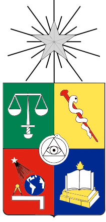


UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

A PRESCRIPTIVE SOFTWARE PROCESS FOR ACADEMIC SCENARIOS

TESIS PARA OPTAR AL GRADO DE DOCTORA EN CIENCIAS
MENCIÓN COMPUTACIÓN

MAÍRA REJANE MARQUES SAMARY

PROFESOR GUÍA:
MARÍA CECILIA BASTARRICA PIÑEYRO
SERGIO OCHOA DELORENZI

MIEMBROS DE LA COMISIÓN:
ALEXANDRE BERGEL
JOCELYN SIMMONDS WAGEMANN
AURORA VIZCAÍNO BARCELÓ

Este trabajo ha sido parcialmente financiado por CONICYT y Fondef

SANTIAGO DE CHILE
2017

Resumen

Tradicionalmente, la ingeniería de software se ha enseñado con clases expositivas. Sin embargo, esta disciplina requiere mucho más que sólo teoría. Con el fin de tratar de entender el estado del arte en este ámbito, se ha realizado una extensa revisión bibliográfica. Además, se llevó a cabo un estudio de la enseñanza de ingeniería de software en las principales universidades de Chile. Los resultados obtenidos indican que hace poco tiempo que estas instituciones han comenzado a enseñar ingeniería de software de una manera teórico-práctica, involucrando a los estudiantes en experiencias de desarrollo de software ya sea en cursos basados en proyectos, como en cursos prácticos de fin de carrera (capstone). Ambos tipos de cursos tienen objetivos distintos; los primeros son generalmente apoyados por procesos más rigurosos, mientras que los últimos son frecuentemente abordados con estrategias de desarrollo ágiles.

Se han propuesto varias estrategias de instrucción y de uso de procesos de ingeniería de software para cursos capstone, pero muy pocos están disponibles para cursos basados en proyectos. Además, aún los procesos más rigurosamente reportados en la literatura no incluyen suficiente detalle para que instructores y estudiantes involucrados puedan reproducirlos en cursos basados en proyectos. Es con el objetivo de llenar este vacío que esta tesis concibe y propone EduProcess: un proceso de desarrollo de software prescriptivo que puede ser utilizado en los cursos de ingeniería de software basados en proyectos que toman parte de programas de computación de pregrado.

En relación al uso de este proceso, esta tesis hipotetiza que EduProcess: (H1) permite que las experiencias prácticas puedan ser reproducidas sin demandar un esfuerzo adicional considerable, y (H2) ayuda a producir resultados positivos en proyectos de software. Este método fue concebido para apoyar a pequeños equipos de software que trabajan de manera distribuida la mayor parte del tiempo y tienen instancias esporádicas de sincronización de sus trabajos. El proceso puede ser utilizado en proyectos de al menos siete semanas de duración. EduProcess incluye dos subprocesos: uno principalmente a cargo de los estudiantes (e involucra a los usuarios y clientes), y el otro a cargo del equipo instruccional del curso y sirve de apoyo a la experiencia de enseñanza-aprendizaje.

La validación de las hipótesis se realizó con un estudio de caso en un curso de la Universidad de Chile. Los resultados obtenidos indican que EduProcess ayuda a los estudiantes a aumentar su coordinación, su sentido de pertenencia al equipo y su efectividad, pero no necesariamente aumenta su productividad. Tambén permite a los equipos hacer un diagnóstico más preciso de su proyecto y priorizar mejor las tareas de modo de maximizar su tasa de éxito. Estos resultados apoyan a la hipótesis H2. EduProcess se aplicó sistemáticamente y con éxito durante cuatro semestres en un curso de ingeniería de software basado en proyectos. Esto muestra que el proceso permite la repetición de estas experiencias, sin demandar de los instructores y estudiantes un esfuerzo considerable. Este resultado apoya la hipótesis H1. En este sentido, EduProcess hace una contribución al avance del estado del arte en el dominio de la educación de ingeniería de software, y específicamente en el proceso de enseñanza-aprendizaje de ingeniería de software en cursos basados en proyectos.

Abstract

Software engineering has been taught over the years mainly using expositive classes, but learning this discipline requires much more than just theory. In order to try to understand the state-of-the-practice in this domain, we performed an extensive literature review and also an analysis about how software engineering is being taught in Chile. The results indicate that universities have started to teach software engineering in a theoretical-practical way, involving the students in software development experiences conducted in project-based and capstone courses. Given that both types of courses have different goals, the former are usually supported by disciplined processes, while the latter are frequently addressed with agile development strategies.

Several instructional strategies and software processes have been proposed for the second scenario, but very few are available for the first one. Moreover, the disciplined processes reported in the literature do not include enough detail to be reproduced by instructors and students involved in project-based courses. In order to help these instructors and students to address these practical experiences, this thesis proposes EduProcess, a prescriptive software development process for use in software engineering project-based courses that are part of undergraduate Computer Science programs.

This thesis work hypothesizes that the use of EduProcess: (H1) makes these experiences repeatable investing an affordable effort, and (H2) helps produce positive results in software projects. This disciplined method was conceived to support small software teams, in which students work in a loosely-coupled way; they work autonomously most of the time and have sporadic instances for synchronizing and integrating their work. It can be used to support software projects of at least seven weeks of duration. EduProcess involves two parallel tracks - one that is mainly in charge of the students (and involves users and clients) and the other that is in charge of the instructional team and supports the teaching-learning experience.

In order to validate our hypotheses, we conducted a case study in a project-based course delivered at the University of Chile. The results obtained indicate that EduProcess helps students increase their coordination, their sense of team belonging and effectiveness, but not necessarily their productivity. It also allows the teams to make a more accurate diagnosis and prioritization in their projects, so students projects have a higher success rate. These results support H2.

On the other hand, EduProcess was systematically and successfully applied during four semesters in this project-based course, which indicates that the process specification allows for the repetition of these experiences, resulting in an effort that is affordable for instructors and students. This result supports H1. In this sense, EduProcess makes a contribution that advances the state-of-the-art in the software engineering education domain, specifically in the teaching-learning process of software engineering experiences of project-based courses.

Publications

The work presented in this thesis has been published in several international conferences. At Frontiers in Education - FIE 2014, I presented [102] the systematic mapping review of software engineering education. A preliminary version of EduProcess was presented at ICER Doctoral Consortium 2015 [96] where I received feedback about it and also received valuable guidance of experts on how to validate it. ICER is an ACM Conference where experts on International Computing Education Research (ICER) gather themselves.

We also presented a status report on how software engineering is taught at Chilean universities. A preliminary version of this report was presented at Chilean Computer Science Conference - JCC 2015. A final version of this work was presented on the Annual Conference on Innovation and Technology in Computer Science Education - ITICSE 2016 [99].

A paper about reflexive monitoring sessions that are part of EduProcess was presented at the Special Interest Group on Computer Science Education - SIGCSE 2016 [98] with a poster in ACM Student Research Competition. At this event, this work won the First Place of the ACM Graduate Student Competition. An extended version of this work was also submitted to IEEE Transactions on Education. It was accepted with major revisions, and we are currently waiting for the second round of evaluations.

We have published other works that are partly related to this thesis. At Grace Hopper Celebration - GHC 2014 [95] a poster about the results of gender differences in software engineering projects was presented. This poster was part of the ACM Student Research Competition. At FIE 2015 [97] we presented a more fine-grained research about gender differences in software engineering education in Chile. At ICSE 2007 [9] we will present a paper on software engineering capstone courses.

To my family...

"Whoever teaches learns in the act of teaching, and whoever learns teaches in the act of learning." Paulo Freire, Pedagogy of Freedom.

Acknowledgements

Throughout my PhD studies, many people have helped me in different ways in the writing of this thesis. Without their support and contributions, this thesis would not exist.

First of all, I would like to thank my supervisors Sergio F. Ochoa and Cecilia Bastarrica for supporting and encouraging me throughout the long process of this thesis work, for their guidance and positive attitude in guiding me and for their patience with me even when I was checking on them more than once a day.

I also recognize, with gratitude, the support from the University of Chile, the two Chilean government projects FONDEF D09I-1171 (ADAPTE project) and FONDEF IDeA IT13I20010 (GEMS project) that has granted me financial support throughout my study. And to CONICYT that granted me a PhD Scholarship (CONICYT- PCHA/Doctorado Nacional/2012-21120544).

Thanks also to all my friends for their moral support, especially my closest colleagues, Luis Silvestre, Fabian Rojas and Alcides Quispe. Thank you for pushing me and hearing my complaints and doubts.

Finally, I wish to express my gratitude to my two kids, Anna y Angelo for their understanding. I am also deeply grateful to my husband Amir for his patience, ears and encouragement.

Contents

1 Introduction **1**

1.1 Teaching Software Engineering with a Practical Approach 2

1.1.1 Types of SE Project Courses 3

1.1.2 Challenges for Implementing Practical Experiences in SE Courses 4

1.2 Problem Statement 5

1.3 Work Hypotheses 6

1.4 Objectives 6

1.5 Research Methodology 8

1.5.1 Information Gathering 8

1.5.2 Definition of the EduProcess 9

1.5.3 Evaluation of the EduProcess 9

1.6 Summary of the Thesis Document 10

2 Related Work **11**

2.1 Teaching Software Engineering 12

2.1.1 Expositive Methods 12

2.1.2 Inverted Classroom 13

2.1.3 Problem-Based Learning Method 13

2.1.4 Hands-on Methods (Learn-by-Doing) 13

2.1.4.1 Hands-on Labs 14

2.1.4.2 Hands-on Simulated Labs 15

2.1.5 Hands-on Remote Labs 16

2.1.6 Service-Learning 16

2.1.7 Case Studies Method 17

2.2 Definition of Software Processes for Academic Scenarios 17

2.2.1 Disciplined Software Processes 18

2.2.2 Agile Software Development Processes 23

2.3 Matching Instructional Approaches And Software Processes 26

2.3.1 Instructional Approaches Considered in the Systematic Study 26

2.3.2 Software Processes Considered in the Systematic Study 27

2.3.3 Mapping Instructional Approaches and Software Processes 27

2.4 Instructional Practices for Software Engineering Courses 29

3 Identifying the SE Best Practices to Be Taught in Undergraduate Programs 33
3.1 ACM-IEEE SE Curriculum Guidelines 33
3.2 SE Best Practices 35
3.2.1 SE Best Practices Being Taught 36
3.2.2 Methodology 36
3.2.3 Results 38
3.3 Software Engineering Education in Chile 43
3.3.1 SE Course Characterization 44
3.3.2 Computer Science Courses 45
3.3.3 SE Knowledge Areas Coverage 45
3.3.4 Courses Evaluations 47
3.3.5 SE Course Temporality 50
3.3.6 Analysis of SE Teaching in Chile 50
3.4 Practices for SE Education 52

4 EduProcess 55
4.1 General Approach 55
4.1.1 Target Software Product 56
4.1.2 Supported Roles 56
4.2 Structure of EduProcess 57
4.2.1 A Brief Description of the Software Process 59
4.2.1.1 Conception 59
4.2.1.2 Iterations 1 and 2 60
4.2.1.3 Deployment 61
4.2.1.4 Work Dynamics 62
4.2.2 The Instructional Process 62
4.2.2.1 Preparedness 63
4.2.2.2 Conception 64
4.2.2.3 Iteration 1 and 2 65
4.2.2.4 Deployment 66
4.2.2.5 Reflexive Weekly Monitoring 66
4.2.2.6 Weekly Work Meetings with Clients and Users 67
4.3 SE Best Practices Included in EduProcess 68
4.4 Process Restrictions 68
4.4.1 Type of Projects to be Addressed 69
4.4.2 Product Type, Size and Complexity 70
4.4.3 Dealing with the Requirements for Academic Software Processes 70

5 EduProcess Evaluation 72
5.1 Case Study Scenario 73
5.2 Data Gathering and Analysis 74
5.2.1 Quantitative Data 74
5.2.1.1 Course Grades 74
5.2.1.2 Project Grades 75
5.2.1.3 Student Survey 75

5.2.1.4 Peer-Assessment . . . 75
5.2.1.5 Amount of Software Built . . . 75
5.2.1.6 Suitability of the Project Implementation . . . 76
5.2.2 Qualitative Data . . . 76
5.2.2.1 Peer-Assessments . . . 76
5.2.2.2 Monitoring Feedback . . . 76
5.3 Results . . . 77
5.3.1 Course and Project Grades . . . 77
5.3.2 End of Semester Student Surveys . . . 80
5.3.3 Peer-Assessments . . . 82
5.3.3.1 Coordination Level . . . 82
5.3.3.2 Sense of Team Belonging . . . 84
5.3.3.3 Team Members Engagement . . . 84
5.3.3.4 Initiative . . . 85
5.3.3.5 Communication . . . 85
5.3.3.6 Commitment . . . 85
5.3.4 Evaluation of Team Productivity and Effectiveness . . . 87
5.4 Discussion . . . 90
5.5 Threats to Validity . . . 93
5.5.1 Construct Validity . . . 93
5.5.2 Reliability . . . 94
5.5.3 Internal Validity . . . 94
5.5.4 External Validity . . . 94

6 Conclusions and Future Work **95**
6.1 Summary of the Thesis Work . . . 95
6.2 Conclusions . . . 96
6.3 Contributions . . . 97
6.4 Future Work . . . 97

Bibliography **98**

Appendix A Software Process Definition **113**
A.1 Project Contexts . . . 113
A.1.1 Type of Projects . . . 113
A.1.2 Product Type, Size and Complexity . . . 114
A.1.3 Our Context . . . 115
A.2 Roles . . . 117
A.2.1 Project Manager . . . 117
A.2.2 Analyst . . . 118
A.2.3 Designer/Developer . . . 118
A.2.4 Quality Assurance Engineer . . . 118
A.3 Introduction to the Phases, Activities and Milestones . . . 119
A.3.1 Conception . . . 120
A.3.2 Iteration 1 . . . 120
A.3.3 Iteration 2 . . . 120
A.3.4 Deployment . . . 120

A.4 Conception Phase . . . 121
A.4.1 Introduction . . . 121
A.4.2 Inputs . . . 121
A.4.3 Activities . . . 121
A.4.3.1 Problem Exploration . . . 121
A.4.3.2 Main User Requirements Identification . . . 123
A.4.4 Reviews (Milestones) . . . 123
A.4.4.1 Problem Presentation . . . 123
A.4.4.2 Prototype Presentation . . . 124
A.4.5 Outputs . . . 124
A.4.6 Deliverables . . . 124
A.5 Iteration 1 and 2 . . . 124
A.5.1 Introduction . . . 124
A.5.2 Inputs . . . 125
A.5.3 Activities . . . 125
A.5.3.1 Project Management . . . 127
A.5.3.2 Prototype Evolution . . . 127
A.5.3.3 User Requirements Definition . . . 128
A.5.3.4 Software Requirements Definition . . . 131
A.5.3.5 Product Design . . . 134
A.5.3.6 Product Implementation . . . 135
A.5.3.7 Test Case Specification . . . 136
A.5.3.8 System Verification . . . 138
A.5.3.9 User Validation . . . 138
A.5.3.10 Deployment . . . 138
A.5.4 Reviews . . . 139
A.5.4.1 Requirements Evaluation . . . 139
A.5.4.2 Design Evaluation . . . 140
A.5.4.3 Demo Presentation . . . 140
A.5.5 Outputs . . . 140
A.5.5.1 Project Plan . . . 140
A.5.5.2 User Requirements . . . 140
A.5.5.3 Software Requirements . . . 140
A.5.5.4 Test Cases . . . 141
A.5.5.5 Requirements Review Notes . . . 141
A.5.5.6 Design Review Notes . . . 141
A.5.5.7 Demo Review Notes . . . 141
A.5.6 Deliverables . . . 141
A.5.6.1 Software Requirement Document . . . 141
A.5.6.2 Design Document . . . 143
A.5.6.3 Software Product - SPr . . . 143
A.6 Software Deployment . . . 143
A.6.1 Introduction . . . 143
A.6.2 Inputs . . . 144
A.6.3 Activities . . . 144
A.6.3.1 Tuning of the Software Product . . . 144
A.6.3.2 Product Deployment . . . 144

A.6.3.3 Client/User Assisted Usage . . . 145
A.6.3.4 Incidence Registration . . . 145
A.6.4 Reviews . . . 145
A.6.5 Outputs and Deliverables . . . 145

Appendix B Instructor Software Process Definition 146
B.1 Phases, Activities and Milestones . . . 146
B.2 Preparedness . . . 146
B.3 Conception . . . 149
B.3.1 Inputs . . . 149
B.3.2 Activities . . . 149
B.3.2.1 Week 1 . . . 149
B.3.2.2 Remaining Weeks . . . 149
B.3.3 Reviews . . . 151
B.3.3.1 Problem Presentation . . . 151
B.3.3.2 Prototype Presentation . . . 151
B.3.3.3 Peer Evaluation 1 . . . 151
B.3.4 Outputs . . . 152
B.3.5 Deliverables . . . 152
B.4 Iteration 1 . . . 152
B.4.1 Inputs . . . 152
B.4.2 Activities . . . 152
B.4.3 Reviews . . . 154
B.4.3.1 Requirements Evaluation . . . 154
B.4.3.2 Design Evaluation . . . 154
B.4.3.3 Demo Presentation . . . 155
B.4.3.4 Peer Evaluation 2 . . . 155
B.4.4 Outputs . . . 155
B.4.5 Deliverables . . . 155
B.5 Iteration 2 . . . 155
B.5.1 Inputs . . . 155
B.5.2 Activities . . . 155
B.5.3 Reviews . . . 157
B.5.3.1 Requirements Evaluation . . . 157
B.5.3.2 Design Evaluation . . . 157
B.5.3.3 Demo Presentation . . . 157
B.5.3.4 Peer Evaluation 3 . . . 157
B.5.4 Outputs . . . 157
B.5.5 Deliverables . . . 157
B.6 Deployment . . . 158
B.6.1 Inputs . . . 158
B.6.2 Activities . . . 158
B.6.3 Reviews . . . 158
B.6.4 Outputs . . . 158
B.6.5 Deliverables . . . 158

Appendix C Reflexive Weekly Monitoring 159

C.1 RWM - Structure 160
C.2 Monitoring Schedule 162
C.2.1 Conception 162
C.2.2 Iteration 1 164
C.2.3 Iteration 2 165

Appendix D SSP versus EduProcess 167
D.1 SSP 167
D.1.1 Phases 167
D.1.2 Roles and Context 168
D.2 Comparison 168

Appendix E A Systematic Mapping Study on Practical Approaches to Teaching Software Engineering - FIE 2014 171

List of Tables

1.1 Comparison of the Industrial and Academic Development Scenarios 7

2.1 Disciplined Software Processes for the Academia 19
2.2 Comparison of Disciplined Processes with the Set of "Minimal" Requirements for Academic Process 21
2.3 Agile Software Processes for the Academia 23
2.4 Comparison of Agile Processes with the Set of "Minimal" Requirements for Academic Process 24

3.1 Specific Areas Considered in the 2004 - ACM-IEEE SE Curriculum Guidelines 34
3.2 Specific Areas Considered in the 2014 - ACM-IEEE SE Curriculum Guidelines 35
3.3 Search Strings 37
3.4 Specific Search Strings for Each Digital Library 37
3.5 Number of the Papers Retrieved from Each Digital Library 38
3.6 Results After the Use of the Guidelines 39
3.7 Primary Studies Selected 39
3.8 Reported Best Practices (to be applied by students) 40
3.9 Reported Best Practices (instructional) 40
3.10 Best Practices Reported in More than One PS 41
3.11 How was Validated the Use of the Best Practices (TiP - Taught in Practice) . 41
3.12 ACM-IEEE SE2014 Curriculum x Reported "Best Practices" 42
3.13 Number of Computer-Related Programs Offered by Chilean Universities in 2015 44
3.14 Number of Software Engineering Courses in Each Program in 2015 46
3.15 Most Frequent Knowledge Units in CS Engineering Programs 47
3.16 Least Frequent Knowledge Units in CS Engineering Programs 47
3.17 Most Frequent Knowledge Units in CS Technology Programs 48
3.18 Least Frequent Knowledge Units in CS Technology Programs 48
3.19 Mandatory Practices 52
3.20 Recommended Practices 53
3.21 Optional Practices 53

4.1 SE Best Practices Included in the EduProcess 69

5.1 Semesters and Students Enrolled in the CC5401 Course 78
5.2 Statistical Analysis of Course and Project Grades 78
5.3 Items of the Student Survey 80
5.4 Student Survey Results 81

5.5 Statistical Analysis of the Peer-Assessments . . . 82
5.6 Team Aspects - Coordination, Sense of Team Belonging, Engagement . . . 83
5.7 Team Aspects - Initiative, Communication, Commitment . . . 86
5.8 Team Productivity . . . 88
5.9 Team Members Productivity and Effectiveness . . . 89
5.10 Statistical Analysis of FP . . . 90
5.11 Team Productivity Detail . . . 91
5.12 Project Deployment (meaning: ✗ - not deployed, ✓- deployed) . . . 92

A.1 Context Restrictions . . . 116
A.2 Software Life Cycle Model . . . 119
A.3 User Requirements Specification Detail . . . 129
A.4 Software Requirements Specification Detail . . . 133
A.5 Test Case Detail . . . 137
A.6 Review Notes Detail . . . 141

B.1 Software Life Cycle Model . . . 147

D.1 Comparison of SSP and EduProcess with the Set of “Minimal” Requirements for an Academic Software Process . . . 169

List of Figures

1.1 General Scenario of Software Development Experiences in SE Courses 3
1.2 Summary of the Research Methodology 8

2.1 Systematic Mapping Process by Petersen et al. [126] 26
2.2 SE Instructional Approaches Reported by Year 28
2.3 Instructional Approaches Reported 30

3.1 Course Characterization in (a) CS Engineering Programs and (b) CS Technology Programs 46
3.2 Knowledge Areas Characterization of CS Engineering Programs 48
3.3 Knowledge Areas Characterization of CS Technology Programs 49
3.4 Courses Evaluation of CS Engineering Programs 49
3.5 Courses Evaluation of CS Technology Programs 49
3.6 Course Temporality in CS Engineering Programs 50
3.7 Course Temporality in CS Technology Programs 51

4.1 General Architecture of EduProcess 55
4.2 EduProcess Tracks 56
4.3 Architecture Assumed for an Information System 56
4.4 Structure of EduProcess 58
4.5 Structure of the Software Process 59
4.6 Structure of the Software Prototype 60
4.7 Relationship Among the Work Products of Iterations 1 and 2. 60
4.8 Structure of the Instructional Process 63
4.9 General Workflow of Iterations 1 and 2 for the Instructional Process 66
4.10 General Structure of the RWM Sessions 67
4.11 Process of Co-work between the Software Team and the Stakeholders 68

5.1 Timeline of the Evaluation Process 73
5.2 Course Grades using (a) SSP and (b) EduProcess 79
5.3 Project Grades using (a) SSP and (b) EduProcess 79

A.1 General Structure of EduProcess 120
A.2 Conception Phase 121
A.3 Problem Exploration 122
A.4 Main User Requirements 123
A.5 Phase - Iteration 1 and 2 126

A.6 Project Management Activity . . . 127
A.7 Prototype Evolution Activity . . . 128
A.8 User Requirement Activity . . . 130
A.9 Software Requirement Activity . . . 132
A.10 Design Activity . . . 135
A.11 Implementation Activity . . . 136
A.12 Test Case Activity . . . 137
A.13 System Verification Activity . . . 137
A.14 User Validation Activity . . . 138
A.15 Optional Deployment Iteration 1 . . . 139
A.16 Deployment . . . 144

B.1 EduProcess-Instructional . . . 146
B.2 EduProcess-Instructional- Preparedness Phase . . . 148
B.3 EduProcess-Instructional - Conception Phase . . . 150
B.4 EduProcess-Instructional- Iteration 1 . . . 153
B.5 EduProcess-Instructional- Iteration 2 . . . 156
B.6 EduProcess-Instructional- Deployment . . . 158

C.1 Structure of a RWM Session. . . . 161

D.1 Structure of SSP . . . 167

Chapter 1

Introduction

In 1994, Moore and Potts [107] stated that "software engineering cannot be taught exclusively in the classroom. Because software engineering (SE) is a competence, and not just a body of knowledge, any presentation of principles and experience that is not backed up by active and regular participation by the students in real projects is sure to miss the essence of what the student needs to learn". Although this work was published more than twenty years ago, this statement is still valid today. In order to address this issue, the ACM and IEEE-CS Society formed the Software Engineering Education Project that ended in 2003 with the publication of the Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering (SE2004) [120], and in 2014 it was updated to a newer version (SE2014) [119]. These guidelines reflect the need for practical experiences in teaching software engineering. Another effort was made by a group of researchers that proposed the Graduate Software Engineering Reference curriculum (GSwE2009) [130] which was also recommended by ACM and IEEE-CS Society. This reference curriculum also shows the need for conducting practical experiences in software engineering courses.

After considering several sources, there seems to be a consensus that contemporary software engineering education should be addressed using a practice-based approach [26, 29, 139]. In a recent study, Lutz et al. [87] compared the undergraduate software engineering program of the Rochester Institute of Technology (RIT) with undergraduate programs based on the Computer Science Curricula 2013 (CSC-2013) [121]. The results indicated that software engineers from RIT were well prepared to start their work in the software market, since RIT's program curriculum emphasizes the use of "learning-by-doing" strategies and teamwork in all courses. On the other hand, in programs based on the CSC-2013, the students usually participated in only one software project during their undergraduate studies, in which they could apply their theoretical and technical knowledge and also enhance their soft skills (e.g., decision-making, communication and teamwork). Software companies that hired RIT graduates believed that the use of this practice-based approach would allow these RIT graduates to match up favorably with companies software engineers who already have five years of experience in industry [87].

We recently conducted a mapping study on any software engineering teaching approach being used and reported. It indicates that 54% of the scientific reports (from a total of 173 selected papers) use learning-by-doing strategies in software engineering courses [102]. This is

quite a low rate, considering the relevance that the use of practical approaches has according to the scientific community of the area. This mapping study also indicates that one of the reasons for using this teaching-learning approach is its flexibility, i.e., it can be adjusted to the features and interests of instructors and students.

According to Hayes et al. [61], the software projects run by students during undergraduate programs should be used to staple the knowledge acquired in previous courses. These researchers also indicate that there is a gap between what students know, what books say about software engineering, and what students do in practice, because students still do not have practice executing software projects. Student experience with software engineering practices, as well as team work is quite different from what the industry expects of them.

A good SE project course should ideally bring together not only basic engineering principles, but also interpersonal communication capabilities and knowledge acquired in complimentary areas, e.g., the design of user interfaces [27, 38]. This integral perspective to address software engineering education gives students a space to put their technical knowledge into practice, and also develop soft skills required for their work in industry. Unfortunately, very few computer science and software engineering programs offer practical courses that allow the students to live real software development experiences, and thus understand the relevance to count on both technical knowledge and soft skills before getting into the software industry [11, 41]. The acquisition of software development capabilities would not only be possible, but also motivating, if students experience professional work while they are enrolled in undergraduate programs. It is important that these practical experiences guide students to successful software projects in which they develop a software that is effectively used by client organizations. These successful outcomes would allow them to realize how to properly carry a software process in order to produce positive results in future projects [87].

The methods used to help generate these positive experiences could be captured through software processes designed to be used in academic scenarios, and more specifically, processes tailored for particular academic settings. These processes should consider the participation of students, clients, users, instructors and teaching assistants. These process designs must not only help produce successful experiences for the students, but also make these experiences feasible and repeatable for educational institutions. Thus, software engineering education can favorably influence the use of a practical approach in a systematic way.

1.1 Teaching Software Engineering with a Practical Approach

Project-based learning [92] is the most common instructional approach used in SE courses that implement software development experiences that try to simulate the industry scenario. These experiences should include two processes that run in parallel during the project duration and are intertwined, i.e., a milestone in one of them typically matches one or more milestones in the other process (see Figure 1.1).

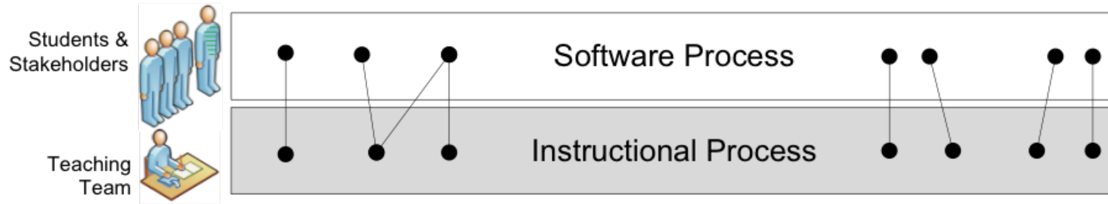


Figure 1.1: General Scenario of Software Development Experiences in SE Courses

The software process is executed mainly by students and stakeholders (e.g., clients and users) in order to build a software product, and the instructional process is guided by the teaching team and followed by the students to make these experiences more fruitful for them as future software engineers. Typically, there is a software process guiding the projects conditions, which states the instructional approach utilized in these experiences. The type of software process usually ranges from disciplined to agile processes [134]. While the former are structured and based on phases and roles, i.e., process-centric, the latter are unstructured, focused on the product and based on team self-organization, i.e., product-centric. Both strategies have shown positive results in SE project courses [43, 148, 155]. However, they have different purposes and requirements, and their suitability depends on the type of course where are used.

1.1.1 Types of SE Project Courses

The literature of the area distinguishes between two types of courses in which project-based course experiences are usually conducted: *project-based courses* and *capstone courses*. A capstone course is the culminating course of a program and usually integrates almost everything a student learns in an educational program. In both types of courses the students work in teams during one or two semesters, executing a project in order to obtain a software product. However, the goals, requirements and dynamics of the practical experiences followed in each type of course is different. The definition of practical experience used in this thesis is knowledge or skills that students get from participating in a software project rather than just reading about it or seeing it being done by other people.

On the one hand, project-based courses are used to help students internalize the theoretical knowledge acquired in previous software engineering courses [153]. These project-based courses, assume inexperienced developers performing loosely-coupled work, since this participation strategy gives them the flexibility to contribute to the project while they are still taking other courses. This working style allows students to perform asynchronous and distributed work, conducting individual activities most of the time, followed by sporadic coordination tasks. According to Robillard and Dulipovici [135], a *disciplined process* is usually appropriate for supporting project-based courses given the particularities of the instructional scenario and the goal assigned to this type of course. This type of process is structured and based on phases and roles, i.e., process-centric.

On the other hand, capstone courses are usually the last software engineering course and sometimes even the last course of the program, where students are no longer novice developers; they already have at least one experience developing software in teams. Moreover,

in capstone courses, students typically perform tightly-coupled work, since they have to work together simulating a regular industrial scenario [47, 113, 140, 166, 171, 177]. *Agile processes* seem to be more appropriate for supporting capstone courses [91], where students have to gain a broader perspective of the software development practice and reinforce their technical and nontechnical skills before becoming a part of the software industry. Moreover, it is recommended that these students have prior experience in developing software, which fits well with those enrolled in capstone courses [135].

1.1.2 Challenges for Implementing Practical Experiences in SE Courses

Both project-based and capstone courses involve several challenges for instructors and institutions [170] that jeopardize the feasibility of teaching these courses in a practical manner. Next, we explain the most relevant challenges, mainly for instructors, reported in the literature.

Complexity of designing the practical experiences. The process of designing practical experiences is complex, particularly when a disciplined software process is used to guide the development [122]. Typically, instructors know the SE domain, but they do not master how to address the instructional aspect required to implement these experiences, which is intertwined with the software process being used. Although the complexity of designing these experiences is high when agile processes are used, the situation is even worse when the process is disciplined, as it usually involves several phases, roles, work products and activities that should be considered and most of them evaluated in the instructional process (the teaching/learning activity can be considered as a process, called the instructional process [42][123]).

Implementation effort. These experiences require an important effort by the involved people (i.e., instructors, teaching assistants, clients, users and students), which is not always recognized by the educational institutions [22]. Particularly, project-based courses are more demanding than capstone courses, since the software process used in these experiences is more structured, there is a wider variety of deliverables, and the developers are novices. Moreover, these experiences are particularly time consuming, which contrasts with the low availability that the involved people usually have to play their role properly.

Capability of supervision. SE instructors usually know the theory very well, but they do not always have much industrial experience to pass on to students [16, 83]. This also limits the feasibility of implementing real software development projects and guiding the students properly during project execution. Broman [16] identifies some additional challenges for the instructor and teaching assistants such as, getting enough resources for supervising students, and having the ability to properly assess and grade their achievements in software projects. A proper evaluation of projects and their products requires that evaluators know not only technical aspects of software development, but also the business area that will be supported by the product being developed [83]. The supervising capability is more important in project-based courses because the developers have little experience.

Student and client commitment. Particularly, in project-based capstone courses, students schedule short time periods to work in a distributed way on their projects (they perform many context switches) [79]. This loosely-coupled work style reduces the sense of team

belonging in students and consequently their commitment to the project. Several researchers report procrastination and the "free-rider" syndrome in project-based courses, due to students assuming an apprentice attitude (i.e. speculation with their work when they feel that the project is under control [144]) when addressing their projects [39, 15, 101, 170]. Moreover, people acting as clients and users frequently have low commitment with these projects (or with the course) [170].

Although these challenges affect both types of courses, they are more severe in project-based courses (particularly when disciplined processes are used) since the instructional scenario is usually more complex than in capstone courses. Moreover, the literature reports several mechanisms to address agile developments in capstone courses, but is almost inexistent when using disciplined development approaches; i.e., used in project-based courses.

1.2 Problem Statement

As mentioned before, the instructional process followed by the students and the instructional team in these experiences is intertwined with the software process that guides the students' development projects. In capstone courses, most reported experiences use agile approaches [17, 19, 33, 34, 156], and particularly simple adaptations of what is currently used in the industry; e.g., Scrum or XP. The companion instructional processes for agile methods are quite simple and involve the use of e-portfolios [21, 89] (Moodle like platforms) and coaching [25, 138] since the software processes are also simple. This allows the instructors to follow this recipe (agile methodologies) in a easy way which is probably the reason why it is so popular in capstone courses. However, that recipe is not suitable for addressing project-based courses due to the reasons indicated in Section 1.1.1; students taking these courses are usually inexperienced developers and can work only in a loosely-coupled way given that they are still taking other courses.

The literature reports some disciplined processes for supporting project-based courses [102], however, none of them are described in enough detail to allow instructors to follow such processes. Therefore, they cannot be considered as potential prescriptive processes able to support these experiences. According to Pressman [128], these processes can be considered prescriptive if they define the set of activities, roles, milestones and work products that are required to engineer high-quality software.

Given the lack of prescriptive proposals for helping the instructor and students address these practical experiences in project-based courses, particularly when they involve disciplined software processes, we can consider using some of the disciplined processes used regularly in the software industry. However, the work context and objectives followed in project-based courses are different from those that are present in industrial settings [136, 124, 171]. Therefore, disciplined software processes defined for industry are not necessarily useful either for supporting developments in project-based courses.

Table 1.1 shows a comparison between both development scenarios highlighting their differences. The results of this comparison indicate that disciplined processes from the industry cannot be directly applied to academic scenarios. Moreover, there is less at stake in the development conducted in the academic environment. According to Rajapakse [132], "In

school, only the course grade is at stake. If the teamwork is not going well, the student has many venues to compensate for the grade, such as scoring more in individual components of the grade, e.g., exams, complaining to the instructor with the hope of obtaining sympathy marks, or doing extra work to make up for the shortcomings of a team member".

The state-of-the art indicates that this is still an open problem that needs to be explored in order to find alternatives to address it. Consequently, the instructors in charge of project-based courses need a disciplined software process, and the corresponding instructional approach that can be followed by the people involved in these courses, in order to make these experiences more fruitful, particularly for the future software engineers.

1.3 Work Hypotheses

To support these practical experiences in project-based undergraduate courses where students work in a loosely-coupled style, this thesis hypothesizes that a prescriptive software process that uses a disciplined approach can:

- (H1) make these experiences repeatable investing an affordable effort.
- (H2) help produce positive results in software projects.

1.4 Objectives

The main objective of this thesis is to define a prescriptive software process, named EduProcess, which helps people involved in project-based undergraduate courses (i.e., instructors, teaching assistants, students, clients and users) to participate in a software project, in a systematic and affordable way. EduProcess must embed software engineering (SE) best practices and help produce positive project results in terms of grades, project quality, requirements done and software deployment. Specific objectives derived from the general goal are the following:

- Identify software processes and instructional approaches that can be used together to support software development experiences in SE project-based courses. Reaching this goal allows us to determine the most promising combinations of software processes and instructional approaches to be considered during the design of EduProcess.
- Identify a core set of SE best practices that should be taught to students in SE undergraduate courses. This requires to consider the international recommendations and also the current state of the SE education at the universities; in our case, we considered the reality of the Chilean universities. Reaching this goal allows us to determine a set of candidate SE practices to be embedded in EduProcess.
- Define the architecture of a disciplined software process that considers the results of the previous goals, and allows to obtain positive results in a repeatable way involving an affordable effort. Reaching this goal allows us to validate the stated hypotheses, and depending on the obtained results, to reach the main objective of the thesis.

Table 1.1: Comparison of the Industrial and Academic Development Scenarios

Industry Scenario	**Academic Scenario**
High degree of team member commitment to projects (one or more) [23, 118].	Lower degree of team member commitment to the project (they have other courses and other priorities) [23].
Formal job (or contract); people know that they will be held responsible [23].	There is no formal way of pressing students to do their jobs (bad grades or even failing is usually not a concern for students) [23].
Team members have experience using a software process, developing software and working in teams [109, 170].	Students have little or no experience using software processes and working in software development teams [23, 109, 170].
Team leaders are experienced project managers [109, 170].	Team leaders are students (peers) with no expertise with managing software projects [109, 170].
Student attitudes are not usual (e.g., free rider, last minute dedication, etc.) [170].	Student attitudes are highly frequent [170].
There is no formal role for someone that monitors the project from a neutral position (like the course instructor).	There should be formal roles for clients, users, instructors and teaching assistants.
The clients are normally committed to the project, because they are paying for a product [23].	The commitment of the clients to these projects is usually low, because they do not pay for the product. Moreover, they do not usually expect too much from student projects [23].
Developers use almost continuous work periods for developing the software [60].	Developers use highly fragmented work periods, with many context switches [60].
The software processes are product-oriented [129].	Academic software processes should be more educational than just product-oriented [129, 22].
The software processes are designed to develop projects that are diverse in terms of size and complexity.	Typically academic software processes need to guide the development of small and simple products [170].

1.5 Research Methodology

The research methodology followed in this thesis considers three stages (see Figure 1.2): information gathering and analysis, definition of the EduProcess, and evaluation of the EduProcess. Next we briefly explain each of them.

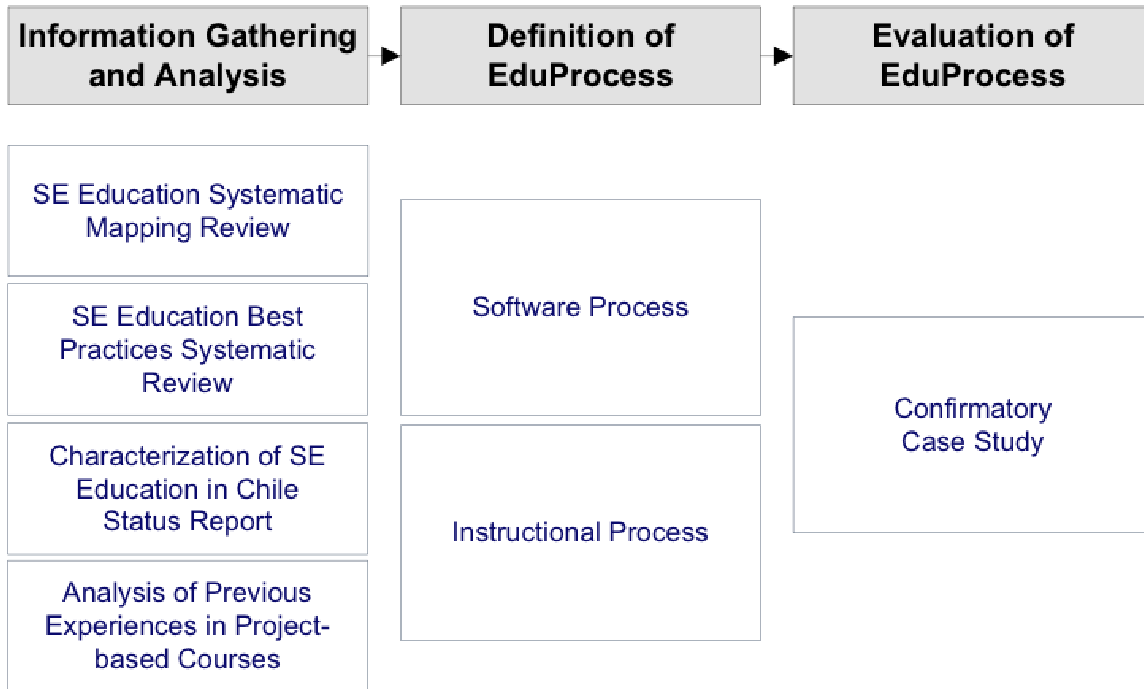


Figure 1.2: Summary of the Research Methodology

1.5.1 Information Gathering

An extensive bibliographic review was performed to characterize how software development is taught in academia [100]. The main goals were to find out software development experiences in academic scenarios and lessons learned that could be used to inform the design of the software process proposed in this thesis. A Systematic Mapping Review was then conducted, following the model proposed by Kitchenham et al. [74], to try and determine the software processes and teaching approaches used to support these experiences [102].

Using these previous studies, we identified (evaluating their relevance on research) the context variables that can be used to characterize these experiences; for instance, the project type and size, the level of participation of the actors, and the product quality and complexity. Having a general characterization of the software development scenario allowed us to specify the work context for which a disciplined software process is proposed. Based on this characterization we defined our target context. Thus, instructors that want to reuse these processes can determine which proposal fits the work context that they have in their own courses. The specification of a software process work context also helps other researchers and instructors to understand the rationale behind the proposal; therefore, they can then extend it or reuse part of it.

Another extensive literature review (systematic literature review) was conducted to evaluate the best practices being taught to students, since the literature reports a large amount of software engineering best practices and work capabilities (skills) that should be learned by the students. Addressing all of them is neither feasible nor appropriate for a single software course, so only some of them were chosen. Chapter 3 presents these two works.

Provided that in our general characterization of software development in academia we did not find any work reporting the state of SE education in Chile, we performed an extensive (83% of the accredited universities in computer related programs) exploratory study to identify the status of the software engineering teaching in Chilean Universities [99]. This work allowed us to identify and understand the strengths and weaknesses of the local scenario and compare it with those reported by the literature in the international context.

Finally, a set of candidate disciplined processes designed for academic scenarios were analyzed to identify reusable knowledge that can be used to design EduProcess. After this analysis we decided to use the SSP (Simple Software Process) [115] as a basis for generating our proposal because it has been quite successful in supporting software developments in the academic scenario. The main weakness of SSP are: it is not a formalized process, is not available to other instructor to follow, does not offer student or instructor support.

1.5.2 Definition of the EduProcess

Using SSP as a basis along with the knowledge collected in the previous studies, we defined EduProcess, that has evolved based on the results of small empirical pilot studies. The proposal includes a software and an instructional process that are both intertwined to help students take advantage of these experiences.

EduProcess includes activities, roles, presentations and deliverables, and the process addresses both the technical and instructional aspects of student experiences. Technical aspects refer to development activities and the way they are evaluated. Instructional aspects refer to the activities intended to assess student work, provide feedback, and make these experiences repeatable for instructors.

The process was formalized using the Eclipse Process Framework Composer (EPF Composer) [48], which is a free open software that allows the publication of a process in an easy and manageable way. The prescriptive process (students and instructors) that we show in this thesis is available online at (http://www.dcc.uchile.cl/~mmarques/), is presented in detail in Appendix A and the Instructor process is presented in Appendix B.

1.5.3 Evaluation of the EduProcess

EduProcess was evaluated in an undergraduate project-based course (CC5401: Software Engineering II) taught by the Computer Science Department of the University of Chile. The details of the context of the course are presented in Chapter 4.

The evaluation of EduProcess involved a case study with multiple sources of qualitative and quantitative data related to student grades, project grades, student surveys, peer evaluations

and the amount of software built by the students. The evaluation considers the results of eight semesters where the SSP process was used in the same course, as well as the results of the last four semesters of the course where EduProcess was used.

The results show that EduProcess can help produce positive results in software projects and a software that solves clients problem. These results are directly related to course and project grades. Moreover, the use of EduProcess has improved the teams communication, initiative, engagement, commitment, sense of team belonging, coordination and responsibility.

In addition, the results show no difference in the amount of software built by the teams that used EduProcess and the teams that did not use it. However, we saw that teams using EduProcess addressed a higher number of critical software requirements, which are the most important requirements for the client.

The results of EduProcess presented in our Case Study show that our two hypotheses are aligned with our results.

1.6 Summary of the Thesis Document

This thesis work is divided into seven chapters. Chapter 2 presents and discusses the state-of-the-art in the project-based software engineering education. Chapter 3 presents the results of the systematic research on how software engineering is taught in Chile and in the rest of the world.

