Ten Unsafe Assumptions When Teaching Topics in Software Engineering

David Vernon(✉)

Carnegie Mellon University Africa, Kigali, Rwanda
david@vernon.eu
http://www.vernon.eu

Abstract. Software engineering is a branch of systems engineering and, to be successful, software engineering students must work in a systems-focussed manner. Instructors, including the author, routinely assume that students have the requisite skills for this or can learn them quickly. This article identifies ten common assumptions that are unsafe to make and, if made, impact negatively on the ability of a student to acquire the essential foundation on which to build their understanding of the technical aspects of software engineering. The ten unsafe assumptions are that students understand how to decompose problems, that they know that systems have to be specified at different levels of abstraction, that they know how to bridge different levels of abstraction, that they understand how software and hardware reflect these different levels, that they can follow instructions and pay attention to detail, that they can easily follow oral or written explanations, that they are able to stress test their own software, that they understand the relevance of professional practice, that they are adept at self-criticism, and they understand the relevance of examples. In each case, we identify the implications for teaching practice of not making these assumptions.

AQ1

Keywords: Teaching practices · Unsafe assumptions · Foundational skills · Effective learning

1 Introduction

Software engineering is a branch of systems engineering and, as such, it is concerned with analysing, modelling, designing, implementing, and delivering large-scale complex software systems. Consequently, there are multiple dimensions to software engineering. One dimension embraces the tools, techniques, methods, and processes required to develop software. A second embraces the management techniques required to organize software projects successfully, to monitor the effectiveness of the development, and to improve the development process. A third addresses the way in which the non-functional attributes of the software being developed are achieved. Non-functional attributes refer not to what the software does (its function) but instead to the manner in which it does

J.-M. Bruel et al. (Eds.): FISEE 2019, LNCS 12271, pp. 1–16, 2020.
https://doi.org/10.1007/978-3-030-57663-9_9

it (its dependability, security, composability, portability, interoperability, sometimes referred to as the "-ilities" [18]).

Software engineering degrees comprise courses that cover all of these topics. However, the instructors who teach these courses, including the author of this essay, often assume that students are well equipped with the requisite foundational skills to learn the technical material in these courses. In the following, I suggest that such assumptions are not always safe[1] and that making them can have a significantly adverse impact on the students' learning experience and the effectiveness of the instructor's teaching practice. On the contrary, I suggest the changes that are required to compensate for not making these assumptions might possibly contribute to more effective teaching practice in general.

While much of what follows might seem trivial and the suggestions almost self-evident, that does not diminish the impact of taking them on board and putting them into practice. As we will see, they are consistent with pedagogical principles, both in computer science and software engineering [11] and in education generally [1]. Most of the suggestions focus on first and second year students. Others, especially the last three which are concerned with professional practice and soft-skills, will benefit students at all levels as they become increasingly accomplished.

The contribution this essay makes to a volume devoted to exploring the frontiers in software engineering education is less about advocating some specific approach such as component-based software engineering, much less about introducing a revolutionary new pedagogy, but rather about adjusting teaching practices to address some of the foundational systems-oriented skills that underpin a student's ability to engage successfully with software engineering education.

2 Unsafe Assumptions

2.1 Students Understand How to Decompose Problems

Software engineering is concerned with solving problems and building reliable software systems that meet the needs of users. When setting software engineering assignments, it is common practice to state the problems in the form of functional requirements. In contrast to the requirements which are elicited when working with an industrial or commercial client, these requirements are usually complete and unambiguous and, hence, rather atypical. The job of a student

[1] When teaching software-focussed courses to students from eighteen countries in Africa I encountered several difficulties in delivering material in a style that had apparently worked well in other parts of the world, including Europe, the Middle East, and Russia. I say "apparently" because, when forced to address these difficulties by adopting a more student-centric stance and questioning what I was assuming about students' foundational skills, it became clear that other students might benefit from the changes to teaching practice that arose from not making these evidently unsafe assumptions.

is to deploy her or his knowledge and know-how to transform these requirements into to a software system. One of the principal difficulties is that mapping from requirements—even complete unambiguous requirements—to finished system requires the student to formulate a feasible way of attacking the problem. There are several aspects to this plan of attack, all of which are evidently very difficult for inexperienced students. They include the processes of problem decomposition, problem modelling (typically requiring the identification of a computational model of the problem), systems analysis (involving the generation of functional, data, and behavioral models, as well as the modelling of non-functional aspects), design (involving the selection of effective and efficient algorithms and data structures), implementation, and various forms of verification and validation testing. In terms of abstraction, the gap between problem statement and final operational solution is big. We will address the pivotal issue of abstraction in more detail in subsequent sections. Here we focus on the very first process: problem decomposition and the identification of a plan of attack.

