what you had in mind. On an Agile project that's bad enough, but on a Waterfall project, you're probably off on another project by then. There must be enough info to get the job done right, and the info must be unambiguous. You also want to make sure that you don't have too much there. For instance, if you were to put in a requirement that Oracle be used when the system was already running on DB/2, that might end up in the installation of extra software and conversion code. If a project fails because of a bad requirements document, it is most likely to happen here, because they'll build what you told them to build, or failing that they'll build what they think you told them to build.

Third, the document will be used by the QA staff to verify that the programmers have done their work correctly. Again, your description of the work to be done needs to have been clear, concise, and unambiguous, or there will be arguments between the programmers and QA. An in such an argument, it's hard to tell who's right, but it's very easy to tell that the spec-writer was wrong. If a project fails because of bad requirements in the QA phase, it's typically because (a) QA thinks you told people to build one thing, and the developers thought you wanted something else, or (b) because you used the word "all" in some requirement, and the testing phase takes ten times as long as it should. For instance, "This must work with all browsers", instead of a list of browsers that would constitute passing.

Last, various IDs in the document will be used for project tracking purposes. This is logically last, but it happens throughout the project. In a use-case document, you'll generally have user stories known by number and name, or date and name, or number and date, or just number. Within those, you'll have primary and first and second alternate paths, and step numbers within those paths. In IEEE-830 format (and other formal formats), every single requirement will have some sort of ID. For instance, some §3.4.5 might have one requirement in it with no ID of it's own. That requirement would use the section number as an ID. But if there is more than one

requirement in a section, then each requirement will have an ID, such as UI-1 or REQ-2.3 or some such, and each one will be tracked, typically as Blocked (meaning that something else has to be done before that one can be started), Development, QA, Failed, Reworking, Retesting, or Done.

2.8. The Step that's Not a Step: Managing Expectations

Throughout the project, you have to make sure that the stakeholders know what to expect, and expect something reasonable. For instance, let's say that you're working for a car company that makes a car that gets 50 MPG. They want a car that gets "better" MPG. You and your team design an equivalent car that gets 60 MPG. The project is a great success, right? Well, maybe. If they expected you to produce a car that gets 55 MPG and you delivered 60, then you exceeded expectations and were wildly successful. If they expected 60 and you delivered 60, then you and the project were successful. If they expected 65, then the project was only partially successful, and if they expected 100 MPG, whether that's a reasonable expectation or not, then the project was a colossal failure. So, the trick here is to manage expectations, making sure that everyone is in the loop, knows what the plan is, and knows what to expect. Make sure that they know what the risks are, especially in R&D. There may be small setbacks, experimental paths that don't yield results, and broken parts or bugs to be fixed. Make sure that they know that although particulars can't be predicted, some amount of parts will fail, bugs will happen, and sick time will be taken. Keep the communications channels open. Promise as little as possible as late as possible, and deliver as much as possible as soon as possible.

2.9. Acceptance

No project is ever a success unless the "customer" (internal or external) accepts the work. What acceptance means and how it works depends on the type of project you're doing.

- If it's an internally funded project for internal use with a user interface, then it's only a success if the users accept and use it.
- If it's an internally funded project for internal use that will function autonomously, then it's only a success if it does what was planned.
- If it's an internally funded project producing a product that will be sold, then it's only a success if people actually buy it, such that the total value of sales exceeds the combined cost of producing the product and of marketing it.
- If it's an internally funded project producing a product that will be leased or otherwise licensed over time, then it's only a success if the outside customers not only buy in, but continue to use the product.
- If it's an internally funded project that is producing some sort of adware, such as a web-site that's free to its users but which carries ads and makes its money that way, then it's only a success if it delivers the audience week after week, in a demographic that the

advertisers are willing to pay for. In this case, both the technology must be sound, and the content must be interesting.

- If it's an externally funded project (bespoke software), and it's a time-and-expenses contract, then the customer is likely paying by the sprint or on deliverables, and the project is likely Agile. In this case, you have to keep the customer happy along every step of the way. That means that you have to make steady progress, and have something to show at every "dog and pony show" or "show and tell" with the customer, and the product has to fulfill that customer's needs.

- If it's an externally funded project, and it's a fixed-price contract, then the customer is likely paying on deliverables or milestones, or just some up front and the rest on delivery. In this case, everything depends on the customer "signing off". That means that you will have to prove (not just claim) that every requirement has been met.

In each of these circumstances, you have to prepare for that final success case from Day 1. You have to know what the win conditions are, so that you can plan for them. In the last case, there is extra work, because each specific requirement item is a separate contractual item, and any one of them can cause project failure, which can in turn cause a huge payout and possibly a lawsuit.

From Sheldon: Trust me on this one; I've been to court on this as an expert witness. People can get very unhappy, and then their lawyers want a lot of money from the other party.

2.10. How Things Can Go Wrong

Obviously, skipping a step can get you into trouble. Getting them out of order can be a minor problem. Note that there are a number of software development methodologies, each with its own sequence of requirements gathering steps. However, don't be a slave to a list. If you need to do something, don't just ignore it because it's not on the list. The two items here that carry the most risk are managing expectations (where the "customer" thinks they're going to get something other than what they're going to get, when they're going to get it, at whatever it's going to cost them; and the acceptance phase. If you don't have an acceptance plan, the entire project can be counted as a failure just because it never gets officially accepted.

2.11. Discussion Questions

1. Which phase of requirements engineering seems to be the most difficult?
2. Where does the most danger for project failure lay?

3. The relationship Between Needs and Projects

We had mentioned before that a perceived need is the reason for a project. Below, many of those reasons, with their business relationships, are explained.

3.1. When the Project is the Business

In the simplest circumstance, the project is the business. Take, for instance, Facebook. The Facebook program (project) is not a program that facilitates the business — it is the business in and of itself. To be sure, there is more to the program than any one user sees. For instance, as a regular user, ads are being served up to you. Somewhere in the program, those ads are being sold and billed.

This brings to mind an important point: When writing a specification, or requirements document, consider all the people involved in the transaction. Here, "transaction" does not just mean monetary things, but any sort of system-to-system interaction, system-to-person, person-to-system, or person-to-person. Note that a project may be composed of more than one program, and may involve hardware too, and may involve interaction. For instance, if I send a message to you, we can't stop at my sending the message. We have to at least get as far as you receiving the message. Maybe we even go to the point of me receiving a delivery receipt. Seeing a transaction through is called "transaction completeness". (There is a second, related, type of transaction in data base systems which often takes place at the same time, but the two are not exactly the same.)

Here, the need is simple: We can't do anything until we have the program done.

Note that a project may be composed of more than one program, and may involve hardware too, and may involve only hardware. While this book is mainly concerned with software requirements documents, the same techniques used here will usually apply to other projects as well, whether or not related to information technology.

3.2. When the Project is the Product

When someone who is not in the information technology (IT) industry thinks of a program, it's likely they're thinking of a packaged product or downloadable application (app). When someone buys an application or pays for an application or service on a regular basis, then the application/project is the product that the company is selling. The success of the company, or at least of the product, is dependent on sales of that product. If the product is good, and at a fair price (you can take "fair" to have both meanings, equitable and likable), then it will be profitable (if the costs to produce it were not too high).

Here too the need is simple: We're going to build it so we can sell (or rent) it.

3.3. When the Project Drives Profits

Very often, the purpose of a project is to drive profits for the company. Driving the profits may happen directly or indirectly. When profits are driven directly by the project, the company will usually call those producing the project a "profit center". When the project indirectly drives profits, the company will call those producing the project a "loss center". A non-programming example: A supermarket chain might call the store a profit center, and might call the warehouse and transportation departments loss centers.

An example of a project that drives profits indirectly is a company web-site. The web-site's job may be to bring customers in so that they spend money with the company, and it may be the web-site's job to actually sell products. An example of a project that drives profits directly could be in a company that sells development services. The company gets a customer who wants a project done, and the successful delivery of the project is the product being sold (hopefully at a profit) to that customer.

3.4. When the Project Decreases Costs

It used to be that if you wanted to buy an airline ticket, there were three ways to do it. One was to go to the airline ticket counter and buy the ticket there. Another was to go to a travel agent and do the same thing. Later, there was a third way to do it: Call them on the phone, and buy that way. Once the World Wide Web (or "WWW", or just "the web") was in place, tickets began to be sold that way. The reason for selling tickets on the web, rather than on the phone was simple: It costs less to have tickets sold that way, because the overall costs of writing the program and maintaining the software and hardware are lower than the overall costs of payroll and benefits for the phone attendants. So, adding web access to the airlines' systems decreased their costs.

Many times, in-office cost-decreasing reasons for a project are present. If we stay with the same example, airline tickets used to be sold on handwritten tickets, or from pre-printed tickets. Pre-printed tickets had to exist in only one place, and the right tickets had to be found to be sold. When the PARS airline reservation system came into use, any ticket could be sold from anywhere, so long as the seller had a PARS terminal.

Note that the World Wide Web and the Internet are not exactly the same thing. The Internet is the transport mechanism for data and the services they carry. The World Wide Web is a collection of services carried on the Internet.

3.5. When the Project Increases Speed or Efficiency

It is often the case that a project is done purely to increase the speed of something, even if that speed increase does not directly increase the bottom line (meaning the company's profitability). For instance, a factory control system might allow a product to be delivered faster, even if at the same cost. Of course, if customers get their products faster, they'll be happier, which will indirectly increase the bottom line.

A product can increase efficiency without affecting other aspects of a product too. For instance, cars used to have carburetors. Now, cars that still burn gasoline have fuel injectors. Those fuel injectors are often computer-controlled, using sensors in the engines and air intakes to determine the optimal amount of fuel to inject on a real-time basis.

Note that in these examples, there is nobody directly inputting any values or parameters, and nobody directly receiving any sort of display output. Don't get into the trap of thinking that every transaction involves a visible input and a visible output. Sometimes a sensor delivers the input, and sometimes an actuator or some sort delivers the output. Some transactions are initiated by sensors, and some transactions are initiated by a clock.

3.6. When the Project Enables Something

Sometimes a project enables something that could not otherwise happen at all. One such example is the SpaceX Falcon system. The rocket's first stage launches the second stage, then returns to earth and lands. In order to do so, calculations must be made faster than any person could (or an immense amount of fuel would have to be carried).

3.7. When the Project Protects Against Something

Protection can be in the form of physical protection, such as a door lock with keypad entry, or in the form of some sort of computational protection, such as fraud detection and prevention.

3.8. Collecting and Selling Data

Collecting and selling data is a big business. It happens as an adjunct to a lot of systems. A lot of big tech companies collect all sorts of data on their users — typically as much as they're legally allowed to — and there are not many laws limiting them. Those companies may sell the data to companies that want the data, either in an individual manner, or in an aggregated manner. For a good example of specific data being collected, sold, and used, check out any real-estate site, such as Zillow. They have fairly detailed property history data on just about everyone's home. Data in the aggregate is much more innocuous. For instance, CalTrans collects data from freeway sensors and sells that data to whoever would like it.

The European Union (EU), a number of countries, and a number of states have instituted laws controlling the amount and type of data that can be collected, and how it can be used. Whenever working on a project that is used to collect data, it would be appropriate to decide, often with the help of corporate counsel (a) whether the system will follow the laws of everywhere that the system will be used, or just follow the laws of where the system is located, and (b) after that decision is made, just what those laws are, and how it will affect the system.

Similarly, note that in any money-involved system, you will likely run into money-handling laws that vary from country to country. Again, those same two sets of decisions will have to be made.

There are basically three ways of handling local laws, whether it be pertaining to disclaimers, fair use, data collection, or money:

- The system can be built to operate under the laws of each country in which it's operating, doing one thing for users in one country, and something else for users in another country. For instance, PayPal operates by the rules of each currency.

- The system can be built to operate under the laws of just one country, and operate from there only, telling other countries "we don't care". For instance, Wikipedia operates from the US, where you can say what you want, and doesn't hold to the rules of countries that tell them what they can and can't say.

- The system can be built in such a way as to operate within all the laws of every country simultaneously by not doing anything illegal in any country. This is often the cheapest way to go.

- The system can be built by ignoring laws, and falsely claiming to be operate legally. This is typical of scammer systems.

3.9. Controlling the Flow of Information

Controlling the flow of information can be done for both good and evil purposes, so be careful. (Or at least know what side you're on.)

The good version of information flow control is most often used for data cleansing, or in protecting a learning system from incorporating incorrect and questionable material. For example, you wouldn't want artificial intelligence (AI) that learns by perusing the seedier side of the web.

The evil version of information flow control gives only one-sided information to people in order to keep the truth from them or to influence them in a particular manner, or to quash information that the paying entity doesn't want widely known.

3.10. Embedded Software

Embedded software is software that is a part of some device. Typically that device doesn't appear to the casual observer to have software, but does. A quick example is the typical microwave oven. The oven *could* have a circuit board that handles the buttons with debouncers and a shift register and a timer and a power controller and who knows what else, or it could just be a computer connected to the keyboard and the microwave controls, with a program in read-only memory (ROM). That ROM-based program is embedded software. Most cars now have a number of computerized functions, including, as mentioned above, the fuel injection system, plus the dashboard and entertainment system. Although most cell-phones now run applications, flip-phones run embedded software.

The specifications for these embedded systems often omit the fact that there is software, and talks only about how the system or device will behave. Sometimes, in specifications, hardware and software requirements are handled separately.

3.11. Systemic Control (Air Traffic Control [ATC], Factories, and the Like)

The idea of embedded systems has a larger, broader counterpart: Cooperative systems. Consider the case of an automated factory. The factory will have construction robots. Parts will be brought to the factory robots using "smart" conveyor belts, and taken from the robots in the same way. The robots' supply needs, such as welding materials, will be supplied by other more mobile robots. Those robots all have to coordinate with each other, either as peers, or as slaves to some sort of master control system.

Specifications for these types of systems will generally involve how the system as a whole acts. But, such a specification will typically have to state how each system within the factory will communicate with each other, or how they will interact with the system that runs the factory as a whole.

Sometimes remote systems need to interact with each other. For instance, a military plane will usually carry an IFF (Identify as Friend or Foe) unit that will communicate with ground-based systems, carrier-based systems, and systems in other planes or missiles. A specification will typically describe the protocols that these things are going to use to interact with each other.

3.12. Modernization

During the first year of the Covid pandemic, one of your authors noted that there were a lot of want-ads for people to work on conversion projects. Very often, these projects involved conversion from on-site systems to cloud systems, from Microsoft Dot Net to Microsoft Dot Net Core, from SQL-based databases to so-called NoSQL databases, from monolithic systems to distributed microservice systems, from mainframes to distributed systems, and many more projects like this.

There are a number of underlying causes for this type of conversion, categorized below.

Note that in many conversion or modernization projects, there is no need to produce any additional software specification or requirements document at all, since the functionality of the system will remain unchanged.

3.12.1. Useful

Of course, optimally, any conversion effort will have a good cause behind it. In the early days of computing, each manufacturer, and even each model of computer was so significantly different that some conversion effort was required to move the program from the old computer to the new computer. The programs were moved because the old computers were too costly to maintain, and because the new computers were so much faster. This happens in modern projects, too, when some special hardware feature becomes available, and specific use of it is to be added.

In modern times, the typical reason for a useful conversion effort is to gain functionality, gain speed, or to decrease the code complexity of the system to lower future development costs. For

instance, a company was using an enterprise management system written in Borland C++. The system had a desktop version, and needed a web-based system sharing the same database. Every user screen, because of the way Borland worked, had been written independently. A large number of new screens were to be added, and a large number were to be modified. Doing the additional work in Borland C++ would have taken a certain amount of effort. Converting the entire system to C# (chosen because the programmers at this company already knew and used C# for another project), plus the effort of the adds and changes required, took less effort than the job would have taken if the software had stayed on the Borland system.

3.12.2. Lemmings

Too often, a project will undergo conversion for no other purpose than "everyone else is doing it".

The business advice: Don't undergo a conversion project unless at least one of these goals will be met: (a) The project will be more functional in some manner (note that this will mean producing a functional requirements specification for the new functionality); (b) The resulting new software will execute more quickly; (c) The resulting system will be easier to maintain, such that the cost of the conversion effort plus the cost of future maintenance would be less than the cost of maintaining old system. If you can't say that one to three of these goals will be met, then you should not be doing the conversion. Be careful too when incorporating new language requirements into the system. For instance, the old system was written in DB/2, C, and CL, and the new system is written in Mongo and Hadoop and Java and JSP and a custom tag library and MQ and HTML and CSS and Javascript and Typescript and Bootstrap and Angular — your job of hiring the next programmer just got way harder.

The career advice: When management tells you to do something, decide whether you agree with it or not. If you agree, good. If you disagree on a technical basis, let management know, once, politely. If you disagree on a moral basis, refuse. One of your authors knows from first-hand experience that the latter will get you fired.

3.13. Mergers & Acquisitions

When a large company buys out a smaller company, or when two companies of approximately the same size merge, they will generally combine their two systems. This will happen in one of two ways.

In the acquisition model, one system is seen as being better than the other. Typically, it's the larger company's system that prevails, sometimes because it's really better, and sometimes just because the larger company is already using it. Sometimes the smaller company's system is used purely because it's better. There have even been cases where the larger company buys the smaller system purely because its system is better. In such cases, the data from one system will be moved onto the other system. This process is called Extract, Transform, and Load (ETL) or Extract, Load, and Transform (ELT). Sometimes this can happen purely by asking for the data to be transferred, but often a specification of how the data is to be transferred and/or translated is

required. In this sort of case, it's not a software requirements specification that gets written, but a data requirements specification.

In the merger model, there is almost always the merging of data as described above, but also the merger of functionalities. Both companies will have functionalities they need to maintain. At first, their respective systems will operate side-by-side, but generally the functions of one will be added to the functionality of the other, and the data will often become a merger of the two original data sets. In this model, a specification is usually written handling both software and data requirements.

3.14. Return on Investment - ROI

The key point in all of these is Is It Worth It? Any project, any sub-project, even any feature, should only be taken on (and thus specified) if the project is worth it. There is a term, "return on investment" (ROI), which is the benefit divided by the cost. If that number is higher than 1, then do it. It the number is 1 or less, don't do it. Note that cost and benefit are not always monetary. Sometimes one or both will be in terms of capabilities, speed, some sort of "goodness" factor, customer good will, or other factors.

There will always be some amount of uncertainty in the ROI. The higher the ROI and the more certainty, the greater the priority for that project, sub-project, or feature set.

3.15. How Things Can Go Wrong

The two main ways to miss the goal here are to leave things out, and to put things in that don't belong there. This sort of problem can happen all at once, or through slow scope creep. Leaving things out is not that big of a deal in the Agile methodology, because things can be added as needed. How to avoid the problem: Ask these questions: Are there any features we need that we don't have? Is there anything in the requirements list that we don't need? Is the entire project, and every feature in it, worth what it will cost?

3.16. Discussion Questions

1. When is it better to upgrade to something more modern?
2. When is it better to stick with what works?
3. Is it better to work for a big, stable company, a friendly medium-sized, or a small start-up? Hint: You'll have a different answer from others, and if you're true to yourself, your answer will be right.
4. Is it better to work for the start-up, or to be the start-up?
5. What do you think is the most likely project reason at a non-internet large company?
6. Same question for large internet company?
7. Same question for medium company?

4. Methods of Elicitation (Gathering the Information)

There are a number of elicitation methods described here. You don't have to limit yourself to just one method, and you don't have to limit yourself to only the ones mentioned here. These are all just suggestions and styles. The only real rule is: Use whatever works.

4.1. Existing Business Process Documentation (if you're lucky)

If you're particularly luck, meaning if management is particularly good at their job, then there will be existing business process documentation, and any existing computer code will be well documented, externally (as in design artifacts such as specifications, designs, as-built documentation, and manuals) and internally (as in meaningful comments and names). Always ask for these documents first, because it can save you an immense amount of time. But, beware, it may be out of date or incomplete.

4.2. Discussion Groups

Discussion groups are generally considered a very informal way to get things done. It can help to have muffins and tea available, or something similar to break the ice. Everyone should generally be encouraged to speak up, without taking turns if the group is well behaved in that manner. Just the same, at least one person should be taking notes, and the sessions should be recorded when feasible.

4.3. Interviews

Interviews are the more formal approach. It can be one on one (the typical method), or there can be multiple interviewers or multiple interviewees at one time. In a more formal interview, you'll generally have some questions written down or in mind before the interview starts. Again, keeping notes and/or recording is best. One on one interviews are also appropriate if and when you don't want one person's answers to your questions tainting what another might say.

4.4. Try it on Sample Groups

Very often, you or someone else will come up with an idea and think "let's put this in the project". Is it a good idea or a bad idea? One way to find out is to produce a prototype of the thing. Maybe a picture. Maybe something that sort-of works. Maybe a few versions of something that actually does work. A development organization can try various concepts on groups of people and see how they work out. If you're trying something on a customer group, the question is usually "do they like it?". If you're trying something on an internal group, the question is usually "is it more accurate?" or "is it faster to use?".

4.5. Tribal Knowledge

Business processes should be documented, but often they're not. Certain people know things. People tell others how to get things done, but never seem to get around to writing them down.

Or, the knowledge might be written down somewhere, but knowledge of where is little known, and only by word of mouth. Those are the cases that are called "tribal knowledge". It will often be your job to be the "archeologist" or the "anthropologist". Be aware that there may be multiple "tribes", in that one group may not know how the other group gets things done, and that these "tribes" may in some manner be in competition with one another, even if they work for the same company.

In such cases, optimally your work product will be both documentation of the existing policies & procedures, and of the new requirements. You may be able to find out who knows what, and be able to get them to document it, or you may have to do a lot of detective work on your own. The "tribe" may be cooperative, but they may also be very protective of their secrets.

4.6. Questionnaires

The simplest form of questionnaire simply asks "What do you think we need?", More pointed (and useful) questionnaires will ask some "closed" questions, such as a question that has a numeric answer, questions with yes/no answers, and questions with multiple-choice answers, but also open-ended questions, such as "What else do we need?", "Is there anything we should avoid?", "Is there anything that's obsolete that should be removed?", and "Are there any business processes you'd like to be streamlined?".

4.7. Observation

Direct observation of what's going on is often the best way to gather information when the job is to streamline or automate current processes. Below, we detail some observation methods.

4.7.1. Individuals' Work

One of the best paths to process improvement and requirements gathering is to observe someone work, as closely as possible. Doing so will allow you to see what really happens, not what the worker remembers happening. Direct observation also allows you to do timing, and to ask why certain things are happening or why they are being done. You will often see that a person is going through too many steps to do something. These sorts of scenarios happens a lot:

The worker does some task (which they asked you to automate), and then does some second task (which they may or may not have asked you to automate). If you ask how often Task B follows Task A, the answer is "always". In this sort of case, you want to automate the combined AB task as a single thing, typically.

Someone will pull data from System A and enter it into System B. This is a special case of the above. These two systems should have an automatic way of handling things.

Sometimes you may see some sort of work being done with no apparent purpose. If the reason for some task or action is not obvious, ask why. If the answer seems hard to believe, check it out; you may just find that one or more steps is completely useless, and amounts purely to "tribal custom", "inertia", or "historical relics". Although in this case you don't get to document any

requirements, bear in mind that as a business analyst, this sort of process improvement (elimination of work), goes straight to the company's bottom line.

It is even possible that you will find out that words that you thought meant one thing can mean another. In the worst case scenario, one worker had said in an interview that she had cut and pasted certain data from three spreadsheets onto a new spreadsheet on a monthly basis. Upon observation, she was seen to do this cutting and pasting with actual scissors and paste.

4.7.2. Listen to the Coworkers Talk

If the company has an open seating arrangement, you'll hear some chatter between the workers. Listen to that chatter. If they're asking each other questions, then they need those answers. If they're asking each other to do things, then they need to have the ability to do those things, or they need a situation in which those things are done automatically, or in which they don't need to be done at all.

4.8. Do the Job

Taking over someone's job, for a day, or even for a few hours, or just being trained to do their job, can give you a great insight into the job. For instance, for one business analyst and code analyst role at Northrup, the new analysts were first trained how to rivet and how to QA (perform quality assurance on) the riveting job.

4.9. Process Improvement

The best you can do as a business analyst is to simplify the processes the business goes through. If you can cut out steps, make the process faster, cheaper, more reliable, or in some way better, with or without that affecting the requirements document, do it.

4.9.1. Ask Why

If you're not 100% sure why something is happening or whether that thing should happen, ask why. If you're not 100% certain that the answer is correct, ask someone else too.

4.9.2. Cutting out Steps

Look for steps that can be left out or combined. This can happen three ways. First, if the result of a step is not used, or is not useful, get rid of the step. You may even find that you'll cut out two or more steps with one action. For instance, if you cut out a step that produces a useless report, you also cut out the distribution and reading of the report. Second, if a step can be simplified, then you're ahead. If two or more steps can be combined, then you're effectively one or more steps ahead in the process. Cutting steps out may take modifications to the procedure manuals (and/or web-sites or SharePoint documentation), telling people what you've discovered, and/or retraining people. Combining or altering steps, of course, means that requirements — additions and/or changes — will go into the requirements document.

4.9.3. Special Case: Excel

In the business world, Microsoft Excel seems to be a special case. The program is great at what it's designed to do, but people seem to find a lot of uses for Excel that it's not good at. Any time an Excel sheet comes out of a program, be suspicious. Here are a number of ways in which Excel is abused:

Sometimes Excel is used as a transfer mechanism. People download their data from System A into Excel, and then upload that same data to System B. In all such cases, even and especially if some data transformation is being done, all such data movement and transformation should be automatic, either on manual demand (such as from a button or menu item), or scheduled (optimal when possible, since scheduled items happen exactly when they're supposed to, whether remembered or not).

Often someone will pull (download) data into or as an Excel file, and then add calculations into that spreadsheet. If that person is doing one-up work (work that will only be done once), then that person is using Excel as designed. But, if the process is going to be repeated, then the work should be moved into the applications layer or into the database layer, as appropriate.

Reports generated by a computer system should be generated in a read-only manner. There is a big tendency in some organizations to pull a report in Excel format, and then to manipulate the data. Generally, in this sort of situation, the report is output as an Excel file, complete with active formulas. People will then adjust numbers, and the totals and other calculations are automatically adjusted by Excel. There are two problems in this scenario. First, the reports themselves become unreliable, in that it becomes hard to tell if the report was generated by the system or by hand (and even one cell changed by hand makes the entire report count as having been done by hand). Second, likely data that should have been entered into the system is not entered into the system, and the system continues to have errant or inadequate data. For instance, someone requests the annual profit and loss statement (P&L). It is generated as an Excel spreadsheet, complete with totaling formulas. The person then notices that an entry was omitted, and inserts a line and adds that entry. The final report is correct (hopefully), but the data in the system is still incomplete, and now at variance with the report delivered. The best practice here is to output the reports in the least editable manner available, such as PDF (or for organizations still using it, paper). In the PDF (Adobe's Portable Document Format) scenario, the person who notices that an entry was missing is forced to do the right thing, which is to enter the correct data and then rerun the report.

Another suspicious item is any instance that involves VBA (Microsoft's Visual Basic for Applications) macros or any other type of macros or plug-ins being used in any type of data-containing file, especially Excel files and Microsoft Access files. The best paradigm for handling data is the Model-View-Controller (MVC) paradigm, in which the model (the data and the metadata, usually contained in the database system), the view (the report or the data entry facility [screen or form]), and the controller (the program [desktop or web]) are kept separate. Putting VBA code into data files often means that when data and/or programs are updated, they no longer match.

Sometimes Excel forms are used as offline data entry forms. Although there are cases when this is the best way to go, those cases are few and far between. Any time you see an offline form in use, ask whether the form could be better done as a paper form, a web page, or a data entry screen in a desktop application.

The bottom line is that any time you see a report being generated in Excel or any other editable format, be very suspicious.

4.9.4. Adding Automatic APIs

An API is an Application Programming Interface. What that means has changed over time. Originally, it meant either how the desktop application interacts with the user, or how a subroutine package can be called and what data it delivers. As of this writing, "API" most often means the manner in which HTTP-based (Hyper-Text Transfer Protocol) distributed applications communicate with each other. It can also mean other socket-based protocols — including custom protocols — used for the same purpose. "API" can also refer to file formats.

If an application is going to be written by two or more companies, or even by two somehow-separated groups within the same company (such as having the project split between two groups who will each work on their own and integrate their deliveries later), then an API must be developed between the two components. This might happen once the project is in the coding phase, but can happen as early as the original requirements definition phase. Sometimes a new project will use existing APIs from existing external projects. For instance, an eCommerce application might use an existing API from a payments processor and a fulfillment center's system.

