Teaching Effective Requirements Engineering for Large-scale Software Development with Scaffolding

Maria Feldgen
Facultad de Ingenieria
Universidad de Buenos Aires
Argentina
mfeldgen@ieee.org

Osvaldo Clua
Facultad de Ingenieria
Universidad de Buenos Aires
Argentina
oclua@ieee.org

***Abstract*—The hardest part of building a software system is deciding what to build. Errors in this part of the work are, overall, the most serious in software development, and the hardest to repair. Therefore requirements elicitation is arguably the most critical. The requirements that drive the decision towards building a distributed architecture for large-scale systems are usually of a non-functional nature, such as scalability, openness, heterogeneity, availability, reliability and fault-tolerance. Requirements are essential to understand concepts about software architectures and software patterns. Therefore teaching large scale software systems design requires covering significant material while ensuring students experience the wicked nature of complex systems. This paper describes a unified project experience with focus on requirements engineering that addresses many of the areas required in a distributed systems development experience. The most important lesson learned is that students benefit from being immersed in and reflecting upon carefully planned activities of large-scale software design with emphasis on its inherent complexity. The planned experience is based on the principles suggested by research related to learn about complex physical and social systems.**

Keywords—complexity; distributed software systems; requirements elicitation; scaffolding.

I. INTRODUCTION

The hardest part of building a software system is deciding what to build. Errors in this part of the work are, overall, the most serious in software development, and the hardest to repair [1]. Therefore requirements elicitation is arguably the most critical part. The requirements that drive the decision towards building a distributed architecture for large-scale systems are usually of a non-functional nature, such as scalability, openness, heterogeneity, availability, reliability and fault-tolerance [2]. Therefore, teaching distributed systems architectures and design imply deliberate practice in functional requirements (FR) and essentially non-functional requirements (NFR) elicitation. Requirements are indispensable to understand concepts about software architectures and software patterns. The architecture is the first artifact that represents decisions on how requirements of all type are to be achieved and the manifestation of early design decisions that are hardest to change. [3] [4] [5]

The requirements engineering process (RE) approach proposed in this paper starts with a scenario description captured in different types of formats, such as text narrative, videos, and supplementary information of the applications domain. Classroom assessment techniques (CATs) and scaffolding are used to give and get feedback of students learning and to identify the critical weaknesses in the approaches that novices typically use when attempting to solve difficult design-oriented problems [6][7]. The focus is on a CAT set known as “Assessing skill in problem solving” [8]. The CAT elements are guidelines and activities to solve the RE process. The solution involves a series of sessions where students work individually and in teams. Students are exposed to different methods of requirements elicitation and decision-making processes in an iterative approach. By the time teams of students reach consensus on FRs and NFRs they write the requirements specification using use cases and UML 2.x diagrams [9]. Finally, a throwaway prototype of the application is developed to validate the requirements. The next steps are the design, implementation, and testing of a middleware using the prototype.

The session’s outcomes are used as formative assessment tools for feedback. Outcomes and in-class observation show faculty when it is beneficial to prompt students by questions to focus on important events evident in the situation (thus activating the perception process), to relate these events and their impact to what is already known from other similar situations (activating the memory recall process), and to reach useful conclusions (activating the reasoning process) based also on the results of their previous outcomes. Such cognitive activity is expected to introduce students in practices of reasoning about systematic exploration of possibilities and scenario-based reasoning to explore assumptions and consequences, stressing their ability to see patterns of meaningful information as experts do. As well as teamwork and communications skills are developed.

This work describes the experience and the learning environment that integrate the aspects of a large-scale distributed system design through a significant unified project experience characterized by an emphasis on requirements engineering. Students are exposed to specific obstacles in a controlled fashion, providing a valuable and positive experience without overwhelming them and jeopardizing the

978-1-4799-3922-0/14/$31.00 ©2014 IEEE

final project success. Finally, formative and summative assessment results and outcomes from a subsequent capstone course corroborate that the five design principles suggested by research related to help students learn scientific ideas about complex physical and social systems [10] can be applied to software design with comparable results.

II. Large Scale Systems

Recent advances in computing/communication technology have empowered to replace centralized management of operations by operations that are partially or fully decentralized in industry and enterprises. Some of these systems are complex systems created by integrating and orchestrating independently managed systems [11]. The design of this class of large-scale systems also called systems of systems (SoS) [12], is significantly more difficult than for centralized control systems, because the SoS elements are operationally and managerially independent. The systems in a SoS work together and interact according to agreed-upon protocols.

Usually, large-scale systems have inherent complexity depending on the number and type of relationships between the system's components and between the system and its environment. Complex systems are nondeterministic, and systems characteristics cannot be predicted by analyzing the system's components. Consequently, the behavior of a SoS is emergent and impossible to anticipate completely. [11]

The engineering best practices for this type of software systems suggest to develop a middleware that, through the provision of proper features, supports distributed applications by masking the distribution and heterogeneity of the execution and networking environment [13].

