Situated Learning Theory and Learner Programming Environments

Austin Cory Bart

Virginia Tech

Abstract

Programming Environments are a key tool for programmers, the place where most of the work of programming is done. As the quantity and category of novice programmers increases, we must find new ways to refocus these programming environments for learners. Situated Learning theory and frameworks built off it offer ways to reapproach the design of these environments from a new perspective. In this paper, we analyze tensions that arise during this design process, especially with regards to authenticating the novices learning experience as we ask the fundamental question of how much we can sacrifice realism. As a guide, four aspects of situated learning environment design are considered – context, content, facilitation, and assessment. Ultimately, a number of a issues are identified along with some recommended partial solutions, but further research on the applicability of SL theory to the design of programming environments is required.

Keywords: programming environments, situated learning, authentication

Situated Learning Theory and Learner Programming Environments

Situated Learning Theory is based on the hypothesis that authenticity is a key element of successful learning; at the same time, novices must be scaffolded so that they are eased into the activity they are learning about. In this paper, I will analyze the tension between authenticity and scaffolding in the context of programming environments – the encompassing tool that developers use on a day-to-day basis to create problem-solving programs. The paper will begin with a background on Situated Learning Theory and the design of learning environments. Then, using theories and frameworks taken from the literature, we will discuss how the design of programming environments affects beginners, using a number of programming environments as reference points.

Situated Learning Theory

Stuated Learning Theory, originally proposed by Lave and Wenger, argues that learning normally occurs as a function of the activity, context, and culture in which it is situated Lave & Wenger (1991). Therefore, tasks in the learning environment should parallel real-world tasks, in order to maximize the *authenticity*. Contextualization is key in these settings, as opposed to decontextualized (or "inert") settings. The key difference is that the learning is driven by the problem being solved, rather than the tools available. For example, Table 1, taken from Heeter (2005), demonstrates two different pedagogical styles, the latter of which might be considered grounded in STL.

A critical element of these situated environments is the need for social interaction and collaboration, as learners become involved in and acculturated by "communities of practice". This interaction occurs not only between individuals and the experts of the community (commonly the teachers) in an apprenticeship model, but also occurs between different learners as they adapt at uneven paces. This communication between peers leads to growth among both individuals, especially when access to authentic experts is limited. As the learner develops into an expert, they shift from the outside of the community’s circle to the center, becoming more and more engaged.

Situated Learning Environments

Before we can dive too deeply into Situated Learning Theory, we must first recognize the concept of a Learning Environment which surrounds a student's pedagogical experience. Sawyer (2005) notes that learning environments must foster cognitive engagement if they are to maximize student learning. But what causes students to be motivated to engage? There are many different theories on what the components of motivation are, including a four-element one from Sawyer (2005): how much the students *value* the subject matter – whether intrinsicly (they are interested in the topic itself), instrumentally (the tasks are related to the student's long-term goals), and attainment (the personal satisfaction that comes from achieving a task); how much the student feels *competent* with the material (which can be supported through scaffolding); how much the students need for *relatedness* (the feeling of belonging and acceptance from their peers and instructors) is satisfied; and how much *autonomy* (the sense of control over the environment and direction of the learning) that the student feels.

An alternative model, proposed by Jones (2009), is the mUSIC (eMpowerment, Usefulness, Success, Interest, and Caring) model of motivation. Although there are obvious overlaps between the two models (e.g., empowerment maps to autonomy), there are some fine distinctions. For instance, competency is instead described as success, the latter of which is a more goal-oriented conceptuatilization (you can succeed at a challenge without being competent). There are many more theories on motivation, each with different emphasises and limitations.

Of course, even assuming students are motivated, success is not guaranteed. Sawyer (2005) identifies several key challenges that arise during the design of learning environments. For example, the commonly established roles of isolated learners with the teacher as a primary source shifts to one where the teacher is more of a guide, and the students must work with their peers to learn more communaly. Another challenge is that the the amount of work required by the system might be greater than the students feel is appropriate, based on their prior experience in traditional learning environments – if the classroom is designed to challenge students to engage more deeply in their learning, employing more sophisticated critical thinking skills and working harder to develop expertise, this can lead to students feeling drained and rebellious. Yet another issue given special mention are the dangers of “Seductive Details”, additions to an environment intended to increase students’ perceived value, but that ultimately backfire and draw attention away from the actual issues – the example from the literature is explosions in a chemistry experiment that, while entertaining and attention-grabbing, do not contribute to a students understanding of the chemical interactions at play. Indeed, this is only a small subset of the challenges that are present in designing successful learning environments.