An API example: A certain dog trainer has a web-site that, among other things, sells products. At some point, the customer will give the web-site a credit card for the sale. The program running at the web-site will then communicate with Authorize.net, a credit card clearing company, through Authorize's API, asking to put a lien on the card for the amount of the sale. When Authorize tells the web-site that the lien is in place (through the same API), the web-site then uses another API to tell One World Distributing, a warehousing and shipping company, to ship the product. Once the shipping company tells the web-site (through that second API) that the product is in stock and will ship, then the web-site uses the first API to tell Authorize to convert the lien to a sale, and then tells the customer, through the web-page, that the product is shipping.

To an extent, paper documents can be treated as an API. Various sorts of automatic document scan, recognition, and data entry programs can be written in a number of systems, most notably Kofax and Google Vision. Electronic documents in various formats (meaning both various document extension types and various lay-outs) can be used, too. Often, programs have screens that list various things, and have a button to enter a list of those things as a spreadsheet document. If and when appropriate, include such a button. But, if you can have a button at a higher level that just reads a document and figures out what to do with it, consider that.

Here's an example: At a particular insurance administrator, carriers would submit annual pricing plans and lists placing zip-codes into regions. Employers would submit lists of eligible employees. The bank would deliver a list of checks for the day, and a host of other forms would be delivered to the system. Rather than having a dozen menu entries for reading various types of documents for each purpose, there was one "Read Document(s)" button, that figured out what type of document is was looking at, and did whatever was appropriate with it.

In a matter relating to "figures out what to do", always be prepared to think outside the box. For instance, one company had a rather tight options entry dialog screen for a report in which there were about 40 controls already. These included things like overall report format, paper size, number of columns, sort order, symbols to be used or not, suppliers to be exclusively used or excluded, contract types to be exclusively used or excluded, geographic area, types of services, languages, and others. Several new controls had to be added. The "in the box" thinking involved deciding how to cram the new controls into the window, or adding a "Next" button that would lead to more controls, or dividing the dialog into tabs, and moving some of the controls to the new tab. But the out of the box method was to get rid of all of the controls, and to have just one text entry area with the label "What do you need?". An example report request for the final text entry area would be "I need a list of all general practice dentists in California and Nevada that speak Spanish, with handicap symbols, valid for July of 2021".

Figure 1: The old way:

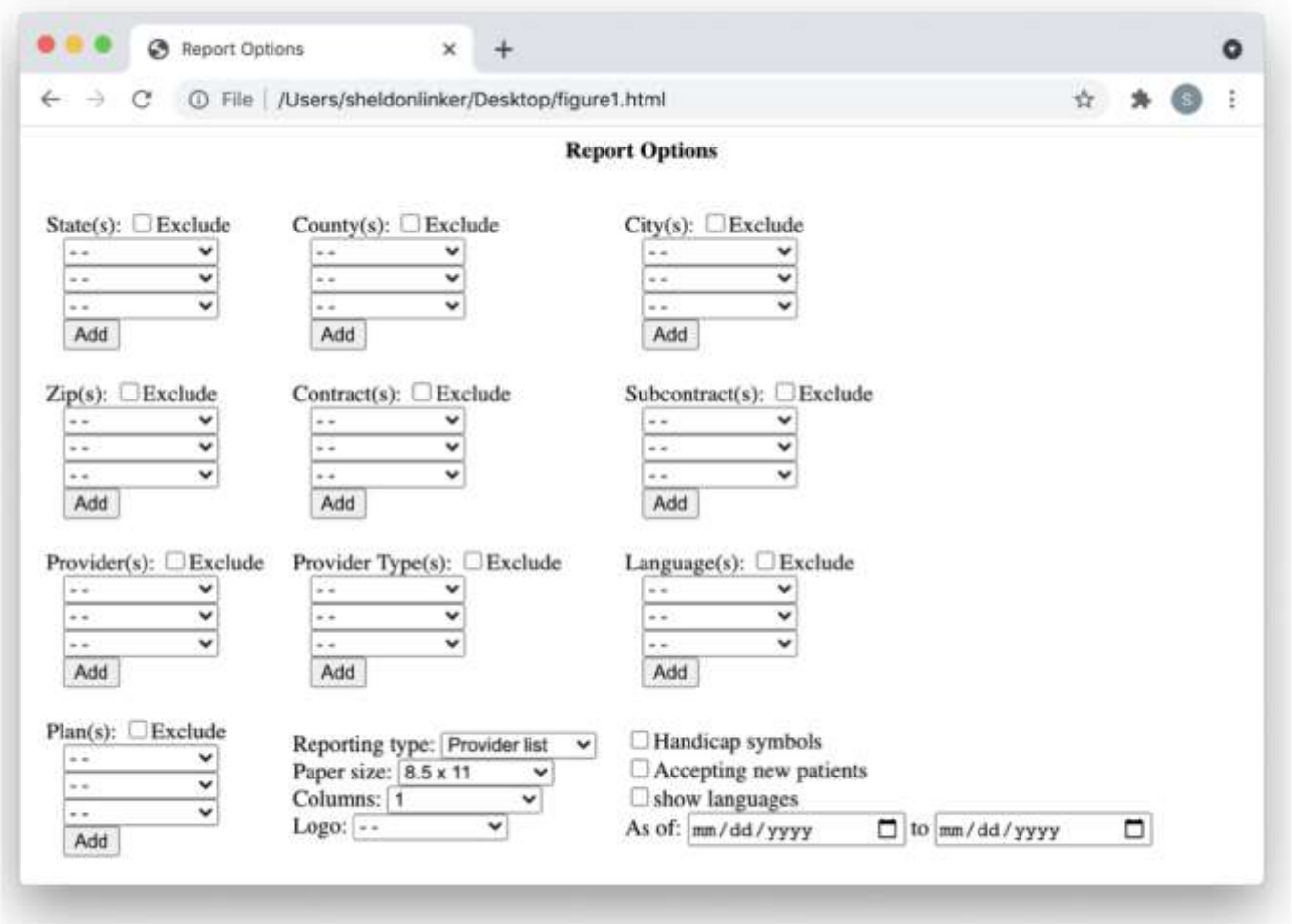
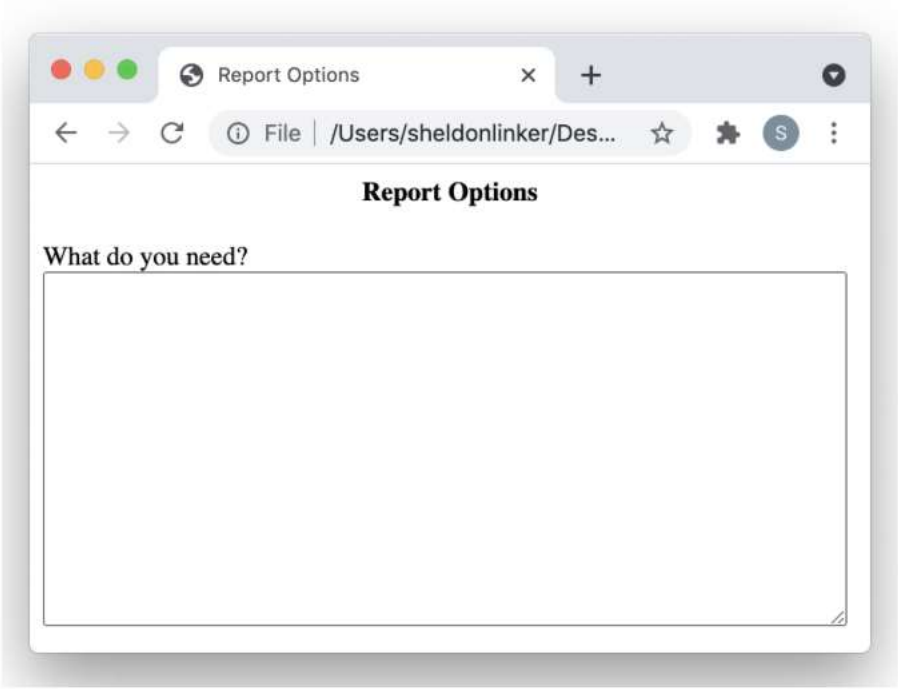


Figure 2: The out-of-the-box way:



4.10. "JAD Sessions"

A JAD Session (Joint Application Development) is really just a more formalized form of a meeting or discussion. There are specifically defined roles and responsibilities. The person who puts the meeting together, prepares for it, sets its agenda, moderates it, and hopefully solves any conflict is called the "facilitator". The facilitator sends the invitations (or in some companies, summonses). The facilitator also prepares the agenda of the key points to be covered. One or more of the attendees is the person or people who own the underlying need. This may be the customer, internal funder, champion, or project manager. Next, we have the expert participants. These people will typically be subject matter experts (SMEs) in some form, either because they know a lot of key facts, because they're highly skilled and experienced, or maybe just because they've seen firsthand and often the nature of the problem to be solved. Last, we have the role called the "scribe", which sounds like the secretary, but is actually the person who documents the requirements that come out in the meeting, and writes the specifications. During the meeting, or JAD Session, requirements will come out, but there will often be issues that come up that need further investigation. The scribe documents these too, and the facilitator will schedule new sessions with their own agendas to deal with those issues. In this manner, inputs are taken, and at the same time, approval is usually given.

The process tends to work for two reasons. One, being a formalized session, with just about everyone in attendance, people don't get lost in side-tracking because of the formal agenda and moderation. Second, the placebo effect is helpful. When the attendees are "just chatting", they feel a lesser stake, but when they're in a "JAD Session", well, now things are happening at a high level. The downside is that with mandatory attendance of a lot of people, JAD Sessions are expensive.

4.11. Demeanor: Asking, Demanding, and "Good Cop Bad Cop"

Generally, in a business setting, everyone is happy to communicate with the business analysts, either because it's their job to do the bidding of the business, or because they realize that in cooperation with the business analysts, they're going to get what they want. Not a bad assumption, since both tend to be true. However, you'll get the occasional person who won't cooperate. There are typically a number of common reasons for this, including forgetfulness, being too busy, not feeling like it, or fear that they're going to be automated out of a job. That last one can be a real concern, since at some companies, that's what happens. If you have any control over that sort of situation, the kindest approach (and safest not to cause an organized labor action, or worse, a disorganized labor action) is to make sure that any reduction in staff or reduction in force (RIF) is done solely through attrition or reassignment.

When someone is not cooperating with you or your team, there are several approaches to the situation. One is to just invite again. Another is to order compliance. In some organizations, ordering compliance is the way to go (the military, for instance). In other organizations people will stare at you blankly, or worse, if you give them an order. Asking nicely, up to begging/pleading can work. Sometimes what does the job is explaining how you're here to help them directly helps. But, one technique that can be used is to have one pushy person, and one kind person. The pushy person more or less demands an answer, and the kind person then asks separately and politely. The person reticent to talk will often confide in the kind person, rather than stay quiet. Use this technique only as a last resort.

You'll find that in most situations, the name you use should fit into the culture of the team, either as a team member, or as an authority figure. For instance, Robert Smith, PhD. might be introduced as "Bob", "Robert", "Robert Smith", "Mr. Smith", or "Dr. Smith", depending on the culture present and relationship desired.

4.12. Wiki? Google Docs?

There is much talk about using wikis (*wikiwiki* is Hawaiian for "community"). Like Wikipedia, any wiki is a collection of pages that refer to each other in no particular order. That makes a wiki great for looking things up, but terrible for attempting to read something from start to finish. Also, in a typical wiki, there are no provisions for pending changes, nor for one person to be in control. Sheldon cautions you against using wikis for any sort of specification document, and encourage you to use them for manuals (although no more encouragement than using standard documents for manuals).

Google Docs, on the other hand, are great for working on requirements specifications. Let's say that two people are in charge of the document, ten people are allowed to write comments into the document, and a hundred are allowed to read it. One of those two people can establish the document, and send an Edit invitation to the other editor, send Comment invitations to the other ten people, and send Read invitations to the rest. A Google Document can have multiple editors, commenters, and readers all at the same time. The editors would be adding their own text, and reacting to the comments. Typically, each comment will be a request for a change (explicitly or implicitly). The change request will be accepted and used, or rejected with another comment.

When the change is accepted and used, the comment is typically marked as Resolved, and when the originator of the comment sees the rejection reason, that person will often mark the comment as Resolved, but maybe argue the point further.

Documents should have versioning information. In PDF and paper documents, those are typically in a table of versions, and change bars are used to show the latest set of changes. Google Docs' Track Changes function allows for the same thing. Best practice would be to have official versions as of a certain date and time, and to have a "live" version which has not yet received a version number.

4.13. How Things Can Go Wrong

Things get missed. Bad data gets delivered. Neither one of these can be considered a failure on its own. They're just correctible risks. You're virtually never going to get 100% of the information on the first pass, and you're virtually never going to get 100% reliable information. Just be aware that this happens, and keep track of confidence level: Certain, Probable, Likely, Scientific Wild Guess (SWAG), or Wild Guess (WAG).

4.14. Discussion Questions

1. Do you think that there is one of these techniques that is likely to work better than others? Which?
2. Have you used any of these? Do you have a favorite?
3. Have any of these been used on you? Did you feel more comfortable in one manner than another?

5. General Styles of Project Management

There are a number of well-defined project management styles, only some of which we will cover here. We could write an entire textbook on those styles, so don't take the fact that a style name is missing here as meaningful. Only a few styles are being pointed out to show, later, how requirements gathering and requirements specifications vary from one project management style to another. We'll only go into enough detail to make that comparison meaningful.

Note that like everything else in the world, each style of management has a reason behind it, and each methodology has its own advantages and disadvantages.

5.1. Waterfall

The Waterfall model assumes that requirements will be stable, and that the work will flow from stage to stage, never to return to its starting point, much like a waterfall. Within a Waterfall project, the Requirements team first gathers the requirements and writes the requirements documentation. Often, this (or these) specify what hardware and/or software is to be built, to a great degree of detail. This is often called "the A-Spec". Following this phase, most or all of the Requirements team will leave the project, and only then will the Design team join the project. The Design team will do the system's design work, laying out how the components will work internally, and how they'll fit together. Any hardware will be designed, and hardware/software interfaces will be designed, and interfaces to or from the outside world will be specified, and often the internal API, class structures (if any), messaging (if any), and database structure (if any) will be designed. The result of this stage is a design specification, or "B-Spec", which allows the programmers to pretty-much work on their own, because there should be no negotiation on how anything should work or interface left to do. At this point, all or most of the Design team will leave the project, and the Engineering and/or Programming team(s) will join. The programming teams will write the code, the unit test plans, and test their code. If there are any additional facts to add, which there certainly are when hardware is built or manuals are to be written, that documentation is added, called the "As Built Spec" or "the C-Spec". At this point, much of this or these team(s) leave, and the Integration-and-QA team comes onto the job. It's their job to get the system integrated and delivered.

In the real world, these phases don't end or start all at once. Some teams will finish ahead of other teams, and some people will finish ahead of other team-mates. It seldom happens that the project manager will have enough people to have all the tasks in a phase done all at once, and so the teams and their members chew through the tasks, which means that some tasks will be pushed to the next phase before others are even started. Thus, even on a Waterfall project, deliveries will often be in phases.

There are some money and time advantages to the Waterfall model: Since the teams are only on the project while they're needed on the project, there is a cost advantage there. There is also an overall speed advantage to the company if they can keep the teams busy, in that with several different teams, each of those teams can be working on a project at the same time.

And, there are some well-known disadvantages to Waterfall, too: Waterfall relies heavily on everything, especially the requirements, remaining completely stable. If the project moves from one phase to the next, and something in an earlier phase needs to be reworked, there can be a problem, because the team(s) that handle the previous phase(s) are gone, and either need to be recalled, or someone else has to pick up work they're not familiar with.

Appropriate pricing models when Waterfall is being used for outside projects (also known as bespoke projects): If the customer presents a set of needs up front, and that set of needs leads directly to a set of requirements, then the project can be done at a fixed price. If things are not nailed down until the requirements are fleshed out, then the requirements can be done using a time-and-expenses contract, and the rest done as fixed price. Of course, all contracts can always be done as time-and-expenses. In the split version, some companies may choose to do the requirements gathering on speculation (which is like free, except that the price is figured into the fixed price portion), hoping to get the big sale. Typically, fixed price contracts will involve some money up front, then payments as deliverables are delivered, and then some final amount on final acceptance.

Sheldon's advice (not widely accepted): When the requirements are extremely stable and well-known, go with Waterfall.

5.2. Rational Unified Process

RUP® or Rational Unified Process® is a tweak on the Unified Process by Rational Software, which is now part of IBM. In many ways, it's part-way between Waterfall and Agile, in that it's really more Waterfall-like, with some hints of Scrum.

RUP sees the work to be done in a project as (a) business modeling, in which the existing business rules and processes are flushed out, (b) requirements, in which the requirements for the system to be built are discovered, categorized, and documented, (c) analysis and design, in which the requirements are analyzed for patterns, and a design for a system that will satisfy the requirements is designed, (d) implementation, in which the system is built and/or coded, (e) test, and (f) deployment.

Despite having these six task areas, RUP divides the project into four phases, which are (1) inception, which is the ideas phase, (2) elaboration, which is the research phase, (3) construction, in which things get built, and (4) transition, in which we move from building to installing and running. RUP sees testing as happening all along the process. Within each of these four phases, there are "iterations", which seem inaptly named, because they're more like Scrum's "sprints".

In Waterfall, there is often a hard boundary. We design, then we stop designing. We build, then we stop building. We test, then we stop testing. But, in RUP, this is seen more like a collection of relatively flat and wide bell curves. We have our business modeling, which peaks and then starts to fall off. While business modeling is at its peak, requirements engineering is in its ascendancy. While requirements engineering is at its peak, analysis and design is at its ascendency, and so on. But, during all of this, testing has flurries of activity in each "iteration", as various "artifacts" (documents, code, or anything else) are delivered to the QA people.

5.3. Agile

Agile is a big category. It involves any methodology where the requirements are expected to shift (or are allowed to shift), and describes the fact that the organization will have to remain agile in its approach, so that it can keep up with those changes. In just about all Agile variants, there is a queue of things to be done, called the "backlog". As the project progresses, the requirements gatherers continue to gather requirements and update the requirements documents. The programmers start very shortly after the requirements gatherers do, and the QA people start the same time as the programmers, so that they can plan their tests while the programmers are producing what needs to be tested. The requirements presented to the programmers are not the whole set of requirements, but the requirements for what has to be done now.

Advantages: Very little should throw an Agile project for a loop. The whole team is there the whole time, and should be able to handle anything.

Disadvantages: The whole team is there for the whole project, and as things change, tests must be rerun often. Thus, virtually all tests need to be automated. This can have a higher cost for the production of the tests, and of course, there can be a higher cost due to the entire set of teams remaining on the project.

Appropriate pricing models: If development starts before the last requirement is known (which is the whole basis behind Agile), then it's almost impossible to know the extent of the costs. Thus, time-and-expense estimates and time-and-expenses contracts are how these sorts of projects are normally billed out.

Sheldon's recommendation: Except in the Waterfall case explained above, and periodic releases of packaged software, go with Agile.

Below are described several "flavors" of Agile. Note that the flavors described below can very easily be mixed within a project, on a team-by-team basis.

In all forms of Agile, the priority of each task is important, as well as whether one task needs to be completed before another task's implementation can start. It is often fully or partially on the business analyst/requirements documentor to make this determination.

5.3.1. Scrum

In the Scrum version of Agile, there are fixed periods of 1-6 weeks, called "sprints". At the start of each sprint there is a meeting run by the "scrum master", in which assignments are handed out to developers from the backlog, or developers are allowed to choose assignments from the backlog. Generally, assignments are graded as to the expected amount of effort to complete. Sometimes these are estimated in hours or days, sometimes in "points", which are just a relative order of difficulty (because some people work faster than others, "hours" for one person might not match hours for someone else), or just "small", "medium", and "large". A developer may be assigned multiple tasks for the sprint, just one, or one task that may span multiple sprints. One

or more business analysts are usually present at the sprints' "kickoff meetings". Each day, there will be a "daily standup meeting" in which the accomplishments and problems of the previous day will be discussed, as well as the plans for the current day. At the end of the sprint, there will be a "retrospective" meeting.

Sprints are usually numbered, and which sprint you're on would usually have to do with which team you're on. For instance, the Requirements team might be on their Sprint 10, while the Development team is on their Sprint 9, and the QA team is on their Sprint 8.

Advantages: Even the least self-directed people will have clear direction in this scheme. The daily standup keeps everyone producing and appraised of the situation.

Disadvantages: (a) There is a lot of meeting time involved. (b) Let's say that there are a number of tasks that are slated to take 4, 5, and 6 days. If the sprint duration is 5 business days, then 4-day assignments are a bit too short to be convenient, and 6-day assignments are a bit too long to be convenient. These can lead to rushed and idle time. (c) The daily standup can be misused to keep everyone pushed to their limits, which can lead to burn-out. This can also lead to the "boy who called 'wolf'" problem, in which something comes up that really does have to be accomplished quickly, and everyone treats it as if it were the standard priority.

5.3.2. Kanban

The name "Kanban" comes from the Japanese word "看板", meaning sign-board. The idea here is that the backlog was posted in the form of index cards on a pin-board, and a developer ready for the next assignment would take one of those cards, evaluating priority, estimated time to complete the task, and personal affinity for the type of work on that assignment. Few if any companies still use cards and a pin-board, but the concept is certainly in use, in a more automated fashion. In this model, either the project manager (PM) or the business analyst will be in charge of the (typically virtual) "cards".

Advantages: There is little meeting time involved. Tasks are not forced into an artificial time-box.

Disadvantages: Anyone in this system who is not self-directed is likely to be less productive.

Sheldon's recommendation: Categorize your workers into the self-directed camp and the "needs direction" camp. Run one camp using Scrum, and the other using Kanban. Just before a Scrum kickoff meeting, the Scrum master will have to assign a number of "cards" to the Scrum team, and during the meeting, assign those tasks to individuals.

5.3.3. Extreme

Extreme Programming (XP) is a lot like Agile, except that we concern ourselves only with one function or functionality or feature at a time. In each sprint or cycle, we pick the single most important item, or the single most foundational item, or the single easiest item, and we implement that. But, we don't just implement it. Let's say that a feature will involve four

functions. First, we write the test driver for the first of those functions we feel like writing, and add the test driver to the master test program. Then, we write the function that we just wrote the test driver for. Then, we run the master test program, and it tests our first function, and reports success and failure. After we're at "success", we can write the second test driver and the second function. After a few days, weeks, or months of this, we have hundreds or even thousands of automatically running tests, and if we ever break something along the way, we should know about it pretty quickly.

Another main tenant of Extreme Programming is that we just do the minimum to clear a single requirement, knowing full well that a lot is going to change in the next few hours or days or weeks. This affects the requirements writing process, in that your software requirements specification will not be some tome delivered from on high as in Waterfall, nor the weekly news magazine of Agile, but more like the daily newspaper or hourly radio news update. In XP requirements engineering, it's more like a constant stream of individual requirements to the programming team. And forget prioritization. The programmers will pick up individual requirements one at a time, in whatever order they see fit. Now, if the customer really wants one item before another, you can let the programmers know, and hopefully that will have some effect.

Although you'll be delivering your requirements to the programmers in groups or one at a time, they'll typically be reporting back "done" one at a time, or in groups. When Sheldon worked an XP-style project, they typically had releases during lunch and at end of day. They'd install before reporting "done". So, management needed to keep a list of requirements, and whether they were done or not. Management wouldn't know what was being worked on, only what had been done.

One more aspect of XP that doesn't make much difference to the requirements engineers: In XP, programmers work in pairs. That allows one to do the work while the other is reviewing and checking the work. So QA and programming happen in very small teams. Each QAs the other's work.

5.4. Hybrids

For the most part, anything that's not in or very close to one of the categories above is considered a hybrid model.

5.4.1. In General

Like most things, hybrid models happen for "good" reasons and "bad" reasons. Whenever someone suggests *any* model, including a hybrid model, the project manager needs to assess whether the model is a good fit or not.

5.4.2. Wagile

"Wagile", or "WAgile", is claimed to be a hybrid between the Waterfall model and the Agile model. We use "claimed", because Wagile models are not usually things that have been well

thought out, but rather things that have just happened. Waterfall and Agile are so different in their goals and assumptions that Wagile is a lot like trying to go north and south at the same time. You may get somewhere, and you may not.

Meeting styles and task assignment styles alone are not enough to consider something a hybrid. For instance, in assigning tasks in a Waterfall project, the project manager may simply assign tasks to people or may let them pick from available tasks. Neither of those makes the project a hybrid. There may be weekly one-on-one or group status and progress meetings or a daily group meeting. Neither of those choices creates a hybrid either.

5.4.3. Spiral

Spiral truly is hybrid to some, and a distinct methodology to others. In the Spiral methodology, there are months-long Waterfall-like development efforts, each leading to a released version of the software. That's followed by an overall retrospective and planning for the next session. So in a way, it's like Agile with 6-12 month sprints.

Sheldon's recommendation: This sort of methodology is optimal for the production of packaged software products. You can use a Waterfall methodology for version 1.0, and then when you go to trade shows, embed the requirements people with the Sales team, to get feedback from customers and potential customers on what should be released in the following versions.

5.5. Command & Control and Negotiation

In some projects, there's a clear top-down control system, and there are orders, rather than negotiation. Those tend to be the easiest jobs to deal with. You won't get into an argument if you can't argue.

But, when that control structure doesn't exist, or you're dealing with outside customers, there will often be negotiation. That can be your department negotiating with another department about what can be done when and for how much. It can also be your company negotiating with a current or potential customer. In the interdepartmental negotiations, you're all on the same team, and generally only really trying to decide what's reasonable, who gets to have the easier time, and possibly even budget share. When you're negotiating with an outside customer, you're on opposite teams. You want to make a profit on the deal (usually money, but occasionally something else, possibly even an intangible), and so does the customer. A deal should only take place if it's going to be a win-win situation — that is, both sides will be happy with the outcome.

That brings us to some key points of negotiation, some of which always apply, and some of which only apply to some situations:

- In an outside negotiation, be prepared to walk away. Remember that the negotiator for the other side is also prepared to walk away. That's because neither of you wants to get stuck in a bad deal.

- Know your facts coming in, but also be prepared to say that there's something you don't know and will have to find out and get back.

- Keep it cordial. (Sheldon negotiated with Stan Lee once. Stan was the hardest bad*** ever, while at the same time, being the nicest guy in the world.) Even if you lose the job, you want the opportunity to bid next time.

- Know what you're trying to get. To the extent possible, try to know what the other side is trying to get too.

- Don't lie, but at the same time, if you're dealing with an outside company, you don't have to tell the whole truth either. For instance, if your estimate is $50,000 to do a job, and someone from another department tells you that there is $100,000 budget for the job, let them know it's a $50,000 job. But, if you're dealing with an outside customer, and you know it's a $50,000 job, and they offer $100,000, the correct response is something on the order of "Can you do $125,000?".

- Have a presentation ready to go, but remember that you may have to throw that presentation away and head off in a different direction at a moment's notice.

- Of course, try to end with a sale or agreement or whatever is appropriate

- Don't get emotionally involved.

- Don't get worried.

- As with all engineering matters, have the ego of a Zen master — present only when needed.

- Be realistic.

5.6. Others

Of course, there are other styles of project management possible. People come up with completely new ones, people make hybrids (intentionally and often unintentionally), and some have no clear methodology at all.

5.7. How Things Can Go Wrong

Picking the wrong project methodology, or at least the wrong major category of project methodology, can be a major financial issue, even if everything technical goes just right. Fixed price projects can be a disaster using Agile. Projects with the likelihood of change can be a disaster with Waterfall. Don't just pick a method and do everything that way without first considering other options.

In a recent project, a large manufacturing company claimed that they were working 50% in Waterfall, and 50% in Scrum. To an extent, that was true. But the problem was, 25% was from Waterfall, 25% was from Scrum, 25% was from an overlap of Waterfall and Scrum, and 25% never got done:

Waterfall tasks	**Waterfall & Scrum tasks**
Some tasks didn't happen!	**Scrum tasks**

5.8. Discussion Questions

1. Do you think there is one best style of project management? If so, why?
2. Do you think that each style of management has an affinity for a certain set of circumstances? If so, what are some of those affinities?
3. Do you think that some style of management is never the best option? If so, what and why?

6. Things We'll Need for the Next Step

Before we go on, we need to describe certain things on their own, so that we can effectively use those definitions in the following discussions.

6.1. Preconditions, Post-conditions, Triggers, Requirements, Dependencies, and Assumptions

Precondition: Anything that must be true in order to use a module's functionality is a precondition. For example, if you need to insert your car's key into the ignition key-slot in order to start the car, then the insertion of the proper key is a precondition to starting the car. In the software realm, many systems have logging in as a precondition to the use of most functions.

Post-condition: A post-condition is any change to the system or to session-specific information as a result of executing some path through the function. For instance, let's say that you use the Unix "touch" command to create or modify a file. After you "touch" a file, it exists, and shows the modification date and time as the time that "touch" was executed. The fact that the file has been created or shows as modified is a permanent post-condition to having executed the "touch" command. If you log into an application, then being logged in is a session-specific post condition, in that it's only valid for the remainder of the session. In some applications, there are also document-specific post-conditions.