A. Middleware

Middleware is an important class of technology which contributes to decrease the cycle-time, level of effort, and complexity associated with developing high-quality, flexible, and interoperable distributed systems. Middleware software modeling integrates several software architectures that are the core concepts of distributed systems [14] such as client/server, service oriented, component-based, product line architectures with protocol design and other concepts, such as separation of concerns, reuse, and IDL/interfaces design. Application systems are developed using reusable software (middleware) component services, rather than being implemented entirely from scratch for each use.

B. Design Issues

Making sense of the characteristics of a complex system is a difficult task because it requires one to think abstractly and often challenges current beliefs regarding phenomena. Consequently, large-scale software design falls into the category of wicked problems. Peter DeGrace et al. [15] pointed out that many of the systems problems facing software developers have all the characteristics of these problems. New requirements emerge out of the design decisions, making it impossible to create a clean separation between requirements and design. The design process lacks any analytical form and because of this there may be a number of acceptable solutions to any given problem. In consequence the process of design will rarely be "convergent", in the sense of being able to direct the designer towards a single preferred solution and suggesting that it is potentially unstable [16]. In conclusion, design problems create the following dilemma: (1) one cannot understand the problem without information about it; (2) one cannot gather information meaningfully unless the problem is understood; and (3) one cannot understand the problem without having a concept of the solution in mind.

Horst Rittel and Melvin Webber [17] observed that wicked problems couldn't be resolved with traditional analytical approaches. As well as the SEI ULSS report [18] and the U.K. LSCITS initiative [19] argued that current traditional centralized engineering methods are inadequate. For example, large-scale systems will have a wide variety of stakeholders with unavoidable different and conflicting requirements. Moreover, LSCITS view according to D. Cliff [20] (project director) focus on that is not so much that the initiative are "redefining" software engineering but rather that they are looking to extend established software engineering tools and techniques in novel and useful ways.

C. Requirements Engineering

The process of eliciting requirements is critical to system development but often overlooked or only partially addressed, in general and particularly by students. This may be a result of being one of the most difficult and least understood activities [3].

Requirements engineering is the process of discovering, documenting, analyzing, and checking the constraints and services. There are two complementary types of requirements. First, the user-centered or functional requirements (FR) that concentrate on the functionalities provided to the system's users. Second, the context-centered or non-functional requirements (NFR) which can be system, process and human requirements. NFR are mainly system's constraints and behavioral properties that may affect greatly the operational environment and design choices a developer may pursue during the development of the software. They include among other things, operational environment: hardware and software interfaces, accuracy, performance: timeliness and storage requirements, security, reliability, maintainability, portability, robustness, and usability. FR describes mainly the visible and external input and output interactions with the system under consideration, whereas NFR are those that impose special conditions and qualities on the system to develop. Consequently, system acceptance testing is based on both functional and non-functional system's requirements. [21]

Literature [22] [23] [24] [25] argued that ineffectively dealing with NFR's has led to a series of failures in software development and that these requirements are the most expensive and difficult to cope with. Nevertheless they have surprisingly received little attention in the literature in spite of that it is possible to find standards that can offer some guidance on eliciting NFRs. But it is difficult to find guidance on how one might to integrate them into design.

III. Significant Issues of the Course

Teaching distributed systems design requires covering a broad range of topics in the areas of distributed systems

paradigms, software design, software architectures, patterns, as well as potentially covering related disciplines such as system safety, user interface design and dependability. The only way to integrate the aspects of a large-scale distributed system design is that students are exposed to a significant unified project experience. This type of experience addresses many of the required topics and methodical engineering processes while it highlights the "wicked nature" a distributed software system posses.

Special attention must be devoted to the requirements engineering processes emphasizing the importance of a careful analysis of FR and NFR to guarantee a successful software product and to provide a valuable and positive experience.

Another issue to be considered is that students lack some skills that hinder learning concepts relevant to understanding complex systems. These are also reported in literature:

A. *Difficulties in understanding complex systems concepts.*

According to a great deal of research on expertise and complex systems [26] [27] [28] [29], making sense of a complex system is a difficult task because it requires to think abstractly and often challenges current beliefs regarding phenomena. Important concepts in complex systems are counterintuitive or conflict with commonly held beliefs and are common patterns observed when students designed distributed software systems. For example, many people tend to favor explanations that assume central control and deterministic causality and refuse ideas describing various phenomena in terms of self-organization, stochastic, and decentralized processes.

The characteristics of complex systems make them particularly difficult to understand when focused on the perceptually available structures. Invisible, dynamic phenomena pose considerable barriers to understanding. Students tend towards very simple causal explanations of complex phenomena. When students reasoned about effects, they missed the connectedness within the system and the complex causal relationships. One reason for this is that learners tend to focus on the structure of systems rather than on the underlying function.

