Teaching Complex Software Engineering Concepts through Analogies

Pawan Saxena, Sanjay K. Singh
Amity Institute of Information Technology
Lucknow, U.P., India
{psaxena1,sksingh1}@amity.edu

Gopal Gupta
Computer Science Department
University of Texas at Dallas
Richardson, TX, USA
gupta@utdallas.edu

Abstract*— In the analogy-based learning method we map a concept that is being learned to a well-understood concept. An analogy is mainly useful when learners lack prior knowledge of the topic being learned. Software Engineering (SE) is a subject whose concepts tend to be highly abstract and therefore difficult for undergraduate students to understand. Analogy-based instruction can greatly reduce a student's burden of learning these abstract SE concepts. Role of Analogy in teaching SE topics has not been adequately explored. In this paper we discuss analogy-based instruction in software engineering and its advantages. Over the last decade we have developed analogies for many complex SE concepts and extensively used them in the classroom at our institution. We discuss these concepts and corresponding analogies and as an illustration discuss one of them in detail. We also present the evaluation of our analogy-based instruction method. Our results indicate that the analogies we have developed are quite effective in improving student learning outcomes.

Keywords*— analogy-based instruction, software engineering, improving learning outcomes

I. INTRODUCTION

Teaching Software Engineering (SE) and Computer Science (CS) concepts effectively to undergraduate students is a challenging task. Students find SE and CS concepts hard to understand for two primary reasons: (i) most students are not exposed to SE concepts during their K-12 education. Even if they learn to program in high school, they will not be exposed to concepts such as modular software design principles or how an operating system works; (ii) most SE/CS concepts are quite abstract, as almost everything is based on interpretation of data expressed in bits and bytes; students are not used to this type of abstract representations and thinking. In this paper we focus on teaching software engineering. We outline our efforts in communicating abstract SE concepts to undergraduate students in an effective manner based on use of analogies. We have employed analogy-based instruction in the classroom now for more than a decade for teaching a large number of core software engineering (and computer science) concepts. Analogy based teaching of SE maps abstract SE concepts (target concept) to concrete everyday concepts known to students (base concept) resulting in better understanding of the target concepts. The analogy may result in further inferences made by the student about the target concept that the student is trying to understand.

Analogies play a major role in "problem solving, decision making, argumentation, perception, generalization, creativity, invention, emotion, prediction, explanation, conceptualization and communication" [12]. Our focus is on using analogies for effectively communicating information pertaining to SE concepts. Analogy-based teaching has been proposed for communicating complex concepts in the past [5,6,8]. Our analogies for SE concepts follow the tenets of structure mapping theory [5]. The structure mapping theory describes psychological processes involved in learning from analogies. The theory describes how familiar knowledge about a base domain can be used to understand a less familiar or an unfamiliar idea in the target domain. A domain consists of objects, their properties, relationship and interaction between the objects, and properties of those interactions. Analogical reasoning then involves recognizing similar structure between the target and the base domains. The closer the correspondence between the domains, the stronger the analogy. For example, one can understand the flow of electrical current by mapping it to flow of water where electric current corresponds to water, pressure differential (that causes water flow) to potential difference (that causes electric current flow), narrowing of a pipe to increased resistance, etc.

With the structure alignment theory in mind, we have designed a large number of analogies for advanced SE and CS subject in the last 10+ years. We have observed that these analogies resulted in improved student learning outcomes. In particular, we have developed detailed analogies for these eight key advanced topics in CS: (i) Pipelining and Parallel Processing in Advanced Computer Architecture (ii) Process states in Operating Systems; (iii) Virtual Memory in OS; (iv) Chomsky's Hierarchy in Automata Theory; (v) Asymptotic Notations in Design and Analysis of Algorithms; (vi) Lists, list processing and stacks in data structures (vii) Design of CPU, Control Unit, and Cache in Computer Architecture; and, (viii) Macros, Compilers, OS in Systems Programming. For software engineering, we have developed analogies for almost every major topic, the main ones being explaining (i) the Waterfall software lifecycle model with a detailed analogy of constructing a house, (ii) Levels of testing in a software system with the analogy of physical testing of computer chips, (iii) Cohesion and coupling in software modules with the analogy of a multi-semester degree program at a University, and (iv) Categories of software maintenance and its analogy with maintaining a physical house. In this paper, we describe one of the analogies (cohesion and coupling in software modules) in detail. The analogies we have developed are quite extensive and deep, in contrast to simple, shallow analogies that have been developed to explain CS/SE concepts to non-specialists and lay persons [3]. Analogy based instruction leads to better learning outcomes as students now learn by building upon their existing knowledge of

everyday life. Our own extensive experience in the classroom confirms this.

II. Analogy based instruction in SE

Teaching CS and SE concepts to undergraduate students is a difficult task. A major problem is that most students are unfamiliar with these concepts as they have not been exposed to them in K-12 (pre-university level). Even if they have taken programming courses in high school, they would have not been exposed to CS and SE concepts such as those related to modular software design, software verification and testing, software maintenance, the waterfall model for software life cycle, etc. Most of the students encounter these concepts for the first time when they take CS/SE courses as part of their undergraduate curriculum. Good examples of such hard-to-understand concepts are the notion of process and process scheduling in operating systems, cache memory and memory hierarchy in computer architecture, and the Chomsky hierarchy in automata theory. In Software Engineering, these concepts pertain to modular software design, software system life cycle, software verification and testing, software maintenance, etc.

