A Brief Guide to the System Development Lifecycle

David Vernon

Informatics Research Centre
University of Skövde
Sweden

Revision 1.4
November 2013

Table of Contents

1. Introduction

The short document discusses the main and usually distinct phases involved in any project concerned with development of a software-based system. It is intended to complement the advice provided in the *Brief Guide to Research Methods* although there is some inevitable duplication of the material that appears in that document.

2. The Software Development Lifecycle

Many students have difficulty at the outset of their first major project. This is understandable: it is the first time they will have had to tackle a large amount of work that is often quite poorly defined. To get started, it helps to know the key activities that result in a successful project. They are:

1. Problem identification
2. Requirements elicitation
3. Problem modelling
4. System analysis and specification
5. System design
6. Module implementation and system integration
7. System test and evaluation
8. Documentation

3 Problem Identification

Problem Identification involves a lot of background work in the general area of the problem. Normally it calls for the use of prior experience, typically experience you may not yet have. It requires an ability to look at a domain (e.g. telecommunications or engine control) and to identify the issue that needs to be addressed and the problem to be solved (*e.g.* elimination of noise or cross-talk on a communication channel, or engine control for temperature-dependent fuel efficiency). It also required an understanding of the theoretical issues by which we can model the problem. So, the first thing you need to do in your project is become an expert in the problem at hand: a *problem-domain* expert.

At the same time, you also need to know how to handle the tools that will enable you to solve the problem. These might include the operating system, the programming language, the application programming interface (API) definitions, class libraries, toolkits, or any application-specific analysis utilities. That is, you also need to become a *solution-domain* expert.

The only way to become an expert in both the problem domain and the solution domain is to learn as much as possible about the area and to learn it as quickly and efficiently as

possible. Many people come unstuck at this first step and they launch themselves into a frenzy of unstructured research, reading much but learning little. Here are some tips to avoid this happening.

- Collect any papers, articles, book chapters you can on the area and make a copy for your own personal archive.

- Make sure you keep a full citation index, *i.e.,* you must record exactly where every article you copy comes from. Typically, you need to record the title of the article, the authors, the name of the magazine/journal/book, the volume and number of the journal or magazine, and the page numbers. If it's a chapter in a book and the author of the chapter is different from the editor of the book, you need to record both sets of names.

- Not all the articles you collect will be equally relevant or important. Consequently, it's not efficient to give each of them the same attention. But it's not easy to know how relevant it is until you read it. So how do you proceed? To solve this dilemma, you should know that there are three levels of reading:

    1. *Shallow Reading*: you just read the article quickly to get an impression of the general idea. Read it twice. This should take a half-an-hour to an hour.

    2. *Moderate Reading*: Read the article in detail and understand all of the main concepts; this will probably require you to read it several times and take a couple of hours to assimilate.

    3. *Deep Reading*: Here you make an in-depth study of the article. This may take you several hours or even a couple of days. After many careful readings, you should know as much about the topic as the author.

    The way to proceed with your 'reading in' is to

    - Shallow read everything and write a 5-line summary of the article

    - If you think the article is directly relevant to your project, label it, and put it on a list of articles to be read at the next level, i.e. Moderate Reading.

    - Read all the labelled articles and write a 1-page summary.

    - If the article is going to be used directly in the project, e.g. as a basis for a hardware design or a software algorithm, then you need to add it to your list of articles to be

read at the next level, i.e. Deep Reading.

- Read all the Deep-Read articles and write extensive notes on them.

It is very important that you realize that, in order to fully understand anything that you read, you must write it up in your own words. If you can't express or speak about a given idea, then you haven't truly understood it in any useful way. This is so important that it's worth saying a third time:

Writing is an Essential Part of Understanding

This is why, in all of the above guidelines, you see recommendations to write things down.

Note that the 'reading in' phase of the project can last quite a long time (there's a lot of reading *and writing* to be done) and it can overlap partially with some of the other early tasks, such as requirement elicitation, the topic of the next section, and problem modeling, the subsequent section.

If you are working on a straight-forward engineering project, this reading in phase may take four to six weeks. However, if you are working on a research project it can take much longer because the problem may not be easily defined or the requirements may not be clear. More especially the problem modeling – which will involve the development of a deep understanding of the theoretical foundations required to solve the problem – will typically require you to go through in detail how other people have tried to address the problem. This takes time. In a research project, this combined understanding of the nature of a problem and the spectrum of possible approaches is encapsulated in what is known as the literature survey. This is a well-structured synthesis which collects together all the relevant ideas, organizes them, and presents each of them in turn, highlighting their strengths and weaknesses. It is not trivial to create this synthesis. It will require many attempts and many re-writes. You must start with your own short summary of each paper you have read. Then you try to organize the ideas, creating different classes of topics and relating them together in the form of a taxonomy, or hierarchical structure. This taxonomy then provides you with a way to structure the literature survey, typically by effecting a breadth-first traversal of the taxonomy tree.

