CHAPTER 4: SOFTWARE DEVELOPERS’ EDUCATION FOR DEVELOPMENT OF BASIC COMPETENCIES

BusinessDictionary.com defines competence as “knowledge of, and skill in the exercise of, practices required for successful accomplishment of a business, job, or task.” Competence is manifested as performance. In 1980’s Rasmussen, Reason, and Norman elaborated upon a three level hierarchy of human performance: skill based, rule based, and knowledge based. As per this model, the lowest level of human performance is skill-based at which the behavior is controlled by a stored sequence of action in space and time. Expert programmers can create low level programming constructs without conscious engagement, whereas novices have to think about such compositions [157]. The expert programmers can make skill-based errors in routine actions because of intrinsic factors of inattention or over-attention. The middle level of human performance is rule-based at which the behavior is controlled by stored if-then rules. The highest level of human performance is knowledge-based at which the behavior is controlled by deliberate logical and analytical reasoning. This behavior is invoked by beginners who start performing a task or by experienced persons who face a novel situation. The errors at this level occur either because of resource limitation of conscious mind or incomplete/incorrect knowledge.

For developing software, the developers have to engage themselves at all these three levels with respect to following five basic competency domains.

1. Technical Competence
2. Computational thinking competence
3. Domain Competence
4. Communication Competence
5. Complex Problem Solving Competence

Software developers’ performance is result of integration of skills-based aspects with rule-based and knowledge-based reasoning. Practice sharpens skills-based aspects of their professional tasks. Rich experiences with varied cases enrich their rule-based reasoning. Knowledge-based reasoning requires critical and reflective thinking. Under the enabling conditions created by
‘competency conditioning attitudes and perceptions,’ software developers use their ‘competency driver - habits of mind’ to acquire, integrate, apply, refine, and extend their competence, i.e., skill, rules, and knowledge in all these five competency domains.

**Section 4.1: Software Developers’ Education for Development of Technical Competence**

Technical competence of professionals is manifested in the practical as well as intuitive understanding required for the executing various technical tasks of a profession. Mosby's dental dictionary defines technical competence for dental care professional as “the ability of the practitioner, during the treatment phase of dental care and with respect to those procedures combining psychomotor and cognitive skills, consistently to provide services at a professionally acceptable level.” A professional’s technical competence requires a coherent and integrated understanding of factual, conceptual, and procedural knowledge in the subject area. It needs the ability to use tools, techniques, procedures, best practices, and standards to solve problems. Further, it requires an intuitive understanding of what is technically feasible, scalable, and reusable.

Engineering and design professionals need to understand the current state of the art, and emerging technologies. Further, they need to be able to use this understanding to assess tractability of the problems [158]. They need to have the patience and wisdom to consider a restricted subset of the problems, till the technology advances to a level where a solution of the original unconstrained problem can be attempted. As experience and technology matures, the focus shift shifts from short term goals to higher long term goals which expand to encompass an entire class of problems. It is imperative for them to have a good understanding of the limitations and risks associated with each piece of work.

**In our 2009 survey** on required competencies for software developers, twenty software professionals assigned ‘technical/domain competency’ and ‘analytical/design skills’ an average rating of 2.95 and 3.0 respectively on a scale of 0-4. A majority of these responses, 60% and 63% respectively recommended these to be critical or very important competencies.
With reference to Appendices A2 and A3, Technical competence of software developers also includes the following:

1. Ability to apply knowledge
2. Technical competence to solve the software solvable problems using tools and techniques
3. Use of open source software
4. Knowledge of industry’s best practices and standards
5. Appreciation of what is technically feasible
6. Ability to identify the risk level of each piece of work
7. Design skills
8. Numerical ability

Curriculum designers continuously face and address the challenge of identifying the required technical competencies suitable for their respective industries. However, often this process gets disengaged from the real continuously evolving industrial requirements. With reference to professional courses like engineering, it is critical to continuously collect required inputs from relevant industry and update the curriculum. The computing industry is evolving faster than the academic discipline of computing. There is a continuous complaint from the industry about severe shortage of well prepared graduates. The continuously evolving work profile of computing engineers is not appropriately reflected in the educational programs. Most of large software companies have their own education wings to train and retrain their developers. Typically large companies have mandatory technical training for their staff every year. The training programs are focused on several aspects like core technologies, development methodologies, project management, etc.

Based on a long industry-academia consultative process, SWEBOK [68] provided an excellent documentation of required technical competencies that software engineers with four years of experience should have. The SWEBOK report gives details about the ten knowledge areas related to software engineering: software requirements, design, construction, testing, maintenance, configuration management, engineering management, engineering process, tools and methods, and quality. With reference to different topics in these areas, Appendix D of the
SWEBOK report specifies the desired level of competence out of the six levels as Bloom’s taxonomy: Knowledge, Comprehension, Application, Analysis, Synthesis, and Evaluation.

**Important Technical Activities in Software Development**

We have attempted to further understand the full spectrum of required technical competencies from the perspective of real technical work profiles in the software industry. We first catalogued the various technical and technically oriented activities through a consultative process. *Sixteen professional engineers in the software industry with high quality and rich industry experience as well as strong academic background were consulted.* Collectively, these experts have a rich work experience of over 330 man-years in various activities of software development. We have grouped various technical activities under seven major categories. Appendix A5 catalogues all these technical and technically oriented activities related to software development activities. These categories relate to planning, design, realization, evaluation, and client interface activities. Our classification also includes two categories of ubiquitous and over-arching activities.

Planning activities relate to project and risk planning. The design related activities encompass diverse design activities at various stages and multiple levels. The design activities are followed by realization activities. Realization refers to the class of activities that relate to implementation and maintenance. Evaluation activities relate to selection and evaluation of tools, technology, products, and process. Many activities require intense interfacing with the client. These client interface activities relate to requirements and support.

Some activities are embedded within almost every function. These ubiquitous activities include process support activities that apply across all phases of a project. The activities in abovementioned six categories need the support of certain overarching activities that are apples companywide across the projects.

Based on this catalogue of activities, *we administered a survey among another group of software developers. Fifty-seven software professionals responded to our survey.* About 14% of these experts have more than fifteen years of experience in software development, 11% have worked for more than 10 years, 42% have more than five years experience, and 21% have more than two
years of work experience. They work in companies like IBM, Oracle, Cadence, EBSCO, TCS, HCL, Wipro, Mahindra Satyam, Bloomberg, LGSoft, Samsung, Deloitte Consulting, and CRIS, etc. The views of our respondents on the most important activities that must be included in the main goals for new curriculum for the future generation of software developers are given in Table 4.1.

**Table 4.1**: Most important activities that must be included in the main goals for a new software curriculum

<table>
<thead>
<tr>
<th>Activity</th>
<th>Percentage of Respondents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm/Computational Procedure/Component and Interface Design</td>
<td>79%</td>
</tr>
<tr>
<td>Application/Product/System Design/Prototyping</td>
<td>75%</td>
</tr>
<tr>
<td>Product/Requirement Definition and Specification/Requirement Engineering</td>
<td>75%</td>
</tr>
<tr>
<td>Code Analysis, Program Comprehension, Re-documentation</td>
<td>68%</td>
</tr>
<tr>
<td><strong>Innovation and research</strong></td>
<td>66%</td>
</tr>
<tr>
<td>Application, Component Development/System Integration</td>
<td>65%</td>
</tr>
<tr>
<td>Group work, people management, and leadership</td>
<td>63%</td>
</tr>
<tr>
<td>Estimation and Costing, Project Scheduling</td>
<td>60%</td>
</tr>
<tr>
<td>Product/Process Quality Assurance and Control</td>
<td>58%</td>
</tr>
<tr>
<td>Validation and Verification (Testing)</td>
<td>54%</td>
</tr>
<tr>
<td>Technical Documentation, Presenting Ideas and Insights</td>
<td>52%</td>
</tr>
<tr>
<td>Test Design</td>
<td>47%</td>
</tr>
<tr>
<td>User Interface Design</td>
<td>45%</td>
</tr>
<tr>
<td>User Acceptance, End-user Documentation, Deployment and Roll-out, Customer Support</td>
<td>42%</td>
</tr>
<tr>
<td>Security Architecture Design, Architecting, Component Selection</td>
<td>39%</td>
</tr>
<tr>
<td>Project Monitoring and Control</td>
<td>36%</td>
</tr>
<tr>
<td>Tools and Technology Selection and Evaluation</td>
<td>36%</td>
</tr>
<tr>
<td>Usability/Value/Impact Analysis</td>
<td>36%</td>
</tr>
<tr>
<td>Resource Planning and Management, Staffing and Team Development</td>
<td>36%</td>
</tr>
<tr>
<td>Risk Planning and Mitigation</td>
<td>36%</td>
</tr>
<tr>
<td>Build and Release, Configuration Management</td>
<td>36%</td>
</tr>
</tbody>
</table>

Many activities of Appendix A5 received the support of less than 35% professionals. We have not included such activities in Table 4.1. This list is part of our proposed framework of pedagogical engagements in software development education (Table 8.3, first column).

**Pedagogical Perspective**

In order to perform these activities, software developers need to have an integrated understanding that hardware and software are two extreme ends of the possible solution space, with the possibility of varying levels of interaction and exploitation between two. The correct identification of the system/environment boundary to define the system/environment interface is crucial to the ability to successfully define, design, and develop a functional system. This often requires a tradeoff in decomposition and allocation of functionality to hardware and software.
sub-systems. The need to define interfaces early is critical in order to support modularity, multi-
team development, and testability.