For instance, consider the topic of software maintenance. Software maintenance is a broad topic---software engineering programs may have a whole course dedicated to it---that includes error corrections, enhancement of capabilities, deletion of obsolete capabilities, and optimization. There are four major categories of software maintenance, which are: (i) **Corrective Maintenance**: refers to modifications initiated by defects in the software, a defect can result from design errors, logic errors and coding errors. (ii) **Adaptive Maintenance:** includes modifying the software to match changes in the ever-changing environment. (iii) **Perfective Maintenance:** means improving processing efficiency or performance or restructuring the software to improve changeability; perfective maintenance refers to enhancements, i.e., making the product better, faster, smaller, better documented, cleaner structured, with more functions or reports. (iv) **Preventive Maintenance**: refers to activities usually initiated from within the maintenance organization with the intention of making a program easier to understand and hence facilitating future maintenance work. This includes code restructuring, code optimization and documentation updating (after a series of quick fixes to software, the complexity of its source code can increase to an unmanageable level, thus justifying complete restructuring of the code).

Software maintenance can be explained very well through the equivalent analogy of maintaining a house. (i) After staying in the newly constructed house, residents realize that certain features do not exactly satisfy their stated requirement. Thus, the design needs to be changed, which is corrective maintenance. (ii) With changing environment in and around the constructed house say a few burglaries in the area may require additional wire fencing on boundary and CCTV installation, i.e., adaptive restructuring and reconstruction carried out. (iii) With new designs having newer features like rooftop swimming pools, additional gym and similar changes are required, this all comes in the category of perfective maintenance. (iv) Builders designing and constructing the house know about the future technology coming up and so they may foresee the future demands by client. For example, they may suggest to the client to install additional concealed cabling and wiring at the time of house construction (as it is not easy to add additional wiring later, if it is concealed wiring). This all comes in category of preventive maintenance.

Many of CS and SE concepts tend to be rather abstract in nature (software itself is quite an abstract entity), making them even harder for students to understand when they encounter them for the first time. A good pedagogical technique is to teach students by building upon what they already know. Due to abstract nature of CS/SE concepts, teaching them by building upon what students already know becomes quite challenging, as their knowledge of advanced CS/SE is minimal.

Research indicates that illustrating abstract concepts through concrete representations and drawing direct connections between them helps students understand abstract concepts better. If purely abstract terms are used, then students have difficulty in fully grasping the concept. At the same time, teaching a new concept in an exclusively concrete manner limits a student's ability to generalize and apply the concept in other contexts. In CS/SE, most of the concepts are themselves quite abstract to begin with as they are represented using bits and bytes (for example, the concept of a process in operating systems or the concept of module coupling and cohesion or inheritance hierarchy). There is really no obvious concrete version of these concepts. Concretization of CS/SE concepts, we believe, is best achieved by resorting to analogies. Thus, the difficulties in communicating CS/SE concepts to undergraduate CS students can be mitigated by employing an approach based on using analogies. These analogies employ everyday concepts that students are already familiar with and that they can easily relate to.

The benefit of using analogies, especially in education, is well established [5,6,8]. Use of analogies for teaching CS concepts has two benefits: (i) Abstract concepts become concretized, making it easier for students to understand them. Since in CS, even concrete concepts are quite abstract, resorting to analogies that map these abstract concepts to concrete concepts in other areas greatly helps in achieving better learning outcomes. Research indicates that understanding abstract concepts directly is significantly harder [10]. (ii) Analogy based teaching reinforces the principle of pedagogy that says that a new concept should be taught by building upon the foundation of existing knowledge that the student possesses. The abstract concept's analogy to an everyday concrete concept indeed allows students to understand new concepts more easily by building upon existing knowledge. Of course, the students must be familiar with the analogy used. For instance, a person who has owned or built a house may achieve better understanding of the concept of software maintenance through the house-building analogy. However, an analogy based on a concept that students may not have directly experienced but that is part of everyday life may still be useful if no better analogy can be found. Thus, the house building analogy for software maintenance described earlier may still be a reasonable analogy as even though most students may not own or may not have built a house themselves, they do live in one. Thus, they can imagine the process of building and maintaining a house: a far more concrete concept than the abstract process of building and maintaining software.