4 Requirements Elicitation

Having chosen your project, you will have in your possession a short description of what is involved in the project. You will realize by now that this is completely insufficient for you as a basis for doing the project. Consequently, your next task must be to find out exactly – and completely – what the project entails. This isn't as easy as it sounds. You might think that

you should just ask your supervisor and he or she should tell you. It doesn't work like that. Quite often, a supervisor won't have an exact (and complete) model of what is required – supervisors are just like engineering clients and customers in the business and industrial world. It is your job to help your supervisor identify exactly what he wants. That's what good engineers do: they help people understand what they want and then they build it for them. Here's how you do it.

1. Talk to your supervisor.

2. Write down everything he or she says (by 'write down' I mean take notes of his or her words).

3. Write up everything he or she says (by 'write up' I mean express what your supervisor said *in your own words*).

4. Build a document describing what you think is required.

5. Go back to your supervisor and ask for her or his comments.

6. Return to step 1, and keep returning until you are both happy with *your* requirements document.

This all translates into one simple rule: find out what you want the final system to do and how it should behave, write it down, and get everyone involved to agree to it … in writing. And don't spare the detail: every single aspect of what's wanted should be teased out and agreed: what it does, what it doesn't do, how the user is to use it or how it communicates with the user, what messages it displays, how it behaves when the user asks it to do something it expects, and especially how it behaves when the user asks it to do something it doesn't expect.

This process is called requirements generation. It's also called *requirements elicitation* because it reflects better the fact that you have to work actively with the client to find out what they really want (as opposed to what they initially say they want, which is a completely different thing). Once you have been though the requirements elicitation process several times and you are happy that you really know where you want to go, you must write it down and get everyone concerned to agree to it. This then becomes part of the *system specification*: it says *what* you are going to do but not how you are going to do it. A good requirements document will allow you to answer the all-important question: how will I know when I have finished the project? The answer is when you have built a system that satisfies all the requirements.

5 Problem Modelling

Once you know the requirements, and are an expert in the problem domain, the next step is to model it computationally: this means we can identify the *theoretical* tools we need to solve the problem. Examples include statistical analysis for the elimination of noise on the communication channel, characterization of the relationship between fuel consumption and engineer cylinder temperature for the engine control; the extraction of facial features from images, and the statistical classification techniques used to match these feature with faces in a database.

This is the foundation of all engineering and science: the creation of a rigorous – usually mathematical – description of the real physical problem to be addressed, be it control, communications, electronics, or some computational model. For example, if your problem concerned with packet routing, you might represent it using a graph and deploy formal graph theoretic tools for its analysis; if your problem is concerned with signal analysis, you might choose a Fourier representation or an eigen-vector representation and deploy the appropriate theorems in Fourier analysis or linear system theory. If your problem is to do with building a database, you will probably model the system with an entity-relationship diagram and validate the model by normalization.

The key to all successful engineering is the use of an explicit model: if you don't have a model, you are probably not doing engineering. Connecting components (or lines of code) together is not engineering, irrespective of whether it works or not. Without the model you won't be able to analyze the system and, thereby, make firm statements about its robustness, operating parameters, and limitations.

You may also wish to consider the use of a symbolic mathematics package such as Maple in the development of your mathematical model.

6 Systems Analysis and Specification

With the requirements document, problem definition, and computational model identified, we can now say exactly what our system will do and under what circumstances it will do it. This is the system specification. In writing the specification, you should begin with the requirements document and then you should identify the following.

1. The system functionality
2. The operational parameters (conditions under which your system will operate, including required software and hardware systems)
3. Failure modes and actions on failure
4. Limitations & restrictions
5. User interface or system interface

It should also include the following models:[1]

1. A *functional model*. This will usually take the form of a functional decomposition: a hierarchical breakdown of the major functional blocks involved in the processing/analysis/transformation. Typically, this will be a modular decomposition of the computational model. Each leaf node in the functional decomposition tree should have a short description of the functionality provided, the information (data) input, and the information (data) output.

2. *A data model*. The identification of the major data-structures to be used to represent input, output, and temporary information. This is sometimes known as a data dictionary. Note that we are not interested here in the implementation of the data-structures (e.g. linked list, trees, arrays) but with the identification of the data itself. Very often, it is useful to use entity-relationship diagrams to capture the data model.