Trigger: A trigger is something that causes some action to occur, or some module to be entered, given that all preconditions (if any) are true. For instance, in starting a car using a key in an ignition switch, inserting the key is the precondition, and turning the key is the trigger. The post-condition is that the car has (hopefully) started. A trigger may be a user action, something that sensors pick up, a clock reaching a certain time, or any other event. In SQL databases, a trigger can be added to a table using a CREATE TRIGGER command, which causes some code to execute before or after certain events (adding, modifying, or deleting a record) occur.

Requirement: A requirement is something that the system (hardware, software, something else, or a combination of those) must be or do. In requirements documentation, requirements apply to the system being developed only, and not to users. Requirements apply to developers, but only indirectly.

Dependency: A dependency is not a requirement or a precondition, but something that will be required. For instance, once we've started the car, driving the car depends on having unobstructed land in front of it. If there's a tree in front of the car, the car will have to stop, back up, or go around it. The car also needs land in front of it. Driving into the water is not advised. But, you can start driving without having to worry about the trees or water. So, unobstructed land is not exactly a precondition for driving, since it's not something that we need to check before driving, but instead something that must be true for the entire time. We depend on that, and so it's a dependency. Don't get too hung up on this. There are things that are wobblers between preconditions and dependencies. In those cases, you'd just have to pick a categorization and go with it.

Assumption: An assumption is something that you reasonably assume to be true, or true in most cases. If something is both an assumption and a dependency, then list it as a dependency only. For instance, in a web browser, we might optimize the use of HTTP and HTTPS (Secure HTTP) web-pages, even though we can also use the browser for perusing FTP (File Transfer Protocol) and SFTP (Secure FTP) sites, because we can justifiably assume that that when the user types a URL without a protocol, that the user intends to use HTTP.

Primary path (or sequence) and alternate path (or sequence): The primary path through a function, functionality, or module is the path that we expect events to take. For instance, someone coming to a web-site and logging in would normally involve the steps (1) going to the web-site, (2) the web-site displaying, (3) the user clicking on "Log in", (4) a "Log in" page displaying, (5) the user entering a name and password and clicking on the submission button, and then (6) the system validates the name and password, and then redirects the user to the main page in the logged-in state. In one alternate path, step 6 would vary in that the name/password might not be accepted. In another alternate path, there might be a missing field. In the missing field case, depending on where checking code executes, this could be spotted in step 5 or step 6. Note that for any given function, there may be zero or more alternate paths.

Sheldon's recommendation: Always check such alternate cases in the equivalent of step 5 and step 6. Checking in step 5 prevents a post of missing data, and checking in step 6 prevents external bad data that can come about as a result of a hacking attempt or the failure of the protective browser-side JavaScript to execute.

Stimulus and response: A stimulus is just what the word normally means, but in a requirements specification, it only applies to the system. A response is also the normal English meaning, but in a requirements specification, it only applies to what the system does about the stimulus.

6.2. Functional and "Nonfunctional" Requirements

"Functional requirements" and "Non functional requirements" are probably not the best terms to have used for requirements documentation purposes, but those are the terms in use by the industry, so we're all stuck with them. As used, a "functional requirement" is a requirement that directly specifies a requirement on the system's main and visible functionality. A "nonfunctional requirement" is one that doesn't relate directly to any one function, or which seldom occurs. For instance, in a web application, the fact that a particular page will do a particular thing constitutes the page's functional requirements. The fact that the page and all other pages will be immune to SQL injection attacks is a "nonfunctional requirement". (An SQL injection attack is one in which commands in the SQL language are inserted into a text field, in hopes that the back-end program will be tricked into executing the SQL code.)

6.3. Functional Composition & Decomposition

In a large, complicated project, even after we've gathered or elicited all the requirements, we can't just sit down and document them all. We have to break things down into logical areas. There are two reasons for this.

In the bottom-up method, where you've gotten a lot of distinct requirements, you may need to group them up or arrange them into some sort of order that makes sense. In Agile projects, where you tend to get a lot of distinct requirements thrown at you from all sides, they use points (a key point of the story, not a numeric thing that can be added), user stories, themes, and epics.

A story point is a key point in the story. A story point is also a thing that happens or must happen. For instance, in the elevator example, one story point is that a passenger breaks the light beam. Another story point is that the doors retract because of the beam being broken. You combine these points into user stories that each make some sense on their own. (Don't confuse story points, or story elements, with Scrum's numeric points, such as a "3 point story".)

These user stories can be combined into themes, or related stories, and further combined into epics of related themes. In medium-sized projects, themes and epics will be the same things. When you combine them, try to list them out in one of these three orders:

- In the order in which they'll have to be built,
- In the order in which these use cases will logically or typically occur, or
- In an order that makes them easiest to understand, with the fewest forward references.

More often, though, you'll have to go through "functional decomposition". In an Agile project, this will involve figuring out what the epics are, and for each epic, figuring out what the themes are, and for each theme figuring out what the stories are, and for each story figuring out what the key points are. In a typical Waterfall project, you'll be figuring out what the systems are, and within the systems, subsystems, then assemblies, modules, and functions, and finally key functional points.

An example from one of Sheldon's assignments: We were going to build a big automated factory to put components together. Robots would have to get the components, visually line them up, weld parts together, and then send those built items to other parts of the factory. The builder robots would interface with robotic conveyor belts, and there would be robotic supply robots, supplying the welders with solder, for instance. So, we've broken the factory concept down to three types of robots. Now, onto the welder robots. The original design called for the welding assembly, the vision system, a manipulator to get things from the conveyor to the work stage and back, a work stage that could grip things, and a user interface used for training. (It turned out that the vision system working together with the welding system could move parts around adequately, so the manipulator was removed.) So, now we've got the robot's components. Within the welding assembly was the feed mechanism for the solder, the weld unit, the sensors, the various axes of motion, and so on. Within the axes of motion, there were various directions.

Basically, you break a big problem down into a number of smaller problems, until you have one of a size clearly documentable as a series of requirements. (When the programmer gets the requirements, those will be broken down into smaller and smaller problems, until each is one line of code.)

6.4. How Things Can Go Wrong

Leaving out preconditions or dependencies can mean that extra error checking has to be done later, or worse, a situation in which the needs and facts are at odds. Getting the triggers wrong will often mean that a needed function never comes into play. Not stating your assumptions can mean that the entire project is built on a false premise, which can cause a 100% failure when first used, or worse, that failure can occur after acceptance. Leaving out the "nonfunctional" requirements can lead to security flaws and general dissatisfaction.

6.5. Discussion Questions

1. Sometimes a requirement may be at the wobbly edge of functional (specific) verus nonfunctional (nonspecific). Do you have any feelings or advice about how to make such categorizations?
2. What do you think is the proper amount of time to take a response without posting something interesting for the user to look at?
3. Do you think it's easier to work compositing micro-requirements into a cohesive theme, or easier to take a broad vision and break it down?

7. The Types of Specifications

There are a thousand ways to do something wrong, and a hundred ways to do it right. That definitely holds for writing requirements specifications. Don't take the limited list presented here to mean that these are the only allowable document formats. Each company may have its own style of writing requirements documents. Below, we present descriptions of some common forms of requirements documentation.

7.1. User Stories (including epics and the like)

In general, a user needs to get things done. Very often, and especially in Agile projects, the first step in documenting a requirement is a "user story". Generally, a user story will have the form "As a «role», I «want or need» to «action» so that «purpose».". Here's an example: "As a pilot, I need to know my altitude so that I don't have unintended terrestrial contact." Often, user stories leave out the purpose clause. Certainly user stories can have additional clauses and even be larger than one sentence. But, they're usually kept short. The user stories relate to requirements documentation in that the user story is the basic indivisible quantity of requirement, often documented as a use case.

In Agile environments, user stories are often grouped into "epics". An epic is a group of user stories which are related to one another and slated to be in the same release or set of sequential releases. A "theme" is a set of related epics. Recently, some people have begun to use "initiative" as a level between "epic" and "theme". These groupings enter into the requirements documentation picture in that documentation is often done in an outline form, grouped by theme, initiative (possibly), epic, and user story. Some organizations have these names in a different order.

7.2. Use Cases

A use case document is generally a document consisting of some introductory material, the cases, and any follow-on data, such as appendices.

More specifically, here's what we tend to find in a use case formatted requirements document:

Title page: The title page goes here. For use in a class, any information required in your syllabus should be present, normally including the document title, your name, the assignment ID, the professor's name, the class name, the section number, and the submission date. For commercial purposes, the list usually includes the project name, the company name, the customer's name (internal or external), the document and/or product version number(s), and the authors' names. For government-related documents, the title page (and other pages) will often have a classification level, such as Unclassified, Confidential, Secret, Top Secret, or other special-access notations as required, with certain rules regarding the size, placement, and color of those tags.

A note on version numbering (which applies to this type of requirements documents, other types of requirements documents, and products): Generally, the first version of something will be

numbered 1 or 1.0, and subsequent start-from-scratch rewrites of documents will be numbered 2 or 2.0, 3 or 3.0, and so on. For products, products which have some incompatibility with previous versions will usually get "bumped" a whole version number. Document versions may or may not match product versions. When they don't match, there is usually text on the title page stating something on the order of "Document version __.__.__ for _____ version __.__.__". When there is no rewrite involved, but a significant change or addition is being made, the second position of the version number will generally be increased. For instance, going from 1 or 1.0 to 1.1, from 1.1 to 1.2, and so on. When very minor changes are made and distributed, the last part of the version number will increase, for instance from 1 or 1.0 to 1.0.1 or 1.0A, depending on the style in use in that company, or from 1.0.1 or 1.0A to 1.0.2 or 1.0B. Note that if you're bumping the third position past 1.0.9, the next version number will be 1.0.10, and not 1.1. If you're bumping the second position past 1.9, the next version number will be 1.10, and not 2.0.

Revision history table: This is a table showing the distributed or published versions of the documentation. The first time the document is distributed, there should be one line in the table of versions. Each time the document is edited and then redistributed, a new line should be added to the table defining what has changed and why. When possible, this table should be on a single page, not split. If the table is more than one page, then the table should be the only thing on the pages it appears on.

Table of contents: Be sure that the table of contents accurately reflects the contents of your document. You don't need to have everything in the document listed in the table on contents, in that if you have some section numbered 1.2.3.4.5, the table of contents might show the full list of section numbers, but might stop (for instance) at 3-part section numbers, showing 1.2.3, but not 1.2.3.1 or 1.2.3.1.1.

Note that when editing a requirements document other than the first version destined for distribution, you should edit in a manner that will show change bars. When editing subsequent versions, the change bars should show changes since the last distribution.

Editor hint: Various editors will let you mark section headers for inclusion into a table of contents and automatically produce the table for you. If you're constructing a table of contents by hand, be sure to have a "decimal tab" or "right tab" separate the section description and the page numbers, so that the page numbers are right-aligned. The style of that tab should be set to dotted underline.

§1: Introduction: Without getting formulaic, the introduction should state what the project is about. It's also permissible to have the introduction area empty, and just use sections 1.1 and 1.2 as the introduction.

§1.1: Purpose: This section should state the purpose of the application to be written. If the application is to be modified, then briefly recap the purpose of the application, and then get into the purpose of the addition or modification.

§1.2: Scope: This area is generally a recap of the project's charter — what the project can do to further the purpose, often involving budgetary and other constraints. This is the area that should,

hopefully, prevent over-scoping, in that if it's not in the definition of the project scope, then it shouldn't be done.

§1.3: Definitions, acronyms, and abbreviations: This section should include definitions of terms, and expansions of acronyms and abbreviations. If there is any chance that anyone reading the document will be unfamiliar with any of the terms, then those terms should appear in this section. Alternatively, each first use of a definition, acronym, or abbreviation can be defined inline the first time it's used. The definition can appear after the first use (for instance, "This document is for the Example application (app).") or the definition can appear before it (for instance, "This document is for the Example app (application)."). If this section is empty (either because there are no unique terms or because they're all defined inline), you can leave this section out. But, pick a style and stick with it. Either define everything in this section, or define everything inline.

§1.4: References: If you mention a document in the requirements specification document, then that outside document should be referenced in the reference section. References in this section should be in the appropriate form as defined by your organization. For most places, especially schools, that's the latest version of the APA (American Psychological Association) format guidelines, which is at the time of this writing, version 7. There should always be a reference or references back to the initiating document(s). For school assignments, this will typically be the assignment link. For other types of assignments, this will typically be an eMail, a purchase order (PO), some previous specification, or orders. If there is more than one document that the requirements specification is based on, then those should be listed, either individually, or as some higher-level collection of documents. So, this section should never be empty.

§1.5: Overview: Often present as a set-up for the remainder of the document. If you refer to the document, use "this" and not "the".

§2: Overall description: This area would be an introduction to the product as a whole as it will be, or to the changes and/or additions that will occur. For a new product, this can be in the future or present tense (future preferred). For changes and/or additions to an existing product, any description of things as they currently are should be in the present tense, and any description of things as they will be should be in the future tense. This area doesn't have to be long, but it doesn't have to be short either. Don't include actual requirements here.

§2.1: Use case model survey: This is where the use cases are introduced. In one style (the most commonly used one), you'll give a brief overall introduction to the list, and then the list, by name of the use case and brief description. That description can be a partial sentence (a phrase of a sentence with the subject and period missing), or one or more sentences. Don't intermix, though. Don't get long-winded here. You're just introducing the list. In another style, the use case name will be listed with the one-sentence user story.

§2.2: Assumptions and dependencies: In this section, list any assumptions and/or dependencies. Each assumption or dependency should be listed as a separate numbered, lettered, or bulleted item. If you have both assumptions and dependencies, don't intermix them, list all assumptions

(or dependencies) before going on to the other category. There should be no requirements in this section either.

§3: Specific requirements: Here is where the requirements proper start. Sometimes this section has text in it, and sometimes not. This section should start on a new page if it does not have text in it. (You'll see why shortly.)

§3.1: Use case reports: This is where the use cases start. Again, sometimes this section has text in it before the first use case report, and sometimes not. If §3 and this section have no text of their own, then §3 should start on a new page. If §3 has text and this section has no text before the first use case, then this section should start on a new page. Within this section there will be one or more use case reports.

Each use case report: Each use case report should have a title that matches the title of the use case in the listing above. The use case reports should be in the same order as the use case list. Each use case report should be numbered (1, 2, 3), lettered (A, B, C), or numbered as subsections (3.1.1, 3.1.2, 3.1.3). Within each use case report, there should be these other items:

Summary: This section should summarize the case in plan English.

Preconditions: This section should be a list of the preconditions, if any, for this step to begin.

Triggers: This section should be a list of the triggers (normally at least one) that start this flow.

The basic flow of events for user interaction scenarios and for sensor-interaction scenarios: This should be a table, with a header and one or more following rows. The heading items should be "Actor" (or some similar word), "System" (or something similar), and "Screen" (or whatever is most appropriate). There should be one step per row, or one pair of actor and system steps per row. The steps should be numbered, starting with 1 for the actor, and normally 2 for the system. On the first row, there should be a picture of the screen, a drawing of the screen, or a description of the screen. In subsequent rows, a new screen picture or description should appear when there are significant changes to the screen. That's assuming that this functional area has a screen. Leave out the third column if not. Don't use the term "actor" within the "Actor" column. If you have just one class of user, use "user". But, if you have more than one class of actor, use the class name, such as "customer" or "administrator". Sometimes, the system will be taking its input from sensors instead of people. For instance, an elevator door uses a light beam and a photocell. When someone breaks the beam, the doors retract if they're closing, and the "close" timer resets. In this sort of case, "sensor" or the like is what goes in the Actor column. If you intermix use actions with sensor actions, there should be a "Screen" column. If you have only sensor interactions (autonomous functionality) with no displayed output, then there should be no Screen column.

The basic flow of events for conditional scenarios: If actions take place because of conditions alone, the "Actor" column should be replaced by a "Condition" column. Back to the elevator example, a condition might be that the doors are open, and it's been 5 seconds since the "Door Open" (or "«»") button was pushed or the beam was broken. The system response would be to close the doors. Again, if there's no display associated with this autonomous action, there should be no screen column.

Alternate flows of events: When there is an alternate flow of events, this should appear, much in the same format as above, but starting at the first divergent step number.

Post-conditions: If there are any changes as a result of the primary or alternate flows to the permanent state of the system (or the outside world) or to the state of the session, then those changes should be listed as post-conditions.

Each use case report should be on its own page, with two exceptions: (1) For the very first use case report, if the §3.1 header had no text of its own, don't start a new page, because that would cause you to have a page with one section header line on it. (2) If two use case reports will fit entirely on one page with each other, do so.

§3.2: Supplementary requirements: If there are any system-wide requirements or so-called nonfunctional requirements, those should be listed here.

§4: Supporting information: Any additional information (not requirements) should go in this section.

Diagrams: Sometimes, one or more diagrams is required, most often some sort of UML (Uniform Markup Language) diagram. Those diagrams can go into the Supporting Information section, or into some other section, or into an added section. The general rule is that the diagram(s) should appear where they would first be useful. (We have a separate section on UML diagrams in a much later section.)

Sheldon's advice: Use UML diagrams and other diagrams only when they add in understanding, or when the boss or customer requires them.

If you have nothing for a given section that has no subjections, either use "Not applicable", use "N/A", or leave the section out, renumbering the other sections as necessary. If you have nothing for a given section that has subjections, such as having nothing to say between the §3 header and the §3.1 header, that's ok. Don't leave §3 out or claim that §3 is not applicable. If a given section has only one subsection, then combine that subsection with the enclosing section.

A complete template appears as Appendix A, and a complete example appears as Appendix B. In Appendix A, blue areas are notes, not to be included in the final document, and fields marked with angle brackets ("<" and ">") are items that should be replaced with your text. The angle brackets should not appear in the final document.

7.3. APIs, Connectors, and other Black-box Techniques

Sometimes you need a system that, given certain inputs, produces certain outputs. If everything the system does can be described in terms of its inputs and outputs, then the system can be described by an interface, or API, description or documentation alone. In those cases, an API reference is written, and the software can use that API reference as a specification. When the functionality of a hardware or hardware-and-software system can be described in this manner, the unit is often called a "black box". (*Notes: The term "black box" is also used in testing and in aircraft. In testing, it means that the testers don't get to see the inside of the equipment or the code. In aircraft, it means the flight recorder, even though it's dayglow orange. "White box" testing means that QA does get to see the insides and code.*)

7.4. IEEE 830 and the Like

In circumstances where the organization or the customer wants complete documentation on everything, especially in aerospace, military, and medical projects, IEEE 830 format or some MilSpec (Military Specification) format is used. Generally, you'll have these sections in an IEEE 830 document:

Title page, table of contents, and revision history (or table of versions) are the same as for the Use Case document.

§1 Introduction should be a title only.

§1.1 Purpose should be the same as in the Use Case format.

§1.2 Document convention should tell us anything special about the typography, such as bold meaning a certain thing, color meaning a certain thing, special symbology, or the like. It's ok to have this section as "N/A".

§1.3 Intended audience and reading suggestions should tell the reader who the intended audience for the document is, and give reading suggestions. For instance, if project management and developers need to read the whole document, but the QA staff only needs to read certain sections, mention that.

§1.4 Product scope and §1.5 References are the same as in the Use Case format.

§2 Overall description has no text of its own, but is only used to hold subsections.

§2.1 Product perspective typically introduces why this product or version is being produced, including who the intended users are.

§2.2 Product features gives a brief introduction to the product's features, including a list of the features' names.

§2.3 Use classes and characteristics gives us a list of the user classes, what's different about one user class from another, and what features apply to each class. On some projects, there is only one class of users, and this section may only be one sentence long. On more complex systems, there may be dozens of user classes, with a table showing which of the thousand features can be used by the dozens of user classes.

§2.4 Operating environment tells us what the intended operating environment is. For hardware, this might entail temperatures or connection to a type of electrical buss. For programs, it might tell us about a cloud system or an operating system or the like. Don't use this section to make decisions for the project manager. This section should only be used to describe existent facts or planned operations. For instance, tell us if there is an existing Unix system that the new components need to be added to, but don't tell us that the system should be written for Unix if there is any way that the system could work outside of a Unix environment. If there is some reason that you need to limit the project manager's choice, in this section or any other section, give a reason. Each specific environmental concern should be in a separate paragraph and have an ID, "OE-" and a number, such as "OE-1".

§2.5 Design and implementation constraints is another section that follows almost the same rules. The design and implementation constraints should optimally be "N/A", leaving all the decisions to the project manager. But if there are constraints required, list them here, with the first design constraint labeled "DC-1" and the first implementation constraint labeled "IC-1".

§2.6 User documentation should list the existing and required documentation. Each entry should state implicitly or explicitly whether the documentation already exists. How to do explicit statements is obvious. Implicit statements should use "is" or "shall be" to indicate existing and future documentation, respectively. If something already exists and must be modified, that needs to be stated explicitly.

§2.7 Assumptions and dependencies is as per the Use Case documentation, except that the assumptions should be labeled with "AS-1" and on, and the dependencies should be labeled with "DE-1" and on.

Before we go on, it must be stated that the idea in this type of documentation is to define things, and then to use that definition to build requirements on. §3 expands on the list from §2.2, and serves as the basis for §4 and §5. Within §5, each stimulus-and-response pair will serve as the basis for the actual requirements.

§3 System features serves as the introduction for the list of product features. There is usually an introductory paragraph. This section contains subsections numbered 3.1 and on, each with a title that matches a feature name, and a brief description, most often one sentence long, but sometimes as many as three.

§4 External interface requirements is solely a section header, with no text of its own.

§4.1 User interface overview is just that. It describes the user interface's guiding principles. It may contain pictures, but might not. The entire user interface can be described (and thus required) here, or that can wait until §5.

§4.2 Hardware interfaces describes how the software interfaces with hardware, or how hardware interfaces with other hardware. This section may be broken down into subsections, so that each functional area's interfaces can be described one at a time.

§4.3 Software interfaces describes how software interacts with other software. If the interaction is internal and can be handled as a "black box" system, don't describe those interfaces any more tightly than you have to. If the software is to interface with the outside world, or with another system, or with a component to be built by another company or entirely separate group, then that interface should be described in detail here.

§5 System Features/Modules: This section may have an introductory paragraph, but mainly it holds one requirements block per functional area. These functional areas will be numbered starting with 5.1, and the list of functional areas should match the earlier list. Within each such area, you'll have these subsections:

> §5.__.1 Description and priority: The description is a plain English description of what the module does, and why, in as much detail as you can provide. The priority can be in the form of High, Medium, and Low. Some even add Emergency and Possible. Alternatively, the priorities can be ordinal, such as first, second, and so on, or ordinary, such as 1, 2, and so on. Priorities can also be in the form of Due By dates.
>
> §5.__.2 Stimulus/Response Sequences: This section should have the stimuli that make things happen (usually, in order, a user input or a sensor report), and the system's response to each stimulus, labeled "Stimulus" and "Response". In the case of a clock going off, rather than using Stimulus and Response, use Condition and Response. If all the items are conditions and responses, relabel the area to "Condition/Response Sequences".
>
> §5.__.3 Functional Requirements: This section first echoes each stimulus/response (S/R) pair (or C/R) as a requirement, in "Upon «stimulus or condition», the «component» «shall or must» «response»" form. Following the echoed requirements, and additional requirements for the area not specifically based on a stimulus (if any) should be listed. Each requirement should have an ID of REQ-__.__, such as REQ-1.1, where the first number matches the center number of the 5.__.3 section, and the second number starting with 1 for each group.

§6 Nonfunctional Requirements: This section lists nonfunctional requirements, as defined above. Often, these nonfunctional requirements are grouped into subsections, such as Performance, Security, and the like. There should almost always be performance requirements. Typically for user interface items, anything the user does should have an immediate visible effect, such as a button going into a "depressed" or "disabled" state, or changing name, and/or the cursor changing to a "watch" cursor. When possible, operations taking longer than 2 seconds

should show a progress bar. For aircraft, the standard is that when the command is accepted but the situation doesn't match the command yet, the indicator should be blue. (For instance, we have a fuel valve which we can control, so the value has a white "controllable" indicator, and we do some action that's supposed to close the value. The control should redraw [assuming it's a drawn control] to the closed position, and the indicator should turn from white to blue to show that the action has not occurred. As soon as the action has occurred, the indicator should return to white.) If some action will take some time to perform, but is not something the user needs to wait for (such as starting a print job), there should be some briefly visible indication that the process has been started. All nonfunctional requirements should be measurable. For instance, it's not ok to say that the system will support a lot of users with no performance degradation. It is ok to say that the system must be able to support 1,000 users with the average response time under 500ms. The first is no ok, because we can't measure what "a lot" is, nor can we ever measure "no performance degradation", because we don't know how many "a lot" is, and because as soon as you get the second user, there is some performance degradation. The second requirement is much better, because we know how many users 1,000 is, and we know how much time 500ms is, and we can run a test with 1,000 loads and measure the average response time.

A template and sample appear at the end of this document as Appendices C and D, respectively. Appendix E gives an example of how a lightswitch might be specified for §5 in this style of documentation.

7.5. Manuals as Specifications

In many circumstances, a manual for a new product or software library or device can serve as a specification for that project. Manuals are much cheaper to produce than software, and to the average stakeholder, easier to understand than the typical requirements document. Also, stakeholders tend to be happy to receive specifications in the form of a manual, because it makes them feel that a lot of progress is being made. However, when doing a manual as specifications, make sure that they understand that the product is not complete at that point.

7.6. Pure R&D

When doing pure research, or research and development, rather than just development, almost by definition, you don't know what you're going to get. In the pure research case, one would expect that the specifications hold only guidance on what type of research is to be done. In pure research and development, the end goal is known, but little will be said about how those goals are to be carried out. For instance, when President Kennedy said that the US would go to the moon within the decade, he didn't say it was going to be a 3-man crew with a separate lander. (The original plan was for two Saturn Vs to put up and assemble a spacecraft that would land on the moon and return.) What's often *called* R&D is really pure development, in that the final result and design is either fully or mostly known.

7.7. How Things Can Go Wrong

This one is pretty easy. Any complete specification that the customer (or internal team) is happy with is a hit. The only problem that can arise here is that if the customer demands one type of

specification, and you deliver another type, you may have to do it over. Often, government and safety-related projects will demand a certain format.

7.8. Discussion Questions

1. It's not the case that you must do your work in exactly one of these formats. When doing your work, do you have an affinity for one of these formats? If so, which and why?
2. If not, do you see things from any of these formats that you think is pretty good?
3. Or things you think should be avoided?

8. The Engineering Triangle

There is a concept in engineering that describes the relationship between cost, speed, and completeness of a project, called "The Engineering Triangle". It applies to requirements engineering as well as just about everything else.

8.1. The Concept

The Engineering Triangle, simply put, states "Good — Fast — Cheap — Pick two". What it really means is that if you want something good and fast, it's not going to be cheap. If you want something that's good and cheap, likely it won't be fast. And if you want something fast and cheap, likely it's not going to be good. In software engineering, "good" tends to mean feature-rich, doing the entire job it's designed for, and as bug-free as can be. "Fast" tends to mean that the product is produced quickly, or that it runs quickly. "Cheap", in software development terms, tends to mean that the cost of producing the software is low. However, in engineering involving hardware, "cheap" often means that the hardware itself is cheap to build.

8.2. Graphically

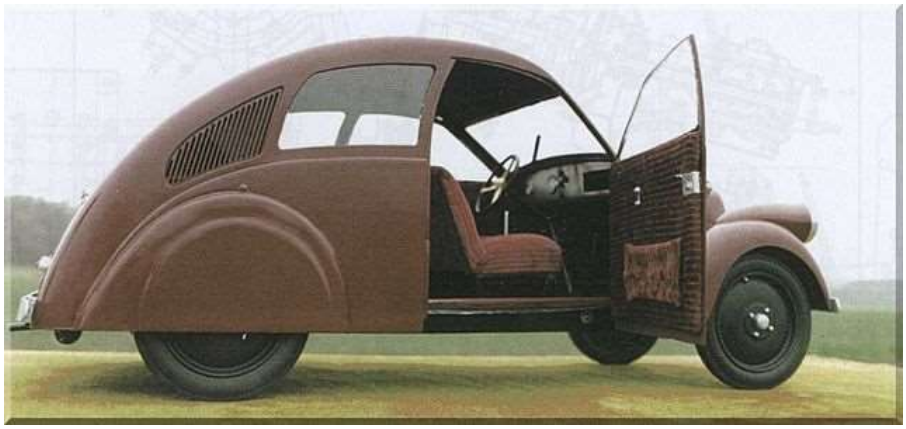
Graphically, this concept can appear as a triangle. Within the triangle, we have our project's aim, goal, or setting, shown here as a small circle. The triangle has a Good edge (and a Fast Edge and a Cheap edge). We can move our goal all the way to the Good edge. Once at the Good edge, we can move that setting all the way to the Good & Fast vertex (where we're also on the Fast edge) or all the way to the Good & Cheap vertex (where we're also on the Cheap edge). As drawn, this image shows a project that's pretty good, and pretty fast, and not even close to cheap.