III. Related work

Analogical reasoning plays an outsized role in everyday life: in teaching, communication, and in research [2, 12]. In the classroom, analogical reasoning can be a catalyst for achieving excellent learning outcomes, yet it has not been widely employed in CS/SE education. Gentner has developed the structure mapping theory to describe the processes involved in analogical reasoning [5]. Our analogies are based on this structural mapping theory. Glynn has developed a theory of teaching with analogies and developed six-step process of teaching with this method [7]. The analogies we have designed for CS and SE concepts also conform to Glynn's methodology. We have made elaborate efforts to ensure that our analogies works for minutest of details and do not break down. Gadgil and Nokes [4] and Nokes and Van Lehn [9] showed that analogy supports collaborative learning, especially when conceptual understanding is essential. Ruef developed a method of teaching that uses analogies in the classroom to explain topics [11]. In this methodology, one always considers the question, "what does this topic remind you of?" The program is designed to build critical thinking skills through analogies. This method is applicable to simpler topics, perhaps at the K-12 level, and is hard to use for CS and SE topics. Alizadeh et al [1] have studied the role of analogies in the context of tutoring of students in computer science data structures. Using techniques such as linear regression analysis they established that the presence of analogy and specific dialogue-acts within the analogy episodes correlate well with improved learning outcomes.

We found some studies of use of analogies for teaching engineering subjects in our literature survey, however, we did not find any studies of use of analogies for teaching advanced CS and SE topics. Most uses of analogies for teaching CS and SE topics are rather simplistic [3, 14] (e.g., explaining the concept of an input to a program as an ingredient in cooking a dish). With respect to engineering, Strunz and Louie have used the analogy of data storage in computer engineering to teach the concept of electrical power storage to students [15], while Plapally has used psychological stress in humans as an analogy for illustrating stress-strain in metals to mechanical engineering majors [16]. Their evaluations show good outcomes. In contrast, our focus in this paper is to use analogies for teaching SE topics.

We next illustrate a detailed analogy for the concept of modularity in software design. Here we attempt to explain to the students how modularity is realized in software engineering. The breaking up of a software system into modules is quite complex, as it is desired that certain constraints are followed in cohesion and coupling of modules. There are various types of cohesion and coupling which exist among modules and between different types of modules [13]. A software should be designed with the consideration that there is maximum amount of cohesion within a module and minimum amount of coupling between different modules.

IV. Concept of Modularity in Software design

A modular system consists of well-defined, manageable units with well-defined interfaces among the units. Desirable properties of a modular system are: (i) Each module is a well-defined subsystem that is potentially useful in other applications; (ii) Each module has a single well-defined purpose; (iii) Modules can use other modules; (iv) Modules should be easier to use than to build; and (v) Modules should be simpler from outside than from the inside. Modularity enhances design clarity (which in turn eases implementation), debugging, testing, documenting and maintenance of the software product. A system is considered modular if it consists of discreet components so that each component can be implemented separately and a change to one component has minimal impact on other components. A question arises that to what extent we should modularize because as the number of modules grow, the effort associated with integrating (and then later maintaining) the modules increases even more. A software system cannot be made modular by simply chopping it up into a set of modules. Each module needs to support a well-defined abstraction and should have a clear interface through which it can interact with other modules. Thus, under-modularizing and over-modularizing of a software system should be avoided.

A. Module Coupling

Coupling is the measure of the degree of interdependence between modules. Two modules with high coupling are strongly interconnected and thus dependent on each other. Two modules with low coupling are not dependent on one another. "Loosely coupled" systems are made up of modules which are relatively independent." Highly coupled" systems share a great deal of dependance between modules, "Uncoupled" modules have no interconnections at all, they are completely independent. A good design will have low coupling. Thus, interfaces should be carefully specified in order to keep low levels of coupling. Coupling is measured by the number of interconnections between modules. For example, coupling increases as the number of calls between modules increases, or the amount of shared data increases. The hypothesis is that design with high coupling will have more errors. Loose coupling, on the other hand minimizes the interdependence amongst modules. This can be achieved in the following ways:

1. Controlling the number of parameters shared amongst modules.
2. Avoiding passing undesired data to calling module.
3. Maintaining parent/child relationship between calling and called modules.
4. Passing only data and not control information amongst modules.

B. Types of Coupling

There are following types of coupling: *content, common, external, control, stamp*, and *data*. The strength of coupling from lowest level of coupling (best) to highest level of coupling (worst) is as follows: Data Coupling (best) → Stamp Coupling → Control Coupling → External Coupling → Common Coupling → Content Coupling (worst). Thus, given two modules A and B, we can identify several ways in which they can be coupled:

Data Coupling: The dependency between module A and B is said to be data coupled if their dependency is based on the fact that they communicate only by passing data, otherwise they are

independent. By ensuring that modules communicate only by exchanging necessary data, module dependency is minimized.

Stamp Coupling: Stamp coupling occurs between module A and B when a complete data structure is passed from one module to another. If a module only needs a part of the data structure, the calling module should be passed just that part, not the complete data structure.

Control Coupling: Module A and B are said to be control coupled if they communicate by passing control information.

External Coupling: A form of coupling in which a module has a dependency to other module, external to the software being developed or to a particular type of hardware. This type of coupling is essentially related to the communication to external tools and devices.

Common Coupling: With common coupling, modules A and module B share data. Making a change to the common data means tracing back to all the modules which share that data to evaluate the effect of change.

Content Coupling: Content Coupling occurs when module A changes data of module B or when control is passed from one module to the middle of another.