Chapter 4 defines EduProcess along with all its related process elements: phases, roles, deliverables, workflows, work products, and assessment activities. Chapter 5 presents our case study, the scenario where the proposed process was used, and where it was used. We also present the results of our case study, a discussion of our results and threats to validity. Finally, we conclude in Chapter 6 with a summary of this thesis, our conclusion and an outline of future research directions.

Chapter 2

Related Work

Learning is the process whereby knowledge is created through the transformation of experience. Approaches providing this concrete experience can differ quite a bit, ranging from the traditional "chalk-and-talk" paradigm, where a lecturer presents information to a group of students, to the "learning-by-doing" technique, where students themselves seek out the knowledge they need to do something. In undergraduate software engineering education, learning-by-doing has always been important. Individual capstone projects during the final year have been universally accepted, while group-based projects are recently more common in many universities [56].

Software engineering (SE) is a knowledge area oriented towards the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software [164]. As any engineering field of study, it is tightly linked to industry; therefore, the software engineering curricula used in computer science programs needs to prepare students for the world of industrial software development [24].

Fox and Patterson [49] pointed out that while new university graduates are good at coding and debugging, employers complain about other missing skills that they think are equally important (e.g., working with non-technical customers, teamwork, etc.). The usual faculty reaction to such criticism is that they are trying to teach principles and fundamentals.

It is well-known that the SE industry needs people capable of working in teams to develop and maintain software. However, training young professionals that are proficient in technical and operative issues is a great challenge for educational institutions [41, 57, 63, 174]. Begel and Simon [11] pointed out that universities are striving to prepare students for the industrial scenario, and allowing them to become life-long learners able to keep up the pace with SE advances. Typically, universities tend to adequately teach the technical aspects of SE, but they do not fully address operative issues such as managing projects/risks, defining/adjusting software processes, and performing teamwork.

In the next section we discuss how software engineering is being taught, as reported in the literature. Section 2.2 presents the requirements that should be considered for defining a software process that will be used in academic scenarios. It also describes a set of disciplined

and agile processes, and analyses how well they fit the previously defined requirements. Section 2.3 presents a systematic literature review that shows the instructional approaches used, and also how they are combined with the existing software processes. Finally, Section 2.4 presents the instructional processes reported in the literature.

2.1 Teaching Software Engineering

Parnas [122] defined three steps required to design software engineering courses: (1) to define the possible tasks that a software engineer is expected to perform, (2) to define a body of knowledge required for software engineers, and (3) to transfer such knowledge through a training program. This definition clearly identifies the need for transferring, in a practical way, the key knowledge and skills required by students who will be future software engineers.

According to Lethbridge et al. [83], practical experiences linked to the industry reality can help students put in practice the SE knowledge gathered during the courses. However, these experiences require that the instructors have the necessary background to guide them, which could represent a challenge.

Recognizing the relevance of the practical experiences, we also mention that SE theoretical knowledge must be an integral component of students' engineering education. Computer science programs in general should therefore include these types of SE experiences. One should present the foundations of the area and the key knowledge in a theoretical (or almost theoretical) way. A second experience should mix theory and practice (usually known as project-based course), where the students apply to a software project most of the knowledge acquired in the first SE course. Finally, a third experience would be a capstone experience that allows students to explore and take on several aspects of professional work, through the execution of a real project in an industrial setting or in an environment that simulates such a work context [135].

Concerning the knowledge included in these courses, the IEEE periodically develops new versions of the Software Engineering Body of Knowledge (SWEBOK) [1], which helps instructors and institutions identify key knowledge to transfer to students. However, this recommendation does not provide guidelines on how to transfer this knowledge.

There are a several instructional approaches to address the problem of teaching SE. We now discuss these initiatives, their usage and results. The major teaching categories found in our mapping review [102] were the following: expositive (traditional) methods, inverted classroom, problem-based learning methods, hands-on methods (learning-by-doing), hands-on labs, hands-on simulated labs, hands-on remote labs, service learning and case studies.

2.1.1 Expositive Methods

The expositive (or traditional) approaches used in SE education involve a lecturer standing in front of the students, with the course content divided into a number of topical lectures [111, 133]. The instructor generally tries to encourage students to discuss the material delivered in the lectures. Sometimes the lecturer assigns relevant readings, normally textbooks

or research papers, in order to enrich these discussions and the material comprehension.

The evaluation is based on students' understanding of the topic and/or their capability to apply the concepts in a mock software project [133]. This passive teaching strategy (for students) [77] seems to be very distant from the strategies required by future engineers to develop software in industry.

2.1.2 Inverted Classroom

The traditional lecture is placed online for students to listen to during their own time. Their classroom time is used for other activities, such as workshops, interactive work time with the instructor, or demonstrations [103]. The inverted classroom approach has been in use in SE courses for quite some time [51], and is reported as being effective. Mason et al. [103] reported that the inverted classroom allowed the instructor to cover more material; students performed as well or better than others in a comparable traditional course; and that students initially struggled with the new format, but they quickly adapted and found the inverted classroom format to be effective.

2.1.3 Problem-Based Learning Method

The Problem-Based Learning (PBL) method is characterized as a learning approach in which students are given more control on their own learning than in traditional approaches. In PBL the students are asked to work in small groups and acquire new knowledge on a need to know basis [133]. The main idea is that these real situations are used to present challenging operational problems, generating learning outcomes that incorporate the professional knowledge, skills and competencies required by graduates.

Richardson et al. [133] reported a two-year research project with graduate students, where at the beginning of the lectures, a real-world problem was presented to the classroom. The students had to perform roles as members of a development team and develop a solution to the stated problem. These researchers also reported the use of PBL as a positive experience for both instructors and students. It provided students with the experience of solving a real-world problem and the opportunity to understand the relevance of the software quality and its potential effects.

Santos et al. [142] reported a case study on using PBL in an SE course with emphasis on usability; they concluded that the PBL approach can be quite useful in SE courses. García and Pacheco [52] concluded that the PBL approach was able to increase the experience of the students along the projects in a gradual form, focusing on communication, participation and time management skills.

2.1.4 Hands-on Methods (Learn-by-Doing)

Ma and Nickerson [88] pointed out that "hands-on experience is at the heart of science learning". Hands-on activities (also known as "learning-by-doing") involve a physically real investigation process. They also advocate that hands-on labs provide the students with real

data and unexpected clashes between the disparity of theory and practical experiments, which are essential for student learning. It can be a challenging and realistic opportunity for students to apply the concepts of software engineering practice and process that they learn as part of their studies [147]. It is also possible to divide this hands-on experience among: labs, simulated labs (imitations of real experiments where the infrastructure required is not real, but simulated by computers) and remote labs (characterized by mediated reality).

Most experiences on hand-on methods focus on training the students in software development. However, there are also some experiences focused on engineering software products – particularly those using a pre-established software process [14, 59].

2.1.4.1 Hands-on Labs

Sebern [147] reports the Real Lab experience, which incorporates real clients when teaching practical software engineering. This researcher recommends extensioning legacy systems as the goal of these labs, rather than engineering new solutions from scratch. He explains that the development of projects from scratch constrains the size of the software system being developed and increases the risk of non-functional or incomplete final products. The Team Software Process (TSP) is the basis of the software process used by students in these labs. He reported that the students were capable of linking together theory and practice, and both students and clients participating in the lab obtained useful benefits.

Andrews and Lutfiyya [4] used the idea of software maintenance to teach software engineering. They defined software maintenance as modifications that are made to the software after its initial release. They based their decision on the fact that many organizations are reluctant to completely replace older software. These maintenance tasks were not classic code modification tasks, they were instead concerned with the development of new code to fit into existing software without changing what already works.

Tvedt et al. [165] created a kind of industrial software factory based on the idea of hands-on experience. Here, the course involves a two-semester project (development from scratch as well as maintenance projects) with real clients and where the students play roles as team members. They reported that at the beginning, both students and instructors had difficulties with the course, but after getting things started, the course ran smoothly and initial reports from students seem to show that the course was working as planned.

Cagiltay [20] chose the development of games as an appropriate scenario to teach SE. Games, or game-inspired exercises, have been used in programming courses for some time because they have the potential to motivate learners. Typically, students are end-users of games, therefore they usually know the core business of the product under construction. As users, students are able to compare different user interfaces among games that they have played over the years [46]. This situation improved student self-motivation during the experience, and some students continued working on their project after finishing the course.

Jaccheri and Osterlie [67] used the participation of students in open source software projects as a scenario for teaching software engineering. Students could freely choose in which open source software project they wanted to work. In those experiences, the researchers found

that the documentation of some projects was “not strong enough" to help young students participate, and that the learning curve that students face can be a problem if the course is too short. However, they also reported that students liked the experience and some of them continued supporting open source projects after completing the course.

Similar to the previous experience, Allen et al. [3] also used open source software projects as a teaching/learning scenario. However, in this case, the instructors chose the projects and documented them before students starded their development. They reported that the students productivity was still below their expectations and that the students’ learning curve was steeper than intended.

Germain et al. [55] conducted a hands-on software course where all teams had to develop the same project using the UPEDU process (Unified Process for EDUcation). This process is a customized version of the Rational Unified Process (RUP) for education [136]. After several experiences, these researchers concluded that some disciplines, such as development and requirements, dominated the teams’ effort and that the learning curve of the process meant that development began after the midway point of the course. Therefore, the students obtained very good interim artifacts (e.g., requirement list and component design), but they were not able to fully develop everything they had planned to develop.

Moore and Potts [107] created The Real World Lab, which was designed to emulate an industrial organization as closely as possible. Industry sponsors propose projects who in turn act as “customers”, providing consulting, reviewing, direction and resources to the students. The authors reported that in the initial stages of this experience, they used the Mini-Task model, which is based on a waterfall lifecycle, but after the first iteration, the students decided to adopt an informal (ad-hoc) process where they were able to include all the processes and formalisms that they thought would be an improvement to the product and the process. In a later work, Moore and Brennan [108] used the CMM model to evaluate the ad-hoc process created by the Real World Lab students and they found this process addressed the key process areas required for a CMM level 3 evaluation. In both papers, the authors indicate the importance of hands-on experiences and highlight how these experiences help students become much more confident and motivated to finish their undergraduate studies.

2.1.4.2 Hands-on Simulated Labs

Vaughn [172] reported an experience in which he used a simulated process as the main instrument to teach software engineering. The experience involved real clients for the student projects, and the students had to act as a real company trying to develop real software. However, the students did not have to implement the product, but only carry out all the process activities considered in a real software company (e.g., requirements specification, design, test plan, documentation and project/budget management). The author reported that the main challenge was to find clients willing to participate in the experience, specially knowing that they will not get any functional software at the project end. Students found these activities time consuming and unpleasant, but still found value in the experience.

Drappa and Ludewig [44] also developed a simulation game named SESAM (Software Engineering Simulation by Animated Models) to teach SE principles and practices. Each

student must play the role of the project manager, and control the project simulation through a textual interface. In order to manage the simulated project, the student can hire team members, assign tasks to them and monitor the project progress. The players' goal in this simulation game is to successful complete the software project. When the project is complete, the student can analyze his/her performance using an analysis tool. This enables students to understand the overall project results, as a consequence of their actions and management decisions. The main limitation of these simulation games is that they have two contrasting goals: to be fun and also to provide an interesting educational value. It is also important to find a balance between the level of challenge offered and the required level of skills. An unbalanced contest with low challenge and high skill games requirement results in boredom; and high challenge, with low game skill generates anxiety [117].

Navarro et al. [111] used a game-based simulation tool called SimSE, a computer based environment that allows the creation and simulation of software development processes. It is a single player game in which students play the role of a project manager in charge of a development team. These researchers reported one semester of usage of this game and indicate that sometimes, students do not understand why they were not able to finish the game properly, but after the instructor's explanation, they understood what went wrong.

Shaw and Dermoudy [152] developed a simulation game for two software development life cycles. The game allows students to gain experience managing a simulated software development project in an environment that is both, graphical and entertaining. These researchers point out that using simulations allows those involved to experiment with the project, which would not be possible in a real-world scenario. The likely cost of implementing changes, potential consequences or even dangers that could result from experimenting with a real system all make simulations an attractive alternative [151]. Shaw and Dermoudy conclude that students benefit from the experience of playing simulation game. Students were successfully able to identify the reasons why their simulated projects had failed or succeeded, and in the case of failure, determine a strategy that would allow them to avoid this in the future.

2.1.5 Hands-on Remote Labs

Bayliss and Strout [10] report an experience of a hands-on remote lab using mediated reality (i.e., all project meetings were computer-mediated). The course was available online and roles were assigned to students and the roles were played in pairs. The authors tracked the students after the course and found that the students who performed well in the course, were able to advance on placement tests (tests used to check the academic skill levels of entering students, the goal is to place each student in classes at the right level) quickly in their undergraduate courses. Monasor et al. [168, 106] propose the use of a mixture of hands-on remote labs and games to improve student soft skills needed in global software development.

2.1.6 Service-Learning

The term service-learning was coined in 1967 to describe the educational practice and philosophy of integrating classroom concepts with a related community service experience. Students develop and use their academic skills to address real life problems within their own environment [141].

Liu [86] proposed that instructors choose an open-source project to help, and that every offering of a course help the open-source community. The author proposed that the instructor choose a project, make an effort to set a contribution to the project and to be able to make a real connection with the community that works on the project. It takes time to be able to make real contributions to an open-source project and sometimes one semester or more. Considering the time students need to understand the project and start delivering working code, the course time is not enough.

Ellis et al. [45] introduces students to the open-source development model to attract a wider variety of students into computing with a service learning software engineering project. They work on a open-source project called Sahana, which has a humanitarian goal. Ellis et al. reported that the students were really engaged in the project, because they were able to envision the real impact of what they were doing; but they also reported that there is a steep learning curve when working with open-source development projects.

2.1.7 Case Studies Method

Bernstein and Klappholz [12] created the idea of live-thru case stories, which is a teaching method developed specifically to enhance students' visibility about the software development process. The method involves discussing failed case stories and recognizing the oversights and mistakes of others. The authors found that by discussing failed case stories and having the students read similar case studies, students were able to recognize other oversights and mistakes but it did not help them recognize their own mistakes. This method focuses on training students to perform software development, and it seems to be a very powerful experiential learning tool. The authors report that the analysis of real situation failures allows students to internalize several issues, such as the need for a software process, the team member attitudes towards customer interaction, the developer's responsibility towards customers. However, live-thru case stories have several limitations, for example they are time consuming.

2.2 Definition of Software Processes for Academic Scenarios

Paula [123] defined a set of "minimal" requirements that a software process has to address in order to match the features of a software development academic scenario. These requirements consider several aspects of such a process:

- *Process architecture* - the educational process should be clearly defined, with its inputs, activities and their inputs and outputs, deliverables and milestones.
- *Team orientation* - the process should be designed for use by small software teams (3-5 students); larger teams are harder to coordinate and unnecessarily increase the development complexity.
- *Project cycle time* - an educational process should last at least four months, and allow a longer life cycle if needed.

- *Standards and practices* - an educational process should expose the student to recognized standards and best practices.

- *Student support* - the process should include materials that support the standards and best practices that students need to learn. Templates should also be provided to help students understand what they have to accomplish.

- *Instructor support* - instructors also need supporting material, such as slides, templates, exercises, assessments goals and as well as the process itself.

This set of "minimal" requirements that Paula [123] presented were taken into consideration for the definition EduProcess.

As mentioned in Chapter 1, project-based software engineering courses tend to use disciplined processes and capstone courses use agile approaches. The first ones are structured and based on phases and roles (process-centric), whereas agile ones focus on the product to be built and based on team self-organization (product-centric) [135, 153].

There are also some industry-oriented software processes that could be adapted for use in academic scenarios; however, in the case of disciplined processes, they tend to be extensive, poorly specified, and they do not include the instructional support for the students and the teaching team. The next two sections discuss the most relevant disciplined and agile processes reported in the literature, considering the academic scenario.

There is research that report the use of more than one process approach simultaneously. The report of these experiences does not indicate which activities of a particular process were used, nor how they were connected with the activities of other processes [140, 172]. Therefore, these mix of processes cannot be repeated or classified as a disciplined or agile process. There are also some reports on the use of ad-hoc process, a process designed specifically to support a software development in a certain work context. This type of process usually uses other processes as source, and then extends or tunes it to address a specific work context. The ad-hoc processes are generally created to fit specific course contexts, which do not fit typical software engineering courses [171].

2.2.1 Disciplined Software Processes

Disciplined processes provide a recipe for inexperienced developers, which helps students internalize the theoretical knowledge acquired in a first software engineering course [135]. When we look at industry, we see that they follow a disciplined software development processes, but adapting these processes to a course schedule requires extensive trimming and tailoring. Table 2.1 indicates the software processes used in educational contexts according to the literature, as well as the weaknesses that were reported or identified in each case.

Table 2.1: Disciplined Software Processes for the Academia

Disciplined Process	**Description**	**Limitations**
UPEDU – Unified Process for EDUcation [136]	This is a customization of the RUP 2000 to be used in academic scenarios.	Germain et al. [54] reported that students need training to use UPEDU. In a previous study these researchers indicated that this process forces students to start implementation too late (at the middle of the course) [55].
PSP – Personal Software Process [65]	A structured process that is intended to help developers understand and improve their performance by using a disciplined and data-driven process. The PSP intends to apply the underlying principles of the CMM (Capability Maturity Model) to the software development practices of a single developer.	This process is not team-oriented; therefore it does not match the work context considered in this thesis.
TSP – Team Software Process [66]	Provides a defined operational process framework that is designed to help managers and engineers organize projects and produce software products. TSP users must first learn to use PSP.	Umphress et al. [166] used the TSP in an academic scenario and reported that students found the process overwhelming. Paula [123] also reports that it does not fit the general contexts that can be found in academic scenarios; he states that this process can have better results when applied with advanced students, who already know and have experience with software engineering issues.
MBASE – Model Based Architecting and Software Engineering [169]	The MBASE framework is an approach for the development of software systems that integrates the system's process, product, property, and success models. The essence of the MBASE approach is to develop the system definition elements concurrently, through iterative refinement, using a risk-driven, Win-Win Spiral approach.	MBASE uses activities in a spiral lifecycle model that includes all MBASE process disciplines in every phase and iteration, so students are required to start the course project with at least all the theoretical knowledge of those disciplines in order to be able to use the MBASE process [125].

Disciplined Process	**Description**	**Limitations**
Praxis [124]	Praxis is a structured iterative process with fixed duration iterations, high student commitment and experience is expected (used mainly in graduate software engineering classes).	This process was designed for use in courses where students (graduate students) already have theoretical knowledge of SE. This process considers projects lasting more than a regular term (typically, around 25 weeks) [124].
Waterfall [128]	Waterfall process includes the typical software lifecycle stages: conception, analysis, design, construction, testing, and maintenance. Development is carried out using a structured process, in which clients and users have minimal participation.	According to Offut [116], "In modern software engineering projects, the traditional waterfall model is almost never used, and the ideas about process change constantly often much faster than textbooks or instructors' knowledge". The main limitation of this process is the lack of participation that clients and users have in the development process.
SSP – Simple Software Process [115]	Simple Software Process is a process created to support the development of small information systems in immature scenarios and involving inexperienced developers. Such a process has been used quite successfully by the authors to support students enrolled in project-based SE courses.	The process and its architecture are not available and the report of its use does not provide enough information to allow other instructors to use it. In other words, it is not specified enough as to be used by other people.

Table 2.2 compares the disciplined software development process mentioned in Table 2.1 according to the set of "minimal" requirements [123] for software processes for use in academic scenarios.

Table 2.2: Comparison of Disciplined Software Development Processes with the Set of "Minimal" Requirements for an Academic Software Process

Disciplined Process	**"Minimal" Requirement Item**	**Coverage**
UPEDU – Unified Process for EDUcation [136]	• Process architecture • Team orientation • Project cycle time • Standards and practices • Student support • Instructor support	• There is a webpage [161] where the architecture is well defined with all the inputs, activities, outputs, deliverables and milestones. • It is team-oriented with nine active roles besides clients and users. • It is based on Agile standards. • It uses standards and practices. • It does not offer any student support. • It does not offer any instructor support.
PSP – Personal Software Process [65]	• Process architecture • Team orientation • Project cycle time • Standards and practices • Student support • Instructor support	• It has a well-defined architecture [149] with all the inputs, activities, outputs, deliverables and milestones. • It is not team-oriented, it is a personal software process. • There is no specific cycle time. • It uses some standards and practices. • It does not offer any student support. • It does not offer any instructor support.
TSP – Team Software Process [66]	• Process architecture • Team orientation • Project cycle time • Standards and practices • Student support • Instructor support	• It has a well-defined architecture [150] with all the inputs, activities, outputs, deliverables and milestones. • It is team-oriented. • There is no specific cycle time. • It uses some standards and practices. • It does not offer any student support. • It does not offer any instructor support.
MBASE – Model Based Architecting and Software Engineering [169]	• Process architecture • Team orientation • Project cycle time • Standards and practices • Student support • Instructor support	• It has a defined architecture but it requires previous knowledge of the process prior to its use. • It is a team process but there are no roles assigned or team size specified. • It has a spiral life cycle that can be changed and used as many times as needed. • It uses some standards and practices. • It does not offer any student support. • It does not offer any instructor support.

Disciplined Process	**"Minimal" Requirement Item**	**Coverage**
Praxis [124]	• Process architecture • Team orientation • Project cycle time • Standards and practices • Student support • Instructor support	• Process architecture is available online where the architecture is well defined with all the inputs, activities, outputs, deliverables and milestones. • It has a process orientation but the roles required in the process should be more "specialized" than what we can expect from undergraduate software engineering students (quality manager, configuration control board, process engineer, quality engineer, data administrator, technical writer, test runner, configuration manager). • It was designed for use in projects with a duration of around 25 weeks. • It uses standards and practices. • It does not offer any student support. • There is a book for the instructor, where they present a software engineering course using Praxis, but the book is written in Portuguese [40] and is only printed on demand.
Waterfall [128]	• Process architecture • Team orientation • Project cycle time • Standards and practices • Student support • Instructor support	• It has a defined process architecture. • It is team-oriented, but there are no assigned roles or team size specification. • It normally has one cycle time. • It is the traditional process and it is considered a standard. • It does not offer any student support. • It does not offer any instructor support.
SSP – Simple Software Process [115]	• Process architecture • Team orientation • Project cycle time • Standards and practices • Student support • Instructor support	• It has a defined architecture but there is not enough information available related to it. • It is team-oriented. • The project cycle time is sixteen weeks. • It uses standards and practices. • It does not offer any student support. • It does not offer any instructor support.

Paula [123] presented in his set of "minimal" requirements important points to take into consideration when evaluating and/or creating a software development process to be used in academia. However, it is also important to evaluate if the process considers clients, if it has

enough material so as to be followed by someone step-by-step with examples (work product), if it considers software and student evaluation, if it can be used by novice students (without practical experience) and if it can be used in a loosely-coupled work environment.

Taking these points into consideration, the presence of a client is considered in only three processes: Praxis, UPEDU and SSP. UPEDU and Praxis also have enough material to be followed step-by-step, but they do not have examples of the work products to be generated. None of the SE processes proposes or considers evaluations, neither for the software nor the students. The processes that can be used by novice students are TSP, PSP, Waterfall and SSP.

2.2.2 Agile Software Development Processes

As mentioned before, agile processes are more suited to be used in capstone courses [89], where students have to gain a broader perspective of the software development practice and reinforce their technical and non-technical skills before being part of the software industry. The agile practices require students to have ample time to work together, since their activities are more tightly-coupled, e.g., pair programming or war room. Table 2.3 shows agile processes and also the weaknesses that were reported when applying them in academic scenarios.

Table 2.3: Agile Software Processes for the Academia

Disciplined Process	**Description**	**Limitations**
OpenUP [59]	OpenUP is a software process framework that is use-case driven, architecture centric, uses an iterative and incremental process. OpenUP only defines core workflows, which allow incorporating additional methods for project management, quality management, and change management very easily.	According to Paula [123], it is a software process oriented towards large teams and does not offer support for students or instructors. OpenUP does not have the role of project manager, everyone is responsible for planning and coordinating all the other roles.
Extreme Programming (XP) [128]	XP is a software development methodology which is intended to improve software quality and responsiveness to changing customer requirements. It advocates frequent "releases" in short development cycles, which is intended to improve productivity.	Some capstone software engineering courses use XP as a software process [73, 166, 62], where students reinforce their technical and non-technical skills before finishing college.

Disciplined Process	Description	Limitations
Scrum [160]	Scrum is an iterative and incremental agile software process. According to Takeuchi defines “a flexible, holistic product development strategy, where a development team works as a unit to reach a common goal”.	One of the basic goals of Scrum is to enable teams to self-organize by encouraging physical co-location or close online collaboration of all team members, as well as daily face-to-face communication among all team members. Many capstone software engineering courses use Scrum [158, 90, 137] to gain a broader perspective of software engineering development.
Other Agile Approaches [28]	There are other agile processes (e.g.; Crystal, TDD- Test Driven Development) and they are all contemporary software engineering approaches based on teamwork, customer collaboration, iterative development, people, process and technology adaptable to change.	Processes that call themselves “agile” tend to focus on just a few of the relevant software engineering aspects privileging implementation, and leaving out key practices, such as requirements elicitation, analysis and management [124]. There are some reports of their use in capstone courses [76, 173].

Table 2.4: Comparison of Agile Software Development Processes with the Set of “Minimal” Requirements for an Academic Software Process

Disciplined Process	“Minimal” Requirement Item	Coverage
OpenUP [59]	• Process architecture • Team orientation • Project cycle time • Standards and practices • Student support • Instructor support	• There is a webpage [31] where the architecture is defined with all the inputs, activities, outputs, deliverables and milestones. • It is a team-orientated process with eight active roles besides the stakeholders. • It is based on agile best practices. • It uses agile standards and practices. • It does not offer any student support. • It offers support for the course developer (creates training materials) and for the trainer (delivers training to end-users) but does not offer any instructor support.

Disciplined Process	**“Minimal” Requirement Item**	**Coverage**
Extreme Programming (XP) [128]	• Process architecture • Team orientation • Project cycle time • Standards and practices • Student support • Instructor support	• It has no formal architecture, it uses the agile principle related to minimal architecture. • It is team-oriented, but tightly coupled. The team must work together to be able to work according to XP principles (pair programming and constant feedback). • The project cycle times are short - normally one or two weeks. • It was the first process of the agile movement and is considered a standard. • It does not offer any student support. • It does not offer any instructor support.
Scrum [160]	• Process architecture • Team orientation • Project cycle time • Standards and practices • Student support • Instructor support	• It has a very loose architecture based on team understanding and communication. • It is team-oriented, but tightly coupled. • The project cycle times are short - normally one week. • It is an agile standard. • It does not offer any student support. • It does not offer any instructor support.
Other Agile Approaches [28]	• Process architecture • Team orientation • Project cycle time • Standards and practices • Student support • Instructor support	• They have a loose architecture with inputs, activities, outputs, deliverables and milestones. • They are team-oriented, but tightly coupled; the team must work together. • The project cycle times are short normally in weeks. • They use agile standard and practices. • They do not offer any student support. • They do not offer any instructor support.

We now analyze these processes using the same criteria we used to discuss the disciplined process. The client is considered in all the agile processes. The first three have enough material to be followed step-by-step and some processes have work product examples. None of these processes considers evaluations, for the software or the students. The processes can be used by novices, but it is encouraged that students have prior experience in developing software before using agile processes [153]. Moreover, none of them were created for use in a loosely-coupled work environment.

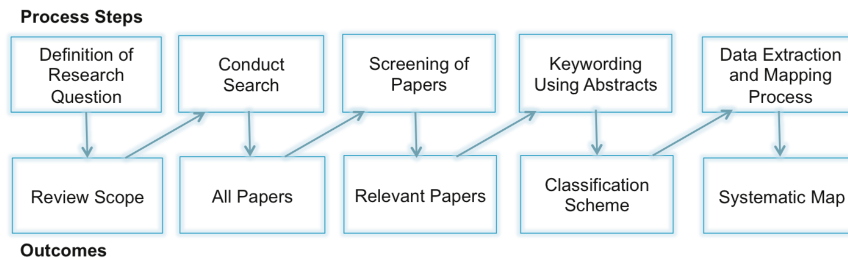


Figure 2.1: Systematic Mapping Process by Petersen et al. [126]

2.3 Matching Instructional Approaches And Software Processes

In order to understand the use of the different instructional approaches and software processes according to the literature, we conducted a systematic mapping study [102]. In this mapping study we followed the procedure proposed by Petersen et al. [126]. The search string used was: "software engineering education" OR "software development course" OR "software development capstone" OR "software development education" OR "software engineering capstone" OR "software engineering course" OR "teaching software engineering" OR "software engineering instruction" OR "teaching software development" OR "software practice education" OR "software engineering projects". The search was performed in the following scientific databases: IEEEXplore, ACM Digital Library, Web of Knowledge (former ISI Web of Science), Science Direct (Elsevier), SpringerLink and Wiley International. A total of 7,517 documents were found.

Provided that many papers are indexed by more than one digital library, we removed duplicate articles and thus obtained a corpus of 3,725 documents related to the research topic; the papers that were indexed by more than one digital library were considered in the library of origin (publisher). As a result of the procedure proposed by Petersen et al. [126] presented in Figure 2.1, we ended up with 173 papers selected as primary studies. The next two subsections discuss the instructional approaches and software processes used in the experiences reported in these papers, and subsection 2.3.3 presents the mapping.

2.3.1 Instructional Approaches Considered in the Systematic Study

In Section 2.1 we presented nine different instructional approaches, and some of these were not found in the systematic mapping review. In the specific case of Hands-on Methods (Section 2.1.4) here we present them divided in four approaches: learning-by-doing, game-based learning, maintenance projects and open source projects; because these are the main approaches reported, and the other approaches reported can be seen as subcategories from these four.

Today the development of new solutions, typically through learning-by-doing mechanisms, is much more frequent than the extension of the existing ones. Other approaches such as games, open source and PBL/OBL have started to be reported over the last ten years and continue being used as an attempt to find new ways of addressing the stated challenge of how

to teach software engineering (see Figure 2.2).

Figure 2.2 shows that learning-by-doing is the most used instructional approach, with 93 studies (54%), followed by the PBL/OBL approach with 21 studies (12%), case studies with 15 studies (9%), games with 12 studies (7%), and then simulation (5%), traditional (4%), open source (3%), service learning (2%) and inverted classroom with only one study.

Learning-by-doing and also PBL/OBL are flexible strategies that can be adjusted to take advantage of the capabilities and interests of the instructors and students. These approaches do not impose important constraints or require special capabilities of the instructors. This could be a reason why these alternatives seem to be frequently used to support these practical experiences.

2.3.2 Software Processes Considered in the Systematic Study

In Section 2.2 we presented disciplined software development processes (UPEDU, PSP/TSP, MBASE, Praxis, Waterfall and SSP), and also the agile development processes (OpenUP, XP, Scrum and other agile approaches). However, in the systematic mapping review, few reports indicated the flavor of agile process used in practice. Therefore, we decided to group the use of agile processes under the same category with such a name.

The waterfall process is called traditional, since it is considered the most known and used. Some researchers reported the use of more than one process approach simultaneously. For instance, using a structured model for developing a particular software, so we classified them as various (due to various software processes being used). We also have a category called ad-hoc, when the paper reports the use of a specific process that does not match any of the other categories. Moreover, we also have a category called "not specified" when the authors did not state the process approach used to address the work.

2.3.3 Mapping Instructional Approaches and Software Processes

By considering the previously presented instructional approaches and software processes we identified the matching between them according to the 173 papers selected as primary studies. Figure 2.3 shows a bubble plot of instructional approaches and software processes. The most used instructional approaches are learning-by-doing and PBL/OBL. Most of the papers do not report the use of a software development process or they do not specify what they are using. However, most reports indicating the process that was used mention the use of agile or ad-hoc processes. Particularly, learning-by-doing and agile seems to be the most identifiable combination.

Finding this combination is not surprising since, according to what was presented in Section 1.1.2 (Challenges for Implementing Practical Experiences in SE Courses), it represents a less demanding approach for instructors. Although such a combination seems to be suitable for addressing capstone courses, it is not used to guide software developments in project-based courses, given that a disciplined software process is required in such type of course, as indicated in Section 1.1.1 (Types of SE Project Courses).

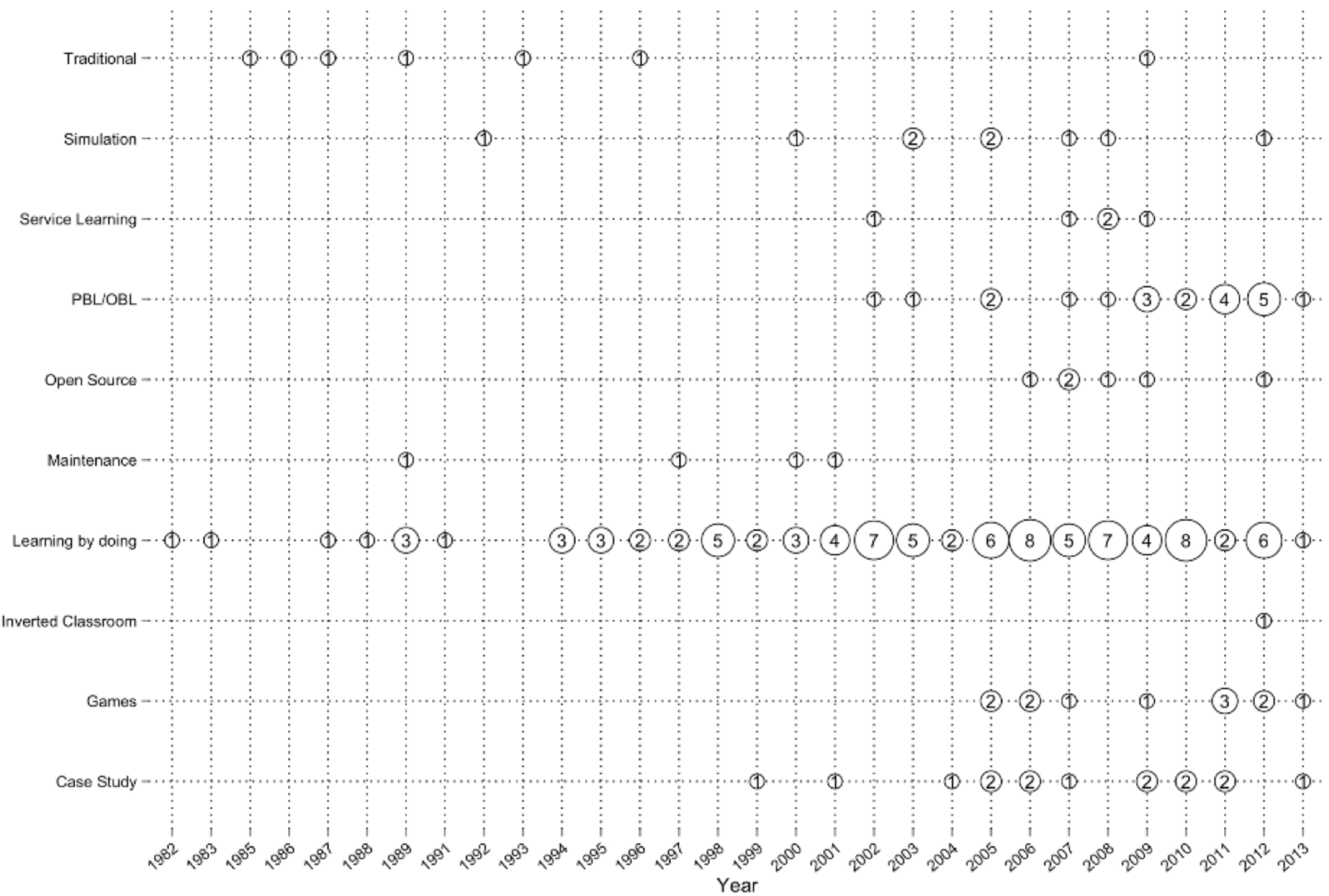


Figure 2.2: SE Instructional Approaches Reported by Year

2.4 Instructional Practices for Software Engineering Courses

The literature also reports several instructional mechanisms (or practices) that can be used jointly with software processes to improve the students experience during project-based software engineering courses. The course characteristics establish the type of software process that is more appropriate (i.e., disciplined or agile), and consequently, the instructional strategies and tools that can be used to support students in the acquisition of technical and non-technical skills during project-based courses.

In global software development there are proposals that use virtual agents to facilitate communication and coordination channels to minimize the need of face-to-face coordination, thus reducing the instructors' workload and scheduling problems [104, 105]. Virtual agents can provide learners with opportunities for self-reflection and self-correction by explaining the consequences and rationale of their actions with regard to team ethics and cultural differences.

One of the strategies used to make the learning experiences more fruitful is coaching, where an experienced developer or instructor assists student teams while they work on their project. This approach is suitable for being used in disciplined and also in agile developments. Recognizing that there are several coaching approaches, the most typical ones focus on helping students address internal issues (e.g., communication and coordination) by mentoring students to use their own knowledge and reviewing their deliverables [7, 25]. Rodríguez et al. [138] propose using an agile coach with pedagogical expertise, who monitors the students' accomplishments and suggests corrective actions and improvements. The evaluation results indicate that students who were coached following agile principles perceived a greater gain in non-technical skills than non-coached students. Despite the good results reported by Rodríguez et al., this approach is more appropriate for capstone courses, since inexperienced students tend to fall back upon the coach, not accepting the responsibility and risks of their projects. If the coach becomes a project manager, then students miss out on a good opportunity to learn more from these experiences.

Monitoring can also be used to make project experiences more valuable for students and instructors. Several researchers indicate that team monitoring reduces motivational losses and increases the likelihood of detecting free riding and social loafing [39, 91]. This helps the team focus their effort on reaching the team goals over individual interests [71]. Monitoring can operate as an implicit coordination mechanism [134] by making team members more aware of the other team members' actions, timing, and performance. Thus, people increase their ability to synchronize their contributions in ways that maximize the attainment of team goals [94].

Typically, in disciplined processes, students conduct loosely-coupled work to address the project tasks. Due to the effort and complexity of monitoring their activities, software tools are used by instructors and teaching assistants. For instance, Collofello [30] proposed a tool where students report the project status on a weekly basis. However, this system only relies on information provided by the students, and therefore the diagnosis could be inaccurate or wrong. A similar tool was proposed by Bakar et al. [6], which also depends on the input provided by students. This characteristic potentially undermines the usefulness of these

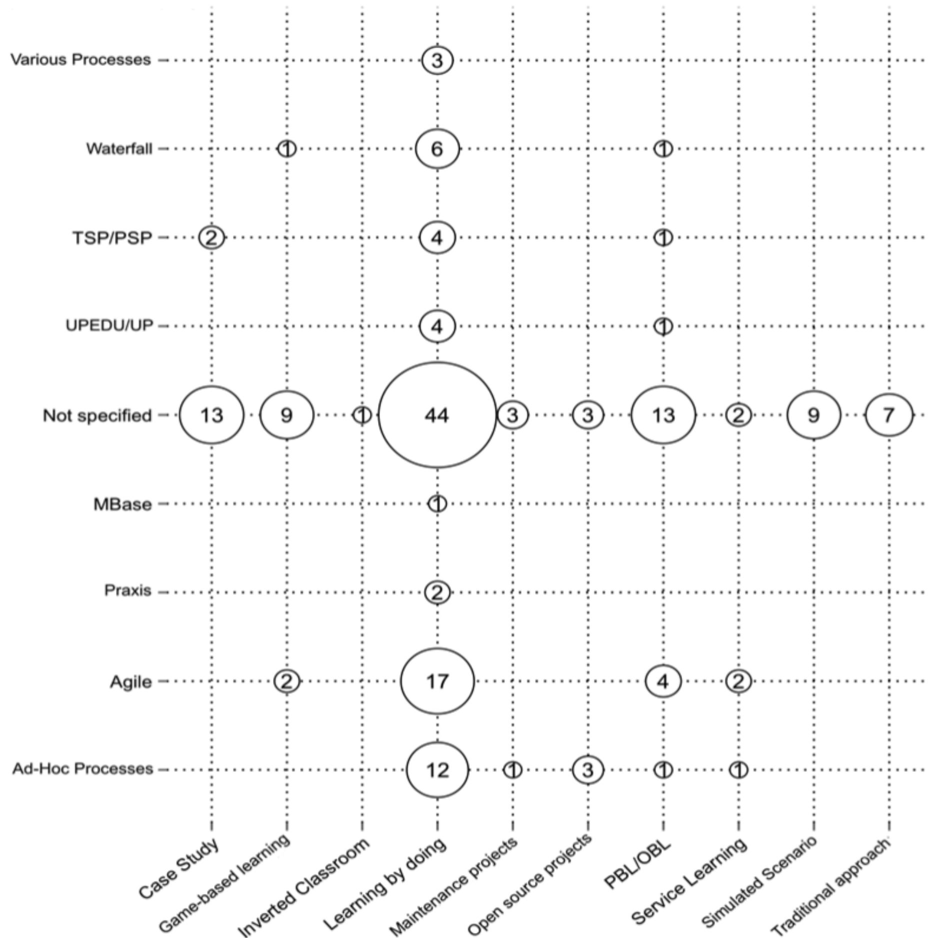


Figure 2.3: Instructional Approaches Reported

systems for several reasons. For instance, students tend to report their activities at the last possible minute, so they report what they remember and not what they have really done. In other cases, they report what is required to avoid conflicts with other team members and the course instructors.

More objective monitoring mechanisms have been proposed based on mining the log files recorded by the tools used by students for supporting their activities, e.g., version control systems [70, 127], issue trackers [43, 72], wikis [72], and mailing lists [43, 72]. Although these mechanisms are useful, they only capture the activities mediated by such tools, which could be appropriate for monitoring global software development scenarios, but are not completely transferrable to project-based and capstone courses, where students and monitors frequently conduct face-to-face interactions.

In the case of using Scrum as software development strategy, Scrum Masters can perform team monitoring. Mahnic [90] proposes that the instructor should assume this role in capstone courses by ensuring that everyone follows Scrum rules and practices. Rodríguez et al. [137], as well as Scharf and Koch [143] agree that the Scrum Master should lead teams applying Scrum practices, but they propose that students with advanced knowledge of the agile philosophy

assume this role. Unfortunately, most agile practices require students with previous experience and that they spend time working together, which is not feasible in a first project-based course.

When using a disciplined software process, the team can have a project manager that assigns tasks and oversees their fulfillment. However, in the context of a software engineering course, if the project manager is the instructor, the demand on the instructor to fulfill these two roles (project manager and instructor) is high. If the manager is a teammate, there is no leverage between students, despite ones' goodwill, to make them do what a peer is asking [132]. In industry there is always something at stake.

In order to enhance project-based learning experiences in this context, some researchers propose the use of e-portfolios where students store their own work products and keep a logbook of their activities and reflections on a particular project [21, 89]. These systems are accessible by the instructor, who can provide feedback and also assess the performance of his/her students. These tools can be used jointly with disciplined and agile development processes. Despite the positive results reported in the literature [89], the use of e-portfolios also produces tension between the privacy of the students' thoughts and the knowledge to be shared with the instructor [21], which may lead to discomfort among students. Moreover, when the e-portfolio information is used as part of the student assessment, there is not much collaboration among students for reasons of plagiarism [21], which further reduces the chances to reinforce learning during these practical experiences.

Following the same goal, project-base learning, Mahnic and Casar [91] propose the use of a tool similar to an e-portfolio that allows the visualization of students' activities in Scrum-based software engineering capstone courses. The visual exposition of task accomplishment not only allows instructor to monitor students, but also prevents procrastination and the "free-rider" syndrome. This tool seems appropriate for monitoring students; however, it was conceived particularly to support teams that use an agile software development process. Therefore, it is probably unsuitable for disciplined developments given that both strategies involve different requirements and are used with different purposes in the academic scenario.

Reflection is important in all forms of software engineering education, be it formal learning in an educational institution or informal learning as part of professional work life. This is what Schon [146, 155] referred to as "reflection-in-action", or reflection during the problem-solving process. In agile software processes, the client provides constant feedback and the tightly-coupled work that team have to face make the reflection-in-action part of the process. However, in disciplined software process the reflection is done at the end of the process in a lessons learned session or document; and it is not normally something people really reflect about [18]. There is a need of reflection in all software engineering projects and the sooner the reflection is done, the sooner problems can be mitigated [18].

Another option for supporting students during these learning experiences is by implementing instances of peer-assessment [25, 154], where they anonymously evaluate the performance of their teammates and provide feedback in order to help their peers overcome conflicts and limitations. This approach has shown good results and is suitable for being used in disciplined developments. However, the comments given to students are mostly based on personal experiences of team members that may not represent the overall team consensus.

Moreover, the feedback given to students through these instruments is not always focused on the key aspects of the team dynamics, due to the lack of student expertise in identifying such aspects [25]. Similar limitations can be found in self-assessment [61, 56].

Summarizing, the literature reports various software processes that could potentially be used to support project-based courses, and also some instructional practices that can be combined with those software processes to make the implementation of practical experiences affordable mainly for instructors, but also for students and stakeholders. Regardless of the limitations that these software processes and instructional practices can have to support project-based courses, it is not clear how to combine them and what to expect of certain combinations. Therefore, process proposals that embed both aspects (i.e., software development and instructional process) are required to help instructors properly address the development experiences in project-based and capstone courses. In this sense, this thesis work makes a step forward in trying to understand and support software engineering practical experiences in project-based courses.