There is a distinction between a hierarchical decomposition for project management vis-à-vis
decomposition for driving development. The managers and the developers need to understand
this distinction, and also the corresponding mapping between the two views. They also need
clarity about the system-environment boundary, interface, system metrics, constraints, and
acceptance criteria.

Given the drastic reduction in the time to market a product, it is essential that the product is
brought to the market the earliest. This forces a designer/developer to exploit all means possible,
including third-party tools, libraries, and sub-systems. Today, software development does not
only require logic building ability, but also hugely depends upon system/platform knowledge.
Efficient development requires inclusion of available in-house source code, commercial off the
shelf components (COTS), and also open source software. Hence, good awareness and ability to
select, include, and modify, the available software components and subsystems into new systems
are now imperative for software developers.

They also need to have theoretical, practical, and intuitive understanding of the entire
programming stack that includes hardware (CPU, memory, cache, interrupts, microcode),
operating system APIs, binary code, assembly, static and dynamic linking, libraries, compilation,
interpretation, garbage collection, heap, stack, memory addressing, processes and threads,
understanding of space-time tradeoff, and data structures and algorithms.

They must have hands-on experience with at least two different instances of each of the
following: architectures, operating systems, programming languages, programming paradigms,
compilation systems, DBMSs, glue/scripting languages, IDEs and productivity tools like
profiling, testing, CASE tools, version control systems, etc.

They need to learn code optimization, performance tuning, defensive programming, assertions,
and mixed paradigm programming. The curriculum should address code organization within and
across files, source code tree organization, source code version control, build automation, deployment, and roll out. Creating awareness and appreciation of upcoming technologies and standards is strongly recommended as an agenda for curriculum designers for software development education.

**Pervasive Knowledge Areas**

Today, web and multimedia (including graphics) nearly have omnipresence in computing systems. Hence, the students must learn multimedia and graphics programming, including use of special APIs for the purpose. All computing students must be given some practice with web-database architecture and programming.

Embedded systems place special requirements on interfacing peripheral and communication protocols. Exposure to peripheral interfacing and communication protocols is highly recommended. Security has emerged as a big concern for users and a challenge for computing professionals. The education program must give good experience with secure programming and security APIs. Use of mobile phones as a computing platform is growing exponentially. Students must be exposed to developing software for at least one such platform.

Use of open source for developing software has become very popular in recent years. Therefore students must be comfortable with identifying, evaluating, modifying, and integrating open source for their work.

*In Section 9.2.1, we discuss our experience in infusing some of these elements in regular computing courses.*

**Need for higher focus on debugging**

Many characteristics like significant work in new development in every project, discrete abstractions, complex interactions among a very large of components, inherent invisibility, large groups of developers, continuous evolution, etc., make software highly vulnerable to errors. Software errors (bugs) result because of lack of attention and also because of misconceptions related to programming, operating systems, compiler, and tools, libraries, etc. Software errors
can be reduced by developing proper technical competence. The students need to learn to avoid, anticipate, identify, track, and remove bugs that often arise due to their misconceptions in their as well as others’ source code.

Debugging activity is by and large ignored by curriculum. SWEBOK [68] refers to debugging in a casual manner, and does not include it at all in its appendix D of specific topics. Debugging has been more seriously attention in interim revision of CS2001 [159]. We take a position that computing curriculum need to address this issue more seriously. Students need to be well versed in the use of tools and techniques for identifying and rectifying errors. The students also need to be exposed to common bugs, their consequences, and remedies. The computing curriculum and education programs need to give much more emphasis on debugging. In our proposed framework of pedagogical engagements for software development education, we include this aspect in Section 8.1. They need to learn to use debugging tools for interactive debugging, static analysis, and dynamic analysis.

Debugging activity requires lot of analytical effort. Metzger draws an analogy between programmer and safety analysts who seek to prevent future problems by doing a root-cause analysis of significant events, e.g., accidents, near misses and potential problems. For effective debugging, he suggests the usage of root-cause analysis techniques like ‘cause and event charting’ and ‘faulty tree analysis.’ We include these as part of proposed framework in Table 8.6.

Metzger observes that design errors may occur because of errors in data-structure, algorithm, or interface specifications related to user-interface, software-interface, or hardware-interface [157]. Annexure AN4 gives a summary of his observations. We have created a taxonomy of software bugs based on misconceptions related to programming, operating systems, compiler, and software architecture [159a]. Our taxonomy of software bugs is given in Appendix A6. We propose to enrich the courses with sufficient exposure to some of these bugs from each category.
Education program needs to give them opportunities to acquire, apply, extend, refine, and integrate their technical competence. Technical competence includes skill, rules, knowledge related to various technical activities as discussed in this section. Much of the routine behavior of experienced programmers is rule-based. These rules are often implicit and unarticulated by them. They use their rules to organize things into patterns [160]. Experienced software developers encode their rules in such a way that enables them to apply their rules with much lesser effort than a novice for solving the same problem. However, their rules depend upon their expertise, and may not cover all cases.

Metzger [157] catalogues the software errors because of rule-based reasoning into two broad categories: (i) misapplication of good rules occur when a time-tested rule is applied by overlooking the additional conditions that warrant another rule, (ii) application of a bad rule occurs when conditions are wrongly represented, or ineffective/inefficient action is chosen. More details of these are discussed in Annexure AN4. Hence, it is necessary for them to understand the scope and limitations of their rules.

Traditional methods of teaching fail to take such a comprehensive perspective of technical competence. In our recently concluded survey “Software developers - (How) Did your college help you in your development?” (Table A10.2 (i) part-I, Appendix A10), huge proportion of the respondents felt that as compared to all other academic engagements, their projects did much better to develop their design skills (92% respondents felt so), implementation skills (90%), debugging skills (84%), technical competence (76%), and analytical skills (75%). Laboratory work (70%), knowledge transmission oriented lectures (54%), and homework and tutorials (48%) were considered as effective for developing technical competence. Laboratory work and industrial training (84% and 49% respectively) were found to effective for implementation skills. Laboratory work, industrial training, and mentoring of juniors (86%, 35%, and 31% respectively) were found to effective for debugging skills. Laboratory work (63%), research literature survey (58%), thinking oriented lectures (54%), homework and tutorial assignments (42%), discussion with other students (38%), and industrial training (33%) nurtured the analytical skills. Laboratory work (61%), industrial training (49%), and thinking oriented lectures (47%) were found to be the main contributor for development of design skills. Traditional knowledge
transmission oriented lectures and discussions with others were found to be least effective for development of analytical skills whereas written examinations were found to be least effective for development of design skills.

**Section 4.2: Software Developers’ Education for Development of Computational Thinking**

Traditionally, software was regarded as belonging to the domain of ‘applied mathematics.’ Many experts view software development as a special type of mathematical problem solving activity which requires the developers to use various mathematical thinking processes like step-by-step approach to decomposition, abstraction, pattern recognition, spatial and temporal modeling, deduction and induction, and synthesis.

In his much debated talk called ‘On the cruelty of really teaching computing science,’ in 1989, Dijkstra emphasized on formalism [64]. He further identified the following two radical novelties of programming: (i) conceptual hierarchies deeper than a single mind ever needed to face before, and (ii) in a discrete world small changes do not imply small effect. In 1991, the joint ACM/IEEE-CS curriculum task force [62] identified twelve unifying and pervasive concepts of computing - binding, complexity of large programs, conceptual and formal models, consistency and completeness, efficiency, evolution, levels of abstraction, ordering in space, ordering in time, reuse, security, and trade-off and consequences.

In our 2009 survey on required competencies for software developers, twenty software professionals assigned abstraction thinking and algorithmic thinking an average rating of 2.9 and 2.8 respectively on a scale of 0-4. An overwhelming majority of these respondents (70%) recommended these to be critical or very important competencies with respect to the requirements of software developers' multi-faceted professional activities.

With reference to Appendices A2 and A3, computational thinking of software developers also relates to the following:

1. Abstraction and transition between levels of abstraction, representation skills, spatial and temporal modeling skills, structuring skills, and theorizing
2 Algorithmic and structured thinking. Logic, pattern matching, logical what-if analysis, problem decomposition and synthesis, etc.
3 Analytical skills
4 Attention to detail: breadth, depth, clarity, accuracy, preciseness, specificity, relevance, significance, completeness, consistency
5 Problem solving skills
6 Critical thinking
7 Design skills
8 Creativity and idea initiation

Algorithmic problem solving activities
Expert programmers think and develop algorithms rather than think in specific language syntax [179]. In 1979, Kowalski postulated that an algorithm consists of logical and control components [180]. The logic components define the knowledge that is needed to solve the problem. The control component determines how to use and sequence such knowledge to do so. Muller and Haberman [181] have enumerated algorithmic problem solving activities.

Problem comprehension is the first activity that involves reformulation of the problem statement in terms of data items, initial state, goal, assumptions, constraints, and scale. This is the most critical thinking stage for designing algorithms. For five consecutive years (2002-07), in data structure and algorithm courses, we emphasized on this aspect by engaging students to generate examples of increasing complexity in terms of scale, diversity, assumptions, goals, initial state, constraints, tolerance, and exceptions. The students were required to first develop the algorithms for the simplest possible case of each problem. With each additional case of increasing complexity of the problem, they were required to identify the limitations of the existing solution, and then modify the same to meet more complex demands. A comparison of problem solving strategies of best performing students of one such class with the best performers of a later class where the faculty used a more traditional textbook oriented approach, showed that the students of the first group showed a much higher level of sophistication in their approach to solve algorithmic problems.
The **second activity of decomposition** is the identification, naming, and listing of subtasks and data items with attributes, objectives, and roles. Analogical reasoning, generalization and abstraction are used for identifying similarities between problems, and extracting prototypes of problems from analogical problems in different contexts. This helps in identifying a problem's prototype for its categorization.