Note that the Good & Fast vertex is expensive, the Fast & Cheap vertex is bad, and the Good & Cheap vertex is slow.

For most projects, you'll be choosing something right on the Good edge, and at or close to Good & Fast, or at or close to Good & Cheap. Although when trying things out for ideas, Fast & Cheap is a good way to go.

8.3. The Vehicles Example, with Q&A

It's always a matter of making the best choices for the project. Let's look at a few examples.

 Axios23, under the Creative Commons license	**Bugatti Veyron** Good Fast Not cheap
 Wikipedia DE user Mb1302 under the GNU license	**VW Bug** Good Cheap Not fast

 Alf van Beem, under the Creative Commons license	**Fiat X-1/9** Fast Cheap Not good

8.4. How Things Can Go Wrong

If you don't know what the victory conditions are, you're extremely unlikely to win. Know what the "win" is, and then shoot for that.

8.5. Discussion Questions

1. Let's say we offered to give you a Veyron, a Bug, or a Ryder truck, one for free. Which would you take?
2. How many pianos can you put in that Veyron?
3. How much does it cost to drive that Veyron or truck?

The moral is: Know where you stand on the Engineering Triangle before you start the project.

9. Contract Styles

Whether the money for a project comes from internal sources or a customer or venture capital, it has to come from somewhere, and on a given schedule. The quantity of money and its scheduling can have an impact on the project's requirements. For instance, one main concern: Given the amount of money available, what can we do? That at least calls for prioritization, and may (and usually should) call for a limitation on the requirements.

9.1. Internal Funding

Internal funding is the most common type of funding on a project. It simply means that the company is paying for its own project. Although funding from venture capitalists is actually external funding, it is usually treated like internal funding, except that the outsiders can pull the rug out from under you at almost any point. Within internal funding, there are several ways that money gets scheduled, listed below.

9.1.1. Budgeting in Total Dollars, or "Man-months"

Sometimes, internal funding is in terms of dollars, or in man-months. A man-month is defined as the amount of work one person will do in a work-month, allowing for vacations, holidays, sick days, and about 25% distracted time. When budgeting is in dollars, it may include hardware costs, and even overhead costs, such as the building and utilities. When budgeting is in man-months, it's just head-count over time. Here is one place that project management type and money interact. If there is a limited budget, then the project management team, usually including one or more business analysts/spec writers, needs to decide on the management style based on requirement stability and money available. If the requirements are very stable and money is tight, then a Waterfall model (executed carefully) will often lead to the lowest expenditure, because only one team is on the project at a time. However, if the requirements are fluid, then the Waterfall model will be disastrous. Likely the requirements are somewhere in between, and a call of best likelihood will have to be made.

Note that the number of man-months required for any given amount of work will vary with the number of people working on the project. One person working for 100 months can do a certain amount of work. But 100 people working for 1 month will need a considerable amount of interfacing and coordination time, and management. As Frederick Brooks points out in *The Mythical Man-Month*, "Nine women can't make a baby in a month.".

9.1.2. Budgeting in Dollars per Month, Staffing Levels, or "Burn Rate"

Many, if not most, internally funded projects, seem to have a given staffing level, or dollars per month. The latter is often known as the "burn rate". This type of funding may or may not have a dollar limit or a time limit. If there is a dollar limit or a time limit, then the requirements must be prioritized and limited. If there is no overall funding limit, and no overall time limit, then the requirements must be prioritized, but not limited. Even if there is no limit, you're still going to need to evaluate whether each requirement (a) needs to be done regardless of cost, (b) is worth

doing, meaning that the return on investment is > 1 (meaning that the product is worth more than is costs to produce, or benefit÷cost), or (c) not worth doing, meaning that the ROI ≤ 1.

9.2. External Funding

When you're producing a product for a specific outside customer, there are two main types of contracts, as described below.

9.2.1. Time and Expenses

In a time-and-expenses contract, the customer pays the producer for the manpower time used, normally at an hourly rate per person. This may be one set rate per worker, or a set rate per working in each individual class of workers, or on a rate set on a person-by-person basis. That's the time component. The expenses component is that the customer pays for expenses, such as travel, equipment, and whatever else needs to be provided.

The relationship between this type of funding and the project management style and the requirements volatility is this: If the requirements are volatile (rapidly changing), then you need to have a time-and-expenses contract, or the customer will do whatever they want to the requirements, and you're left holding the bag, meaning that you'll likely take a loss. If the requirements are stable, then you have the option of using time-and-expenses or fixed price. But, there is a limit to the volatility of the requirements in relation to customer happiness. If you allow the requirements to vary too quickly, then the bill on the customer will be run up, and the customer will not be happy. You'll have a huge profit on this one contract, and no future business from that customer nor potential customers that the unhappy customer talks to. If you hold requirements to what is truly called for, this one contract will be a bit smaller, and there will likely be follow-ons and word-of-mouth referrals.

9.2.2. Fixed Price Contracts

A fixed-price contract is one in which the parties agree that the value of the contract is a certain fixed amount of money, no matter how much the job really costs. In this type of contract, the requirements should be finished before the contract starts. So, if the customer is paying for requirements engineering, then that requirements engineering will be (a) done by the customer, (b) paid for by the customer in a time-and-expenses manner, or (c) just done for free by the producer for the customer, in hopes of recouping the expense in the fixed-price contract.

If you enter into a fixed-price contract, then the requirements are locked in, and you might as well use a Waterfall model, since in this circumstance, it's likely to be the cheapest to execute. Of course, you can always use an Agile model, even though the requirements should be known at the onset.

Even in a fixed-price Waterfall project, prioritization of requirements is important. First, if a given requirement is slated for an earlier release than some other requirement, then the earlier-scheduled item must have a higher priority. But often, the contract doesn't say what's in each release, only what's in the final delivery. But just as often (at least), is that various components

of the system each carry a part of the price. For instance, Component A has a delivery value of $5,000 and Component B has a delivery value of $10,000. If Component A will cost 100 work units (whatever those are in your organization), and Component B will cost 160 work units, then the delivery price per work unit of Component A is $50 per work unit, and the delivery price per work unit of Component B is $62.50, and so, the requirements for Component B should be prioritized higher than those of Component A (assuming that Component A does not rely on Component B), because that brings in more money faster.

9.3. Hybrid Funding

Of course, not all projects fit into the tidy boxes presented above. Sometimes a project has an external purpose, but the project is funded internally in hopes of selling the project later. Sometimes there are cooperative ventures between two companies on a big project. In those cooperative ventures, some parts of the project will be paid for by each company as internal projects, but other parts of the project will be on a time-and-expense or fixed price basis between the two companies.

9.4. Offer, Acceptance, Performance, and Payment

Generally, the four steps in a contract are Offer, Acceptance, Performance, and Payment. One party offers the other party a deal, and that second party accepts. One or both parties perform some obligation and/or work under that contract, and then (or during), there's payment. If a controversy arises, any three of those things is enough to prove or force the other:

- The trivial case: If there was acceptance, performance, and payment, then there must have been an offer.
- If there was an offer, and there was performance and payment, then the offer will be deemed to have been accepted.
- If there was offer and acceptance, and payment, then performance or a refund can be compelled.
- If there was offer, acceptance, and performance, then payment can be compelled.

In many jurisdictions, the contract still exists, even if it's not on paper, although there may be a monetary limit to that.

9.5. How Things Can Go Wrong

Contracts are often the driving factor, so we can't say that there can be something wrong with the contract style, but having a disconnect between the contract style and the development methodology can be a big problem. Make sure that these two factors are compatible.

9.6. Discussion Questions

1. In a contract, who is responsible for any ambiguities?
2. If a problem arises, can renegotiation be forced?

10. Formal Processes

Most of what you do as the business analyst and requirements documenter will be unstructured time and informal processes, but there will be some scheduled time and formal processes, too. Below, many of those are listed.

10.1. Non-Agile

To a great extent, "non-Agile" means Waterfall, but it can also mean formal processes before or outside of Agile processes. Some of these processes are outlined below.

10.1.1. Request for Proposal — RFP

RFP stands for Request For Proposal. It generally means that some large company or a governmental agency wants a project done. They will issue a Request for Proposals, outlining what the project has to be or do, and it's up to the bidding team to produce a proposal to answer the request.

10.1.2. Proposal

The proposal in response to a RFP is not just a quote. Often the RFP is fairly vague as to exactly what has to be done, instead just laying out the need that needs to be handled. For instance, recently NASA announced that they want to return to the moon in a new spacecraft, and issued an RFP to that effect. They didn't give more detailed requirements than that, and it was up to the people writing the proposal to determine the exact requirements and how it would be done. When you get an RFP, you need to respond with a proposal (if you're going to respond at all), and this will usually take some amount of requirements engineering to determine what will be in the proposal, and then what that proposal will be priced at.

10.1.3. Bidders' Conference

Some time after the RFP and before the proposal is due, there will often be a bidders' conference. In the bidders' conference, you and the other prospective bidders are allowed to ask questions, in front of all the other bidders. So, there's a strategy involved. You want to get your questions answered, but at the same time, you don't want to tip your hand to the other potential bidders.

10.1.4. Request for Quote — RFQ

RFQ means Request For Quote. The company or agency will issue a RFQ when the requirements are already well-known (usually because they issued the RFQ with a complete set of requirements), and they just need a numeric bid. If the requirements are not well known, or are ambiguous, be careful.

10.1.5. Quote

Of course, the response to a RFQ is the quote. Often, that's just a numeric bid, but often other information is required.

10.1.6. Change Request

In a time-and-expenses contract, you'll get informal change requests all the time. You just handle those without a second thought.

But, in a fixed-price contract, you'll often be constrained to a formal process, because the contract will specify that. Normally, the process starts with the customer issuing a formal change request. That change request may be in the form of changed requirements, but it may be in some more vague terms, and will need to be translated into proposed requirements changes. You would reply, showing them those proposed requirements changes, and with a proposed price change, if any. If there is more work to do, or work that would need to be redone, there will generally be an increased price. But, if the work to be changed is only future work, and there will not be added costs, then you may or may not want to charge them a fee. If the change request will somehow actually save you money, then you would just send them a note that their change request has been accepted as a change order (see below).

10.1.7. Change Order

Once the customer has accepted your bid on their request, they'll issue a change order. That means to go ahead and make the change.

A change order will cascade into requirements tracking, in that some requirements will become moot, some requirements will change and some requirements will be added. In very formally controlled projects, there will be a requirements traceability matrix. In this matrix, one coordinate is the list of all the requirements, usually by ID and a short description, and the other coordinate is a list of controlling documents. A requirement present and unchanged since the onset of the project would only trace back to the initial document, but a requirement that was modified by two change orders would show that three documents control it (the original and the two change orders).

This brings up a slightly broader subject: Who logs, tracks, and reports on the progress through the requirements. Sometimes it's the project manager (PM), sometimes it's the lead business analyst (BA), and on some Scrum projects, it's the Scrum Master. On smaller projects, those two or three positions will be the same person.

10.1.8. Engineering Change Proposal — ECP

Sometimes the producing end of a fixed price contract wants to make changes to the plan. When this happens, because you found a better or cheaper way, or some component becomes unavailable and needs to be replaced, you'll develop a set of changed requirements, and issue an Engineering Change Proposal (ECP) to the customer, which they'll either accept or reject. If

they reject it, you just keep making whatever you were making. If they accept it, then the new requirements lock in.

10.1.9. Progress Reports

Often, progress reports or intermediate deliverables are required. Even if they're not required, it's generally a good idea to send them to the customer. In your progress reports, state what happened in the last period, and what is planned for the next period, and any problems you're up against. There are some pluses and minuses to this: You don't want to give them too much information, but at the same time, you don't want to give them too little. Customers don't like surprises, so use the progress reports to prevent surprises. Unfortunately, it's impossible to give more specific advice here.

10.2. Agile

Where non-Agile projects tend to have documentation and notices, Agile projects tend to have events, often called "ceremonies". Many of those are detailed below.

10.2.1. Sprint Planning

As mentioned above, a sprint is the unit of time on an Agile project between versions, or at least the period of time on which assignments are handed out. This is typically a week to a month.

In each sprint, the team will plan on accomplishing a certain amount of work. What work will be done during the sprint is generally worked out with the project manager, the scrum master, and a business analyst. Sometimes this sprint planning is done with the whole team, and sometimes not. Generally, during the sprint planning session, points, sizes, or estimated hours are assigned to each task to be done during the sprint. Tasks can be sized any time after the requirements have been documented, so a task may be sized but not included in the sprint. Assignments of tasks to people is then done, either in a Scrum session, or through a Kanban board.

10.2.2. Daily Stand-up

All Scrum-based projects, and many other projects, have a Daily Stand-up meeting. It's called "Stand-up", because the intent of the meeting is to be a quick recap of yesterday's events and the plans for today, with the idea that if everyone stands for the whole meeting, that the meeting will tend to take less time than it would if people were seated.

10.2.3. Retrospective

When the sprint finishes, there is a Retrospective meeting. In the Retrospective, there is generally a discussion of what went well, what didn't, and how things could be improved for the next sprint, as a means of process improvement.

10.3. Other Forms of Tracking and Paperwork

There are sometimes additional types of work done in tracking a project. We won't claim that each of these is always useful to a project, nor will we claim that each of these is always a waste of time. Each of the following have their purposes, advantages, and costs. Although there are a lot of management-related charts, graphs, and lists done, this section will only deal with those pertaining directly to requirements engineering.

10.3.1. The RACI Matrix

RACI stands for Responsible, Accountable, Consulted, and Informed. The idea here is that you (or someone in management) will draw up a spreadsheet. One dimension will list these four classes of stakeholders, and the other dimension will list each of the project phases and components. So in concept, the matrix is really three dimensional, but two of the dimensions, phase and component, are generally listed from top to bottom. Within each cell is a list of the people or positions who fulfill the attribute for the phase and component. You're "doing it right" if you have one or more people listed in each cell.

The advantage to doing this is that you know that you've covered your bases of who needs to be talked to. The disadvantage is that if this is all obvious, there's no point in doing this.

10.3.2. PERT Charts

PERT stands for Program Evaluation Review Technique, but almost nobody knows that or cares. What they know is that a PERT Chart will show, visually and clearly, what relies on what. It's an excellent way to do schedule planning no matter what the management style, and there are a lot of programs that help you to product this chart. The idea is simple. You have a number of rectangles (or rounded rectangles) that represent tasks. Very often, a task is the implementation of a requirement, function, or user story. But, they can also represent tasks, such as flushing out the requirements for a user story, theme, or epic. The box will be labeled as to it's title or requirement ID, in the center. Once people and/or equipment (collectively, "resources") are assigned to the task, that assignment appears under the title. The left side of the rectangle is called the "input", but not in the programming sense. Anything that has to happen before the task can be started is the tasks "inputs", "preconditions", or "blockers". So, the leftmost rectangle will be the project kickoff, and it has no inputs. The right side of the rectangle is called the "output". The outputs, or "reliances" are connected to any task that requires this task to be done before they can start. So, the rightmost box will be "Done", and has no outputs. The boxes are connected to each other in such a way as to indicate what relies on what. By assigning time to complete, man-hours, level of effort, or "points" (the kind you can add up) to each box, you can get an estimate of the work remaining by adding up the remaining boxes. The path from left to right that has the highest score (hours, points, dollars, or whatever) is the "critical path", in that any delay on that path will delay the entire project's completion. Often color is used to denote Blocked, Started and on pace, Started and running late, and Completed, often with some sort of bolding to denote the critical path.

In any nontrivial project, tracking what has to be done next, and knowing the critical path is very important. The upside of this far outweighs the downside virtually every time, unless no component or task relies on another.

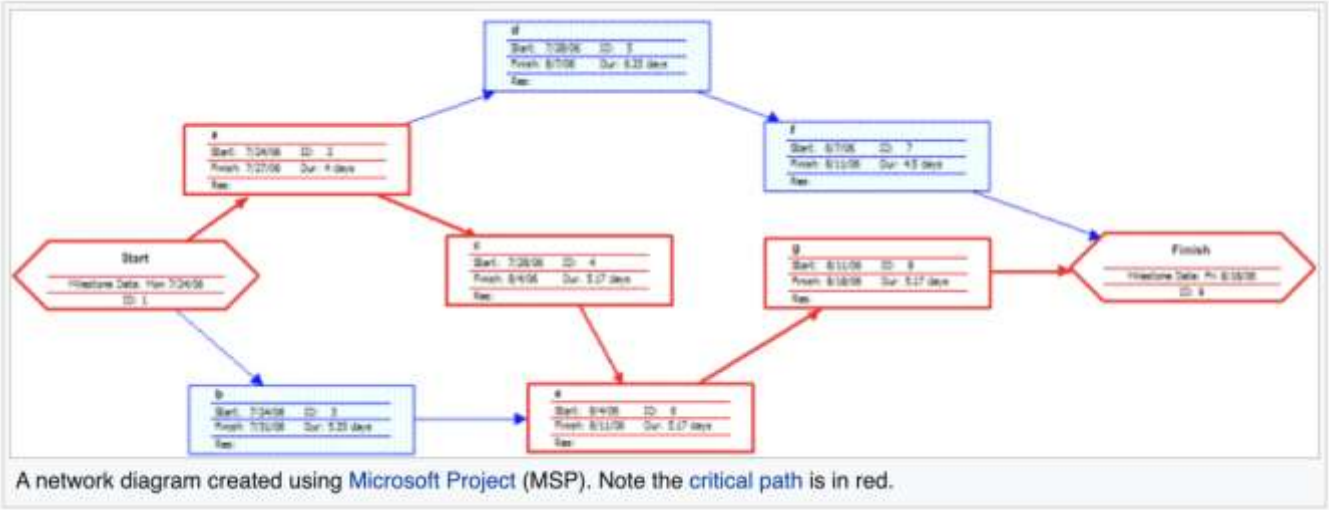


Image by "Rm w a vu"

10.3.3. The Keeping of Statistics

A lot of projects keep a lot of statistics and benchmarks. These may include how many requirements each person gathers over any given time, how much of their work is reworked, how many ambiguities are found in their work, how complex areas are, the average size of each story or function, and the like. There are also an even larger number of statistics that can be kept on the programmers and QA people. Having gathered these statistics, you can find various indices, such as the requirements quality index, the productivity index, and others.

On the plus side, you can find out about how the organization is really working, find out everyone's strengths and weaknesses, make better predictions about the next project, and improve the process. On the plus-and-minus side, you can get rid of whomever is not producing. On the drawback side, you can find yourself spending an inordinate amount of time gathering and calculating these values, and you can purposefully or inadvertently make people feel so much pressure that they'll burn out and/or leave.

10.4. Decommissioning of Obsolete Requirements

In an Agile process, things change all the time. That will mean that some information is refined to a better depth of understanding, some things will flat-out change, including from a No to a Yes or a Yes to a No, some requirements will be added, and some requirements will be deleted. When a requirement changes in a Waterfall project, it's much easier to notice, because there's a formal process for change management. But either way, you'll be left with a different set of requirements than before.

- An added requirement is the easiest to deal with. You just add it to the backlog in an Agile project, or in a Waterfall project, add it to the task list (in the specification document, in the requirements traceability matrix, and in the PERT Chart). Eventually, someone will be assigned to it. Be sure to make any documentation changes using some sort of change tracking, so that change bars appear in the document at least, and to note the change in the version history log.

A modified requirement is a bit harder to deal with, as there are a few possibilities:

- If a requirement is modified before work has begun on it, then only documentation changes to the backlog or the requirements document need be made.

- If a requirement is modified once development has started, and before QA is started, then in addition to the document changes above, the developer must be notified immediately. Scheduling adjustments will also most likely have to be made.

- If a requirement is modified once QA has begun work on the affected item(s), then in addition to the document changes above, both the developer and QA must be notified immediately, so that QA can stop their work, and so that the developer can either begin the rework or at least schedule it.

- If a requirement is modified after QA has signed off, then all of the above holds, and one or more components may have to be recalled, and the release schedule may be affected.

The deletion of a requirement might involve more work than you might expect:

- When a requirement is deleted (often called an obsolete software requirement or OSR), the first step is documentation. Never just remove the items or requirements from the backlog or the documentation. Edit the documentation, usually using strikethrough text ("~~like this~~") or a shaded background ("like this") to indicate that the requirement is known but no longer valid. The reason for this is to prevent the deleted requirement from being "rediscovered", which is a strong risk. Typically someone will come up with the rejected idea too, and add it, or find that the deleted requirement is "missing" and add it back. If the requirement is still in the document, showing as deleted, then you won't get into the trouble of accidentally reinstated requirements.

- As above, if the item is in development or in QA, it must be stopped for rework.

If a deleted requirement has already cleared QA, then it must be categorized:

- If the deleted-but-completed requirement is still useful, but was only deleted because the cost was no longer deemed worthwhile, then the work-product of that requirement falls into the "you bought it; you own it" category. The feature is no longer required, but it's still a feature. Remove it from the requirements set, but leave it in the user documentation and in the product.

- If the deleted-but-completed requirement is not useful, but is of no direct harm, then schedule the code behind it for removal, but at the absolute last priority. The reason to schedule the code for removal is that in the product's life cycle, added code complexity means added costs on maintenance, if for no other reason that people have to consciously ignore it, But, if dead code ends up being maintained, that's an unneeded cost. It's cheaper to remove the code at the end of primary development than to maintain it later.

- If the deleted-but-completed requirement uses any significant amount of system resources, such as screen-space, power, memory, disk, or CPU time, then the story is the same, but the priority on removing it goes up commensurate with its load.

10.5. How Things Can Go Wrong

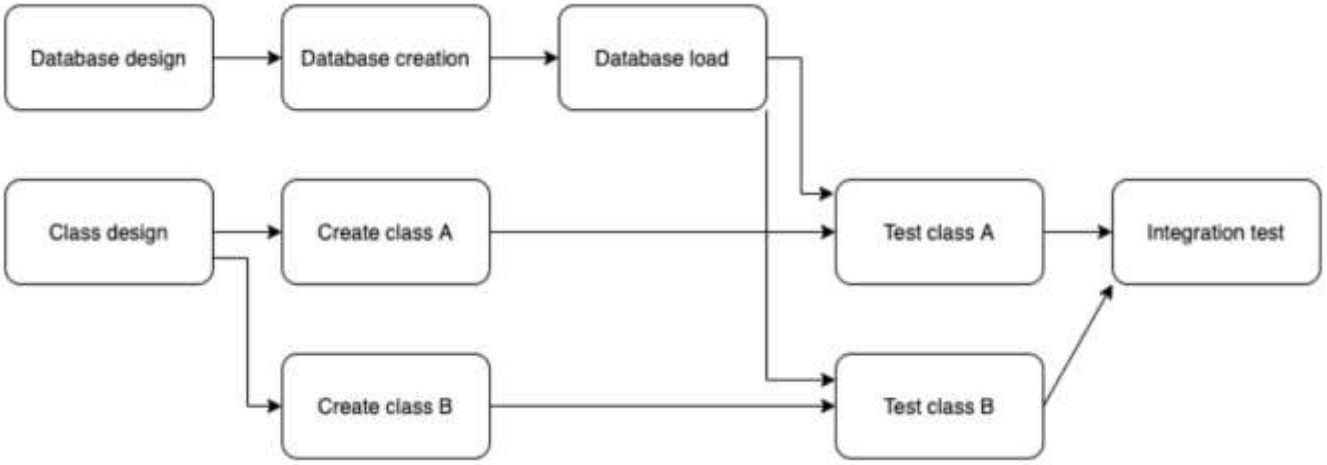
Skipping a formal process that's contractually required can become a compliance, contractual, and money issue. Skipping any regular periodic step from time to time can cause confusion in the staff. Skipping the PERT chart can lead to (a) resources (mostly people) going idle, and/or (b) not understanding the critical path. Both of which can cost time and money. Keeping too many statistics is a waste of time, and keeping too little can mean that you don't know something you need to know; so know what it is you need to know, and plan for that. Keeping obsolete requirements active in the system will usually mean too much maintenance cost.

10.6. Discussion Questions

1. Have you been on any projects where any of these techniques have helped?
2. Have you been on any projects where any of these techniques has been a hindrance?
3. Of these techniques, which seems most useful (if any)?

11. Maintaining the Product Backlog

Some projects have a scheduled set of tasks to accomplish, and a dependency chart. The dependency chart is most often used in Waterfall and Spiral projects where one component will be relied on for another component to function. Here's an example of such a chart:



This type of chart and scheduling shows which tasks have to be completed in order for the next task to be completed. There are also often annotations showing who is assigned to the task, estimated workload remaining, and already expended work, plus some coloration or shading to show at a glance which tasks have been completed.

In Agile projects, especially Scrum projects, things are generally built up a little at a time, so that (using the above example), just enough database design will occur to allow the database creation and load which Class A will need, so that those can be, to an extent, a single sprint's worth of work. Because the work on a "thing" is done incrementally, rather than all at once, tasks tend not to rely on each other much. (On the other hand, these "things" need to be revisited to be flushed out on many occasions.) And so, in Agile projects, all that really needs to be kept track of (for scheduling purposes) is the "backlog". The backlog consists of everything wanted, but not yet completed.

12. When to Watch your Words Carefully

In requirements engineering, there are a number of cases in which you must watch your words carefully. Some of them are listed below.

12.1. Contracts

Disclaimer: We're not lawyers. Nothing in this (or any other) section is legal advice. Rather, this is all academic teaching and business advice. For actual legal advice, consult your company's lawyer.

Very often, a fixed price contract will "include the requirements specifications by reference". What that means is that although it doesn't seem like you're writing a contract while you're writing the software (and possibly hardware) requirements specifications, that (or those) specification document(s) may be included as a part of the contract, simply by the contract stating that the document is included by reference. Thus, you must keep in mind contractual rules while writing the document(s).

One of the most pertinent rules in writing contracts is that the person (or party) who did not write the contract is entitled to any reasonable interpretation of each clause of the contract. There are two take-aways from this: (a) Don't have any ambiguities in anything you write, and (b) watch out for the phrase "this Agreement is the work-product of both parties" or anything like that in a contract, because when you see that, the person who wrote the contract is lying with the intent of having you pay for their mistakes.

12.2. Ambiguity

The biggest way to trouble in a contract or in a project is to have ambiguities in the contract or in the specifications. The best way to avoid ambiguities is to examine every sentence in the documents, and ask yourself "Is there any way that I can think of to misinterpret this, either through stupidity or malevolence?" Both of those can and will be used against you. If you have to explain something in depth, do so. If you have to draw a picture, do so. If you have to use a different word, do so.

If you're working with any speakers of a foreign language, and their language inflects entire sentences to the positive or negative, don't have any double negatives anywhere in your document, because double-negatives are not translatable into most languages.

Another negative effect of ambiguity is that you'll inadvertently pit the development team against the QA team, costing both teams a lot of wasted time. If you're ever in a situation where the developers think that a certain requirement means one thing, but the QA people think that the requirement calls for something else, then it's not the fault of the development team, and it's not the fault of the QA team. It's then the fault of the requirements team. The remedy is not to declare one team right and one team wrong — the fix is to fix the requirements. A corollary: If a developer (or anyone else) asks you what something means or how to interpret it, don't just

answer the question — answer the question and fix the requirements so that nobody will ever have to ask again.

Some managers, customers, or suppliers may use ambiguous phrasing to gain "wiggle room". Wiggle room is exactly the opposite of what we're trying to do in requirements engineering, so correct any such word usage.

12.3. The Deadly Word "All"

Use of the word "all" in a requirements document can be extremely costly. An oft-seen mistake is something like this: "The web-site shall support all browsers." That seems an innocuous enough phrase, but it can cost a fortune, in two ways. First, testing can go on for months, when QA brings in 600 browsers to test with and has at it. Second, your customer may find that 601st browser that the product doesn't work with, and hold you over the coals for some monetary concession, or worse, they may be using that 601st browser for their actual production environment. "All" doesn't just hold to browsers. It's too easy to say "all operating systems", "all languages", "all applications", and so on.

When the word "all" or "most" or any unquantified word appears, remove it, and replace it with something else. Typically, you'd use "including but not limited to", followed by a list, usually with version numbers. For instance, "shall work with all browsers" could be replaced with "shall work with browsers including but not limited to Chrome version 90.0.4430.93 (x86_64), Safari version 13.0.2," and so on.

12.4. The Words List

Requirements specifications should be written entirely in terms of what the facts are now, what the facts will be, what is desired, what the system must do, and what the users are likely to do, often with the added means to deal with what might happen. Specifications should be written in the third person only, meaning that "I", "we", "my", "mine", "you", "your", and "yours" should not appear. Exception: Use Stories are generally written in first person.

With that limitation, you've got everything you need to know about phrasing for the less formal specification styles, such as Use Case. But, when writing in the very formal specification styles, such as IEEE 830 or MilSpec, and especially in contracts, certain words are preferred and have special meanings:

Can: "Can" means that something is possible, and not that we're giving permission. For instance, a disk drive can fail at any time.