B. *Lack of expert design strategies.*

David Wright's [30] literature review highlights the significant differences between expert and novices strategies when designing software systems. In spite of the body of research about learning to design software systems is scarce, researcher agree that it takes a lot of deliberate reflective practice to move from the mindset of a novice to the mindset of an expert.

Students (novice designers) view design problems much differently than expert designers [31]. In addition, complex system heuristics are difficult to develop because they often go against the linear system heuristics that appear to be more grounded in students' previous experiences [9].

C. *Lack of problem solving abilities.*

These are the problems reported when students are engaged in complex, ill-structured problem-solving tasks [32]:

Students have a tendency to treat all problems as tame, perhaps because these problems are easier to solve, reinforced by the lack of understanding about wicked problem dynamics and the tools and approach they require [33]. Rowland [34] found that novices do not see the problem as being ill structured and thus assume that the information and variables are clearly specified. They do not think beyond the written description of the problem.

Students fail to apply knowledge learned in one context to another, especially when solving complex problems. While students' difficulties in problem solving are partly attributed to misconceptions or superficial conceptions of domain knowledge, they are due to a lack of metacognitive and structural knowledge. Structural knowledge is knowledge of how concepts within a domain are interrelated. Structural knowledge requires integration of declarative knowledge into useful knowledge structures.

D. *Deficiences in communication skills for the requirements engineering process.*

Students are inexperienced in eliciting requirements and they need to learn the required communications skills for the job. The communication skills for requirements engineering process come in two forms [35]:

Oral communication skills are needed to interact with the stakeholders in a language that they can understand. Moreover, interviewing skills are needed to ask the right type of questions during the elicitation of the requirements. As well as negotiation skills since much of the requirements are prioritized through negotiations with relevant stakeholders.

Written Communication skills are needed to write specifications that exhibits quality attributes such as being precise, concise, unambiguous, consistent, complete, and thus correct.

IV. PROCEDURE

David Wright [30] suggests a framework that provides novice software designers with a simple yet rich and flexible guide to help them quickly cultivate the key processes and behaviors of expert designers. Furthermore, Kolodner [36] describe deliberate reflective practice of targeted skills involves learning in the context of doing that includes monitoring one's own experience of learning, and frequent, timely, and interpretable feedback. Deliberate means that the skills are practiced in a context that promotes learning; reflective means that their practice is discussed and lessons drawn out from that discussion. Feedback is timely when one can use it to determine the quality of one's conceptions and, comes right after one has performed some task and is easy to interpret. In addition, embedded scaffolds are designed to support certain learning and inquiry processes. Our suite of learning tools are:

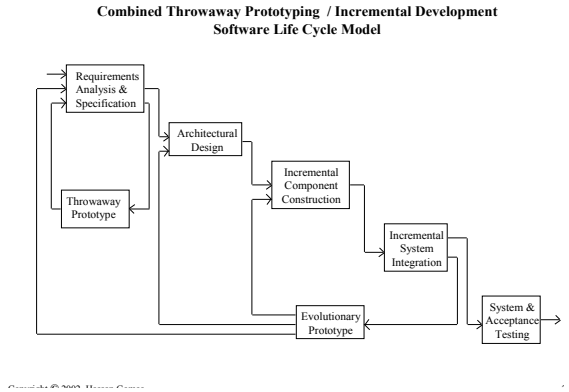
A. *The project experience*

It is a semester master course that aims at teaching students how to develop distributed systems within a group of 10 to 20 students. Participants already have basic knowledge about traditional software engineering techniques and concurrent programming. Within the course UML 2.0 modeling, software

architectures, patterns, and distributed systems paradigms are given.

The project experience is a whole class project with weekly deliverables of increasing complexity that follow the combined throwaway prototyping with incremental development software life cycle model (Fig. 1) suggested by H. Gomaa [37]. The throwaway prototype of an application is developed to clarify the requirements and to develop the services (interfaces) provided by the middleware. It is a simulation of the real application to show synchronous and asynchronous interactions, concurrency, decentralized control, self-organization and emergent behavior. After the requirements are understood and a specification is developed an evolutionary prototype of the middleware is build in several iterations.

Fig. 1. Software life cycle [37]



The life cycle activities serve as a flexible and adaptable "design thinking" process for guiding students through the design decision-making process. The process explicitly incorporate scaffolding for looking at the big picture of the system, the use of rules, principles, and/or practices for examining the design and generating solution alternatives, selecting and evaluating candidate alternatives, and reflecting on what has been accomplished in an iterative approach.

The life cycle activities are supported with CATs related to Assessing Skills in Problem Solving [8]. CATs outline the activities of each life cycle phase, provide guidelines and procedures of "what to do" in the form of reflective writing (written arguments, annotations, and questions) and sketching or informal diagrams [6] [7] [38] [39]. Heuristics to map textual elements in the case documentation to requirements are given for the requirements engineering phase. The outcomes are for communicating ideas and decisions for the in-class discussions and peer-review sessions.