However, one of the biggest and most important complications is the difficulty of making environments *authentic*, tying back into the theory of Situated Learning. As previously discussed, SL theory posits that learning is more effective when authentic and situated. But authenticity is a complex concept, and can be achieved in several different ways. First, the educational material can be connected to the world at large, e.g. using factory production to teach the algebraic concept of rate-of-change (cars produced per hour). Second, the material can be connected to the students own day-to-day experiences, such as teaching grocery-shopping mothers about ratios in the context of unit prices. Third, the material can be taught using skills and information relevant to the eventual discipline of the student, for example teaching biologists about cells by having them use a microscope. These are not disjoint approaches, but are best when mixed together.

Design of Situated Learning Environments

We now go beyond the question of “what is” with regards to learning environments, and instead ask “how” – how can we design learning environments that offer authentic, situated experiences? As a theory, Situated Learning is only a vague guide, so we turn to the literature for frameworks that can offer structure to the design question.

Choi & Hannafin (1995) asks us to consider four different aspects when designing Learning Environments that offer authentic experiences. First is *context*, described by (Rogoff & Lave, 1984, pg. 2) as "... the problem's physical and conceptual structure as well as the purpose of activity and the social milieu in which it is embedded," meaning that context is driven not just by the atmosphere of the problem at hand, but also by the background and culture surronding the problem. It is misleading to think of contexts as being "artificial" or "natural", since the former's implication of a human presence is always present. Instead, they can be viewed as ranging from formal to informal (with a growing "nonformal" movement falling somewhere in between). Students may not learn as much in formal environemnts if work isn't done to integrate everyday cognition. Context relies heavily on authenticity, since this is when students can find recognizable elements and build on prior understanding, eventually being able to transfer to new contexts at will.

The next aspect is *content* – the information intending to be conveyed to the students. If context is the backdrop to the learning, then content might be seen as the actors and script. Naturally, context and content are deeply intertwined with each other, and its difficult to talk about one without referencing the other; in fact, content is an abstract entity that needs to be made concrete when it is delivered to the learner. If the information is too abstract, than it will never connect with the learner and will not be transferable to new domain. However, if it is too grounded in a domain, then it will not be clear how it can be re-applied elsewhere. Ultimately, content must be given in a variety of forms to maximize transfer. Two useful methods for building content are anchored instruction (exploring scenarios, or anchors, in the context based on the content) and cognitive apprenticeship (mediating knowledge from an expert to the novice learner in a mentoring relationship).

The third aspect is *facilitations*, the modifications to the experience that support and accelerate learning. They provide opportunities for students to internalize what they are learning by lowering the barriers that can surround situated experiences, at the cost of some amount of authenticity. Choi & Hannafin (1995) identifies a number of different forms that facilitation can take:

Modelling A keystone for cognitive apprenticeships, modelling is when an expert demonstrates a process to a novice, either physically or by vocalizing underlying thoughts.

Scaffolding Scaffolding is a form of support that is intended to extend what a learner can accomplish on their own. This support is required at the onset of the learning process, but is unnecessary once a sufficient threshold has been passed; during this transition, the amount of scaffolding can be tuned to the learners understanding.

Coaching When coaching, the expert acts as a third-party observer offering advice and guidance (ideally subtly, rather than explicitly). Sometimes this means reminding the learner of steps they forgot, sometimes it means extending the scope of the problem at hand, and sometimes it's just-in-time scaffolding and modelling modifications.

Collaborating Just as coaching is the interaction between the novice and the expert, collaborating is the interaction between novice peers. Students must find consensus in their understanding of the concepts and processes at hand as they develop a shared understanding.

Fading Fading is simply the removal of supports such as Scaffolding and Modelling as the learner improves – the metaphorical taking off of the training wheels. This critical interpolation is when students should develop self-regulatory behaviour and take stronger control of their learning.

Cognitive Tools This is the vaguest and most encompassing category, and describes tools provided to or designed by students that support their learning, such as a calculator for mathematics students or a collaboratively-written wiki for students investigating a historical event.