This is followed by the **problem's structure identification**, i.e., composition, identifying the relation between subtasks, data items, state transitions, data flow, and distinguishing between logic and control. Schematizing a problem's structure using diagrams helps a great deal in this process. Flow chart has great limitations in terms of its inability to show data or states.

A new diagramming technique called ‘concept mapping’ (Appendix A19) has been developed and used in various classes as mentioned above. The students who were exposed to concept mapping in their introductory data structures course continued to use it, or a self-modified notation, even after graduating. Based on this analysis, this notation has been re-introduced in an introductory data structure course, and the concerned faculties as well as students are finding it useful.

Finally, algorithm thinking requires **evaluation and appreciation** of efficiency and elegancy, reflecting on problem-solving processes and strategies to draw conclusions for the future, and verbalization of ideas and differentiating between an idea and its implementation.

**Lethbridge’s Study on Most Important and Influential Topics**

Lethbridge et al [46-48] surveyed approximately 200 practicing software engineers and managers. Their report shows that **five out of the thirteen subject categories did not contribute even a single topic to the list of twenty-five most important and influential topics**, while these categories were felt by the respondents to be over emphasized in the curriculum. **These subject categories are theoretical computer science, mathematical topics in computer science, other hardware topics, general mathematics, and basic science.**
Computational Thinking: Beyond Traditional College-level Mathematics and Algorithmic Thinking

In 2009, we initiated an online discussion among the online community of software professionals on LinkedIn. Nearly 30% respondents felt that proficiency in mathematics indicates a high capability to handle abstractions, the ability to go into detail, ability to plan and approach a problem in a methodical/structured fashion. On the contrary, the other majority suggested that this relationship between mathematics and software has been exaggerated, and gave reasons like mathematics education does not necessarily enhance lateral thinking for problem solving. However, many respondents grounded software development competency into puzzle-solving ability.

Wing [183] viewed computational thinking as an approach to problem solving, system designing, and also understanding human behavior, by drawing on the concepts fundamental to computer science. Isbell et al [182] shift the emphasis from algorithm to interaction and suggest that computing problem solving is not so much about finding answers but more about creating services, interfaces, and behaviors. Fant [53] argues that, unlike mathematics, computer science is more concerned with issues related to creation and actualization of process expressions.

In our experience, students without good background in school level mathematics, especially in topics like algebra, geometry, trigonometry, functions, etc., have been found to perform poorly in software development oriented courses. However, performance in college level mathematics courses like higher calculus, differential equations, and linear algebra, etc., seem to have no correlation with the performance in software development skills of college level engineering students. There are many exceptional programmers whose performance in college level mathematics has been poor, and there are many poor programmers with very good performance in college level mathematics.

According to Wing [183], computational thinking is about producing executable descriptions, i.e. automable or automatically manipulatable models. She strongly recommended that computing faculty teach courses on computational thinking which includes thinking in terms of: constraints, abstraction, decomposition, heuristics, algorithms, recursion, concurrency,
synchronization, efficiency, elegance, tradeoffs between processing and storage, caching, interpreting code as data and data as code, and prevention, detection and recovery from worst-case scenarios. Two relevant intuitions for computing are the concepts of \textit{having something and being in a state} [184]. In 1975, the chief designer of many programming languages, e.g., Pascal and Modula, Nicklaus Wirth, wrote a book titled, ‘Algorithms + Data Structures = Programs.’

There is a need to review the college level mathematics content from this perspective. Whenever mathematics courses succeed in engaging students in representing real-life problems into mathematical or computable problems, and then solving those problems using mathematical tools, they provide direct help in enhancing the analytical thinking skills required for software development. Courses on \textit{puzzle-solving and mathematical modeling} have a higher potential to make such direct contribution. 

\textit{We include this aspect in our propose framework of pedagogical engagements (Table 8.6).} In 1999, SEI- CMU published a report to define the discipline of software Engineering [67]. The mathematics requirements included ‘mathematical logic and proof systems,’ ‘discrete mathematical structures,’ ‘formal systems,’ ‘combinatorics,’ and ‘probability and statistics.’

Isbell et al [182] also take a position that though \textit{computing overlaps with various disciplines like mathematics, science, engineering, arts, humanities, and social sciences, it is neither of these} and is a discipline in itself that requires a distinguished kind of mindset which they term \textit{computationalist thinking}. They posit that the \textit{equivalence of model, language and machine} is the key idea of computing. According to them, \textit{computing marries the representations of some dynamic domain and dynamic machine to provide theoretic, empirical, or practical understanding of domain or machine.}

Computational thinking requires thinking in terms of \textit{data attributes, data flow, relationships, and state transitions}. It also involves thinking about \textit{system-environment boundary, interface, system metrics, scale, sequence flows, transactions, composition, exception handling, testability, evolution, and documentation}. Today, \textit{user interaction has become equally important}. Isbell et al [182] posit that computationalist thinking focuses on \textit{model, abstraction, interpretation, scales}
and limits, simulation, and automation. They insist that computationalists must understand how to create, analyze, and critique models.

**Abstraction as an Integral Part of Computational Thinking**

Hazzan and Tomakyo [124] highlight the importance of mental habit of abstraction and the ability to make transitions between levels of abstraction as an important skill for software developers. Computational thinking involves stepwise refinement with different notations at different levels. It involves thinking about reality at different levels of abstractions and to model the same through executable formalisms. The fundamental feature of computational thinking is abstraction of a situation/system/problem in such a way that the selected details in the model make it executable by a machine. The choice of the selected executable abstractions of the problem is driven by its purpose [185]. The purpose may be: (i) automation, or (ii) simulation either to get deeper insights or to create virtual worlds.

Abstraction is informally described as the process of mapping a representation of a problem onto a new representation. Philosophers like Aristotle, Hume, and Locke have taken a reductive perspective of the abstraction process and see it in terms of the filtering-away of irrelevant components and specifics, with the aim of extracting content or meaning. Constructivist perspective of abstraction emphasizes selection and combination of relevant constituents. Each new abstraction identifies a new phenomenon and becomes a potential constituent for further abstraction [186]. Abstraction concepts include association, aggregation, composition, classification, or generalization.

The computing worlds consist of things (objects), events, and actions (activities, processes, and operations). Kramer viewed computational abstraction as generalization to identify the common core or essence, manipulating symbolic and numerical formalisms, and also moving from an informal and complicated real world to a simplified abstract model [187]. Wing [183] sees it first as a process of deciding what details we need to highlight/ignore, and then choosing an appropriate representation to model the relevant aspects of a problem. It takes several iterations to fine tune computational abstractions. The maximum challenge is to gather a ‘complete’ overview of the given problem.
Computational abstractions are to be discovered by balancing creation against reuse, with a strong preference for reuse of things that are already tried and tested.

**Abstraction of Real World**
Nicholson et al [188] caution that since software developers solve problems that exist in the real world, their solutions must ultimately succeed in the real world, not just on the abstract level used to define the solution. They also suggest critical evaluation of computational abstractions because abstractions may become too generic/specific. The details removed in an abstraction may reemerge in a way that requires that they be considered. Any representation can have consequences for how the subject of the abstraction is understood. The existing computational abstractions may cross into new contexts by accident or default, and the same subject may recur at multiple layers of abstractions with different aspects and context. They insist on identification of the context of use and then defining the computational abstraction accordingly. For identification of the context of use, their recommendation is to understand the abstractions that are already used within the relevant context, and the socio-political context thereof. Software developers also need to identify the reusable ideas/components in the application and technology domain. Finally, regarding simultaneously working with multiple layers of abstractions, it is important to understand how the different layers of abstraction relate to each other, and always clearly indicate the layers being currently dealt with.

**A Key Principal for Designing Hierarchy of Abstractions**
In his classic paper, Miller had suggested that humans have an upper limit of the number of items that they can simultaneously hold in their temporary memory for further cognitive processing. This is in the range of seven plus/minus two [189]. Software developers should keep this in mind as they develop their hierarchy of abstractions.

**Pedagogic Perspective**
In our recently concluded survey, “Software developers - (How) Did your college help you in your development?” (Table A10.2 (i) part-II, Appendix A10), a majority felt that as compared to all other kind of academic engagements, their student projects did much better to develop their
algorithmic (72%) and abstraction (57%) thinking respectively. Algorithmic thinking was felt to improve through engagements of thinking oriented lectures (60%), laboratory work (58%), research literature survey (40%), and knowledge transmission oriented lectures (36%). Abstraction competence was felt to improve through engagements of research literature survey (40%), thinking oriented lectures (38%), and laboratory work (32%). Discussion with others, knowledge transmission oriented lectures and written examinations were felt to be least effective with respect to development of abstraction competence. Discussions with others, faculty as well as other students were found to be ineffective with respect to development of algorithmic thinking.

Student assignments need to be designed keeping the objectives of strengthening various aspects of computational thinking as discussed in this section. These assignments can be designed as per our proposed framework of pedagogic engagements in Chapter 8.