Could: "Could" means that something is possible, but usually conditional on something else.

Did (including implied "did" using an "ed" at the end of a verb), Does (including implied "does" using an "s" at the end of a verb), Is, and Will: These words are all direct statements of fact. Use these words only when the fact holds with or without the action of your team.

May: In many circumstances, "may" is unclear. It can mean that something *can* happen, that we have given *permission* for something to happen, or that something *may* happen. In any circumstance where "may" is equivalent to "can", use "can". In any circumstance where "may" means "might", use "might". You may use it where permission is given, but it's better to state that something is permitted or allowed.

Might: "Might" is the word to use when an event has some likelihood of occurrence.

Must and Shall: These are the words to use when something is a requirement. You generally want to lay a "must" or a "shall" on the system. You must not attempt to lay a "must" or a "shall" on a user, because you can't control the users, nor are the users listening to your guidance 100% of the time. The use of "shall" is traditional in requirements, but the more modern "must" is certainly acceptable. Exception: It's ok to state that the user must do one step in order to proceed to some other step, but since that's really a requirement on the system, even this exception would best be rewritten as a requirement on the system. *Note: In the UK, "shall", when used with the first or second person, means "almost certainly will", but since we're not writing specifications in the first or second person, that doesn't matter.*

Should: "Should" is the much weaker relative of "shall". It is advisory only, and has no bearing on requirements at all. Think of it as meaning "it would be nice if". If this word appears in a requirements document at all, it should only appear when no other word will do. QA personnel should ignore any sentence with "should" in it.

Would: "Would" is the conditional form of "will". It should be used rarely, if at all, and even then, mainly to describe possible external stimuli.

12.5. How Things Can Go Wrong

Use of the wrong words can mean that you're obligated to things you hadn't planned on, or that your subcontractors are not obligated to something you needed from them. The "deadly words" can cost a lot in QA time even when no contract is involved.

12.6. Discussion Questions

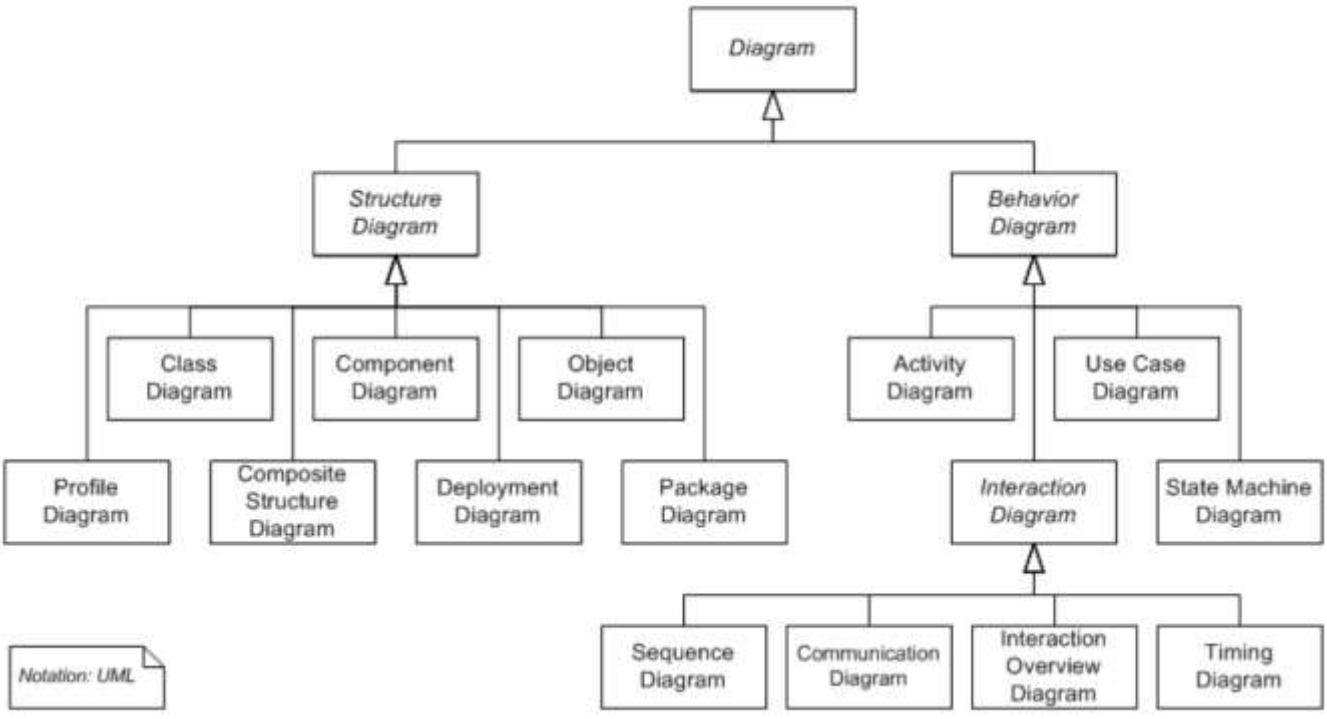
1. What are good ways to replace "all" or "any" in specifications?
2. When should "all" and "any" be used, and when not?
3. Where you live what are the differences, if any, between "must" and "shall"?

13. UML and other Diagrams

When diagrams are used, the diagrams used are typically from a category called "UML diagrams", with "UML" meaning "Unified Modeling Language". That doesn't mean that you can only use UML diagrams. UML is shown because its use is fairly common, and you may be required to use certain UML diagrams by management. The main rule in the use of diagrams, though, is that diagrams should be used to make things clearer, and not to check off some box. If UML will make your document clearer, use it. If some other diagram or image will make your document clearer, use that. If text alone will make your document clear, stick to text alone. *Diagrams used in the examples below are used under the Wikimedia License. Each is taken from the Wikipedia section on UML diagrams, and author names shown are Wikipedia user names.*

13.1. The UML Class diagram

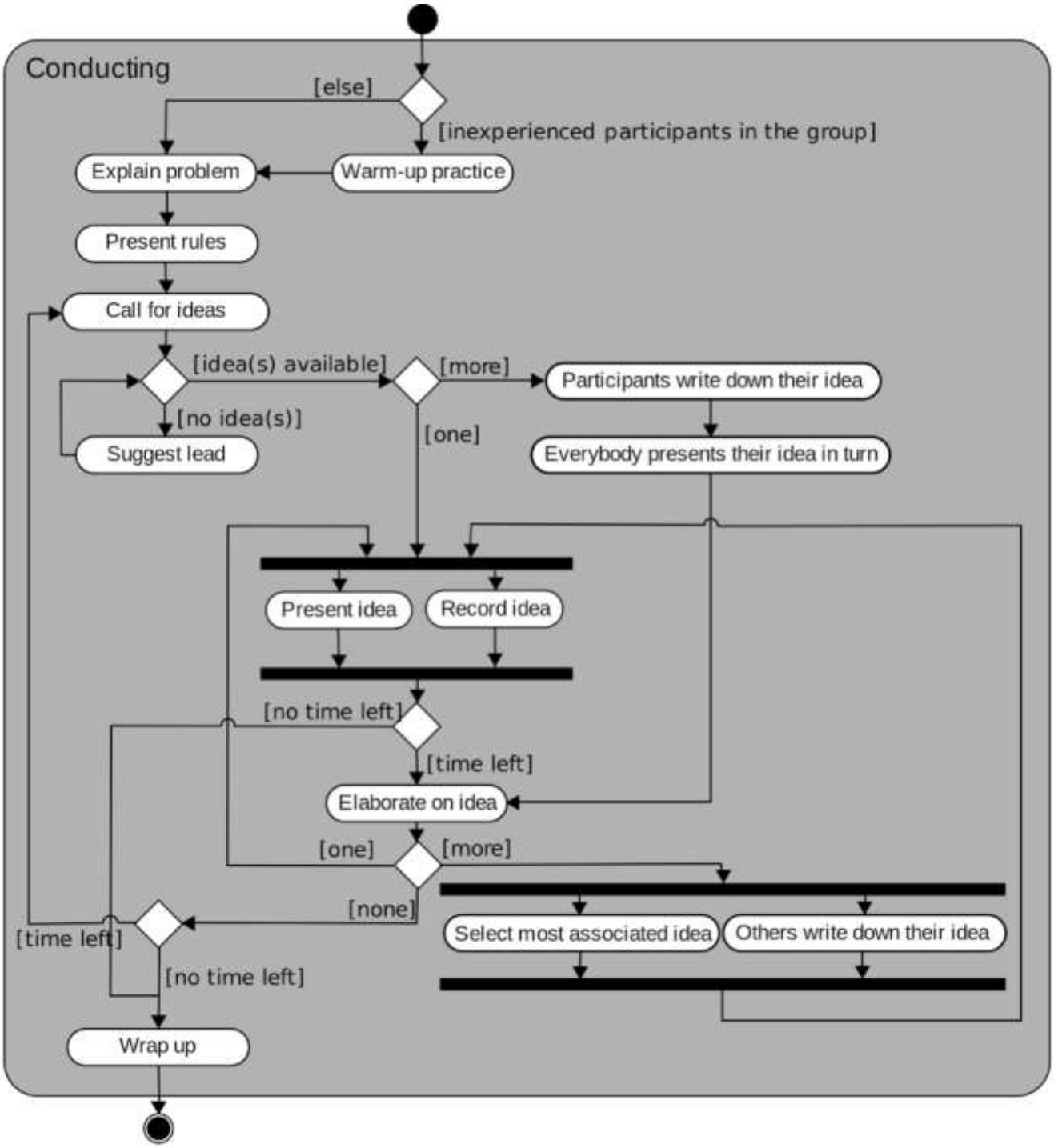
The Class diagram is used to show hierarchies of any sort. In a requirements document, you might use this to show an enterprise system, the component systems, and subsystems. You would typically not show class-oriented language's classes, because that's best left to the designers of the internal system. In this instance, "class" means anything with an "is a" relationship. Here's an example:



The boxes indicate a categorization or class. The arrows indicate a "child of" or "is a" relationship. The symbology in the lower left is a note or title.

13.2. The UML Activity Diagram

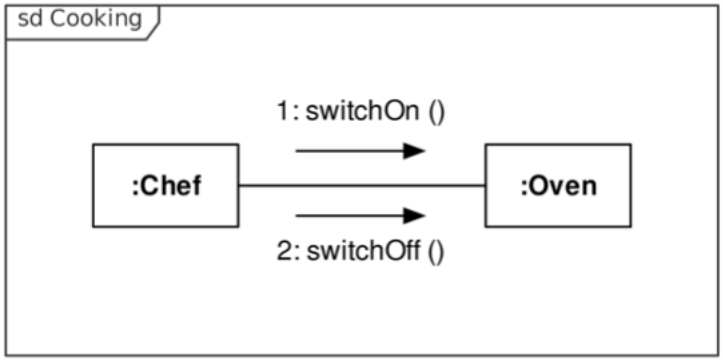
The UML Activity diagram is very close to a standard flowchart. It is used to show the sequence of events, for people, for programs, and for other processes. It includes activity steps and decision-making steps. Here is an example (by Gwaur):



In this symbology, the black circle indicates the start of the activity-set, and the circled black circle indicates the end of the activity-set. Thin lines and arrows indicate what comes next. Note that when lines cross, one of the lines will "hop over" the other. When two lines merge, the lines simply meet, The ovals/rounded rectangles (for lack of a better name) show activities or tasks. The diamonds indicate a decision point. When the flow of work or execution leaves a decision point, each decision is labeled (with brackets) on the outgoing lines. When the flow of work or execution splits into parallel tasks, or joins from parallel tasks back into a single flow of work or execution, there is a wide bar to indicate that.

13.3. The UML Communications Diagram

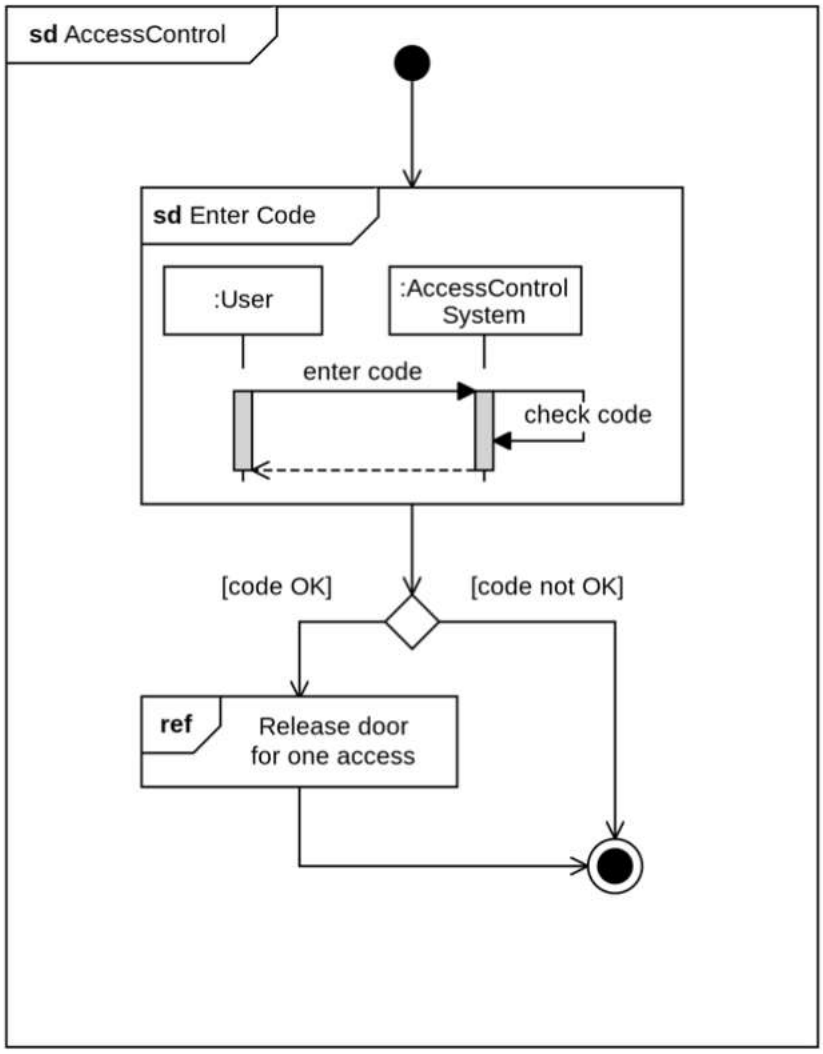
This type of diagram is used to show communications paths. The image shown below (by Ommler) is a trivial example:



The various items that communicate with each other are shown as boxes, with the items' names in the boxes. Colons are a part of this, but we're not sure why. The lines connect the communicating entities, and the arrow-and-message pair show the direction of communication and the message. These could involve message names, IDs, and/or the methods called to invoke the message. This example shows IDs and methods calls. The item in the corner is the title.

13.4. The UML Interaction Diagram

The UML Interaction diagram shows interactions between (typically) the users, the system, and possibly peripheral devices. It is also much like a flowchart. A set of UML diagrams should use flowcharts, Interaction diagrams, or Activity diagrams, or no diagrams of this nature, but these should not generally be mixed. Here is an example of an Interaction diagram (by Stkl):

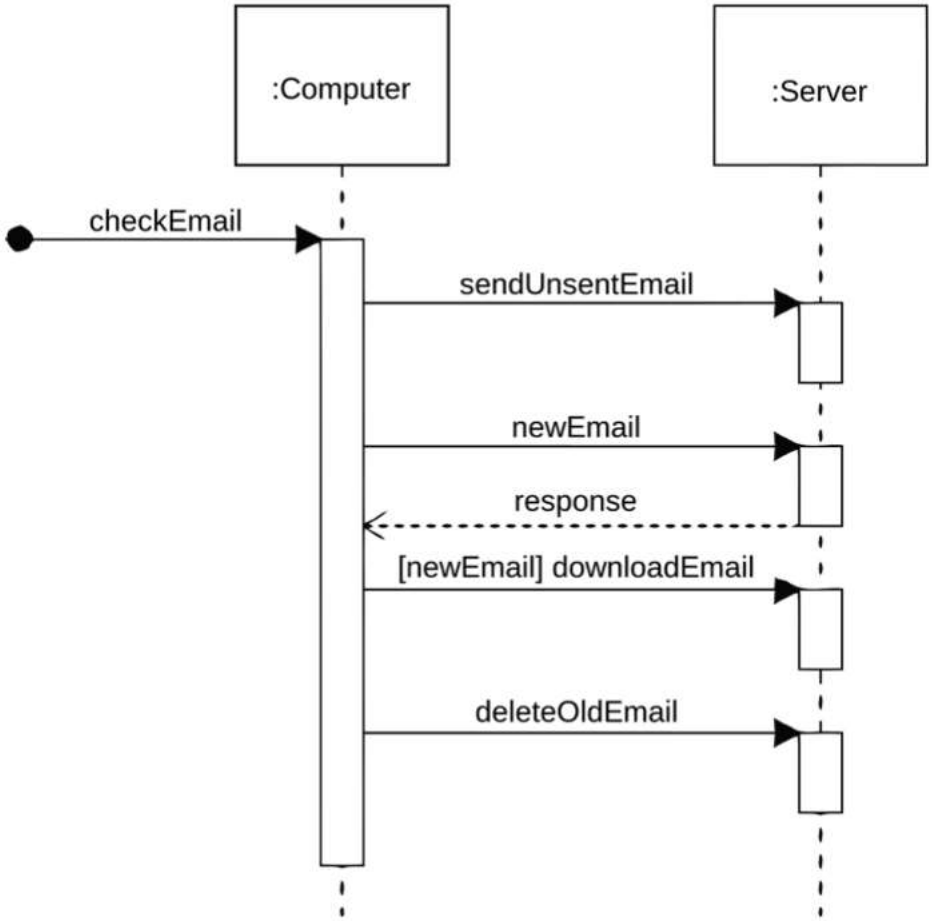


Much of this chart is as explained above for the activity chart. Note that this one also has titles and sub-titles, This type of diagram intermixes flow with timing (seen below).

Sheldon's recommendation is to stay away from this type of diagram.

13.5. The UML Sequence Diagram

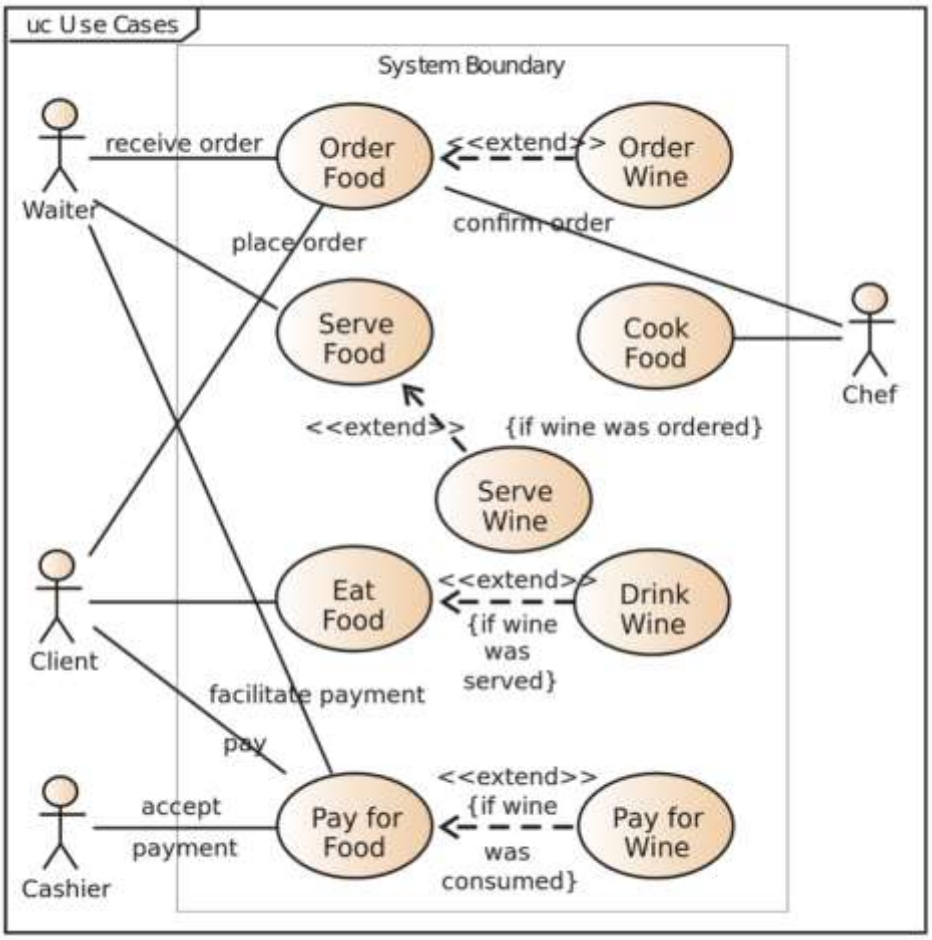
The UML Sequence diagram is useful to show complex timing requirements, and often involves around three communicating entities. Simple timing requirements generally don't need any sort of diagram. This sample shows two entities:



This is actually one of the more important forms of diagram, and is used when the interaction timing or message timing is very important. The boxes at the top are the process, system, or function names (again with colons). The dotted lines are just to keep things easy to relate to the function. The black dot indicates the trigger or initiation or start of the transaction set. The vertical rectangles indicate processing time, with time passing top to bottom. The arrows and the names over them indicate calls or messages. Sometimes a dotted arrow is used to indicate a response, and sometimes to indicate an optional message.

13.6. The UML Use Case Diagram

The UML Use Case diagram is commonly used to show how functions fit together. The diagram (by Kishorekumar) shown here is a bit more complicated than you'll usually see, but it shows all the components that can appear:



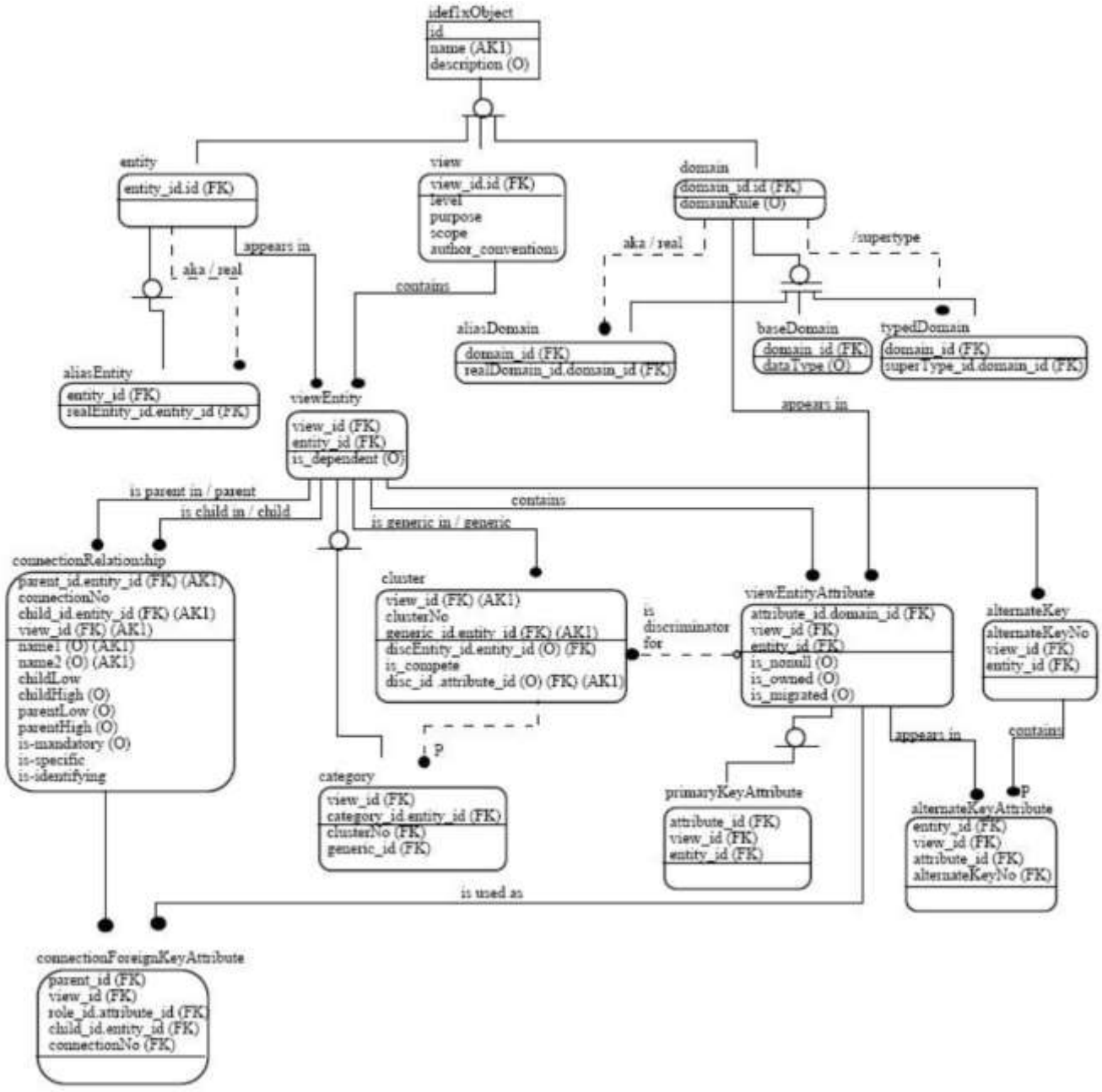
Shading or coloring is not required. The symbols shown here as "<<" and ">>" are usually shown as "«" and "»". If you're on a Mac, those are typed as Option-\ and Option-|, respectively. If you're on a PC, hold down ALT, then type 171 or 187, respectively, on the numeric keypad, and then release the ALT key.

The actors are depicted by stick-figure people, with names. The system under consideration is not usually depicted as an actor, but if it (or some other system) needs to be, then an addition, "«System»" will appear. Functions or functional areas will appear as labeled ovals, and lines will connect actors to functions, when the actors and the functions interact, and functions to functions when they interact with each other. For instance, In sending an eMail, the Sender actor would connect to the Send function, which would connect to the Receive function, which would connect to the Receiver actor. When one function extends or includes another, a dashed arrow and "«Extend»" or "«Include»" appears. When there is more than one system involved (or

optionally always), a faint rectangle appears around the functions of each system, grouping them, with a label indicating which system. Also appearing in the corner of this sample is a note or title.

13.7. IDEF1X Data Relationship Diagram

The most common form of visually documenting data structures is the following format:



You should probably not need to draw any diagrams of this nature to do requirements, but you may need to be able to read it, especially for Gap analysis purposes. Generally, each rectangle or

rounded rectangle will indicate a table. With that, there is a line, and over the line is the primary key of the table. Below the line are the other data. "FK" means Foreign Key, which references the primary key of another table (or in rare cases, back into the same table). The lines show how those records (usually using FKs) connect. Often, this will be combined with "crow's foot" notation, in which a line or "1" crossing the line means that there is exactly one connection at this end of the line, a circle or "0" crossing the line means that there may be one connection at this end of the line, and a crow's foot or ">" or "<" crossing the line means that there may be any number of connections at this end of the line.

13.8. Other UML Diagrams

There are a number of other UML diagram types, but none of these are recommended for requirements documents: Component, Composite Structure, Deployment, Object, Package, Profile, and State diagrams.

13.9. How Things Can Go Wrong

Leaving out a diagram that clears things up may make the document hard to understand. Remember the adage "a picture is worth a thousand words". Putting in a diagram that's not needed is a waste of time for you and for your readers.

13.10. Discussion Questions

1. When should you use diagrams? When not?
2. What are the most common diagrams.
3. Each of these diagrams is in use for a particular reason. Can you name the reasons?

14. Gap Analysis (and GAAP)

So far, just about everything talked about has been for projects involving entirely new work. But in many circumstances, you'll be working on changes to existing systems, including replacing systems, and tying existing systems together. When there is an existing system, you generally want to know (and specify) "where do we want to go from here?" and "how do we get there?". That's where the Gap Analysis comes in.

The "gap" is the gap between where you are and where you want to be. A very common form of gap analysis is to first document the entire state of the current system, and then to document the entire state of the new or modified system, and to then show the gap as a list of things that need to be added, things that need to be modified, and how, and things that need to be removed. Sometimes, you can just directly gather the list of things to add, change, and remove.

When prioritizing work to be done (after the Gap Analysis), you'll almost always want to put deletions as last priority, since the deletions don't tend to add value. If the changes can be made without the additions, those will generally be prioritized first, since changes tend to be the "low hanging fruit", meaning that that work has a better return on investment than does new implementation. But, if the new implementation is required to make the other changes, then obviously the new implementation needs to be done first.