The fourth and final of the aspects is Assessment, which is usually seen as dual-purpose: in the long term, there is a need to measure the success of the environment; in the short term, there is a need to measure the learning of the student. Choi & Hannafin (1995) gives special attention to the "teach to the test" problem, and how assessment needs to change to measure students ability to solve authentic problem (as opposed to their ability to solve the test's specific problem), and to be able to transfer their understanding when solving different but related problems. It is important that assessment is measured against the individualized goals and progress of a learner, requiring that any standards used be fluid and adaptable to different learners personal situations.

Of course, assessment should be an on-going part of the learning process, providing feedback and diagnostics. Ultimately, the learner should join in the process of assessment as they transition to an expert – being able to meta-cognitively evaluate the effectiveness of ones methods and communicate results to others are key abilities of experts. Choi & Hannafin (1995) offers three examples of tools that can be used to gather more situated, authentic measures of learning:

Portfolios Collections of students work-in-progress or completed projects – their analysis helps students understand the velocity of their growth.

Performance Assessment As opposed to simple recall-based examinations, a performance assessment requires a student to actively utilize their understanding to perform a task.

Concept Maps The limitation of recall-based questions is that they leave facts isolated, which runs contrary to the heavily linked understanding of an expert. A concept map, which relates ideas and methods with each other, forcing students to structure their knowledge in a way that mimics experts mental graph.

These four aspects – context, content, facilitations, and assessment – are complex and multi-faceted. Still, they all build off the tension that arises when trying to authenticate learning environments. Before moving on to discuss the design of programming environments within this framework, it is worthwhile to talk about the phenomenon of *preauthentication*: attempting to design for authenticity without sufficient knowledge of the audience. Petraglia (1998, pg. 58) gives a compelling example:

> The task of balancing a checkbook, for instance, may be an authentic task from the perspective of a 21-year-old, but we would question its authenticity from the perspective of a 5-year-old. But more to the point, even among 21-year-olds, for whom we believe the task should be authentic, there are some who will find any given lesson in personal finance irrelevant, inaccurate, or otherwise inappropriate.

Preauthentication stems from over-generalizations and run-away assumptions; if you attempt to reduce potential learners to a list of likes and dislikes, you are running a serious risk of ignoring each individual learner's rich history and background that they will be building off. It is difficult to plan for and work against this ever-present danger when designing virtual environments. You can only design the environment to be so flexible and be adaptable to so many different learners; over-developing one aspect can come at the detriment of the development of another component. Petraglia (1998) recommends that rather than attempting to design around students prior understanding, it is better to simply convince the learner of the authenticity of the environment. Although a notable strategy, this can be limiting, since it completely ignores the potential a student offers. Ultimately, there is no simple answer to this problem, and designers must simply keep the problem in mind, and build with flexibilty in mind.

Programming Environments

Programming is a purely digital activity performed with the goal of solving a seemingly-infinite number of computationally-susceptible problems, such as summing large lists of data, processing and rendering images, or creating complex interactive media. Programmers write instructions to the computer that will be executed on command, instructions that come in a variety of forms called *languages.* These sequences of instructions – the eponymous *programs* – were originally hard-wired into the machinery of a computer, but are now created using programs called *programming environments.* Today, these environments are arguably the primary tools of a developer, and are where both experts and novices spend the majority of their time. The learning environment of a programmer, therefore, are these programming environments.

It is interesting to compare programming environments with Virtual Learning Environments. Although traditionally learning environments are physical, the past few decades have seen a rise in distance and blended learning through the use of Virtual Learning Environments – technological, online platforms designed for student learning. There are many kinds of environments, all of which can be used for different purposes, such as to teach a specific skill, to provide out-of-classroom lectures, to assess student knowledge, or to provide a place for students to collaborate. These environments are usually built on modern web technologies that enable real-time communication between students, classmates, and teachers - for instance, a student might chat with another student about how to solve a complicated problem, or a teacher might have an interactive lecture at-a-distance. Of course, these environments can also be run asynchronously to allow students to move at their own pace. Programming Environments are virtual in the technical sense, since they are real tools that expert programmers use, but they strongly overlap.