3. *A process-flow model***.** This model specifies what data flows into and out of each functional block (i.e. into and out of the leaf nodes in the functional decomposition tree). Normally, data-flow diagrams are used to convey this information, and are organized in several levels (i.e. DFD level 0, DFD level 1, etc.) The level zero DFD is equivalent to the system architecture diagram, sometimes called the context diagram, and shows the sources and sinks of information outside your system.

4. *A behavioural model*. This will typically use a state-transition diagram to show the behaviour of the system over time, i.e. the different states it can be in, the event and triggers that cause a change in state, and the functional blocks associated with each state. It is also often useful to create a control-flow diagram: a version of the data-flow diagram with events and triggers superimposed on each process.

5. A clear and detailed definition of all the user and system interfaces; one of the best ways of encapsulating this information is to create a *user-manual*.

In addition to the functional specification, it is often very useful to provide a specification of the non-functional characteristics of the system. The non-functional characteristics refer not to what the software does (its function) but instead to the manner in which it does it: its dependability, security, composability, portability, reusability, interoperability, and re-usability, for example. The non-functional characteristics very often reflect on the quality of the system.

[1] For a software engineering project, these terms refer to the procedural or imperative programming model; different but related terms apply to the object-oriented model.

All this information is collectively known as the '*system specification'* and is the result of an activity know as systems analysis.

Once you have this specification, before proceeding you must return and see if it actually matches what the user needs: *i.e.* you need to validate that the system specification satisfies the requirements (you would be surprised how often it doesn't). If it does, you can proceed to the next activity: software design. If it doesn't, then either the requirements were wrong and need to be changed, or the specification was wrong, and needs to be changed, or, more likely, both were wrong and need to be changed.
You should get the explicit agreement of your supervisor that all is in order. If it isn't, then you must go back to requirements if necessary and revise them and the specifications (with your supervisor's agreement on everything). After this, you validate again, and you keep doing this until everyone agrees. Then, and only then, should you proceed to the next phase of the execution of your project: Design. Here you make the significant transition from what you system will do to how it will do it. To an extent you have already considered the theoretical aspect of how it will do it in the problem model. However, when you consider design you decide how to put this theory into effect, typically by identifying the algorithms and data-structures which need to be used to instantiate the theory and build a system that meets the specification and satisfies the requirements.

7 System Design

You are now in a position to design your system using whatever design methodology is appropriate for the area (and these will inevitably be specific to the particular area, be it filter design, amplifier design, software design, and so on). That said, there are a few general guidelines that apply to all areas:

- Identify several design options – algorithm, data-structures, files, interface protocols – and compare them.

- Analyze your design to ensure it is technically feasible (*i.e.* validate its realizability). Remember, you can't always build everything you design, either for theoretical reasons (ideal filters, for example) or for pragmatic reasons (a 1-Farad capacitor would make for some interesting implementation problems).

- Analyze your design to ensure it meets the specifications (i.e. validate its operational viability)

- Cost your system (*i.e.* validate its economic viability)

- Choose the best design. *You* will have to define what 'best' means for your particular project. It might mean the cheapest to manufacture, it might mean the fastest, and it might mean the smallest – it all depends. It's up to you to identify the test for optimality. As John Canny in MIT once put it when comparing different filtering techniques: *you choose your optimality criterion, and you take your choice.*

  Note well that this is the hallmark of good engineering: the practice of qualitative and quantitative assessment of different options. Note too that our original definition of engineering is reflected in this design process: the creation of effective, efficient, and beneficial systems.

8 Module Implementation & System Integration

Finally, we are at the point where we can build the hardware and/or write the software. There is not much to say here since the construction methodologies are so domain specific, even more than in the case of design. However, there is one small piece of advice which is applicable to all areas: *use a modular construction approach.* Don't attempt to build the entire system in one go in the hope that, when you switch it on or run it, it will work. This is the so-called Big Bang approach (everything comes into existence at one instant) and its name is very appropriate for it almost always results in initial chaos. It is much better to build (and test) each component or modular sub-system individually and then link them or connect them together, again one component at a time.