**Section 4.3: Software Developers’ Education for Development of Domain Competence**

All professional societies, including those that are associated with the professions of engineering or software development, strongly advocate that their profession’s main aim is to work for the welfare of society. Welfare requires a balanced fulfillment of the human needs at multiple levels of need-hierarchy as per Maslow’s need-hierarchy [141a], in compliance with concerns of sustainable development. Many technologies have mainly been supporting human activities that facilitate fulfillment of lower level human needs as per Maslow’s model of need hierarchy. Potentially, software can even support some human activities that facilitate fulfillment of people’s needs in various domains even at upper levels of Maslow’s hierarchy. Outgrowing the initial goal of doing repetitive mathematical calculations, computers have already permeated almost all spheres of human activities. **Software not only supports, but also facilitates the reorganization of business and/or production process itself.** Similarly, the social networking is now helping to transform and create new form of human-social interactions.

After decades of experience in creating and using software solutions, few domains, especially those that are related to science, engineering, governance and business, are more mature than
many others in terms of understanding of domain specific software possibilities both by software developers as well as concerned domain experts. Many new domains are fast emerging as big users of software. Both domain as well as IT experts are exploring new possibilities of creating IT enabled operations and services in these domains. A large number of new users and applications are emerging in the domains of business analytics, mass communication, customer relationship management, social marketing, security, energy management, environment management, compliance governance and risk management, healthcare, life sciences, and collaborative work. A significant number of novel applications are being developed for arts and sports as well.

In our 2009 survey on required competencies for software developers, twenty software professionals assigned ‘technical/domain competency’ an average rating of 2.95 on a scale of 0-4. A majority of 60% of these respondents recommended it to be a critical or very important competency with respect to the requirements of software developers' multi-faceted professional activities. In order to acquire required domain knowledge in varied application domains, a broad understanding of various processes and diverse human tasks is very helpful for software designers.

With reference to Appendices A2 and A3, domain competence of software developers also relates to the following:

1. “Be the customer” mentality
2. Analytical skills
3. Design skills
4. Imagination: storyboarding, extrapolation, visualization, cognitive flexibly: ability to transfer and models of solutions of one situation/field to another, multi-perspective thinking, lateral thinking, creativity and idea initiation, and innovation
5. Problem solving skills
6. Project planning

99
Armour [120] [148] viewed software development as a learning activity, rather than a production activity, and advocated that software developers need more training in domains, learning, and knowledge structuring mechanisms rather than in software itself. Effort to acquire required domain knowledge varies from few days to several decades depending upon the complexity of the problems. Domain training, and even certifications, have become common part of continuous training programs of software developers. Since the application domains are now virtually encompassing all kinds of human activities, it should be presumed that a fresh graduate is required to work in a new domain, to start with.

It is very common for software teams to do lot of rework because of insufficient understanding of the application domain. Lack of domain knowledge is a very significant problem in software projects and because of his deficiency, the requirements appear to fluctuate [168]. Domain specific knowledge enables developers to identify problems in logic [157]. Most software developers generally have a tendency to blame the users for fluctuating requirements. Hence, domain knowledge is well recognized as the key contributor to enhancing productivity of software development processes.

The software development processes essentially try to map the application domain requirements to programming constructs. Shirley identified four levels of skill in student programmers’ work: expedient, constructional, operational, and structural. Using the structure of the problem to devise the solution is considered the most sophisticated approach to programming [169]. A student at the structural level of programming skill first carries out an interpretation of the problem within its domain, then structures the problem before coding.

Many domain specific languages (DSLs) have been developed and continue to evolve for various domains. Domain driven design approach is becoming increasing popular among software designers. It is based on the premise that primary focus of software designers should be on the domain and domain logic, rather than on the particular technology used to implement the system. In general, students need to learn to capture the processes involved in a domain, identify the actors, events, schedules, compliances, etc., map the information flow, decision making based on information, identify the gaps and redundancies, optimize the processes through business and
process re-engineering, and most importantly, substantiate the value delivery of IT in the enterprise.

Domain specific conceptual knowledge and technical skills are important aspects of the domains. However, domains are characterized by prominent thinking processes. Different kinds of thinking processes are prominent in different domains. Increasingly, software is being developed to support the cognitive processes of domain specialists both for analytical as well as creative tasks.

Software is no more limited to only providing rapid and reliable data storage/transfer/access. It is increasingly transforming computers as cognition support systems through various devices for data transformation, analysis, and synthesis. In order to develop domain specific cognition support systems, an understanding of domain specific cognitive tasks is essential for the software developers. A sound understanding of a specific domain enables software developers to look for problem cases, failure modes, and benefits to the actual users.

Computing does not just open new ways of doing domain specific activities; it also requires new conceptualizations of the domain that require automated processing. The opportunities of automated processing in turn further open new ways of re-conceptualization in application domains [176]. Sometimes, software developers can also help in inventive problems solving in specific domains by infusing different thinking patterns developed through their experience in other domains.

**Breadth and Diversity**

**Diversity of Disciplines**

In 1973, Biglan classified academic disciplines along three dimensions. Each of these dimensions were broadly classified into two categories - hard vs soft, pure vs applied, and life vs non-life [170-173a]. Hence, for our purpose, we will treat these dimensions as bi-level. As per this classification, hard disciplines follow a single common paradigm, whereas the experts of soft disciplines differ in their methodologies and concepts. Table 4.2 summarizes this classification.
Table 4.2: Biglan’s classification of disciplines

<table>
<thead>
<tr>
<th>Pure</th>
<th>Hard Life</th>
<th>Hard Non-life</th>
<th>Soft Life</th>
<th>Soft Non-life</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pure</td>
<td>Biology,</td>
<td>Mathematics,</td>
<td>Psychology,</td>
<td>Linguistics,</td>
</tr>
<tr>
<td></td>
<td>Biochemistry,</td>
<td>Physics,</td>
<td>Sociology,</td>
<td>Literature,</td>
</tr>
<tr>
<td></td>
<td>Genetics,</td>
<td>Chemistry,</td>
<td>Anthropology,</td>
<td>Communications,</td>
</tr>
<tr>
<td></td>
<td>Physiology,</td>
<td>Geology,</td>
<td>Political Science,</td>
<td>Creative</td>
</tr>
<tr>
<td></td>
<td>etc.</td>
<td>Astronomy,</td>
<td>Area Study,</td>
<td>Writing,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Oceanography,</td>
<td>etc.</td>
<td>Economics,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>etc.</td>
<td></td>
<td>Philosophy,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Archaeology,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>History,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Geography,</td>
</tr>
<tr>
<td>Applied</td>
<td>Agriculture,</td>
<td>Civil Engineering,</td>
<td>Recreation,</td>
<td>Finance,</td>
</tr>
<tr>
<td></td>
<td>Psychiatry,</td>
<td>Telecommunication Engineering,</td>
<td>Arts,</td>
<td>Accounting,</td>
</tr>
<tr>
<td></td>
<td>Medicine,</td>
<td>Mechanical Engineering,</td>
<td>Education,</td>
<td>Banking,</td>
</tr>
<tr>
<td></td>
<td>Pharmacy,</td>
<td>Chemical Engineering,</td>
<td>Nursing,</td>
<td>Marketing,</td>
</tr>
<tr>
<td></td>
<td>Dentistry,</td>
<td>Electrical Engineering,</td>
<td>Conservation,</td>
<td>Journalism,</td>
</tr>
<tr>
<td></td>
<td>Horticulture,</td>
<td>Computer Science,</td>
<td>Counseling,</td>
<td>Library And</td>
</tr>
<tr>
<td></td>
<td>etc.,</td>
<td>etc.</td>
<td>HR Management,</td>
<td>Archival Science,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>etc.</td>
<td>Law,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Architecture,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Interior Design,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Crafts,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Arts,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Dance,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Music,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>etc.</td>
</tr>
</tbody>
</table>

The hard-pure disciplines are concerned with universals and simplification, whereas soft-pure disciplines are concerned with particular cases. The thinking approaches significantly differ for these categories. The hard-pure disciplines have an atomistic approach and rely more on linear logic, facts, and concepts whereas soft-pure disciplines have a holistic approach, and rely more on the breadth of intellectual ideas, creativity and expression. The hard-applied disciplines focus on problem solving and application of knowledge to create products and techniques, whereas, soft-applied disciplines focus on personal growth, reflective practice, and lifelong learning to create protocols and procedures. The hard-pure disciplines are concerned with mastery of physical environment, whereas soft-applied are concerned with enhancement of professional practice.

As per this classification, computer science is classified in hard, non-life, applied category of disciplines. The algorithm design and programming part of software development surely belongs to this category. However, software development also includes many other tasks like project management, requirement analysis, user interface design, and usability analysis. These tasks relate to people, and hence software developer like engineering or computer science cannot be classified to the single category as per this classification.

Application domains of software include all disciplines and hence are well spread over all categories of Biglan’s classification. Hence, it is imperative for software developers to
understand the concerns, focus, aim, knowledge structures, and thinking approaches of application domains that belong to all categories of Biglan classification.

Unlike traditional engineering based waterfall model, evolutionary approaches to software development view users’ requirements as tentative, evolving, and open to change. Paulsen and Wells [174] found that as compared to the students of pure fields like science, fine arts, social science, and humanities, the students of applied areas like business and engineering hold more naïve beliefs about the structure of knowledge and speed of learning. Engineering students were also found to have more naïve views about certainty of knowledge. Theirs, and earlier, research showed that as compared to engineering students, the students of soft fields like social science, fine arts, humanities, education, and business are more likely to view knowledge as diverse, tentative, and open to change. Hence, a good grounding in soft disciplines becomes even more important with respect to current and emerging trends of software development methodologies. The teaching of computing courses also needs to be restructured in order to develop these epistemological beliefs.