In the remainder of this paper, we discuss the design of "Learner Programming Environments" based on the theory of Situated Learning and within the framework offered by Choi & Hannafin (1995) (i.e., in regards to context, content, facilitations, and assessment). We will pay particular attention to the issue of authenticity as it arises naturally in the design process. The idea of

The Context of LPEs

Context is the surrounding culture of the environment, and requires the developer to answer the questions, “what is the purpose of the environment, and who is using it?" In this case, the purpose is to learn how to program in order to solve computationally-susceptible problems. Programming is an extremely complicated task that requires skills in problem-solving, logic, and a host of other areas. The second question, however, is a little easier to pin down in a general sense.

Historically, programming was the craft of Computer Scientists and Software Engineers. Their work could almost be described as problem-solving for the sake of problem-solving, given how abstract it is. As a consequence of the virtual world that they work in, many were interested in improving the systems that they developed on (e.g., the operating system, the internet, the programming environments themselves) in a bootstrapping style. The major outreach was with fields that were strongly connected: electrical engineering helped improve the physical existence that programs rely on, while mathematics helped to strengthen the formalization and theoretical underpinnings of the field.

However, this trend is changing rapidly as computing takes a more dominant role in society. Building on the interesting interdisciplinary successes found when Computer Science was synthesized with subjects such as Biology, a seminal paper by Wing (2006) kickstarted a movement calling for universal education of all students on the principles of Computational Thinking, regardless of age and primary discipline:

> Computational Thinking is a fundamental skill for everyone, not just computer scientists. To reading, writing, and arithmetic, we should add

> computational thinking to every child's analytical ability... Professors of Computer Science should teach a course called "Ways to Think Like a Computer Scientist" to college freshmen, making it available to non-majors, not just computer science majors.

The scope of this movement, extending across ages and disciplines, brought a new market for Computer Science Education: non-majors. They will not become Software Engineers, or do research in Computer Science. They will be interdisciplinary workers who leverage Computational Thinking in their own domain, making them distinctive from both communities. In fact, this shift means that their "Community of Practice" might not even exist yet. Establishing authenticity is a serious challenge, and there will not be a one-size-fits all solution – the dangers of pre-authenticating are even more rampant here than before. Still, some lines can be drawn: Computational Literature Analysis deal with a very different set of problems than Computational Geologists do.

Ultimately, we are faced with a diverse crowd of people who will use programming environments. Some will end up using programming on a day-to-day basis and are firmly entrenched in the computer science community, but some will only use programming as a minor tool in a vast and diverse toolkit. The LPE can support the audience – the programming environment R[1], for instance, offers powerful tools for statistical analysis for mathematicians and beginner programmers who require such tools for data analysis. The LPE can also ignore the context of the programmer – the LPE Scratch[2], created for children, is intended to teach programming but not intended for anything besides creating interactive media. The expectation behind Scratch is that is more useful to appeal to the situational interest (colorful graphics, pretty animations, and exciting sounds) rather than the long-term interests (being able to do complex calculations). The tension between situational and long-term interest seems to correlate with age; older, more mature students

[1]`www.r-project.org`

[2]`http://scratch.mit.edu`

would most likely find Scratch to be fun but inappropriate.

Another dimension for analyzing context is formality, which has extreme range when it comes to programming environments. Some programmers start off working on problems in their free-time (informally), some might learn in high school or university settings (formally), and now there is a growing movement for free, Massively-Online Open Classrooms and local, volunteer classes (nonformal). Formal LPEs are usually designed with lecture and laboratory interactions in mind; the Pythy[3] environment supports this process by enabling code to be shared rapidly from the instructor to students. Still, in the context of formal environments, most LPEs completely ignore the possibility of being used in a classroom environment, forcing the instructor to rely on third-party modules or software (e.g., a Content-Management System) to interface with students. From this, it can be argued that most programming environments are oriented around informal environments.

Content in LPEs

The next aspect of design is content. What should beginner programmers learn? The biggest answer is computational problem-solving. Although the exact kind of problem to be solved depends heavily on the context of the learner, the environment is always in support of the generic goal. The task of problem solving in computer science is usually viewed as the transformation of data in one structure to another through the use of programs (indeed, a seminal text in Computer Science Education by Wirth (1978) was titled “Data Structures + Algorithms = Programs”) - for example, writing a program to calculate average temperature requires lists of temperatures to be transformed into a single number.