Experienced practitioners, including instructors, find the process of problem decomposition and coming up with a plan of attack straightforward [16]. It is evident that inexperienced students find the very opposite: the ability to decompose problems doesn't come naturally to many students. The solution is straightforward: demonstrate how it is done.

Since demonstrations are most successful when they matter to the person observing the demonstration and they have something to gain from it, one successful approach is to fashion laboratory exercises for each assignment with detailed instructions on how to attack the problem, typically ten to twenty individual steps, each step comprising a distinct fine-grained sub-problem that is clearly-stated and amenable to direct attack. As the course progresses, assignments are accompanied by laboratory exercises comprising fewer and fewer instructions, progressively weaning the students off this form of assistance. The final assignment provides none. Employing this approach changes the class dynamic and results in significantly greater motivation on the part of the students, increased rate of learning, and, ultimately, the acquisition of reasonable skill in problem decomposition.

2.2 Students Know that All Systems Have to Be Specified at Different Levels of Abstraction

One of the most important concepts we teach in software engineering is the power of abstraction [11]. In the previous section, we noted the big gap between the high level of abstraction of a problem statement and the low level of abstraction of final operational solution. This gives rise to two key difficulties faced by students: (a) to recognize that this gap exists at all, and (b) to learn how to bridge the gap effectively.

Regarding the first, it is apparent that many students do not realize that problems, solutions, and systems in general need to be understood at different levels of abstraction and that doing so is necessary if one is to map from problem to solution. The concept of abstraction is routinely taught in courses on data

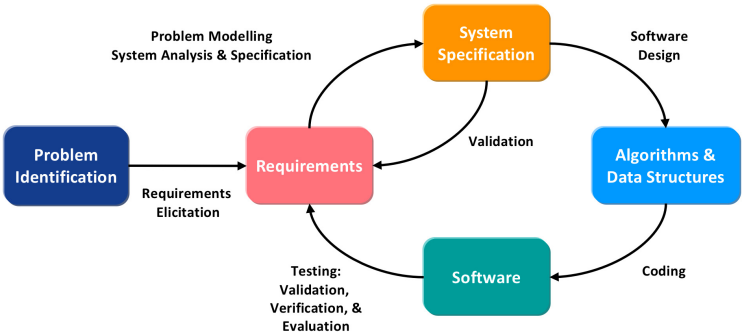

Fig. 1. One version of the software development life cycle.

structures and algorithm when introducing abstract data types (ADTs) and the associated concept of data hiding [12]. The same applies at the implementation level, manifested in programming languages that expose functionality through abstract interfaces [11]. However, here we are generalizing this beyond the design and implementation phases and we are applying it across the complete software development life cycle, from problem modelling to solution implementation (see Fig. 1). The key idea is that the level of abstraction become progressively lower as we proceed from the early problem modelling phases (which are typically understood at a high level of abstraction) through to system modelling, to design, and finally to implementation (which is specified at the lowest level of abstraction) [7]. As we progress, more and more detail is added to the description as the concepts are specified in less abstract terms. One of the most common difficulties encountered by students is that, almost always unwittingly, they suffer from level confusion where they don't realize that they are applying the wrong form of analysis in the wrong phase of the development life cycle, e.g. establishing a theoretical understanding of the solution when developing an algorithm or, most common of all, formulating an algorithm when writing the code during the implementation phase.

More generally, all systems can be viewed at different levels of abstraction, successively removing specific details at higher levels and keeping just the general essence of what is important for a useful model of the system.[2]

As part of his influential work on modelling the human visual system [8], David Marr advocated a three-level hierarchy of abstraction, often referred to at the *Levels of Understanding* framework; see Fig. 2. He argued that the framework applies to any information processing system. At the top level, there is the computational theory. Below this, there is the level of representation and algorithm. At the bottom, there is the hardware implementation. At the level of the computational theory, you need to answer questions such as "what is the goal of the computation, why is it appropriate, and what is the logic of the

[2] The remainder of this section on levels of abstraction follows closely the treatment in [19].