C. Module Cohesion

Cohesion is a measure of the degree to which the elements of a module are functionally related. A strongly cohesive module implements functionality that is related to one feature of the solution and requires little or no interaction with other modules. Cohesion may be viewed as a glue that keeps the module together. Hence, an important design objective is to maximize the module cohesion and minimize the module coupling. There are 7 types or levels of cohesion, given a module that carries out operations X and Y, we can describe various forms of cohesion between X and Y: Functional Cohesion (best) → Sequential Cohesion → Communicational Cohesion → Procedural Cohesion → Temporal Cohesion → Logical Cohesion → Coincidental Cohesion (worst).

Functional Cohesion: Operations X and Y are part of a single functional task. This is very good reason for them to be contained in the same module.

Sequential Cohesion: Operation X outputs some data which forms the input to operation Y. This is the reason for them to be contained in the same module.

Communicational Cohesion: Operations X and Y operate on the same input data or contribute towards the same output data. This is okay, but we might consider making them separate modules.

Procedural Cohesion: Operations X and Y are structured in the same way. This is a poor reason for putting them in the same procedure. Thus, procedure cohesion occurs in modules whose instructions although accomplish different tasks yet have been combined because there is a specific order in which the tasks are to be completed.

Temporal Cohesion: X and Y both must perform around the same time. So, module exhibits temporal cohesion when it contains tasks that are related by the fact that all tasks must be executed within the same time span. This is not a good reason to put them in the same procedure.

Logical Cohesion: X and Y perform logically similar operations. Therefore, logical cohesion occurs in modules that contain instructions that appear to be related because they fall into the same logical class of functions. Considerable duplication can happen between X and Y.

Coincidental Cohesion: X and Y have no conceptual relationship other than shared code. Hence, coincidental cohesion exists in modules that contain instructions that have little or no relationship to one another. That is, instead of creating two separate modules, only one module is made with two unrelated components.

V. ANALOGY FOR MODULARITY

We have found that the concept of coupling and cohesion in software design is very hard for students to understand. This is primarily because they are still sophomore or junior students and have not been engaged in carrying out a large software design and implementation project where modular design is of utmost importance and in which module coupling and cohesion become critical design issues. To get these concepts across, thus, we resorted to using analogy-based instruction. We have found that our analogy-based method for teaching concepts of modular software design has been quite effective, and our experimental evaluation indeed confirms this. We use a degree program, B. Tech. in CS for example, to illustrate the ideas. Since all students are in the program, they can easily relate with the analogy. In our analogy, a degree program design corresponds to a software system design. While the analogy may not be perfect at times, it serves as an excellent basis for students to understand the concept, as our qualitative and quantitative results indicate. The first author has used this analogy in teaching the concept of Modular Design of Software for the last 10+ years with excellent results.

Let us consider that a department (say, a Computer Science Department) of a university designs a B-Tech (CS) program. This is equivalent to a software system being designed. The entire program is designed to last eight semesters (modules) starting from beginning semesters to intermediate ones and further onto final year semesters. Different courses in a semester (module) are equivalent to *submodules* or *functions* in a module. Now, the theory in "concept of modularity in software design" is analogous to above written concept starting from the definition of modularity.

Modularity enhances design: an eight semester [module] program has a proper structure and design, clarity (eight modules provide ease of implementation), debugging, testing, documenting (class tests, continuous internal assessment, final exams, and grade cards). The mapping between various concepts in our analogy is shown below.

SW MODULARITY CONCEPT	ANALOGICAL CONCEPT
Software system	Degree Program (e.g., B.Tech. CS)
Module	Semester
Submodule/Function/Procedure	Course (or Subject)
Parameter or Data	Concept or a Subject Topic
Design Assessment	Continuous Internal Assessment
Design Refinement	Program Updates and Changes
Software Debugging and Testing	Class Tests & Exams
SW Designers	Faculty & Administration
Users of the Software	Students in the Program

Next, we explain how the concepts of coupling and cohesion map to our semester and courses analogy.

A. Module Coupling

Current semester's subject (submodule) topics are needed to understand the topics of a subject (submodule) taught in future semesters. Similarly, a current semester subject requires a student to have deep knowledge of earlier semester subjects. In such a case, we would say that semesters (modules) are strongly coupled (less desired). If there is less dependence on prior subject knowledge, then the semesters are loosely coupled. Module coupling requires "Controlling the number of parameters amongst modules." Knowledge covered in each broad topic in a course (e.g., software testing in a software engineering course) is equivalent to a parameter. If too many concepts of a semester are required in other semesters, we can say that parameter passing is uncontrolled which should be minimized in a modular (semester) system and is obtained by designing the syllabi of courses with greater care. As an example, if we are designing a course in AI, then students must know one of the symbolic languages (Prolog or Lisp), and suppose we choose to teach Prolog. In such a case, it would be more advisable to include the teaching of Prolog in the AI course itself, rather than in a separate earlier course (say, the Programming Languages course). In the latter case, the AI and Programming Languages course will be strongly coupled, which is less desired. Better to keep the coupling between the AI course and the Programming Languages (PL) course weak. Now, if for inevitable reasons, we are not able to accommodate the teaching of Prolog in the AI course (due to there being too many topics to cover), then it's fine to teach Prolog in the PL course, but it should be avoided if possible. Now, the PL course will have to be taught earlier than the AI course, and there will be a dependence of AI on PL.