If learning how to program is viewed as the end-goal of a LPE, then environments like Scratch that offer tools to scaffold the process are more suitable for beginners. However, for the computational thinking programmers, the problems need to be given just as much

[3]`https://pythy.cs.vt.edu`

weight. So what would it look like for a programming environment to offer structured problems alongside algorithm development? One model for this is Codeacademy[4], an online interactive platform for learning a variety of popular, textual languages. Students begin by choosing a language, and then stepping through a structured curriculum. As they learn syntactical features of a language, they are given a situated problem to solve. For instance, they learn about functions in the context of calculating travel package costs (how much it costs to book a hotel and air travel with tax). This goes beyond the assignments typically provided to students, because it integrates directly into the programming environment. Taken to an extreme, relevant data could be built into the environment; in the travel package example, there could be data from actual travel websites available through the program. This example highlights a weakness of relying on external problems instead of tight integration: students must either manually enter in such data (a painstaking process, with very little value) or learn the process of bringing in data themselves (an educationally valuable task, but beyond the skill level of most beginner students). This lack of integration means that a valuable opportunity to authenticate the environment is lost.

Facilitations in LPEs

The third aspect of design is facilitations, which is extremely relevant – A programming environment is built to provide facilitation to the act of programming. The ones designed with learners in mind often have a balance between how authentic they are compared to the general-purpose environments used by professionals. Of course, there is a wide range of programming environments, littered along a scale of how much facilitation they offer. At one end, there's the textual programming environments, the simplest of which are literally just text editors. On the other, there are more sophisticated environments that offer useful features for syntax checking and refactoring (massively restructuring a program using simple patterns, such as renaming all instances of a variable

[4] `http://www.codecademy.com`

or redefining a class heirarchy). These features are the cognitive tools of the environment, and are very useful for advanced programmers. However, some environments have such a dizzying array that they can be counter-productive to beginers, as reported in Boyd & Allevato (2012). A successful LPE should work to minimize cognitive load required to work in an environment, possibly Fading this out to the expected number of utilities.

Scaffolding can come in many forms within a beginner programming environment. For instance, the beginner environment Dr. Racket allows images to be imported and manipulated as first-class objects (like text or numbers), significantly decreasing the difficulty of working with them. Similar functionality is available in Scratch and Alice[5], both of which are Visual Programming Languages, which offer another powerful form of scaffolding. In a Visual Programming Language, programs are created by dragging and dropping puzzle piece-like code blocks together. It is impossible to write invalid programs in a VPL. Unfortunately, the graphical nature of VPLs quickly leads to a loss of screen real-estate, and makes professional VPLs are a distinct rarity. A third kind of scaffolding that is present in many teaching languages is an interactive debugger and code stepper – a tool designed to walk the programmer through the execution of the program to identify errors and mistakes. This is perhaps best demonstrated in the Interactive Python Tutor by Guo (2013), which visually shows the underlying program state as it walks through the program. These tools are not designed for production-sized environment, and collapse under the size of a large codebase; once again, these are only scaffolds.

Modelling, Coaching, and Collaboration (the social facilitations) are not common elements of a programming environment. Still, there is recent work by Carter (2012) to add these into a general programming environment through plugins, with the goal of creating a Social Programming Environment. They offer several features that facilitate these processes. For instance, the Chat View and the Code View allow students and teachers, or just students and other students, to show off their code and converse about it, aiding both

[5]`http://www.alice.org`

Coaching and Collaboration. However, the Code View is a read-only operation – teachers can not model a programming action. A social VPL would need to have first-class operations for sharing code snippets between two programmers, with a model that enables programmers to fork and even merge changes, similar to modern Version Control Systems.

Other environments offer less real-time interaction with students, such as Scratch, which has a strong informal community oriented around its "remixing" methodology. After a programmer has created a Scratch game or animation, they can upload it to the community website, where other programmers can download, modify, and re-share their work. Although very popular, this collaboration is severely limited because it only occurs after the development process. Students only build off the success of each other; they can't see the other students challenges, discuss with them during the development environment, or other useful partnership outcomes.

Assessment in LPEs

The final aspect of design is assessment. Expert programmers measure the correctness of their programs using Unit Tests – each individual unit of source code (such as a function or object, depending on the paradigm of the language) is given sample data along with the expected, correct output. These unit tests can be run constantly during development to assess the ongoing validity of the code. In some ways, these are simple measures of student success, and the test-driven development methodology has taken off to the point that one modern language, Pyret[6], requires that all functions be written alongside its unit tests before it will even compile.