B. Scaffolding

- *Question prompts.* In addition to the life cycle activities, the students are receiving prompts consisting of elaboration and metacognitive questions for each activity. Research reported in [40] had found question prompting to be an effective instructional strategy for directing students to the most important aspects of a problem, as well as encouraging self-explanation, elaboration, planning, monitoring and self-reflection, and evaluation.
- *Peer review.* The peer review mechanism was designed to enable students to see multiple perspectives from peers' reports and help them notice things they might not have thought about previously. By reviewing their peers' thinking, students are supposedly compelled not only to attend more closely to their peers' ideas, rationales, plans and solutions, but also to their own for self-reflection.
- *Expert modeling.* Presenting students with expert's reasoning during discussion meetings provides expert modeling examples. This support mechanism is expecting to offer students an opportunity to observe the expert's reasoning, which they would compare with their own reasoning. It has been assumed that the comparison would result in disequilibrium.

C. Project summary

The goal of the project is the design and implementation of a middleware for a distributed application, where the major part of application development is reduced to assembling domain-specific services that comply with a set of declaratively specified policies.

The project is an open-ended design exercise in which students follow the same process, demonstrate a certain set of technical skills, apply concurrent and distributed paradigms and experience design process challenges. Faculty's role is as facilitator and stakeholders.

The project of the last semester was an automotive assembly plant. Students were given the problem statement and additional documents containing stakeholders' scenario description of the sections of the assembly line, videos of the assembly line activities, and supplementary application domain information. The scenario description was taken from the book Handbook of Simulation chapter 15 [41]. An excerpt from the narrative is:

"Four major categories can also be identified in automotive manufacturing plants: equipment and layout design issues, issues related to variation management, product-mix sequencing issues, and other operational issues. The equipment and layout design issues include typical problems such as number of machines required, cycle-time verification, identification of buffer storage locations, buffer size (strip conveyors and buffers for sequencing) analysis, and conveyor length and speed determination. Examples of typical problems in the variation management area are repair and scrap policy analysis, order-size variation, and paint booth scheduling. The product-mix sequencing issues typically include trim line and body shop sequencing, shift scheduling, and trim and final assembly line balancing. In the other operational issues area, typical applications involve priority assignment at traffic intersections, assembly-line sequencing, and shift and break scheduling.

An automotive assembly plant typically has three major sections with respect to the stages of the assembly process: body shop, paint shop, and trim and final assembly. Each of these areas has different types of processes with unique features. Following is a discussion of the typical issues…."

D. Project requirements engineering.

As stated before, the requirements engineering process is challenging for expert analysts and more for inexperienced students. The activities in requirements engineering are requirements elicitation, specification and validation.

The activities in requirements elicitation for discovering FR, NFR, and constrains are:

- background and domain knowledge preparation: the analyst must understand the background and the domain knowledge of the application that is being developed;
- passive observation: the analyst observe the people at work using videos;
- gathering the requirements: The processes are: (1) gathering requirements from the users' (stakeholders') documentation; (2) questionnaire to ask the stakeholders to clarify requirements and conflicts between NFR such as scale and performance; (3) interviews with the stakeholders to complement the questionnaire and to negotiate.

The general objective of requirements analysis is to understand the requirements, and detect their overlaps and conflicts. A prerequisite of this activity is to understand the rationale underneath the requirements (i.e., why particular requirement is selected out of the others or prioritized, what are the business and technical motivations for achieving them). The rationale contains the arguments for and against each alternative requirement, including the FR and NFR [42]. When this information is captured in the rationale for a requirement, it provides reasoning facilities for detecting the conflicts between requirements (between two FRs, two NFRs or FR and NFR) on the software system [43].

Furthermore, NFR presents the developer with a number of tradeoffs, such as centralized versus decentralized control, sequential versus concurrent distribution, service versus server, scale versus performance among other [44] In software engineering of large-scale systems, a tight relationship exists between NFRs and software architectures (SAs). Different NFRs call for different software architectures and middleware. Giving special attention to requirements elicitation, mainly NFRs not only promote a deeper understanding of complexity, it also supplies different scenarios to introduce software architectures and models using the same case with just a few changes in the NFRs.

Being requirements elicitation an imprecise and difficult process students experience most of the underlying difficulties that hinder the identification/definition of the user's needs: [18][45]

- Poor preparation for elicitation and gathering of requirements. More often than not, analysts approach to this activity with little or no preparation at all.
- Misunderstanding about requirements. Analysts understood requirements wrongly or elicited only the core requirements stated by the stakeholders. Ancillary functionality requirements are often missed out.
- Vague requirements. It is easy to miss requirements that are not objective in nature or difficult to interpret or understand, e.g. "easy of use", while establishing the requirements.
- Modeling issues. The diagrams are great if used properly in understanding the system. But often analyst draw diagrams in a complicated manner making it difficult to make any sense out of them and avoid narration.
- Prioritization of requirements. This aspect is often overlooked and the fulfillment of requirements would be based on the convenience of the project team rather than the necessity of the end users.
- Tracing and tracking of requirements. It is not uncommon to miss out some requirements specified by the end users.