Another thing that's pronounced "gap" is GAAP, which stands for Generally Accepted Accounting principles. Whenever you're doing anything with accounting, follow GAAP unless directed otherwise. Check with the firm's accountant if in doubt. These often occur together, as in "The Gap Analysis shows that the system doesn't follow GAAP.". The two most common GAAP violations are not balancing each credit with a debit, and not tracking where money is coming from or where it's going to.

15. SRS, BRD

There are a few closely-related document types that you'll come across:

SRS, SRD, Software Requirements Specification, Software Requirements Document: This is the type of document that you'll normally be producing. It is the requirements for the software, hopefully incorporating the how, what, and why of the software to be produced or modified.

BRD, BRS, Business Requirements Document, Business Requirements Specification: This is a document that specifies what the business needs, and why. This often comes before the SRS and is the reason for the SRS.

FRS, FRD, Functional Requirements Specification, Functional Requirements Document, or just Functional Spec: This document is sometimes produced in Waterfall projects, as the detailed as-built documentation, produced during the internal design and development stages. This last document is generally not the concern of the business analysis or requirements documentor.

16. BDD, DDD, TDD, *etc.*

There are a number of approaches to how to handle the development effort, especially (for this section, at least), in what order to do things, and in what order to be concerned about things. Most often, this sort of thing won't affect the requirements team, except that in TDD (to be explained below), tests that the various components might have to pass might be considered the most important part of the requirements. These various styles are listed here:

ATDD, Acceptance Test Driven Development, TDD, Test Driven Development: In test-driven development, writing the test is the thing that's done first. Software is considered valid if it passes the test cases. In poorly-managed projects, the test plan or even worse, the test code, serves as the requirements, and you'll be held to write tests. In well-managed TDD projects, the tests are written from the specifications, and QA will start their work in the same sprint as the developers, or even before.

BDD, Behavior Driven Development: This is a form of development in which the users' behavior drives the development plan. Observing the users' work, or prospective users' work, is a great way to gather information on the real requirements of the system, and when you do that, you're in effect using BDD.

DDD, Domain Driven Development, Domain Driven Design: Although this term is not often used, this type of development is the most common one. The development effort focuses on the problems to be solved, which means that your requirements document is driving the work.

FDD, Feature Driven Development: In this type of development, the necessary features are the key to development. Here, what is developed relies on the features you lay out in your user stories and software requirements specification documents.

OADD, Object Oriented Design and Development: In this model, everything is relegated to classes, as the term "class" is used in object oriented programming languages. Normally, the use of this type of development effort should not affect the requirements gathering or documentation, since all of this should normally be behind the black-box line; but, in rare cases, the requirements documentor may be called upon to lay out which classes are to do what.

You may have noticed that there's a great deal of overlap in these descriptions. Don't worry about any of that, unless you find yourself working for someone who is a fanatic about one of these schemes. In such a case, just make sure that you phase each explanation of what you're up to in terms of that development style, and that you don't violate any of the rules of that style. Absent fanatics, you'll likely do some of each, as each circumstance warrants.

17. Overscoping

"Overscoping" is defined as including unneeded items into the design, and thereby causing the project to take too long to complete and/or cost too much. In some cases, overscoping can cause a project to fail completely, by producing a product that's hard to use, too slow, or just-plain ugly, or worse, by failing to produce a product at all, by running out of money or time.

How overscoping tends to happen varies with the overall project methodology.

17.1. Waterfall

Waterfall projects are the least susceptible to overscoping, but when it happens, it happens up-front, and all at once. The requirements-gathering phase happens all at once as the first phase in the project. If the requirements gathering phase lasts too long, it delays the entire project. The key to avoiding overscoping in a Waterfall project is to ask "Is this required?", "Is this feature worth the effort?", and "Is it within our stated scope/charter?". Priorities don't matter in reducing overscoping in Waterfall projects, because the priorities only influence delivery schedules/sequencing, not whether or not something will be done.

17.2. Agile

In an Agile project, overscoping generally won't prevent a project from being delivered and won't even delay anything, if things are prioritized properly, but it can make the project go on forever, with virtually unlimited costs. There are two ways that overscoping tends to happen in an Agile project. The first is that some needed feature is itself overscoped, in that there are unneeded sub-features in a feature, or features in a module. This type costs the most, because the extra bells and whistles ride in with the needed feature. The second is that functionality that is not at all needed may be added to the backlog independently. These are only of minor concern, because if characterized properly as very low priority, those features will remain at the end of the backlog, and at some point, project management must look at the remainder of the backlog and decide "we don't need any of this — we're done". That's easier suggested than done, because often in an Agile project, nobody wants to end the project, because it could mean loss of a leadership position, or otherwise having to find something else to do. Don't fall into the trap of "needing" to find more requirements. It's ok to say "done" and go on to the next project.

17.3. Spiral

In the Spiral development model, there are two ways that overscoping happens. The first is trying to include too much in a release, because it may be that including those features would take too long or cost too much. The second way that Spiral overscoping can occur is that at the end of the release lifetime of the product (the point at which it's no longer worthwhile to issue new versions), producing more versions may cost more than they're worth on the open market.

17.4. Discussion Questions

1. Under what circumstances is expanding the scope a good idea?
2. Under what circumatances is it a bad idea?

18. Friendly and Adversarial QA

One would hope that in all projects, the QA staff and the development staff would get along well. QA's job is to validate Development's work. In most projects, that should be a cooperative venture. If there are disagreements between QA and Development, the likely cause is substandard requirements that leave room for interpretation.

But, there is a special case in which the relationship between QA and developers is an adversarial relationship: The rule of thumb in aerospace and defense projects is that the average bug will kill one person. QA's job, on many such projects, is to prove that the work is not yet ready to ship. This puts them in an adversarial, but hopefully still friendly, position. In this type of project, your requirements will be subject to QA themselves.

19. Best Practice: No Path to an Error

Let's say you have a calculator that does square roots. If the display shows "-1", and you press the "√x" key, you can assume that you'll be getting some sort of error message or indication of an error. On a physical calculator, there are physical keys that are there all the time, so of course, someone can press the key that leads to an error.

But, in modern programming, the best practice is to give the user no path to an error. So, using the calculator example, in a calculator with a drawn keyboard, the best course would be to disable the "√x" key when the accumulator is negative, thus preventing the error condition in the first place. The rule is simple: Whenever possible, leave the user no path to an error condition. This will mean temporary disablement of buttons, menu items, and other controls and parts of controls.

20. The Intersection between Requirements, Business, and the Law

Sheldon's perspective:

> The small print: The framed parchments here say "Computer Science" and "Software Engineering". They say nothing about being a lawyer, nor anything about having a business degree. Yet, I find myself doing business and writing and evaluating contracts. Just the same, don't take anything in this book as financial or legal advice.
>
> When I left school, I got a job. I thought that I was going to be a programmer. Three weeks into the job, I was told that I was now a data base consultant. When I arrived at the project to which I was assigned, I was told that the project was in trouble, and that I was to get the data base "handled". I quickly found that the data base requirements were unknown, and that the nature of the interface between the programs and the data base were also not well defined. That was one of the main reasons that the project was in trouble. That, in turn, caused contractual issues, which in turn was on the verge of causing legal issues, which would have caused financial issues.

Every business is much like a living organism. Organisms want to eat and grow. Businesses want to earn money and grow. Even nonprofits want to earn money and grow, even if the money coming in is used for charitable purposes, or for a cash reserve. The programming (and the requirements gathering which elicits the programming) takes place so that the business can better its bottom line. Depending on the nature of the business, bettering the bottom line usually means one or more of spending less in the long run, earning and/or producing more in the long run, and/or becoming more effective. Your job as a programmer or analyst will seldom directly be producing "cool" stuff; it will be the job of producing the needed stuff. If that stuff is cool, so much the better. If what you're producing doesn't affect the bottom line as described above, then it's useless or worse.

We've mentioned above, that various types of business arrangements, uses, and release plans, and how they lead to the use of certain project methodologies, with recommendations to Waterfall, Agile, Spiral, and others. We've also mentioned how those methodologies tie in with time-and-expenses contracts, and with fixed-price contracts.

Time-and-expense contracts are very simple, generally. Such contracts are basically a statement that we're going to charge them so much an hour, plus any direct expenses. There are parameters on how much we can spend, and the time period. For instance, with one customer, the contract was for 80 man-hours per week for 5 months, at a given price per such hour. In such contracts, the customer usually has the right to terminate the work with no notice. Here, it's not the contract that controls what's going on. Instead, it's the customer's perception of how well you're doing for them. Show them good stuff early and often, and you'll keep getting the money. Succeed in the project (whatever that means to the customer), and you're likely to get repeat business. That's where the best profits come from.

Fixed-price contracts, on the other hand, almost always force you into the Waterfall model. That's because in the fixed price model, you're going to get a specific amount of money for doing

a specific thing. Sheldon most strongly suggests that in such a contractual arrangement, that the specifications must be complete before entering into the fixed price contract. Here's why: Let's say the customer offers you $30,000 for doing a job for them. If the customer wants you to do $30,000 worth of work for that, everyone will be happy. If it turns out that it's only $30 worth of work, the customer will not be happy, and the contract might or might not be enforceable, since there's a "Doctrine of Fairness". If there's $30,000,000 worth of work to do, you're going to go bankrupt and/or get sued or be the one who sues. Don't get yourself into that. If the customer wants a fixed-price contract, then the customer needs to deliver the specifications as to what is needed, or the requirements gathering process needs to be a separate time-and-expenses contract.

When combining time-and-expense specification writing with fixed-price systems production, keep the two contracts completely separate. Once the specifications are done, you and the customer can negotiate a price, or the customer may go elsewhere for that phase. That's not always a bad thing. For instance, a company produced a specification. The company then bid a price for the project based on that specification. The customer didn't like the price and went elsewhere. That was fine, since the company could not have delivered the specified product for the price that the customer and the third-party provider decided on. (As it turned out, that third-party couldn't produce the product for the price they quoted, either.)

A use-case style specification is usually fine for time-and-expense projects. Both sides know generally what's going to happen, and things pretty-much go in that direction. If minor or even major mid-course corrections are needed, then they can be made within the wiggle room of the use cases or user stories. The cases and stories can be adjusted as needed, too. Fixed-price contracts are best served by IEEE-830-style specifications. The more everything is nailed down, the better everything will go for both sides. Very often, a fixed price contract will either directly include the specification document, or include it by reference. Thus, the specification is, in effect, a contract in and of itself. Thus, the specifications should be written in contract-like language.

Recall §9.4 (Offer, Acceptance, Performance, and Payment).

A note from a paralegal reviewer: Parties to a contract also need Capacity, which is the legal right and ability to enter into a contract (for instance, I can't sign a contract between you and IBM, because I don't have the legal right to represent IBM), and have the (supposed) ability to actually perform (for instance, I can't be held responsible to turn straw into gold overnight).

The language of contracts and specifications must be as clear, concise and unambiguous as possible. Define everything, especially abbreviations. Generally, an abbreviation or defined term will generally be of a form in which you use a term in its full form, and then the capitalized abbreviation in parentheses and quotes. For instance:

> This agreement is by and between The First Fake Bank of the West ("Lender") and John Q. Smith ("Borrower"). Lender intends to...

The exact same sort of thing is useful in specifications. but without the quotes. For instance:

> The data base management system (DBMS) will communicate with the workstations using the local area network (LAN). The DBMS shall...

Note that the language of the contract is important, and that translations don't always work well. If a language or a specification is in more than one language, or used in translation, make sure that one of the versions is the reference version or controlling version. Despite all of its ambiguity, English is one of the least ambiguous languages.

One more thing in the area of translations, which seldom applies, except to notices: If two or more countries are involved, a notice in French is a notice, whether or not any of the countries involved speaks French, so if you ever get a document in French, read it or have it translated and then read it.

21. The REST versus SPA Dilemma

REST stands for REpresentative State Transfer, and SPA stands for Single Page Application. These terms both apply to web-borne applications.

In a REST application (or web-site), the URL (Uniform Resource Locator) carries the state information (either visibly, or as hidden data in the HTTP PUT or POST request). For instance, we could go to an internal web-site named Example to log in, and then get directed to Example/MainPage for our main menu page. If I then select "Employees" to see a list of employees, I'd then be at the URL Example/Employees. If I then select employee #100, the URL might change to Example/Employee?id=100. Note the "?id=100" part of the URL. That is the "argument", setting "id" to 100. If there are multiple arguments, they will be separated by "&", for example, "?id=100&dept=10". Pro: You can bookmark pages in a browser, and come back to that location in the browser, such as employee #100, without having to track through the menu system. Con: Each time the application goes to a new URL, the whole page needs to regenerate and reload.

In a Single Page Application (or web-site), portions of the page are loaded through AJAX (Asynchronous JavaScript And XML [eXtensible Markup Language]) calls, in which the web page makes a request to the server in a form of an AJAX packet, and receives a response from the server, which it then uses to modify itself. It doesn't matter to us as requirements writers, but the request and response also typically moves via the HTTPS protocol, and any data is typically transmitted as arguments, with the response typically being in the form of JSON, HTML, or XML. The pros and cons are exactly the opposite of REST: Pro: Faster than REST, because only pieces of pages are being generated and transmitted. Con: There is no way to save or restore the state from the user's end.

The reason this is being brought up here is because the difference between REST and SPA applications is visible to the user, and must be decided as a part of the requirements when the application is in the form of a web-site.

If you look at job listings, you'll see ads for REST programmers, and other ads for people with significant SPA experience. That's because there is one group who *know* that REST is the right way to do things, and that SPA is not. Of course, there's that other group who *know* that SPA is the right way to do things, and that REST is not. *Sheldon's advice: Agree with the boss or customer. If they don't have an opinion, and the site is customer-facing and single use, use SPA. If the site is internal-facing and continuous use, use REST. If it's external and continuous, or internal and single, it's a toss-up.*

22. Prioritization

Unless you have enough staff to work on everything all at once, you need to prioritize things. There are a number of factors that go into prioritization, and we go into those below. It would be nice if there was a formula we could apply on those factors and just come up with a sorted ordering, but there's not. In each business circumstance, you have to weigh the factors against each other to come up with each task's prioritization, each module's prioritization, or each individual requirement's prioritization. Those are ordered from most likely prioritization scheme to least likely.

When you write a requirements specification, you'll usually include the priorities. Some companies use High, Medium, and Low. Some also add Emergency, Eventually, and even Maybe or the equivalent. Other companies label things as First priority, Second priority, and so on. Still other companies give items a priority score, 1 to 100, 1 to 10, 1 to 5, or whatever number they find convenient. If your documentation uses numbered priorities, make sure people know what the numbers mean.

22.1. Urgency

Urgency is often the most important prioritization concern. If it's December 1st, and there's a year-end piece of work that needs to be done as well as something that has to be done for tax time (usually April 15th in the US), then clearly the year-end work has a higher urgency. So, one way to sort out urgency is by using due dates. However, that won't always do the job, especially when things don't have specific due dates. There can also be chain-of-command issues. For instance, if your boss wants Thing A done, somebody else's boss wants Thing B done, and the CEO wants Thing C done, then political or chain-of-command issues say that the prioritized order is Thing C (because the CEO outranks your boss), Thing A (because you answer to your boss), and finally Thing B. There are also the "what if" urgency issues. What if the item under consideration doesn't happen? What if it does?

22.2. Stability

Some items to be done are very stable, and some are very volatile, and most are somewhere in-between. Normally, it's best to prioritize the stable ones first. Here's an example why: Let's say we have Thing A to do, which is very stable, and Thing B to do, which is very volatile. If we do Thing A now (and by "do", we can mean anything between gathering the requirements and delivering the release containing that Thing), in the next iteration (be that iteration a sprint, a release, or any other scheduling term), we can get to Thing B. But, if we do the volatile Thing B first, in the next iteration, we may very well be working on Thing B again. In the worst case, Thing A will never get done.

22.3. Certainty

When we first start gathering requirements, in the worst case, we don't know anything about anything, and we know that we don't know anything about anything. (Ok, there is an even worse case, in which we think we know, but we're wrong, but let's leave that one out of the

consideration for now.) As we do our research, and possibly some tests or experiments, we come to believe that some of the requirements are rather certain, and some remain rather questionable. The more certainty we have about a requirement, the more priority it should get. You don't want to release the requirements for an airport, only to find out after it has been built that you needed an aircraft carrier.

Some companies use the statuses of Ready and Not Ready. Some have a few more categories: WAG (Wild Guess), SWAG (Semi-scientific Wild Guess), Almost Certain, and Certain. Others (again) use 1 to 100, 1 to 10, or 1 to 5. Again, if you use numbers, make sure that people know what the numbers mean.

22.4. Size

The size, or estimated resource requirement, of a task is often measured in hours, man-hours, man-days, or man-months. (Those are the terms used, and those terms are generally considered ungendered.) Some companies use a "work unit", where the conversion between work units and man-hours is some constant or fuzzy constant particular to that company. (In Sheldon's company, that's 2-3 hours for time-and-expenses contracts and the price of 2½ hours for a fixed-price contract.) Some companies use points. Points are different than work units, in that a 2-point item takes longer to do than a 1-point item, but not necessarily twice as long. Some companies use XXS, XS, S, M, L, XL, and XXL. Often, the smaller items are prioritized ahead of the larger ones, so that the group can show the maximum number of items as completed at any time.

22.5. Benefit - Cost

Weighing cost against benefit can be important. Sometimes this is Benefit less Cost, either as a dollar amount, some other amount, or some immeasurable quantity. Sometimes this is the Cost/Benefit Ratio, measured as Benefit divided by Cost. Sometimes this is just a billable number. Note that Benefit can also be stopping a continuing loss or just improving something. Often, you (or the project, or at least the project manager) will want to get the greatest possible benefit in the shortest possible time. That affects prioritization rather directly. When Benefit-Cost can be expressed numerically, tasks become easy to sort into priority order.

22.6. Reliance

Oftentimes, one module will rely on another being done. In such cases, the relied-upon module must be prioritized before the reliant one.

22.7. Waterfall

Remember that in a Waterfall project, there will often be deliveries before the final one, often having version numbers starting at 0.1 to indicate pre-release copies. All of these prioritization rules apply, even though the final product is what counts the most.

22.8. Agile

In an Agile product, there are either continuous deliveries, or a distinct set of scheduled releases. In an Agile project, all of these rules apply all the time, because it matters which release (especially in sprint-based Agile projects) the feature, fix, or improvement will get into.

22.9. Spiral

In a Spiral project, you're most often dealing with a set delivery date, such as having a project ready to sell at a specific trade show. Prioritization becomes a matter of deciding the feature set. For instance, you may decide on 100 new features, but only 50 can be engineered into the product. Not only do you need to prioritize directly to determine which features are in the release and which will have to wait, but you also have to decide which features work well with the others to make a complete suite of related features. So, some features will be prioritized individually, and some will be prioritized in a group.

22.10. Removals

If something is to be removed from a project, that is often prioritized last. Often the real reason to remove something from a product is simply to make future maintenance less expensive.

22.11. Discussion Questions

1. Without looking at the list here, or elsewhere, can you make a list of items for consideration in prioritization?
2. Keeping in mind that this will actually vary from project to project, can you order your list from generally most important to generally least important in doing prioritization?

23. Doneness

The last step in requirements engineering is determining both the "doneness" of the requirements, and of the product.

23.1. Waterfall

In the Waterfall model, doneness of the requirements means that all of the requirements have very high confidence, and that the requirements gathering-and-documenting process is ready to end, with the Requirements team moving onto their next project.

23.2. Agile

In an Agile project, there are two kinds of "done", sometimes three.

A requirement may be considered gathered once you know about the requirement. You can then document it, and put that requirement's document or documentation into the "backlog" (the things yet to do queue), or put it into the "requirements backlog", to be later refined and moved to the regular backlog.

You may also reach the step in which there are no more requirements to gather and document. That's when the whole project is done. What "done" means varies from company to company, and within a company, from project to project. In some companies and project, "done" means that the project is completely done, and that the requirements people can go to their next project, and that the developers have one more sprint, and then they go to their next project, and that the QA people have two more sprints to go, and then the project is completely over. In other companies and projects, "done" means that the backlog is completed, and that he project will go into "maintenance" mode, in which minor change requests will continue to come in, and sprints may shorten to as little as four hours, typically with a skeleton crew left on the project, even as low as a fractional developer, meaning one who is assigned to a project less than 40 hours per week.

23.3. Spiral

In the Spiral model, there are also two versions of "done".

In any one loop around the spiral (meaning a given release cycle), doneness pertains to finishing the requirements set for the release (which relies heavily on prioritization and what the Sales department says will sell), and then completing the release.

All things come to an end, and for each product, there will be a last release, even in successful companies. (Even Apple stopped making Apples, and Microsoft stopped selling MS/DOS.) Sometimes a product comes to the end-of-life point because new sales are not expected to cover the costs of production and marketing, but sometimes because there's a newer, better product, and the company doesn't want to split the market. Ray Dolby, talking about that latter reasoning, said it this way: "Put yourself out of business before someone else does".

23.4. One Last Extra Step

After everything is done, there is often one last step: A retrospective of the whole project to determine what could have been done better, so that the next project might go more smoothly.

23.5. Discussion Questions

1. If your experience (if you have any in this regard) do internal projects get to "done". If so, what percentage?
2. Similarly, if you have experience with projects and customers, has "done" gone smoothly, or have there been hangups? If so, what?

24. Combining & Refactoring

As you gather requirements, you will often notice that many features, functions, screen, pages, and other items will have a great deal of commonality. For instance, in a business-control setting, you're likely to have a number of screens that search for various things, and list the results, such as finding and listing customers, finding and listing suppliers, finding and listing employees, and so on. Similarly, you're likely to have screens to edit a customer record, to edit a supplier record, edit an employee record, and so on. You don't want to be listing requirements for each of these separately, for two reasons. First, if there are 100 find and list screens, you don't want to describe the action of the Search button 100 times, but even more importantly, you don't want people implementing 100 separate Search buttons. As a cascaded effect, 100 implementations of Search would also need 100 QA efforts.

The remedy for this is for you to notice that, in general, there is a Search function, and in general, there is an Edit function. Typically, you'd document the requirements for a search-in-general, and for an edit-in-general, and then document those as "abstract" requirements, meaning that the requirements will hold for specific functions down the line, but are not a function in-and-of themselves. Sometimes you'll be in a position to do this sort of documentation up-front, especially on Waterfall-oriented projects, but on an Agile-oriented project, you're more likely to have a function or two documented before you see the same thing pop up again and realize the commonality of the functions. When that happens, the process is called "refactoring", whereby the common part of the functions is factored out to its own abstract area, and the functions' documentation just refers to the common area, and incorporates it by reference.

Typically, you won't be involved in the code area or code design, but there are some things you should know about it anyway: In object-oriented (OO) programming (OOP) and design, we have "parent objects" and "child objects". The "parent object" is often marked as "abstract", meaning that it can't operate on its own, but that it lays out certain data structures and/or behaviors that each of the "child objects" will "inherit". For instance, if there's going to be an "object" to search and display employees, named "SearchEmployee", it will be marked as a child of some general search function, perhaps named "SearchGeneral". In pure procedural programming, the search functionality will generally be in or moved to a generalized search function and a generalized listing function, which will be "called" by the specific search and list functions. In a Waterfall project, things will just be built this way. In an Agile project, things will be built this way if you didn't have to refactor your requirements documentation, and will be refactored on the code side when you do have to refactor your requirements documentation. Don't worry about upsetting the programmers with such requirements changes, because they're used to doing this sort of refactoring, and may even notice and do this if you miss it, hopefully bringing it to your attention in the process.

25. Other Things that BAs Should Know

Most of the recommendations presented here are based on experience, and are heavily laden with opinion. Don't take what is said here as gospel, but at the same time, don't discard what is said here without consideration if your opinion differs. We welcome discussion on any of the issues presented here. The recommendations appear in italics.

This book covers requirements gathering and documentation, and to an extent, requirements tracking. However, to be an excellent practitioner of this art, it would be most helpful to have some additional domain-specific knowledge. Requirements engineering is a very broad field. The concept applies not only to software engineering, but to all other forms of engineering, and to most forms of businesses. For instance, if we asked you to set up a food establishment for us, as after successful completion of this course, you'd be better at it than someone off the street, because you'll have a better idea of what sort of questions to ask. But, back to software engineering: The following is a list of some things that would be very helpful to know before visiting with your first customer or stakeholder.

25.1. Language Classes

Likely you already know that there are a lot of programming languages. But also as likely is that you don't know that there are classes of languages. Below, we outline the more common language classes.

25.1.1. Shells

Shells are system-level command programs where you issue commands to run programs. This category includes Windows .BAT and PowerShell files, Unix/Linux shells (such as the Bourne Shell, sh, the Bourne Again Shell, bash, the C Shell, csh, and the Korn Shell, ksh), and IBM's mainframe control languages, CL (Control Language), JCL (Job Control Language), and TSO (Time Sharing Option).

25.1.2. Programmable Shells

Shells with some programming capability, such as a form of conditional command, such as IF and some sort of looping command, such as FOR. Most modern shells are actually in this category. When using shells, use programmable shells, and make sure to check for error conditions at each stage of the processing. It's a good idea to include the checking of return codes (the success/failure codes that programs generate) as a "non-functional requirement" in requirements documents. This requires that all programs should return meaningful return codes.

25.1.3. In-betweeners

This category includes shell languages that are so programmable as to seem more like programming languages, and programming languages that are often used to start other programs.

If efficiency is an issue, and it often is, even if you don't plan it that way, plan on using a compiled language.

25.1.4. JavaScript

JavaScript is the preeminent language used to program front-end web pages. JavaScript can also be executed at the back end. When used at the back end, it's often called NodeJS. There are pros and cons of using JavaScript at the back end. JavaScript is not as efficient as some other languages, but using it at the back end means one less language that the staff need to know. Recommendations: Use JavaScript at the front end, but not where HTML in combination with CSS can't do the job. When scalability is an issue, JavaScript should be used to enhance client-side validity checking, and there should always be "nonfunctional requirements" that validity is checked on the client side and again on the server side. Checking on the client side allows real-time visibility on whether any Submit button can be clicked, and prevents transmission of HTTP transactions that are going to be rejected (and thus also preventing the overhead of sending a rejection page), and server-side checking prevents outside hacking or bad data slipping through when a browser is set to ignore JavaScript.

There are some add-on languages to JavaScript. These include TypeScript, which enforces types of JavaScript, and "front-end frameworks", such as Angular and React (usually paired with Redux), or a subroutine package such as JQuery.

25.1.5. Loose versus Tight Languages

Loose languages allow item type to be changed at run time. Tight languages do not, and thus provide more checking capability. Using loose languages will get you running faster, so are nice for prototypes, but tight languages will decrease your overall costs, because fewer bugs will slip through.

25.1.6. SQLs

SQL stands for Structured Query Language. It's an ANSI (American National Standards Institute) standard for querying from and setting up Relational Data Base Management Systems (RDBMSs). SQL is generally conceptually divided into the Data Manipulation Language (DML), which does the creation, reading, updating and deleting (CRUD) of records, and the Data Definition Language (DDL), which sets up and modifies tables and the relationships between those tables.

Here are some other things you need to know:

25.1.6.1. Brands & Loyalties

You will find that some people have brand loyalty. *Don't argue with them.* Others do not. *For those without brand loyalty, if they have an SQL in use already, stick with it. If not:* SQL Server is typically the fastest at ad hoc SQL. Oracle typically stores data in the most compact form. DB/2 is typically the fastest at stored queries. MySQL, PostgreSQL, and the like are free.

25.1.6.2. Tables

Data is stored in tables, in rows and columns. There are zero or more rows that come and go, but one or more columns, which are relatively static for a table.

25.1.6.3. Queries

A query (SELECT command) is how we get data from tables, as is, or processed/correlated in some manner. Queries may be stored in the database as views. It's better (faster and less error-prone) to use a stored view than to send a query from your program.

25.1.6.4. Joins

Joins are how programs link the data from two or more tables together (or in some cases, how they link data from multiple corresponding parts of the same table). These are the main join types:

Inner Join: A join from table A to table B in which only matched rows are used.

Left Join: A join from table A to table B in which the rows from table A are used even if no corresponding B row is present. (A Right Join does the opposite.)

Outer Join: A join from table A to table B in which the rows from tables A and B are used, even if no corresponding B or A row is present (so long as there is a row from table A or table B).

Exception Join: A join from table A to table B in which the rows from table A are used, except where there is a matching row in table B.

25.1.6.5. Indices

An index is a way to vastly speed up reading at the cost of slower writes and more storage. Indices (or indexes) can be regular or enforce uniqueness. Proper index design is the key to fast execution in SQL systems. A missing index will cost execution time, as will an unused index.

25.1.6.6. Insert, Update, and Delete

These commands add rows, change values within rows, and delete rows, each affecting zero or more rows, depending on conditions.

25.1.6.7. Stored Procedures

Stored procedures are SQL programs stored on the database server. Whenever possible, move logic involving multiple SQL commands into a stored procedure.