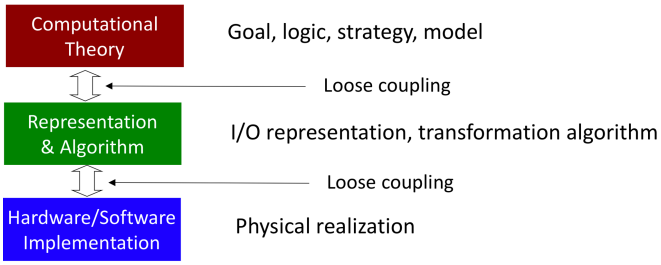

Fig. 2. The three levels at which an information processing system should be understood and modelled: the computational theory that formalizes the problem, the representational and algorithmic level that addresses the implementation of the theory, and the hardware level that physically realizes the system (after David Marr [8]). The computational theory is primary and the system should be understood and modelled first at this level of abstraction, although the representational and algorithmic level is often more intuitively accessible.

strategy by which it is carried out?" At the level of representation and algorithm, the questions are different: "how can this computational theory be applied? In particular, what is the representation for the input and output, and what is the algorithm for the transformation?". Finally, the question at the level of hardware implementation is "how can the representation and algorithm be physically realized?" In other words, how can we build the physical system? Marr emphasized that these three levels are only loosely coupled: you can—and, according to Marr, you should—think about one level without necessarily paying any attention to those below it. Thus, you begin modelling at the computational level, ideally described in some mathematical formalism, moving on to representations and algorithms once the model is complete, and finally you can decide how to implement these representations and algorithms to realize the working system. Marr's point is that, although the algorithm and representation levels are more accessible, it is the computational or theoretical level that is critically important from an information processing perspective. In essence, he states that the problem can and should first be modelled at the abstract level of the computational theory without strong reference to the lower and less abstract levels.[3]

Marr illustrated his argument succinctly by comparing the problem of understanding vision (Marr's own goal) to the problem of understanding the mechanics of flight.

> "Trying to understand perception by studying only neurons is like trying to understand bird flight by studying only feathers: it just cannot be done.

[3] Tomaso Poggio recently proposed a revision of Marr's three-level hierarchy in which he advocates greater emphasis on the connections between the levels and an extension of the range of levels, adding *Learning and Development* on top of the computational theory level (specifically hierarchical learning), and *Evolution* on top of that [13]. Tomaso Poggio co-authored the original paper [9] on which David Marr based his more famous treatment in his 1982 book *Vision* [8].

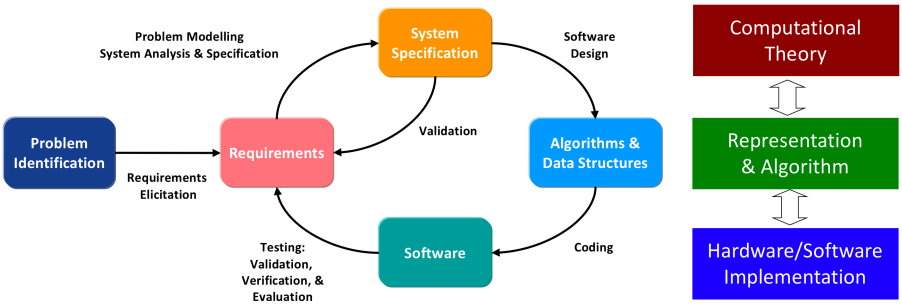

Fig. 3. The different levels of abstraction mapped to the software development life cycle.

> In order to understand bird flight, we have to understand aerodynamics; only then do the structure of feathers and the different shapes of birds' wings make sense"

Objects with different cross-sectional profiles give rise to different pressure patterns on the object when they move through a fluid such as air (or when a fluid flows around an object). If you choose the right cross-section then there is more pressure on the bottom than on the top, resulting in a lifting force that counteracts the force of gravity and allows the object to fly. It isn't until you know this that you can begin to understand the problem in a way that will yield a solution for your specific needs.

Of course, you eventually have to decide how to realize a computational model but this comes later. The point Marr was making is that you should decouple the different levels of abstraction and begin your analysis at the highest level, avoiding consideration of implementation issues until the computational or theoretical model is complete. When it is, it can then subsequently drive the decisions that need to be taken at the lower level when realizing the physical system.

Marr's dissociation of the different levels of abstraction is significant because it provides an elegant way to build a complex system by addressing it in sequential stages of decreasing abstraction. It is a very general approach and can be applied successfully to modelling, designing, and building many different systems that depend on the ability to process information.