Because of the nature of their curriculum, engineering students in computing disciplines are already well over-exposed to the approaches of hard-pure and hard-applied disciplines. This tends to limit their perspective and approach. Hence, it is strongly recommended that engineering students of computing disciplines are well exposed to soft-pure and soft-applied disciplines also, especially as application domains. We have further developed and included this approach in our proposed framework of pedagogical engagements in software development education (Table 8.7 and Table 8.8).

Good observation skills, enquiring mind, diversity of interests, empathy, and reflection skills, are the prerequisites for building the required domain understanding. The software development education program needs to expose the students to diverse types of domains and domain categories as per Biglan’s classification. It also needs to nurture the ability to learn nuances of newer domains. This exposure to application domains needs to introduce the students to specific attributes: (i) context: users, functions, concerns, constraints, compliance requirements, (ii) operations: procedures, practices, methods, evolution of IT applications, (iii) domain specific
vocabulary and its semantics, (iv) domain experts’ cognitive tasks, and (v) challenges: complexities, risks, uncertainties, and complications.

**Diversity of Learning Styles**

Kolb [152] identified four main learning styles. Kolb also discovered prominent patterns of correlation of the styles with respect to domains, and also with concerned persons’ functions [175]. These four styles are given in Table 4.3. Rather than following the commonly popular perspective that subjects are linked with specific learning styles, *we take a position, that different styles are relatively more suitable for learning different aspects of a single subject. Hence, an integration of these styles enhances learners’ ability to learn different aspects of any domain*. Kolb proposed ‘experiential learning cycle’ for facilitating deeper learning.

*A liberal arts kind of broad based educational model that includes exposure to diverse disciplines, not just science, mathematics, engineering, or management is potentially suited to make the students more ready for developing software for diverse domains*. Breadth of multi-disciplinary exposure also offers the opportunity to enrich and diversify students’ repertoire of learning style. However, such multi-disciplinary courses will be effective, only if they succeed in *engaging the students in prominent cognitive tasks with the domain experts to some extent*. We use Kolb’s style to enrich our proposed framework of pedagogic engagements in section 8.3.1.

<table>
<thead>
<tr>
<th>Table 4.3: Kolb’s learning styles</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. <strong>Divergent</strong>: involves reflection on concrete experience, requires abilities of concrete experience as well as reflective observation. This style is associated with valuing skills: relationship, helping others, and sense making. Such people have broad interests and tend to be imaginative and specialize in arts, literature, psychology, etc. Effective communication and relation building requires this style.</td>
</tr>
<tr>
<td>2. <strong>Convergent</strong>: involves active experimentation to test/apply abstractions, requires abilities of abstract conceptualization as well as active experimentation. This style is associated with decision skills like quantitative analysis, use of technology, and goal setting. Such people like to deal with technical rather than people related aspects, and tend to specialize in technology and medicine. Bench engineering and production requires this style.</td>
</tr>
<tr>
<td>3. <strong>Accommodative</strong>: involves active experimentation on concrete experiences, requires abilities of concrete experience as well as active experimentation. This style encompasses a set of competencies that can best be termed acting skills: leadership, initiative, and action. Such people tend to specialize in education, social service, sales, communication, nursing, etc. Decision making in uncertain situations requires this style.</td>
</tr>
<tr>
<td>4. <strong>Assimilative</strong>: involves reflection on abstractions; requires abilities of abstract conceptualization as well as reflective observation. This style is related to thinking skills: information gathering, information analysis, and theory building. Such people tend to specialize in mathematics and physical sciences. Planning and research activities require this style.</td>
</tr>
</tbody>
</table>
**Pedagogic Perspective**

In our recently concluded survey, “Software developers - (How) Did your college help you in your development?” (Table A10.2 (i) part-II, Appendix A10), a large fraction of 61% felt that as compared to all other kind of academic engagements, their projects did much better to develop their domain competence. This was followed by research literature survey and knowledge transmission oriented lectures (51% each), laboratory work (39%), homework (35%), written examinations and mentoring juniors (31% each).

Many universities have started domain specific computing degree programs at master’s level e.g., bio-informatics, digital arts, computational finance, computational economics, health informatics, computational mathematics, computational physics, computational social science, computational psychology, archive and museum informatics, etc. In such programs the non-computing discipline is strongly shaped and heavily influenced by computing principles. Typically, the students in such programs have an undergraduate degree in the domain. In 2005, we also proposed the design of a two year master’s program in Archaeo-heritage Informatics [177], given in Appendix A7.

**Recommendations for Breadth Courses for Developing Domain Competence**

*Training in general systems thinking helps in quickly understanding even unfamiliar areas* [178]. Weinberg considered that linguistic and mathematical competencies are essential foundations for general systems thinking. In Section 6.3, we further elaborate upon system thinking.

Broad based education in diverse disciplines is likely to enrich linguistic sensibility and competence. While computing courses need to bring a higher focus on a systems approach in their delivery, the breadth courses in other disciplines can also very significantly contribute to develop general systems thinking by specifically bringing it as one of the prominent learning objectives. In order to help in general systems thinking, the courses need to be selected and redesigned with this aim. In order to develop system thinking and ability to learn a new domain, the breadth courses too should try to enhance their focus on: (i) diversity and multi-perspective thinking, (ii) inter-disciplinary integration and applications, (iii) and systems approach.
Repeated exposure to *complexity, complications, nonlinearity, uncertainties, and risks*, and as highlighted and illustrated within the context of each of the specific breadth courses is likely to significantly enhance their ability to understand the nuances of unfamiliar domains, and also to orient their mindset to decision making in complex situations.

Within the context of many knowledge disciplines in sciences, mathematics, engineering, management, social sciences, and humanities, a body of knowledge has already been created around systems and systems thinking. Appendix A8 suggests some such breadth courses that can develop and reinforce systems thinking and help in developing the ability to learn new domains.

**Section 4.4: Software Developers’ Education for Development of Communication Competence**

There are several kinds of communication in a software development project [161]. These include communications (i) between the development team and the customers, (ii) between the developers and the project manager, and (iii) among the developers. A typical project manager in IT spends about 90% of the time in communication with various stakeholders. Often communications competence is misinterpreted as making exciting presentations or impressive speaking or writing skills. *However, the communication needs of software developers are very different from the communications needs of sales or marketing professionals.* Communications skills do not make up for the deficiency in thinking ability. Good communication requires keeping track of who, what, when, and why. It mainly involves *listening with understanding and empathy.*

The communication competence of software developers encompasses the need to communicate their difficulties and vision to their clients, management, colleagues, and end-users, and also preparing technical documentation, and also end-user documentation. One needs to keep himself in the shoes of the end-user to give a useful product. *Communication encourages the exchange of ideas and project related knowledge among the people engaged in the project: clients, managers, and developers.*
Communication among project participants is formalized through various documents. The forms of documentation include requirements, specifications, architectural documents, detailed design documents, quality documents, and also low-level design information such as source code comments. Hence, effective verbal and written communication skills are also essential for software developers.

**In our 2009 survey** on required competencies for software developers, twenty software professionals assigned ‘communication skills’ an average rating of 2.75 on a scale of 0-4. A majority of 65% of these respondents recommended it to be a critical or very important competency with respect to the requirements of software developers’ multi-faceted professional activities.

With reference to Appendices A2 and A3, communication competence of software developers also relates to the following:

1. Ability to work in teams
2. Listening skills
3. “Be the customer” mentality
4. Persuasion, negotiation, consensus building, and conflict resolution skills.
5. Mentoring, coaching, and training skills
6. Organizational skills

Agile manifesto emphasizes face-to-face communication over written documents. Extreme Programming (XP) relies on four values: simplicity, communication, testing and courage. Chau et al [162] posit that software engineering is a knowledge-intensive process with a very strong need for knowledge sharing support to enable software organizations to:

7. effectively share domain expertise between the customer and the development team,
8. identify the requirements of the software system,
9. capture non-externalized knowledge of the development team members,
10. bring together knowledge from distributed individuals to form a repository of organizational knowledge, and
retain knowledge that would otherwise be lost due to the loss of experienced staff; and

improve organizational knowledge dissemination.

They observe that while traditional software development approaches support knowledge sharing primarily by documents or repositories, agile approaches rely heavily on socialization through communication and collaboration.

Cockburn [163], one of original authors of the agile manifesto, highlighted the following principles regarding communication in setting and running of software projects:

1. Larger teams need more communication elements.
2. Interactive, face-to-face communication is the cheapest and fastest channel for exchanging information.
3. Increased communication and feedback reduces the need for intermediate work products.

Outsourcing and offshore development has added to the communication related challenges in the software development process. Documentation becomes more important with offshore development. Xiaohu et al [161] cite previous research that had shown that because of communication and coordination issues, distributed projects take about two and half times longer to complete as similar projects where the project team is centralized.

Today’s software development situation sees two types of trends: document driven outsourcing, and offshore project and agile approaches emphasizing a lot of face-to-face communication. Attempt to blend these two are also underway.

Responding to our online polls (results summarized in Table 4.6), one of the senior-level respondents wrote, “Software development life cycle is completely dependent on the communication effectiveness. Communication tends to break at every stage of the software development life cycle. Hence communication skills (mainly comprehension and listening) are of paramount importance in software engineering.” Another senior level respondent recalled his
customer saying “being convinced about giving the work to you is a challenge because we are not sure if your team has accurately understood our requirements.”