Chapter 3

Identifying the SE Best Practices to Be Taught in Undergraduate Programs

This chapter presents a study that tries to determine which SE best practices should be taught to the students as part of their software engineering education. Moreover, it presents a set of instructional practices that can be used by instructors to help address the practical experiences.

The next section reviews the ACM-IEEE Software Engineering Curriculum Guidelines that establishes the corpus of knowledge and their corresponding best practices, which should be taught and used in practice by students in their undergraduate studies. Given that such corpus of knowledge is too large and impossible to address through an undergraduate Computer Science curricula, we conducted a systematic literature review (see Section 3.2) to determine which knowledge units are reported as used in software engineering education. Moreover, we conducted an interview study involving software engineering instructors belonging to relevant Chilean universities in order to elicit the practices that are taught in undergraduate programs and also the way in which they are taught (theoretical, practical or a mix). The results (see Section 3.3) present the results of the status report of the SE education in Chile, including the teaching of the best practices recommended in the 2014 ACM-IEEE Software Engineering Curriculum Guidelines [119].

3.1 ACM-IEEE SE Curriculum Guidelines

In 1999, Parnas [122] discussed the need and the difference between software engineering and computer science curricula. In order to address this issue, ACM and IEEE-CS formed the Software Engineering Education Project, which ended in 2003 with the publication of the Software Engineering Curriculum Guidelines (SE2004) [81], where the need of practical experience was reflected in both guidelines and on the curriculum itself. The SE2004 is a recommendation for an undergraduate curriculum for software engineering that should be taught in four years. In the first year, students get familiar with the basics of software engineering, and in the second and third years they study the core knowledge of software engineering through specific courses. This core knowledge should be used in practice through methods and technics known as “best practices”. These best practices come from experience and research, and they have proven to reliably lead to a desired result in software development.

Table 3.1: Specific Areas Considered in the 2004 - ACM-IEEE SE Curriculum Guidelines

Knowledge Area	Units of Knowledge
Software Modeling and Analysis.	Modeling foundations, Types of models, Analysis fundamentals, Requirements fundamentals, Eliciting requirements, Requirements specification and documentation, Requirements validation.
Software Design.	Design concepts, Design strategies, Architectural design, Human computer interface design, Detailed design, Design support tools and evaluation.
Software Verification and Validation.	Verification and Validation terminology and foundations, Review, Testing, Human computer UI testing and evaluation, Problem analysis and reporting.
Software Evolution.	Evolution process, Evolution activities.
Software Process.	Process concepts, Process implementation.
Software Quality.	Software quality concepts and culture, Software quality standards, Software quality process, Process assurance, Product assurance.
Software Management.	Management concepts, Project planning, project personnel and organization, Project control, Software configuration management.

Finally, everything is put together in a final capstone project course that usually takes place during the fourth year, where the students use best practices related to the SE core knowledge. SE2004 is divided in ten knowledge areas and each knowledge is divided in units of knowledge that have specific topics to be taught. Table 3.1 shows the knowledge areas and units of knowledge directly related to SE.

In 2010, a task force was appointed by the ACM and IEEE CS to determine whether SE2004 needed updating, and, if so, how much effort would be required to complete these updates. The task force reached out to academia, industry, and government through workshops at technical conferences and online surveys. Members of the task force identified sections of the original guidelines that needed updating and started making revisions. During this process, they continued to reach out to stakeholders through presentations and workshops at technical conferences. At the end of 2015, a new version was published: the 2014 ACM/IEEE Software Engineering Curriculum Guidelines (SE2014) [119]. The last version of this recommendation considers ten knowledge areas: the same three basic ones (*Computing Essentials*, *Mathematical & Engineering Fundamentals* and *Professional Practice*) and seven specific areas about software engineering (shown in Table 3.2). They created two specific new knowledge areas: *Requirements Analysis and Specification* (in the 2004 version it was part of the *Software Modeling and Analysis Knowledge* Area) and *Security* (in the 2004 version it was not considered in full, it was only a topic of *Quality*). They also divided the *Software Management* knowledge area and put the units of knowledge in other knowledge areas. They also put *Software Evolution* inside *Software Process*. Each knowledge area is still divided into units of knowledge and each unit describes the topics that should be taught. Table 3.2 presents the specific knowledge areas and their corresponding units of knowledge that are directly related to SE.

Table 3.2: Specific Areas Considered in the 2014 - ACM-IEEE SE Curriculum Guidelines

Knowledge Area	Units of Knowledge
Software Modeling and Analysis.	Modeling foundations, Types of models, Analysis fundamentals.
Requirements Analysis and Specification.	Requirements fundamentals, Eliciting requirements, Requirements specification and documentation, Requirements validation.
Software Design.	Design concepts, Design strategies, Architectural design, Human computer interaction design, Detailed design, Design evaluation.
Software Verification and Validation.	Verification and Validation terminology and foundations, Review and static analysis, Testing, Problem analysis and reporting.
Software Process.	Process concepts, Process implementation, Project planning and tracking, Software configuration management, Evolution process and activities.
Software Quality.	Software quality concepts and culture, Process assurance, Product assurance.
Security.	Security fundamentals, Computer network security, Developing secure software.

Many universities that have software engineering programs have the SE2004 implemented and the ones that do not have it fully implemented try to implement as much of it as possible. The SE2014 curriculum guidelines was published published in the end of 2015, when we did this study there were no reports of its implementation. So, the SE Curriculum Guidelines can be considered a set of software engineering education best practices.

The SE curriculum guidelines (SE2004 and SE2014) present all the knowledge areas (units of knowledge and topics) that should be taught in a software engineer program. However, in some contexts, the undergraduate programs are in Computer Science (CS) or Informatics, where software engineering is one of the several areas that must be taught. Therefore, it is not possible to deliver all units of knowledge included in these guidelines to students. Considering this situation, a doubt emerges: how should a program choose the appropriate units of knowledge (areas of knowledge) to teach? In order to clarify this point, we carried out a systematic literature review to find out what the literature reports as SE best practices that must be taught to students enrolled in CS and informatics programs.

3.2 SE Best Practices

Software engineering education is concerned with finding a good way for teaching students how to analyze problems, evaluate alternatives and apply the technical skills they have learned (related to computer science and its application domain) in order to be able to develop high quality software solutions for their clients.

To do that, a lot has been said about the use and teaching of software engineering "best practices". But, what are software engineering "best practices"? Where can we find them? Typically, a best practice is a well-defined method that contributes to a successful step in

software development. According to Surendran [159], the Software Engineering Body of Knowledge (SWEBOK) [1] is probably the best-known reference of best practices.

The ACM-IEEE Software Engineering Curriculum Guidelines [119] can also be considered a source of best practices that includes the SWEBOK. However, this recommendation includes so many units of knowledge (and their corresponding practices) that it is impossible to teach all of them in a Computer Science or Informatics undergraduate program. Therefore, the units to be taught should be selected, but determining which of them to teach is not clear at all. Moreover, Bailey et al. [5] mention that "as the software industry matures, new development methodologies are invented and some of these methodologies transition into best practices. Our role as university educators is to teach these best practices". This increases the complexity of the problem.

Thompson and Fox [162] found a lack of guidelines in their research of leading edge practices in Software Engineering. They also found that there was no clear indication of the best practices being used in industry. Therefore, the concern about what are the best practices that should be taught remains as an open question. Trying to address this question, we conducted a systematic literature review on software engineering "best practices".

3.2.1 SE Best Practices Being Taught

In a context where it is not possible to teach the whole curriculum guideline proposed by ACM/IEEE, the decision of which practices to teach or not, is not an easy task. Therefore we decided to review the literature to identify which practices were reported as being used in the context of software engineering education. Particularly, we performed a systematic literature review of the reported best practices being used in software engineering education. Moreover, we also compared these practices to the SE2014 curriculum guidelines to determine which of the practices that were part of the curriculum guidelines, were reported as taught and which ones are not being taught. One thing to have in mind is that not everything that is taught is always reported.

3.2.2 Methodology

Nowadays, systematic literature reviews are used as a means to consolidate scientific knowledge. One of the major strengths of this technique is that it follows a defined protocol that allows verifiability and rigor. To guarantee that, there are guidelines to follow. In our case we followed the guidelines of Kitchenham et al. [75]. So we did a systematic literature review, whose goal is to answer three research questions:

- RQ1 - Which software engineering best practices are reported as being taught in academia?
- RQ2 - Are they taught in a practical way? RQ2.1 - How?
- RQ3 - Was the use of software engineering "best practices" being taught validated somehow? RQ3.1 - What was the research approach used to report? The categories used are based on those proposed by Petersen et al. [126] and Nacimento et al. [110].

Table 3.3: Search Strings

Term	Keywords	Synonyms
A	software engineering education	"software engineering education"
B	best practices	"best practices", "recommended practices", "edge practices"

Table 3.4: Specific Search Strings for Each Digital Library

Digital Library	**Search String**
ACM	("software engineering education") and ("best practices" or "edge practices" or "recommended practices")
IEEE Xplore	(("software engineering education") AND ("best practices" OR "recommended practices" OR "edge practices"))
ScienceDirect	(("software engineering education") AND ("best practices" OR "recommended practices" OR "edge practices"))
Scopus	TITLE-ABS-KEY ("software engineering education") AND TITLE-ABS-KEY ("best practices" OR "recommended practices" OR "edge practices")
Springer Link	("software engineering education") AND ("best practices" OR "recommended practices" OR "edge practices")
Wiley	"software engineering education" in All Fields AND "best practices" OR "edge practices" OR "recommended practices" in All Fields

Search Strategy

Based on these questions, we identified the keywords to be used to search for the primary studies. Table 3.3 shows the words we are looking for and related synonyms that we considered in our search.

The search for primary studies was carried out on the following digital libraries: ACM Digital Library, IEEE Xplore Digital Library, ScienceDirect, Scopus, Springer Link and Wiley. These libraries were chosen because they are among the most relevant sources of scientific articles in SE education [85]. We did not consider limitations in terms of publication year and we performed our search at the beginning of April 2015, so only papers published before March 30, 2015 were considered.

Table 3.4 shows the specific search strings used in each database considering the restrictions that we have mentioned earlier, and Table 3.5 presents the results obtained performing the searches in the databases. A total of 712 primary studies were selected.

After performing the search on the selected databases, we evaluated the relevant primary studies found according to the following inclusion and exclusion criteria:

- Inclusion criteria:

Table 3.5: Number of the Papers Retrieved from Each Digital Library

Digital Library	Search Results
ACM	119
IEEE Xplore	321
ScienceDirect	69
Scopus	35
Springer Link	139
Wiley	29
Total	**712**

1. The paper is in English.
2. The paper is a peer-reviewed article and it was obtained from a journal, conference or workshop.
3. Documents reporting teaching/learning software engineering experiences.
4. Documents presented as full papers; short papers and work in progress.
5. The reported approach and its validation are reasonably present.

- Exclusion criteria:
  1. The paper is not available online.
  2. Documents whose full text is not available.
  3. Documents that are only available as an abstract.
  4. Documents describing a panel.
  5. Documents that are a letter from the editor.
  6. Documents in which the context is not related to software engineering education.
  7. Documents that report the use of software engineering as a means of teaching a specific programming language or framework.
  8. Documents that report opinion papers and philosophical theories.

After applying the inclusion and exclusion criteria on titles and abstracts, 61 papers were selected for the next phase. We them applied the inclusion and exclusion criteria after reading the full papers. We ended up with a final corpus of 15 papers as our final set. These numbers are summarized in Table 3.6.

3.2.3 Results

The study results indicate that 40% of the primary studies selected were published in the last four years; the others were published in a time span of 15 years. Table 3.7 shows the primary studies selected in our systematic literature review.

Table 3.6: Results After the Use of the Guidelines

Phase	Amount of papers
All papers	712
After paper screening	61
After full reading	15

Table 3.7: Primary Studies Selected

Year	Venue	Authors
1996	Conference	Tomayko [163]
1999	Conference	Upchurch et al. [167]
2001	Conference	Groth et al. [58]
2003	Conference	Bailey et al. [5]
2005	Journal	Schneider et al. [145]
2007	Conference	van der Duim et al. [170]
2009	Conference	Koolmanojwong et al. [5]
2010	Journal	Cheng et al. [25]
2011	Journal	Chen et al. [23]
2012	Conference	Neto et al. [112]
2012	Conference	Bareis et al. [8]
2013	Conference	Gary et al. [53]
2014	Journal	Rajapakse [132]
2014	Conference	Radermacher et al. [131]
2015	Journal	Lehtinen et al. [82]

It was surprising that from an initial set of 712 studies we ended up with only 15 (2.11%). We did a small sample search over some of the papers selected in the first phase to see what happened, and found that many of these papers only mention the words "best practices", but did not clearly state what they meant by best practices, what they are or anything else. Some examples: citation [50] states that "Students work with the faculty team members to make sure that best practices are being used and that academic requirements are met."; in [69] the author said that "... students can experience and appreciate the benefit of design principles and best practices only in a frame of larger projects than assignments"; and in [125] we can find that "Such processes should expose the students to well-known notations, techniques, and best practices, offering them practical grounds to decide about their adoption in actual work situations." The classification schema of the answers found for our first research question are on Table 3.8 and Table 3.9, which also include the selected Primary Studies (named as PS). These tables also present the publication year and the best practices reported.

Analyzing the 15 primary studies selected we expected to see the best software engineering practices taught to students (practices that students can use/incorporate in their developments), but we also found some instructional best practices used to teach software engineering. We therefore split the best practices into instructional best practices (teaching methodology) and software engineering best practices. Table 3.8 shows software engineering best practices (best practices to be applyed by students) and Table 3.9 it is possible to see the instructional best practices that were reported. Some authors reported these two types of best practices in the same paper [53][112][170].

Table 3.8: Reported Best Practices (to be applied by students)

PS	Year	Reported Best Practices
[163]	1996	**reflexive practices**, **inspections**, formal methods, **integrating process**, mentors, client satisfaction, boot camp, **requirements**, PSP
[167]	1999	project postmortems
[58]	2001	prototyping, **quality assurance**
[5]	2003	**code inspection**
[145]	2005	XP practices (planning game, small releases, metaphor, simple design, testing, refactoring, **pair programming**, collective ownership, **continuous integration**, forty-hour week, on-site customer and coding standards)
[170]	2007	contacts, reciprocity, feedback, time tasks, respect
[78]	2009	**requirement management**, object-oriented analysis and design, risk management, **quality management**, peer reviews, configuration management, and value based software engineering
[25]	2010	**coaching**, design reviews, prototyping
[112]	2012	on-line daily meetings, group leader, iterative and incremental development, activity redistribution, version control, self manageable groups, **on-line pair programming**
[8]	2012	**coaching, peer review**
[53]	2013	preparation, **reflection**
[132]	2014	**monitoring, peer review**
[131]	2014	design, testing, **requirements**, software life cycle, **monitoring**
[82]	2015	root cause analysis

Table 3.9: Reported Best Practices (instructional)

PS	Year	Reported Best Practices (instructional)
[170]	2007	active learning, high expectations, respect
[23]	2011	industrial involvement
[112]	2012	internal and external groups training
[53]	2013	practice (labs) and project centered learning

The primary studies reported a total of 54 best practices; from these, 9 are mentioned more than once and 45 best practices are reported as being taught in software engineering education. Practices reported being taught in more than one primary study (they are in bold text in Table 3.8 and in detail in Table 3.10). The other best practices are: active learning, activity redistribution, boot camp, client satisfaction, coding standards, collective ownership, configuration management, contacts, feedback, formal methods, forty-hour week, group leader, high expectations, industrial involvement, internal and external groups training, iterative and incremental development, mentors, metaphor, object-oriented analysis and design, on-line daily meetings, on-site customer, peer reviews, planning game, postmortems preparation, prototyping, PSP refactoring, risk management, root cause analysis, self-manageable groups, simple design, small releases, time tasks, value based software engineering and version control.

Various of the found practices are described with different levels of granularity in the

Table 3.10: Best Practices Reported in More than One PS

Reported Best Practices (to be applied by students)	PS1	PS2
Reflective practices	[163]	[53]
Inspections	[163]	[5]
Integration	[163]	[145]
Requirements	[163]	[78]
Quality	[58]	[78]
Pair programming	[145]	[112]
Coaching	[8]	[25]
Prototyping	[58]	[25]
Monitoring	[132]	[131]

Table 3.11: How was Validated the Use of the Best Practices (TiP - Taught in Practice)

PS	TiP?	How?
[163]	Yes	Studio (one year project course where students assume distinct roles over time and have industrial clients)
[167]	Yes	Reflexive writings at the end of the project based course
[58]	Yes	Project based course, reporting and monitoring formalized in four different paths
[5]	Yes	Individual project using inspections
[145]	Yes	The paper only states that they are taught
[170]	Yes	The paper only states that they are taught
[78]	Yes	The paper only states that they are taught
[25]	Yes	Project-based course in collaboration with industry
[23]	Yes	Project based course in collaboration with industry
[112]	Yes	The paper only states that they are taught
[8]	Yes	Project-based course, involving fictitious client
[53]	Yes	The paper only states that they are taught
[132]	Yes	Project-based course in collaboration with industry
[131]	Yes	Project-based course in collaboration with industry
[82]	Yes	Project based course and use of cause effect diagrams

primary studies. For example, it is clear what “code inspections” means, but it is not clear when “requirements” is mentioned, as requirements is a knowledge area that can have several subareas and can even have an entire course dedicated to the topic [81].

In Table 3.11, we show the Primary Studies selected, if they are taught in practice, practical focus (column TiP) and how they are being taught. We found that all the primary studies selected reported that the best practices were being taught in practice, but 45% of the studies did not report how these best practices were being taught. Forty-five percent of the studies reported that best practices were taught in project-based courses and only one study reported that the best practices were being taught using individual projects (individual homework style).

We also performed a comparison with the “best practices” that we found as being taught with the units of knowledge proposed in ACM-IEEE curriculum guidelines. The comparison was not straightforward because the granularity of the “best practices” found and the units

Table 3.12: ACM-IEEE SE2014 Curriculum x Reported “Best Practices”

Knowledge Area	Related Best Practices
Professional Practice	Reflexive practices, mentors, project post-mortems, metaphor, pair programming, collective ownership, forty-hour-week, contacts, reciprocity, feedback, respect, peer reviews, group leader, activity redistribution, self-manageable groups, coaching, and monitoring.
Requirements Analysis and Specification	Requirements engineering.
Software Design	Simple design, refactoring, object-oriented analysis and design.
Verification and Validation	Inspections, prototyping, peer reviews and testing.
Software Process	PSP, formal methods, continuous integration, configuration management, iterative and incremental development, version control, root cause analysis.
Software Quality	Client satisfaction, quality assurance, coding standards, quality management

of knowledge of the curriculum are quite different. For example, Table 3.8 presents the requirements engineering as one of the “best practices” being reported, but there is no detail on what exact is being done to teach requirements and in the curriculum guidelines there is a whole knowledge area about requirements (Table 3.1), which includes four units of knowledge. We therefore first performed a general comparison of knowledge areas and “best practices”. This comparison was conducted according to our interpretation of the meaning of each “best practice” reported. Table 3.12 shows this comparison. It shows that all the six knowledge areas that are related to software projects have at least one related “best practice” that is being taught.

In a second analysis, we compared the units of knowledge that each knowledge area has with the list of “best practices”. The comparison is in Table 3.12 where it is possible to see that from the 27 units of knowledge associated with software projects, 14 of them have a related “best practice”. This means that almost 52% of the units of knowledge of the ACM-IEEE Curriculum Guidelines are covered with some “best practice”. Half of the units of knowledge do not have a high amount of coverage, which can be considered surprising.

Why did this happen? We envision three possibilities: the “best practices” that are mentioned/suggested in the guidelines are not taught; the “best practices” that are on the guidelines are taught but not reported as such, or that the “best practices” that are on the guidelines are not considered relevant by instructors.

We established that there are some practices that were reported as being used in more than one primary study, which can be considered a good indicator that they are really software engineering “best practices”, since they were more considered than the others within the context of software engineering education.

The low amount of papers reporting "best practices" of software engineering being taught and the low amount of coverage that these "best practices" have, when compared with the ACM-IEEE Curriculum Guidelines, let us hypothesize that most practices used in SE education are not necessarily reported in the literature or they are not characterized as a best practice. For example: Cheng et al. [171] uses coaching and it is a software engineering best practice, but in his work it is not described as such, so it was not considered in our procedure for this systematic review. This suggests that performing a systematic research related to best practices in software engineering education even using a standard procedure is not flawless, because there are reported work that uses best practices but are not reported as expected.

3.3 Software Engineering Education in Chile

We only had informal data concerning software engineering education in Chile. Therefore, in order to understand the reality in SE education, at least in this country, we conducted a status report of local SE education. In this status report, we elicited universities teaching computing programs, their curriculum, their courses and the specific courses related to SE. Our main goal was to understand what was being taught, and how it was being taught. Since the context of a general career (that has to be taught in one program a little bit of all the 5 curriculum guidelines) are the same. To be able to evaluate and see the real picture of software engineering education in Chile we needed a study to assess the actual situation, based on real data, and not just informal communication. To be able to do that we conducted a study comparing SE education in Chile with the SE2014 Curriculum.

In this study we considered Chilean universities that have a computer related undergraduate program. We began looking for accredited universities. There were 43 universities accredited by the Chilean government, where 25 of them belong to CRUCH[1] [36], an organization that groups traditional universities. Two of them do not have computer-related programs and another four were not available to answer our questions. Therefore, we considered 19 universities and also included the information about Universidad Andrés Bello that, even though it is not part of CRUCH, it is also a highly-recognized university (see Table 3.13). This means that from the 24 universities that formed our universe, we managed to contact 20 universities, which represents 83% of the universe.

As a first step in this study, we wanted to know which type of computer-related programs each university offers: CS Engineering or CS Technology. Table 3.13 shows that all universities offer CS Engineering and only eight of them deliver CS Technology. The CS Engineering career has a duration of 11 to 12 semesters, and the students are prepared with a strong analytical background on computer science fundamentals. In CS Technology, the programs last between 9 and 10 semesters and prepares students with a focus on implementation.

The data collection was conducted between October and November, 2015. This procedure has five steps:

1. Identification of CS programs and curriculum gathering from Web pages or by contacting the programs.
2. Curriculum analysis to identify mandatory software engineering courses.

[1]Board of Presidents of Chilean Universities (in Spanish)

Table 3.13: Number of Computer-Related Programs Offered by Chilean Universities in 2015

University	**CS Engineering**	**CS Technology**
Universidad Austral de Chile	1	-
Universidad del Bío-Bío	1	1
Pontificia Universidad Católica de Valparaíso	1	1
Pontificia Universidad Católica de Chile	1	-
Universidad de Tarapacá	1	-
Universidad Andrés Bello	1	1
Universidad Católica Santísima Concepción	1	-
Universidad Católica de Temuco	1	-
Universidad Católica del Maule	1	-
Universidad Católica del Norte	1	1
Universidad de Atacama	1	-
Universidad de Chile	1	-
Universidad de Concepción	1	-
Universidad de La Frontera	1	1
Universidad de Playa Ancha	1	-
Universidad de Santiago de Chile	1	1
Universidad de Talca	1	-
Universidad de Valparaíso	1	2
Universidad Técnica Federico Santa María	1	-
Universidad Tecnológica Metropolitana	1	1

3. Identification of the course instructor by searching the Web or by contacting the program.

4. Obtaining course information from Web sites or by contacting instructors.

5. Clarification of doubts by interacting with the course instructor (when needed).

This information allowed us to characterize the SE courses included in these programs.

3.3.1 SE Course Characterization

After identifying at least one software engineering instructor per university, we contacted them by email to participate in a phone interview to talk about the SE courses at their respective universities. In these interviews, we asked some basic questions about programs and courses, e.g.:

- Are these courses (identified by the authors) the only ones related to SE in this program?
- Can you briefly describe the units included in each of these courses?
- How are the students taking these courses evaluated?
- Do these courses have student projects? In the case of a positive answer, what is the project size? What relevance does the project score have in the final grade?
- What is the students' perception about the software engineering courses they have to take?

When the course contents were not available online, we asked them to provide us the material. Considering the existence or not of a project as part the course, as well as the project size and impact of the project score in the final grades, we defined four course categories:

- **Theoretical** - Courses that mainly deliver theoretical knowledge through lectures, and students are usually evaluated using exams, readings and homework.
- **Theoretical-Practical** - Courses with an important theoretical component that also include small projects or case studies whose grade has some impact on the course final grade (less than 50%).
- **Practical-Theoretical** - Courses focused on projects that also include theoretical lectures that support project work. Typically, the project grade represents more than 50% of the final grade.
- **Practical** - Courses that focus almost exclusively on student projects.

3.3.2 Computer Science Courses

Analyzing program curricula we found that CS Engineering has an average of 3.4 SE related courses and CS Technology has 2.8 (Table 3.14) SE related courses being taught. Provided that CS Engineering programs include (in average) 46 courses, SE courses represent only 7.4% of the total. In the case of CS Technology programs, the average number of courses is 30, therefore the SE courses represent for 9.3% of the total.

We then classified the courses of each program according to the characterization presented in Section 3.3.1. Figure 3.1a shows the course characterization in CS Engineering programs, where we can see that Theoretical-Practical courses account for more than half of the courses (51%), and Practical-Theoretical courses account for 25%. Only 15% of the courses are Practical and 9% are Theoretical. Figure 3.1b shows the same characterization for CS Technology programs. Here Theoretical-Practical courses have an even larger relevance (61%) and there are no Theoretical courses. These numbers show the high relevance that practice has in the instructional approaches used to teach software engineering in Chile. Although these approaches are fine, they are usually focused more on operational aspects; e.g. database creation, requirements; of software development than on concepts; e.g. teamwork coordination, client negotiation, prioritization, risk analysis; behind the software engineering in Chile. The first one is short-term knowledge and the second one represents mid-term or long term knowledge.

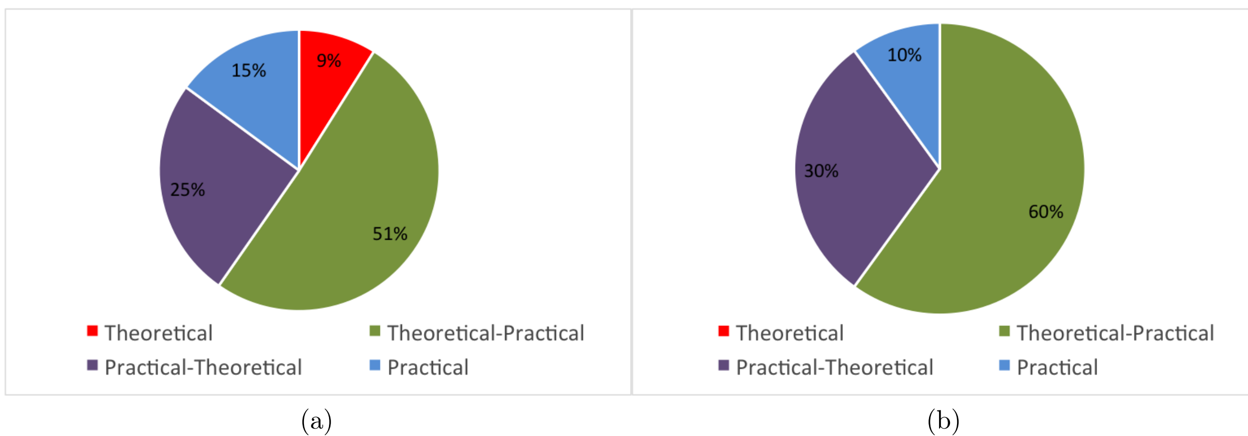
3.3.3 SE Knowledge Areas Coverage

After reviewing the knowledge areas of the ACM/IEEE SE 2014 Curriculum and their associated units (Table 3.2) that are being taught as part of CS programs in Chile, we built a list of units indicating the number of times each one was taught as part of a mandatory SE course.

We calculated the frequency with a unit that was taught by taking the average number of courses that cover that unit in each program. Tables 3.15 and 3.16 show the SE units most and least frequently taught respectively in the CS Engineering programs. Similarly, Tables 3.17 and 3.18 present the results for CS Technology programs.

Table 3.14: Number of Software Engineering Courses in Each Program in 2015

University	CS Engineering	CS Technology
Universidad Austral de Chile	4	-
Universidad del Bío-Bío	2	3
Pontificia Universidad Católica de Valparaíso	4	2
Pontificia Universidad Católica de Chile	2	-
Universidad de Tarapacá	3	-
Universidad Andrés Bello	2	3
Universidad Católica Santísima Concepción	3	-
Universidad Católica de Temuco	3	-
Universidad Católica del Maule	6	-
Universidad Católica del Norte	5	3
Universidad de Atacama	3	-
Universidad de Chile	3	-
Universidad de Concepción	2	-
Universidad de La Frontera	4	2
Universidad de Playa Ancha	3	-
Universidad de Santiago de Chile	5	4
Universidad de Talca	3	-
Universidad de Valparaíso	3	3
Universidad Técnica Federico Santa María	4	-
Universidad Tecnológica Metropolitana	3	2
Average	3.4	2.8
Median	3	3



(a) (b)

Figure 3.1: Course Characterization in (a) CS Engineering Programs and (b) CS Technology Programs

Table 3.15: Most Frequent Knowledge Units in CS Engineering Programs

Units	Courses
Design Concepts	2.45
Design Strategies	2.40
Analysis Fundamentals	2.20
Types of Models	2.15
Requirements Fundamentals	1.95
Architectural Design	1.75
V&V Terminology and Foundations	1.75
Eliciting Requirements	1.70
Project Planning and Tracking	1.55
Requirements Specification & Documentation	1.50
Human Computer Interaction Design	1.50
Process Concepts	1.50

Table 3.16: Least Frequent Knowledge Units in CS Engineering Programs

Units	Courses
Communication Skills	0.90
Computer and Network Security	0.60
Problem Analysis and Reporting	0.60
Developing Secure Software	0.50
Evolution Process and Activities	0.50
Software Configuration Management	0.40
Product Assurance	0.30
Process Assurance	0.25
Modeling Foundations	0.05
Professionalism	0.05

We then aggregated the number of units considering their corresponding knowledge areas, taking into account the courses characterization. Figures 3.2 and 3.3 show that *Software Design* and *Requirements Analysis and Specification* are the most emphasized areas. The other knowledge areas, except *Security* and *Software Quality*, also have a relevant presence in the software engineering education.

3.3.4 Courses Evaluations

Another point of interest of this study was to determine how students are evaluated in SE courses. The review of course content allowed us to identify six types of evaluations: exams, readings, homework, case-studies, projects and peer-assessments. We found certain relationships between course characterization and evaluations. For instance, there were no exams in practical courses, and there were peer-assessment only in practical and practical-theoretical courses, as expected. Figure 3.4 shows the course evaluations for the CS Engineering and in Figure 3.5 for the CS Technology programs.

In both cases, projects and exams are the most frequent instruments of evaluation; however, homework have its significancy for CS Engineering and not for CS Technology. CS Engineering theoretical courses are evaluated mainly with exams and homework, and also with readings and case-studies to a lesser extent.

Table 3.17: Most Frequent Knowledge Units in CS Technology Programs

Units	Courses
Types of Models	2.25
Analysis Fundamentals	2.20
Design Concepts	2.25
Design Strategies	2.13
Requirements Fundamentals	2.00
Requirements Specification & Documentation	1.75
Eliciting Requirements	1.63
Architectural Design	1.50
Human Computer Interaction Design	1.50
V&V Terminology and Foundations	1.50

Table 3.18: Least Frequent Knowledge Units in CS Technology Programs

Units	Courses
Reviews and Static Analysis	0.87
Testing	0.87
Software Quality Concepts and Culture	0.62
Software Quality Process	0.62
Developing Secure Software	0.50
Software Configuration Management	0.50
Problem Analysis and Reporting	0.37
Process Evolution and Activities	0.37
Product Assurance	0.25
Process Assurance	0.12
Professionalism	-
Modeling Foundations	-
Evolution Activities	-

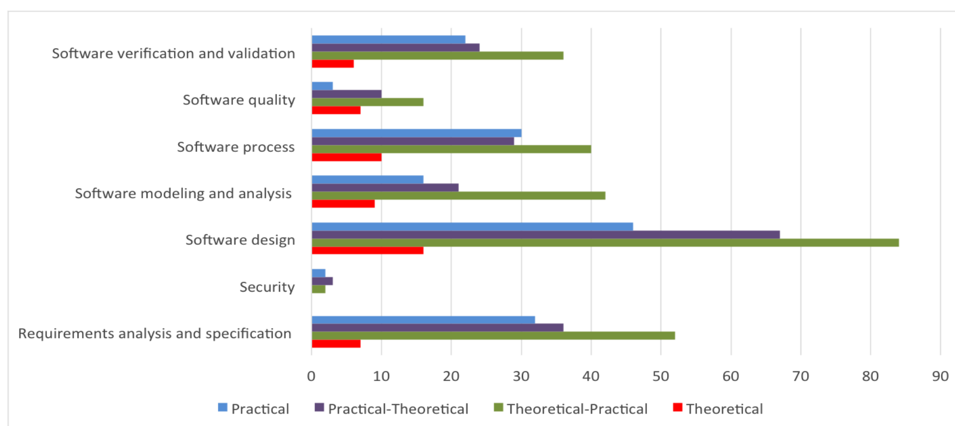


Figure 3.2: Knowledge Areas Characterization of CS Engineering Programs

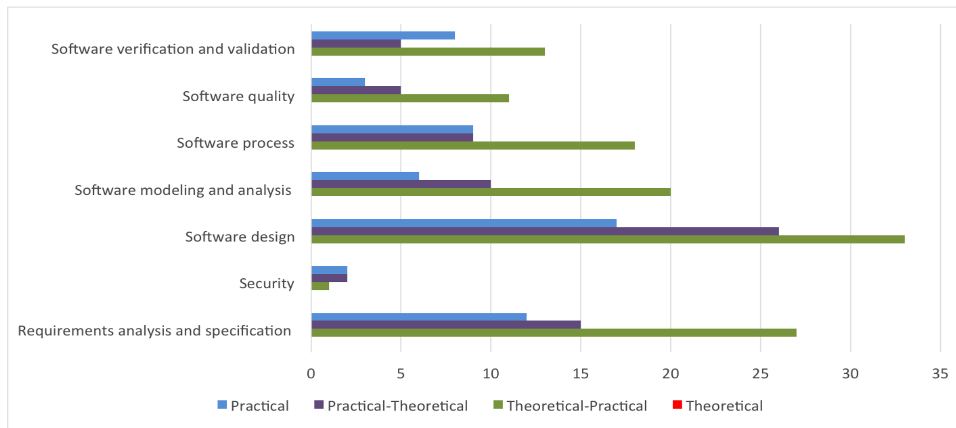


Figure 3.3: Knowledge Areas Characterization of CS Technology Programs

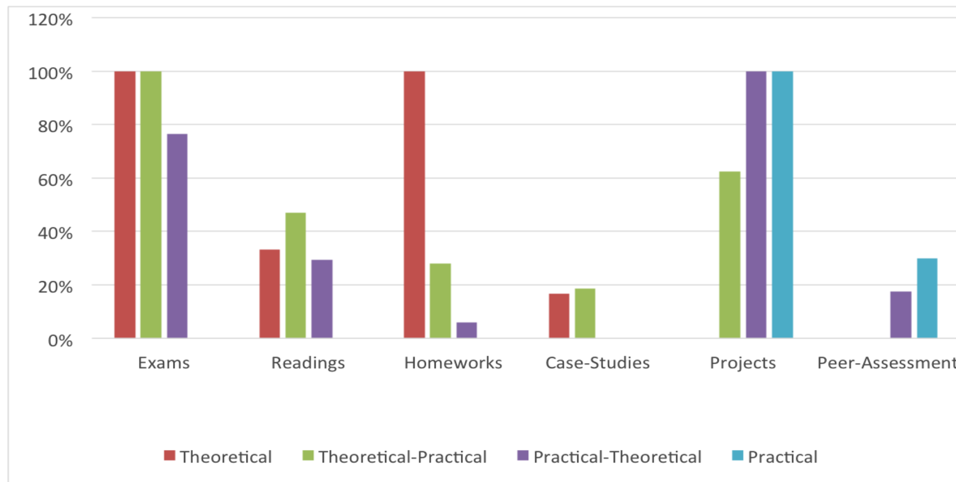


Figure 3.4: Courses Evaluation of CS Engineering Programs

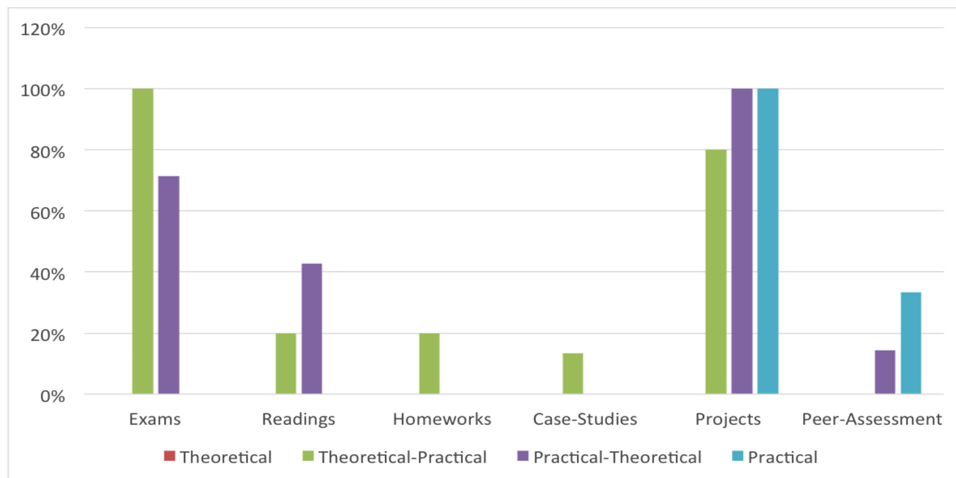


Figure 3.5: Courses Evaluation of CS Technology Programs

The courses that are purely practical are the capstones courses, and these are normally the last course of the program, where the main goal is to put students to work together in a team solving a real problem for a real client or for someone emulating a client. The idea is

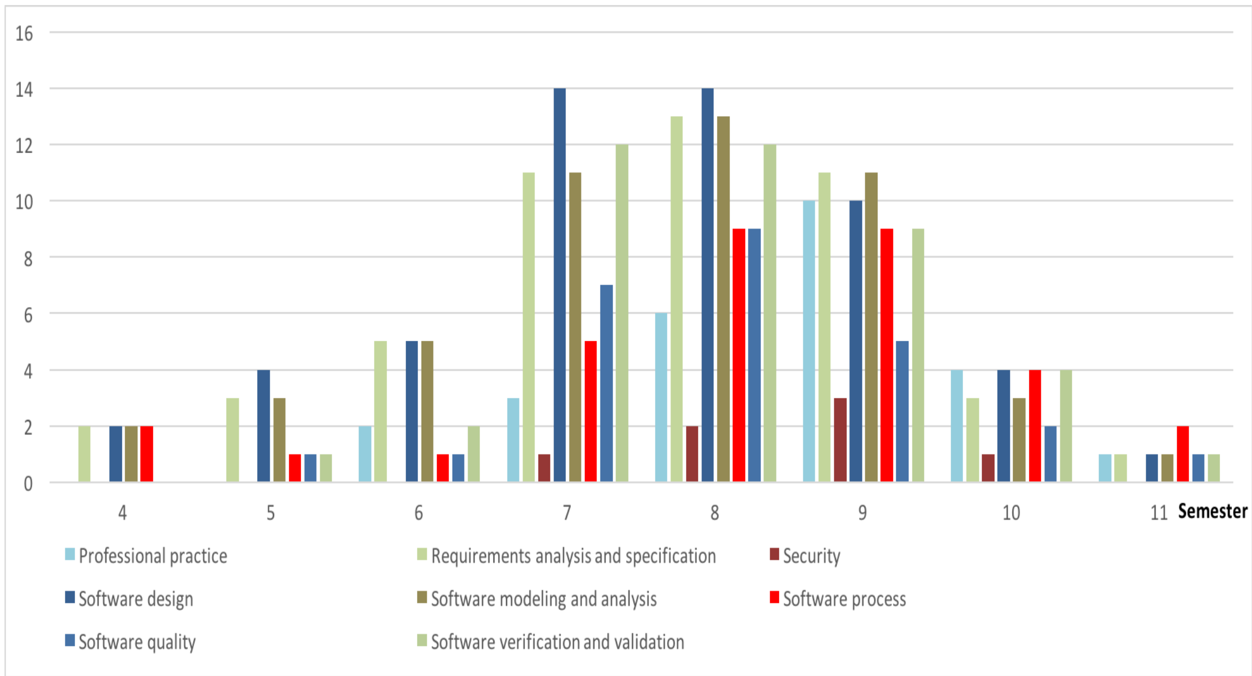


Figure 3.6: Course Temporality in CS Engineering Programs

to show students the reality of the software engineering industry in a guided way. Capstone courses usually follow agile processes, since their students are more committed with their work (scheduled time workload), and they already have some experience working in teams and developing software from prior SE courses. The courses that have a practical-theoretical approach are normally project-based SE courses, where students have to build software to solve a problem for an internal client from within the university or a problem defined by the instructor. However, it is one of the first project-based courses where students have to face working in teams, and they normally do not have experience with that, so they use a disciplined approach in these courses.

3.3.5 SE Course Temporality

In order to determine when the students are introduced to the different knowledge areas, we counted the number of times that units of each knowledge area were mentioned as being addressed in a semester for the considered programs. In Figure 3.6 we show when the knowledge of each area is introduced for CS Engineering programs, while Figure 3.7 presents the data for CS Technology programs. These results indicate that the latter introduces software engineering knowledge earlier than the former; this may be due to the more technical orientation of CS Technology and also because the complete program is shorter.

3.3.6 Analysis of SE Teaching in Chile

We identified the number of SE courses in both types of programs and which knowledge areas and units are covered by them. Both types of programs include around three SE courses that cover most of the knowledge areas included in the ACM/IEEE Software Engineering 2014 Curriculum Guidelines.

The study results also show that most courses follow a practical or practical-theoretical approach in both kinds of programs; however, CS Technology programs do not include any

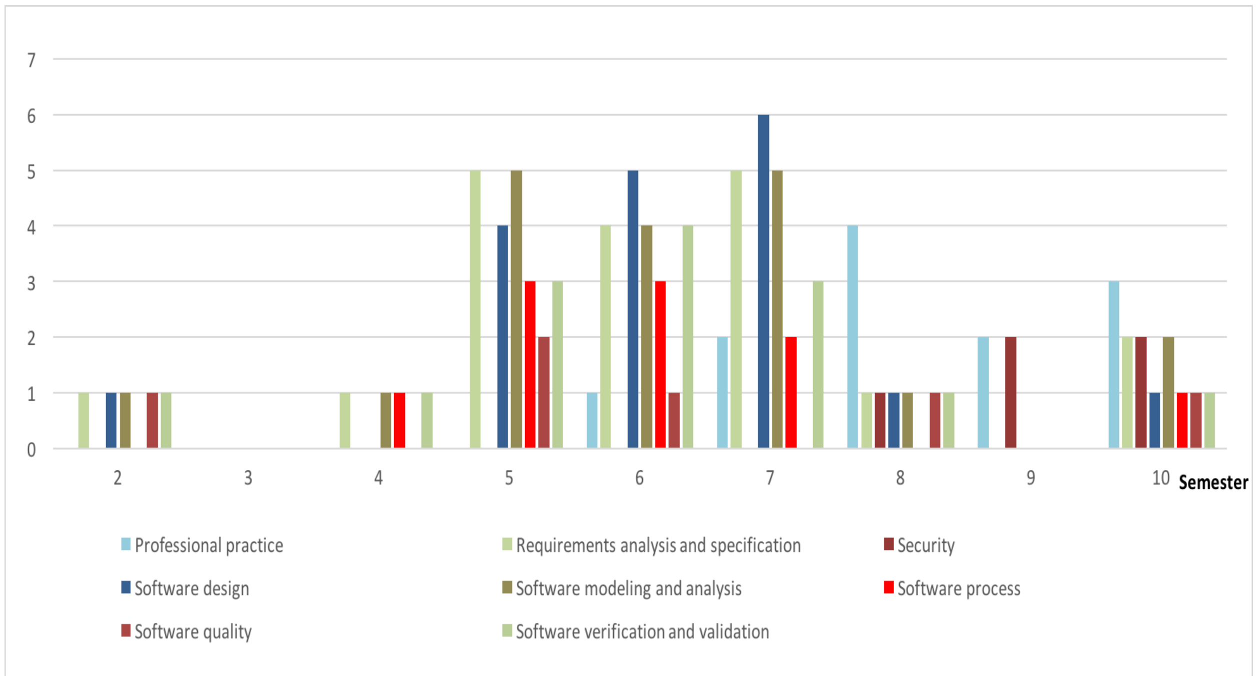


Figure 3.7: Course Temporality in CS Technology Programs

purely theoretical courses at all, which could indicate that the knowledge students obtain is the minimum required to run software projects and not to be life-long learners. Exams and projects are the most frequent types of evaluations for software engineering courses, but homework also plays a relevant role in CS Engineering.