Introducing Marr's levels of understanding framework is a very effective way of exposing students to the need to treat problems at different levels of abstraction at different phases of the software development life cycle (thereby addressing the first difficulty identified at the start of this section) and it also provides a way of mapping it to the software development life cycle itself; see Fig. 3.

2.3 Students Know How to Bridge Different Levels of Abstraction

Knowing that any complex system can be modelled and understood at several levels of abstraction is essential in being able to engineer effective, efficient, and

appropriate software. However, there still remains the problem of knowing how to map from one level of abstraction to another. This isn't trivial [7]. As we saw above, this mapping happens when one goes from the computational model to the design and from the design to the implementation. When teaching software engineering, we often use the mapping from computational theory to design as the archetypal example of this process, pointing out how a given model and the associated solution strategy often has many possible algorithms and data structures. We call these *design choices*. For example, a digital filter can be modelled as a spectral process using the Fourier transform but there are several Fourier transform algorithms that can yield the required transformation from temporal or spatial data to frequency spectral data, e.g. the discrete Fourier transform (DFT), the fast Fourier transform (FFT) [4], and different implementations, e.g. the fastest Fourier transform in the west (FFTW) [5]. Sorting a list can be characterized as the identification of the permutation of the elements of the list that satisfies an ordering constraint, with many different sorting algorithms to choose from, and with the choice depending on the required computational complexity, space complexity, and need for stability or not.

However, the level-of-abstraction mapping problem occurs at the transition of every phase of the development life cycle and the goal is to help students understand that there are choices to be made at each transition and that the only way to do the mapping effectively is to enumerate the choices and identify the selection criteria. By continually making these choices explicit, the message is absorbed by the interested student.

There exists yet another situation in which this level-of-abstraction mapping problem arises: when developing solution strategies. Typically, these are initially expressed in high-level abstract strategies and the difficulty is to translate this conceptual understanding into detailed low-level unpacked tactics [16].

The key to overcoming these difficulties is through the copious use of examples. People learn by induction: by inferring general principles from multiple instances of particular cases. The second key to overcoming these difficulties is though practice: repeated engagement with the process so the inductive insight is revealed quicker with experience.

The general approach, then, is to expose students to the practice of *progressive deepening*: covering the same material multiple times at different levels of abstraction or detail.

Some readers may be uncomfortable at this point with an approach based on establishing an intuitive appreciation for the process and an understanding that it requires as much skill and experience as it does knowledge. These readers might justify these concerns on the basis that we are dealing here with science and engineering, and that, as such, the problem would be finessed if we simply followed established engineering practices or methodologies. Model-driven software engineering [3,14] or formal methods [2,10,15] would, to some extent, alleviate these concerns. However, it should be remembered that we are speaking here of both understanding the process and using an effective methodology to bridge the gap in abstraction between requirements and implementation in

several steps. Even if in possession of tools to help automate this process, it is essential that the student understand what the tool is helping to automate.

2.4 Students Understand How Software and Hardware Reflect the Different Levels of Abstraction

As we have said, software engineering rightly emphasizes the importance of abstraction [7,11]. As we noted in Sect. 2.2, it contributes to clarity and transparency in all phases of the software development life cycle, including computational modelling, design, and implementation. Abstraction hides the unnecessary details of the levels below. However, there are times when these details matter and they can matter at every level of abstraction and at every phase of the life cycle [7]. In Marr's levels of understanding framework, the levels are loosely-coupled, not entirely decoupled. This is where an in-depth understanding of all layers in the realization of a software system—from application to the hardware architecture—can help with developing effective, efficient software. Developers with this knowledge are often referred to as *full-stack* engineers or developers.[4] Regrettably, many students are exposed only to high-level abstract knowledge and this has two negative effects.

First, it means that the instructor is not able to revert to low-level implementation details, typically at the level of middleware, operating systems, and computer architecture, to explain key topics.

For example, when teaching pointers, it is often helpful to explain to students that they are effectively memory addresses and that referencing and dereferencing involves the manipulation of addresses of data and access to the data at given addresses. The purely abstract concept of referencing and dereferencing, while valid, involves more sophisticated semantics than the semantics of location and content and often takes students longer to grasp. Furthermore, understanding that pointers are memory addresses helps students understand why dereferencing an uninitialized pointer is dangerous and potentially harmful without some form of memory protection. It also helps when explaining the meaning of segmentation errors, i.e. attempts the access memory via pointers that lie outside the user's memory space.