Following the module coupling principle, we should design our courses in such a way that they are 'self-contained' as much as possible, meaning the fewer pre-requisites a course has, the more flexibility students will have in taking them. This can have impact on graduation rates and, hence, student satisfaction. This is analogous to software module design, where the fewer dependencies there are between modules, the easier it is to maintain and debug them, resulting in better programmer productivity. There will be fewer issues with the use of software system, and user satisfaction will be higher. To push our analogy further:

1. Avoid passing undesired data to calling module: relevant concepts of earlier semester's subject can be recalled in teaching current semester subject, not the ones that are irrelevant or peripherally relevant. This means that there should be significant dependence between courses to declare one a pre-requisite of the other.
2. Maintain parent/child relationship between calling and called modules: initial semesters are like parent (calling module) and future semesters are like child (called module); knowledge of concepts of initial semesters is passed to later semesters.
3. Pass data but not control information: subject knowledge (data) of previous semester (module) can be recalled in coming semesters but faculty member who taught the subject in the prior semester should not be called to clarify previous semester's knowledge in the current semester (hence "control" is not transferred).

For two submodules A and B (equivalent to topic A and topic B covered in two different courses of the same semester or of different semesters) we can identify several ways in which they can be coupled:

Data Coupling: If a subject in a semester (module A) uses contents of previous semester (module B), then this can be considered as data coupling between semesters, otherwise the semesters (modules) are independent. Semesters (modules) communicate only necessary data (previous semester topics/content) and nothing else; module dependency is minimized.

Stamp Coupling: In stamp coupling full data structure is passed between modules, equivalent to full content of a previous semester's subject. That is, all topics of a course are required in current semester (module). This is undesirable as modules should communicate by passing only necessary data. This corresponds to our earlier example of teaching Prolog in the PL course and using that knowledge in the AI course.

Control Coupling: Module A and B are said to be control coupled if they communicate by passing control information. In our analogy we consider that instead of reviewing the knowledge needed from prior semesters in a given course, we require that the professor of previous semester's subject comes and reviews the topics needed in the current course. Thus "control" (professor teaching the course) is "passed" from previous semesters to the next. Control coupling is the most undesired form of coupling for modules. Requiring that an instructor from a previous course come and assist in the teaching of the current course is highly undesirable (and inefficient).

External Coupling: A submodule A (subject A) is externally coupled with submodule B (subject B) if subject B is from some other program (module) altogether. For example, in B-Tech CS, the subject of Data Analytics requires the knowledge of another subject, say, Probability and Statistics from the Mathematics department, then we say module A is externally coupled with

module B. This coupling is not desired. From a degree program design perspective, such an external dependency can potentially create a problem because the CS Department does not have any control on how often the Probability and Statistics course is offered, and thus CS students may be affected. An alternative is for the CS Department to teach its own course in Probability and Statistics to avoid this external coupling.

Common Coupling: In Common Coupling a submodule A (course A) and a submodule B (course B) of different modules (semesters) have a particular topic in common. Suppose a new concept for this topic is developed and covered in course A, then this new concept should be covered in all courses (submodules) that contain this topic, including course B. Common coupling is not desired. Similarly, a topic should be covered in only one course. Covering the same topic in two different courses is not advisable. It is better to have the course where this topic is also needed have the other course as its pre-requisite. Thus, either course A should be a pre-requisite of course B, or *vice versa*. Similarly, data should not be shared among modules, it should be put in a single module, say, M1, and the other module, say, M2, should access/manipulate it through the interface provided by M1.

Content Coupling: In content coupling submodule A changes topic/content in submodule B. Suppose some concept, code, or algorithm, developed in a subject A, is implemented in other related subject B as well. For example, ideally, the design and analysis of Dijkstra's shortest path algorithm as well as its implementation should be taught in a single course. It would be ill-advised to split the teaching of design and analysis of the algorithm in one course and its implementation in another. Content coupling is an undesired form of coupling, considered to be worst coupling, however, sometimes it is unavoidable, in a manner similar to, where, in many computer science curricula, the algorithms course will mainly focus on design and analysis, and not algorithms' practical implementation (primarily, because there is not enough time in the course).

B. Module Cohesion

Cohesion is the measure of the degree to which the elements (submodules/subjects) of a module (semester) are functionally related. A strongly cohesive module implements functionalities that are interrelated and leads to a focused solution and is not related to features of other modules.

Functional Cohesion: Submodule X (course X) and submodule Y (course Y) form the part of a single functional task, then they should be contained in same module (semester). For example, a subject A (submodule A) and its laboratory classes (submodule B) are part of a course so that a student can become proficient in that topic. Hence, the course and its laboratory counterpart should be organized in the same semester (module). This is a desired form of (functional) cohesion in software module design.