E. Heuristics for requirements elicitation from the description

Requirements can be identified and marked in the given description with the following heuristics. The heuristics described here are that transform stakeholder's documentation into requirements [46][5]. Some examples for Use Case elements are:

- Headings of sections or subsections typically contain names of Use Cases.
- Phrases like "only by", "by using", "in the case of" can be markers for Use Case preconditions.
- Phrases like "normally" "with the exception", "except" can mark Use Case extensions.
- Nouns define real objects.
- Activities or system functions are those elements that were marked as features that contain a verb

Procedures and heuristics are not enough to beat these difficulties and the novice's simple causal explanations of complex phenomena.

F. Requirements engineering launcher activities.

Gentner et al. [47] opined that specific instructional intervention, such as accelerated example-based learning, might improve students' ability to solve problems in an expert-like manner. Experts reason according to principles whereas novices tend to look mostly at visible surface characteristics of a problem. The higher-level cognitive processes needed for this task is analytical thinking, integration of ideas and logical reasoning. The recommended activities to achieve this type of thinking are [48]:

- Elaborating on content to add details and examples, relating new material to what one already knows. Elaborations are incorporated into existing knowledge, reorganizing mental models and thereby improving both the domain expert's

and the requirements engineer's understanding of the domain.

- Explaining ideas and concepts to relate the why and how of issues being explained to what is already known.
- Asking thought-provoking questions to encourage students to think with and about the material, which are thought to establish elaborate cognitive representations of knowledge, facilitating the connection of new ideas with prior knowledge.
- Argumentation through evidence based reasoning to substantiate or change views on one's claims. Beside its convincing power, argumentation can be used to explore issues and create deeper understanding of them.
- Resolving conceptual discrepancies between an individual's own understanding and other's views of issues, also through substantiating and explaining one's own views.
- Modeling of cognition by taking skilled use of questioning and explaining by others as an example to refine one's own skills.

An approach commonly used with medical students is Case-based reasoning [50]. Case-based reasoning [49] refers to reasoning based on previous experience. It means solving a new problem by adapting an old solution o merging pieces of several old solutions. The goal is helping students to "re-index" old experiences and to abstract out generalizations from experiences and reviewing background knowledge with a different approach [50].

The launcher or warm-up activities' approach for the project starts with a set of short cases to analyze. Students in informal teams elicit, analyze, validate, and document the requirements of short concurrent problems as homework following the CATs guidelines. Students had to apply background knowledge of concurrency concepts and software engineering techniques. Each case is an open-ended problem of the type they had solved using concurrent paradigms in a previous subject. [7]

To have a close observation of how students' work most of the team meetings are held in class. Student teams' CATs reports are used as supportive information for brainstorming in the classroom to identify the requirements, to elaborate questionnaires for the stakeholders, and to interview the stakeholders; and for peer-review to reach consensus about the requirements elicitation and specification. The meetings are a means of practicing and improving critical thinking skills, personal expression, and tolerance of others' opinions. The homework activities and the in-class discussions are complemented with question prompts. Question prompts direct students to the information they need to take into account conflicting requirements or hidden needs. In addition, oral and written communication skills are developed.

After students reach consensus or get stuck in their debates, faculty in their role as expert designers and mentors, participate validating the outcomes and revealing hidden relational properties of the problem. Experts notice features and meaningful patterns of problem solving that are often not noticed by novices. The expert-modeling scaffold smooth the way for introducing student's in practices of reasoning about systematic exploration of possibilities (finding gaps, or finding unexplored relationships between problems and potential approaches) and scenario-based reasoning to explore assumptions and consequences. These practices show them how to revisit or reconsider a decision, particularly when such reconsideration would entail reversing it.

Each session ended with a self-reflection writing activity with a few reflective questions. Self-reflection is an important mechanism to supplement the expert modeling mechanism and allows students to observe an expert's reasoning facilitating students' reflection on the gaps between student's and the expert´s reasoning [40]. Providing time after each session for rethinking and revising and perhaps reanalyzing and reporting once again is an effective technique to promote self-monitoring, self-regulation, and self-directed learning.

CATs act as guidelines and as formative assessment tools to give feedback and "just in time" mentoring during the meetings for reflection, in the sense of thinking about something again.