When teaching the semantics of the CAR, CDR, and CONS function in Common Lisp [6], understanding that lists are implemented as linked lists with two fields, both of which are pointers (and that CAR and CDR derive from reference to an address register), helps greatly in getting the semantics of these functions across to students. Without that implementation level understanding,

[4] In the past few years, the concept of a full-stack developer has changed in a significant way, referring not to a developer with knowledge of all levels of the implementation of a system, including the underlying middleware, operating system, and computer architecture, but instead to a developer who is conversant with both front-end and back-end development in client-server architectures. This alternative meaning loses the emphasis on mastery of the many levels of abstraction, substituting instead a mastery of user-interface programming and information processing.

the different forms of equality operator, e.g. EQ, EQL, EQUAL, and EQUALP are very difficult to grasp.

When explaining the semantics of recursion and the role of the implicit stack, it helps to refer to the implementation of this stack in the operating system, specifically the use of the process stack for dynamic memory allocation and the associated time and space cost of pushing and popping the function state when calling functions recursively and returning from recursive calls. Given the importance of recursive algorithms and their dependence on the persistence of local variables during recursion, this low-level understanding, well-below the level of abstraction necessary to explain the semantics of recursion, helps student understanding.

Similarly, when explaining dynamic memory allocation by a client programmer, knowledge of the operating system heap is helpful (and is essential if one is to make any sense of "stack overruns heap" fatal error messages). It also helps understand why buffer overruns can be difficult to trace and why memory leakage is such an important problem.

Knowledge of computer architecture too, even the basic von Neuman architecture, can help students understand key concepts such as caching and the relevance of cache memory. Arguably, it is essential to understand why caching may not be of any value when accessing dynamic data structures that have physical addresses which are not all be present in the current memory cache.

These are just some examples to make the point that full-stack knowledge can aid both in instruction and in building a more complete picture for the student of the trade-offs required when designing efficient software.

We mentioned above that there are two negative effects from being exposed only to high-level abstract knowledge. The second is to do with how learning works. As Ambrose *et al.* point out, [1], p. 49, experts and novices organize knowledge in different ways. Specifically, experts have a much higher density of connections between concepts, facts, and skills. Exposing these connections and highlighting cross-linkage among topics helps novices transition to expert-level understanding. Exploiting full-stack knowledge is an effective way of demonstrating this and, in the process, helping the student build a deeper understanding of software engineering. Professionals—experts—typically see the software in the context of the full stack, always working at the correct level of abstraction but always being aware of the network of connections to the other levels.

2.5 Students Can Follow Instructions and Pay Attention to Detail

Students are human beings and human beings have limited working memory and a natural capacity for pattern recognition. Consequently, students often tend to look at things— problem statements, algorithms, and code—from a global holistic perspective first and then infer the details. This causes many problems because, while this may well help in forming an intuitive understanding of the issue, it is very problematic when it comes to bridging the gap to the necessary detailed description of the issue at hand.

One striking example of this is the inclination of students to look at a code segment or a pseudo-code representation of an algorithm and then try to understand what is happening by looking for a pattern in the description as a whole. Unfortunately, the meaning of the whole, at least in software, emerges from an understanding on the individual statements and their relationship to each other: meaning emerges from the detail and an understanding of that meaning requires students to inhibit their predisposition to look at the code and infer what it means. Consequently, it turns out that the key to success in this is to get the students to adopt a letter-box view of code, seeing only one statement at a time, inhibiting the natural tendency to link it to all the other parts of which you are aware, and build the understanding of the process, step by step. As we will see in Sect. 2.6, fine-grained diagrams help greatly with this.

The same issue arises when students attempt to assimilate written material. There is a prevalent tendency to try to get the gist of the material by skimming through it. While this is of itself fine, the problem arises with the belief that complete understanding follows multiple skims through the material. This becomes particularly problematic when following detailed written instructions, e.g. detailed software installation instructions or lengthy on-line tutorials [17]. Students very often don't have the patience or discipline to follow each instruction exactly, one step at a time. Either they skip steps in the instructions or they guess what is meant without digesting fully what is meant. The idea that instructions mean exactly what the say and only what they say is often hard for students to grasp. The problem is exacerbated by providing fewer richer instructions because these individual instructions then become subject to the same skimming process to establish the gist and guess the meaning as larger tracts of material. Paradoxically, the solution is to provide very fine grained highly-explicit detailed instructions that minimize the tendency to skim and to demonstrate that following them exactly does lead to success.