Christiansen [164] has concluded that these additional challenges arise because of factors like different cultures, different languages and accents, thin communication channels, different platforms, and different time zones. Christiansen has also given some suggestions to overcome these challenges: put stress on synchronous communication, adapt to and understand other cultures, put emphasis on spoken language skills, rotate people between shores, use artifacts properly, align IT infrastructure, use requirement specifications with care, and invest time and money in transferring implicit knowledge.

Backer et al [165] had studied aspects related to competency requirements for computing professionals with respect to various micro-level communications skills: writing, reading, speaking, listening, presentation, and nonverbal. They had also studied extent of engagement of technical professionals in these micro level communications. With reference to software developers, communication involves frequent translations between the domain/business descriptions to/from technological descriptions, in both directions.

The braintrack.com provides the summary of responses of the perceived importance of communication abilities for various professionals. Table 4.4 summarizes their finding with respect to programmers and systems analysts.

<table>
<thead>
<tr>
<th>Communication skill</th>
<th>Respondent programmers who find it important for their work</th>
<th>Respondent systems analysts who find it important for their work</th>
</tr>
</thead>
<tbody>
<tr>
<td>Written comprehension</td>
<td>77%</td>
<td>66%</td>
</tr>
<tr>
<td>Oral comprehension</td>
<td>66%</td>
<td>63%</td>
</tr>
<tr>
<td>Oral expression</td>
<td>60%</td>
<td>60%</td>
</tr>
<tr>
<td>Written expression</td>
<td>56%</td>
<td>56%</td>
</tr>
</tbody>
</table>

In order to understand the relative importance of micro-level communication skills for software developers, we conducted two polls among software developers.
The first poll, Poll-A, asked the respondents to choose the most important micro-level communication skill with reference to the requirements of software development work. The micro level communication skill considered included: (i) speaking with clarity, (ii) making impressive presentations, (iii) writing with clarity (iv) reading with comprehension, and (v) listening with understanding and empathy.

With respect to these micro level communication skills, the second poll, Poll-B, asked them to identify most serious weakness of typical Indian engineering graduates. We respectively received 84 and 69 responses for these two polls. Table 4.5 gives the summary of the profile of the respondents.

Table 4.5: Profiles of the respondents for the two polls about communication competence among software developers

<table>
<thead>
<tr>
<th></th>
<th>Poll-A Importance (84 responses)</th>
<th>Poll-B Weakness (69 responses)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Age</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt;55 years</td>
<td>Nil</td>
<td>5%</td>
</tr>
<tr>
<td>35-54 years</td>
<td>42%</td>
<td>26%</td>
</tr>
<tr>
<td>25-34 years</td>
<td>54%</td>
<td>58%</td>
</tr>
<tr>
<td>18-24 years</td>
<td>4%</td>
<td>11%</td>
</tr>
<tr>
<td><strong>Company size</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Enterprise</td>
<td>29%</td>
<td>36%</td>
</tr>
<tr>
<td>Large</td>
<td>29%</td>
<td>29%</td>
</tr>
<tr>
<td>Medium</td>
<td>21%</td>
<td>14%</td>
</tr>
<tr>
<td>Small</td>
<td>21%</td>
<td>21%</td>
</tr>
<tr>
<td><strong>Job function</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Consulting</td>
<td>21%</td>
<td>38%</td>
</tr>
<tr>
<td>Engineering</td>
<td>37%</td>
<td>31%</td>
</tr>
<tr>
<td>Product</td>
<td>21%</td>
<td>15%</td>
</tr>
<tr>
<td>Sales</td>
<td>14%</td>
<td>Nil</td>
</tr>
<tr>
<td>Academics</td>
<td>7%</td>
<td>8%</td>
</tr>
<tr>
<td>IT</td>
<td>Nil</td>
<td>8%</td>
</tr>
</tbody>
</table>

Table 4.6 summarizes their responses.

Table 4.6: Summary of responses for these two polls about communication competence

<table>
<thead>
<tr>
<th>Micro level communication skill</th>
<th>Poll-A Importance (84 responses)</th>
<th>Poll-B Weakness (69 responses)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Listening with understanding and empathy</td>
<td>55%</td>
<td>37%</td>
</tr>
<tr>
<td>Writing with clarity</td>
<td>16%</td>
<td>14%</td>
</tr>
<tr>
<td>Speaking with clarity</td>
<td>15%</td>
<td>33%</td>
</tr>
<tr>
<td>Reading with comprehension</td>
<td>7%</td>
<td>5%</td>
</tr>
<tr>
<td>Making impressive presentations</td>
<td>4%</td>
<td>8%</td>
</tr>
</tbody>
</table>
The responses show a very interesting aspect. All the respondents who identified “making impressive presentations” as the most serious communication skill related weakness of Indian engineering graduates, belong to the youngest age group of 18-24 years. It is further very interesting to note that 50% of the respondent in this age group thought so. In the last few years, there has been a sudden increase in the emphasis on communication skills; often this is misunderstood in terms of making impressive presentations. Our poll shows the extent of this misconception among our fresh engineers.

One of the responding enterprise IT architect from India commented, “Indian engineering graduates tend to just skim the surface and are not that well prepared during discussions and meetings.”

**Pedagogic Perspective**

Many studies have showed that multi-paradigm disciplines like humanities, social sciences, and psychology had a positive influence on self-reported growth in communication skills by students. However, Li et al have found that self-perceived gains in communication skills most significantly depended upon the degree of their integration into the social community of the university rather than their discipline of study [166]. Further, the quality of curriculum was found to be the most significant factor for influencing their social integration.

In our recently concluded survey, “Software developers - (How) Did your college help you in your development?” (Table A10.2 (i) part-II, Appendix A10), a large fraction of 84% felt that, as compared to all other kind of academic engagements, their discussions with other students did much better to develop their communication skills. This was followed by mentoring juniors and discussions with faculty (71% each), discussion with others (51%), and industrial training (45%). All other engagements were found to be inadequate in this regard by the respondents.

Etlinger [167] has suggested a framework for teaching communication skills to computing students. This framework has three concentric circles. The inner circle is about the critical ideas: (i) purpose - inform, instruct, or persuade, (ii) strategy- form of communication, organization of information, and tone of communication, and (iii) audience- hostile or receptive, supportive or
neutral, internal or external, interested in the entire artifact or only in part of it. The middle circle of skills includes: reading and writing, listening and speaking, reviewing and evaluating, and, thinking. The outer circle focuses on process issues, more general traits, and quality.

Rather than only depending upon the communication skill related courses offered by humanities, language or management department, many authors have experimented with the strategy of including improvement of communication skills as one of desired learning outcome of their regular computing courses and also project work. Many view project work as an effective way of improving desirable kind of communication skills. At some universities, special courses on technical communication have been offered by the computing faculty.

The communication competence for software developers is significantly different from the communication competence for sales professionals. It is essential for understanding the needs of the consumers, the difficulties of their clients and co-developers. It is required for knowledge acquisition as well as knowledge sharing. *We posit that active and collaborative engagements as included in our proposed taxonomy of pedagogical engagements in Sections 8.3.1 and 8.3.4 respectively contribute towards developing the required communication competence of computing students.*

**Section 4.5: Software Developers’ Education for Development of Complex Problem Solving Competence**

**Programming and Problem Solving**

Gomes and Mendes view *programming as problem solving* [190]. Booth identified conceptions of programming and ‘learning to program’ generally held by students [169]. As per her model, *a student’s conception about programming* grows in sophistication from the initial level of computer related activity to problem oriented activity to product oriented activity. She also identified the stages of increasing sophistication in a student’s conceptions of ‘learning to program’: (i) learning a programming language, (ii) learning to write a program in a programming language, (iii) learning to solve problems in the form of programs, and (iv) *becoming part of the programming community.*
Further, the *conceptions related to programming languages* grow in sophistication from simplest levels of viewing a programming language as a utility program that enables programs to be written, to the second level of code as a set of instructions, commands, symbols, and constructs. The third level in this order is viewing programming language as a means of communication between programmer and computer to enable communication between computer and user. The *highest level views programming language as a medium of expression for the programmer to express solutions.*

**In our 2009 survey** on required competencies for software developers, twenty software professionals assigned ‘problem solving ability’ an average rating of 3.2 on a scale of 0-4. A large majority these respondents (80%) recommended it to be a critical or very important competency with respect to the requirements of software developers' multi-faceted professional activities.

With reference to Appendices A2 and A3, complex problem solving of software developers also relates to the following:

1. Ability to convert ill-defined problematic situations into software solvable problem
2. Problem orientation, problem definition and formulation, generations of alternatives
3. Emphasis on elegant and simple solutions
4. Ability of infusing different thinking patterns developed through their experience in other domains
5. Inclination for reuse and synthesis by integration
6. Solution implementation and verification
7. Project planning and management
8. Sense of urgency and stress management

**Good Solutions**

Conceptualizing programming to become part of the programming community and conceptualizing programming language as a medium of expression of solution are indicators of the possibility of multiplicity of solutions for the same problem. The solutions may suffer from
shortcomings like over-simplification, over decomposition, under-decomposition, or disordered management of complexity in a disordered manner. It is not sufficient to somehow solve complex problems. The need of elegance, i.e., ordered management of complexity, increases with increasing number and diversity of items, relations, correlations, and systems of relations. Elegant management of complexity affords overall comprehension and continuous orientation. Like good literature, architecture, or some other work of art, elegant software exhibits clarity, simplicity, precision, minimized interfaces, orderliness, coherence, and consistence, without compromising on integrity and performance. Avoiding unnecessary complications is a necessary requirement for elegant solutions. Software developers’ aesthetical sense, urge for elegance, patience, and systems-level perspective are necessary driving force for imbibing elegance in their solutions.