We also found that both kinds of program teach only a few software engineering courses during the first half of the program. Most courses are concentrated in the middle. As CS Technology is a shorter program, software engineering courses start in the fifth semester while in CS Engineering in the seventh. Even though we believe that students should learn software engineering earlier in their programs, they need to count on knowledge provided by other courses such as programming or databases before software engineering can be useful and meaningful for them.

The instructional practices used in SE courses in Chile normally go hand-in-hand with the software process. The capstone courses (purely practical) uses agile processes, so they usually have a coach helping them and they have to perform self-assessments during project development. The practical-theoretical courses, more specifically the project-based courses uses a disciplined software process, so they normally have to use some tools like e-portfolios to manage their project, peer-assessment, and reviews. Finally, the courses that have a theoretical-practical or theoretical approach follow other instructional approaches such as case-studies, simulated scenarios and the traditional approach.

The results suggest that the number of software engineering courses is quite low. However, we will need to evaluate if this is enough for students to perform satisfactorily in industry once graduated. Analyzing professional performance is part of our current work; we will use this information as a feedback for curriculum design taking into account the starting point reported in this work.

Table 3.19: Mandatory Practices

Knowledge Area	Best Practices
Software Modeling and Analysis	Basic problem analysis (problem analysis, use cases, data flow diagrams, business process).
Requirements Analysis and Specification	Requirements definition, requirements elicitation, requirements specification, requirements categorization (functional, quality, etc.) and requirements documentation.
Software Design	Design definition, definition of design and requirements interaction, definition of design principles, design strategies (function-oriented, object-oriented, data-structured, aspect-oriented).
Software Verification and Validation	Code peer reviews, unit testing, and test-driven development.
Software Process	Project planning and tracking.
Software Quality	-
Security	-

3.4 Practices for SE Education

After analyzing the SE2014 (ACM/IEEE SE Curriculum Guidelines), the SE best practices reported as being used in a systematic review, revising the SE2014 adherence on Chilean universities and in light of what we discussed in the Related Work (Chapter 2), we classified the best practices reported in the 2014 ACM/IEEE SE Curriculum Guidelines and then defined them as mandatory (they must be taught), recommended (it is advised to be taught) and optional (to be taught if there is time). Our categorization of best practices is based on their reported use found in published research papers, and reported as being used in Chilean universities.

Table 3.19 shows the best practices that we consider as mandatory. Table 3.20 presents the recommended best practices, and in Table 3.21 are the best practices that are optional. All the tables presenting the best practices here are summarized by knowledge areas, according to the SE2014 Curriculum Guidelines. It is worth mentioning that this categorization does not reflect our opinion about which practices are more important than others, we just categorize the practices according to the data collected.

Regarding the instructional practices, as reported by the literature and as they were found out in our status report in Chile, there is a difference in the software process used in project-based courses and capstone courses. Although both types of courses use a hands-on approach, capstone courses usually follow an agile software process and project-base courses follow a disciplined software process. Consequently, the instructional approaches utilized by instructors and teaching assistants are directly related to the chosen software process.

In agile software processes the instructional process usually involves coaching and tightly-coupled work. In the case of disciplined software processes loosely-coupled work is assumed, but there are few to no instructional approaches reported in the literature, particularly for this type of process. However, there are some transversal techniques, e.g., peer-assessment and the use of e-portfolios, which could be used in both scenarios.

Table 3.20: Recommended Practices

Knowledge Area	Best Practices
Software Modeling and Analysis	Modeling elicitation (entity-relationship, state diagrams, business process, workflows, etc.).
Requirements Analysis and Specification	Prototype validation, inspection, and requirements revision.
Software Design	Elicitation of architectural design, elicitation of hardware architecture, prototype navigation, visual design definition, design notation definition, database design definition computer interaction design, detailed design, and elicitation of metrics to design evaluation.
Software Verification and Validation	Verification and validation planning, code inspections, validation through requirements verification, and integration tests.
Software Process	Software process definition, software process usage, software configuration management, refactoring, and software maintenance documentation.
Software Quality	Quality requirements attributes, and software quality standard definition.
Security	-

Table 3.21: Optional Practices

Knowledge Area	Best Practices
Software Modeling and Analysis	Modeling principles definition (decomposition, generalization, etc.).
Requirements Analysis and Specification	-
Software Design	-
Software Verification and Validation	Debugging techniques usage, problem resolution and protocol.
Software Process	-
Software Quality	Process assurance, product assurance.
Security	Security definitions, network security threats definition, and cryptographic protocols selection.

Summarizing, ACM and IEEE developed with the help of many SE instructors and experts from industry a SE curriculum guideline to help universities to teach what really matters in SE. The guidelines were first published in 2004 and in 2014 it were updated. The ACM/IEEE curriculum guidelines are considered best practices to be taught to SE students. But sometimes is not possible to create a SE program. And SE have to be taught in a more general program, in that case what SE best practices should be taught? To answer this question we performed a systematic literature review of the best practices being taught to SE students. The results show a set of practices that are reported as being taught but the amount of primary studies selected was a lot less then expected. In this systematic literature review we did not find any work from Chile, so we performed a study to evaluate the SE education in Chile. In this study we reported that there are an average 3 courses dedicated to SE, and that SE is being taught in a practical fashion (with projects). And we ended proposing a list of mandatory, recommended and optional SE best practices to be taught to students in a more general computer related program.

Chapter 4

EduProcess

This chapter presents an overview of the proposed prescriptive software process: the EduProcess. The individual phases, activities, artifacts and milestones are presented in detail in the Appendices. Appendix A presents the student track of EduProcess, Appendix B presents the instructor track and Appendix C presents the monitoring process.

4.1 General Approach

EduProcess is a disciplined software development process intended to help guide the systematic development of information systems in project-based courses, considering the critical needs of the project stakeholders. This process involves four phases (see Figure 4.1): Conception, Iteration 1, Iteration 2 and Deployment. It is important to remark that the phases are sequential, except for iterations 1 and 2, where there is an overlapping (approximately 50%).

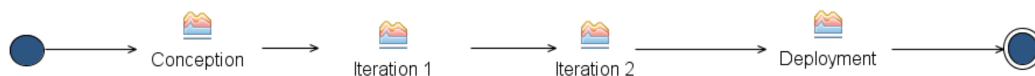


Figure 4.1: General Architecture of EduProcess

During the first phase, the problem to be addressed and the project context are elicited and understood by student software teams, and a first idea of the solution is represented through a simple prototype, which is validated with the stakeholders. The idea is to agree on a common goal among all involved people, in order to establish the initial direction of the project. During the next two phases, the solution is developed in an incremental way, with clients and users participating in weekly meetings. Finally, in the last phase (deployment) the solution is tuned and put into production.

This process can be used to support software projects of at least seven weeks of duration, considering student teams that work in a loosely-coupled way. However, given the process structure, it is recommended that projects lasts at least 10 weeks.

The process shown in Figure 4.1 involves two parallel tracks: one used to guide student work (and also includes other stakeholders) and the other that is used to guide the teaching team (see Figure 4.2). In the first track, students assume roles as part of the development team, and the activities of that track are focused on obtaining a software product that helps clients and users deal with the problem being addressed. The second track is used to help students put into practice the SE theoretical knowledge and carry out the different aspects involved in a software project. As mentioned in Chapter 1, both tracks are intertwined although they pursue different purposes.

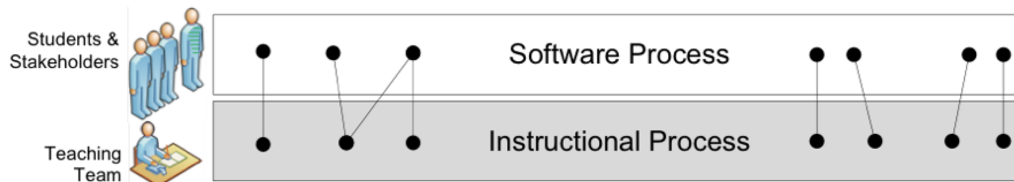


Figure 4.2: EduProcess Tracks

In order to help the reader understand the rationale behind the design of the EduProcess, the next subsections briefly explain the type of products to be developed using this process, and also the roles played by participants.

4.1.1 Target Software Product

EduProcess is focused on supporting students during the development of information systems. We have chosen this type of product since they are easy to conceive and represent most of the development projects that are currently being done by the software industry at a global level. These products are initially structured through a three-tier architecture (see Figure 4.3), where the upper layer corresponds to the user interface, the lower layer corresponds to the data, and the middle layer is the glue (typically related to the business logic). This structure helps students understand separation of concerns, and organize the conception and development of the final product.

4.1.2 Supported Roles

EduProcess was conceived to support small software teams (five to seven students), in where the students work in a loosely-coupled way; i.e., they work autonomously most of the time and have sporadic instances (at least once a week) for synchronizing their activities and integrating their work. These teams need to use a defined prescriptive process, since there is

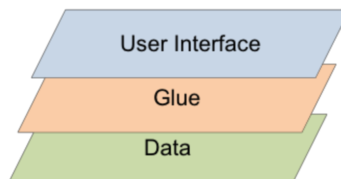


Figure 4.3: Architecture Assumed for an Information System

no time directly assigned to students for face-to-face interaction, and their other courses and personal lives make almost impossible to perform tightly-coupled work.

Team members can play one of the following roles: *project manager, requirement analyst, designer-developer* or *quality assurance engineer*. Each team defines its own structure and assigns the human resources depending on the work to be addressed; however, there should be at least one person playing each one of the previously mentioned roles. Details of the roles activities and characteristics are presented in Appendix A.

There are also *clients* and *users* that represent the other stakeholders. One person can play both roles, but it is better if they are played by different people. Typically, clients help the team establish development priorities of the project. The agreement establishing the project scope and development priorities is signed by the client and the development team and delivered to the instructor. This agreement represents the project contract, which can be modified with a settlement between both parts.

Concerning the instructional team, we can identify three roles: *instructor, teaching assistant*, and *monitor*. The first two roles are mainly focused on evaluating the development process conducted by the students and also their results; more details about these roles can be found in Appendix B. The last role (monitor) is neither an evaluator nor a coach, but an agent that promotes and facilitates self-reflection among team members, and uses this as a mechanism to help them find solutions by their own, for their individual and team problems (like a psychologist). More details about these roles are available in Appendix C.

4.2 Structure of EduProcess

Figure 4.4 describes the four phases of EduProcess (i.e., Conception, Iterations 1 and 2, and Deployment). Based on that, the figure summarizes the major software development activities that should be addressed by students, deliverables (work products), reviewing activities that allow clients to validate work products, and milestones related to the development of the software product. The activities highlighted in orange are part of the software process, those in light blue belong to the instructional process, and those in green are part of both processes.

EduProcess also indicates the instructional activities performed by the instructor, teaching assistants and monitors, where most of them are intertwined with the software process activities. Particularly, the product reviews, the monitoring of the development process, and the work sessions performed jointly between the software team and its clients and users. The teaching team is in charge of product reviews (checkpoints) and sometimes these activities also involve the clients and users; in particularly, these people participate in the last review of each phase. The next subsections explain in more detail each phase of the EduProcess; i.e., the software development process and the instructional process.

ACTIVITIES / ITEMS	CONCEPTION (CO)	CO / R	ITERATION 1 AND 2						IM / R	DEPLOYMENT (DP)	DP / R
			Analysis (AN)	AN / R	Design (DE)	DE / R	Implementation (IM)				
MAJOR SOFTWARE DEVELOPMENT ACTIVITIES (conducted by the students)	1. Identification of the problem to address, project goal and size. 2. Identification of processes to support, actors, and main user requirements. 3. Development of a rapid prototype.		1. Specification of the user and software requirements. 2. Determination of the operational environment. 3. Test case sPecification.		1. Definition of the processes to be supported. 2. Definition of the basic software architecture. 3. Software interfaces and components design. 4. Data support design.		1. Coding. 2. Testing (unit, integration and system). 3. User tests.			1. Fine tuning of the software. 2. Solution deployment . 3. Final acceptance tests.	
DELIVERABLES OF THE SOFTWARE PROCESS (arrows implies under change control)	Presentation of the Software Conception (SCo) Software Prototype (SoP) SCo → SoP →		Requirements Specification Document (RSD) Software Prototype (SoP) Test Cases (TeC) RSD → SoP → TeC →		Software Prototype (SoP) Design Document (DeD) SoP → DeD →		Software Product (SPr) SPr →			Adjusted Software Product (ASPr) Source Code Project Historical Document (PHD) ASPr → Code → PHD →	
TEACHING ACTIVITIES (teaching activities done by the instructor)	1. Definition of development teams and project assignments. 2.Examples of Activities Presentation / Products to Address During Project.		1. Examples of Activities Presentation / Products to Address During the Project.		1. Examples of Activities Presentation / Products to Address During the Project.		1. Examples of Activities Presentation / Products to Address During the Project.			1 .Examples of Activities Presentation / Products to Address During the Project.	
REVIEWING/MONITORING ACTIVITIES	1. Reflexive Weekly Monitoring	FTR/ LSR	1. Reflexive Weekly Monitoring	FTR	1. Reflexive Weekly Monitoring	FTR	1. Reflexive Weekly Monitoring		LSR	1. Reflexive Weekly Monitoring	LSR
WORK SESSIONS W/CLIENTS & USERS (joint work between the team and clients/users)	1. Weekly Meeting with Clients/Users		1. Weekly Meeting with Clients/Users		1. Weekly Meeting with Clients/Users		1. Weekly Meeting with Clients/Users			1. Weekly Meeting with Clients/Users	
SUPPORTING ACTIVITIES (activities Carried out by teaching assistants)	1. Teaching Operative Issues 2. Informal Review of Software Artifacts		1. Teaching Operative Issues 2. Informal Review of Software Artifacts		1. Teaching Operative Issues 2. Informal Review of Software Artifacts		1. Teaching Operat. Issues 2. Informal Review of Software Product			1. Informal Review of the Software Product	
DELIVERABLES OF THE INTRUCT.-SOFTW. PROCESS	Monitoring Record (MoR) Peer-Assesment (PeA) MoR → PeA →		Monitoring Record (MoR) MoR →		Monitoring Record (MoR) MoR →		Monitoring Record (MoR) Peer-Assesment (PeA) MoR → PeA →			Monitoring Record (MoR) MoR →	
MILESTONES (projects checkpoints)		▲ SCo Approved		▲ RSD Approved		▲ DeD Approved			▲ SPr Approved Provisional Acceptance		▲ ASPr Approved Final Acceptance

Activities and work products related to the ***software process***
Activities and work products related to the ***instructional process***
Activities and work products related to both ***processes***

FTR: Formal Technical Review
LSR: Live Software Review

Figure 4.4: Structure of EduProcess

4.2.1 A Brief Description of the Software Process

As shown in Figure 4.4, each phase of the process ends with an assessment (or reviewing activity) that is usually conducted through a Formal Technical Review (FTR) or a Live Software Review (LSR). Complimentarily, Figure 4.5 indicates that EduProcess implements a sequence among Conception, Iterations and Deployment, but there is some overlap between the two iterations. This strategy helps people playing the stated roles keep engaged with the project and the team, since they have some work to do at all times.

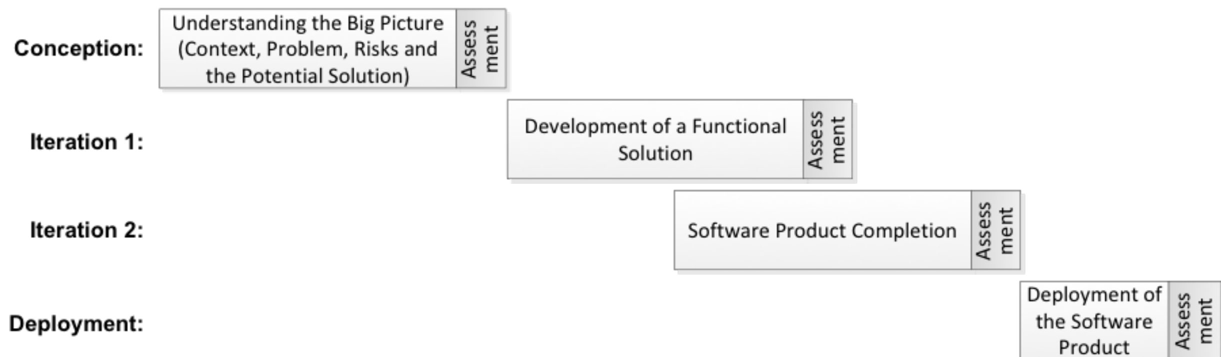


Figure 4.5: Structure of the Software Process

4.2.1.1 Conception

The main objective of the conception phase is to understand the client's problem, its scope and the context in which the problem appears. In order to address it, each team has to identify the main user requirements of their own client, project risks and make sure that the client's problem is well understood. Based on that, each development team imagines the software product they think would help clients and users to deal with their problem.

The first deliverable of the development team during this phase is an informal presentation where they explain to the rest of the students, the instructional team and the stakeholders, the problem that they have to address, the organizational and business context around the problem, and the actors that should be involved in the solution (mainly, the types of users to be supported). This preliminary evaluation instance pushes the students to address the project in an early stage, and it also makes the students' knowledge about the problem visible.

At the end of this phase, the team should deliver two products: (1) a refined and extended version of the previous presentation (Presentation of the Software Conception - SCo), and (2) a software prototype (SoP) that does not need to be functional, but that must include the main user interfaces, a draft of the workflow that the software will support and the data model to be used (see Figure 4.6).

The evaluation of these products helps the instructional team to determine if the development teams and clients' goals and expectations are aligned. Moreover, it intends to determine if the clients' problem can be mitigated with the software product envisioned by the team. The product prototype also helps the team to better understand the clients' problem and manage the expectation of all stakeholders. In the Conception phase, the students are not assigned to roles, all the students work together on understanding the clients' problem and conceiving a possible solution.

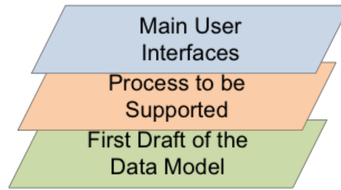


Figure 4.6: Structure of the Software Prototype

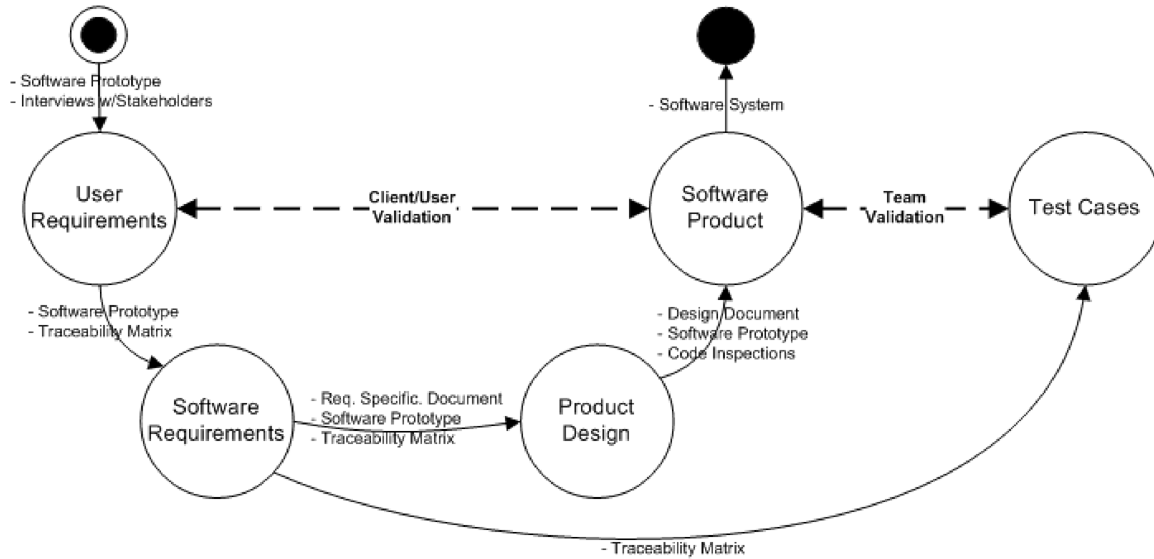


Figure 4.7: Relationship Among the Work Products of Iterations 1 and 2.

4.2.1.2 Iterations 1 and 2

Iterations 1 and 2 are very similar, so we will present them together in this section. We will discuss the points where they have some differences in the section. The purpose of this phase is to use the prototype created during the Conception phase as an instrument that helps determine the user and software requirements, the product design and also to begin the implementation of the services provided to the end-users. At the beginning of this phase, all students should assume a specific role (i.e., project manager, analyst, designer-developer or quality assurance engineer). The role configuration is proposed by each team to the instructor, who can accept, request modifications or reject their proposal. Once the team structure is agreed, each member must perform particular activities according to the role that he/she is playing. The team can change its role configuration only at the end of a phase.

The work to be done in this phase consists on performing the following major activities: analysis, design, implementation (that includes testing), and the corresponding review of the work products obtained in these activities. The relationships among the work products are shown in Figure 4.7.

The process starts identifying the user requirements (UR), using the software prototype obtained in the conception phase, as well as interviews with the stakeholders as input information. These requirements are then translated into software requirements (SR). This translation is validated by the team using a UR/SR traceability matrix, and with the stakeholders using an improved version of the software prototype (SoP).

The Requirements Specification Document (RSD) is obtained as a result of the first two activities. This document has to be presented at Formal Technical Review (FTR) to the rest of the students and the instructional team. In this instance (indicated as AN/R in Figure 4.4), the main user and software requirements are validated by the instructional team and the rest of the review attendees, including the quality assurance engineer who has to report not only his/her analysis of the Requirements Specification Document, but also the comments and observations raised in the FTR.

Using as input the RSD, the result of the FTR and also the software prototype, the product design is created and specified in the Design Document (DeD). This document must present the process that the software will support, the environment where the software will be inserted into the design of its components, the navigability, the data design (database) and the software interfaces (evolved prototype). The responsibility of the DeD falls on the designer/developer role, who is responsible for presenting it during the design Formal Technical Review (FTR) session.

The quality assurance engineer has to analyze the DeD to guarantee that the design meets the user and software requirements. He/she has also the responsibility of creating test cases for all software requirements.

The last activity of this phase is the implementation of the software product, which must take into consideration all the previous work products: RSD, DeD, SoP and all the observations received during FTR sessions. The designer/developer team has to code the software, perform unit, system and integration tests of the code done (code inspections). After the code is written, the quality assurance engineer has to validate all the test cases, and after the software pass the tests, the software faces the client/user validation. Each Iteration finishes with a Software Product (SPr) that has to be presented at a Live Software Review (LSR).

4.2.1.3 Deployment

This is the last phase of the EduProcess, and it could also be called the handover phase. During this phase, the students must install the software in the previously defined operational environment and demonstrate to the client/users all its capabilities. The software has to be completely validated by the user, installed at the client's facilities, configured, and eventually the data of legacy systems should be migrated to the new application. In summary, the team has to deliver a complete running software solution.

The most important activities that must be performed during this stage include the final adjustments to the implemented system, the deployment of the software solution, and conduct the final acceptance tests with users and clients. Almost all the roles contribute to this phase. The quality assurance engineer has to completely validate the product and perform the tests with the client and users. The developers/designers have to deploy the software (ASPr) into its running environment and put it into production. The project manager must deliver the Source Code of the application and the Project Historical Document (PHD) to the client. The level of approval of the ASPr by the client determines the level of success of the project; it is a binary evaluation.

4.2.1.4 Work Dynamics

It is important to remark that each development team has assigned one hour per week to interact face-to-face with their stakeholders (clients and users). The interactions are conducted during a pre-established time slot that should be agreed between the team and the stakeholders (e.g., every Monday from 10:00 to 11:00). The use of this time slot is not mandatory, but any time not used is considered lost. .

It is also expected that each team has a short weekly session for checking the project status and coordinating team activities. At the beginning of the project, these coordination sessions can last 20-30 minutes and by the end they require 5-10 minutes. Typically, they are conducted immediately before or after course lectures. Although coordination using digital communication media is very welcome and used, frequent face-to-face interactions to discuss project issues helps increase the sense of team belonging and the commitment of its members.

Students who take a project-based course normally have other courses with similar time consuming demands. Therefore, the coordination effort within the team is not negligible. The role of the project manager is transversal to all phases and it is critical to the success of the project, since he/she needs to coordinate the work of team members and keep in contact with clients and users. The project manager assigns tasks to the team members, who have to address these assignments on their own time. Therefore, the teams usually work in a loosely-coupled fashion.

Once a week each team has to meet with its monitor, who gathers information about the project and team status. To this end, the monitor asks the team members simple questions, which require direct answers; for instance: "Has everybody accomplished their assigned tasks?". The monitor's questions are used to make an initial diagnosis of the project progress, team members' commitment, and perceive the quality of the teamwork. The answers to the monitor's questions must be agreed among team members. Finding a consensus pushes the students to analyze various opinions and points of view, and it helps them create a common view about the team and project current status. More detail about monitoring can be found in Appendix C.

4.2.2 The Instructional Process

The instructional process is the track of the EduProcess that supports the teaching-learning experience. The instructional process is divided into the same four phases as the software process. As shown in Figure 4.8, this process is sequential, and it includes a first phase (Preparedness), which must be conducted before Conception, and it is focused on preparing the practical experience. Then, the process considers the conception, two iterations and the project closure. Appendix B describes the instructional process in detail.

At the end of each phase, there are at least two assessment activities: (1) a Formal Technical Review (FTR) or a Software Live Review (SLR) depending on the product being evaluated, and (2) a Peer Assessment (PeA). The result of both activities has an impact on the students final grade. The first one is led by the instructional team and it is focused on evaluating the work products obtained by the team. The second one is a peer-assessment, where the students anonymously evaluate the performance of their teammates and provide feedback for helping them overcome conflicts and limitations. In this second type of assessment, we

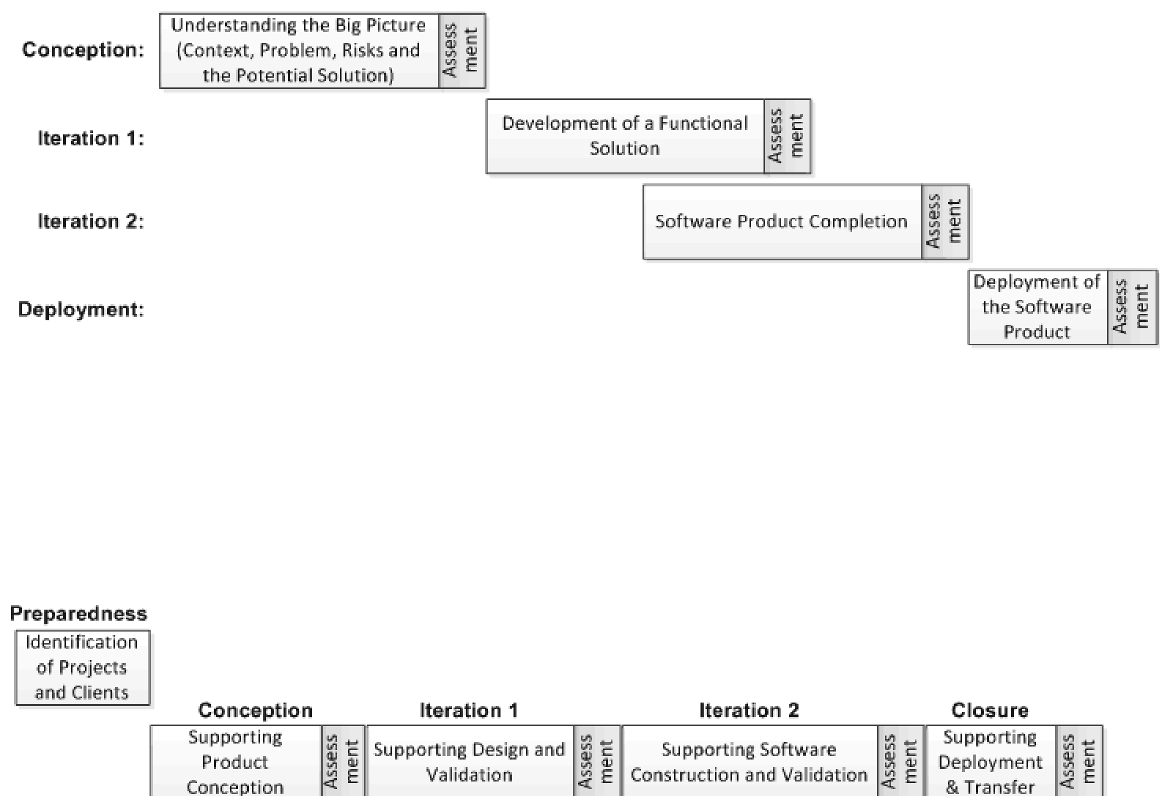


Figure 4.8: Structure of the Instructional Process

recommend using the instrument proposed by Silvestre et al. [32], which was designed to support peer assessment during project-based and capstone courses.

4.2.2.1 Preparedness

According to Silvestre et al. [32], commitment is a key variable that influences the results of the software projects. In order to generate engagement, the instructor should try to count on a pool of projects that potentially interest the students, and involve clients and users that are really attentive in the final product. Finding these projects requires time, and there is no particular time frame to execute such an activity; therefore, it is recommended that the instructor conduct a search for suitable projects at any time that he/she considers convenient. Thus, it is possible to count on a pool of suitable project/problems that need a software solution, and that can be addressed in a project-based course.

Typically, universities (including their departments) have several needs in terms of administrative services or systems, which can be addressed as projects in these courses. Developing solutions for this scenario has several advantages: (1) the clients and users are usually close to the development team, (2) the students have a better chance to know the business problem to be addressed, (3) the clients and users understand that the development is in an academic scenario, and (4) the instructor can provide a direct benefit to his/her institution.

The project should fulfill the following requirements in order to be eligible for a project-based course supported by the EduProcess:

- *Type of project.* The product to be developed should be an information system for

several reasons. Usually this type of system is not too complex and it is the most frequently developed type of system in industry. These systems can be addressed in the time available in these courses, even if the developers do not have a deep knowledge about the business domain that the problem belongs. Most students already have some experience implementing software, as part of an academic activity or for the industry; therefore, they are in some way conscious of the activities to perform and challenges to address. There is at least a common knowledge domain that can be expected of this kind of project, since they are all information systems related somehow to the university, besides the fact that, typically they have similar complexity among them. Moreover, the students feel that learning to develop this kind of systems is valuable for them.

- *Project complexity and size.* All team projects should be of a similar size and complexity, and it should be possible to develop the product in the available time. Normally, web information systems are not considered very complex projects, and their front-end and back-end are equally important to the success of the project. Moreover, addressing projects of similar size and complexity demands evaluations more fair and balanced for all the students. The instructor can also learn from past experiences and improve the course in the future.
- *Clients and users.* Each project should have a client and a user, both available at least one hour per week to interact with the students. Since students using the EduProcess are considered novice, they do not have enough expertise in developing software in teams, nor in the business domain of the user/client. Therefore, these stakeholders should be available to meet weekly with the team during the whole project, since the students will have different requirements for the client/user according to the phase of the process in which they are working on. A committed client/user can make a difference in projects when the team is not versed in the business domain.
- *Business domain.* Web information systems can be applied in several business domains, some of which can be known or understandable by the students. Considering the fixed duration of the projects, it is reasonable that the projects do not involve business domains that are difficult to understand. If so, students would spend a lot of time in getting knowledge of the business domain instead of learning to develop software in teams.

Considering the pool of projects, their eventual priority, and the number of students enrolled in the course each semester, the instructional team must decide how many and which projects to develop. Then, the students should be assigned to teams of 5 to 7 students. Silvestre et al. [32] proposed a software tool that considers the socio-technical profile of the students, in order to recommend a set of balanced and socio-technically compatible software development teams. This tool was used to perform the teams conformation.

Finally, a particular project (with the corresponding client and users) must be assigned to each team. In order to avoid conflicts among teams, we recommend using random assignments.

4.2.2.2 Conception

Provided that the students have to understand client's problem, its scope and context, during this phase the instructional team has to support them. Therefore, classes during this

phase should focus on topics like client interviews, problem evaluation, context understanding and software prototyping. Moreover, the instructional team should check the advance of the software projects and eventually support the student teams (in some way) during this first stage, since it is the most challenging of the whole process because the students must take the ownership of the project.

The instructional team evaluates the software teams twice during this phase. The first one is informal and focused on providing feedback on the project goal and scope, and the second one is formal and focused on evaluating the proposal (Presentation of the Software Conception (SCo) and the Software Prototype (SoP). The instructional team has to be concerned with giving guidelines to the students about how to perform the team work, team communication and coordination, and also highlighting the points that need improvement or attention.

As mentioned before, every week the development team must meet with the stakeholders, and also with their monitor who helps the students reflect on the activities done by the team, understand their consequences, and try to learn for future situations. These activities are transversal, i.e., they are conducted during the whole project, and they are explained in more detail in Appendix C.

4.2.2.3 Iteration 1 and 2

These two phases, Iteration 1 and 2 are very similar, so we will present them here as one and we will point it out when significant differences appear. The main role of the instructional team during this phase is to support the design and validation of the software being built. Particularly, performing exercises during classes that show the students how to perform certain activity in a practical way (e.g., how to interview a client), making an important difference in how effective students are when conducting these activities. These practical exercises act as live examples that allow students to gain confidence and know to carry out their assigned activities in a suitable way. Complimentarily, making available examples of the work products to students should produce and also helps them accomplish the activities results. In summary, the instructional team should help students put into practice the theoretical knowledge acquired in the previous software engineering courses.

In order to do that, during these phases, the instructional team has to support the students by reviewing their Requirements Specification Documents (RSD), Software Prototypes (SoP) and Test Cases (TeC), which were developed based on the software requirements. These supporting activities include identifying ambiguities in the software requirements specification, validating the project goals and scope, and evaluating if the team is able to do the amount of work agreed to with the client. Similar to the Conception phase, the weekly meetings of the development team with the stakeholders and the monitor continue until project deployment. Figure 4.9 shows the general workflow of the Instructional Process for this phase (the same for both iterations).

During the Software Design and Evaluation activities, the instructional team focuses on providing live experiences that help students to perform the development activities considered in the process. In this track, the deliverables are formally evaluated and detailed feedback is given to the students. Just like in the Requirement Analysis activity, examples of the deliverables are made available online to students (e.g., the Design Document (DeD) or the Software Product (SPr)), as this helps them address product development in a more effective

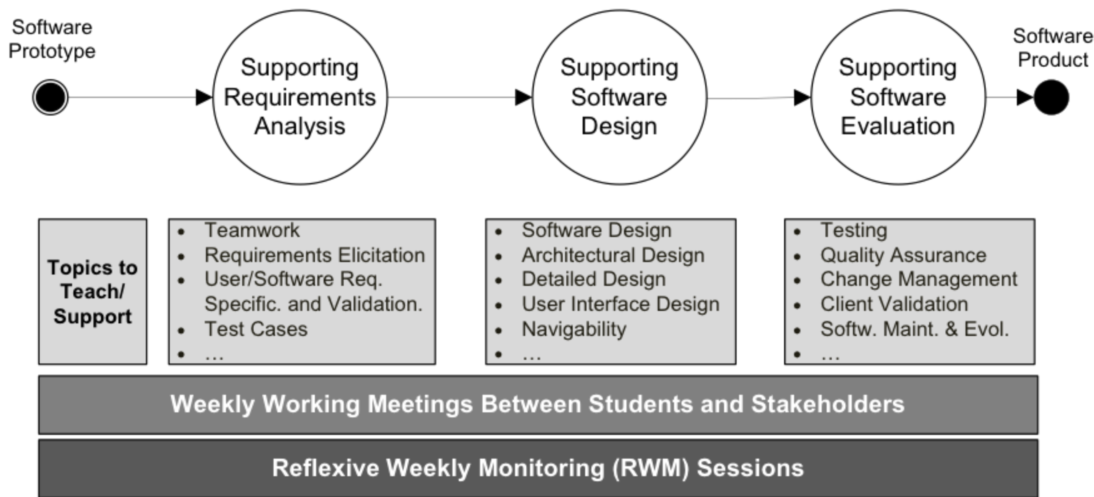


Figure 4.9: General Workflow of Iterations 1 and 2 for the Instructional Process

way.

At the end of each iteration a Software Product is obtained, which is evaluated through a Live Software Review (LSR). In the evaluation activity, all those involved participate (i.e., the development team, the instructional team and the stakeholders) and the functionality related to the main software requirements is shown in a live demo. The instructional team and the stakeholders can ask the developers to present a demo of any part of the product, in order to validate/verify the behavior of the software components. Moreover, the development team must make the software product available to the instructional team and stakeholders, who should evaluate more in depth the product after the LSR. All comments and errors detected during the LSR and the following review are recorded by the software quality engineer in a review log file. Those items must be addressed by the developers, and they will be re-checked during the next product review.

4.2.2.4 Deployment

During this last phase, the instructional team supports the students with the process of deploying the solution, training users and addressing software incidents. However, the most important part of this phase is oriented to make students reflect on the activity they just experienced, identifying lessons learned, realizing the relevance of team work and co-work with clients and users, and determining the value of the results obtained. Getting valuable knowledge for reusing in future experiences is the most important goal pursued in this phase of the instructional process.

4.2.2.5 Reflexive Weekly Monitoring

The Reflexive Weekly Monitoring activities is part of the instructional process. In these sessions, that have a duration of 30-40 minutes, the monitor helps students reflect on the past experiences and their consequences, understand the challenges and the conflicts that are usually present in software projects, and overcome (on their own) difficulties within the team, with their clients/users and also with the product being developed. In these sessions, the monitor acts as a facilitator, a counselor, who makes the people reflect and identify the solutions to their problem/challenges.

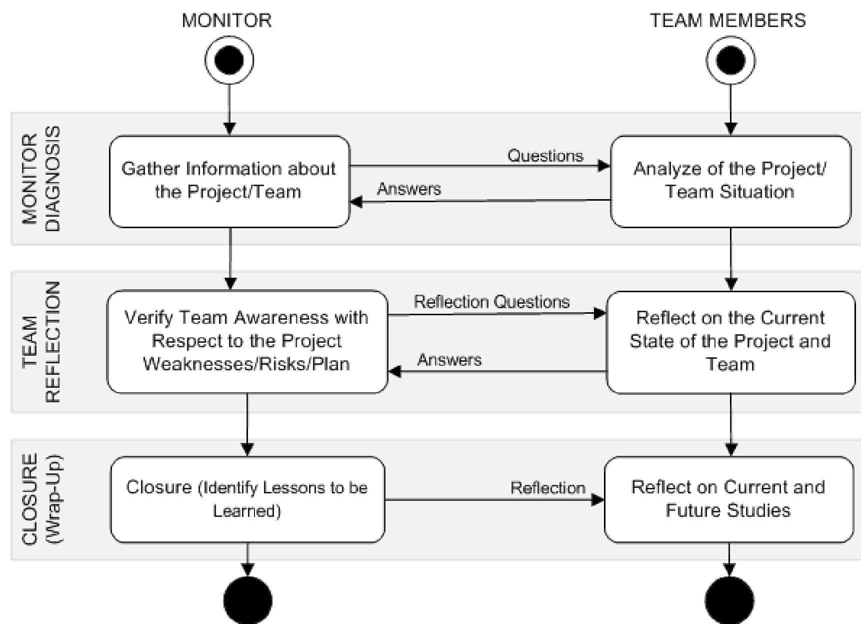


Figure 4.10: General Structure of the RWM Sessions

The monitoring sessions involve the three stages indicated in Figure 4.10: monitor diagnosis, team reflection and closure. The first one is focused on diagnosing the status of the team and the project, and also to know if the team is aware of the current situation. The second stage makes the students reflect on the issues identified in the diagnosis, and also find solutions to overcome them. Finally, during closure, the team establishes a plan to deal with these issues, and tries to envision if the current situation will allow them to meet the project goals on time.

The monitor keeps a Monitoring Record (MoR) of the sessions, which are given to the instructor only at the end of the course; this document is used as formative feedback to the instructor to change future offerings of the course. The idea is not to use these records to evaluate the students or keeping the instructor aware of the project or team situation, but to help students improve knowing that they can be honest and sincere during these sessions because recognizing their mistakes only can bring good results in the future. Appendix C explains the RWM process in detail.

4.2.2.6 Weekly Work Meetings with Clients and Users

The general goal of the weekly work meetings is to create and keep a common and updated understanding between the software team and the stakeholders (clients and users) in terms of the project goals, scope, deadlines and deliverables. Therefore, these meetings have specific purposes depending on the stage of the process on which the team is working on. Figure 4.11 shows a general workflow that characterizes the purposes of these meetings.

The first couple of meetings focuses on trying to understand the problem to be addressed and the organizational/business context in which the problem appears. Based on that understanding, the software team can define user requirements that determine the project goal and scope. Such a definition is validated with the stakeholders using a prototype (not necessarily functional) of the software product envisioned by the development team. If the result of the validation is positive, the team continues with the software requirements

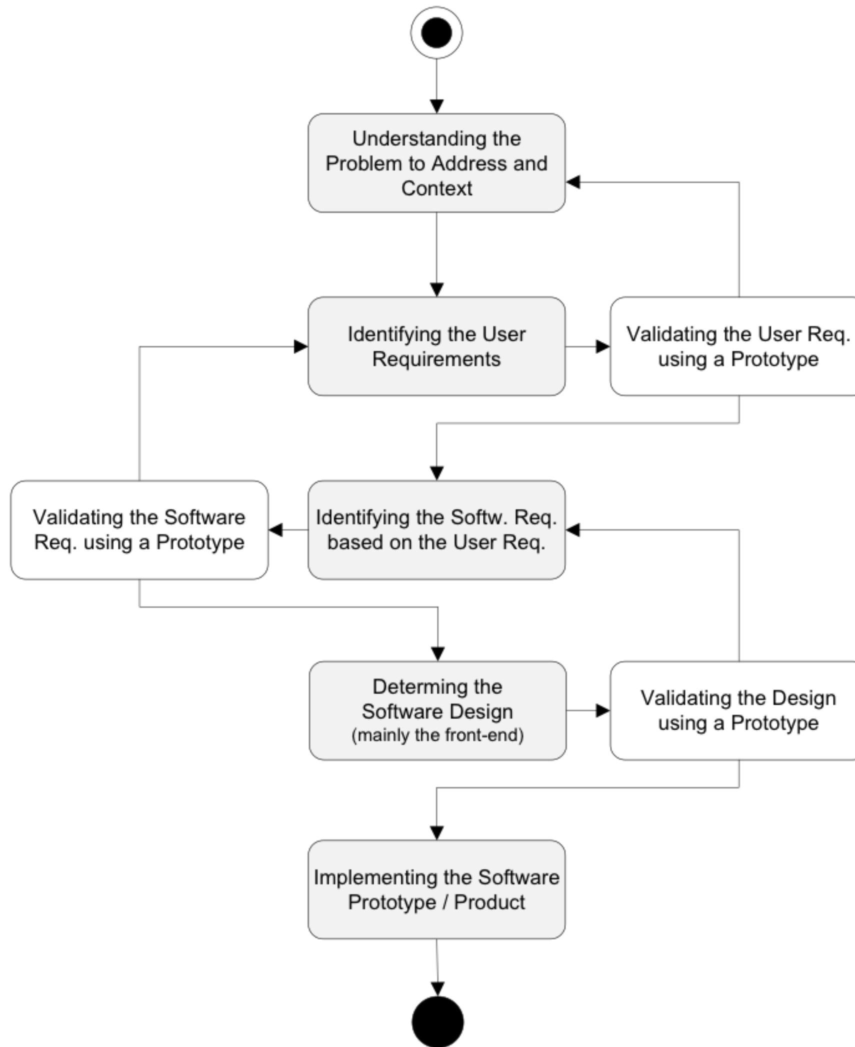


Figure 4.11: Process of Co-work between the Software Team and the Stakeholders

definition, based on the user requirements. Such a definition follows the same validation process and (in case of a positive result) allows the developers to formally start the definition of the product design. Once again, the meetings with the stakeholders are used to validate this design, which is embedded in a prototype of the product. Finally, the programmers can build the product once the design is approved, envisioning that the final system will meet with the users and clients expectations.

4.3 SE Best Practices Included in EduProcess

Table 4.1 presents the Software Engineering Best Practices that have been included. These best practices include all the mandatory and recommended practices mentioned in Section 3.4. Some of them are addressed more in depth than others, more detail can be found on Appendix A .

4.4 Process Restrictions

Different software development process usually target the development of certain types of projects or the development of particular products. For instance, RUP (Rational Unified Process) is probably the most suitable process to support large projects that involve the development of large (and eventually complex) systems. In this sense, EduProcess was

Table 4.1: SE Best Practices Included in the EduProcess

Knowledge Area	**Best Practices**
Software Modeling and Analysis.	Problem analysis, use cases, data flow diagrams, information modeling, behavioral modeling, architectural modeling, analyzing correctness.
Requirements Analysis and Specification.	Definition of requirements, requirements process, requirements characteristics, analyzing quality, requirements evolution, traceability, prioritization, requirements management, eliciting requirements, requirements documentation, reviews and inspections, prototyping, validating product quality attributes, inspections.
Software Design.	Design and requirements interaction, architectural design, detailed design, design evaluation, prototype navigation, visual design definition, database design definition.
Software Verification and Validation.	Unit testing, integration testing, developing test cases, code inspections, validation through requirements verification.
Software Process.	Process concepts, project planning and tracking, software configuration management, team self management, software process usage, software configuration management.
Software Quality.	Quality requirements attributes.
Security.	-

conceived to support the development of small-sized Web information systems, to be used by novice developers in academic scenarios; particularly in project-based courses. Next we explain these restrictions more in detail.