Of course, students also need to learn to follow coarse-grained instructions. To facilitate this, the instructions for laboratory exercises that support assignments (see Sect. 2.1) might take the form advocated above, i.e. many fine-grained detailed explicit instructions, but reducing their number and making them progressively more coarse-grained over the course of a semester.

2.6 Students Can Easily Follow Oral or Written Explanations

We depend on spoken and written language as the main medium of instruction when teaching. As with all forms of effective communication, one needs to keep the vocabulary focussed and not use language that is unnecessarily complex. The exception, of course, is where one is introducing new technical vocabulary and the semantics of new concepts. However, this, in itself, is not enough. Something more is needed: graphic depiction of material. Diagrammatic illustration matters, far more than one might expect. This becomes particularly evident when performing structured walkthroughs of software code. The need to understand both the syntax and the semantics of the code is often difficult for many students. This is compounded when you factor in the temporal nature of code

execution. Unwinding iterative constructs by describing in words what is happening is inadequate, even if it's common practice. Unwinding recursion is even worse. However, when supported by diagrams that show each step graphically—each change of state that results from each statement in the software—using a very fine-grained approach, the computational processes become much clearer to the students. In essence, diagrams matter when describing structure and they matter even more when describing processes. Processes need to be illustrated at a fine level of temporal detail. Depending on language-based explanation alone doesn't work well. Conversely, using diagrams to unwind temporal processes and visualize key concepts makes a significant difference to the student's learning experience.

2.7 Students Are Able to Stress Test Their Own Software

Every software engineering course has at least a few classes devoted to testing. We distinguish between verification (of functionality against specifications), validation (of functionality and non-functional characteristics against requirements), and benchmarking (against the performance of other solutions or systems). We explain the important of static testing, dynamic testing, black box testing, white box testing, grey box testing, stress testing, unit tests, integration tests, system tests, acceptance tests, and regression tests. And we assume that students can learn and apply the testing techniques effectively and efficiently. This assumption may not be safe, especially when it comes to stress testing.

Despite being made aware of Dijkstra's warning that testing can only be used to demonstrate the presence of errors in software, not their absence, students often see testing as demonstration that their software "works". Reversing the logic and seeing testing as an exercise in trying to break the software is not something that comes naturally to students, particularly when testing their own code. This is a natural consequence that they have a vested interest in their code—their creation—and they are psychologically ill-equipped to intentionally damage it by subjecting it to data that undermines its functionality, i.e. to be aggressive in developing test strategies and to identify particularly difficult unforeseen boundary test scenarios.

The resolution of this problem turns out to be quite straightforward. By adopting a policy of providing sample test cases when issuing an assignment but of always assigning marks on the basis of performance against a large set of blind test cases, students quickly learn to be more aggressive in testing. It may take three or four assignments to grasp the message, but eventually, as marks are consistently deducted for failures on boundary cases, students come to understand that showing the program works on typical data is far from sufficient to get a good grade. That requires a more aggressive approach and, as a result, the stress tests become more thorough.

In all of this, however, it is equally important to emphasize the truism of software engineering that you can't test quality into a program: it must be designed in.

2.8 Students Understand the Importance of Professional Practice

Students are often required to engage with companies, either individually or in teams, as part of their program of education in order to prepare them for the realities of working in a commercial or industrial environment after they graduate. Often, these exercises involve the development and delivery of some form of software systems. Typically, the industry client has some problem that needs to be solved or some new capability they would like and they expect the students to solve it or provide the required system. More often than not, they only have vague notions of exactly what they want delivered at the end of the project but they are very adept at recognizing what they do not want.

Students, when charged with liaising with clients are invariably shocked to discover that the client doesn't have a clear set of requirements and, worse, that they are often too busy to engage in a lengthy requirements elicitation process with the students. This is because, while the difficulty of the requirements elicitation process is always taught in class, the reality is that students are usually given assignments couched in terms of a clear requirements document and with clear acceptance tests in the form of a grading rubric. Students rarely have to concern themselves with what is unknown about the project they have to complete and, over time, this lulls them into a false sense of security that the requirements elicitation process is straightforward. Nothing, of course, could be further from the truth.

Having to confront these difficulties provides one of the most valuable lessons a student of software engineering can learn. They begin to appreciate the importance of dealing with uncertainty and ambiguity, striving to understand what the client needs, rather than what the client says they need. In the process, they also learn some of the most important soft skills that underpin success in the commercial world of software engineering, specifically how to manage the client—with politeness, respect, and patience—in order to maximize the value of the limited time the client is willing to invest when meeting with the students. This requires two adjustments in teaching practice: (a) don't always provide complete specifications and let the students grapple with that uncertainty (this point appears in a different guise in the section on testing), and (b) be prepared to tutor students as much in professional practice, e.g. the soft skills of client liaison, meeting etiquette, writing minutes, timely delivery of documents being tabled for discussion, as you do in the technical aspects of software engineering.