Developing elegant software also requires good understanding of the specific problem and also its context. Multidimensional complexity of application domains has to be registered, facilitated, and expressed in software constructs. Hence, not just technical and computational thinking competence, but domain competence is also necessary for developing elegant software.

Good software also includes defense, error-handling, and recovery mechanisms for various kinds of errors: hardware-level, programming, or user induced. It also affords testability and portability.

**Problem Solving**

In a study [191], almost unanimously, i.e., 97.7% of 1023 experts rated ‘problem solving’ as an important element of human intelligence. In its most simplistic interpretation, a problem is something that cannot be solved in a single, obvious step. Gomes and Mendes also provide some of the following interesting definitions of problem and problem solving:

*Pérez et al.* - problem is a situation for which there isn’t an evident solution.

*Perales* - problem is any situation that produces, on one hand, a certain degree of uncertainty and, on the other, behavior in search of a solution.
Gagné - problem solving is a process where the apprentice/learner discovers a combination of rules previously learned that he/she can apply to reach a solution for a new problematic situation.

Nickols [192] opines that what characterizes a problem is uncertainty about action, having a goal and not knowing how to achieve it. Problem solving depends upon cognitive processes of problem anticipation and identification, problem understanding, problem definition, problem formulation, problem representation, generations of alternatives, decision making and planning, implementation and integration, monitoring, evaluation, improvisation, and solution communication.

Jonassen [193] has proposed a taxonomy of problems based on variations in problem types and representations. The problem types vary in a three dimensional continuous space of three factors: structured-ness, complexity, and degree of domain specificity. Software problems are domain specific, complex, and ill structured. Based on the cognitive task analysis of various kinds of problems, Jonassen has identified eleven different kinds of problems. Software developers typically deal with all these kind of problems. Nickols’ typology of problem solving approaches [198] comprises of (1) repair approach, (2) improvement approach, and (3) engineering approach. Software developers mainly adopt the later two approaches, because seemingly, repair problems in software systems are actually improvisation problems. Annexure AN6 gives more details of these models.

Expert programmers are found to be good at logical thinking; many of them also enjoy puzzle-solving [194]. Metzger [157] has viewed debugging as a search problem like mathematical problem solving that is solved using a variety of search strategies like binary search, greedy search, depth-first search, and breadth-first search. In his classic, ‘How to solve it,’ famous mathematician Polya [195] listed four phases of problem solving: (i) understand the problem, (ii) plan the solution, (iii) execute the plan, and (iv) review the results. More details are given in Annexure AN6.

Drawing analogies between debugging and mathematical problem solving, Metzger [157] explains many heuristics for solving debugging problems: (1) stabilize the problem, (2) create a
standalone test case, (3) categorize the problem with reverence to correctness, completion, robustness, and efficiency, (4) describe the problem according to a standard methodology, (5) explain the problem to someone else, (6) recalling a similar problem, (7) drawing diagrams like control flow graph, data flow graph, and complex data structures with pointers, and (8) choosing a hypothesis from historical data. Further, he has also suggested some strategies like program slice strategy, deductive reasoning strategy, and inductive reasoning strategy for debugging.

Cognitive psychologists have studied problem solving methods for the last few decades. Galotti collates some general domain independent techniques for puzzle-like problems [196]. These include: generate and test, means-ends analysis, working backward backtracking, reasoning by analogy. Annexure AN6 gives more details of these.

**Creative Problem Solving**

Stoycheva and Lubart [197] have elaborated upon a creative problem solving process. This process involves five main activities of data finding and mess finding, problem finding, idea finding, solution finding, and acceptance finding. The first activity of data and mess finding is carried out by collecting data from the senses, experiences, knowledge, feelings, opinions, emotions, memories, fantasies, future projections, interaction with others, information on the social roles, and situation. Data collection can also be purposefully unstructured, random, and divergent. Data and mess finding also involves evaluation of relevance, interconnectedness, and importance of collected data. Through the analysis of this data, mess is discovered, created, or recreated.

The second activity of problem finding (or problem defining) is most creative in the problem solving process. Creative people try out many formulations and interpretations until one is found that best fits the data and offers the best opportunities for solving the problem. In this reference, Nickols [198] insists on defining the problem and also the solution state. He recommends to clearly detailing out the boundaries, distinguishing characteristics, the nature, and meaning of solution state. He [199] also insists on defining objectives and goals through inquiring about what are we trying to achieve, preserve, avoid, or eliminate?
The third activity of creative problem solving, **idea finding**, aims to filter out the most promising options which are identified for further elaboration. It involves multi-perspective thinking about concepts and experience. The fourth activity of **solution finding** is about examining selected alternatives from multiple perspectives for their pluses, minuses, and other interesting aspects. It involves exploring and finalizing the criteria for evaluation of alternatives. Further, alternatives are evaluated using the chosen criteria, and the most appropriate is chosen for implementation. Finally, the activity of **acceptance finding** is about successful implementation. It also requires envisaging how different stakeholders will react to the innovation.

With reference to abovementioned creative problems solving process, *we recently conducted a LinkedIn Poll among software professionals*. Seventy-six software professionals responded to this poll. The respondent professionals were well distributed in terms of age and job functions. In all, 7% respondents were older than 55 year, 33% were in the age group of 35-54, 47% were in the age group of 25-34, and 13% belonged to the age group of 18-24 years. In terms of job function, their distribution was: 13% in consulting, 38% in engineering, 38% in product, and 13% in creative functions. All respondents belonged to large or enterprise organizations.

With respect to the problem solving skills for software work, the respondents identified the most serious weakness of Indian engineering graduates as follows:

a. **Idea and/or solution finding** (36%),
b. **Problem (re)formulation** (22%),
c. Implementing (18%),
d. Mess identification (12%), and
e. Stakeholders’ acceptance (12%).

Some respondents also commented as follows:

"...engineers tend to assume the cause of the problem and jump to solutioning. This usually leads to compounding the situation. Engineers tend to overlook the importance of problem assessment, analysis and ascertainment."

"...our grads are enthusiastic and want to provide a quick fix to the problem, which is preventing them from thinking in terms of the "5 Why's"..."

*It shows the importance of creativity with reference to problem solving through software development.*

This issue is discussed in details again in Section 5.3.
Section 4.5.1: Expert problem Solvers

Research on expertise has shown that it takes approximately ten years to turn a novice into an expert. Hence, the four year undergraduate education needs to prepare the student to make the rest of the progress. In early 1970s, Gordon Institute proposed a famous four-stage conscious competence theory. As per this theory, the competence has four stages: unconscious incompetence, conscious incompetence conscious competence, and unconscious competence. Nonaka added a fifth stage to this and called it reflective competence [200].

Winslow [201] refers to the five levels of expertise as suggested by Dreyfus and Dreyfus in 1985. These are the levels of novice, advanced beginner, competence, proficiency, and expert. In the specific context of computing professionals, Denning [202] has refined Dreyfus levels, has added two more levels (master and legend) after expert. We have merged Gordon Institute’s and Denning’s levels into a single ladder. The merged levels are shown in Table 4.7. First seven levels of this are included in our proposed framework of pedagogical engagements in software development education (ref: Table 8.2, first column).
Table 4.7: Competency ladder (Integrating the ladders by Gordon Institute, Dreyfus and Dreyfus, and Denning)

<table>
<thead>
<tr>
<th>Level</th>
<th>Description with respect to software professionals</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Unconscious incompetence</td>
<td>Does not recognize the competency deficit, nor desires to learn.</td>
</tr>
<tr>
<td>2 Conscious incompetence</td>
<td>Recognize the competency deficit, without addressing it.</td>
</tr>
<tr>
<td>3 Novice (beginner)</td>
<td>Aims to learn objective facts, features, and rules for determining actions without being context sensitive. Focus on syntax etc. Learn through memorization and drill.</td>
</tr>
<tr>
<td>4 Advanced beginner</td>
<td>Recognizes common situations that help in recalling which rules should be exercised, starts to recognize and handle situations not covered by given facts, features and rules. Learns through problem solving and repeated practice with common situations.</td>
</tr>
<tr>
<td>5 Entry-level Professional (competent)</td>
<td>Performs most standard actions without conscious application of rules after considering the whole situation. Handles new situations through appropriate application of rules, can design systems. May lead. Learns through advanced problem solving, projects, extensive practice in common and exception situations, and participation in professional networks.</td>
</tr>
<tr>
<td>6 Proficient professional</td>
<td>Effortlessly deals with complex situations, no longer has to consciously reason through all the steps to determine a plan, appropriate actions come from experience and intuition. Design and manage complex systems, ingenious solutions. Learns through apprenticeship to experts, coaching, putting self into wide range of situations, membership and contributions to professional networks.</td>
</tr>
<tr>
<td>7 Expert</td>
<td>Consistently inspiring and excellent performance. An expert generally knows what to do, base upon mature and practical understanding. Performance standards are well beyond those of most practitioners. Extensive experience with large systems, appreciate subtle and indirect design issues and customer concerns, leads well. High productivity. Learns through apprenticeship to masters, advanced coaching, and development of breadth. Years/decades of experience.</td>
</tr>
<tr>
<td>8 Master</td>
<td>Capacity for long range strategic thinking and action. Sees historical drifts and shifting clearings. Has developed a distinctive style. Has produced innovations, altered the course of history in the field. Teaches others to be experts and masters. Develops new methods, admired for long. Learning by working with other masters. Creates and leads professional networks.</td>
</tr>
<tr>
<td>9 Legend</td>
<td>Has attained high standing. Work has widely accepted impact. Shapes directions of the field.</td>
</tr>
</tbody>
</table>

Costa and Kallick [203] have identified sixteen characteristics of what intelligent people do when they are confronted with problems, the resolution to which is not immediately apparent. These are listed in Annexure AN6.