The next activities of the requirements engineering process follow the same outline. The same strategies are applied for the whole project providing temporary support when concepts and skills are being first introduced to students. The repetitive character of the life cycle iterations gradually increases students' confidence, in accordance with the principles of scaffolding. Scaffolding is gradually added, then modified and finally removed as students developed their self-awareness and self-regulatory abilities [40].

G. Project outcomes and middleware evolution.

After the requirements engineering phase, the system is designed and developed in four versions. The outcomes of each version using architectural design are:

- The first version of the distributed application is the design and development of a working throwaway prototype of the application. The prototype shows the interfaces between application's active objects (distributed objects) and their interactions (communication, coordination and synchronization) on a single computer (using operating systems IPCs).
- The second version is the first evolution of the middleware that comprises the distribution of subsystems with asynchronous communication patterns over the network using client-server architecture with RPC and the TCP/IP sockets programming API.
- The third version is the next evolution of the middleware with one broker pattern. It includes the distribution of the subsystems with synchronous communication patterns.

- Finally, the fourth version is a middleware with multiple broker patterns (distributed control). It is the tool to analyze how the architecture fulfills the NFR requirements and to analyze emergent behavior in presence of faults or network abnormal functioning.

V. CONCLUDING REMARKS.

Research [10] related to help students learn scientific ideas about complex physical and social systems suggest five design principles that may yield promising learning sciences research: (a) experiencing complex systems phenomena; (b) making the complex systems conceptual framework explicit; (c) encouraging collaboration, discussion, and reflection; (d) constructing theories, models, and experiments; and (e) learning trajectories for deep understandings and explorations.

To conclude we summarize the features of our learning environment using the five design principles:

A. Experiencing Complex Systems Phenomena

Students have opportunities to experience complex systems phenomena in ways that will let them enhance both their ontological and conceptual understandings [10]. The purpose of the student's learning experience is twofold: first through requirements engineering and design of appropriate small "wicked" cases to make evident that (1) complex systems depends on distributed control, cooperation, influence, cascade effects, orchestration, and other emergent behaviors as primary compositional to achieve its purpose; (2) emergent behavior is the inevitable consequence of the independent management, operation, and evolution that characterize this type of systems and is unavoidable in the presence of autonomous constituents; (3) the importance of emergent effects in determining the global characteristics of the systems mainly driven by NFRs. Second to apply these skills to deal with complexity at a larger scale (the project). It provides a robust design experience without overwhelming students and jeopardizing the final project success.

Devoting in-class time to explore requirements engineering and small cases showing different scenarios result in a deeper comprehension and experimentation of the distributed systems concepts and also is a prerequisite for proper understanding of why the discipline exists.

B. Making the Complex Systems Conceptual Framework Explicit

A second principle for designing learning environments and tools is to make the organizing conceptual framework explicit to the student [10]. The cases, project and its life cycle are the organizing and integrating conceptual framework for learning the core concepts related to distributed software systems.

C. Encouraging Collaboration, Discussion, and Reflection

Contemporary views of learning acknowledge important ways that knowledge and beliefs about the world are shaped and constructed in situated and socially mediated contexts [10].

Students solve authentically interesting problems and projects that involve collaborative and cooperative interactions. The learning interactions occur between peers or between peers and experts that involve real-time face-to-face and distributed synchronous or asynchronous computer-mediated communications. The environment involves collaborations and discussions in which students are provided metacognitive scaffolding and questions for reflection such as "What underlying mechanisms might give rise to the observed behavior?" "How sensitive is the outcome to changes in the model's parameters or assumed environment?" "How predictable is the behavior of this system and why?" These collaborative interactions and the construction of shared artifacts of the system and representations of the design help students articulate or reify their ideas about complex systems, help them reflect on the possible limitations of their initial ideas and theories, and help them see how complex systems ideas might be plausible and useful for understanding particular systems of interest.

D. Constructing Theories, Models, and Experiments

A central tenet of constructivist and constructionist learning approaches is that a learner is actively constructing new understandings, rather than passively receiving and absorbing "facts" [10]. The development of the cases and project are the explicit linking of model building and the experimentation when they run and test the implemented solution. It helps students to understand that the development of a complex system is fundamentally grounded on cycles of theorizing, model building, and experimentation, which in turn iteratively lead to further theory and model revisions, and so on.

E. Learning Trajectories for Deep Understandings and Explorations

Students exhibit far transfer by applying these understanding across subject areas in our subsequent capstone course. The goal of the capstone course is to develop a system for a laboratory of the University. It is a real project with real stakeholders and diverse domains. The complex systems concepts learned in the previous course (e.g., core concepts such as decentralization, feedback, self-organization, emergence) form a conceptual or representational toolkit that students use and enhance in the capstone course.

ACKNOWLEDGMENT

The work report here was partially funded by the UBATIC 2011-2014 program 200201001003 grant of the University of Buenos Aires.

REFERENCES

[1] F. P. Brooks Jr., "No Silver Bullet: Essence and Accidents of Software Engineering". IEEEComputer,10-19, April 1987.