2.9 Students Are Adept at Self-criticism

Just as we teach students that one of the main goals of software engineering is continuous process improvement, including quantitative and qualitative quality assessment, we often assume that students understand that they are part of the process and, consequently, the same goals apply to them and the manner in which they conduct themselves as aspiring professionals. However, it is evident that often students do not make the connection with the attendant need to continually question everything they say and everything they do. In other words,

students don't automatically see the need for self-referential (or reflexive) quality assurance. Moreover, even when they are made aware of this, they don't typically have a ready disposition to engage with it.

Teaching quality assurance in software engineering—something that should not be relegated to a single course or even a part of a course—offers another opportunity to help students learn the importance of professional practice and soft skills. Since a software engineer is an intrinsic part of the software engineering process, the two issues—quality assurance in software engineering and quality assurance in professional practice—are just two sides of the same coin.

The magnitude of the task of inculcating a disposition to engage in the practice of self-referential quality assurance is more considerable than one might imagine. People are generally not self-critical. We met a similar trait in Sect. 2.7 on stress testing your own code. There are many ways one can approach it. For example, by imposing exacting standards for the manner in which students frame questions in class (and elsewhere): saying exactly what they mean and meaning exactly what they say. Often, students will have a vague notion of what they don't understand and, when asking a question, it's not uncommon for them to begin to speak while not yet having a clear idea of what it is they want to establish in the answer they expect. Very often, they expect the instructor to interpret their poorly- and partially-framed question and infer their intent and meaning. And very often, we do. However, in doing so, we do them no favors because we then reinforce this poor style of questioning and, implicitly, teach them poor professional practice. It can be painful and slow, but working with a student to have them re-frame their question and re-frame it again and again until they are finally asking exactly what the need to know. It comes down to precision of expression, something that should permeate the professional life of a software engineer. The same approach can be applied to written reports, to software documentation, to algorithm design, and to computational modelling: to every aspect of the software development life cycle.

Finally, as instructors in software engineering, it is imperative that we, in turn, embrace this almost obsessive-compulsive attachment to precision in all our dealings with students so that they learn by example. This is not as easy as it sounds. It means we must adopt a teaching practice that is founded on leading by example and by being transparently self-critical in everything we do. The transparency is important. We must correct ourselves in class when we identify a weakness or flaw in what we've said. When writing reports or papers, we routinely make multiple passes at editing the text, applying red ink copiously and without restraint in an effort to polish our writing in search of precision (as well as brevity, simplicity, and clarity). It can be a helpful device to keep these multiple edits on hand to show students that we apply to ourselves the same standards that we expect of them. This leads to the tenth and final unsafe assumption.

2.10 Students Understand the Relevance of Examples

Examples provide essential illustration. However, examples can expose far more than just the meaning of the principle, practice, or technique currently being examined. Important though that is, examples also provide an opportunity to leverage several of the evidence-based principles of how learning works, as expounded by Ambrose *et al.* [1]. These principles can be exploited in many other ways, of course, but here we will take the opportunity to show how examples provide a concrete way to leverage them to achieve more effective educational practice.

One of the principles is that "to develop mastery, students must acquire component skills, practice integrating them, and know when to apply what they have learned" [1], p. 95. When coupled with Meyer's observation that "to learn a technique or a trade it is best to start by looking at the example of excellent work produced by professionals, and taking advantage of it by (in order) using that work, understanding its internal construction, extending it, improving it – and starting to build your own" [11], the power of a well-presented example is clear.

Examples demonstrate the process of application. Furthermore, if the examples relate to one another, they provide another opportunity to demonstrate the difference between the way an expert organizes knowledge and the way a novice organizes knowledge, as we noted already, with the former having a much higher density of connections among concept, facts, and skills. Exposing these connections and highlighting cross-linkage among topics can be accomplished by stating them but demonstrating them through examples is even more effective.