Sequential Cohesion: Function A outputs some data which forms the input to function B, this is a sufficient reason for them to be in the same course/subject (submodule). For example, the concept of *syntax analysis* is needed to understand *syntax driven code generation* in the Compiler Design course. So, obviously, the two must be included in the same course (on Compiler Design), as they usually are in most curricula. It's possible to teach *syntax analysis* in the Automata Theory course, but that would be less desirable.

Communicational Cohesion: Submodules X and Submodule Y operate on the same input data (topic/concept) or contribute towards same output data, then submodules X and Y are to be placed in same module. The knowledge of Unix Environment is needed for both the Operating Systems course as well as the Computer Network course. Both these courses (OS and Computer Networks) could be taught in the same semester (module), although as the theory of Communicational Cohesion says, these submodules (the OS and the Computer Networks course) can be placed in separate modules (semesters) as well.

Procedural Cohesion: Submodule X (course X) and Submodule Y (course Y) are structured in the same way and that is why they should be in the same module (semester). Now suppose two subjects X and Y have the same syllabus structure (same number of lectures on theory part of the course, same number of laboratory classes, assignments, and case studies, and the order in which different tasks are to be executed is also the same), then they are placed in the same module (semester). As is obvious, this is not a good reason for the two different submodules (courses) to be placed in the same module (semester).

Temporal Cohesion: Submodule X (course X) and Submodule Y (course Y) have tasks which are to be done within the same time span, then we say that temporal cohesion exists. For example, the two separate courses (submodules) on Machine Learning and Data Science could be done in parallel in the same semester (module), as they both involve many overlapping concepts. This is okay but it should not be a compulsion to put them in the same module (semester). The course on Machine Learning can be studied in one semester and Data Science can be studies in another semester. Hence temporal cohesion may exist, but it is not compulsory to achieve

Logical Cohesion: Submodule X and Submodule Y perform logically similar operations and hence are placed in same module (semester). For example, the subject of Mathematical Logic (Propositional logic) and subject of Logic Circuit Design have similar operations/topics (AND, OR, NOT, X-NOR, CNF, DNF, etc.) but this is not a good reason for placing the two subjects (submodules) in the same module. Hence logical cohesion may exist but is normally not desired.

Coincidental Cohesion: Two unrelated submodules (two random courses) are placed in a module (semester). Ideally, each semester (module) must have only related courses (submodules). Coincidental cohesion is the most undesired type of cohesion. However, sometimes it may be unavoidable. In case of the degree program, due to the restriction of there not being more than eight semesters in the program, some unrelated courses (submodules) have to be put in the same semester (module).

VI. Evaluation

The first author has been teaching CS concepts using analogy-based methods for the last 10+ years. Our preliminary research showed that analogy-based instruction is indeed effective for teaching CS subjects [17]. We want to establish that they are effective for SE subjects as well. In fact, the various analogies mentioned earlier have been developed through considerable investment of effort once the benefit of teaching CS students via analogies was realized. We invested effort in developing analogies that were realistic and structurally strongly aligned. These analogies have been refined over several years. Our observation and evaluation confirm that analogy-based instruction produces excellent learning outcomes. Our conversations with the students indicate that students prefer to register for classes where analogical methods are used.

A. First Experiment and Analysis

In our first experiment, we took a cohort of 102 undergraduate information technology majors at another institution who were in the second year of their degree program and who were taking their first required software engineering course. Permission of the Information Technology Department Head was obtained. The 102-student cohort consisted of *all* the students taking the required 2nd year software engineering course. The first author took over the class when the time came to teach modular software design. These students were divided into two (almost) equal parts: odd roll numbers in one section and even roll numbers in the other (the roll numbers of student cohorts in a major are assigned consecutive numbers at the institution). Both groups were taught Principles of Modular Software Design by the first author. One half of the students (n = 50: 39 males, 11 females) were taught using the analogy described in this paper, and the second half (n = 52: 40 males, 12 females) were taught without the use of analogies. After finishing the unit on modular software design, the two groups were given a 20-question quiz consisting of objective type questions. Both sections were given the same quiz that was designed by the instructor teaching that course (recall that the first author only taught these students the topic of modular software design). The quiz was given the following day. Maximum points that could be earned were 20 (1 point per question). The quiz was developed by the course instructor who designed the quiz, evaluated it, and emailed the results to the first author.

A few sample questions used in the quiz (Q1, Q7, Q14 and Q17) are shown below.