[2] R. Bahsoon and W. Emmerich, "Economics-Driven Architecting for Non Functional Requirements in the Presence of Middleware" in Relating Software Requirements and Architectures, chapter 20, P. Avgeriou et al. (eds.) Springer-Verlag, 2011, pp. 353-371.

[3] M. G. Christel and K.C. Kang, "Issues in Requirements Elicitation", Technical Report, CMU-SEI-92-TR-012, 1992.

[4] S. Ullah, M Iqbal, and A.M. Kahn, "A survey on issues in non-functional requirements elicitation", Computer Networks and Information Technology, July 2011.

[5] M. W. Maier and E. I. Rechtin, The Art of Systems Architecting, CRC Press, Boca Raton, FL, 2002.

[6] M. Feldgen and O. Clua, "Distributed Systems Design supported by Reflective Writing and CATs ", FIE 2011, 2011.

[7] M. Feldgen and O. Clua, " Scaffolding Students in a Complex Learning Environment", FIE 2013, 2013.

[8] T. Angelo and K. P. Cross, Classroom Assessment Techniques. A Handbook for College Teachers, 2nd. Edition. Jossey Bass, 1993.

[9] UML 2.x, http://www.uml.org/

[10] M. J. Jacobson and U. Wilensky, "Complex Systems in Education: Scientific and Educational Importance and Implications for the Learning Sciences", The Journal of the Learning Sciences, Lawrence Erlbaum Associates, Inc., vol 15 No. 1, 2006, pp. 11–34.

[11] I. Sommerville, D. Cliff, R. Calinescu, J. Keen, T. Kelly, M Kwiatkowska, J. McDermid, and R. Paige, "Large-Scale Complex IT Systems", Communications of the ACM, Vol 55, Nº 7, 2012, pp. 71-77

[12] M.W. Maier, "Architecting Principles for systems-of-systems", Systems Engineering, Vol 1. Issue 4, Wiley, 1998, pp. 267-284.

[13] R. E. Schantz and D. C Schmidt, "Middleware for Distributed Systems", BBN Technologies, http://www.dre.vanderbilt.edu/~schmidt/PDF/middleware-encyclopedia.pdf

[14] P. Grace, G. Coulson, G. S. Blair, and B. Porter, "Deep middleware for the divergent grid". In Proceedings of the IFIP/ACM/USENIX Middleware 2005, 2005.

[15] P. DeGrace and L. H. Stahl, Wicked Problems, Righteous Solutions. Prentice Hall, Yourdon Press, 1990.

[16] D. Budgen, Software Design, 2nd Edition, Addison-Wesley, 2003

[17] H. Rittel and M. Webber; "Dilemmas in a General Theory of Planning", Policy Sciences, Vol. 4, Elsevier Scientific Publishing Company, Inc., Amsterdam, 1973, pp. 155-169

[18] L. Northrop, P. Feiler, R. P. Gabriel, J. Goodenough, R. Linger, T. Longstaff, R. Kazman, M. Klein, D. Schmidt, K. Sullivan, and K. Wallnau, "Ultra-Large-Scale Systems. The Software Challenge of the Future", SEI Report, Carnegie Mellon, 2006, http://resources.sei.cmu.edu/asset_files/book/2006_014_001_30542.pdf

[19] D. Cliff, R. Calinescu, J. Keen, T. Kelly, M. Kwiatkowska, J. Mcdermid, R, Paige, and I. Sommerville, The U.K. Large-Scale Complex IT Systems Initiative 2010; http://lscits.cs.bris.ac.uk/docs/lscits_overview_2010. pdf

[20] G. Goth, "Ultralarge Systems: Redefining Software Engineering?", IEEE Software March/April 2008, pp. 91-94.

[21] K. Saleh and A. Al-Zarouni ,"Capturing non-functional software requirements using the user requirements notation", The 2004 International Research Conference on Innovations in Information Technology, 2004, pp. 222-230.

[22] L. M. Cysneiros and J. C. S. do Prado Leite, "Nonfunctional requirements: From elicitation to conceptual models", IEEE Transactions on Software Engineering, Vol 30 No. 5 2004, pp.328-350.

[23] L. Chung. E. Yo, and J. Mylopoulos, Non- Functional Requirements in Software Engineering, Boston: Kluwer Academic Publishers, 2000.

[24] D. J. Finkelstein, "A comedy of Errors: The London Ambulance Service Case Study," IEEE Computer Society Press,1996, pp. 2-5.

[25] A Matoussi, R. Laleau, "A Survey of Non-Functional Requirements in Software Development Process," TR-LACL-2008-7, 2008..

[26] M. J. Jacobson, "Problem Solving About Complex Systems: Differences Between Experts and Novices", B. Fishman & S. O'Connor-Divelbiss (Eds.), Fourth International Conference of the Learning Sciences. Mahwah, NJ: Erlbaum, 2000, pp. 14-21.