4.4.1 Type of Projects to be Addressed

There are three types of projects that can be addressed with EduProcess: *new developments, system extensions* and *reengineering*. The first type corresponds to green field projects; i.e., new projects that are conceived and developed from scratch, even if they replace a legacy solution. Extension and reengineering projects represent brown field initiatives. Extensions typically involve adjusting the functionality of an already implemented software and adding new functionalities. Reengineering projects focus on rethinking or re-conceiving an already implemented system, eventually reusing some legacy components.

These project types represent a challenge for student teams due to different reasons. New developments push the students to help the client clarify the problem to address and identify the project context, because the project goal and scope are usually not clear at the beginning. Moreover, these projects require that the developers know enough (or learn enough) about the application domain in order to propose a suitable solution that really solves or mitigates the proposed problem.

Extension projects assume that the product is already designed (at least its architecture and main components), and also the technology used to implement the product (programming language, database, and sometimes, development frameworks). In this situation, the developers must develop an understanding of the design and code authored by other people, which is

always a challenge. Usually, in this type of project the problem to address is clear and the knowledge of the business niche is more available than in new developments.

Reengineering projects are normally a re-thinking and re-implementation of an existing system, which typically involves changes to the business logic, process to be supported, functionality to be offered to the end-user, and eventually, changes of the technologies used for implementing the solution. Frequently these systems provide services or information to other systems; therefore, reengineering projects impose several functional and technological challenges to the developers, particularly if they need to reuse components of the legacy system.

Typically the project type is a variable that is part of the project context, and demands that the software process be adjusted for addressing each particular project type. Meaning that the software process must address each project type. However, EduProcess was designed to deal with these three types of projects, which allows the instructor to keep only one course calendar and one set of assessment activities. This considerably eases the load of the instructional, monitoring and assessment activities, although the focus of the product evaluation changes depending on the project type.

4.4.2 Product Type, Size and Complexity

As mentioned before, EduProcess was conceived to help guide students building mainly Web information systems, since these can be built during a semester and they are the most frequently developed product by the software industry. These products can be developed in a project-based course, even if the students (developers) do not have a deep knowledge of the business domain to which the problem belongs to. If the projects to be addressed are almost exclusively Web information systems related somehow to the university, they tend to have similar complexity and involve a business domain that is not completely unknown for the students, which is highly convenient for these experiences. Moreover, the students perceive that learning to develop this kind of system is valuable for them.

Provided that semesters have a fixed duration, the project to be built in project-based courses should be selected and defined to ensure that a reasonable product can be obtained in such a period. Therefore, the size and complexity of the product to be developed should be bounded, also to avoid the creation of false expectations by the client. We recommended that the course instructor and client agree on a first scope for the project; and then the development team must negotiate the final scope with the client, sticking to the agreements reached with the instructor. Thus, the students learn to negotiate, but keep the project to a size and complexity that they can manage and correctly develop within the course time frame.

4.4.3 Dealing with the Requirements for Academic Software Processes

In Section 2.2, we presented a set of "minimal" requirements for a software process to be used in an academic scenario, developed by Paula [123]. EduProcess address all the requirements listed by Paula.

- *Process architecture* - The complete educational process is clearly defined with its inputs, activities, outputs, deliverables and milestones. EduProcess is specified in detail and is available online (http://www.dcc.uchile.cl/∼mmarques/).

- *Team orientation* - EduProcess was designed to be used by small software teams, from five to seven students, with adequate roles for a prescriptive software process and related activities fairly distributed.

- *Project cycle time* - The cycle time of EduProcess is measured in weeks. EduProcess can be used in courses from seven to fifteen weeks of duration.

- *Standards and practices* - EduProcess uses SE standards and best practices; e.g.; requirements, design, testing and others. The standards and practices used in EduProcess are not arbitrary they were selected based on our already analysis of SE best practices already presented.

- *Student support* - The process includes guidelines and templates to help students to understand what they have to accomplish. And EduProcess also includes the reflexive weekly monitoring process, which helps students during the whole project.

- *Instructor support* - EduProcess has a specific track that specifies the instructors support activities.

Chapter 5

EduProcess Evaluation

The main goal of this thesis work is to define a prescriptive software process that helps all the involved stakeholders to run software projects in academic scenarios, considering loosely-coupled teams. The process should be usable in a systematic and affordable way, by students, instructors, clients and users. To this end, we created and formalized the EduProcess. However, we have to determine wether the EduProcess makes the practical experiences of project-based software engineering courses, repeatable, while producing positive results. In order to do this, we used EduProcess during four semesters in a project-based course at the Universidad de Chile, and evaluated this activity using a case study.

A case study investigates a contemporary phenomenon (the case) in its real-world context, especially when the boundaries between the phenomenon and its context are not clear. According to Yin [176], case study evidence may come from six sources: documents, archival records, interviews, direct observation, physical artifacts and participant observation. In our case, we use the first five sources of evidence.

The use of multiple sources of evidence in case study research allows a researcher to address a broad range of historical and behavioral issues. However, the most important advantage is the development of potentially converging lines of inquiry; particularly, triangulation gives strength to the findings since they can be corroborated with data from multiple sources. Thus, any case study finding or conclusion is likely to be more convincing and accurate if it is based on several sources of information.

This case study intends to determine the validity of the stated hypotheses, that a prescriptive software process for guiding SE development projects using a disciplined approach (EduProcess) can:

(H1) make these experiences repeatable while investing an affordable effort.

(H2) help produce positive results in software projects.

By repeatable we mean that the process can be used more than once, having the same pattern of results; and by affordable we mean that the time and skills required from all stakeholders to follow the process, should be aligned with what we can expect in an academic scenario. We consider "positive results": a suitable level of teamwork, and projects that

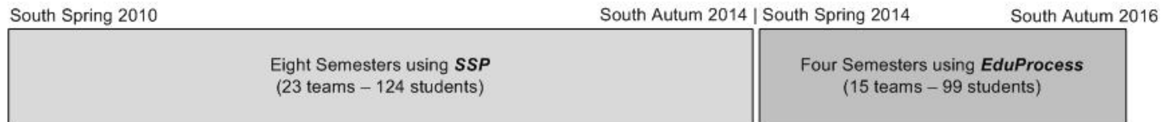


Figure 5.1: Timeline of the Evaluation Process

successfully deploy the resulting software product. In order to evaluate these two hypotheses, qualitative and quantitative data was collected, analyzed and compared with previous experiences.

As mentioned, EduProcess was used in a project-based course during four semesters, involving 14 software teams that obtained positive results in their projects. Given that the same pattern of results was evidenced in the four semesters, it is possible to say that the process and the results are repeatable. These results partially support the stated hypotheses. However, in order to complete the validation process, we compared the performance of the teams that used EduProcess with those that used SSP [115] (the predecessor of EduProcess). The same exercise was done by considering the effort required for using processes.

The case study considered the results of the last eight semesters in the Software Engineering II course (CC5401) in which SSP was used to support the course (involving 23 teams, and 131 students), as well as the last four semesters (14 teams, 94 students) in which EduProcess was used in the same scenario (see Figure 5.1). As it will be shown, both experimentation scenarios are comparable.

The results of this comparison indicate that both processes are repeatable; however, EduProcess can be repeated by third party teams using the process definition (i.e., its formalization), which opens the possibility that other instructors and students can take advantage of this proposal. The comparison also shows that the use of both processes demands a similar effort from instructors, students, clients and users, and such an effort is affordable by these people, which supports H1.

On the other hand, both processes were able to get positive results, but the quality of the software products and the consistency of this attribute was higher when using EduProcess. The average (mean) quality of the products obtained with SSP was rated as 5.71 (S.D.: 1.03) on a scale from 1 to 7. In the case of EduProcess, the average quality was rated as 6.31 (S.D.: 0.37), which is higher than SSP and also more consistent, considering the standard deviation. These results supports H2.

In the next sections we present the details of the case study, the statistical analysis of the results, and the hypotheses validation. The chapter finishes with the threats to validity of the case study, and a discussion of the results. All the data shown here is presented in a comparative way, using a two independent samples test. Particularly, we show the results of using SSP and also those resulting from using EduProcess.

5.1 Case Study Scenario

We conducted this case study in the Computer Science Engineering program of the Universidad de Chile. This program lasts 11 semesters and addresses most knowledge areas

considered in the ACM-IEEE [119] Curriculum Guidelines. This program has three mandatory SE courses: the first one is mainly theoretical, and the other two have a more practical focus. These last two courses use different development approaches to support student projects, since one is a project-based course and the other is a capstone course. The former (CC5401 - Software Engineering II) uses a disciplined process and the development is conducted by students in their own time; therefore, most project activities are decentralized. The latter course (CC5402 - Software Project) uses agile practices, the development is conducted by students at the client facilities, and the students work more or less together (i.e., they conduct a tightly-coupled development).

EduProcess was designed to support the course CC5401 (i.e., the project-based course), to help reduce the coordination, effectiveness and productivity problems evidenced by the student teams enrolled in this course, when using SSP. The two scenarios considered in this case study (i.e., with SSP or with EduProcess) are structurally comparable due to several reasons: the instructor and the teaching team was the same, the course materials hardly change, the projects type/duration/complexity were similar, and the students were comparable. The only major difference between these scenarios was the software process supporting the practical experiences: SSP vs. EduProcess.

CC5401 is a mandatory course for students of ninth semester of the program, where students must work in teams for 12 weeks to develop a Web information system for a real client. This client normally is a department of the University; so they are aware that the major goal of the course is to teach students how to run software projects and not just to obtain a software product. The development teams are 5 to 7 students that self-organize, but each student must play one of the following specific role in the team: project manager, requirement analyst, designer-developer or quality assurance engineer.

5.2 Data Gathering and Analysis

Over the last six years, we worked twice per year with the CC5401 course, observing the process in order to improve it, gathering qualitative and quantitative data during twelve observed semesters. As mentioned, 23 teams (131 students) used SSP during 8 semesters, and after that, 14 teams (94 students) used EduProcess during 4 semesters, with the same purpose.

5.2.1 Quantitative Data

The quantitative data sources that we used in this case study are the following: course grades, project grades, student surveys, peer-assessments, amount of software built (in terms of function points) and suitability of the project implementation (a Boolean variable that indicates if the product was deployed or not).

5.2.1.1 Course Grades

The course grades are assigned on scale from 1 - 7. In order to pass the course, a student needs to obtain a grade of 4 or higher. The software product and the process used during the semester accounts for 60% of the final grade, and the other 30% comes from two written midterms. Students with a midterm average of less than 5 must take a final exam.

5.2.1.2 Project Grades

Projects are graded on the same scale as courses, i.e., from 1 to 7. However, the project grades involve different checkpoints according to the role played by each team member. At each milestone defined by the EduProcess, there is a role that has a major relevance, and the weight of the grades changes according to the role of the students. For instance, the requirements evaluation activity (a formal technical review) is mostly responsibility of the analyst, but the work of the quality assurance engineer and the project manager is also evaluated (others roles are not). Moreover, the client also grades the final product and this value accounts for 5% of the final project grade.

5.2.1.3 Student Survey

At the end of each semester, the university deploys a survey, allowing students to grade the courses they were enrolled in during the current semester. The results of this survey are completely anonymous; only the average results and anonymous comments about the course are disclosed. This survey was established by the engineering school, and students rate several items using a Likert scale. The survey system then assigns a score (from 1 to 5) to each item using the students' answers.

5.2.1.4 Peer-Assessment

At the end of first three phases of the process (i.e., three times during each course offering), the students evaluate their teammates through peer-assessment. This activity is mediated by a Web system that shows each student the average of what his/her teammates think about his/her work. The average of the peer-assessment accounts for 5% of the course final grade. The main idea of this activity is to measure, according to student's perspective, the coordination capability, motivation, sense of team belonging and effort invested by each team member in the project. In Appendix B we present the items involved in this peer-assessment.

5.2.1.5 Amount of Software Built

In order to analyze the amount of software implemented by the student teams, we have used the historical record of the software requirements involved in each project. Every team in the course has to use a requirements management tool (MainReq) which was specifically developed to support the students and the instructor in project tracking activities. This tool keeps a record of the project requirements and their evolution. For each requirement, the system keeps a detailed record of its definition, e.g., its description, type, degree of completion, priority and criticality. Using this information, we determined the amount of software developed by each team, and the relevance of the addressed requirements. The effort to deal with each functional requirement was estimated by the author of this thesis using the RESC (Raw Estimation based on Standard Components) method [114]. This estimation method is an adaptation of PROBE (Proxy-Based Estimating) [64] that specifies the development effort in terms of function points (FP). The use of RESC allowed us to represent in a single metric (FP) the effort spent by the students on their projects, and thus to compare these efforts. To minimize a possible bias in the estimation of the effort required to address each requirement (due to subjectivity), two experienced software engineers were asked to make the same estimation using RESC. Values estimated by these engineers were then compared to those estimated by the author. This comparison showed a difference of 14% in average between the estimations of

both groups (the largest difference was of 18%, the lowest one was of 5% and the median was of almost 10%). Given that the estimations of the experts were similar to those guaranteed by the thesis author, we used the latter values in our analysis.

5.2.1.6 Suitability of the Project Implementation

Our last quantitative attribute is represented by a Boolean variable that indicates the suitability of the project implementation. It indicates if the project was successfully deployed and used by the client organization or not. Here we only have two options: successful projects (i.e., those that deployed a software product) and unsuccessful projects (i.e., those that for any reason did not end up being deployed).

5.2.2 Qualitative Data

Qualitative information was gathered through two mechanisms: peer-assessments (that capture the strengths and weaknesses of each teammate) and monitors feedback (records of the weekly reflexive monitoring sessions). All the qualitative data was analyzed in three steps, following the guidelines of grounded theory: coding comments, keyword categorization and comment categorization. We now describe these steps:

- *Step 1*: Coding comments. During this step, we semantically analyze the students' comments, with the goal of finding the most important keywords according to their content.

- *Step 2*: Keyword categorization. After coding the comments, several keywords are associated to similar concepts, so we grouped them into more general concepts.

- *Step 3*: Comment categorization. During this stage we established which comment of the peer-assessments belonged to each concept, according to the keywords used in step 2.

All the students quotes mentioned in this chapter are excerpts of the qualitative data analysis performed of the relevant comments made by students and monitors.

5.2.2.1 Peer-Assessments

The peer-assessment involved eight quantitative items that the students must rate using a Likert scale (1-completely disagree to 5-completely agree), as well as two open questions, where students can comment on the strengths and weaknesses of their teammates. These last two open questions are optional. However, the students that provide comments about their teammates strengths and weaknesses usually contribute with relevant observations.

5.2.2.2 Monitoring Feedback

During the monitoring sessions, the monitor uses a "suggested" script to guide these sessions. Prompted by this script the monitor report what she/he finds by the reflexion meetings. The records of these sessions are processed and only handed in to the instructor when the course has finished, in order to avoid interference during the current course. Therefore, we have qualitative weekly data about how the teams were working, their problems, and how and when they were solved.

5.3 Results

Table 5.1 presents general data about year and semester the teams were enrolled in the course, the number of students in each team and the software process used. In the University of Chile there is no specific amount of students that can be enrolled in a course; therefore, we have a high variation in the number of teams participating in the course every semester.

In order to determine the impact of using EduProcess in this course, we stated the following null hypothesis: "there is no difference between the teams that used EduProcess and those using SSP". We considered this null hypotheses for all our variables: course grades, project grades, student surveys, coordination level, sense of team belonging, engagement, initiative, communication and commitment. We also performed an analysis of the quantitative information with respect to this null hypothesis. Next, we describe each variable and the results we obtained from these different perspectives.

5.3.1 Course and Project Grades

Table 5.1 also shows the average of course and project grades for all the teams analyzed. The course grade mean for the teams that used the SSP was 5.51, and it was 6.09 for those using EduProcess: a difference of 10.7%. The project grade mean for the teams that used the SSP was 5.71, and for those using EduProcess it was 6.31, a difference of 10.5%. The distribution of course and project grades was tested for normality using D'Agostino and Pearson omnibus test K^2 [37]. This statistical test is able to detect deviations from normality due to skewness or to kurtosis. For project grades K^2=20.76 (SSP) and K^2=4.26(EduProcess), p-value $= 0.01 < .05$ and for course grades K^2=25.8 (SSP) and K^2=8.62(EduProcess), p-value $= 0.01 < .05$; so course grades and project grades data are normally distributed. A two tailed independent samples t-test was used to evaluate how far the improvement of the grades was from the null hypotheses; the results are presented in Table 5.2. The t-test shows that the null hypotheses for the improvement of course and project grades were rejected with a statistically significant difference (α<0.05). All statistical analysis was done using IBM SPSS version 23. The effect size expresses group differences in terms of standard deviation units, and it has a more meaningful interpretation than a difference in percentages. For the course and project grades, the average effect size was d=0.76 for course grades and d=0.68 for project grades, which is noticeably larger than zero.

Figure 5.2 presents two pie charts comparing course grades in ranges. A direct comparison between course grades is not possible due to the unequal number of teams that used the SSP and EduProcess. Figure 5.2(a) presents the SSP data and Figure 5.2(b) presents EduProcess data. It is possible to see that grades are higher in the teams that used EduProcess than those using SSP. Moreover, there are no course grades within the ranges of 4.00-4.49, 4.50-4.99, 5.00-5.49 in the teams that used EduProcess; however, 37% of the teams that used the SSP fall in this overall range.

A similar comparison can be done using project grades. Figure 5.3(a) presents the scores obtained with SSP, and Figure 5.3(b) illustrates this information for EduProcess. This figure shows an increase in the percentage of the teams with project grades between 6.00-6.49; they represent 50% of the teams when using SSP, and 79% when using EduProcess. Moreover, there are no teams with score within the ranges of 4.00-4.49, 4.50-4.99, 5.00-5.49 when EduProcess

Table 5.1: Semesters and Students Enrolled in the CC5401 Course

	Year	Semester	Number of Students	Team ID	Course Grade	Project Grade
SSP	2010	Spring	6	Team 1	5.78	6.46
	2011	Autumn	6	Team 2	5.55	6.12
	2011	Autumn	6	Team 3	4.48	5.44
	2011	Autumn	6	Team 4	5.55	6.03
	2011	Autumn	6	Team 5	4.96	4.74
	2011	Autumn	6	Team 6	5.08	4.88
	2011	Spring	5	Team 7	5.94	5.95
	2011	Spring	5	Team 8	5.81	5.97
	2012	Autumn	7	Team 9	5.95	6.14
	2012	Autumn	7	Team 10	5.35	5.93
	2012	Autumn	6	Team 11	5.85	6.34
	2012	Spring	6	Team 12	5.57	4.55
	2012	Spring	5	Team 13	4.15	4.00
	2013	Autumn	6	Team 14	5.85	6.07
	2013	Autumn	6	Team 15	6.05	6.17
	2013	Autumn	6	Team 16	5.97	6.28
	2013	Autumn	6	Team 17	4.83	5.12
	2013	Spring	5	Team 18	5.92	6.32
	2013	Spring	5	Team 19	4.92	5.08
	2013	Spring	5	Team 20	6.10	6.40
	2014	Autumn	5	Team 21	6.12	6.32
	2014	Autumn	5	Team 22	5.96	6.20
	2014	Autumn	5	Team 23	4.72	4.88
	Total:	8 semesters	131 students	Average	5.51	5.71
EduProcess	2014	Spring	7	Team 24	6.13	5.77
	2014	Spring	7	Team 25	6.09	6.29
	2014	Spring	7	Team 26	5.90	6.07
	2015	Autumn	7	Team 27	6.39	6.64
	2015	Autumn	7	Team 28	6.21	6.47
	2015	Autumn	7	Team 29	6.24	6.44
	2015	Autumn	6	Team 30	5.92	6.08
	2015	Autumn	6	Team 31	6.12	6.27
	2015	Spring	7	Team 32	5.63	5.96
	2016	Autumn	7	Team 33	6.19	6.40
	2016	Autumn	7	Team 34	5.90	6.09
	2016	Autumn	7	Team 35	6.06	6.19
	2016	Autumn	6	Team 36	6.17	6.35
	2016	Autumn	6	Team 37	6.37	6.45
	Total:	4 semesters	94 students	Average	6.09	6.31

Table 5.2: Statistical Analysis of Course and Project Grades

Grades	SSP		EduProcess		t-test(t-value)	Sig(2-tailed)	Cohen d
	Mean	Std. Deviation	Mean	Std. Deviation			
Course	5.51	1.01	6.09	0.38	5.41	<0.01	0.76
Project	5.71	1.03	6.31	0.37	4.74	<0.01	0.68

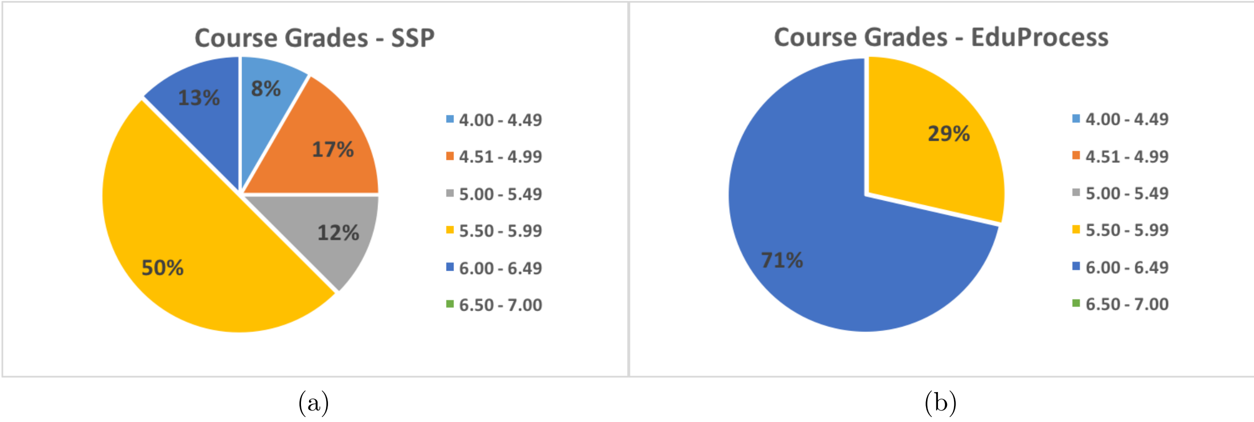


Figure 5.2: Course Grades using (a) SSP and (b) EduProcess

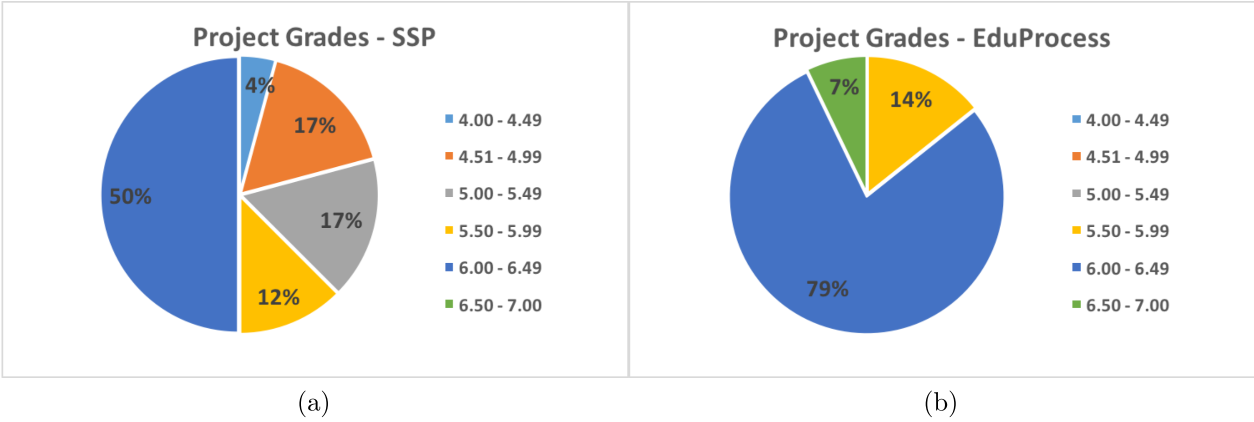


Figure 5.3: Project Grades using (a) SSP and (b) EduProcess

Table 5.3: Items of the Student Survey

Domain	**Items Description**
Domain Knowledge	The instructor demonstrates mastering of the course material and knowledge domain.
Pedagogical Skills	The instructor uses teaching strategies that encourage student participation. He/she creates opportunities for reflection proposing appropriate and challenging problems.
Course Organization	The instructor uses relevant literature related to the course. There is a close relationship between the course syllabus and the development of curricular activities. The instructor evaluates in a timely manner. The complementary activities successfully contributed to the learning process.
Course Design and Workload	The course syllabus is covered in the stipulated amount of time.
Interpersonal Relations	The instructor promotes attitudes of tolerance, social commitment and respect for differences (social, cultural, economic and others). He/she considers the opinion of all students during the course.
Assessments	There is a consistency between what is being taught and what is evaluated in assessments. The instructor informed students of the course evaluation criteria in a timely manner.

was used; however, they represent 38% when using SSP.

5.3.2 End of Semester Student Surveys

At the end of the semester the university conducts a formal survey asking students about specific aspects of the courses in which they were enrolled. The items specifies about course instructors are presented in Table 5.3. The students use a Likert scale to give their opinion for each item (from 1-strongly disagree, to 5-strongly agree).

The results presented in Table 5.4 indicate that there is no statistical difference in the items of the survey related to the instructor (domain knowledge, pedagogical skills and interpersonal relations), since the course instructor did not change in these semesters. However, there is a difference in the items where EduProcess has an impact (course organization, course design and assessments). Next, we present some quotes from the positive comments made by students in this survey:

"I really liked to work on a real project and being part of a real team".

"Good to have a practical experience".

"Very good course, you can learn a lot about how work will really be on the "street" as it says the instructor. Even if something is done wrong, or if the goals are not met, the instructor and the monitor are there to support and ensure that we learn a few lessons".

"This is a very good course, although many says otherwise (even students union), it comes quite well what is the reality of working with a client. In addition, the instructor and the monitor know a lot, you notice that they have a good grasp of the course topics".

Table 5.4: Student Survey Results

	Course Timing	Domain Knowledge	Pedagogical Skills	Course Organization	Course Design	Interpersonal Relations	Assessments
SSP	Spring 2010	6.57	6.46	5.46	5.25	6.69	5.86
	Autumn 2011	6.85	6.50	6.36	5.67	6.29	5.82
	Spring 2011	6.87	5.83	4.43	5.25	6.25	5.29
	Autumn 2012	6.64	6.45	5.78	5.18	6.50	6.14
	Spring 2012	7.00	6.33	5.50	5.83	6.60	5.00
	Autumn 2013	6.50	6.23	6.14	5.41	6.47	6.04
	Spring 2013	6.40	6.25	6.22	5.93	6.45	5.80
	Autumn 2014	6.45	5.93	6.00	5.42	6.64	5.73
Average		6.66	6.25	5.74	5.49	6.49	5.71
EduProcess	Spring 2014	6.75	6.67	6.68	6.03	6.60	6.79
	Autumn 2015	7.00	6.71	6.71	6.76	6.79	7.00
	Spring 2015	7.00	6.54	6.48	6.22	6.70	6.52
	Autumn 2016	6.54	5.77	6.17	6.10	6.78	6.08
Average		6.82	6.42	6.51	6.28	6.72	6.60
Difference		2.44%	2.80%	13.49%	14.29%	3.57%	15.54%

"It is very interesting to have a course that deals and shows what the work will be like, especially to be able to work for our community, with some internal organization of the university or a non-profit organization (so you do not feel that there are people who are taking advantage of our work)".

"The instructor showed a great commitment to student learning. In retrospect, his critical attitude towards the group presentations was understood".

"I find it very interesting to work with clients and to interact with them".

"It was something amazing to have examples of documents, requirements, design specifications. With only the lectures, things could be very abstract".

"Excellent course, very interesting dynamic that exists with the project allows a first approach to the relationship with a client who does not necessarily know how to develop software. There was always support and constant concern of the instructor and the monitor, it was really appreciated and needed."

On the other hand, we also received some negative comments:

"I realized later that MainReq (our tool that managed requirements) was really useful despite its shortcomings".

"It would be interesting to have projects that were not software information systems. This would force us as students to learn other types of architecture, which are not only repeatedly MVC models".

"The course should not have written evaluations".

Table 5.5: Statistical Analysis of the Peer-Assessments

Peer-Assessment Item	SSP		EduProcess		t-test(t-value)	Sig(2-tailed)	Cohen d
	Mean	Std. Deviation	Mean	Std. Deviation			
Coordination	4.17	0.95	4.68	0.72	14.31	<0.01	0.59
Belonging	4.18	0.99	4.63	0.71	12.38	<0.01	0.51
Engagement	4.28	0.96	4.62	0.73	9.40	<0.01	0.39
Commitment	4.00	1.08	4.44	0.78	12.11	<0.01	0.50
Communication	4.20	1.01	4.50	0.86	7.74	<0.01	0.32
Initiative	4.00	1.11	4.34	0.96	7.98	<0.01	0.34

"Students should be allowed to choose the project they will work on".

"A weakness of the peer-assessments is that some students prefer not to say the weakness of their colleagues, not to hurt their grades. I was personally guilty of it and paid for it".

"Since we are not good at designing, we could have a designer, to help the teams. It was one of the biggest weaknesses we have. It wold also allow us to deliver better software to the clients".

5.3.3 Peer-Assessments

The anonymous peer-assessment performed by the students at the end of each iteration have 8 questions, and was used to determine the coordination level of teams that used EduProcess, as well as the teams that used SSP. Using this instrument, students evaluate six aspects about their teammates [154]: engagement, sense of team belonging, initiative, communication, coordination and commitment. Each aspect was represented through one or more sentences (items) describing an expected attitude of the teammates. Students evaluate these statements using one of the following options: 1-never, 2-almost never, 3-sometimes, 4-regularly, and 5-always. In the following subsections, we analyze the results of each aspect of peer-assessment.

5.3.3.1 Coordination Level

The aspect of team coordination was evaluated using the following statement: *"He/she fulfills his/her assignments properly, working transparently and in coordination with the rest of the team"*. Using this information, we calculated the average coordination level of each team. Table 5.6 shows these values for teams that used the SSP and also for those using EduProcess. The distribution of coordination level was tested for normality using D'Agostino and Pearson omnibus test (K^2) [37] K^2=38.44 (SSP) and K^2=25.10(EduProcess), p-value $= 0.01 < .05$; so coordination level data are normally distributed. These results show a significant difference in the coordination level of teams that used EduProcess (M=4.68, SD=0.72) and teams that used SSP (M=4.17, SD=0.95) under the following conditions t(2,317)=14.31 and $\alpha > 0.05$. With an $\alpha = 0.05$ of confidence interval, the mean difference was 0.37 to 0.51. Therefore, these statistical results support the second hypothesis (H2 - EduProcess helps produce positive results in software projects). For coordination, the average effect size was found to be d=0.59, which is considered of medium impact. Table 5.5 shows the details of each team using both SSP and EduProcess.

The comments in the peer-assessment indicate some coordination limitations for both types of teams after the first iteration, quoting *"Learning to coordinate ourselves was a big deal"*.

Table 5.6: Team Aspects - Coordination, Sense of Team Belonging, Engagement

	Team	Coordi-nation	Sense of Team Belonging	Engage-ment	Cronbach's Alpha
SSP	Team 1	4.53	4.29	4.60	0.82
	Team 2	3.92	4.01	3.90	0.93
	Team 3	4.50	3.72	4.23	0.89
	Team 4	4.47	4.43	4.65	0.99
	Team 5	3.74	3.65	4.00	0.88
	Team 6	3.83	4.08	4.55	0.89
	Team 7	4.36	4.40	4.45	0.91
	Team 8	4.54	4.72	4.61	0.92
	Team 9	4.67	4.23	4.65	0.84
	Team 10	4.17	4.33	3.77	1.00
	Team 11	3.50	4.00	3.55	0.93
	Team 12	3.60	4.40	3.90	0.84
	Team 13	3.77	4.38	3.77	0.86
	Team 14	3.83	3.90	3.97	0.82
	Team 15	4.17	4.09	4.43	0.89
	Team 16	4.66	4.24	4.62	0.80
	Team 17	3.72	4.00	4.32	0.93
	Team 18	4.04	4.24	4.28	0.90
	Team 19	3.08	3.88	3.69	0.83
	Team 20	4.57	4.54	4.21	0.89
	Team 21	4.10	4.22	4.14	0.82
	Team 22	4.51	4.70	4.45	0.99
	Team 23	4.17	4.04	4.50	0.82
Average		4.17	4.18	4.28	0.88
EduProcess	Team 24	4.45	4.46	4.12	0.93
	Team 25	4.80	4.56	4.67	0.88
	Team 26	4.67	4.59	4.72	0.79
	Team 27	4.75	4.78	4.80	0.83
	Team 28	4.57	4.65	4.63	0.89
	Team 29	4.57	4.66	4.72	0.92
	Team 30	4.61	4.63	4.58	0.80
	Team 31	4.45	4.54	4.54	0.81
	Team 32	4.36	4.18	4.17	0.83
	Team 33	4.91	4.71	4.70	0.79
	Team 34	4.76	4.70	4.63	0.88
	Team 35	4.69	4.65	4.72	0.77
	Team 36	4.80	4.73	4.64	0.82
	Team 37	4.82	4.91	4.73	0.81
Average		4.68	4.63	4.62	0.84

"Working with people we do not know is hard". However, after the second iteration this kind of limitation only appears in teams that used SSP. This matches the observations of the monitors that participated in EduProcess, who reported that *"There seems to be an improvement in the team members coordination after the first iteration"*, and also with the final comments of some participants of teams that used EduProcess, who stated that *"The monitoring sessions showed us the need of coordinating our activities. Unfortunately, we understood this only during the last stage of the project"*, and also that *"We started working uncoordinatedly, but this changed after each [monitoring] session. I feel that we finished working as a team"*.

5.3.3.2 Sense of Team Belonging

The first two statements of our peer-assessment were used to determine the sense of team belonging of each student. These sentences state: *"He/she assumes the project as a team effort, providing support in project tasks"* and *"He/she is able to seek help when there are problems"*.

The distribution of sense of team belonging was tested for normality using D'Agostino and Pearson omnibus test, K^2=34.96 (SSP) and K^2=28.42(EduProcess), p-value $= 0.01 < .05$; so sense of team belonging data is normally distributed. Table 5.6 shows that, according to the peer-assessment, there is a significant difference in the sense of team belonging between teams that used EduProcess (M=4.63, SD=0.71) and those that used SSP (M=4.18, SD=0.99) under the following conditions t(2,317)=12.38 and α=0.05. Similar to the previous case, by using a 0.05 of confidence interval, the mean difference was between 0.43 to 0.57, which also supports the second hypothesis (H2 - EduProcess helps produce positive results in software projects). The average effect size for this aspect was d=0.51, which is considered of medium impact. The details per team are shown in Table 5.6.

The monitors participating in EduProcess reported that the students made important efforts to become a team during the first iteration: *"Most students worked hard during the first iteration to become part of their teams, and most of them seem to succeed because their relationships become more relaxed and friendly during the second part of the project... The free riders become visible to everybody"*. Moreover, some students on teams that used EduProcess mentioned: *"The sessions [monitoring] help us realize that people can contribute in several ways, and this understanding made us a great team"*. *"The weekly discussions of the team limitations allowed us to overcome them... This project brought us closer"*.

Teams using SSP did not have weekly team meetings, and their comments allude to a lack of coordination because of this. *"We finished acting as a group of developers, because we were not monitored... Two peer-assessments are not enough to help us improve our internal work"*. *"We worked in a distributed way, as we usually do, but it was not a good idea. We had communication and coordination problems"*.

5.3.3.3 Team Members Engagement

Engagement is related to the commitment that team members have with the project activities. This aspect was evaluated considering the following statement: *"He/she assumes the project is a team effort, providing support in project tasks"*. The distribution of engagement was tested for normality using D'Agostino and Pearson omnibus test, K^2=51.89 (SSP) and K^2=28,24(EduProcess), p-value $= 0.01 < .05$; so engagement data is normally distributed.

Table 5.6 shows an average score of 4.62 for teams that used EduProcess and 4.28 for teams that did not use it, which is 7.83% higher. In Table 5.5 illustrates the difference of the engagement when the teams used EduProcess (M=4.62, SD=0.73) and SSP (M=4.30, SD=0.96) under the following conditions t(2,317)=9.40 and α=0.05. With a 0.05 of confidence interval, the mean difference was 0.26 to 0.41. These statistical results also support the second hypothesis (H2 - EduProcess helps produce positive results in software projects). In this case the average effect size was d=0.39 that is considered of low to medium impact. The following student comment from a team that used EduProcess support this result: *"The monitoring sessions made me know in a hidden way what was happening on the project and to do my part and not let the others down"*.

5.3.3.4 Initiative

Student initiative was measured using the opinions gathered for the following statement: *"He/she demonstrates initiative to achieve project success"*. The distribution of initiative was tested for normality using D'Agostino and Pearson omnibus test, K^2=17.12 (SSP) and K^2=34.93 (EduProcess), p-value $= 0.01 < .05$; so initiative data is normally distributed. Table 5.7 shows an average score of 4.34 for initiative in teams that used EduProcess and 4.00 for teams that used SSP (8.78% higher the former). Table 5.5 presents the results of the statistical analysis. There we can see a difference in the initiative results of teams that used EduProcess (M=4.34, SD=0.96) and teams that used the SSP (M=4.00, SD=1.11) considering t(2,317)=7.98 and α>0.05. With a 0.05 of confidence interval, the mean difference was 0.26 to 0.44. Once again, these statistical results support the second hypothesis. The average effect size for initiative t was found to be d=0.34 (considered as low impact). A student that used EduProcess mentioned: *"This needs to be done! If he is not able to do it, I will help him to code"*, corroborating this result.

5.3.3.5 Communication

The communication aspect was measured using the following two statements: *"He/she shows a communicative attitude facilitating the teamwork"* and *"He/she has maintained good communication with the client, generating value to project execution"*. The distribution of communication was tested for normality using D'Agostino and Pearson omnibus test, K^2=34.18 (SSP) and K^2=39.12 (EduProcess), p-value $= 0.01 < .05$; so communication data is normally distributed. Table 5.7 shows the average score for communication, which was 7.28% higher when EduProcess was used. Table 5.5 presents the communication results of teams that used EduProcess (M=4.50, SD=0.86) and also those that used SSP (M=4.20, SD=1.01) with t(2,317)=7.74 and α>0.05. The mean difference was 0.22 to 0.38 for a confidence interval of 0.05. This also supports the second hypothesis. For this aspect the average effect size was d=0.32 (considered of low impact). To support this finding, we can mention the following comment of a student that used EduProcess: *"At this point of the project we (developers) need to be in constant contact with each other; the course schedule pushed us to have meetings at least once a week"*.

5.3.3.6 Commitment

The commitment was measured using the following two statements: *"He/she demonstrates commitment to the project and the team"* and *"He/she adequately fulfills the assigned tasks"*. In order to clarify the difference between engagement and commitment, we use the definition

Table 5.7: Team Aspects - Initiative, Communication, Commitment

	Team	Initiative	Communication	Commitment	Cronbach's Alpha
SSP	Team 1	4.18	4.47	4.05	0.77
	Team 2	3.92	3.88	3.53	0.81
	Team 3	3.94	3.83	3.68	0.93
	Team 4	4.43	4.53	4.33	0.88
	Team 5	3.63	4.00	4.13	0.83
	Team 6	3.83	4.05	4.08	0.80
	Team 7	4.47	4.52	4.36	0.88
	Team 8	4.58	4.42	4.66	0.92
	Team 9	4.50	3.85	4.21	0.79
	Team 10	4.17	4.47	4.07	0.80
	Team 11	3.50	4.50	4.00	0.91
	Team 12	3.63	4.40	3.30	0.88
	Team 13	3.46	3.77	3.54	0.86
	Team 14	3.60	4.47	3.80	0.80
	Team 15	3.23	3.96	4.09	0.85
	Team 16	3.90	4.62	4.41	0.78
	Team 17	3.52	4.16	3.32	0.86
	Team 18	3.32	4.32	3.80	0.78
	Team 19	3.20	4.12	3.52	0.89
	Team 20	3.75	4.32	3.43	0.85
	Team 21	3.55	3.95	3.55	0.86
	Team 22	4.20	4.50	4.35	0.88
	Team 23	4.57	4.83	4.25	0.86
Average		4.00	4.20	4.00	0.89
EduProcess	Team 24	3.68	4.24	3.97	0.87
	Team 25	4.38	4.58	4.61	0.76
	Team 26	4.13	4.70	4.36	0.88
	Team 27	4.69	4.68	4.71	0.85
	Team 28	4.30	4.43	4.44	0.91
	Team 29	4.20	4.53	4.39	0.80
	Team 30	4.54	4.18	4.51	0.88
	Team 31	4.43	4.22	4.34	0.90
	Team 32	4.02	4.08	4.06	0.82
	Team 33	4.52	4.61	4.48	0.85
	Team 34	4.31	4.50	4.36	0.85
	Team 35	4.67	4.65	4.55	0.87
	Team 36	4.55	4.66	4.63	0.80
	Team 37	4.49	4.88	4.77	0.79
Average		4.34	4.50	4.44	0.85

by Albrech [2], who indicates that commitment refers to student's satisfaction as well as identification with the team; however, engagement goes a step further and involves the student making discretionary efforts towards the attainment of project goals. The distribution of commitment was tested for normality using D'Agostino and Pearson omnibus test, $K^2=30.79$ (SSP) and $K^2=48.14$ (EduProcess), p-value $= 0.01 < .05$; so commitment data is normally distributed. Table 5.7 shows the average scores for commitment, which was 11.93% higher when EduProcess was used. Table 5.5 presents the statistical analysis indicating the commitment results of teams that used EduProcess (M=4.44, SD=0.78) and also those from the teams that used SSP (M=4.00, SD=1.08) considering t(2,317)=12.11 and $\alpha>0.05$. The mean difference was 0.39 to 0.55 using a confidence interval of 0.05. The average effect size was d=0.50 (considered of medium impact). These results also support the second hypothesis.

5.3.4 Evaluation of Team Productivity and Effectiveness

Provided that all the teams had to use MainReq to manage the project scope and status, it was quite simple to identify the software requirements involved in each project and also the status of each one (i.e., fulfilled, unfulfilled or ambiguous). This information can be considered reliable since it is the result of the last software inspection conducted by the instructor and teaching assistant just before project closure.

Table 5.8 shows the amount of Function Points (FP) that each team had to address in their projects. The table also indicates how many software requirements were fulfilled (FP Addressed), which were partially fulfilled (FP Ambiguous) and which were not fulfilled (FP Not Addressed). With these numbers, we determined how much software was implemented. The results show that the teams that used EduProcess were not more productive than the teams that used SSP, contradicting what we expected.

The second column in Table 5.9 indicates the percentage of critical requirements addressed by each team. A requirement is considered "critical" if it must be addressed before putting the software into production (i.e., it is mandatory). The criticality level of each requirement was defined by the development team along with the client, and verified by the instructional team using the MainReq.

The percentage of fulfilled critical requirements when using SSP and EduProcess does not seem to be very different. However, the most interesting aspect is the percentage of successful projects in each scenario. Thirteen out of fourteen teams that used EduProcess were able to address all critical requirements, thus making it possible to deploy a final product. This represents a 92,85% of effectiveness. In case of teams that used SSP, this number falls to 69.56%; there were 16 successful out of 23 participating teams.

Table 5.10 shows the results of the statistical analysis related to productivity in FP. There was no statistical difference in the total FP, addressed FP, not addressed FP , ambiguous FP or in the %FP addressed, since the p in all these analyses was higher than 0.05. There is some difference in the amount of FP fulfilled per team member, since the $\alpha>0.05$. The results of teams that used EduProcess are M=13.39, SD=2.12 and teams that used SSP are M=16.97, SD=4.32 considering t(37)=2.88 and $\alpha=0.05$. With a 0.05 of confidence interval, the mean difference was 1.24 to 3.60. Therefore, these statistical results do not reject the null hypotheses.