Q1. The most desirable form of Coupling is
- (a) Control Coupling
- (b) Data Coupling
- (c) Common Coupling
- (d) Content Coupling

Q7. Functional cohesion means
- (a) Operations are part of single functional task and placed in same procedure.
- (b) Operations are part of single functional task and placed in multiple procedures.
- (c) Operations are part of Multiple tasks.
- (d) None of the above

Q14. Independence of module is assessed using two qualitative criteria. What are those criteria?
- (a) Cohesion and Coupling
- (b) Module and Modularity
- (c) Cyclomatic complexity and modularity
- (d) None of the above

Q17. What is the typical relationship between Coupling and Cohesion?
- (a) There is no relationship between coupling & cohesion.
- (b) As cohesion increases, coupling increases
- (c) As cohesion increases, coupling decreases
- (d) Relationship between coupling and cohesion is difficult to define

	Group A (with Analogy)	Group B (w/o Analogy)
n	50	52
Mean	16.60	14.57
S.D.	1.55	1.82
Median	17	14
Mode	18	13

Table 1: Statistics for marks earned in 20-question quiz (Max Score = 20)

Mean, Standard Deviation, Median and Mode of students' quiz scores are given in Table 1. Students who were taught with analogy averaged full 2 points above those who were not taught with analogy; a 10% improvement in score using analogy-based instruction. The standard deviation of the analogy-taught group was also lower, signifying that their performance clustered around the higher average while higher standard deviation of the non-analogy-taught group indicated that their scores were more divergent. Similarly, the median for the analogy-taught group was 17, while that of the non-analogy-taught group was 14, and the mode was 18 and 13, respectively. The two groups are very similar in composition, as they were created by us based on the odd and even roll numbers. Students are assigned roll numbers upon admission randomly, so as far as we can assess, there is no bias in assigning students into the two groups. Student demographics for an entering cohort at this university are quite similar as they mostly come from middle class families in a provincial state capital in the country.

Next, we need to establish that the observed differences are significant. To confirm that the separation of the mean values of the two groups by 2.03 points is not due to random chance, we conducted the two-sample t-test. First, we checked that the distribution of student scores in the two groups follows the normal distribution assumption for the two-sample t-test. This was indeed confirmed as can be seen in Figure 1. The curves were obtained using the R software. Next, we computed the two-sample t-test using the values for mean and standard deviation shown in Table 1. The two-sample t-test value was computed to be 6.8758 with a degree of freedom 100 and p value of 5.4×10^{-10} ($t = 6.8758$, $df = 100$, $p = 5.4 \times 10^{-10}$). The extremely low p value demonstrates the very low statistical odds of the two groups having the same mean score. Our

experiment, thus, demonstrates that teaching a subject using analogies had material impact on students' exam performance.

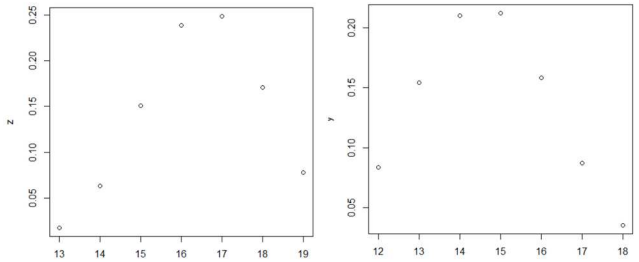

Figure 1: Distribution of Student Scores

An informal validity study was also conducted. **Content validity** (namely, test content is relevant) holds as the topics we covered with and without analogy in the two groups are narrow (modular software design), the quiz administered covers the subject matter taught, and the quiz scores were used as a measure of understanding the subject. **Face validity** (namely, test appears to measure what it claims to measure) is established from the fact that the test relies on measuring the understanding of a topic by administering a quiz. The researchers involved in the experiment agreed that the quiz that was given to the two groups reasonably measured students' understanding of the topic. Further, **construct validity** follows: given that the quiz directly assesses the knowledge of the topics covered, using quiz scores as a measure of understanding is reasonable.

Our results above clearly indicate that teaching software engineering concepts using analogies leads to better understanding and greater retention of the subject matter by the students. It results in better learning outcomes, as our experiment illustrates. As mentioned earlier, the first author has been using analogies including the analogy for modular software design described in this paper and has qualitatively observed the benefits of this analogy-based approach. Very frequently the students will send an email to the instructor (the first author) after the final semester exams, commenting to him that the analogy that was taught to them helped them answer some specific questions in the exam better.

B. Second Experiment and Analysis

We have also evaluated our analogy-based teaching methods through feedback from students. Through this instrument we wanted to gauge student sentiment about the use of analogy. Our hypothesis was that students would prefer analogy-based method and find it more engaging. We sent a questionnaire to all 50 students who were taught the topic of Modular Software Design with analogy (it does not make sense to survey the other group). The survey was sent after the quiz had been administered. Thirty-eight students of the fifty responded. The questions are reproduced below.

1. Q1: The B. Tech. CS Degree Program Design analogy helped you understand the topic of **Modular Software Design** better.
2. Q2: The use of analogy for explaining the concept of **Module Coupling** helped you in understanding it better.
3. Q3: The use of analogy for explaining the concept of **Module Cohesion** helped you in understanding it better.
4. Q4: You found topics for which an analogy was given *more engaging*.
5. Q5: You found topics for which an analogy was NOT used, *less engaging*.
6. Q6: You would like as many topics as possible in Software Engineering to be taught by giving concrete analogies.

	Q1	Q2	Q3	Q4	Q5	Q6
Mean	3.95	4.39	4.24	4.16	4.24	4.16
Median	4	5	4	4	4	4
Mode	4	5	4	5	5	4

Table 2: Summary of Student Responses

Response designed in Likert scale (Strongly Disagree = 1, Disagree = 2, Neutral = 3, Agree = 4, Strongly Agree = 5).