25.1.6.8. Types

Within a program, data will have a type. Not all SQLs have all of these types, and more types are possible.

Boolean: Stores true/false values. In SQLs that don't have Booleans, use the CHAR(1) type with a value of 'Y' or 'N'.

Numbers: The NUMBER type can be integers, decimals, or floating point numbers. Decimal numbers are stored as true decimal numbers, but floating point numbers may be stored by some servers as decimal numbers, and by some as binary floating-point numbers. Some servers give you a choice of the floating point format used.

Strings: Strings can be fixed width, variable width with a maximum length, or of unlimited size (but slow performance).

Dates and times: The DATE type specifies a day; the DATETIME type stores time to the second, and the TIMESTAMP type specifies a microsecond. In Oracle, the DATE type is actually a DATETIME.

Selectors: A selector is a column that specifies a value in another table, usually actually stored as a numeric type (but sometimes CHAR) with a FOREIGN KEY constraint.

Nulls: The "no value" value. Nulls are not equal to anything, even to the extent that NULL is effectively not equal to NULL.

25.1.6.9. Normalization

Database designers sometimes get into trouble with "unnormalized" also known as "denormalized" databases. (But sometimes, they need to violate the following rules for speed.)

1:1 relationships: Don't store two kinds of things in one table, and don't store one kind of thing in two tables.

Move common data: If the same kind of multi-part data is stored in two or more places, consider moving that data to a table specifically for it.

Don't store derivable data: Don't store data you can figure out, unless it takes too long to figure it out.

Requirements advice: It's ok to list normalization as a "nonfunctional requirement", but only if people are given an out. Sheldon suggests "The database(s) used are to remain normalized at all times, except in such instances as are required for speed, and only when the reasoning for such instances are called out in the design documentation or comments. This rule does not apply to scratch tables, intermediate result tables, or reporting-only tables."

25.1.6.10. Constraints

The database can do run-time checking to protect itself from bad data. *Take advantage of every opportunity to add constraints. They take a little time, but protect the data from bugs and in some cases from malicious action.* Constraints also serve as documentation on how the data all fits together.

NOT NULL: This constraint prevents a column's value from being NULL in a row.

CHECK: A CHECK constraint states a fact that must be true of each row, as a formula.

UNIQUE: The UNIQUE constraint states that the value in this column or these columns must be unique within the table.

PRIMARY KEY: The PRIMARY KEY constraint states that this value for this column or these columns are both UNIQUE and NOT NULL. This constraint is also required in order to use a FOREIGN KEY constraint (below).

FOREIGN KEY: The FOREIGN KEY constraint states that this column (or these columns) must (if not null) refer to the primary key of a given table. This constraint can also be used to define what happens when a deletion or change is made to the record in the referenced table, with the choices being (a) don't allow a referenced record to be deleted or have its Primary Key changed, (b) if the referenced record is deleted, delete the record(s) that reference it, and of the Primary Key is changed, change the record(s) that reference it so that they continue to reference it, or (c) if the referenced record is deleted or its Primary Key is changed, change the reference(s) in any referencing record(s) to NULL.

Requirements advice: Consider a "nonfunctional requirement" that states that any relational database should be protected by database constraints to the extent feasible.

25.1.6.11. Program Interfaces

How people and programs interface to SQL:

Command line: You can type commands (called "queries" even when they're not really asking anything) at SQL databases. These "queries" are useful for creating the database and for producing ad hoc reports.

ODBC, JDBC, and manufacturer-specific: The programs communicate with the database via subroutine calls. This is the most common interface.

C*Plus, Cobol*Plus, and the like: A means used by Oracle and IBM in which SQL is embedded directly into the calling program, without subroutine call syntax, also called EXEC SQL. This can be a big time-saver, as it allows compile-time SQL checking.

Stored procedures and views: As mentioned above, move SQL logic to the database server whenever possible.

25.1.7. Compiled Languages versus Not

Some languages compile to native code. Some compile to byteodes (such as Java and .Net), and others are interpreted. Native code gives the fastest execution, but self-cleaning bytecoded languages are easier to write in. For most business applications, self-cleaning languages are the most cost-effective, but when you need the fastest possible execution, native code is the way to go.

25.1.8. Object Orientation

Object-oriented programming (OOP) is a style of programming in which classes of data are assigned behaviors, as opposed to pure procedural programming, in which things are done to data. Each can do everything the other does.

Inheritance: In object oriented programming, one class of objects can be built on another, inheriting its data and behaviors, but overriding some. This forms "class hierarchies", which are sometimes part of the requirements documentation. *Such class design (and even whether to use object orientation or to use pure procedural programming) belongs in the design documentation, and not in the requirements.*

25.1.9. Importance of Interfaces

Many things are called "interfaces"...

User interfaces: The user interface (UI) or GUI (Graphical User Interface) is the form the user interface takes. This might be a windowed interface, something entirely graphical, such as a game, the ol' 24x80 green screen, or something embedded, such as a steering wheel and pedals.

Internal program interfaces: Internal application program interfaces (APIs) are the interfaces between components of a system. If components of the system are to be built by two or more teams, then the entirety of the API had better be a part of the requirements document, but this part of the documentation is usually written by a system engineer, rather than a requirements gatherer or documentor.

Storage: Programs will read and write data to and from external storage, such as disk files or SQL data. If data is to be shared by components produced by two or more teams, then the entirety of the data structure had better be a part of the requirements document, but again, this part of the documentation is usually written by a system engineer, rather than a requirements gatherer or documentor.

Things called "interfaces" in object-oriented languages: In object-oriented languages, an "interface" is a definition of a set of methods (procedures and function calls) that a class promises to implement.

25.1.10. Importance of Sameness

This section is pure recommendation: You need to implement consistency in these areas:

User interfaces: Consistency of user interface is important to the learning curve of the user. Maintain consistency, except where you need to deviate. The best book on this, regardless of your platform, is Inside the Mac, Volume 1.

*Coding style: Consistent coding style makes the program easier to understand, and thus cheaper to test and maintain. Coding standards should be in place for a project, but should be at the level of suggestion, rather than requirement. Coding style guides that are absolute rules tend to be more frustration than they're worth. Sheldon's style suggestions: (a) "{" goes on the same line as the command it's part of, (b) don't put spaces around the operator with the highest precedence on the line (for instance "a + b*c"), and (c) never place more than one statement on a line.*

Language class: At any given tier (as in client, web server, or database), there should be one language in use, or at least all languages should be of the same class. For instance, using C# and Visual Basic.Net together works well. PL/1 and Cobol do. C# and C++ don't.

25.1.11. Frameworks and Inversion

Inversion is where we take control outside of a program, and move it to some outer, containing framework. *Sheldon's advice: A little of this, such as you find in redirection in Unix and data definition in JCL is helpful. Too much, and your program becomes unfollowable. Spring, Spring Boot, and .Net MVC can fall on either side of this line, depending on how they're used.*

25.1.12. Aspects

There is something called aspect-oriented programming, in which you divide classes into aspects. Each aspect does a different type of thing. *Sheldon's advice: 100% of the time, this is a waste of time and effort.*

25.2. Tiers

Programs are often divided up into two or more of the following tiers, which typically run on separate computers: Database, Business, Web service, Presentation, and Client.

25.3. Solution-Oriented Architecture (SOA)

Various servers handle different parts of the problem. *Sheldon's advice: This is a good idea if two separate companies are involved. This is a bad idea if not.*

25.4. EMail

Servers can send and receive eMail and texts. EMail becomes insecure the moment it leaves the building. EMail is secure if both users are on the same server. Requirements tie-in: Know what has to be secure and when. *When in doubt, require that it be secure.*

25.5. Web Languages

There are a number of languages used in constructing a web site:

HTML to hold the structure and content at the client (browser) tier.

CSS to hold the styling controls at the client tier.

JavaScript to control programmability and behavior at the client tier (and possibly add-ons, as discussed above).

JSON or XML (including the SOAP variant) to hold data as it moves between JavaScript control and the server tier.

Back-end languages (hundreds of them) to control what happens at the server tier and above.

25.6. Ways

There are a lot of ways to get things done: *(Sheldon's rule: Any time you want to do something, there are a thousand ways to do it wrong and a hundred ways to do it right.)*

25.6.1. SQL or Not

SQL and NoSQL are not the only ways to store data. Consider also indexed sequential (ISAM) files, random-access/direct access files, and sequential files.

Data that is tabular in nature (data which can easily be stored in tables) can be most efficiently stored and retrieved using either SQL-type storage (when queries happen ad hoc, or by keys or conditions other than the primary key), ISAM storage (when only the primary key is used to retrieve or filter records and the primary key is not both high-density and sequential), random access (when only the primary key is used to filter or retrieve records, and the keys are high-density and sequential, and you don't retrieve the whole file on each use), or in sequential files (when you retrieve the whole file in each use).

Data that is not tabular in nature, but which is document-oriented (and which has some sort of primary key) is best stored in a NoSQL or Document-Oriented database.

Data which is going to be stored in a tabular manner or in a document-oriented manner should be stored in a processing queue of some sort.

Large data that doesn't fit into any of these categories generally goes into a rather unstructured pool of data, currently called a "data lake".

You will find those who think that all data should be stored in SQL, and people who think that all data should be stored in NoSQL systems. Sheldon believes that both of these groups are wrong. Less specifically, Sheldon believes that anyone who says that everything should be done in any one manner is wrong.

25.6.2. OS

Operating systems can have a large influence on how things are done. OSes fall into a number of categories:

Unix/Linux: Unix is commercial; Linux is the freeware equivalent. Both run on everything. Properly written Unix/Linux programs are fully hardware portable.

Windows: The server version of Windows doesn't need a keyboard or screen. Windows is not portable. Serving more users costs more than serving fewer users. There's a lot of proprietary software here.

IBM: These are mainly mainframe OSes. They support big-iron solutions in which very powerful machines with very wide data paths are thrown at a problem. They include batch programs which will finish their task or restart themselves even after an unplanned reboot.

Embedded real-time operating systems (RTOS): Real-time operating systems can execute tasks at exactly the right time. Embedded systems are those that have no user interface that's obviously part of a computer.

Sheldon's advice: Generally, I don't recommend requiring any specific operating system. The two exceptions are (a) when using big-iron hardware such as an IBM z/Series or Oracle Exadata, you don't get the whole hardware boost without the OS that knows how to use the specialty hardware, and (b) when you need a real-time solution, call that fact out in the requirements. Real-time operating systems are not as general-purpose as most operating systems, but they react much faster.

25.6.3. Stacks

A "stack" in this context is a collection of programs (operating system, web server, web languages, and specific programs) that "sit on top of each other", such as the Windows stack (Windows, Internet Information Server, and ASPX or some other server web language, and Microsoft SQL Server), LAMP (Linux, the Apache web server, the MySQL database, and PHP), and Java web service (Linux/Unix/zOS, Apache, a database, Java, and likely Java Server Pages).

25.7. Security

Security falls on a spectrum. Some programs don't need any, like games. Other products need a lot, like military and banking. Know the required security level, and the probable threat level. In

the requirements document, document the visible security requirements in the function requirements, the invisible security requirements in the "nonfunctional requirements", and the threat levels in the assumptions area.

Outside: How secure your software is from all types of outside attacks. *Sheldon's benchmark: I did some work for a banking system comprising thousands of banks (not just branches). It took them eleven months to patch the holes I was able to get through.*

Inside: This is the level of security needed and used inside the building, from none through just as much as Outside security. *Ken Ellson's rule: Your security should be good enough that it will take the bad-guys 40 hours to get through. If they're going to spend 40 hours on this, they're just not going to stop. Sheldon's observation: Ken is correct. The longest it has taken me is 240 hours.*

Database: The database may need to be secured, such as internal permission levels and encrypted columns. For instance, passwords may be stored with one-way encryption, so that they can be compared encoded, but can never be decoded.

Code at the user tier: Once code is delivered to the user tier, it's in the user's control. Never trust code running at the user tier, such as the JavaScript you sent.

Penetration: Attacks in which there is an attempt at some sort of "penetration" of your system, or the insertion of something into it. The most common form of this is logging into control ports by guessing passwords, and "SQL Injection", in which parts of SQL commands are sent to data fields, hoping that your program doesn't protect itself from that.

25.8. Nice Ways In & Out

Consider these during the requirements-gathering phase:

Input validation: Whenever a field is entered, the program should reject invalid types of characters. For instance, in a numeric field, ignore letters. Always check every field for valid input. Checking during data entry is optimal, but not all checks can be run at each keystroke.

Crystal Reports™, Microsoft SSRS, PowerBI, Tableau, and the like: these let you set up a report format, and have that report filled from server data, using parameters. Best practice: Separate the report from the view or procedure that delivers the data. *Feel free to specify what goes into the report, including format, but best not to specify a particular reporting program.*

Kofax™ and Google Vision: These are scanning and character recognition systems which can scan documents, determine their types, do validation, and put data into the database. Kofax is a complete commercial off-the-shelf (COTS) solution, and Google Vision is a callable library.

If you find any of these (or similar) techniques or products useful, the use of any of these will likely affect the requirements document(s).

25.9. CPU Scaling

Scalable systems allow your system to continue to run once its usage grows.

The Microsoft Windows way: Sends your maintained data to the user's tier, encrypted, so that the user can't modify it. The user's system can reconnect to any of a number of servers, delivering it accurate knowledge of what the state and data is. Pro: Virtually unlimited PC scaling. Con: Extra data transmission. This is called "Horizontal scaling".

The PHP way: Machines are told which machines to reconnect to. Pro: You can connect a lot of cheap machines quickly. Con: It's hard for the systems to coordinate with each other, so the problem may have to be broken down. This is also Horizontal scaling.

The Google, Azure, and Amazon Web Services (AWS) way: Makes copies of the data to each server, so that no one machine needs to communicate with another for most circumstances. This is also Horizontal scaling.

SOA (also known as Solution-Oriented Architecture or Microservices): As discussed above.

The Unix, Linux, and Oracle way: You can write your application for small machines, and run it on the largest systems built. Pro: Cheap & easy at first. Any level of power may be applied to a problem, a little at a time, without any coding to account for scaling. Con: Requires replacement of machines. You don't necessarily take full advantage of any special hardware present. This is called Vertical scaling.

The IBM way: Choose a big-iron machine, and take advantage of all of its special features from the get-go. Pro: It's fast to get going, and there is no accounting for the possibility of scaling. Con: It's expensive, and you've got unused capacity at first.

The notion of scaling is important at requirements time. For instance, if the data and all possible processing power will fit in one cheap machine, then ignore any possibility of scaling. If the number of users, size of data, or complexity of processing can grow without limits, then the system must be built for scalability in the first place. Although it's not your job as the requirements writer to decide what kind of scaling will be used, it is your job to know and document what kind of scaling might be needed in the future. This can happen in the Assumptions section (for instance, an assumption that there will be few users to start, but that there will be billions later), and/or in the "nonfunctional requirements" (for instance, "Response times shall be within 500ms when fewer than 100,000,000 users are using the system simultaneously.").

25.10. Database Scaling

When all of your data fits in the same processor as is handling your web-site, there's nothing to worry about. If the workload gets too big, then it's time to separate the database from the web server. If you're running a database back-end to desktop applications, then you're already in this circumstance. At some point, the data or the processing load may get to the point where a single

x86-class machine can't handle the load any more. At this point, there are several options to grow the system.

One method to switch to a federated database. In a federated database system, there is one processor that acts as the database server, but it's not the whole server. It's mainly a facade for the rest of the database server. In a typical system, there may be one front-end server, with 16 processors behind it. Whatever work you give the federated server, that work is split. For instance, let's say you execute the SQL command "SELECT SUM(balance) FROM Account". That asks the system to look at every record in the Account table in the database, adds up the Balance column, and reports the result. This means that every record in the Account table has to be read. That can take a while in a regular database server. But in a federated server system, that Account table has somehow been split up. For instance, if we have constituent servers numbered 0 through 15, one way of splitting up the work might be to divide the account number by 16, and use the remainder of that division (called the modulus). For instance, for account #20, 20÷16 is 1 remainder 4. So the processor number we would use for that record is 4. All this happens behind your back. In this example, processor #1 would sum the records for accounts numbered 1, 17, 33, and so on. Processor #2 would sum the account records for accounts numbered 2, 18, 34, and so on. All 16 back-end processors would come up with 16 sums, and the front-end processor would add those 16 sums together, and return the result. Thus, the 17 processors can work almost 16 times as fast as a single processor for this type of work. Pro: This type of processor stays "consistent", in that there is only one copy of the data, so if I ask the total of all balances, and you ask the total of all balances, we're going to get the same answer, if there are no changes in the intervening time. Con: However, let's say that just before I ask for the sum of all account balances, someone else starts a transaction which will move money into or out of an account, locking or freezing one of the account records. The processor handling that account will hold my request until the locking transaction is complete, which will mean that one of the 16 sums to the main processor will be delayed, which will mean that my whole answer will be delayed. This type of set-up is ideal for when data consistency is of paramount importance. Although the example and explanation show an SQL database, there are equivalents for NoSQL databases.

The second method of getting faster results is to have fully replicated data. For instance, let's say that we have a big search engine site. We have thousands or millions of web crawlers (programs that search the web in some predictable manner), looking for web-site contents and links. They add to or update our database of everything that's out there on the web, in some sort of majorly indexed manner. We also have thousands or millions of search engines, doing searches on that database. Do we want millions of servers reading from a single database and millions of crawlers writing to this database all at once? Probably not. That would be quite a load, and even the fastest server is unlikely to be able to keep up. So, what's done is that there are Replicator servers, whose job it is to make copies of the indexed web database. Each of those search engines, or more likely, each gang of search engines shares a copy of the web database, showing the contents of the web as it is right now, or a few minutes ago, or a few hours ago. Not all of these copies will be made at once. Some copies will be newer than others. So, if I do a search for some trending topic, my search may run on a server with a 10-minute old copy of the internet, and when you do the same search at the same time, your search may run with a 5-minute old copy of the internet. Thus, you and I may get differing results, even though we ran

our queries at the exact same time. This is called "eventual consistency" and "replicated data". Pro: This type of system is very fast, because nobody waits for anything, except during a copy, and even then, if there are copies which are switched out, nobody waits at all. Con: The data has no guarantee of consistency. This type of set-up is ideal for mass searches of big data without the need for consistency over a short period of time.

There are hybrid databases (both SQL and NoSQL) in which there is apparently one copy of the database, but there are two or more copies behind the scenes, and there is a process (or processes) to merge the data. There is some replication, and synchronization processes are running all the time to keep that data as synchronized as is feasible, but until each synchronization completes, there are minor inconsistencies. A social media web-site will typically run this way. It may take a few seconds for my comments to show up in your feed. Pro: The system gets a good deal of speed, and a good deal of consistency. Con: It's not the fastest, and it's not 100% consistent all the time. These systems are also called "eventual consistency", because the data will sync up soon. Have you ever ordered something that the system says it has when it starts processing your order, and then when you get the delivery, there's a message that says "Sorry, we were out of stock"? That's the result of "eventual consistency" being used where full consistency should have been used.

So, why this great level of detail in this database engineering matter in a requirements engineering book? Because you need to know what the consistency requirements are for the system you're building. What are the ramifications to an inconsistency? If, in the case of a big search engine, speed is more important than consistency, go that way. If, in the case of a bank, consistency is more important than speed, go that way. If, in the case of social media, you want the best of both worlds, with being a bit out of sync is fine, go that way. But know what you need, and document that need.

25.11. In-house, Cloud, or Edge?

Unless you're building an embedded system, it doesn't much matter where the computer is, so it's not going to be a requirements issue whether the computer system is in-house, in the cloud, or "on the edge", meaning that parts are in-house and parts are in the cloud. There are a few exceptions:

- Embedded systems need to be embedded (of course).
- Security concerns may mean that data can't leave the premises.
- Real-time systems typically can't stand the lag to get to or from the internet.
- If the system is isolated or otherwise incommunicado, it needs to be self-sufficient.
- Management or a customer contract dictates one scheme or another.

25.12. How Things Can Go Wrong

Get one of these design decisions wrong, and you're likely to violate one of the edges of the engineering triangle, in that…

- Good: The project may not do what it needs to do (or may not do it well),
- Cheap: The project may cost more than it should to develop, use, and/or maintain; and/or
- Fast: The project may take too long to deliver and/or the project may take too long to \run.

26. Common Writing Mistakes to Avoid

Do things right by avoiding common mistakes:

- "B" is an abbreviation for bytes. "b" is an abbreviation for bits.
- Check your facts. If you're going to state something as a fact, it had better be a fact.
- Make sure your data and technology are not obsolete.
- Don't use any impossible requirements, such as 100% uptime.
- Don't assume knowledge on the part of your reader.
- If you promise something will appear later, make sure it does.
- Don't use JPEG for images with lines or thin text, because JPEG is lossy.
- Don't have blank lines or rows in a table.
- "Things", "thing's", and "things'" are multiple, single ownership, and multiple ownership.
- Check for run-on and fractional sentences.
- Check punctuation, capitalization, and spelling. Punctuation must follow spacing rules.
- Make sure there are no missing articles, such as "a", "an", or "the".
- "Login" and "setup" are events. "Log in" and "Set up" are actions.
- "Than" is a comparison. "Then" is a time.
- "Ran" and "run" are two different words. "Was ran" is always wrong.
- "Different" needs a "different than what". "Various" doesn't.
- "Prospective" is looking ahead. "Perspective" is just looking.
- "-" in a compound word or at the end of a line. "—" as a thought separator.
- "Until" and "'til" are times. "Till" is a cashbox.
- "Software", "firmware", "hardware", and "water" are indiscreet. "A water" is never right.

27. Job Search Terms

If we're going to teach you how to do the job, we might as well teach you how to get the job. The following, and their abbreviations, are typical terms you'll see for the people who do the requirements engineering job:

Analyst
Business Analyst (BA)
Business Process Analyst
Business Solution Analyst
Business Solutions Analyst
Business System Analyst
Business Systems Analyst
Functional Analyst
Requirements Engineer
Subject Matter Expert (SME)
Technical Analyst
Technical Business Analyst (Technical BA)
Technical Business System Analyst
Technical Business Systems Analyst.

Appendix A – Use Case Template

<Project Name>
Software Requirements Specification
For <Subsystem or Feature>

Version <1.0>

[Note: The following template is provided for use with the Rational Unified Process. Text enclosed in square brackets and displayed in blue italics is included to provide guidance to the author and should be deleted before publishing the document. A paragraph entered following this style will automatically be set to normal.]

*[Note: The Software Requirements Specification (SRS) captures the complete software requirements for the system, or a portion of the system. Following is a typical SRS outline for a project **using use-case modeling**. This artifact consists of a package containing use cases of the use-case model and applicable Supplementary Specifications and other supporting information. For a template of an SRS **not** using use-case modeling, which captures all requirements in a single document, with applicable sections inserted from the Supplementary Specifications (which would no longer be needed), see rup_srs.dot.]*

Many different arrangements of an SRS are possible. Refer to IEEE-93 for further elaboration of these explanations, as well as other options for SRS organization.]

<Page Break>

Revision History

Date	Version	Description	Author
<dd/mmm/yy>	<x.x>	<details>	<name>

<Page Break>

Table of Contents

<Table of Contents>

<Page Break>

1. Introduction

[The introduction of the SRS should provide an overview of the entire SRS. It should include the purpose, scope, definitions, acronyms, abbreviations, references and overview of the SRS.]

1.1. Purpose

[Specify the purpose of this SRS. The SRS should fully describe the external behavior of the application or subsystem identified. It also describes nonfunctional requirements, design constraints and other factors necessary to provide a complete and comprehensive description of the requirements for the software.]

1.2. Scope

[A brief description of the software application that the SRS applies to; the feature or other subsystem grouping; what Use Case model(s) it is associated with, and anything else that is affected or influenced by this document.]

1.3. Definitions, Acronyms and Abbreviations

[This subsection should provide the definitions of all terms, acronyms, and abbreviations required to interpret properly the SRS. This information may be provided by reference to the project Glossary.]

1.4. References

[This subsection should provide a complete list of all documents referenced elsewhere in the SRS. Each document should be identified by title, report number (if applicable), date, and publishing organization. Specify the sources from which the references can be obtained. This information may be provided by reference to an appendix or to another document.]

1.5. Overview

[This subsection should describe what the rest of the SRS contains and explain how the SRS is organized.]

2. Overall Description

[This section of the SRS should describe the general factors that affect the product and its requirements. This section does not state specific requirements. Instead, it provides a background for those requirements, which are defined in detail in section 3, and makes them easier to understand. Include such items as product perspective, product functions, user characteristics, constraints, assumptions and dependencies, and requirements subsets.]

2.1. Use-Case Model Survey

[If using use-case modeling, this section contains an overview of the use-case model or the subset of the use-case model that is applicable for this subsystem or feature. This includes a list of names and brief descriptions of all use cases and actors, along with applicable diagrams and relationships. Refer to the use-case model survey report, which may be used as an enclosure at this point.]

2.2. Assumptions and Dependencies

[This section describes any key technical feasibility, subsystem or component availability, or other project related assumptions on which the viability of the software described by this SRS may be based.]

3. Specific Requirements

[This section of the SRS should contain all the software requirements to a level of detail sufficient to enable designers to design a system to satisfy those requirements, and testers to test that the system satisfies those requirements. When using use-case modeling, these requirements are captured in the use cases and the applicable supplementary specifications. If use-case modeling is not used, the outline for supplementary specifications may be inserted directly into this section.]

3.1. Use-Case Reports

[In use-case modeling, the use cases often define the majority of the functional requirements of the system, along with some non-functional requirements. For each use case in the above use-case model, or subset thereof, refer to or enclose the use-case report in this section. Make sure that each requirement is clearly labeled.]

Use Case <N>: <Use Case Name>

Summary: *[Give a brief synopsis of the overall purpose and flow of this functionality. Often, a an implementation priority is given.]*

Preconditions: *[List the preconditions, if any, that must be true for this scenario to start, either as a single item, or as a bulleted list.]*

Triggers: *[List the trigger(s) for this function, typically as a single item, but possibly as a bulleted list.]*

Basic course of events (main scenario):

Actor	System	Screen
[Numbered steps here, starting with the triggered as step 1, one step per table row. You'll almost always be using multiple rows.]	*[Numbered steps here, typically starting with step 2, one step per row, but often with the response to the Actor's action on the same row.]*	*[A picture, drawing, or description of the screen contents for this step, assuming that there is a screen for this step, and assuming that we're on step 1, or that the screen contents have changed for this step. If the picture is too large to fit here, move it to an appendix, and just have a reference here.]*

Alternate courses of events (alternate scenarios)

Actor	System	Screen
[Same rules as above, but omit any steps here that are the same as the main scenario.]	*[Same rules as above, but omit any steps here that are the same as the main scenario.]*	*[Same rules as above, but omit any screens here that are the same as the main scenario.]*

Post-conditions: *[List the post-conditions, if any, that will have occurred to permanent state, or to session state, either as a single item, or as a bulleted list.]*

3.2. Supplementary Requirements

[Supplementary Specifications capture requirements that are not included in the use cases. The specific requirements from the Supplementary Specifications which are applicable to this subsystem or feature should be included here, refined to the necessary level of detail to describe this subsystem or feature. These may be captured directly in this document or refer to separate Supplementary Specifications, which may be used as an enclosure at this point. Make sure that each requirement is clearly labeled.]

4. Supporting Information

[The supporting information makes the SRS easier to use. It includes: a) Table of contents, b) Index, c) Appendices. These may include use-case storyboards or user-interface prototypes. When appendices are included, the SRS should explicitly state whether or not the appendices are to be considered part of the requirements.]

Appendix B — Sample Use Case Document
(only 2 use cases shown out of the 23 in the original)

Photo Software
Software Requirements Specification

Version 1.0

<Page Break>

Revision History

Date	Version	Description	Author
04/05/2006	1.0	Initial document	Michael Brown
05/06/2019	1.0.1	Slight modernization	Sheldon Linker
05/10/2021	1.1	Modern best practice: No path to an error case	Sheldon Linker

<Page Break>

Table of Contents

<Table of Contents>

<Page Break>

1. Introduction

Mitropoulos, Inc. is entering the highly completive digital camera market. Although this market is saturated with cheap products, Mitropoulos Inc. has located a third-world country that has not been tainted with labor laws, human rights and social programs. With this advantage Mitropoulos, Inc. can corner the market of low-end digital cameras. To help accomplish this goal, Mitropoulos, Inc. needs photo album editing software to accompany the cameras.

Mitropoulos, Inc. has contracted with Brown LLC to create photo album editing software. Brown LLC specializes in quality software with small feature sets. They also have an excellent track record for producing software on schedule, which is important to Mitropoulos, Inc.

This document is the Software Requirement Specification (SRS) for the photo album editing software. It will outline all of the agreed upon features of the software.

1.1. Purpose

The purpose of the SRS is to provide a clear, documented model of the requirements for the system. This will be used by Brown LLC to construct the software and provide it by summer 2006.