Table 5.8: Team Productivity

	Team	FP	FP Addressed	FP Not Addressed	FP Ambiguous	% FP Addressed
SSP	Team 1	108	98	10	0	91%
	Team 2	67	64	3	0	95%
	Team 3	144	142	2	0	99%
	Team 4	94	81	13	0	86%
	Team 5	72	56	22	0	79%
	Team 6	63	34	30	3	53%
	Team 7	75	50	26	0	66%
	Team 8	112	69	43	0	62%
	Team 9	98	98	0	0	100%
	Team 10	73	72	0	2	99%
	Team 11	60	47	6	8	78%
	Team 12	147	105	40	2	71%
	Team 13	120	56	64	0	47%
	Team 14	102	83	19	0	81%
	Team 15	108	52	56	0	48%
	Team 16	100	84	16	2	84%
	Team 17	102	79	23	6	77%
	Team 18	93	78	9	5	84%
	Team 19	73	43	24	6	59%
	Team 20	60	48	12	0	80%
	Team 21	57	57	0	0	100%
	Team 22	45	35	10	0	78%
	Team 23	55	28	17	10	51%
Average		88	67	20	2	76%
EduProcess	Team 24	70	51	19	0	73%
	Team 25	99	88	11	0	89%
	Team 26	92	67	17	8	73%
	Team 27	91	88	3	0	97%
	Team 28	72	63	9	0	88%
	Team 29	80	64	16	0	80%
	Team 30	98	95	2	2	97%
	Team 31	60	58	0	2	97%
	Team 32	109	65	26	18	60%
	Team 33	100	70	28	2	70%
	Team 34	86	39	47	0	45%
	Team 35	98	94	4	0	96%
	Team 36	86	64	22	0	74%
	Team 37	72	69	3	0	96%
Average		87	70	15	2	81%

Table 5.9: Team Members Productivity and Effectiveness

	Team	FP Fulfilled per Team Member	% of Fulfilled Critical FP
SSP	Team 1	16.33	100.00%
	Team 2	10.67	91.00%
	Team 3	20.38	100.00%
	Team 4	13.57	100.00%
	Team 5	11.33	100.00%
	Team 6	15.87	88.00%
	Team 7	13.11	100.00%
	Team 8	18.65	100.00%
	Team 9	16.22	100.00%
	Team 10	11.75	100.00%
	Team 11	28.80	100.00%
	Team 12	20.93	100.00%
	Team 13	24.14	74.00%
	Team 14	16.69	88.00%
	Team 15	21.88	100.00%
	Team 16	17.34	100.00%
	Team 17	16.59	71.00%
	Team 18	18.23	100.00%
	Team 19	14.79	65.00%
	Team 20	15.16	100.00%
	Team 21	14.27	100.00%
	Team 22	11.36	100.00%
	Team 23	18.28	88.00%
Average		16.36	92.43%
EduProcess	Team 24	10.98	86.00%
	Team 25	14.24	100.00%
	Team 26	13.46	100.00%
	Team 27	12.87	100.00%
	Team 28	10.11	100.00%
	Team 29	11.33	100.00%
	Team 30	16.21	100.00%
	Team 31	12.32	100.00%
	Team 32	18.42	100.00%
	Team 33	14.54	100.00%
	Team 34	14.08	100.00%
	Team 35	13.97	100.00%
	Team 36	14.34	100.00%
	Team 37	12.19	100.00%
Average		13.50	99.50%

Table 5.10: Statistical Analysis of FP

Item	SSP		EduProcess		t-test(t-value)	Sig(2-tailed)	Cohen d
	Mean	Std. Deviation	Mean	Std. Deviation			
FP	87.75	27.34	86.64	14.13	0.14	0.78	0.05
FP Addressed	66.60	26.79	69.64	16.33	0.38	0.70	0.13
FP Not Addressed	19.90	17.46	4.79	13.12	0.95	0.34	0.33
FP Ambiguous	2.13	2.99	2.29	5.01	0.12	0.90	0.03
%FP Addressed	75.83	17.53	80.97	15.97	0.89	0.36	0.30
FP Fulfilled per Team Member	16.36	4.32	13.50	2.12	2.88	0.01	1.04
Critical FP Fulfilled	42.00	25.31	48.02	12.50	0.83	$<$0.01	0.50

On the other hand, with respect to the amount of critical FP fulfilled, the data shows that there is statistical difference. The results of teams that used EduProcess was (M=48.02, SD=12.50) and for the teams that did not use it (M=42.00, SD=25.31) considering t(36)=0.83. With a 0.05 of confidence interval, the mean difference was 6.02 to 7.50. For this, the average effect size was found to be d=0.50, which is considered of medium impact. These results indicate that EduProcess helped teams to maintain the focus on the critical aspects of the product being developed.

These findings were corroborated with the student and monitor opinions. A monitor reported: *"The problem with the monitoring sessions (part of EduProcess) is that they show the students what is mandatory in the project. Therefore, they work only on these aspects of the project"*. Table 5.9 presents the teams and the amount of software developed per team member. The teams that used EduProcess were able to build on average 13.50 FP per team member, while the teams that did not use EduProcess were able to achieve 16.36 FP per team member; more or less the same amount of code was written by teams that used EduProcess and SSP. However, in Table 5.11 is possible to see that teams that used EduProcess focused more on addressing the critical requirements.

When we look at the final results of each team regarding the software implemented, the results are not the same. Table 5.12 shows the software outcome of each team. The green cells correspond to successful projects, i.e., those that were able to put into production a software product that was useful for the client. Projects that did not reach such a goal were considered unsuccessful, and they were indicated with a cross and a red cell. The results show that the teams that used EduProcess almost always had a positive outcome; however, teams that did not use EduProcess have a major incidence of negative outcomes (i.e., positive and negative). All the teams that used EduProcess (except one) were able to deploy the software, and the ones that used the SSP had a 69.5% of deployment rate.

5.4 Discussion

In the introduction Chapter we presented two hypotheses: That the use of a prescriptive software process, like the EduProcess, for guiding development projects conducted in academic scenarios can:

- (H1) make these experiences repeatable while investing an affordable effort.
- (H2) help produce positive results in student software projects.

In this chapter, we presented a case study with multiple results that show that a prescriptive software process can help produce positive results in software projects. There are positive

Table 5.11: Team Productivity Detail

		Addressed			Not Addressed			Ambiguous		
	Team	**Critical**	**Desirable**	**Unnecessary**	**Critical**	**Desirable**	**Unnecessary**	**Critical**	**Desirable**	**Unnecessary**
SSP	Team 1	91	4	3	-	5	5	-	-	-
	Team 2	31	21	10	3	-	2	-	-	-
	Team 3	120	22	-	-	2	-	-	-	-
	Team 4	70	6	-	-	13	-	-	-	-
	Team 5	30	26	-	-	15	-	-	-	-
	Team 6	30	4	5	5	14	-	5	-	-
	Team 7	51	-	-	-	-	24	-	-	-
	Team 8	39	19	2	-	37	-	-	15	-
	Team 9	69	28	-	-	-	-	-	-	-
	Team 10	54	14	3	-	-	-	-	2	-
	Team 11	33	14	3	-	5	-	-	10	-
	Team 12	81	24	-	-	36	4	-	2	-
	Team 13	20	36	-	7	54	3	-	-	-
	Team 14	61	22	-	8	11	-	-	-	-
	Team 15	24	26	-	-	53	3	2	-	-
	Team 16	72	19	-	-	5	-	6	-	-
	Team 17	29	34	16	12	16	-	-	2	-
	Team 18	57	24	-	-	6	-	-	5	-
	Team 19	30	10	3	18	6	-	3	3	-
	Team 20	28	20	-	-	12	-	-	-	-
	Team 21	32	19	-	5	8	6	-	-	-
	Team 22	25	15	-	-	10	-	-	-	-
	Team 23	26	2	-	5	12	-	10	-	-
EduProcess	Team 24	45	12	-	-	-	-	-	-	-
	Team 25	55	33	-	-	6	5	-	-	-
	Team 26	48	17	2	-	2	15	4	4	-
	Team 27	49	39	-	-	3	-	-	-	-
	Team 28	50	13	-	-	3	6	-	-	-
	Team 29	17	36	11	-	-	16	-	-	-
	Team 30	33	44	18	-	-	2	-	3	-
	Team 31	37	21	-	-	-	-	-	2	-
	Team 32	58	-	7	-	-	26	-	18	-
	Team 33	29	40	1	-	5	23	-	2	-
	Team 34	36	3	-	-	22	25	-	-	-
	Team 35	47	38	9	-	-	4	-	-	-
	Team 36	36	23	6	-	-	21	-	-	-
	Team 37	61	8	-	-	3	-	-	-	-

Table 5.12: Project Deployment (meaning: ✗ - not deployed, ✓- deployed)

	Team	Project Result
SSP	Team 1	✓
	Team 2	✗
	Team 3	✓
	Team 4	✓
	Team 5	✓
	Team 6	✗
	Team 7	✓
	Team 8	✓
	Team 9	✓
	Team 10	✓
	Team 11	✓
	Team 12	✓
	Team 13	✗
	Team 14	✗
	Team 15	✓
	Team 16	✓
	Team 17	✗
	Team 18	✓
	Team 19	✗
	Team 20	✓
	Team 21	✓
	Team 22	✓
	Team 23	✗
EduProcess	Team 24	✗
	Team 25	✓
	Team 26	✓
	Team 27	✓
	Team 28	✓
	Team 29	✓
	Team 30	✓
	Team 31	✓
	Team 32	✓
	Team 33	✓
	Team 34	✓
	Team 35	✓
	Team 36	✓
	Team 37	✓

results directly related to the SE course in general: course and project grades. The use of EduProcess seems to improve different aspects of team marks (in order of relevance according to the Cohen d): coordination, sense of team belonging, commitment, engagement, initiative and communication.

We also showed that there is no significant difference in the amount of software built by the teams that used EduProcess and the teams that used SSP. However, we saw that there is a statistical significant difference in the amount of critical software requirements built by the teams that used EduProcess, which is the most important result from clients' point of view. This finding can be corroborated with the data related to software deployment: the deployment rate of the teams that used EduProcess is much higher than that of the teams that used SSP. Therefore, the results presented here support the first hypothesis because with similar effort (time spent), teams were able to develop more useful software for their clients.

To enhance the validity of our case study we presented quantitative and qualitative data. We have also shown data from different sources: peer-assessment (gathered by the instructional team in an anonymous fashion), end of semester survey (gathered by the university anonymously), course and project grades and project effort data (FP). This case study is therefore strengthened by the use of different sources of data, showing the same result patterns when using the triangulation strategy for case studies. This gives strengths to our findings and supports the stated hypotheses.

This case study shows that it is possible to use EduProcess more than once with the same pattern of results. This indicates that EduProcess is repeatable with an affordable effort, which supports our first hypothesis. Furthermore, with the data presented in the case study we showed statistical evidence that EduProcess helps produce positive results in software engineering project-based courses. To be more specific, the positive results are related to course grades, project grades and teamwork skills (communication, initiative, engagement, commitment, sense of team belonging and coordination).

5.5 Threats to Validity

The threats to validity of our case study are analyzed according to the taxonomy and methodology proposed by Yin [175]: construct validity, reliability, internal validity and external validity.

5.5.1 Construct Validity

According to Yin [175], this aspect of validity is concerned with the extent to which the concepts being studied really represent what the researcher claims to study. In order to address this concern, EduProcess was created after an extensive study, where we searched for approaches that were being used to teach software engineering, processes that were being used to teach software engineering in practice, and also best practices for teaching software engineering. Additionally, we also researched the state-of-the-practice Chilean SE education, trying to answer several questions, e.g., what software engineering courses do these programs have, how they were being taught and how they were being evaluated.

5.5.2 Reliability

This aspect is concerned with the fact that the case study can be repeated by other researchers, obtaining the same or a similar result. In this sense, we provide a detailed process that allows other people to repeat these experiences. EduProcess is fully documented (with examples) and available online (http://www.dcc.uchile.cl/∼mmarques/), so any researcher can use it, but we recognize the fact that we have not yet implemented it at another university as a threat.

5.5.3 Internal Validity

This aspect is concerned with how well our case study was conducted, especially whether it avoids confusing more than one possible independent cause working at the same time. Since the instructor was the same in all twelve semesters, the threat is minimized in this aspect therefore, more monitors were hired and they followed the EduProcess Instructor process.

On the other hand, although EduProcess is fully documented, it is a process that needs to be used by the students and the instructional team, therefore there are some concerns related to the human aspects of the process. In this sense, there may be a threat related to the quality of the data collected from the students' peer-assessment. We expect that students fill peer-assessments in a fair manner, but this is not always the case. In order to mitigate this threat, we removed some outlying opinions and corroborated the results using the monitors' records.

5.5.4 External Validity

This aspect of validity is concerned to findings generalization. The evaluation process of this case study captured the reality of a particular SE project-based course. It is possible that the scenario we observed is specific to the Computer Science Department or the Engineering School of the Universidad de Chile. Conducting similar experiences at other universities may not have the same results, but we believe that they follow the same pattern. Replicating the experience in different scenarios would help assess whether these results can be generalized.

Chapter 6

Conclusions and Future Work

In this Chapter we present a summary of this thesis, its conclusions, contributions, restrictions to the solution. We finish the chapter with a discussion of future work.

6.1 Summary of the Thesis Work

Before creating EduProcess we extensively analyzed SE education in general with a mapping review, a systematic literature review on SE education best practices and also a field study on how software engineering is taught in Chile. Moreover, we analyzed the previous experiences in a project-based course to identify aspects that could be improved.

In these previous studies we found some processes for supporting project-based courses, however none of them have enough detail to allow instructors to follow these processes. Therefore, these proposals cannot be considered as potential prescriptive solutions able to support these experiences in a repeatable way. This thesis work presents EduProcess; a prescriptive software process for supporting small software teams working in a loosely-coupled way in SE undergraduate project-based courses, which is a practical experience that allows students to get closer to industry reality. This process involves two parallel tracks: one that is in charge of the students (and involves the users and clients) and the other that supports the activities of the instructional team (instructor, teaching assistants and monitors). In the first track the students assume specific roles as part of the development team, and the activities are focused on obtaining a software product that helps clients and users deal with the problem being addressed. These practical experiences help students put in practice the theoretical knowledge gathered in previous SE courses, and realize the different aspects involved in a software project. The second track is the instructional process, where the instructional team will find support to help students learn and take advantage of these experiences.

The two tracks of EduProcess (for the students and for the instructional team) were extensively and completely formalized in EPF Composer, and both processes are available online with examples and guidelines for use by any instructor (http://www.dcc.uchile.cl/∼mmarques/). These tracks have intertwined activities, for instance, through the product reviews, the monitoring of the development process, and the work sessions performed jointly between the software team and its clients and users. Therefore, these tracks have to maintain their coordination between them.

This thesis hypothesizes that the proposed software process can: (H1) make the SE project-based experiences repeatable investing an affordable effort, and also (H2) help student teams produce positive results in their software projects. In order to evaluate these hypotheses, we conducted a case study that considered the projects run during the last twelve semesters in a SE project-based course, which is part of the undergraduate computer science engineering program taught at the Universidad de Chile. The obtained results support the stated hypotheses.

6.2 Conclusions

In our prior analysis of software engineering education in Chile, we saw that on average, computer science undergraduate programs have a little more than three software engineering courses (3.3), they typically have a theoretical, a project-based and a capstone course. They are taught starting mid-program and they follow the logical learning pattern from theoretical to practical instructional approach.

The results of the case study, focused on a project-based course, indicate that EduProcess helps students increase their coordination, sense of team belonging, commitment and effectiveness, but not necessarily their productivity. Nevertheless, EduProcess allows teams to more accurately diagnose and prioritize their projects so they are able to achieve a higher deployment rate. The study results also show that EduProcess improves team results in terms of course grades and project grades. This evidence supports the second hypothesis.

Moreover, many teams that did not use EduProcess failed in determining the team and project challenges that had to be addressed in their development projects. We believe that the lack of a formal reflection space delayed the problem identification, and therefore most teams tended to react late to the problems. EduProcess seems to make team members aware of their project status and planning, as well as team member engagement and behavior. This opens space for discussions and analysis that finally enhance these learning experiences and make teams more coordinated and effective, which also supports the second hypothesis.

On the other hand, we should also consider the impact that a defined process has on the instructors and students' activities. A guideline like EduProcess helps the involved people to know exactly what to do and when, eliminating wasted time of trying to determine what they have to do next. Teams that used EduProcess had a set of defined activities that included examples and guidelines. This made the process more affordable, but also repeatable for the participants in the course. This result supports the first hypothesis.

After using EduProcess during four semesters and obtaining positive results in a recurrent way (and also results higher than when SSP was used), we can say that this process can be repeated spending an affordable effort. The effort of using EduProcess is similar to that required by SSP, which have already shown to be repeatable. These results support the first hypothesis (H1).

Therefore, we can say that the case study results are aligned with both stated hypotheses. Having positive outcomes in software projects performed in the academia is not frequent, but it is highly important for future software engineers. In this sense, EduProcess represents a useful instrument to help students and instructors reach such a goal. This is particularly important given the lack of formalized proposals for supporting software development in

project-based courses.

6.3 Contributions

In this thesis we have three main contributions to the state-of-the-art: (1) a review of the software engineering education from various perspectives, (2) EduProcess, a formalized prescriptive software process that can be re-used by others based on materials published online, and (3) the Reflexive Weekly Monitoring Process. Although this last process is part of EduProcess, it was designed from scratch providing an alternative to the current instructional proposals for supporting students, like coaching and the use of a Scrum master. It can be used separately from EduProcess as a way to help students to improve teamwork in a more general way.

Concerning the first contribution, we conducted a systematic mapping that sheds light on how the practical experiences in undergraduate software engineering courses are being addressed in several scenarios. Moreover, considering the best practices proposed by the 2014 ACM-IEEE Software Engineering Curriculum Guideline, we analyzed the state-of-the-practice in SE education in Chile. As a result we developed a status report that shows what and how we are teaching software engineering in Chile, while identifying strengths and weaknesses. These results should help institutions and SE instructors self-evaluate their courses and identify opportunities for improvement.

Concerning the second contribution, EduProcess represents the first recipe to support project-based courses, which can be reused not only by other instructors and students, but also has proved to produce positive results in a consistent way. Recognizing the variety of work contexts of these courses, more evaluation is required to determine the impact of EduProcess in all these contexts. However, this proposal represents a first step that helps other instructors or researchers propose new processes based on it.

The third contribution, the Reflexive Weekly Monitoring (RWM) process, is an alternative to other approaches already mentioned (coaching, Scrum master, etc.). RWM has shown very positive results, for instance, helping team members put their effort on what really matters, minimizing the internal friction among team members and increasing the sense of team belonging. It also lets students know they are not drifting away, giving them confidence on what they are doing makes them more aware of their own work, and they can therefore work in a more coordinated and effective way. In this sense, RWM is an important component of EduProcess that recurrently helps instructors and students get positive results.

6.4 Future Work

Although we will continue using EduProcess at Universidad de Chile, we intend to have instructors from other universities (with other contexts) using EduProcess to support their SE project-based courses. There are also plans to use it at the Universidad de Santiago de Chile (USACH) during the second semester 2017, and maybe in the future, at Universidad de O'Higgins.

We also want to improve the use of EduProcess by connecting the specified process with a software manager tool (such as RedMine) that helps students deal with the task allocation and

monitoring. Going a little bit further we also want to connect EduProcess to the repository of the code being used by the teams (Bitbucket). The connection between the software management tool, the code repository and EduProcess will give us further information about where the weaknesses are, and how we can overcome these limitations.

The thesis author has also applied for a Post-Doctorate Fellowship to CONICYT (currently under evaluation) in order to make a more in depth diagnosis of the software engineering education in Chile, and propose additional instruments for helping improve such a reality. This is also slated for future work.

Bibliography

[1] Alain Abran, Pierre Bourque, Robert Dupuis, and James W. Moore. *Guide to the Software Engineering Body of Knowledge-SWEBOK*. IEEE Press, 2001.

[2] Simon L. Albrech. Handbook of Employee Engagement: Perspectives, Issues, Research and Practice. *Human Resource Management International Digest*, 19(7), 2011.

[3] Eric Allen, Robert Cartwright, and Charles Reis. Production Programming in the Classroom. In *ACM SIGCSE Bulletin*, volume 35, pages 89–93. ACM, 2003.

[4] James H. Andrews and Hanan L. Lutfiyya. Experiences with a Software Maintenance Project Course. *IEEE Transactions On Education*, 43(4):383–388, 2000.

[5] Delbert Bailey, Tracey Conn, Brian Hanks, and Linda Werner. Can we Influence Students' Attitudes About Inspections? Can We Measure a Change in Attitude? In *Proc. of the International Conference on Software Engineering Education and Training (CSEE&T)*, pages 260–267. IEEE, 2003.

[6] Marini Abu Bakar, Norleyza Jailani, Zarina Shukur, and Noor Faezah Mohd Yatim. Final Year Supervision Management System as a Tool for Monitoring Computer Science Projects. *Procedia - Social and Behavioral Sciences*, 18:273 – 281, 2011. Kongres Pengajaran dan Pembelajaran UKM, 2010.

[7] Ray Bareiss and Martin Griss. A Story-Centered, Learn-by-Doing Approach to Software Engineering Education. *ACM SIGCSE Bulletin*, 40(1):221–225, March 2008.

[8] Ray Bareiss and Todd Sedano. A Gentle Introduction to Learn by Doing. In *Proc. of the International Conference on Software Engineering Education and Training (CSEE&T)*, pages 81–84. IEEE, 2012.

[9] Marques M Bastarrica C, Perovich D. What can Students Get from a Software Engineering Capstone Course? In *to be presented at Proc. of the International Conference on Software Engineering (ICSE)*, 2017.

[10] Jessica D. Bayliss and Sean Strout. *Games as a Flavor of CS1*, volume 38. ACM, 2006.

[11] Andrew Begel and Beth Simon. Novice Software Developers, All Over Again. In *Proc. of the International Workshop on Computing Education Research*, pages 3–14. ACM, 2008.

[12] Lawrence Bernstein and David Klappholz. Getting Software Engineering Into Our Guts. *CrossTalk: The Journal of Defense Software Engineering*, pages 25–26, 2001.

[13] Phyllis C. Blumenfeld, Elliot Soloway, Ronald W. Marx, Joseph S. Krajcik, Mark Guzdial, and Annemarie Palincsar. Motivating Project-Based Learning: Sustaining the Doing, Supporting the Learning. *Educational Psychologist*, 26(3-4):369–398, 1991.

[14] Barry Boehm and Daniel Port. Educating Software Engineering Students to Manage Risk. In *Proc. of the International Conference on Software Engineering (ICSE)*, pages 591–600. IEEE Computer Society, 2001.

[15] Maura Borrego, Jennifer Karlin, Lisa D. McNair, and Kacey Beddoes. Team Effectiveness Theory from Industrial and Organizational Psychology Applied to Engineering Students Project Teams: A Research Review. *Journal of Engineering Education*, 102(4):472–512, 2013.

[16] David Broman. Should Software Engineering Projects be the Backbone or the Tail of Computing Curricula? In *Proc. of the International Conference on Software Engineering Education and Training (CSEE&T)*, pages 153–156. IEEE, 2010.

[17] David Broman, Kristian Sandahl, and Mohamed Abu Baker. The Company Approach to Software Engineering Project Courses. *IEEE Transactions On Education*, 2011.

[18] Cristopher N. Bull, Jon Whittle, and Leon Cruickshank. Studios in Software Engineering Education: Towards an Evaluable Model. In *Proc. of the International Conference on Software Engineering (ICSE)*, pages 1063–1072. IEEE, 2013.

[19] Christian Bunse, Raimund L. Feldmann, and J. Dörr. Agile Methods in Software Engineering Education. *Agile Processes in Software Engineering*, 2004.

[20] Nergiz Ercil Cagiltay. Teaching Software Engineering by Means of Computer-Game Development: Challenges and Opportunities. *British Journal of Educational Technology*, 38(3):405–415, 2007.

[21] Nicholas L. Carroll, Lina Markauskaite, and Rafael A. Calvo. E-portfolios for Developing Transferable Skills in a Freshman Engineering Course. *IEEE Transactions On Education*, 50(4):360–366, 2007.

[22] Jeffrey Carver, Letizia Jaccheri, Sandro Morasca, and Forrest Shull. Issues in Using Students in Empirical Studies in Software Engineering Education. In *Proc. International Software Metrics Symposium*, pages 239–249, Sept 2003.

[23] Chung-Yang Chen and P. Pete Chong. Software Engineering Education: A Study on Conducting Collaborative Senior Project Development. *Journal of Systems and Software*, 84(3):479–491, 2011.

[24] Jianguo Chen, Huijuan Lu, Lixin An, and Yongxia Zhou. Exploring Teaching Methods in Software Engineering Education. In *Computer Science & Education, 2009. ICCSE'09. 4th International Conference on*, pages 1733–1738. IEEE, 2009.

[25] Yung-Pin Cheng and Janet Mei-Chuen Lin. A Constrained and Guided Approach for Managing Software Engineering Course Projects. *IEEE Transactions on Education*, 53(3):430–436, Aug 2010.

[26] Oumout Chouseinoglou and Semih Bilgen. Introducing Critical Thinking to Software Engineering Education. In *Software Engineering Research, Management and Applications*, pages 183–195. Springer, 2014.

[27] Kajal Claypool and Mark Claypool. Teaching Software Engineering through Game Design. *ACM SIGCSE Bulletin*, 37(3):123–127, 2005.

[28] Alistair Cockburn. *Crystal Clear: a Human-Powered Methodology for Small Teams*. Pearson Education, 2004.

[29] César A. Collazos, Sergio F. Ochoa, Sergio Zapata, Fáber D. Giraldo, María Inés Lund, Laura Aballay, and Gisela Torres de Clunie. CODILA:A Collaborative and Distributed Learning Activity Applied to Software Engineering Courses in Latin American Universities. In *Proc. of the International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom)*, pages 1–9, 2010.

[30] James S. Collofello and Marla Hart. Monitoring Team Progress in a Software Engineering Project Class. In *Proc. of the Frontiers in Education (FIE)*, volume 1, pages 11B4/7–11B410 vol.1. IEEE, Nov 1999.

[31] Eclipse Community. UP. `http://epf.eclipse.org/wikis/openup/`, April 2012.

[32] Magister en Ciencias Mención Computación and Luis Gregorio Silvestre Quiroga. Diseño de Equipos de Desarrollo de Software en Escenarios Universitarios. 2012.

[33] David Coppit and Jennifer M. Haddox-Schatz. Large Team Projects in Software Engineering Courses. *ACM SIGCSE Bulletin*, 2005.

[34] Cyril Coupal and Kelvin Boechler. Introducing Agile into a Software Development Capstone Project. In *Agile Conference*, pages 289–297. IEEE Comput. Soc, 2005.

[35] Tony Cowling, Shawn Bohner, and Mark A. Ardis, editors. *Proc. of the International Conference on Software Engineering Education and Training (CSEE&T)*. IEEE, 2013.

[36] CRUCH. Consejo de Rectores de Chile. `http://www.consejoderectores.cl`, January 2015.

[37] Ralph B D'agostino, Albert Belanger, and Ralph B D'Agostino Jr. A suggestion for using powerful and informative tests of normality. *The American Statistician*, 44(4):316–321, 1990.

[38] Deepak Dahiya. Teaching Software Engineering: a Practical Approach. *ACM SIGSOFT Software Engineering Notes*, 35(2):1–5, 2010.

[39] Bart A. De Jong and Tom Elfring. How Does Trust Affect the Performance of Ongoing

Teams? The Mediating Role of Reflexivity, Monitoring, and Effort. *The Academy of Management Journal*, 53(3):535–549, 2010.

[40] Wilson de Pádua Paula Filho. *Engenharia de Software*, volume 2. LTC, 2003.

[41] Peter J. Denning. Educating a New Engineer. *Commun.ACM*, 35(12):82–97, 1992.

[42] Walter Dick, Lou Carey, and James O Carey. The Systematic Design of Instruction. 2006.

[43] Daniel Dietsch, Andreas Podelski, Jaechang Nam, Pantelis M. Papadopoulos, and Martin Schäf. Monitoring Student Activity in Collaborative Software Development. *CoRR*, abs/1305.0787, 2013.

[44] Anke Drappa and Jochen Ludewig. Simulation in Software Engineering Training. In *Proc. of the International Conference on Software Engineering (ICSE)*, pages 199–208. ACM, 2000.

[45] Heidi J. C. Ellis, Ralph A. Morelli, Trishan R. de Lanerolle, and Gregory W. Hislop. Holistic Software Engineering Education Based on a Humanitarian Open Source Project. In *Proc. of the International Conference on Software Engineering Education and Training (CSEE&T)*, pages 327–335. IEEE Computer Society, 2007.

[46] Michel Ferrari, Roger Taylor, and Kurt VanLehn. Adapting Work Simulations for Schools. *Journal of Educational Computing Research*, 21(1):25–53, 1999.

[47] Pierre Flener. Realism in Project-Based Software Engineering Courses: Rewards, Risks, and Recommendations. In *International Conference on Computer and Information Sciences*. Springer-Verlag, November 2006.

[48] The Eclipse Foundation. Eclipse Process Framework Project (EPF). `https://eclipse.org/epf/`, 2015.

[49] Armando Fox and David Patterson. Crossing the Software Education Chasm. *Communications of the ACM*, 55(5):44–49, 2012.

[50] Gerald C. Gannod, Kristen M. Bachman, Douglas Troy, Steve D. Brockman, et al. Increasing Alumni Engagement Through the Capstone Experience. In *Proc. of the Frontiers in Education (FIE)*, pages F1C–1. IEEE, 2010.

[51] Gerald C. Gannod, Janet E. Burge, and Michael T. Helmick. Using the Inverted Classroom to Teach Software Engineering. In *Proc. of the International Conference on Software Engineering (ICSE)*, pages 777–786. ACM, 2008.

[52] Ivan A. Garcia and Carla L. Pacheco. Using TSPi and PBL to Support Software Engineering Education in an Upper-Level Undergraduate Course. *Computer Applications in Engineering Education*, 22(4):736–749, 2014.

[53] Kevin Gary, Tommie Lindquist, Sunny Bansal, and Arbi Ghazarian. A Project Spine

for Software Engineering Curricular Design. In Cowling et al. [35], pages 299–303.

[54] Éric Germain and Pierre N. Robillard. Towards Software Process Patterns: An Empirical Analysis of the Behavior of Students Teams. *Information and Software Technology*, 50(11):1088–1097, 2008.

[55] Éric Germain, Pierre N. Robillard, and Mihaela Dulipovici. Process Activities in a Project Based Course in Software Engineering. In *Proc. of the Frontiers in Education (FIE)*, volume 3, pages S3G–7. IEEE, 2002.

[56] George G.Mitchell and James Declan Delaney. An Assessment Strategy to Determine Learning Outcomes in a Software Engineering Problem-Based Learning Course. *International Journal of Engineering Education*, 20(3):494–502, 2004.

[57] Narasimhaiah Gorla and Yan Wah Lam. Who Should Work With Whom?: Building Effective Software Project Teams. *Communications of the ACM*, 47(6):79–82, 2004.

[58] Dennis P. Groth and Edward L. Robertson. It's All About Process: Project-Oriented Teaching of Software Engineering. In *Proc. of the International Conference on Software Engineering Education and Training (CSEE&T)*, pages 7–17. IEEE, 2001.

[59] Michael Halling, Wolfgang Zuser, Monika Köhle, and Stefan Biffl. Teaching the Unified Process to Undergraduate Students. In *Proc. of the International Conference on Software Engineering Education and Training (CSEE&T)*, pages 148–159. IEEE, 2002.

[60] Lile Hattori, Alberto Bacchelli, Michele Lanza, and Mircea Lungu. Erase and Rewind Learning by Replaying Examples. In *Proc. of the International Conference on Software Engineering Education and Training (CSEE&T)*, pages 558–558. IEEE, 2011.

[61] Jane Huffman Hayes, Timothy C. Lethbridge, and Daniel Port. Evaluating Individual Contribution Toward Group Software Engineering Projects. In *Proc. of the International Conference on Software Engineering (ICSE)*, pages 622–627. IEEE Computer Society, 2003.

[62] Orit Hazzan and Yael Dubinsky. Teaching a Software Development Methodology: the Case of eXtreme Programming. In *Proc. of the International Conference on Software Engineering Education and Training (CSEE&T)*, pages 176–184. IEEE Computer Society, 2003.

[63] Thowas B. Hilburn and Dondd J. Bagert. A Software Engineering Curriculum Model. In *Proc. of the Frontiers in Education (FIE)*, volume 1, pages 12A4–6. IEEE, 1999.

[64] Watts S. Humphrey. *A Discipline for Software Engineering*. Addison Wesley, 1995.

[65] Watts S. Humphrey. Using a Defined and Measured Personal Software Process. *IEEE Software*, 13(3):77, 1996.

[66] Watts S. Humphrey. *Team Software Process (TSP)*. Wiley Online Library, 2000.

[67] Letizia Jaccheri and Thomas Østerlie. Open Source Software: A Source of Possibilities for Software Engineering Education and Empirical Software Engineering. In *Proc. of the International Workshop on Emerging Trends in FLOSS Research and Development (FLOSS)*, pages 5–5. IEEE, 2007.

[68] Ivar Jacobson, Pan-Wei Ng, Paul E. McMahon, Ian Spence, and Svante Lidman. *The Essence of Software Engineering: Applying the SEMAT Kernel.* Addison-Wesley, 2013.

[69] Stan Jarzabek. Teaching Advanced Software Design in Team-Based Project Course. In Cowling et al. [35], pages 31–40.

[70] Curt Jones. Using Subversion As an Aid in Evaluating Individuals Working on a Group Coding Project. *J. Comput. Sci. Coll.*, 25(3):18–23, January 2010.

[71] Gareth R. Jones. Task Visibility, Free Riding, and Shirking: Explaining the Effect of Structure and Technology on Employee Behavior. *Academy of Management Review*, 9(4):684–695, 1984.

[72] Judy Kay, Nicolas Maisonneuve, Kalina Yacef, and Osmar Zaïane. Mining Patterns of Events in Students' Teamwork Data. In *In Educational Data Mining Workshop, held in conjunction with Intelligent Tutoring Systems (ITS)*, pages 45–52, 2006.

[73] Karen Keefe and Martin Dick. Using Extreme Programming in a Capstone Project. In *Proc. of the Australasian Conference on Computing Education*, pages 151–160. Australian Computer Society, Inc., 2004.

[74] Staffs Keele. Guidelines for Performing Systematic Literature Reviews in Software Engineering. In *Technical Report, Ver. 2.3 EBSE Technical Report. EBSE*. 2007.

[75] Barbara Kitchenham and Stuart Charters. Guidelines for Performing Systematic Literature Reviews in Software Engineering. Technical Report EBSE 2007-01, Keele University and Durham University, July 2007.

[76] Sami Kollanus. Test-driven Development Still a Promising Approach? In *Proc. of the International Quality of Information and Communications Technology (QUATIC)*, pages 403–408. IEEE, 2010.

[77] Clóvis Luís Konopka, Martha Bohrer Adaime, and Pedro Henrique Mosele. Active Teaching and Learning Methodologies: Some Considerations. *Creative Education*, 6(14):1536, 2015.

[78] Supannika Koolmanojwong and Barry Boehm. Using Software Project Courses to Integrate Education and Research: An Experience Report. In *Proc. of the International Conference on Software Engineering Education and Training (CSEE&T)*, pages 26–33. IEEE, 2009.

[79] Supannika Koolmanojwong and Barry Boehm. A Look at Software Engineering Risks in a Team Project Course. In *Proc. of the International Conference on Software Engineering Education and Training (CSEE&T)*, pages 21–30. IEEE, 2013.

[80] Christian Köppe. Teaching Software Process with OpenUP. 2010.

[81] Richard J. LeBlanc, Ann Sobel, Jorge L Diaz-Herrera, Thomas B. Hilburn, et al. *Software Engineering 2004: Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering.* IEEE Computer Society, 2006.

[82] Timo O. A. Lehtinen, Mika V. Mäntylä, Juha Itkonen, and Jari Vanhanen. Diagrams or Structural Lists in Software Project Retrospectives–An Experimental Comparison. *Journal of Systems and Software*, 103:17–35, 2015.

[83] Timothy C. Lethbridge, Jorge Diaz-Herrera, Richard J. LeBlanc Jr., and J. Barrie Thompson. Improving Software Practice Through Education: Challenges and Future Trends. In *Proc. of the Future of Software Engineering (FOSE)*, pages 12–28. IEEE, 2007.

[84] Rensis Likert. A Technique for the Measurement of Attitudes. *Archives of Psychology*, 1932.

[85] Liana Barachisio Lisboa, Vinicius Cardoso Garcia, Daniel Lucrédio, Eduardo Santana de Almeida, Silvio Romero de Lemos Meira, and Renata Pontin de Mattos Fortes. A Systematic Review of Domain Analysis Tools. *Information and Software Technology*, 52(1):1–13, January 2010.

[86] Chang Liu. Enriching Software Engineering Courses with Service-Learning Projects and the Open-Source Approach. In *Proc. of the International Conference on Software Engineering (ICSE)*, pages 613–614. ACM, 2005.

[87] Michael J. Lutz, J. Fernando Naveda, and James R. Vallino. Undergraduate Software Engineering. *Communications of the ACM*, 57(8):52–58, 2014.

[88] Jing Ma and Jeffrey V. Nickerson. Hands-on, Simulated, and Remote Laboratories: A Comparative Literature Review. *ACM Computing Surveys (CSUR)*, 38(3):7, 2006.

[89] José A. Macías. Enhancing Project-Based Learning in Software Engineering Lab Teaching Through an E-portfolio Approach. *IEEE Transactions On Education*, 55(4):502–507, 2012.

[90] Viljan Mahnic. A Capstone Course on Agile Software Development Using Scrum. *IEEE Transactions On Education*, 55(1):99–106, 2012.

[91] Viljan Mahnic and Anze Casar. A Computerized Support Tool for Conducting a Scrum-Based Software Engineering Capstone Course. *International Journal of Engineering Education*, 32(1):278–293, 2016.

[92] Thom Markham. Project Based Learning A Bridge Just Far Enough. *Teacher Librarian*, 39(2):38, 2011.

[93] Michelle A. Marks, John E. Mathieu, and Stephen J. Zaccaro. A Temporally Based Framework and Taxonomy of Team Process. *Academy of Management Review*, 26(3):356–

376, 2001.

[94] Michelle A. Marks and Frederick J. Panzer. The Influence of Team Monitoring on Team Processes and Performance. *Human Performance*, 17(1):25–41, 2004.

[95] Maíra Marques. Analyzing Gender Difference in Improving Self-Efficacy on Software Engineering Education. In *ACM-SRC, Grace Hopper*, pages 1–8. ACM, 2014.

[96] Maíra Marques. A Prescriptive Software Process for Academic Scenarios. In *ICER - Doctoral Consortium*. ACM, 2015.

[97] Maíra Marques. Software Engineering Education - Does gender matter in Project results? A Chilean Case Study. In *Proc. of the Frontiers in Education (FIE)*, pages 1–8. IEEE, 2015.

[98] Maíra Marques. Monitoring - An Intervention to Improve Team Results in Software Engineering Education. In *ACM-SRC, SIGSE, First Place - Graduate Competition*. ACM, 2016.

[99] Maíra Marques, Cecilia Bastarrica, and Sergio F. Ochoa. Software Engineering Education in Chile - Status Report. In *ITICSE*. ACM, 2016.

[100] Maíra Marques and Sergio F. Ochoa. Transferring Software Development Knowledge and Skills in the Academia: A Literature Review. Technical report, Technical Report TR/DCC-2013-2, Computer Science Department, University of Chile, 2013.

[101] Maíra Marques and Sergio F. Ochoa. Improving Teamwork in Students Software Projects. In *Proc. of the International Conference on Software Engineering Education and Training (CSEE&T)*, pages 99–108. IEEE, 2014.

[102] Maíra Marques, Alcides Quispe, and Sergio F. Ochoa. A Systematic Mapping Study on Practical Approaches to Teaching Software Engineering. In *Proc. of the Frontiers in Education (FIE)*, pages 1–8. IEEE, 2014.

[103] Gregory S. Mason, Teodora Rutar Shuman, and Kathleen E. Cook. Comparing the Effectiveness of an Inverted Classroom to a Traditional Classroom in an Upper-Division Engineering Course. *IEEE Transactions On Education*, 56(4):430–435, 2013.

[104] Miguel J. Monasor, Aurora Vizcaíno, and Mario Piattini. A Tool for Training Students and Engineers in Global Software Development Practices. In *International Conference on Collaboration and Technology*, pages 169–184. Springer, 2010.

[105] Miguel J. Monasor, Aurora Vizcaíno, and Mario Piattini. An Educational Environment for Training Skills for Global Software Development. In *2010 10th IEEE International Conference on Advanced Learning Technologies*, pages 99–101. IEEE, 2010.

[106] Miguel J. Monasor, Aurora Vizcaíno, and Mario Piattini. VENTURE: Towards a Framework for Simulating GSD in Educational Environments. In *Research Challenges in Information Science (RCIS), 2011 Fifth International Conference on*, pages 1–10.

IEEE, 2011.

[107] Melody Moore and Colin Potts. Learning by Doing: Goals and Experiences of two Software Engineering Project Courses. *Software Engineering Education*, pages 151–164, 1994.

[108] Melody M. Moore and Terence Brennan. Process Improvement in the Classroom. In *Software Engineering Education*, pages 121–130. Springer, 1995.

[109] Ana M. Moreno, Maria-Isabel Sanchez-Segura, Fuensanta Medina-Dominguez, and Laura Carvajal. Balancing Software Engineering Education and Industrial Needs. *Journal of Systems and Software*, 85(7):1607–1620, 2012.

[110] Debora Maria Nascimento, Ken Cox, Telmo Almeida, Wendell Sampaio, Roberto Almeida Bittencourt, Richard Souza, and Christina Chavez. Using Open Source Projects in Software Engineering Education: A Systematic Mapping Study. In *Proc. of the Frontiers in Education (FIE)*, pages 1837–1843. IEEE, 2013.

[111] Emily Oh Navarro, Alex Baker, and André Van Der Hoek. Teaching Software Engineering Using Simulation Games. In *Proc. of the International Conference on Simulation in Education*. Citeseer, 2004.

[112] Crescencio Rodrigues Lima Neto and Eduardo Santana De Almeida. Five years of lessons learned from the software engineering course: Adapting best practices for distributed software development. In *Proc. of the International Workshop on Collaborative Teaching of Globally Distributed Software Development*, pages 6–10. IEEE Press, 2012.

[113] Duc Man Nguyen, Tien Vu Truong, and Nguyen Bao Le. Deployment of Capstone Projects in Software Engineering Education at Duy Tan University as Part of a University-Wide Project-Based Learning Effort. In *Learning and Teaching in Computing and Engineering*, pages 184–191. IEEE, 2013.

[114] Sergio F. Ochoa, José A. Pino, and D. Andrade. Strategies to Estimate the Development Effort of Web Projects in Immature Scenarios. *Revista de la Pontificia Universidad Católica del Ecuador*, (81):125–171, 2007. In Spanish.

[115] Sergio F. Ochoa, José A. Pino, Luis A. Guerrero, and César A. Collazos. SSP:A Simple Software Process for Small-Size Software Development Projects. volume 219 of *IFIP*, pages 94–107. Springer, 2006.

[116] Jeff Offutt. Putting the Engineering into Software Engineering Education. *Software, IEEE*, 30(1):96–96, 2013.

[117] Emily Oh and André van der Hoek. Adapting Game Technology to Support Individual and Organizational Learning. In *Proc. of SEKE*, pages 347–362. Citeseer, 2001.

[118] Lennart Ohlsson and Conny Johansson. A Practice Driven Approach to Software Engineering Education. *IEEE Transactions On Education*, 38(3):291–295, 1995.

[119] Joint Task Force on Computing Curricula Association for Computing Machinery/IEEE Computer Society. Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering. `http://www.acm.org/binaries/content/assets/Education/se2014.pdf`, February 2015.

[120] Joint Task Force on Computing Curricula IEEE Computer Society Association for Computing Machinery. Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering. `http://sites.Computer.org/ccse/`, January 2004.

[121] Joint Task Force on Computing Curricula IEEE Computer Society Association for Computing Machinery. Curriculum Guidelines for Undergraduate Degree Programs in Computer Science. `http://www.acm.org/Education/CS2013-final-Report.pdf`, 2013.

[122] David Lorge Parnas. Software Engineering Programs are not Computer Science Programs. *Software, IEEE*, 16(6):19–30, 1999.

[123] Wilson P. Paula Filho. Requirements for an Educational Software Development Process. In *ACM SIGCSE Bulletin*, volume 33, pages 65–68. ACM, 2001.

[124] Wilson P. Paula Filho. A Process-Based Software Engineering Course: Some Experimental Results. *Proc. of the JIISIC-3as. Jornadas Iberoamericanas de Ingeniería de Software e Ingeniería del Conocimiento*, 2003.

[125] Wilson P. Paula Filho. A Model-driven Software Process for Course Projects. In *Proc. of the Educators' Symposium of the ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pages 33–40. Citeseer, 2005.

[126] Kai Petersen, Robert Feldt, Shahid Mujtaba, and Michael Mattsson. Systematic Mapping Studies in Software Engineering. In *Proc. of the International Conference on Evaluation and Assessment in Software Engineering*, volume 17. sn, 2008.

[127] Wouter Poncin, Alexander Serebrenik, and Mark van den Brand. Mining Student Capstone Projects with FRASR and ProM. In *Proc. of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA, pages 87–96, New York, NY, USA, 2011. ACM.

[128] Roger S. Pressman. *Software Engineering: a Practitioner's Approach.* Palgrave Macmillan, 2005.

[129] Teade Punter, René L. Krikhaar, and Reinder J. Bril. Software Engineering Technology Innovation–Turning Research Results into Industrial Success. *Journal of Systems and Software*, 82(6):993–1003, 2009.

[130] Art Pyster et al. Graduate Software Engineering 2009 (GSwE2009) Curriculum Guidelines for Graduate Degree Programs in Software Engineering. *Stevens Institute of Technology*, 2009.