[27] P. J. Feltovich, R. J. Spiro, and R. L. Coulson, "The nature of conceptual understanding in biomedicine: The deep structure of complex ideas and the development of misconceptions", The cognitive sciences in medicine Cambridge, MA: MIT, 1989, pp. 113–172.

[28] M. Resnick, "Beyond the centralized mindset". The Journal of the Learning Sciences, 5, 1996, pp. 1–22.

[29] U. Wilensky and M. Resnick, "Thinking in levels: A dynamic systems perspective to making sense of the world", Journal of Science Education and Technology, vol. 8 No. 1, 1999, pp. 3–19.

[30] D. R. Wright, "Inoculating Novice Software Designers with Expert Design Strategies", AC 2012-4544, American Society for Engineering Education, 2012.

[31] R. T. Dufresne, W. J. Gerace, P. T. Hardiman, and J. P. Mestre, "Constraining novices to perform expert-like problem analyses: Effects on schema acquisition", The Journal of the Learning Sciences, 2, 1992, pp. 307-331.

[32] X. Ge and S. Land, "Scaffolding Students' Problem-solving Processes on an Ill-structured Task using Question prompts and Peer interactions", Educational Technology Research and Development", Vol 51, Issue 1, 2003, pp. 21-38.

[33] J. Conklin, "Wicked Problems and Social Complexity", CogNexus Institute. http://www.cognexus.org/id42.htm .

[34] G. Rowland, "What do instructional designers actually do? An initial investigation of expert practice", Performance Improvement Quarterly, Vol. 5 No.2,1992, pp. 65-86.

[35] D. Zowghi and S. Paryani,"Teaching Requirements Engineering through Role Playing: Lessons Learnt", Proceedings of the 1th IEEE International Requirements Engineering Conference, 2003.

[36] J. L. Kolodner, "Facilitating the Learning of Design Practices: Lessons Learned from an Inquiry into Science Education", Journal of Industrial Teacher Education, vol. 39, No. 3, 2002.

[37] H. Gomaa, Software Modeling and Design. UML, Use Cases, Patterns and Software Architectures, Cambridge, 2011.

[38] M. Feldgen and O. Clua, "Promoting Design Skills in Distributed Systems", FIE 2012, 2012.

[39] M. Feldgen and O. Clua, "Fostering Writing Habits in Computer Science: A Road to Meaningful Learning", *International Conference on Engineering Education Proceedings*, Oslo, Norway, 2001

[40] X. Ge, L. G. Planas, and N. Er, "A Cognitive Support System to Scaffold Students' Problem-based Learning in a Web-based Learning Environment", Interdisciplinary Journal of Problem-based Learning, vol. 4(1) Article 4, 2010.

[41] O. Ulgen and A. Gunal, "Simulation in the Automobile Industrie",J. Banks (ed.) Handbook of Simulation: Principles, Methodology, Advances, Applications, and Practice, Wiley, 1998, pp 547-570.

[42] J. Burge, J. Carroll, R. McCall, and I. Mistrik. "Rationale and Requirements Engineering", Rationale-Based Software Engineering, 2008, pp. 139–153.

[43] P. Liang, P. Avgeriou, and V. Clerc, "Requirements Reasoning for Distributed Requirements Analysis using Semantic Wiki", 2009 Fourth IEEE International Conference on Global Software, 2009, pp. 388-393.

[44] P. Verissimo and L. Rodrigues, Distributed Systems for Systems Architects. Kluwer Academic Publishers, 2001.

[45] H. Saiedian and R. Dale,"Requirements engineering: making the connection between the software developer and customer", Information and Software Technology, 42, 2000, pp. 419-428.

[46] I. John and J. Dörr, "Elicitation of Requirements from User Documentation", Ninth International Workshop on Requirements Engineering: Foundation for Software Quality. 2003.

[47] D. Gentner, J. Loewenstein, and L.Thompson ,"Learning and transfer: A general role for analogical encoding.", Journal of Educational Psychology, 95(3), 2003, pp. 393-408.

[48] L.A. Scheinholtz and I. Wilmont, "Interview Patterns for Requirements Elicitation", Requirements Engineering: Foundation for Software Quality, LNCS 6606, Springer, 2011, pp.72-77.

[49] J. L. Kolodner, C. E. Hmelo, and N. Hari Narayanan, "Problem-based learning meets case-based reasoning", in International Conference on Learning Sciences. 1996, pp. 188-195.

[50] M. Srinivasan, M. Wilkes, F. Stevenson, T. Nguyen, and S. Slavin, "Comparing Problem-Based Learning with Case-Based Learning: Effects of a MajorCurricular Shift at Two Institutions", Academic Medicine, Vol. 82 (1), pp. 74-82, 2007.