Table 2 shows the mean, median and mode for the six questions for the 38 responses returned. Clearly, students found analogies from everyday life for teaching advanced SE concepts (modular software design) to be quite useful for understanding (Q1, Q2, Q3). Students showed clear preference for analogy-based instruction (Q4, Q5, Q6) as they found the discourse to be more engaging. These results are consistent with qualitative feedback from students, who expressed considerable appreciation for being able to understand a complex topic taught via an analogy. These results confirm our hypothesis. The most important observation is that use of analogies increased *student engagement*. Increased student engagement is the key ingredient for improving student learning outcome. Learning outcomes were clearly improved as our experiment and measurements show (Section VI.A).

VII. Conclusions and Future Work

We reported on our experience teaching advanced software engineering concepts using analogies. We listed some of the analogies that we have developed and used over the last 10+ years for teaching core software engineering concepts at our institution. We explained one of our analogies in detail. Our experimental results quantitatively establish that analogy-based instruction leads to improved student learning outcomes and increased student satisfaction. We plan to continue developing analogies for more software engineering topics, as well as refine the ones that we have already developed. We plan to publish a compendium of these analogies soon.

Acknowledgement: We are grateful to Prof. Laxmi Shankar Awasthi, Dean of Academics, Lucknow Public College of Professional Studies, for permitting us to teach software engineering classes at his institution. We sincerely thank Dr. Santosh Kumar, Associate Prof., LPCPS, for help and cooperation, and Dr. Ajay Pratap, Assistant Professor, Amity University, Lucknow, for his help with statistical analysis. We also thank the anonymous referees for insightful comments.

[1] Alizadeh, M., Di Eugenio, B., Harsley, R, Green, N. A, Fossati, D., and Omar, A. (2015). Study of Analogy in Computer Science Tutorial Dialogs. In Proc. CSEDU conference. pp 1-6.

[2] Bartha, Paul (2019). Analogy and Analogical Reasoning. The Stanford Encyclopedia of Philosophy (Spring 2019 Edition), Edward N. Zalta (ed.).

[3] Cheo, C. (2022). Forty Key CS Concepts Explained in Layman's Terms. http://carlcheo.com/compsci. Accessed March 2022.

[4] Gadgil, S. and Nokes, T. (2009). Analogical scaffolding in collaborative learning. In Proc. annual meeting of the Cognitive Science Society, Amsterdam, The Netherlands

[5] Gentner, D. (1983). Structure-Mapping: A Theoretical Framework for Analogy. Cognitive Science. 7 (2): 155–170.

[6] Gentner, D. and Colhoun, J. (2010). Analogical processes in human thinking and learning. In Towards a theory of thinking, pages 35–48. Springer, Berlin, Heidelberg.

[7] Glynn, S. (1994). Teaching Science with Analogies: A Strategy for Teachers and Textbook Authors. Nat. Reading Res. Ctr. Research Rep.15. https://eric.ed.gov/?id=ED373306

[8] Hofstadter, D. R. (2001). Analogy as the core of cognition. The analogical mind: Perspectives from cognitive science, pages 499–538.

[9] Nokes, T. J. and Van Lehn, K. (2008). Bridging principles and examples through analogy and explanation. In Proceedings of the 8th international conference for the learning sciences, Volume 3, pages 100–102. International Society of the Learning Sciences.

[10] Newby, T.J., Stepich, D.A. (1987). Learning abstract concepts: The use of analogies as a mediational strategy. Journal of Instructional Development 10, 20–26.

[11] Ruef, K. (2011). The Private Eye. Accessed Jan., 2021. http://www.the-private-eye.com/

[12] Wikipedia contributors. (2021). Analogy. In Wikipedia, The Free Encyclopedia. Retrieved Jan., 2021 from https://en.wikipedia.org/w/index.php?title=Analogy.

[13] Wikipedia contributors. (2021). Coupling. In Wikipedia. The Free Encyclopedia. Retrieved Jan., '21 https://en.wikipedia.org/wiki/Coupling_(computer_programming).

[14] Jiménez Toledo, J.A., Collazos, C.A., Ortega, M. Discovery Model Based on Analogies for Teaching Computer Programming. In Mathematics 2021, 9, 1354.

[15] Strunz, K. and Louie, H. Cache Energy Control for Storage: Power System Integration and Education Based on Analogies Derived From Computer Engineering. In IEEE Transactions on Power Systems, vol. 24, no. 1, pp. 12-19, Feb. 2009.

[16] Plappally, A. The effect of joint role of creative analogy and concept-in-context map on the learning interest and performance of first year mechanical engineering undergraduates. In Proceedings of the 2016 IEEE 8th International Conference on Engineering Education (ICEED), 2016; pp. 126–130.

[17] Saxena, P., Singh, S.K., and Gupta, G. Analogy-based Instruction for Effective Teaching of Abstract Concepts in Computer Science. In Proc. 7th International conference in Higher Education Advances (HEAd'21), Valencia, Spain. 2021. pp. 377-385