9 Testing

Many people misunderstand the meaning of the word *testing*. They think it means showing that that their project works. But it doesn't. Testing means much more than this. Certainly, you need to show that it works (*i.e.* that it meets the requirements and operates according to the specification), but a good testing strategy also attempts to break the system: to show not where it works but where it fails. This is sometimes referred to as stress testing. A well-engineered system will always have been stress-tested: that is, taken beyond the point at which it was expected to operate to see how it behaves under unexpected circumstances. This is particularly important for safety-critical systems (*e.g.* a heart pacemaker, an airline navigation system, a stock-exchange transaction processing system …). In engineering, we normally formalize the testing process by referring to three distinct goals:

1. *Verification*
2. *Validation*
3. *Evaluation*

Verification is the testing that establishes whether or not the system is correctly computing the required results and correctly implements the underlying theory. Simply stated,

verification answers the questions: *Have I built the system right? Is it computing the right answer?* This is what most people understand by testing. However, showing that everything is working fine based on only a single test image or data set is not sufficient. Verification testing needs to use a much wider variety of test data (*e.g.*, in the case of visual attention, images that focus on each of the three features of intensity, colour, and motion in isolation, *i.e.* images with intensity cues only, images with colour cues only, and images with velocity cues only). Thus, the verification tests should exercise each module (or computation) independently and also the system as a whole; first checking that the individual component work correctly and then checking that they have been integrated correctly. Finally, verification needs also to run on live data, not just on data in test files.

Validation is the testing that establishes whether or not the system meets the client's (your supervisors') requirements. In this case, the questions are: *Have I built the right system? Does it satisfy the requirements?* It may seem obvious, but you'd be surprised the number of times that the system which is built isn't what is wanted at all. You should compare the system's behaviour with the original requirements and system specification. Validation is extremely important and it should be carried out with great attention to detail. For example, this includes the final layout of the user interface for your system, what controls/inputs need to be adjustable by the user, what outputs are expected, etc. At the very least, it is essential that your system allow the user to adjust all the main parameters upon which your system depends (e.g. any constants or thresholds used by the system). The first step in this is to list all these parameters. Note well that the specification and design of your interface should be done on paper before even thinking about implementing it, and it should be agreed by your supervisors before proceeding. Once you have an application that works correctly and satisfies the requirements, you can then - and only then - move on to consider the third component of testing: performance evaluation.

Finally, in *evaluation* tests, we ask: *How good is the system?* The hallmark of good engineering is to assess the system's performance and compare it to that of other similar systems. Ideally, you should identify some quantitative metric by which to compare the systems, since numbers are the best and perhaps the only way to objectively describe performance. For example, a metric might be the mean time between failures (MTBF) or the number of incorrect rejections in a pattern recognition system. Quite often, we use statistical measures as our comparative metric, *e.g.* the mean and standard deviation of some performance measure when the system is subjected to a large variety of input parameters and conditions.

There are two aspects to evaluation: one is the evaluation with respect to what is known as ground truth data (data for which you know the correct result) and the other is evaluation with respect to other similar systems. In general, we focus on the first in undergraduate and

postgraduate projects but both are important. The next issue is the choice of metrics which are used to measure the performance. Once these are identified, you then move on to design the require test scenarios. Regarding metrics, these typically measure performance as a function of some independent variable (e.g. the complexity of the scene or data set, the number of features present, the speed of motion, the size of the target object, the number of objects, etc.). Finally, the output of a metric is typically either binary (success/failure) or continuous-valued (0-1, indicating the level of success or the amount of error, *e.g.* the distance between the computed result and the real - ground truth - result). Finally, the evaluation tests need to be automated since you will probably have to run them several times as you encounter and fix unforeseen defects (bugs).

10 Documentation

We noted earlier that writing is an essential part of understanding. We note it again here but in a different sense. In this case, writing is essential in order for others to understand what you have done. There are two reasons why you want others to understand your work:

1. So that you can be given credit for it;
2. So that others can carry on your work and develop or maintain your system.

It is extremely important that you document your work at every stage of your project. We saw already that documentation is essential in the initial reading-in, requirements, and specification phases but it is equally important in the design, implementation, test, and maintenance phases.

The best way to organize your writing is to *keep a log book* of all work in progress. You should go out and buy a nice hard-cover notebook and write everything you do on the project into this log book *every day*. Every thought and observation you have on your project should go into this book, along with notes of meetings with your supervisor, results, theoretical developments, calculations, everything. This log book will become an invaluable source of material when you come to write up your project in the final report.

However, don't wait until the end of the project to begin the process of formal documentation. At the end of each phase of the project (or at the end of each task) you should write up a formal report on that phase. These reports will, in turn, become an excellent basis for your final report.

Reports – user manual, design documents, reference manual – are called external documentation. If the project involves software development, you should also prepare internal documentation. These are the comments that explain your code. Detailed guidelines for the documentation of source code exist and you should refer to them.