Problem solving requires cognitive and meta-cognitive processes and also affective and conative elements of self-confidence, perseverance, open-mindedness, motivation, and mindful effort. Meta-cognitive aspects have been discussed under ‘critical and reflective thinking.’ The affective and conative elements are elaborated in sixth chapter.

Galotti [196] describes some findings related to factors that hinder problems solving. Mental set is the tendency to adapt a certain framework, strategy, or procedure, or more generally, to see things in a certain way instead of another. It causes people to make certain unnecessary
assumptions even without awareness. *Incomplete or incorrect representations* make problems solving much harder.

Jonassen [193] and Galotti [196] have discussed the individual differences in problem solvers. The prior experiences of problem solvers enrich their mental corpus of problem schemas, enabling them to recognize different problem states, and move faster towards implementation. *Expert programmers* have been found to have this characteristic [204]. They are persistent, and their mental models of their program comprehension exhibit the following five characteristics: hierarchical and multilayered, explicit mapping between layers, recognition of basic patterns, well-connected internally, and well-grounded in the program text. They also choose and mix their richer mental models in an opportunistic way [201].

Experts in any domain are able to more easily pick up more perceptual information, recognize more patterns, create more hypotheses, perform skills, and also represent the problems at more deeper and abstract levels. *Expert programmers* have good problem solving skills, determination, and persistence. They gather clues, in the form of facts and information to help in problem solving, and are also efficient planners [194]. They are also more likely to reflect and check errors in their thinking. Expert programmers have the habit of breaking down the problems into minor sub-problems [205].

Problem solvers with higher cognitive flexibility [206] and cognitive complexity can consider more alternatives, and hence, are better experts. The epistemological beliefs of the problem solvers about the nature of problem solving also affect their natural ways of approaching the problems. The stages of cognitive development discussed later in Section 6.1 effect these beliefs.

**Pedagogic Implications**

Jonassen [193] and Linda S. Gottfredson [207] have consolidated earlier research on problem solving and highlighted the distinctions between academic and practical problems. These differences are given in Table 4.8.
Table 4.8: A Comparison of typical academic and real life problems

<table>
<thead>
<tr>
<th>Academic Problems</th>
<th>Real life practical problems</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Tend to be formulated by other people</td>
<td>1 Require (re)formulation.</td>
</tr>
<tr>
<td>2. Well-defined or well-structured</td>
<td>2 Ill-defined or ill-structured</td>
</tr>
<tr>
<td>3. Tend to be complete. Presented with all the parameters and constraints. Usually consist of a well-defined initial state, a known goal state, and a constrained set of logical operators.</td>
<td>3 Require information seeking. One or more elements of the ill-defined problem are unknown or not known with certainty. The goals of real-life practical problems are usually vaguely defined with unstated constraints.</td>
</tr>
<tr>
<td>4. Typically posses only a single answer</td>
<td>4 Usually possess multiple acceptable solutions.</td>
</tr>
<tr>
<td>5. Tend to encourage single method of obtaining a correct answer</td>
<td>5 Allow multiple paths to solution.</td>
</tr>
<tr>
<td>6. Require application of a finite number of concepts, rules, and principles</td>
<td>6 Present uncertainty about useful and usable concepts, rules, and principles as well. Further, in case of ill-defined problems, the relationships between concepts, rules, and principles may be inconsistent between cases.</td>
</tr>
<tr>
<td>7. Divorced from ordinary experience</td>
<td>7 Embedded in and require prior experience. This requires the problem solver of ill-structured problem to distinguish important from irrelevant, and construct a problem space for generating solutions.</td>
</tr>
<tr>
<td>8. Tend to be of little or no intrinsic interest</td>
<td>8 Require motivation and personal involvement</td>
</tr>
</tbody>
</table>

Real-life ill-defined problems are not constrained by the content domain, may require the integration of several content domains, their solutions are not predictable or convergent, possess multiple criteria for evaluating solutions, and no explicit means for determining appropriate action. They require the solver to express personal opinion or belief, make judgments, and also defend them. Earlier it was believed that experiences with well-defined problem solving easily transferred to solving ill-defined problems. However, research in problem solving has demonstrated that performance on well-defined problems is not correlated with performance on ill-defined problems.

In our recently concluded survey, “Software developers - (How) Did your college help you in your development?” (Table A10.2 (i) part-II, Appendix A10), a large fraction felt that as compared to all other kind of academic engagements, their student projects (78%) did much better to develop their problem solving skills. This was followed by laboratory work (59%), thinking oriented lectures (51%), discussions with other students (49%), homework (37%), research literature survey (36% each), industrial training (33%), and discussions with faculty (31%). Discussions with others and traditional knowledge transmission oriented lectures were found to be least effective in this regard by the respondents.
Complex Problem Solving Techniques

Literature on ill-defined problem solving offers some excellent general purpose techniques that have been used in various professions, especially management and design. These techniques essentially help in analysis of complex ill-defined problems. Some of these techniques are given in Table 4.9.

Table 4.9: Some techniques for solving complex ill-defined problems

| 1. | Flow charts (understanding how a process works) |
| 2. | Concept mapping |
| 3. | Systems diagrams (understanding the way factors affect one-another) |
| 4. | SWOT (Strength, Weakness, Opportunity, and Threat) analysis |
| 5. | Appreciation (extracting maximum information from facts by repeatedly asking ‘so what?’) |
| 6. | 5 Why’s (asking "Why?” five times, successively, to understand the ultimate root cause), |
| 7. | Cause and effect diagram (identifying possible causes of problems) |
| 8. | Affinity diagrams (organizing ideas into common themes) |
| 9. | Appreciative inquiry – 4D approach (solving problems by looking at what's going right in four phases of problem solving: Discovery, Dream, Design, and Deliver) |

Many of these techniques, especially, flow chart, system diagram, 5 why’s, and cause and effect diagram are already being used by many software developers in the industry. Flow chart and systems diagrams are already being used in some computing courses. Various kinds of conceptual modeling diagrams, especially UML diagrams are used by software designers. Metzger [157] recommends the usage of Nassi-Shneiderman diagram and Warnier-Orr diagrams for program design conception stage. Most of the other techniques in above list can also be very effectively used by all software developers for various activities of software development. Hence, computing students should be well exposed to these techniques through their curriculum. Integration of these and some other similar techniques in software development education is on our future agenda. Active engagement in our proposed framework of pedagogic engagements incorporates using these techniques in various problem solving activities (ref: Table 8.5). Further, they should also need to learn to adapt existing techniques, and if required, also develop new techniques especially diagrammatic techniques.

Further, with reference to software development, analyzing and solving complex ill-defined problems usually requires approaching problems and solution from a systems-level perspective. The details of systems-level perspective are discussed in Section 6.3. Evolutionary nature of software development also makes it necessary to continuous reflect upon the problem and iterate.
over the solutions. Hence, in the context of software development, reflective thinking, discussed in Section 5.2 is also very important for complex ill-defined problem solving.

**Section 4.6: Chapter Conclusion**

In this chapter we discussed that the basic competence for software developers includes skill, rules, and knowledge related to various technical activities of software development, application domains, communication (mainly in terms of understanding user needs and knowledge sharing with different stakeholders), computational thinking, and general purpose complex ill-defined problem solving.

Repeated practice with similar problems enhances skill. Variety, richness, and complexity of problem cases actively examined, solved, and/or critiqued by learners expand their ‘rule base’ and ‘actionable knowledge base,’ and hence, their competence. The implicit rules, their limitations, and exceptions are learned and refined by reflective practice. Problem cases with subtle differences can result in rule failure to solve problems. Such situations create conditions for the learner to recognize the limitations and exceptions to their rules and further refine them.

Hence, during software developers’ education, a large variety and number of such experiences are necessary for them to build a sophisticated, rich, and actionable mental repository of implicit rules. *No single method of teaching and learning can help the learner to build such a repository. Only a proper integration of active, integrative, reflective, and collaborative engagements with theoretical, as well as practical, problem cases can help to create a large number of such varied opportunities.*

As per our studies discussed in this chapter, *student-centric pedagogical activities, especially projects* have been found to be most effective for development of all basic competencies, discussed in this chapter. Well designed projects, if administered properly can engage the students in a variety of learning oriented tasks. We further discuss this issue in seventh, eighth, and ninth chapters.
Rule or rule-base refinement is a knowledge-based activity that requires revising the mental model of the problem, knowledge domain, and/or the mapping between the two. This exercise is driven by the mental faculties of attentions, critical analysis, reflection, and also creativity and innovation. Hence, we call these mental faculties the competency driver-habits of mind. In the next chapter, we carry out a detailed discussion about these faculties as we see them in the context of software development education.