[131] Alex Radermacher, Gursimran Walia, and Dean Knudson. Investigating the Skill Gap

Between Graduating Students and Industry Expectations. In *Companion Proc. of the International Conference on Software Engineering (ICSE)*, pages 291–300. ACM, 2014.

[132] Damith Rajapakse. Peer Feedback in Software Engineering Courses. *Overcoming challenges in Software Engineering Education: Delivering Non-Technical Knowledge and Skills, IGI Global*, pages 111–121, 2014.

[133] Ita Richardson, Louise Reid, Stephen B. Seidman, Bob Pattinson, and Yvonne Delaney. Educating Software Engineers of the Future: Software Quality Research Through Problem-Based Learning. In *Proc. of the International Conference on Software Engineering Education and Training (CSEE&T)*, pages 91–100. IEEE, 2011.

[134] Ramón Rico, Miriam Sánchez-Manzanares, Francisco Gil, and Cristina Gibson. Team Implicit Coordination Processes: A Team Knowledge Based Approach. *Academy of Management Review*, 33(1):163–184, 2008.

[135] Pierre N. Robillard and Mihaela Dulipovici. Teaching Agile versus Disciplined Processes. *International Journal of Engineering Education*, 24(4):671–680, 2008.

[136] Pierre N. Robillard, Philippe Kruchten, and Patrick D. Astous. Yoopeedoo (UPEDU): a Process for Teaching Software Process. In *Proc. of the International Conference on Software Engineering Education and Training (CSEE&T)*, pages 18–26. IEEE, 2001.

[137] Guillermo Rodríguez, Álvaro Soria, and Marcelo Campo. Virtual Scrum: A Teaching Aid to Introduce Undergraduate Software Engineering Students to Scrum. *Computer Applications in Engineering Education*, 23(1):147–156, 2015.

[138] Guillermo Rodríguez, Álvaro Soria, and Marcelo Campo. Measuring the Impact of Agile Coaching on Students Performance. *IEEE Transactions On Education*, PP(99):1–1, 2016.

[139] Roshanak Roshandel, Jeff Gilles, and Richard LeBlanc. Using community-Based Projects in Software Engineering Education. In *Proc. of the International Conference on Software Engineering Education and Training (CSEE&T)*, pages 472–476. IEEE, 2011.

[140] Amalia Rusu and Mike Swenson. An Industry-academia Team-Teaching Case Study for Software Engineering Capstone Courses. In *Proc. of the Frontiers in Education (FIE)*, pages F4C–18. IEEE, 2008.

[141] Pete Sanderson. Wheres the Computer Science in Service-Learning? *Journal of Computing Sciences in Colleges*, 19(1):83–89, 2003.

[142] Simone C. dos Santos and Felipe S. F. Soares. Authentic Assessment in Software Engineering Education Based on PBL Principles: a Case Study in the Telecom Market. In *Proc. of the International Conference on Software Engineering (ICSE)*, pages 1055–1062. IEEE Press, 2013.

[143] Andreas Scharf and Andreas Koch. Scrum in a Software Engineering Course: An In-Depth Praxis Report. In *Proc. of the International Conference on Software Engineering*

Education and Training (CSEE&T), pages 159–168. IEEE, 2013.

[144] Jeffrey C. Schlimmer, Justin B. Fletcher, and Leonard A. Hermens. Team-Oriented Software Practicum. *IEEE Transactions On Education*, 37(2):212 – 220, 1994.

[145] Jean-Guy Schneider and Lorraine Johnston. eXtreme Programming—-Helpful or Harmful in Educating Undergraduates? *Journal of Systems and Software*, 74(2):121–132, 2005.

[146] Donald A Schön. Teaching Artistry Through Reflection-in-Action. In *Educating the Reflective Practitioner*, Handbook of Stuff I Care About, chapter 2, pages 22–40. Jossey-Bass Inc., San Francisco, California 94104, 1997.

[147] Mark J. Sebern. The Software Development Laboratory: Incorporating Industrial Practice in an Academic Environment. In *Proc. of the International Conference on Software Engineering Education and Training (CSEE&T)*, pages 118–127. IEEE, 2002.

[148] Yvonne Sedelmaier and Dieter Landes. Practicing Soft Skills in Software Engineering: A Project-Based Didactical Approach. *Overcoming Challenges in Software Engineering Education: Delivering Non-Technical Knowledge and Skills: Delivering Non-Technical Knowledge and Skills*, page 161, 2014.

[149] SEI. PSP. `http://www.sei.cmu.edu/Reports/00tr022.pdf`, November 2000.

[150] SEI. TSP. `http://www.sei.cmu.edu/Reports/00tr023.pdf`, November 2000.

[151] Andrew F. Seila. Introduction to Simulation. In *Proc. of the Conference on Winter simulation*, pages 7–15. IEEE Computer Society, 1995.

[152] Katherine Shaw and Julian Dermoudy. Engendering an Empathy for Software Engineering. In *Proc. of the Australasian Conference on Computing Education*, pages 135–144. Australian Computer Society, Inc., 2005.

[153] Linda B. Sherrell and Sajjan G. Shiva. Will Earlier Projects Plus a Disciplined Process Enforce SE Principles Throughout the CS Curriculum? In *Proc. of the International Conference on Software Engineering (ICSE)*, pages 619–620. ACM, 2005.

[154] Luis Silvestre, Maíra Marques, and Sergio F Ochoa. Understanding the Design of Software Development Teams for Academic Scenarios. In *Chilean Computer Science Society (SCCC), 2015 XXXIV International Conference of the*. IEEE, 2010.

[155] M. K. Smith. Donald Schön: Learning, Reflection and Change. http://infed.org/mobi/donald-schon-Learning-reflection-change/, Jan 2004. Encyclopedia of Informal Education, Last visit: Oct. 20, 2014.

[156] T. Smith, K. M. L. Cooper, and C. S. Longstreet. Software Engineering Senior Design Course: Experiences with Agile Game Development in a Capstone Project. In *Workshop on Games and Software Engineering*, 2011.

[157] Hongzhi Song, Guodong Si, Lei Yang, Huakun Liang, and Lixia Zhang. Using Project-Based Learning and Collaborative Learning in Software Engineering Talent Cultivation. In *Proc. of the International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 1288–1293, Nov 2011.

[158] Diane E. Strode, Sid L. Huff, Beverley Hope, and Sebastian Link. Coordination in Co-Located Agile Software Development Projects. 85(6):1222–1238, 2012.

[159] Ken Surendran, Hays Helen, and Macfarlane Andrew. Simulating a Software Engineering Apprenticeship. *IEEE SOFTWARE*, 19(5):49–56, Sep 2002.

[160] Hirotaka Takeuchi and Ikujiro Nonaka. The New New Product Development Game. *Harvard Business Review*, 64(1):137–146, 1986.

[161] UPEDU Team. UPEDU. `http://upedu.org`, January 2014.

[162] J. Barrie Thompson and Anthony J. Fox. Best Practice: Is this the Cinderella Area of Software Engineering? In *Proc. of the International Conference on Software Engineering Education and Training (CSEE&T)*, pages 137–144. IEEE Computer Society, 2005.

[163] James E. Tomayko. Carnegie Mellon's Software Development Studio: a Five Year Retrospective. In *Proc. of the International Software Engineering Education*, pages 119–129. IEEE, 1996.

[164] L. Tripp. IEEE Standards Collection Software Engineering. *New York, NY: Institute of Electrical and Electronics Engineers*, 1994.

[165] John D. Tvedt, Roseanne Tesoriero, and Kevin A. Gary. The Software Factory: Combining Undergraduate Computer Science and Software Engineering Education. In Hausi A. Müller, Mary Jean Harrold, and Wilhelm Schäfer, editors, *Proc. of the International Conference on Software Engineering (ICSE)*, pages 633–642. IEEE Computer Society, 2001.

[166] David Umphress, T. Dean Hendrix, and James H. Cross. Software Process in the Classroom: The Capstone Project Experience. *IEEE Software*, 19(5):78, 2002.

[167] Richard L Upchurch and Judith E Sims-Knight. Reflective Essays in Software Engineering. In *Proc. of the Frontiers in Education (FIE)*, volume 3, pages 13A6–13. IEEE, 1999.

[168] David Valencia, Aurora Vizcaíno, Juan Pablo Soto, and Mario Piattini. A Serious Game to Improve Students' Skills in Global Software Development. In James Onohuome Uhomoibhi, Gennaro Costagliola, Susan Zvacek, and Bruce M. McLaren, editors, *CSEDU 2016 - Proceedings of the 8th International Conference on Computer Supported Education, Volume 1, Rome, Italy, April 21-23, 2016.*, pages 470–475. SciTePress, 2016.

[169] Ricardo Valerdi and Ray Madachy. Impact and Contributions of MBASE on Software Engineering Graduate Courses. 80(8), August 2007.

[170] Louwarnoud van der Duim, Jesper Andersson, and Marco Sinnema. Good Practices for Educational Software Engineering Projects. In *Proc. of the International Conference on Software Engineering (ICSE)*, pages 698–707, 2007.

[171] Jari Vanhanen, Timo O.A. Lehtinen, and Casper Lassenius. Teaching Real-World Software Engineering through a Capstone Project Course with Industrial Customers. In *Proc. of the International Workshop on Software Engineering Education Based on Real-World Experiences*, pages 29–32. IEEE Press, 2012.

[172] Rayford B. Vaughn. A Report on Industrial Transfer of Software Engineering to the Classroom Environment. In *Proc. of the International Conference on Software Engineering Education and Training (CSEE&T)*, pages 15–22, 2000.

[173] Kera Z. Watkins and Tiffany Barnes. Competitive and Agile Software Engineering Education. In *IEEE SoutheastCon*, pages 111–114. IEEE, 2010.

[174] Carol A. Wellington, Thomas Briggs, and C. Dudley Girard. Examining Team Cohesion as an Effect of Software Engineering Methodology. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 1–5. ACM, 2005.

[175] Robert K. Yin. *Software Engineering: a Practitioner's Approach*. Sage Publications, 2011.

[176] Robert K Yin. *Case Study Research: Design and Methods*. Sage Publications, 2013.

[177] Hadar Ziv and Sameer Patil. Capstone Project: From Software Engineering to "Informatics". In *Proc. of the International Conference on Software Engineering Education and Training (CSEE&T)*. IEEE Computer Society, March 2010.

Appendix A

Software Process Definition

This appendix presents in detail EduProcess, our prescriptive development process. The individual phases, artifacts, milestones and tasks are described in the next subsections of this chapter.

A.1 Project Contexts

Software development projects usually have more than one possible solution for most problems. However, these solutions should be restricted according to the development context (i.e., development team, project features, client availability, etc.). In this sense, EduProcess is not applicable to any software projects, but only for some types of academic projects.

A.1.1 Type of Projects

There are three project types that are considered in our process:

- new development - new development projects occurs in case of green field projects; a new project will be conceived and developed from scratch.
- extensions - extensions could involve adjustments of already implemented software, but mainly the development of new functionalities.
- reengineering/replacements - reengineering/replacement projects focus on rethinking or re-conceiving an already implemented system, and eventually reusing some legacy components.

New development are greenfield projects, extension or reengineering projects represent brown field projects. All these projects types represent a challenge for student teams, but for different reasons. The new developments push the students to help the client to clarify the problem and context involved in the project, because the project goal and scope are usually not clear at the beginning. Moreover, these projects require that the developers know enough (or learn enough) of the application domain of the problem to address and to propose a solution that can really solve it.

The extension projects assume that the product design (at least its architecture and main components), and the technology used to implement the product are both well defined (programming language, database, and sometimes development frameworks). Moreover, the developers must understand and interiorize the design and code authored by other people, which is always a challenge. Usually in this type of project the problem to address is more clear and the knowledge of the business niche is more available than in new developments.

The reengineering/replacements projects are normally updates of an already existing system; the update can be in terms of programming language, performance improvement, algorithm improvement, databases, frameworks or even everything already mentioned at the same time. Depending on the reengineering project, the team may reuse components of the legacy system. When the reengineering is not about performance or user interface, it is possible to reuse the legacy system design either completely or partially. But there is no challenge with this type of project in terms of business knowledge.

The project type is a variable that is part of the project context, and demands that the software process be adjusted for addressing each particular project type. Therefore, we propose a general process that can be adapted to deal with new developments, extensions and reengineering/replacements. Regardless of the type of project to address, EduProcess allows the instructor to keep only one course calendar and one set of assessment activities, which considerably eases the monitoring and assessment of the projects evolution and obtained product.

A.1.2 Product Type, Size and Complexity

This process was conceived to mainly build web information systems for several reasons. Usually this type of system is not too complex and is the most frequent type of projects developed in industry. They can be addressed in the course available time, although the developers do not have a deep knowledge of the business domain where the problem belongs. Most of the students already have some experience implementing software as part of an academic activity or for the industry; therefore, they are in some way conscious about the activities to perform and challenges to address. There is at least a common knowledge domain that can be expected of this kind of project, since they are all web information systems related somehow to the university. Typically they have similar complexity among them. The students perceive that learning to develop this kind of system is valuable for them.

Normally, web information systems are not considered very complex projects, because they are mainly visual software projects. They are normally projects that have a ratio between backend and frontend where the front-end has a major relevance and the backend does not have more than 40% of the development effort. This is a main issue with our fifth year students; our studies show that students are not so prone to understand client's problems when they cannot somehow visualize them.

Provided that the semesters have a fixed duration, the project should be selected and defined to ensure that a reasonable product can be obtained in such a period. Therefore, the size and complexity of the product to be developed should be bounded. The course instructor and the client should agree to a first scope; the development team should negotiate the project

scope details. The main idea of this first agreement between the instructor and the client is that the project general scope has to be defined to avoid false expectations on the client's side. The students team will have a project that they can manage and correctly develop within the course time frame.

A.1.3 Our Context

The context of this thesis was inspired in the Software Engineering II (CC5401) course, a mandatory project-based course of Universidad de Chile, that was used as a laboratory to develop and validate the hypotheses of this thesis. The primary goal of this course is to teach students software engineering in a "hands-on" way, by resembling an industrial setting. The students enrolled in this course are in the fifth year of a computer science engineering undergraduate program. They usually do not have much experience in real developments. The course demands 10 hours per week of dedicated student work, and it is divided into: (1) three hours of lectures per week, (2) one hour and a half of auxiliary classes, and (3) at least five and a half hours per week of project development. The duration of the course is fifteen weeks (one semester).

The educational approach of the course is the Project-Based Learning [13]. The project-based learning is a comprehensive approach to classroom teaching and learning that is designed to engage students in investigation of authentic problems. According to Blumenfeld "Within this teaching approach students pursue solutions to nontrivial problems by asking and refining questions, debating ideas, making predictions, designing plans, collecting and analyzing data, drawing conclusions, communicating their ideas and findings to others, asking new questions and creating artifacts." This approach also places students in realistic, contextualized problem-solving environments; the projects are the bridges that students need to link classroom contents and real-life experiences. In order to be effective, the project-based learning approach must be designed in such a way that projects have to sustain motivation over time. Project-based learning has great potential to enhance students' motivation and learning [157].

The development teams (5 to 7 students) are formed considering their technical and social abilities [154]. The projects are selected by the instructor, who makes sure that the projects are within the context restrictions of the course and that the clients will have the time to meet the students at least an hour per week. The instructor randomly assigns the projects to the teams.

The students are evaluated using two exams (40% of the final grade) and the project results (60% of the final grade). The grades in Chile range from 1 to 7 with 4 being the passing grade. But students at this course have to pass the project and the exams separately, so they have to have a grade equal or larger than 4 in project grades and in exam grades. According to [170], any software engineering course involving practical activities should have learning outcomes defined in terms of cooperation level among team members, predictability of the project results and software quality. In our course it is possible to say that, with the implicit process that was being used, SSP, we were short on team cooperation and software quality [101]. Being more specific to our context, we have some restrictions that characterize our context that are detailed in Table A.1.

Table A.1: Context Restrictions

Restriction	**Our Context**	**Literature**
Type of student	5th year students, usually neither used nor prone to work in teams.	"Most computer science students have little intrinsic motivation for the social aspects of software engineering." [170]
Small projects	Since the course is only 15 weeks, the projects need to fit this short time frame of development. Moreover, all the projects should be of a similar size among team projects (around 100 function points).	The typical project develops a software with size around 100 function points. This size proved to be large enough to exercise a meaningful set of important software engineering practices. [125]
Real clients	Clients are internal to the university, but they normally do not expect much of the project. Typically, the clients are not fully committed with the project.	It is hardly possible to set up a project that gives students the same structure that the industry are used [170]. So a "lightweight" client may be handy.
Technology	Most are brownfield projects, meaning that students do not have much of a choice in the technology that will be used for implementing the solution.	While software engineering education is most frequently based on greenfield development, in industry, software is rarely developed from scratch. Instead, when the project starts, there is often already a software system in place and the goal of the project is to enhance or optimize the system [170].
Type of project	All software products to be developed are web information systems, and they typically have similar complexity among them.	Normally the project develops a small information system application [125].
Roles	Each team member plays one or more roles (project manager, analyst, designer/developer or quality assurance engineer).	Many software processes and industries assign different roles for team members and software engineering education needs to reflect this [80].
Product quality	Students should use some of the software engineering best practices to verify the quality of the product being developed, i.e.; test cases, unit tests among others.	In student software projects assertions on the quality of the product normally cannot be made [80].

The teams have specific roles that are assigned according to the size of the team, and if possible according to student's preferences. EduProcess have five different roles that may be assigned to students: project manager, requirement analyst, designer-developer or quality assurance engineer. Depending on the number of students assigned to each team the role of the quality assurance engineer can be acumulated with other roles such as analyst or designer. Also, depending on the size of the teams, they will have one or more developers.

Lecture class attendance is not mandatory but auxiliary class attendance is mandatory in our course. During the auxiliary classes each team is assigned to a monitor that will perform the Reflexive Weekly Monitoring (RWM) explained in detail in Chapter C, therefore students are required to come to class and attend a team meeting at least once a week. The main idea of this weekly monitoring is to help students with their project and to prevent the work from being done at the end of the course term [80] as we empirically saw with the previous process; and to avoid last minute project integration failures (typical explanation: "It was working yesterday!").

During the project students must use a version control system; the department provides a SVN account on the department server but students can use another version control system if they wish. Students should also report user and system requirements, the design, the tests and the project status on a tool called MainReq (http://mainreq.dcc.uchile.cl). This tool is a new version of a tool that already managed requirements in student software projects over the last ten years. It helps students create the necessary documentation of the project and keep track of its status.

At the end of each project iteration, students have to answer a peer evaluation stating their opinion about their teammates. A tool called CoEvaluaciones (http://coevaluaciones.dcc.uchile.cl) is used for peer evaluation.

Throughout the semester students may encounter some conflict with other i.e.; one team member that does not do his/her tasks, does not show up for meetings. As a first step, the monitor will try to help the students to solve the problems within the team. In the case that the monitor is not able to solve the problem, the instructor tries to solve and manage the problem. As a last resource, if the monitor and the instructor are not able to solve the conflict, the project manager has the power of "firing" the team member that is generating the problem; this student is automatically assigned a project grade of 1 on the project and as a consequence he/she will fail the course.

A.2 Roles

A.2.1 Project Manager

The project manager is responsible for the project orchestration. He/she is one of the most critical roles of the team. Among his/her responsibilities are:

- Delimit the scope of the project (with the team and the client)
- Plan/Re-plan and manage the project

- Create a test plan and an implementation plan
- Coordinate all team members
- Interact with the client
- Assure that the objectives, dates and requirements committed are being fulfilled

A.2.2 Analyst

The analyst should be a business-focused person concerned with the business process, capturing requirements, and producing specifications. The main tasks of the analyst are to:

- Determine the objectives and scope of the project (together with the team).
- Identify and interview clients and users.
- Clarify and specify requirements.
- Support the quality assurance engineer in the specification of test cases.
- Ensure that the design meets the requirements (with the quality assurance engineer).
- Ensure that the final product meets the requirements (along with quality assurance engineer).

A.2.3 Designer/Developer

The designer/developer is responsible for the generation of the design of the front-end and backend of the software and to develop the software. His/her responsibilities are:

- Generate the architecture design and the detailed design based on the requirements documentation
- Implement the fast prototype to evaluate the requirements
- Do the first battery of software tests (unit tests) and adjust the software as needed
- Design and perform the deployment planning (with the project manager)
- Ensure that the final product is adjusted to the design proposed

A.2.4 Quality Assurance Engineer

The quality assurance engineer is responsible for the assurance of the quality of all team products (documents, prototypes, code, etc.). Some of his/her responsibilities are:

- Coordinating the revisions of all project products

- Generating the post-revision reports
- Conducting a follow-up of the identified short-comings
- Reporting to the project manager about identified risks
- Assuring that the standard of the project is being adopted
- Assuring the completeness and exactness of the documentation
- Assuring the quality of the final product
- Specifying the test cases to be executed

A.3 Introduction to the Phases, Activities and Milestones

The product of a software development project will be delivered in a timely manner and be appropriate for its purpose. Software development activities will be systematically planned and carried out. Our life cycle model structures project activities into four phases and defines which activities occur in which phase. Table A.2 presents the basic phases **EduProcess**.

Table A.2: Software Life Cycle Model

Items	**Conception**	**Iteration 1**	**Iteration 2**	**Deployment**
Major Activities	Problem and context identification Prototype construction	Identification of user and software requirements Design definition Software construction	Identification of user and software requirements Design definition Software construction	Tuning Final acceptance tests Deployment
Deliverable Items	Prototype	Requirements Document Design Document Alpha Demo	Requirements Document Design Document Beta Demo	Documentation Source Code Software
Reviews	Problem Presentation Prototype Presentation	Requirements Evaluation Design Evaluation Demo Presentation	Requirements Evaluation Design Evaluation Demo Presentation	Deployment Presentation
Major Milestones	Problem Presentation Prototype Presentation	Requirements Evaluation Design Evaluation Demo Presentation	Requirements Evaluation Design Evaluation Demo Presentation	Deployment Presentation Delivery

Our process is divided in four major phases: Conception, Iteration 1, Iteration 2 and Deployment. Each phase has its own focus and its own objectives. Despite the fact that these four phases are represented here in Figure A.1 as a sequence, they are not exactly a sequence, there is an overlap between the beginning of each phase and the end of the prior phase. The full process is available online: (http://www.dcc.uchile.cl/∼mmarques/).

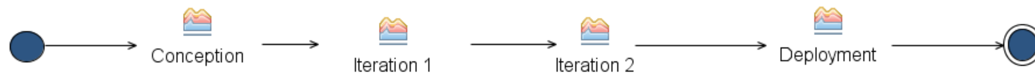


Figure A.1: General Structure of EduProcess

A.3.1 Conception

The main objective of the conception phase is to understand the client's problem, the scope and context of the problem. In order to be able to do that, the team has to elicit the main user requirements and make sure that the client's problem is well understood and that will be correctly addressed. At the end of this phase, the team has to present a mockup prototype that does not need to be functional yet. The focus is to assure that team and client are well aligned and that the clients problem will be solved with the solution that the team is presenting.

Previously, the prototype was not required, and the students and client expectations did not match in some cases at the end of the term, because the team normally discovered that they did not understand the client's problem by the middle of the semester and there was no time left to correctly address the client's problem. The prototype help the team better understand the clients problem and to manage the expectation of the client of what he/she will have at the end.

A.3.2 Iteration 1

The main objectives of Iteration 1 are to translate the user requirements into software requirements, do the product design and begin the product implementation. In our process there are two activities of evaluation, an internal one that is the System Verification and an external one, the User Validation, - where the client/user have to validate the software functionality that was negotiated between the team and the client for this phase.

A.3.3 Iteration 2

The main objective of Iteration 2 is to finish the product implementation; but during this iteration the team has to update the user and software requirements according to the client's observations and prioritization done on Iteration 1. By the end of this interaction there is a formal activity named: User Validation, that gives the client the final opportunity to validate all the software functionality developed that was agreed upon the team and the client.

A.3.4 Deployment

After Iteration 2 comes the Deployment phase. The team has to correctly deploy the software in the client's environment. The software has to be completely validated by the user, installed at the clients facilities, configured and migrated; as a summary the team has to deliver a complete running software.

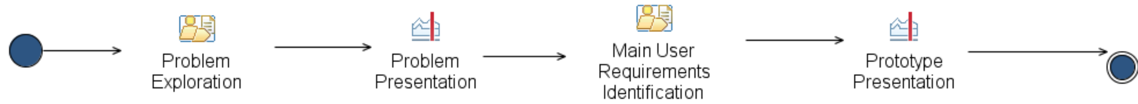


Figure A.2: Conception Phase

A.4 Conception Phase

A.4.1 Introduction

The main objective of the conception phase is to understand the client's problem, its scope and context. In order to do that, the team has to elicit the main user requirements and make sure that the client's problems are well understood and that they will be correctly addressed by the team. The team has to present a mockup prototype at the end of this phase. The focus is to assure that team and client are well aligned and that the client's problem will be solved with the solution that the team is presenting.

The most important product of this phase is the prototype. The use of prototypes to test customer reaction and design feedback is common to many engineering disciplines. It reduces the risk in a project through practical experience. The prototype will be refined by additional client's input and development work along the course to evolve to the final product to be deployed.

In this phase there are no formal differentiated roles in the team as all team members should be equally involved on the project. The team should have a clear idea of the client's problem and context and what direction the software solution should have.

A.4.2 Inputs

Inputs are not strictly required, although interviews, surveys, studies are often helpful for understanding the client's problem and context.

A.4.3 Activities

The main goal of the Conception phase is to understand the client's problem and context to be able to build a prototype. Figure A.2 shows the activities of the phase. In the Conception phase we have two activities and two reviews (milestones), that are detailed below.

A.4.3.1 Problem Exploration

It is the phase of the life cycle where the problem and the scope are defined. The purpose of this phase is to refine the idea of what is expected from the development. This activity is constituted mainly by interviews with the client. *Problem Exploration* activity is divided in four tasks (Figure A.3), and each task is divided into steps.

- Identification of Problem and Problem Context - meeting with the client, meeting notes
- Identification of Stakeholders and Users - client inquiry

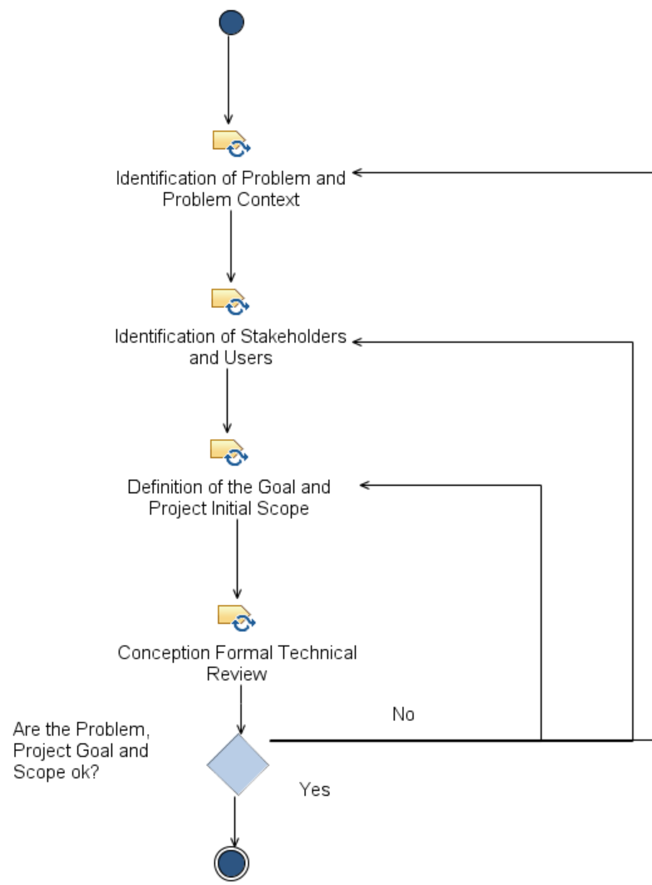


Figure A.3: Problem Exploration

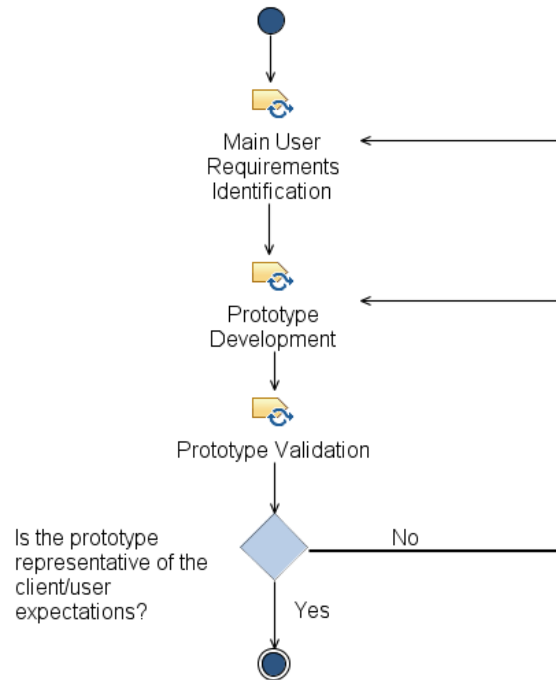


Figure A.4: Main User Requirements

- Definition of the Goal and Project Initial Scope - client meetings
- Conception Formal Technical Review - problem presentation

A.4.3.2 Main User Requirements Identification

The Main User Requirement Identification is divided in four tasks (Figure A.4). Each task is divided in steps:

- Main User Requirements Identification - elicit main user requirements; analyze main user requirements
- Prototype Development - sketch a prototype version of the software on paper; make this version in digital format for client validation
- Prototype Validation - prototype presentation, feedback, notes of the feedback

A.4.4 Reviews (Milestones)

A.4.4.1 Problem Presentation

This is a short presentation where the team has to present the problem. The client needs to address with a software solution. The team should also understand the context of the problem, have people and/or software identified that will have any link to the software developed; as input source or as user of any output of the software. Another important point to evaluate

is if there is any restrictions to the solution that should be known from the beginning. The presence of the client in the presentation is mandatory.

The presentation should contain at least:

- Identification of team members, clients and users
- Problem
- Context of the Problem
- Objectives (solution main objectives)
- Data model (preliminary)

A.4.4.2 Prototype Presentation

The team should present a mockup interface of the future software. The interfaces can be built in a specific software or with powerpoint and there is no need to use a programming or a software framework. The prototype should have all the software interfaces that the user/ client will have at the end of the project.

This prototype is not the final design of the software, it is only a sketch of what the software will be to validate the concepts and the problem solution.

A.4.5 Outputs

The main output of the conception phase is a prototype of the software solution, but it is not mandatory to present a functional prototype. The prototype helps the team and the client to manage their expectations and to minimize the risks of misunderstanding between one another. Another output of the Conception phase is the role definition of each team member. Team members must choose the roles they want to play in the next phases, but the team should agree on the chosen roles because all the roles must be filled.

A.4.6 Deliverables

The non-functional prototype is the only deliverable of the conception phase. The prototype helps students better understand the client's problem and also help the team to focus on what has to be done. The prototype works as a guide for the team, to know exactly what it has to develop, and the client knows what to expect from the team.

A.5 Iteration 1 and 2

A.5.1 Introduction

The Iteration 1 phase is one of the most critical phases of our life cycle. The purpose of this phase is to refine the prototype created in the Conception phase and elicit and specify

the requirements of the expected software solution. In this phase students start working in their specific roles, and they have to fulfill each task they are assigned in a responsible and timely way.

The Iteration 2 phase is the third phase of our process. The purpose of this phase is to finish the software to deliver to the client what is expected. This phase can be interpreted as a second round of the Iteration 1 because they have almost the same activities. The only activity that is not part of the Iteration 1 and it is part of this phase is the Deployment activity, since it is not always possible due to time restrictions of deploying in Iteration 1.

Figure A.5 offers an overview of this phase.

A.5.2 Inputs

In Iteration 1 this phase receives as input the two outputs from the conception phase: the team role definition and the prototype that has to evolve during this phase. The meeting notes can be considered as an informal input and also the problem and prototype review done in the two milestones of the prior phase.

In Iteration 2 all the outputs of Iteration 1 are inputs, so the inputs for iteration 2 are:

- User Requirements (output of Iteration 1)
- Software Requirements (output of Iteration 1)
- Test Cases (output of Iteration 1)
- Requirements Review Notes (output of Iteration 1)
- Design Review Notes (output of Iteration 1)
- Demo Review Notes (output of iteration 1)
- Software Requirement Document (deliverable of Iteration 1)
- Design Document (deliverable of Iteration 1)
- Code - Alpha Version (deliverable of Iteration 1)

A.5.3 Activities

In this phase there are four activities: user requirement definition, software requirement definition, product implementation and one optional activity Figure A.5. The requirement definition activities (user and software) are the critical ones for the team, because this is where the team has to understand the client's need in detail and the whole project depends on the requirements specified. And the product implementation is where the effort of the team is needed, is where the team will spend their time developing what was specified. The activities and their brief description follows below. Iteration 2 is an improvement and extensions of

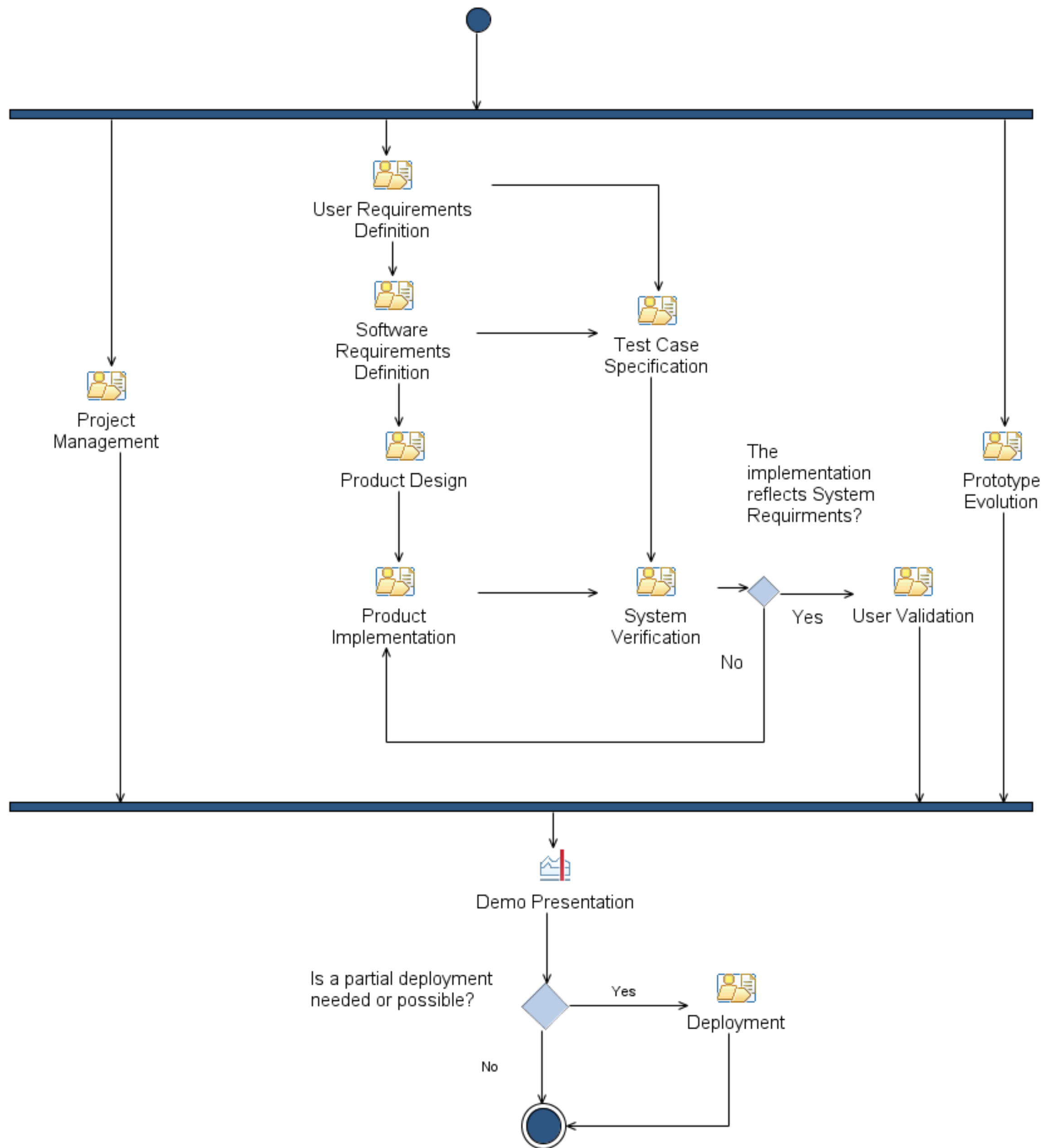


Figure A.5: Phase - Iteration 1 and 2

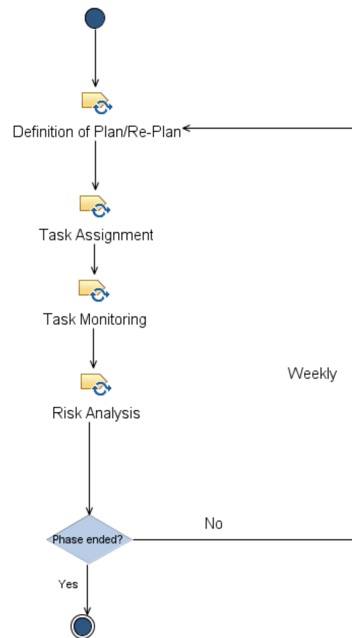


Figure A.6: Project Management Activity

Iteration 1 so, there is a lot of work already done.

A.5.3.1 Project Management

The Project Management activity is transversal through the whole phase. The project manager should plan/re-plan every week the tasks that need to be finished by the end the phase. The Project Management activity is divided in four tasks (Figure A.6).

- Definition of Plan/re-Plan - the project manager has to constantly (weekly) plan/re-plan the project.
- Task Assignment - the project manager has to assign specific tasks to each member of the team and define a deadline for each task.
- Task Monitoring - the project manager has to monitor all the assigned tasks progress properly.
- Risk Analysis - the project manager has to analyze the risks that the project may face during its development.

A.5.3.2 Prototype Evolution

The Prototype Evolution activity is another transversal activity of this phase. During the whole phase the team has to update the prototype or/and change the prototype screens as needed to real and functional ones. This activity has just one task: the Continue Prototype Validation. All the roles are responsible for this activity.

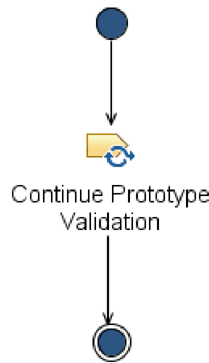


Figure A.7: Prototype Evolution Activity

A.5.3.3 User Requirements Definition

The main activity of the User Requirements Definition is to capture the user requirements and document them. The scope of the software has to be established and the interfaces with external systems identified. User requirements are originated as a simple perception of needs, because of that they should be clarified through the criticism and experience of existing software and prototypes. A comprehensive possible agreement about user requirements should be established through interviews. The user requirements is an iterative process, and may have to be repeated a number of times before the user requirements are ready for review. The most important role in this activity is the analyst, but he/she has to be supported by the project manager and the quality assurance engineer.

In Iteration 2, most user requirements were already elicited and specified and the team has to evaluate with the clients if there is a need to change something (new requirements, delete requirements, update requirements) and also reprioritize the user requirements. Again the user requirements evaluation may have to be repeated a number of times before the user requirements are ready.

The User Requirement Definition activity is divided in five tasks that are divided in steps (Figure A.8).

- User Requirements Elicitation - Meeting with the client - The team should have at least two or three meetings with the client to elicit and gather all the user requirements.
- Specification of the Logical Model of the Process to be Supported - The team has to be able to show that they understood where the software will be inserted by creating a DFD (Data Flow Diagram) or something that shows the current status, where the data of the software came from and where it is used.
- User Requirements Specification - The team has to write down the user requirements in the MainReq (course app). More detail of the specification follows below.
- User Requirements Prioritization - The team has to meet with the client and find a consensus on what they will do as part of the first iteration and what they will do in

the second iteration. In each iteration the client has to prioritize the user requirements according to their importance.

- User Requirements Validation - The user requirements should be validated with the prototype. The team has to upgrade the prototype of the conception phase to show the client a new version of the prototype where the client can see all the main user requirements, and imagine in a more concrete way how the software will work when finished.

At the end of the User Requirement Specification task the requirements should contain all the information needed to fulfill this item in the Software Requirement Document, the SRD. The User Requirements Document, the URD, describes in detail all the user requirements. They are written with a language oriented to the client or user. The use of technical terms should be avoided. Each user requirement should be specified according to the structure mentioned in Table A.3.

Table A.3: User Requirements Specification Detail

Item	**Description**
ID	Unique code used to identify and formally refer to this requirement in the project. In our management system (MainReq) it is automatically assigned incrementally using the "RUXXXX" format.
Name	Name in colloquial language.
Description	Description of the requirement (the aspects involved, what it is, etc.).
Priority	Level of urgency associated with the requirement. The priority of a requirement may be "urgent", "soon", "normal" or "if possible".
Stability	This attribute determines whether or not the requirement may be subject to change during the software development activities. Stability can be "negotiable" or "non-negotiable".
Type	User requirements can be classified into 3 types: 1. Functional: Describes a capacity that must be supported by the software product. 2. Quality: A quality requirement is a non functional feature that can be expressed on a scale. 3. Restriction: They indicate what constraints have to be considered during the development and/or operate the software, depending on cost, time, personnel, operational environment, hardware, networks, etc.
Source	Document or person from which the requirement arises.
State	The current state of compliance with the requirement in the development, may be "Compliant", "Non-Compliant" or "Ambiguous". The default when creating status is "Non-Compliant".

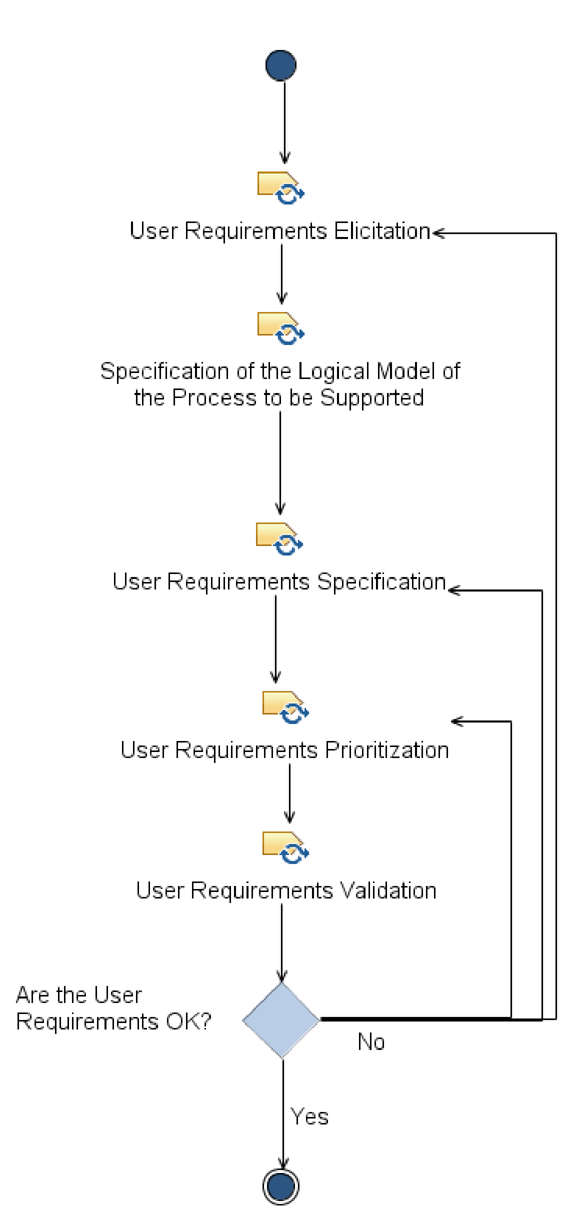


Figure A.8: User Requirement Activity

A.5.3.4 Software Requirements Definition

The purpose of this activity is to analyze the statement of user requirements and produce a set of software requirements as complete, consistent, unambiguous and correct as possible. In this activity software requirements are defined. They are written in a more developed team-oriented language. It is possible to use technical terms. These software requirements are derived from user requirements. The most relevant role in this activity is the analyst, but he/she has to be supported by the project manager and the quality assurance engineer.

In Iteration 2 most software requirements were already elicited and specified. The team has to evaluate with the clients if there is a need to change something (new requirements, delete requirements, update requirements) and also has to reprioritize the requirements.

The Software Requirement Definition activity is divided in six tasks and has one milestone (Figure A.9):

- Software Requirement Elicitation - Identify software requirements from user requirements.
- Logical Model of the Process to be Supported - The team has to be able to show that they understood where the software will be inserted by creating a DFD (Data Flow Diagram) or something similar, that shows the data flow after the software is delivered, where the data of the software comes from and where it is used.
- Software Requirement Specification - The analyst has to write down the software requirements unambiguously in the MainReq system.
- Software Requirement Analysis - The Analyst creates the traceability matrix between SR (software requirements) and UR (user requirements). This matrix must show the SR associated with each UR. It must take into consideration: that all URs should be associated with at least one SR, all SRs should implement at least one UR.
- Software Requirement Validation - The Project Manager checks the traceability matrix (SR x UR) to make sure that all URs have at least one SR, and all SRs are associated with at least one UR. If one of this situations fails to occur, the Analyst must make the adjustments until the traceability matrix is consistent.
- Recording Formal Technical Review Result of Requirement Presentation - The feedback given by the client/user, instructor and teaching assistants has to be recorded (notes or audio recorded) for later evaluation.