The other aspect of examples is that they provide an opportunity for practice through well-designed assignments, especially ones that reflect authentic, real-world tasks: these are the types of example that feed student motivation. The importance of assignments is well understood but it is worth re-emphasizing: "Students must learn to assess the demands of the task, evaluate their own knowledge and skills, plan their approach, monitor their projects, and adjust their strategies as needed" [1], p. 191, reflecting another of the principles that goal-directed practice coupled with targeted feedback are critical to learning [1], p. 125. It is even better if scaffolding can be built into assignments, later assignments leveraging what has been learned in earlier assignments. This demonstrates the importance of reuse in a very practical manner while also speaking to the core importance of systems thinking in software engineering.

3 Conclusion

In this essay, I have tried to convince that questioning assumptions about the degree to which students possess foundational skills serves to reveal ways in which teaching practice might be improved. It is important to emphasize that this makes no value judgement about the aptitude or ability of the student but rather targets the idiosyncrasies of how people learn to master the *skills* that underpin software engineering. The contention is that recognizing these idiosyncrasies can

help improve educational practice and contribute to more effective teaching. It rebalances a possibly unbalanced approach that might favor a focus on content over the object of teaching, i.e. the student. This more balanced approach is captured succinctly by the exhortation to "Teach students, not content" [1].

Acknowledgements. Many thanks go to the reviewers of an earlier version of this paper for their helpful and constructive comments.

References

1. Ambrose, S.A., Bridges, M.W., DiPietro, M., Lovett, M.C., Norman, M.K., Mayer, R.E.: How Learning Works: Seven Research-Based Principles for Smart Teaching. Wiley, Hoboken (2010)
2. Bjørner, D., Havelund, K.: 40 years of formal methods. In: Jones, C., Pihlajasaari, P., Sun, J. (eds.) FM 2014. LNCS, vol. 8442, pp. 42–61. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-06410-9_4
3. Bruyninckx, H., Klotzbücher, M., Hochgeschwender, N., Kraetzschmar, G., Gherardi, L., Brugali, D.: The BRICS component model: a model-based development paradigm for complex robotics software systems. In: Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC 2013, pp. 1758–1764. ACM, New York (2013)
4. Cooley, J.W., Tukey, J.W.: An algorithm for the machine calculation of complex fourier series. Math. Comput. **19**(90), 297–301 (1965)
5. Frigo, M., Johnson, S.G.: The design and implementation of FFTW3. Proc. IEEE **93**(2), 216–231 (2005)
6. Graham, P.: ANSI Common Lisp. Prentice Hall (1996)
7. Kramer, J., Hazzan, O.: The role of abstraction in software engineering. In: Proceedings of International Conference on Software Engineering (ICSE 2006), vol. 31, pp. 1017–1018. ACM SIGSOFT Software Engineering Notes, Shanghai (2006)
8. Marr, D.: Vision. Freeman, San Francisco (1982)
9. Marr, D., Poggio, T.: From understanding computation to understanding neural circuitry. In: Poppel, E., Held, R., Dowling, J.E. (eds.) Neuronal Mechanisms in Visual Perception, Neurosciences Research Program Bulletin, vol. 15, pp. 470–488 (1977)
10. Why don't people use formal methods? https://www.hillelwayne.com/post/why-dont-people-use-formalmethods/
11. Meyer, B.: Touch of Class - Learning to Program Well with Objects and Contracts. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-540-92145-5
12. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. Commun. ACM **15**(12), 1053–1058 (1972)
13. Poggio, T.: The levels of understanding framework, revised. Perception **41**, 1017–1023 (2012)
14. Schlegel, C., Steck, A., Lotz, A.: Model-driven software development in robotics: communication patterns as key for a robotics component model. In: Introduction to Modern Robotics. iConcept Press (2011)
15. Schumann, J.M.: Formal methods in software engineering. In: Schumann, J.M. (ed.) Automated Theorem Proving in Software Engineering, pp. 11–22. Springer, Heidelberg (2001). https://doi.org/10.1007/978-3-662-22646-9_2

16. Skiena, S.: The Algorithm Design Manual, 2nd edn. Springer, Heidelberg (2010). https://doi.org/10.1007/978-1-84800-070-4
17. CRAM simple mobile manipulation plan. http://cramsystem.org/tutorials/intermediate/simple_mobile_manipulation_plan
18. Vanthienen, D., Klotzbücher, M., Bruyninckx, H.: The 5C-based architectural composition pattern: lessons learned from re-developing the iTaSC framework for constraint-based robot programming. J. Softw. Eng. Robot. **5**(1), 17–35 (2014)
19. Vernon, D.: Artificial Cognitive Systems – A Primer. MIT Press, Cambridge (2014)