And yet Unit Tests are limited measures of achievement in the view of Situated Learning, which advocates for more active measures such as performance assessments. They can be easily gamed by students if they are not carefully monitored, e.g., by providing weak test cases that are easy to pass even though the code is not truly fully

[6]`http://www.pyret.org`

functional. So instead, performance assessments could be arranged through the aforementioned social facilitations; by logging into a code view, instructors could challenge and observe students, possibly offering live, diagnostic feedback.

Most programming environments offer integration with their computers built-in directory structure, which is in some sense a rough, cluttered portfolio. Still, these typically only live on the students computer, rather than the instructors, requiring secondary software to manage the portfolio. Modern, purely online environments such as Pythy and Koding[7] both circumvent this problem, but ignore the underlying problem: portfolios should be curated galleries, not a complete collection of everything a student has ever typed. Before it would be suitable for assessment, a proper LPE based on SL theory would need to have a phase where students could earmark certain works for their portfolio before they are passed on to the instructor – this process of reviewing and earmarking work adds to the value of this assessment technique.

Conceptual Maps could be developed for commands in the environment – recognizing that loops, recursion, and conditional jumps are isomorphic, for instance, is a key element to transferring programming knowledge between languages. However, it's not necessarily clear that this should be an element of the environment itself. Assessment is an ongoing challenge, especially in any long-term context; not all of the techniques offered through this theory may fit.

Conclusion

As more and more students enter into Computer Science with more and more diverse contexts surrounding them, we must re-evaluate the fundamental assumptions that go into our tools. Modern Situated Learning theory gives us a starting place to reconsider changes. In this paper, we have leveraged frameworks built from this theory to consider tensions inherent in the design of one of the most important tools for any programmer, the

[7]`https://koding.com`

programming environment. Ultimately, there are many tensions to consider when designing a programming environment for learners, including how to authenticate the learning experience. Although we offer a number of potential solutions, further work is required to find a suitable balance.

References

Boyd, E., & Allevato, A. (2012). Streamlining project setup in eclipse for both time-constrained and large-scale assignments (abstract only). In *Proceedings of the 43rd acm technical symposium on computer science education* (pp. 667–667).

Carter, A. S. (2012). Supporting the virtual design studio through social programming environments. In *Proceedings of the ninth annual international conference on international computing education research* (pp. 157–158). New York, NY, USA: ACM. Retrieved from `http://doi.acm.org/10.1145/2361276.2361309` doi: 10.1145/2361276.2361309

Choi, J.-I., & Hannafin, M. (1995). Situated cognition and learning environments: Roles, structures, and implications for design. *Educational Technology Research and Development*, *43*(2), 53–69.

Guo, P. J. (2013). Online python tutor: Embeddable web-based program visualization for cs education. In *Proceeding of the 44th acm technical symposium on computer science education* (pp. 579–584). New York, NY, USA: ACM. doi: 10.1145/2445196.2445368

Heeter, C. (2005). *Situated learning for designers: Social, cognitive and situative framework.* Michigan State University.

Jones, B. D. (2009). Motivating students to engage in learning: The music model of academic motivation. *International Journal of Teaching and Learning in Higher Education*, *21*(2), 272–285.

Lave, J., & Wenger, E. (1991). *Situated learning: Legitimate peripheral participation.* Cambridge university press.

Petraglia, J. (1998). The real world on a short leash: The (mis) application of constructivism to the design of educational technology. *Educational Technology Research and Development*, *46*(3), 53–65.

Rogoff, B. E., & Lave, J. E. (1984). *Everyday cognition: Its development in social context.* Harvard University Press.

Sawyer, e., R. Keith. (2005). *The cambridge handbook of the learning sciences.* Cambridge University Press.

Wing, J. M. (2006). Computational thinking. *Communications of the ACM*, *49*(3), 33–35.

Wirth, N. (1978). *Algorithms + data structures = programs.* Upper Saddle River, NJ, USA: Prentice Hall PTR.

Appendix

Figure 1. Comparison between Situated and Traditional Learning

Approach 1: classroom *(decontextualized, inert)*	**Approach 2: authentic** *(situated in real-world problem)*
For example, go through the Photoshop reference manual, tool by tool, in alphabetical order, learning how each tool (line, paint, bucket, select, etc.) works including all possible optional settings.	For example, start with a visualization task you want to accomplish (such as, create a logo for a company.) Look up and learn only a few particular tools you realize you may need to use to accomplish the design.