The software will implement many Use Cases, which are shown in Section 3. The Use Case inventory is as follows:

1. Create Album
2. Delete Album

1.2. Scope

The software to be created is photo album editing software. It will perform basic functions that will allow the actor to create multiple photo albums. Within a photo album they will be able to insert photos. The software also supports sub-albums, which can be used like chapters within the software.

1.3. Definitions, Acronyms and Abbreviations (Glossary)

There are a variety of terms used in this SRS relating to the software being created. Most of the terms are self-explanatory and are common to physical photo albums. However, for completeness, all terms related to the software are provided.

- Album – an album is a selection of content at the highest level
- Content – content can be a photo or a sub-album
- Photo – a digital picture in JPEG or GIF format
- Slideshow – the automatic presentation of photos
- Sub-album – a collection of content within a album or sub-album

1.4. Risk Analysis

There are a variety of risks to developing this software. As mentioned before, it is important that the software be finished by the summer of 2006. This is the key window to allow Mitropoulos, Inc. to prepare for holiday shopping season. To address this risk, Brown LLC is attempting to finish the software by mid-summer. This will provide extra time if obstacles slow down the project.

A secondary risk to this software project is changing requirements. Mitropoulos, Inc. could post clarifications to the requirements that might adversely affect this project. Brown LLC will attempt to create a very flexible software design that will allow changes to the software quickly and easily.

1.5. Overview

The following SRS is organized into two major sections: Overall Description and Specific Requirements. The Overall Description describes the requirements at a high level, while the Specific Requirements describe all of the relevant requirements of the system.

2. Overall Description

The photo editing software allows the actor to create virtual photo albums. The primary features are to allow the actor to create logical photo albums in the system. Each photo album can contain content, which consists of actual photos or sub-albums. A sub-album is a logical collection of content. Sub-albums can also contain content, which is photos or other sub-albums.

A second set of features allows the actor to change the photo albums. Order of content and albums is important in the photo album editing software. A set of features allows the actor to move albums or content up or down. Additional features allow the actor to copy, cut and paste photos. There is also a search feature that will allow the actor to search for photos bases on key words. Actors can change many characteristics of the pictures like contrast and brightness.

The final set of features allows the actor to create slideshows. A slideshow is an automated process of showing the photos. The actor can select background music that can be played. There are two types of slideshows. One type shows the pictures only once. A second type will continuously show the pictures until the actor terminates the feature.

2.1. Use-Case Model Survey

There are 23 Use Cases that make up the requirements for this system. The detailed Use Cases are in section 3.

Use Case	**Description**
1. Create Album	Creates an album in the photo album library.
2. Delete Album	Deletes an album in the photo album library.

The screenshots in the Use Cases are there to present a visual representation of the screen will look like. Red circles are used to direct the audience's attentions to different areas of the screen. Do not assume that the actual content on the screens corresponds to the path of the Use Case.

2.2. System Evolution

If Mitropoulos, Inc. is successful in carving out a spot in the digital camera market; enhancements to this software will be needed. Such enhancements could include having the software interact directly with the digital camera and moving photos from the software to the digital camera. This would only take place if Mitropoulos Inc. decides to compete in the high-end digital camera market.

If Mitropoulos, Inc. sees moderate success in the low-end camera market; they will maximize profits by cuttings costs. No new features will be added to the software. Only bug fixes will be performed.

3. Specific Requirements

The Specific Requirements section will provide the Use Case Reports specifying the 23 Use Cases that make up this system.

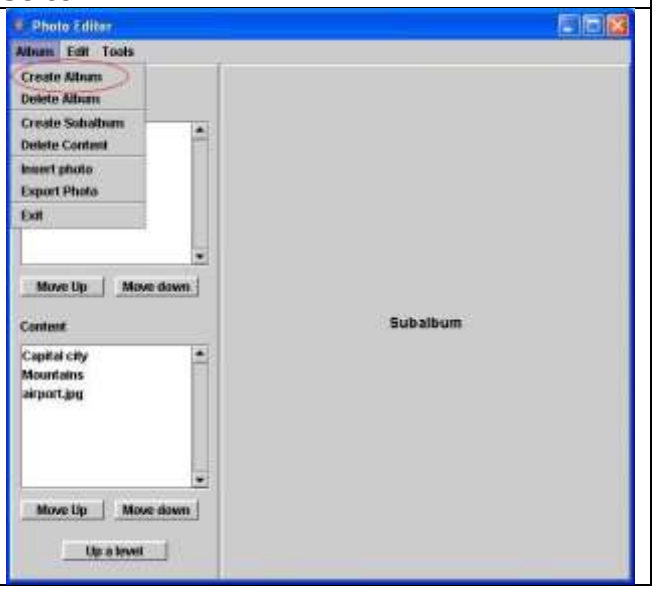
3.1. Use-Case Reports

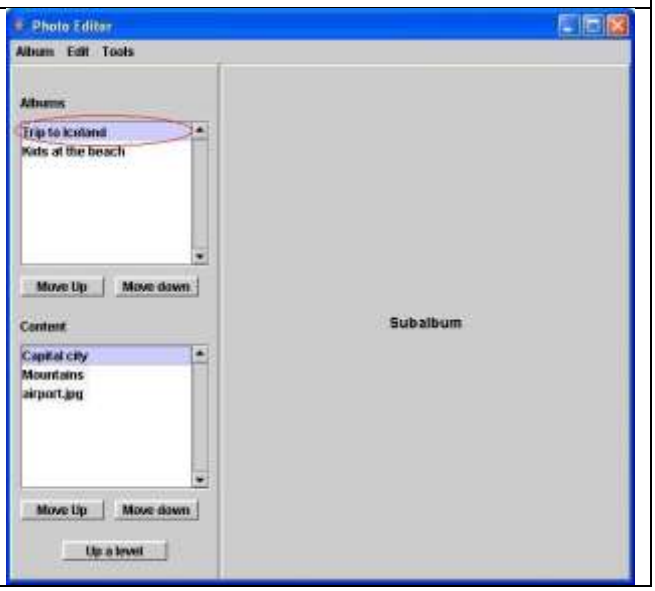
3.1.1. Use Case Name: Create Album

Summary: The Create Album feature allows the actor of the software to create a new album in the album library.

Triggers: The actor selects the Create Album menu item.

Basic course of events (Scenario):

Actor	System	Screen
1. The user selects Create Album.		

Actor	System	Screen
	2. The system requests the name, description, and date of the new album.	
3. The user completes fields and selects OK.		
	4. The system adds that album to the album area.	

Alternative paths:
A. The album created is canceled.

Actor	System	Screen
3. The user selects Cancel.		
	4. The system removes the create album screen.	

B. Duplicate album name.

Actor	System	Screen
	4. The system determines that the new album name already exists and produces error message.	
5. The user acknowledges the error message.		

Post conditions:
1. A new album has been created in the album area.
2. The new album is selected.

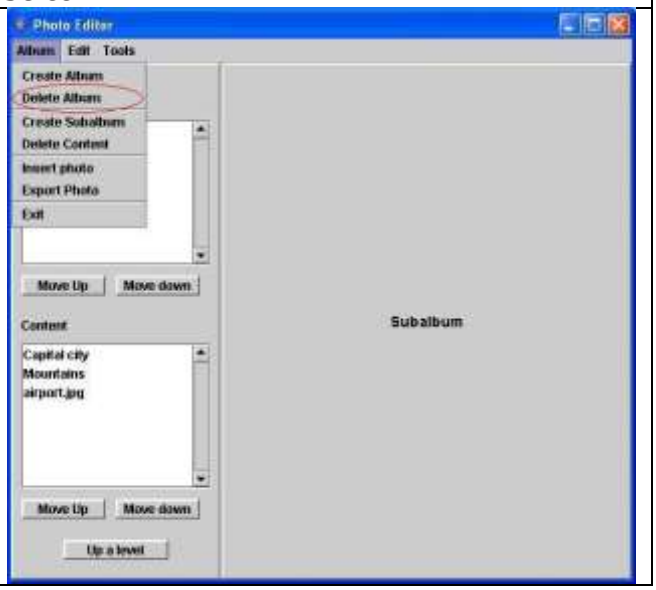
3.1.2. Use Case Name: Delete Album

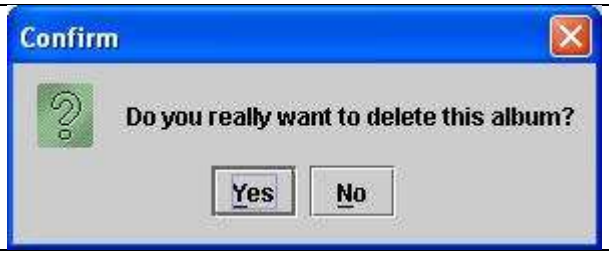
Summary: The Delete Album feature will remove an album from the system.

Preconditions: An album is selected.
The album is empty.

Triggers: Actor selects the Delete Album menu item.

Basic course of events (Scenario):

Actor	System	Screen
1. The user initiates deletion of an album.		

Actor	System	Screen
	2. The system confirms that delete of selected album.	
3. The user confirms.		
	4. The system removes the selected album.	

Alternative path:

Actor does not confirm deletion.

Actor	System	Screen
3. The user rejects the album deletion.		
	4. The system removes the confirmation alert.	

Post Conditions: After successful deletion, no album is selected.

3.2. Supplementary (Non-functional) Requirements

- The software shall run on the latest versions of the Windows operating system, including Windows 7 through 10.
- Any Menu item or button whose functionality is precluded because the preconditions to the function are not met must be disabled when the condition is not met, and re-enabled when the conditions are again met.

Appendix C — IEEE 830 Template

Software Requirements Specification for <Project Name>

Version <# matching the last line of the versions table>

Prepared by <author>

<organization>

<date created>

<Page Break>

Table of Contents

<Table of Contents>

Revision History

Name	Date	Reason for Changes	Version
<Your name>	<Today>	<"Initial version" or actual reason>	<Version>

<Page break>

1. Introduction

1.1. Purpose

<Identify the product whose software requirements and design are specified in this document, including the revision or release number. Describe the scope of the product that is covered by this specification.>

1.2. Document Conventions

<Describe any standards or typographical conventions that were followed when writing this specification, such as fonts or highlighting that have special significance. For example, state whether priorities for higher-level requirements are assumed to be inherited by detailed requirements, or whether every requirement statement is to have its own priority.>

1.3. Intended Audience and Reading Suggestions

<Describe the different types of reader that the document is intended for, such as developers, project managers, marketing staff, users, testers, and documentation writers. Describe what the rest of this document contains and how it is organized. Suggest a sequence for reading the document, beginning with the overview sections and proceeding through the sections that are most pertinent to each reader type.>

1.4. Product Scope

<Provide a short description of the software being specified and its purpose, including relevant benefits, objectives, and goals. Relate the software to corporate goals or business strategies when applicable.>

1.5. References

<List any other documents or Web addresses to which this document refers or was based on. These may include user interface style guides, contracts, standards, system requirements specifications, use case documents, or a vision and scope document. Provide enough information so that the reader could access a copy of each reference, including title, author, version number, date, and source or location. Include the assignment document in the references.>

2. Overall Description

2.1. Product Perspective

<Describe the context and origin of the product being specified in this specification. For example, state whether this product is a follow-on member of a product family, a replacement for certain existing systems, or a new, self-contained product.>

2.2. Product Features

<Summarize the major features the product must perform or must let the user perform. Details will be provided in Section 3, so only a bullet list is needed here. Organize the features to make them understandable to any reader of the specification. A picture of the major groups of related requirements and how they relate, such as a top level data flow diagram or object class diagram, is often effective.>

- <Thing 1>
- <More things as required>

2.3. User Classes and Characteristics

<Identify the various user classes (intended target for the product) that you anticipate will use this product. User classes may be differentiated based on frequency of use, subset of product functions used, technical expertise, security or privilege levels, educational level, or experience. Describe the pertinent characteristics of each user class. Certain requirements may pertain only to certain user classes. Distinguish the most important user classes for this product from those who are less important to satisfy.>

2.4. Operating Environment

<Describe the environment in which the software will operate, including the hardware platform, operating system and versions, and any other software components or applications with which it must peacefully coexist.>

2.5. Design and Implementation Constraints

<Describe any items or issues that will limit the options available to the developers. These might include: corporate or regulatory policies; hardware limitations (timing requirements, memory requirements); interfaces to other applications; specific technologies, tools, and databases to be used; parallel operations; language requirements; communications protocols; security considerations; design conventions or programming standards (for example, if the customer's organization will be responsible for maintaining the delivered software).>

2.6. User Documentation

<List the user documentation components (such as user manuals, on-line help, and tutorials) that will be delivered along with the software. Identify any known user documentation delivery formats or standards.>

2.7. Assumptions and Dependencies

<List any assumed factors (as opposed to known facts) that could affect the requirements stated in the specification. These could include third-party or commercial components that you plan to use, issues around the development or operating environment, or constraints. The project could be affected if these assumptions are incorrect, are not shared, or change. Also identify any dependencies the project has on external factors, such as software components that you intend to reuse from another project, unless they are already documented elsewhere (for example, in the vision and scope document or the project plan).>

3. System Features

<The purpose here is to give a brief introduction to the system features, so that §4 has something to reference. Then, the full explanation of the system features is given in §5, likely referencing §4. The numbering in §3 and §5 should match.>

3.1. <Feature Name>

<Brief description of the feature, generally 1-3 sentences>

3.<N>. <Additional feature blocks as needed>

4. External Interface Requirements

4.1. User Interfaces Overview

<Describe the high level functionality of the system from the user's perspective. Explain how the user should be able to use your system to complete all the expected features and the feedback information that will be displayed for the user (for each screen the user will see). For example, specify whether it is GUI or text based interface and how at high level it would work. Discuss at high level how user will interact with the game such as clicking buttons, typing in some selection character, etc.>

<Describe the minimum requirements for the interface. This may include some sample screen images whether for text or GUI approach, any GUI standards or product family style guides that are to be followed, screen layout constraints, for GUI standard buttons and functions (e.g., help) that must appear on every screen, keyboard shortcuts, error message display standards, and so on>

4.2. Hardware Interfaces

<Describe the logical and physical characteristics of each interface between the software product and the hardware components of the system. This may include the supported device types, the nature of the data and control interactions between the software and the hardware, and communication protocols to be used.>

4.3. Software Interfaces

<Describe the connections between this product and other specific software components (name and version), including databases, interpreters, operating systems, tools, libraries, and integrated commercial components. Identify the data items or messages coming into the system and going out and describe the purpose of each. Describe the services needed and the nature of communications. Identify data that will be shared across software components. If the data sharing mechanism must be implemented in a specific way (for example, use of a global data area in a multitasking operating system), specify this as an implementation constraint.>

5. System Features/Modules

<This template illustrates organizing the functional requirements for the product by system features, the major services provided by the product.>

5.1. <System Feature Name>

5.1.1 Description and Priority

<Provide a short description of the feature. Give its priority, too.>

5.1.2 Stimulus/Response Sequences

<"Stimulus" or "Condition">: <Describe the stimulus or condition>
Response:<Describe the response>

<Additional Stimulus/Response pairs as needed>

5.1.3 Functional Requirements

<Itemize the detailed functional requirements associated with this feature. These are the software capabilities that must be present in order for the user to carry out the services provided by the feature. Include how the product should respond to anticipated error conditions or invalid inputs. Requirements should be concise, complete, unambiguous, verifiable, and necessary. Each requirement should be uniquely identified with a sequence number or a meaningful tag of some kind.>

REQ-1.1: <Some sentence goes here, in "shall" form, usually something like this: "Upon «some event ot condition», the «system, or the name of a module» shall «perform some action»." Note that the entire requirement must be stated in the requirements area. Everything else is just a lead-up to this. There should be one such entry per S/R pair.>

REQ-1.<N>: <Additional S/R-related requirements as needed.>

REQ-1.<N>: <List any additional requirements needed in "Shall" form, but not necessarily in "Upon/Shall" form.>

REQ-1.<N>: <Additional requirements as needed.>

5.<N>. <Additional feature blocks>

6. Nonfunctional Requirements

6.1. Performance

NF-1.1: <Any performance-related issue>

NF-1.<N>: <Any performance-related issue>

6.2. Security

NF-2.1: <Any security-related issue>

NF-2.<N>: <Any security-related issue>

6.<N>. <Any additional "nonfunctional" blocks>

Appendix D — IEEE 830 Sample
(only 2 functional areas shown out of the 23 in the original)

Software Requirements Specification
for
Photo Software

Version 2.0

Prepared by Michael Brown & Sheldon Linker

UMGC

31-August-2021

<Page Break>

Table of Contents

<Table of Contents>

Revision History

Name	Date	Reason for Changes	Version
Michael Brown	5-Apr-2021	Initial document	1.0
Sheldon Linker	6-May-2019	Slight modernization	1.0.1
Sheldon Linker	10-May-2021	Modern best practice: No path to an error case	1.1
Sheldon Linker	31-Aug-2021	Conversion to IEEE-830 format	2.0

<Page break>

1. Introduction

1.1. Purpose

Mitropoulos, Inc. is entering the highly completive digital camera market. Although this market is saturated with cheap products, Mitropoulos Inc. has located a third-world country that has not been tainted with labor laws, human rights and social programs. With this advantage Mitropoulos, Inc. can corner the market of low-end digital cameras. To help accomplish this goal, Mitropoulos, Inc. needs photo album editing software to accompany the cameras.

Mitropoulos, Inc. has contracted with Brown LLC to create photo album editing software. Brown LLC specializes in quality software with small feature sets. They also have an excellent track record for producing software on schedule, which is important to Mitropoulos, Inc.

This document is the Software Requirement Specification (SRS) for the photo album editing software. It will outline all of the agreed upon features of the software.

1.2. Document Conventions

There are no special typographical conventions. The sample images are shown in the Windows 8 user interface format. The look of the application will likely appear stylistically different on Windows 10 or other operating systems.

1.3. Intended Audience and Reading Suggestions

This document is intended for all stakeholders in the project — Those who will authorize it, those who will develop it, and those who will test it. An of course, the real audience is the set of students who will read this appendix.

1.4. Product Scope

The software to be created is photo album editing software. It will perform basic functions that will allow the actor to create multiple photo albums. Within a photo album they will be able to insert photos. The software also supports sub-albums, which can be used like chapters within the software.

1.5. References

Normally, I'd reference the initiating document in APA v7 format here, but the eMail asking me to participate in this book was not meant for distribution.

1.6. Definitions, Acronyms and Abbreviations (Glossary)

There are a variety of terms used in this SRS relating to the software being created. Most of the terms are self-explanatory and are common to physical photo albums. However, for completeness, all terms related to the software are provided.

Album – an album is a selection of content at the highest level
Content – content can be a photo or a sub-album
Photo – a digital picture in JPEG or GIF format
Slideshow – the automatic presentation of photos
Sub-album – a collection of content within a album or sub-album

2. Overall Description

2.1. Product Perspective

This is new photo editing software for internal use.

2.2. Product Features

The photo editing software allows the actor to create virtual photo albums. The primary features are to allow the actor to create logical photo albums in the system. Each photo album can contain content, which consists of actual photos or sub-albums. A sub-album is a logical collection of content. Sub-albums can also contain content, which is photos or other sub-albums.

A second set of features allows the actor to change the photo albums. Order of content and albums is important in the photo album editing software. A set of features allows the actor to move albums or content up or down. Additional features allow the actor to copy, cut and paste photos. There is also a search feature that will allow the actor to search for photos bases on key words. Actors can change many characteristics of the pictures like contrast and brightness.

The final set of features allows the actor to create slideshows. A slideshow is an automated process of showing the photos. The actor can select background music that can be played. There are two types of slideshows. One type shows the pictures only once. A second type will continuously show the pictures until the actor terminates the feature.

- Create Album — Creates an album in the photo album library
- Delete Album — Deletes an album in the photo album library.

2.3. User Classes and Characteristics

For this application, there is only one user class, and that is User. There are no privilege levels.

2.4. Operating Environment

OE-1: The software shall run on Windows, MacOS, and iOS, at least.

OE-2: The application shall present the native interface. (This document shows a Windows interface.)

2.5. Design and Implementation Constraints

DC-1: The system shall run either in native code, or in a built-in VM, such as .Net on Windows.

DC-2: The system shall support all commonly used image formats, including, at least, all the image formats which the Chrome browser can display.

2.6. User Documentation

Although the plan is that the application should be simple and straightforward enough that the user should need no documentation, there shall be a printable PDF distributed with the application explaining its features and usage.

2.7. Assumptions and Dependencies

AS-1: We assume that the user has knowledge of how standard PC features work, such as the mouse, keyboard, and standard Open and Save dialogs work.

DE-1: Since this software contains no device drivers of its own, it thus depends on image-capturing hardware having its own means of either getting the images onto the host system, or as acting as a file system connectable to the host system.

3. System Features

The basic functional areas of the system are as follow: *(Note that since this is a sample, only 2 are shown.)*

3.1. Create Album

An album is the top level of the storage hierarchy. This function allows the user to create a new album, which can then contain images.

3.2. Delete Album

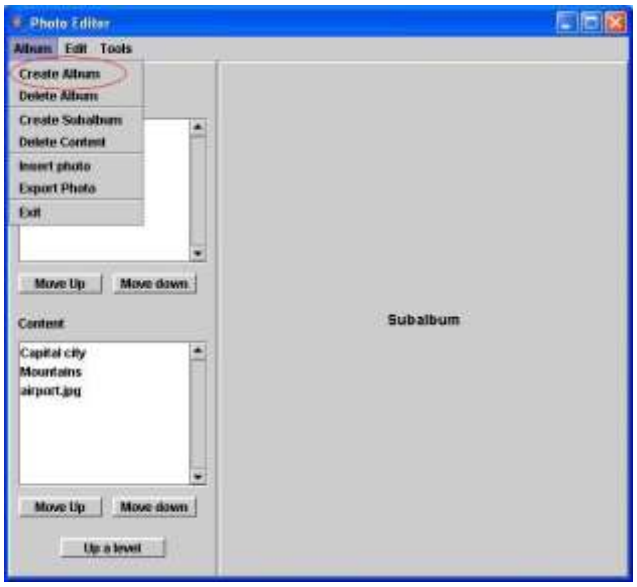
This function allows the deletion of an album, along with any images attached to it. Deletion of a non-empty album will involve a warning dialog.

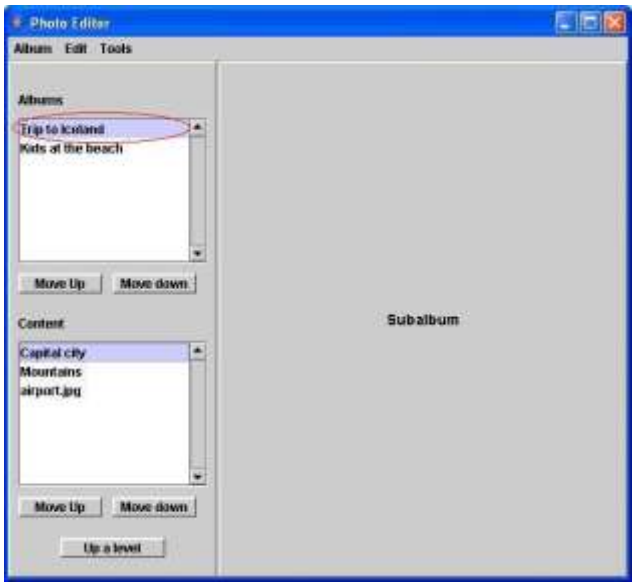
4. External Interface Requirements

4.1. User Interfaces Overview

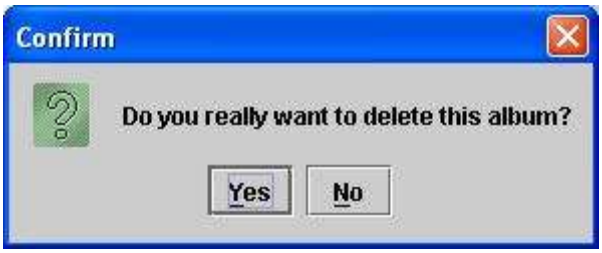
The various functionalities below will each have their own interface screen designs, as shown below. *(Note that only the screens for the first 2 examples are shown in this sample.)*

These image show the menus, plus the main lay-out area, and the lay-out area without the menu, but with contents:





This image shows the album creating dialog, a typical error dialog, and a confirmation dialog:



4.2. Hardware Interfaces

N/A — There need not be any special hardware interfaces in the program, and the program may rely solely on the host operating system's ability to load images from cameras and the like, or to show such devices as file systems.

4.3. Software Interfaces

N/A — This software does not directly communicate with other software.

5. System Features/Modules

The following provides a more in-depth explanation of the functional areas, including full requirements. *Note that only 2 areas are shown here for purposes of example.*

5.1. Create Album

5.1.1 Description and Priority

The primary focus of this product is to hold, organize, and contain imagery. Albums are the primary means on organizing the images. A user may create an album at any time, and there is no limit on the number of albums. Album names must be unique. Priority=High.

5.1.2 Stimulus/Response Sequences

Stimulus: From the **Album** menu, the user selects **Create Album**.
Response: The system displays the album creation dialog, as shown in §4. In the dialog, the **Ok** button starts in a disabled state.

Stimulus: The user enters, modifies, or deletes data in either of the top two text entry areas (name and date).
Response: At each operation, the fields are evaluated for validity, and the Ok button will enable or disable so as to be enabled when both fields contain valid data.

Stimulus: The user clicks Ok.
Response: If the name given is unique, then the name is added to the list of albums (and has no content). If the name is not unique, then the error alert as shown in §4 will display in a modal manner.

5.1.3 Functional Requirements

REQ-1.1: Upon the user choosing "Create Album" from the "Album" menu, the system shall display the album creation dialog as shown as "Create Album" in §4.1. The initial state of the "Ok" button shall be disabled.

REQ-1.2: As the user enters, modifies, or deletes data in the Name and/or Date fields, the validity of those two fields shall be reevaluated, and the "Ok" button shall be enabled if and only if those two fields are valid.

REQ-1.3: Upon the user clicking "Ok", or when using operating systems that link the "Return" or "Enter" button the the "Ok" button, pressing either of those keys, the system shall evaluate whether the album name is unique, and if so, shall add the ablum name, at the bottom, to the list of albums, in a persistent manner. If name is not unique, then the system shall display the "Duplicate Name" error alert, as depicted in §4.1.

5.2. Delete Album

5.2.1 Description and Priority

Anything created must be deletable. This function allows for the deletion of an album created above. Priority=Low.

5.2.2 Stimulus/Response Sequences

Stimulus: The user clicks on an album in the album list.
Response: The system will enable the **Delete Album** menu item if the album is empty, and disable the item otherwise.

Stimulus: From the **Album** menu, the user chooses **Delete Album**.
Response: The systems posts the confirmation dialog as shown in §4.

Stimulus: The user clicks Ok.
Response: The album is deleted, and removed from the list.

5.1.3 Functional Requirements

REQ-2.1: Upon each user action that changes the state of which album is selected (including starting the program), and upon each user action that changes the number of images in an album, the system shall revaluate whether the "Album" menu's "Delete Album" menu item should be enabled, and shall enable that item if and only if there is exactly one album selected in the album list and that album is empty.

REQ-2.2: Upon the user selecting the "Delete Album" menu item, the system shall post the "Confirm" dialog, as depicted in §4.1.

REQ-2.3: Upon the user clicking "Ok", the system shall remove the album from the list, in a persistent manner.

6. Nonfunctional Requirements

6.1. Performance

NF-1.1: Any operation that is expected to take 0.1s or less shall not post any sort of wait indicator.

NF-1.2: Any operation that is expected to take more than 0.1s, but less than 1s, shall post the system Wait cursor.

NF-1.3: Any operation that is expected to take 1s or more shall post a progress bar, count-down timer, or other system-appropriate wait/progress indicator.

6.2. Veracity

NF-2.1: The original image format shall be preserved.

Appendix E — Lightswitch Example for IEEE 830 Format

5.1. Lightswitch

5.1.1. Narrative

This function allows the user to turn a light on and off using a switch.

5.1.2. Stimulus/Response pairs

Stimulus: The user flips the lightswitch to the Up position.
Response: The light illuminates.

Stimulus: The user flips the lightswitch to the Down position.
Response: The light goes out.

5.1.3. Functional Requirements

REQ-1.1 Upon the user flipping the lightswitch to the Up position, the light shall illuminate.

REQ-1.2 Upon the user flipping the lightswitch to the Down position, the light shall go out.

REQ-1.3 The light shall run on 115VAC, and is permitted to draw up to 50W of power.

REQ-1.4 The light socket shall permit rapid replacement of the bulb, without the need for tools.

Note that in the functional requirements, the first 2 requirements match the two stimulus/response pairs. There is a 1:1 correspondence between the stimulus/response pairs. We did not say "the system allows the user to control the light.". Instead, we described how the system allows the user to control the light. Following those requirements, we added some additional requirements. Not all such sections will have such additional requirements, but some may.