At the end of the Software Requirement Specification task, the requirements should contain all the information needed to fulfill this item in the SRD (Software Requirement Document). The SRD describes in detail all the system software requirements. They are written in a technical language, targeted to the development team. The analyst reviews the specified user requirements, assisted by the designer, to formally establish what needs to be done. Each software requirement must be specified with the structure presented in Table A.4.

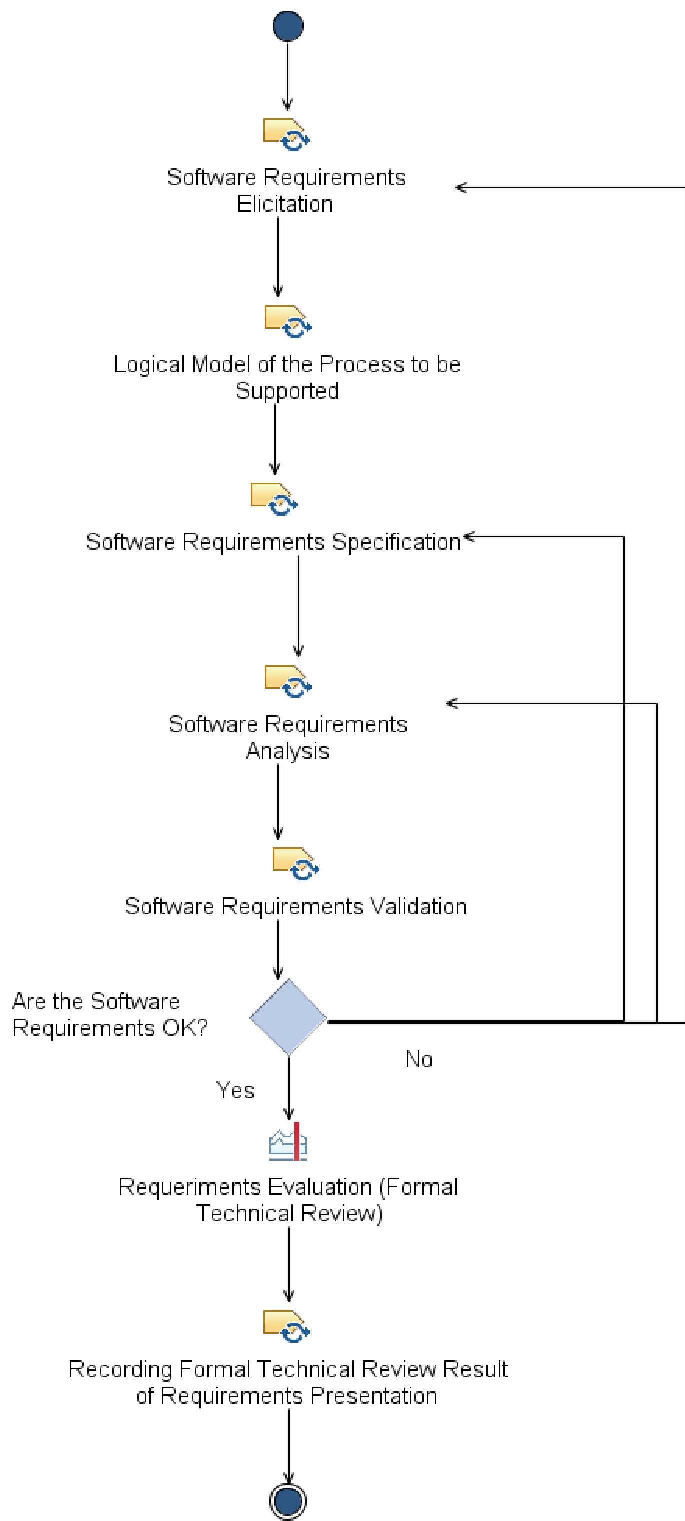


Figure A.9: Software Requirement Activity

Table A.4: Software Requirements Specification Detail

Item	**Description**
ID	Unique code used to identify and formally refer to this requirement in the project. In our management system (MainReq) it is automatically assigned incrementally using the “RUXXXX” format.
Name	Name in colloquial language.
Description	Description of requirement (the aspects involved, what it is, etc.).
Priority	Level of urgency associated with the requirement. The priority of a requirement may be “urgent”, “soon”, “normal” or “if possible”.
Stability	This attribute determines whether or not the requirement may be subject to change during the product life-cycle management. Stability can be “negotiable” or "non-negotiable”.
Source	Document or person from which the requirement may arise.
State	The current state of compliance of the requirement, the possibilities are: “Compliant”, “Non-Compliant” or “Ambiguous”. The default when creating a requirement is “Non-Compliant”.
UR Association	Each software requirement should be associated with their corresponding user requirement (can be more than one).

Item	Description
Classification Type	• Functional: Indicate the capabilities that the software should have. They are derived from the logic model. • Interface: Specify the hardware, software or database elements with which the system or its components interact and communicate. • Operational: Specify how the system will run and how it will communicate with human operators. They include all user interfaces, human-computer interaction, and logistical and organizational requirements. • Resources (Operating Environment): Specify the lower bounds of physical resources such as processing power, memory usage, disk space, etc. • Usability: They are related to the effort of the user to use it. • Maintainability: Related to the effort to make modifications, specifying how easy it is to repair faults and software adaptation to new requirements. • Portability: Related to system's ability to be transferred from one environment to another. • Reliability: They are related to the ability to maintain, under certain conditions and for a time an adequate level of service. Specifying acceptable mean times between failures. • Interoperability: Ability to interact with certain other systems. • Performance: They set numerical values to measurable variables related to system performance. • Documentation: Specify if the project has a specific need for documentation. • Scalability: Maintain or improve the performance when the number of users grows.

A.5.3.5 Product Design

The purpose of this activity is to clarify all the software components and detail the design and the decisions that need to be made to construct the software. The most important role in this activity is the designer, but it is advised that he/she consult and make decisions with the help of the other team members.

In Iteration 2 the design of the software is already done, since there is a first version of the software already delivered. The designers need to evaluate if there are changes that were made in the implemented software that are not reflected on the design already written.

The Product Design activity is divided in five tasks and has one milestone (Figure A.10):

1. Operational Environment Specification - The designer/developer has to find out more about the environment where the final product will be deployed.

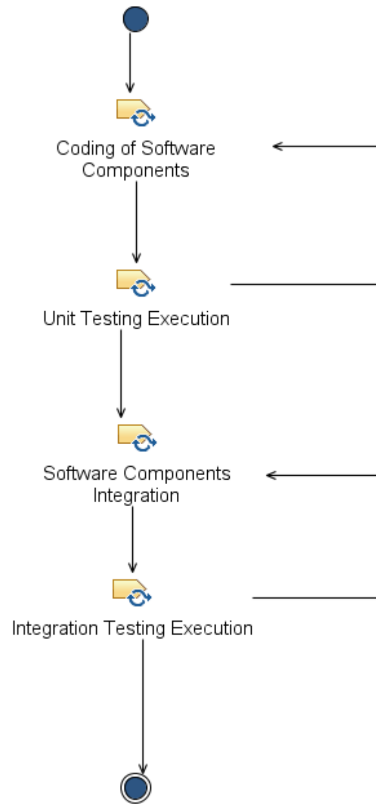


Figure A.10: Design Activity

2. Definition of Data Model - The designer must specify the data model that will be used in the software design.

3. Software Architecture Definition - Analyze software requirements to evaluate what kind of architecture the software needs. It is important to take into consideration the quality requirements that may impact the architecture.

4. Detailed Design Definition - Create the detailed design, create the navigation design and create the navigability design.

5. Recording Formal Technical Review Result of Design Presentation - The feedback given by the client/user, instructor and teaching assistants has to be recorded (notes or tape recorded) for later evaluation.

A.5.3.6 Product Implementation

This is the activity where the software will be built, where preliminary versions will be changed to fully working versions. The Product Implementation activity is divided in four tasks (Figure A.11):

- Coding of Software Components - The developers have to implement the code to satisfy the software requirements following the design decisions that were made in the design documentation.

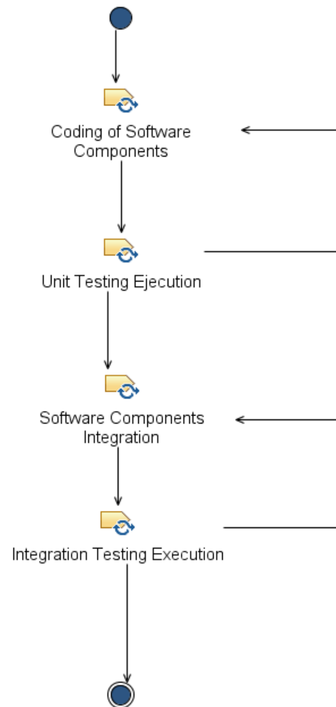


Figure A.11: Implementation Activity

- Unit Testing Execution - The developers have to implement and apply unit test cases to assure the quality of the code.
- Software Components Integration - The developers have to integrate their code with the code of the other developers.
- Integration Testing Execution - The developers have to assure that all the integrated code continues to function properly.

A.5.3.7 Test Case Specification

The Test Case Specification activity has only one task: Test Case Development (Figure A.12). In the test case development task, the quality assurance engineer must create and implement test cases that will be used to detect errors in the developed system to verify that the software product conceived has interpreted correctly all the specified requirements.

At the end of the Test Case Specification task, each test case should contain all the information needed to fulfill this item in the SRD (Software Requirement Document). A test case should be associated with a user requirement or a software requirement and may be associated with various types of users indicating the people who should perform each test. Table A.5 presentes the detailed specification of a test case.

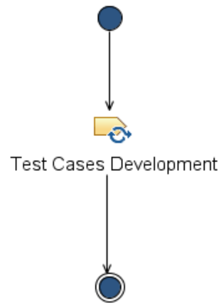


Figure A.12: Test Case Activity

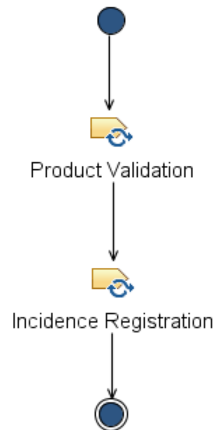


Figure A.13: System Verification Activity

Table A.5: Test Case Detail

Item	**Description**
ID	Unique code used to identify and formally refer to this test case in the project. In our management system (MainReq) it is automatically assigned incrementally using the "CPXXX" format.
Name	Simple test case name.
Description	The procedure to apply the test case is described here.
Acceptable Result	Describes the minimum acceptable outcome to this test that makes it successful.
Optimal Result	Describes the best expected result after testing.
State	The current compliance status of the test case can be "Compliant", "Non-Compliant" or "Ambiguous". The default when creating status is "Non-Compliant".

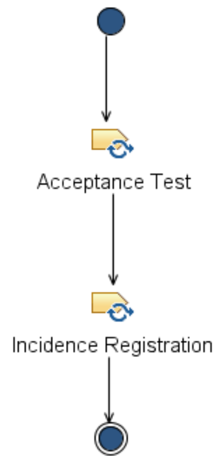


Figure A.14: User Validation Activity

A.5.3.8 System Verification

In this activity the most important role is the quality assurance engineer, he/she has to fully validate the software built based on the software and user requirements. The System Verification activity is divided in two tasks (Figure A.13):

- Product Validation - In this task the quality assurance engineer has to fully validate the software requirements against the code implemented by the developers.
- Incidence Registration - The quality assurance engineer has to report all the incidences found during the execution of the test cases.

A.5.3.9 User Validation

This activity is where the client/user has to evaluate the software built and report to the team in case there is something not working properly. Iteration 2 is the last instance the client/user has to test and evaluate the software prior to the end of the project, so the client/user has to put effort in their evaluation. The User Validation activity is divided in two tasks (Figure A.14):

- Acceptance Test - The client/user must test the software to accept it.
- Incidence Registration - In case there is something not working properly, the client has to report the incidence to the team.

A.5.3.10 Deployment

This activity is optional in Iteration 1 and in Iteration 2 is a phase and not just an optional activity. If possible, the software developed should be deployed in this phase to allow users/clients to fully use it. The Deployment activity is divided in four tasks and one milestone (Figure A.15):

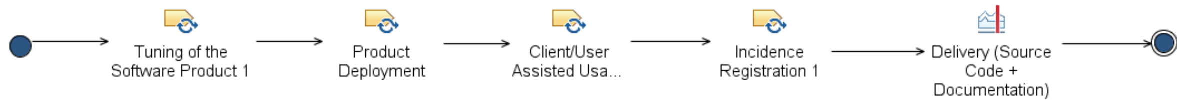


Figure A.15: Optional Deployment Iteration 1

- Tuning the Software Product - The developer team may have to perform some tuning of the software.
- Product Deployment - The team has to deploy the software in the client environment.
- Client/User Assisted Usage - The team has to help the client/user in their first interactions with the software.
- Incidence Registration - The client/user has to report to the team all the failures or problems that they find in the software.

A.5.4 Reviews

In our process we consider all reviews as milestones of the project. In this phase we have three reviews:

A.5.4.1 Requirements Evaluation

This review is a formal presentation where the team has to present user and system requirements. The analyst is the principal role of this evaluation, so it is recommended that he/she do the presentation. Each team has to do a presentation that should last 20 minutes and should contain:

- A reminder of the problem to address
- A context of the project
- A reminder of the process that the software will support
- The most important user requirements
- The most important screens of the software
- The traceability matrix of user and software requirements

After the team presentation, the instructor asks the audience (course classmates) if someone has a question or something to add. If anything is said, the quality assurance engineer has to take notes. After that, the instructor asks the quality assurance engineer about the presentation and about user and software requirements; the quality assurance engineer has to speak what he/she found about it. At the end the instructor comments the presentation and the identified user and software requirements. The quality assurance engineer has to take notes or record everything because he is responsible for doing the Review Notes of this evaluation.

A.5.4.2 Design Evaluation

This review is a formal presentation where the team has to present everything about the design of the software. The presentation should have a maximum of 30 minutes and contains:

- The solution Architecture
- The data Model
- The detailed design of the main modules
- The system Navigation
- The main System Interfaces
- The traceability matrix

A.5.4.3 Demo Presentation

This review is the formal presentation that closes each Iteration phase. It is the instance where the team has to present what they were able to do in this phase. The presentation is a live demo of the software and it should last no more than 30 minutes. The team may do a walkthrough the use cases addressed in the first iteration. The team should also give a clear view of which user and software requirements were done or not. The software that manages the project (MainReq) should also be shown up to date to reflect what user requirements, software requirement and test cases are complaint, non-complaint or ambiguous.

A.5.5 Outputs

A.5.5.1 Project Plan

The project manager has to create a project plan with the tasks that all the team members have to perform to finish the project on schedule. The project plan has to take in consideration all the other outputs and deliverables of the process. Each task have to be assigned to at least one team member.

A.5.5.2 User Requirements

The User Requirements is an output of the User Requirements Definition activity, but is not a delivery per se. It is part of the Software Requirements Document. The User Requirements should be specified as they contain the information already mentioned in Table A.3. The team just has to update the information that changed between Iteration 1 and 2 and create/delete the User Requirements as needed by the project.

A.5.5.3 Software Requirements

The Software Requirements is an output of the Software Requirement definition activity and it is also a part of the Software Requirement Document. The Software Requirements Document must be created and it should contain the information already mentioned in

Table A.4. The team must also update the information that changed between Iteration 1 and 2 and create/delete the Software Requirements as needed by the project.

A.5.5.4 Test Cases

The Test Cases are an output of the Test Case Specification activity and it is also a part of the Software Requirements Document. A test case should be associated either with a user requirement or a software requirement. Each test case must be specified with the structure already mentioned in Table A.5. The team must also update the information that changed between Iteration 1 and 2 and create/delete the Test Cases as needed by the project.

A.5.5.5 Requirements Review Notes

The Requirements Review Note is an output of the Software Requirement Definition activity. It reports the feedback given by classmates, instructor and teaching assistants to be evaluated after the course ended. All the report reviews should have the same items as mentioned in Table A.6.

A.5.5.6 Design Review Notes

The Design Review Note is an output of the Product Design activity. It reports the feedback given by classmates, instructor and teaching assistants to a later evaluation. The report should have the same items as mentioned in Table A.6.

Table A.6: Review Notes Detail

Item	Description
1	Name of team members that made the presentation.
2	Main considerations of the quality assurance engineer regarding the presentation.
3	Main considerations of the quality assurance engineer regarding the document delivered.
4	All the considerations made to the project during the presentation (made by the instructor, classmates or clients).

A.5.5.7 Demo Review Notes

The Demo Review Note is an output of the Demo Presentation. It reports the feedback given by clients/users, classmates, instructor and teaching assistants to be evaluated when the course ends. The report should have the same items as mentioned in Table A.6.

A.5.6 Deliverables

This phase has three deliverables, they are listed and described below.

A.5.6.1 Software Requirement Document

The main sections and their purpose in the Software Requirement Document are:

- Development team and Counterpart - the document should include names, role and contact email of everyone involved in the project.
- Link to prototype - URLs and necessary user / passwords (if applicable) should be placed here to test the current version of the software developed. In the case of the Conception phase, it should have the URL to the mockup prototype. In case the current version is Iteration 1 or 2, the URL should contain the link to the running software developed until the moment.
- Introduction - This introduction briefly describes the context, objectives and scope of the system being developed.
- Purpose - It describes what the system is and what it does. For small projects half page should be more than enough.
- Scope - Describes how far the project goes, what it is and what it is not.
- Context - Gives information regarding the context of development and the context in which the system is inserted. Technologies that will be involved, previous work, links with other systems, etc.
- Definitions - Places all the definitions or "keywords" to be used in the document, and that the reader does not necessarily know also placing the meaning of all acronyms or abbreviations employed. Both acronyms have to express what the analyst understands and do not necessarily give general definitions. The idea is to just understand the full extent of the document being read and it must be listed in alphabetical order to facilitate the search for concepts.
- References - Lists the documents and literature used as support to build this document. Places dates and versions of documents where appropriate.
- General Description - This section describes the functional requirements of the users/-clients, their external interfaces, exception conditions and types of tests executed to check that the requirements are met.
- Users - In this section the types of users are identified, the general attributes of each type and the number of people in each category.
- Product - It describes customers and user perspectives about the product. Each type of user has different expectations, and each type of client as well.
- Environment - Describes the main components of the operational environment where the solution will be used. This may describe the current scenario, or the future, if equipment will be purchased. The description is made at the level of hardware, servers, databases, networks, etc.
- Related Projects - This sections specifies whether this project is related to some already implemented system, another project underway or planned.

A.5.6.2 Design Document

The main sections and their purpose in the Design Document are:

- Design - This session gives a general design overview of the next sections of the document.

- Physical architecture - The physical architecture is the physical layout of a system and its components in a schema. It refers to some representation of the structure or organization of the physical elements of the system. The development of the physical architecture consists of one or more logical models or views of the physical solution. The logical models or views may consist of conceptual design drawings, schematics, and block diagrams that define the system's form and the arrangement of the system components and associated interfaces. The development of a physical architecture is an iterative process and will evolve together with the requirements and functional architecture.

- Logical architecture - The logical architecture defines the processes (activities and functions) that are required to achieve the user requirements. The Logical Architecture is independent of the technologies and implementations. It can be represented in many different ways: DFD (data flow diagram), use cases, UML diagrams and others.

- Data model - Data modeling is the process of documenting a complex software system design as an easily understood diagram, using text and symbols to represent the way data needs to flow. The diagram can be used as a blueprint for the construction of new software or for re-engineering a legacy application.

- Module details - This section of the document shows what modules the system has, what each one of them does and how they will work and interact between among one another.

- Navigation - It presents the navigation diagram for each user. Each of the diagrams have to show the interfaces that each role can access, it has to show also the number of steps (clicks) that the user has to do to be able to go from one point on the interface.

- Interfaces - It presents the interfaces that system will have. If the system already has some interfaces working (not the mockup) they should be presented here; otherwise it should contain the mockup interfaces of the prototype.

A.5.6.3 Software Product - SPr

If possible, the team should deliver and deploy the software built in this phase to the client, but this in not mandatory. The main idea is to allow the client a chance to get familiar with the new software and be able to fully test all its potential.

A.6 Software Deployment

A.6.1 Introduction

This is the last phase of our process, it is also known as the handover phase. The purpose of this phase is to install the software in the operational environment and demonstrate to

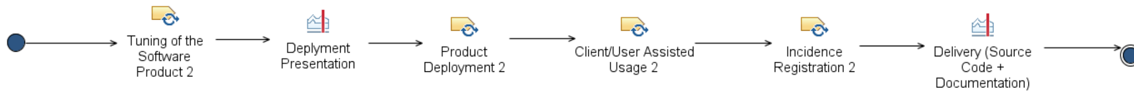


Figure A.16: Deployment

the client/users all the capabilities of the software built. The software has to be completely validated by the user, installed at the client's facilities, configured and migrated. In summary the team has to deliver a complete running software (Figure A.16).

A.6.2 Inputs

The inputs to the Deployment phase are:

- User Requirements (output of Iteration 2)
- Software Requirements (output of Iteration 2)
- Test Cases (output of Iteration 2)
- Requirements Review Notes (output of Iteration 2)
- Design Review Notes (output of Iteration 2)
- Demo Review Notes (output of iteration 2)
- Software Requirement Document (deliverable of Iteration 2)
- Design Document (deliverable of Iteration 2)
- Code - Beta Version (deliverable of Iteration 2)

A.6.3 Activities

The Deployment phase has four activities and two milestones. The activities are described below:

A.6.3.1 Tuning of the Software Product

The team has to take into consideration all review notes as input and make fine adjustments, and some tuning to the software prior to installation.

A.6.3.2 Product Deployment

The team has to install the software and everything needed for the client to use it in the client environment.

A.6.3.3 Client/User Assisted Usage

If some training is needed, someone from the team should help the client/user in their first attempts using the software.

A.6.3.4 Incidence Registration

If the client finds any problem in the software, the client/user has to report the failures or problems to the team.

A.6.4 Reviews

In this phase there is only one review: Deployment Presentation. In this instance the team has to give a live presentation of the software. The team has to show that they have addressed all the points the client/user addressed in the earlier reviews (inputs to this phase) and that they have a final software solution up and running.

A.6.5 Outputs and Deliverables

The Outputs and Deliverables of this phase are the same. The team has to deliver to the client/user and to the instructor the code developed and the documentation of the project. The final documentation of the project can be obtained automatically through our MainReq tool and the code has to be delivered on a CD.

Appendix B

Instructor Software Process Definition

EduProcess is a prescriptive software process that can be followed in practical courses of software engineering. But how to accommodate all these phases, activities, deliveries and everything in a course schedule? To help with that, we have also developed a software process to be followed by the instructors: EduProcess-Instructional.

B.1 Phases, Activities and Milestones

The life cycle model from both process (students and instructional team) is the same. The only thing that change is the point of view of the process and in the roles and what is needed to be done. Table B.1 presents the basic phases of our instructional process.

Our process is divided in five major phases: Preparedness, Conception, Iteration 1, Iteration 2 and Deployment. Each step has his own focus and his own objectives. Since this process is conceived to be used by the instructor, all the activities inside the phases are related to a course calendar with week classes. The full process is available online: (http://www.dcc.uchile.cl/∼mmarques/).

B.2 Preparedness

The Preparedness phase is where the instructor has to find projects for student teams and to allocate students into teams of five to seven students (Figure B.2).

Figure B.1: EduProcess-Instructional

Table B.1: Software Life Cycle Model

Items	**Preparedness**	**Conception**	**Iteration 1**	**Iteration 2**	**Deployment**
Major Activities	Project selection Team allocation	Problem and context identification Prototype construction	Identification of user and software requirements Design definition Software construction	Identification of user and software requirements Design definition Software construction	Tuning Final acceptance tests Deployment
Deliverable Items	Projects Teams	Prototype	Requirements Document Design Document Alpha Demo	Requirements Document Design Document Beta Demo	Documentation Source Code Software
Reviews	-	Problem Presentation Prototype Presentation	Requirements Evaluation Design Evaluation Demo Presentation	Requirements Evaluation Design Evaluation Demo Presentation	Deployment Presentation
Major Milestones	-	Problem Presentation Prototype Presentation	Requirements Evaluation Design Evaluation Demo Presentation	Requirements Evaluation Design Evaluation Demo Presentation	Deployment Presentation Delivery

Figure B.2: EduProcess-Instructional- Preparedness Phase

The following are several requirements that should be accomplished for a potentially useful project-based course supported by EduProcess.

- Type of project: The product to be developed should be an information system for several reasons. This type of system is not usually too complex and is the most frequent type of project developed in industry. They can be addressed in the course available time, although the developers do not have in depth knowledge on the business domain where the problem belongs. Most of the students already have some experience implementing software as part of an academic activity or for the industry; therefore, they are in some way conscious about the activities to perform and challenges to address. There is at least a common knowledge domain that can be expected of these kind of project, since they are all web information systems related somehow to the university, besides the fact that they typically have similar complexity among one another. The students perceive that learning to develop this kind of system is valuable for them.

- Complexity: The project should have a size and complexity similar to the other projects of the course, and it should be possible to develop the product in the available time. Normally, web information systems are not considered very complex projects, because they are mainly visual software project. They are normally projects that have a ratio between backend and front-end where the front-end has a major relevance and the backend has not more than 40% of the development effort. In our context, i.e., fifth-year students, this is a main issue, because we have previous experience showing somehow that students are not so prone to understand the client's problems when they cannot visualize them.

- Client: The project should count on a client and a user with one hour per week to interact with the students. Since students using EduProcess are considered novice, they do not have enough expertise in developing software in teams, or in the business domain of the user/client. Therefore, the client needs to be available to meet the team weekly during the whole semester. Since, students will need different feedbacks and will have different needs from the client according to where they are on the process. A committed client/user can make a difference in projects when the team is not versed on the business domain.

- Time Frame: Provided that the semesters have a fixed duration, the project should be selected and defined to ensure that a reasonable product can be obtained in such a period. Therefore, the size and complexity of the product to be developed should be bounded. The course instructor and the client should agree on a first scope; and then the development team should negotiate the project scope details. The main idea of this first agreement between the instructor and the client is that the project's general scope has to be defined to avoid false expectations on client's side; the students team will have

a project that they can manage and correctly develop within the course time frame.

- Team Allocation - Each student should be allocated to a team, friends preferences should be avoided and the team allocations should ideally be one that tries to create the best team possible according to the number of students and projects.

B.3 Conception

The Conception Phase is divided in weeks of the course (Figure B.3).

B.3.1 Inputs

The only input needed in the Conception phase is that of the student teams enrolled in the semester.

B.3.2 Activities

The activities of the Conception phase are the weeks detailed below. We show details of the first week and some ideas of contents to other classes. The Conception phase ideally can be of 2 to 5 weeks.

B.3.2.1 Week 1

During the first week the process, the evaluation method and the rules of the course are presented. It should be also presented to students the teams they were allocated, as well as the projects and clients they were assigned.

B.3.2.2 Remaining Weeks

During the second to the n (n is the number of the weeks of conception phase the project - from 2 to 5 weeks) week of the conception phase we suggest the topics below to classes:

- Interviews, some suggested topics: interview relevance, open interview, focused interview, validation interview and tips on how to create a bond with the client.

- The class should focus on the main points of software engineering such as, software as a solution of a problem, context of the problem, software engineering as a discipline, big software engineering projects and software engineering horror stories.

- Some suggested topics: What are user requirements, how to write user requirements, how to clarify, and relevance.

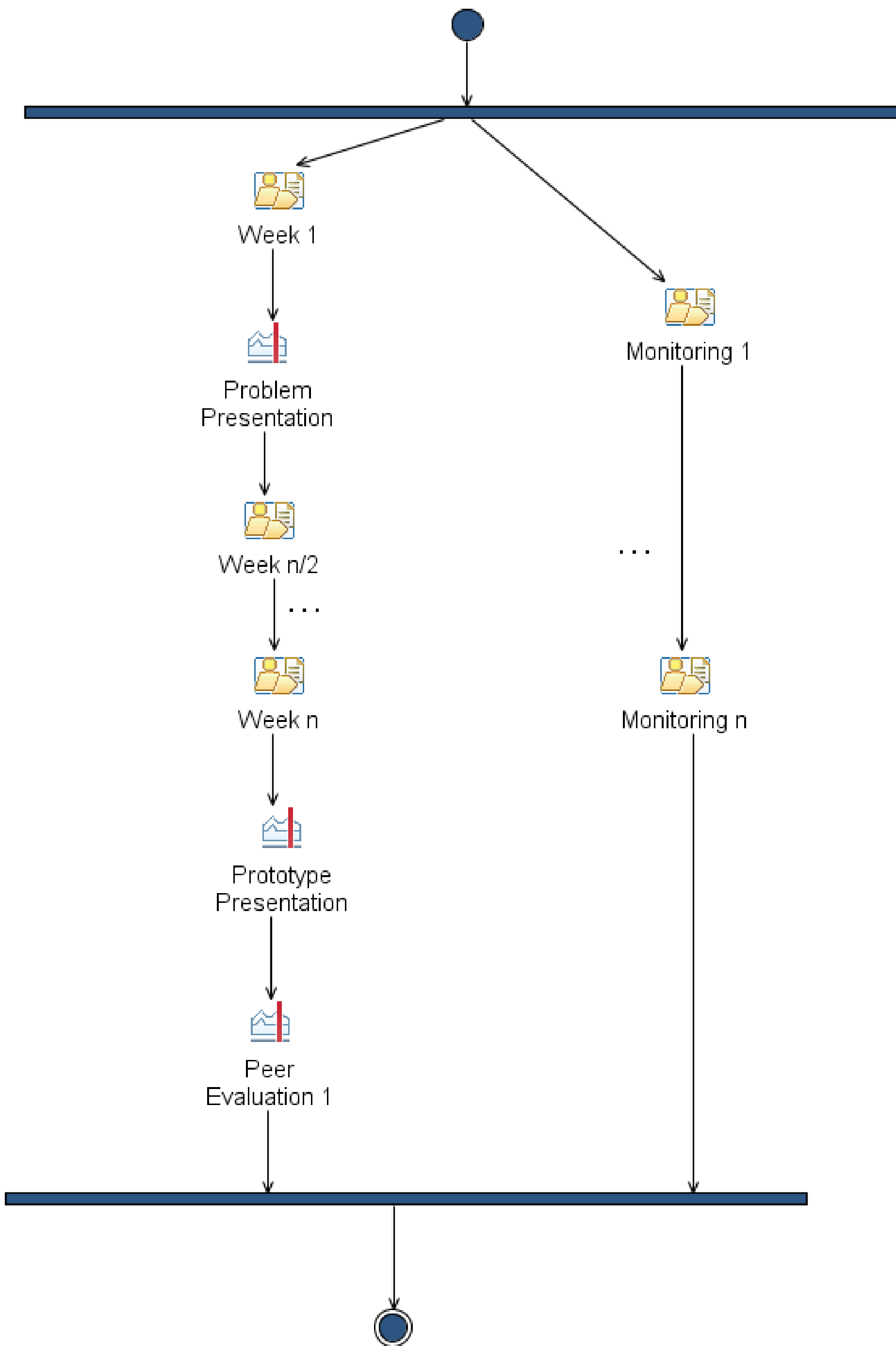


Figure B.3: EduProcess-Instructional - Conception Phase

B.3.3 Reviews

B.3.3.1 Problem Presentation

In this short presentation (10 minutes) the team has to present what problem the client has that needs a software solution. The team should also understand the context of the problem, people/software that will serve as input or use an output of the software. Another important point to evaluate is the restrictions of the solution, if there are any they should be known from the beginning. The presence of the client in the presentation is mandatory. The instructor should evaluate if the presentation has the minimum contents needed and the understanding of the team of the problem.

The presentation should contain at least:

- Team members, clients and users
- Problem
- Context of the Problem
- Objectives (solution main objectives)
- Data model (preliminary)

B.3.3.2 Prototype Presentation

The team should present a mockup interface of what the future software will be. The prototype should have all the interfaces that the user/ client will have at the end of the project. The instructor has to evaluate the interfaces developed by the team to see if they are enough to solve the client's problem, if they are not too much or too little to be done during the course.

B.3.3.3 Peer Evaluation 1

At the end of each phase a peer evaluation is sent to the students. We currently use a web tool (coevaluaciones.dcc.uchile.cl) to evaluate peers. Students are assured that all opinions are anonymous and that there are no formal grades given based on that peer evaluation process. The instrument used to perform the peer evaluation process considers eight items to rate. The rates were indicated using a five-point Likert scale (from "strongly disagree" to "strongly agree") [84]. The evaluated items were the following:

1. He/she assumes the project as a team effort, providing support in project tasks.
2. He/she is able to seek help when problems are found.
3. He/she fulfills his/her tasks properly, working transparently and generating the most value out of each working day.
4. He/she demonstrates initiative to achieve project success.

5. He/she shows a communicative attitude facilitating the teamwork.

6. He/she maintains good coordination between his/her duties and those of the team

7. He/she demonstrates interest in improving performance on the execution of his/her activities and role within the project.

8. He/she is able to admit to mistakes and accept criticism.

B.3.4 Outputs

The outputs of the conception phase are the comments that the instructor gave to students in the Problem Presentation and in the Prototype presentation.

B.3.5 Deliverables

The only deliverable from the Conception phase is the grades of this phase for all students.

B.4 Iteration 1

The Iteration 1 phase is one of the most critical phases of our life cycle in both processes. This phase could also be interpreted as a proof of concept, a proof that the solution (prototype) idealized by the team is possible to develop and deliver to the client. Figure B.4 offers an overview of this phase. In this phase students start working in their specific roles.

B.4.1 Inputs

This phase receives as input the two outputs from the conception phase: Problem Review, Prototype Review and the team role definition.

B.4.2 Activities

This phase is also divided in 1 to 5 weeks.

Some suggested topics to the classes are:

- A practical example of a problem should be presented to students. Students should have to find the problem, the context, elicit and define some user and system requirements. At the end of the class the instructor should ask students about their answers and analyze them in class.

- In this class students should be reminded of the main concepts of design and their representation: UML (activity diagram, use cases), DFD, detailed design, navigability design, screen design among other design related artifacts.

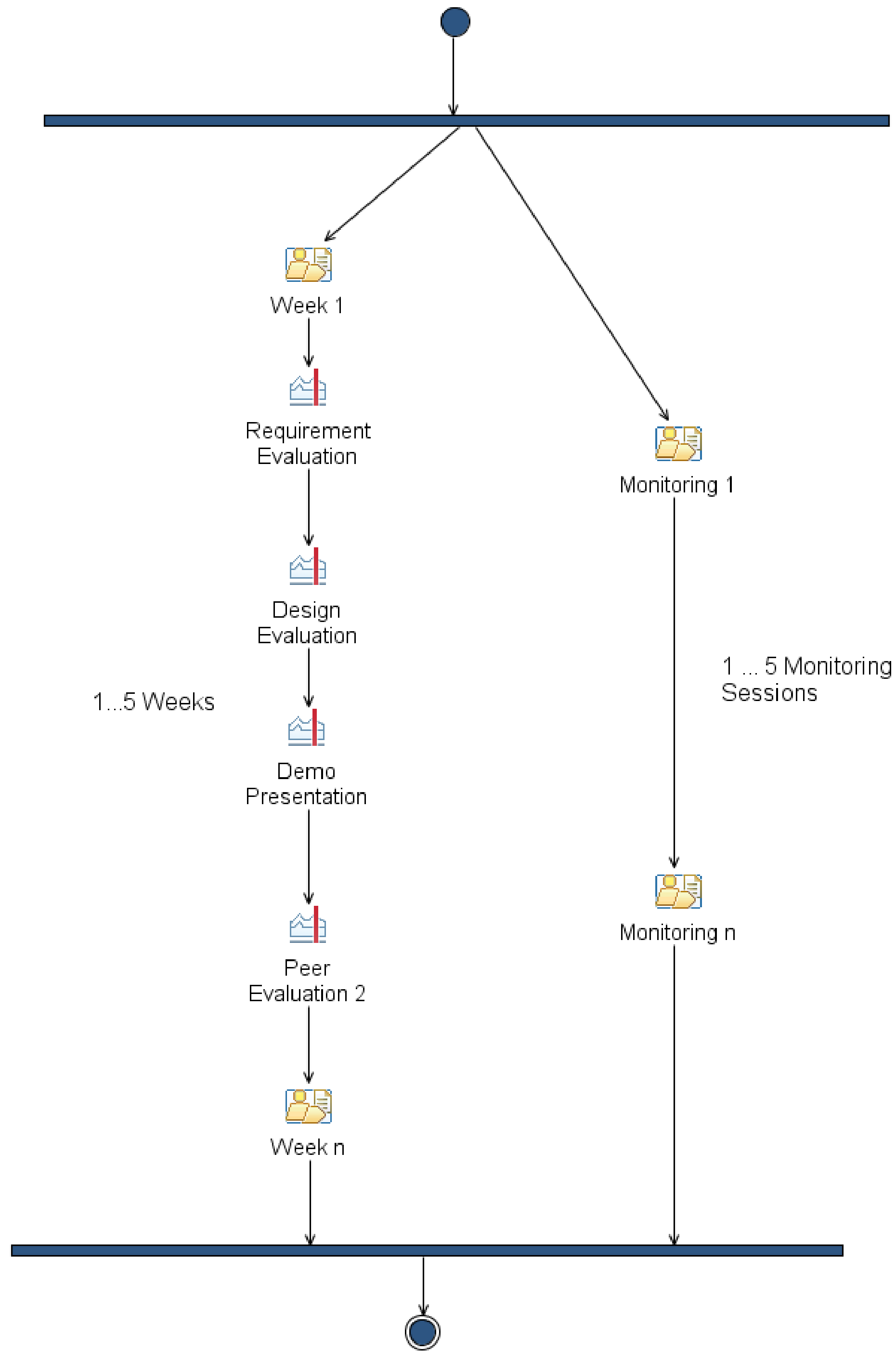


Figure B.4: EduProcess-Instructional- Iteration 1

- Software life cycle, waterfall, agile, RUP, Incremental.
- Team x group, team types: democratic, hierarchic, supervised, mixed teams.
- How to validate software (unit test, integration test, black box, white box, automated test and stress test).

B.4.3 Reviews

In our EduProcess-Instructional we consider all reviews as milestones of the project and it has one review more than the EduProcess. In this phase we have four reviews:

B.4.3.1 Requirements Evaluation

This review is a formal presentation where the team has to present user and system requirements. Each team has to do a presentation that should last 20 minutes and contain:

- Reminder of the problem to address
- Context of the project
- Reminder of the process that the software will support
- The most important user requirements
- The most important screens of the software
- Traceability matrix of user and software requirements

The instructor has to evaluate if the requirements presentation and Requirements Document are clear, complete and enough to start the project development.

B.4.3.2 Design Evaluation

This review is a formal presentation where the team has to present everything about the design of the software. The presentation should has a maximum of 30 minutes.

- Solution Architecture
- Data Model
- Detailed design of the main modules
- System Navigation
- Main System Interfaces
- Traceability matrix

The instructor has to evaluate if the design presentation and if the Design Document are clear, complete and enough to start the project development.

B.4.3.3 Demo Presentation

This review is the formal presentation that closes the Iteration 1 phase. It is the instance where the team has to present what they were able to do in this phase. The presentation is a live demo of the software, it should last no more than 30 minutes. The instructor has to evaluate whether the software is really addressing the demands of the client and if it works properly against the documentation already done.

B.4.3.4 Peer Evaluation 2

The students have to answer another peer evaluation of how their teammates behaved during this phase.

B.4.4 Outputs

The outputs of the Iteration 1 phase are the comments that the instructor gave to students in the Requirements Presentation, Design Presentation and in the Demo Presentation.

B.4.5 Deliverables

The only deliverable from the Iteration 1 phase is the grades of this phase for all students.

B.5 Iteration 2

The Iteration 2 phase is the third phase of our process. The purpose of this phase is to finish and deliver to the client the software he/she is expecting. In Figure B.5 is possible to see an overview of this phase.

B.5.1 Inputs

This phase receives as input the three outputs from the Iteration 1 phase: Requirements Presentation, Design Presentation and in the Demo presentation.

B.5.2 Activities

This phase is also divided in 1 to 4 weeks.

Remaining Weeks
The suggested topics to the classes during this phase are:

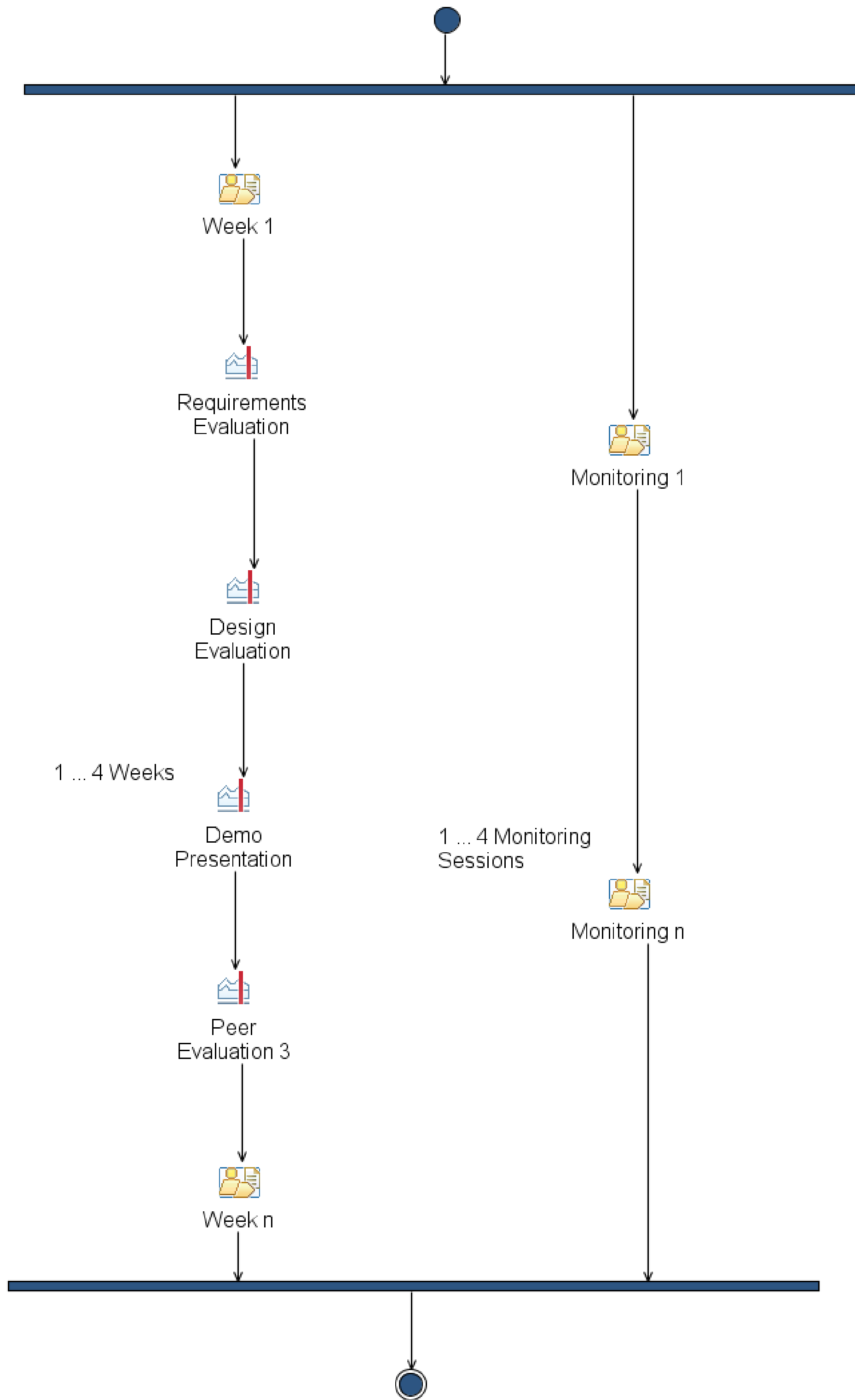


Figure B.5: EduProcess-Instructional- Iteration 2

- Strategies to estimate software: function points, user stories, COCOMO, among other strategies.
- Ethics of software development.

B.5.3 Reviews

In the Iteration 2 phase we have the same reviews of the Iteration 1.

B.5.3.1 Requirements Evaluation

This review is a formal presentation where the team has to present user and system requirements. Each team has to do a presentation that should have 20 minutes. The instructor has to evaluate if the requirements presentation and if the Requirements Document are clear, complete and enough to start the project development.

B.5.3.2 Design Evaluation

This review is a formal presentation where the team has to present everything about the design of the software. The presentation should last a maximum of 30 minutes. The instructor has to evaluate if the design presentation and if the Design Document are clear, complete and enough to start the project development.

B.5.3.3 Demo Presentation

This review is the formal presentation that closes the Iteration 2 phase. It is the instance where the team has to present what they were able to do in the project. The presentation is a live demo of the software, it should last no more than 30 minutes. The instructor has to evaluate the software, if it is really addressing the demands of the client and if it works properly against the documentation already done.

B.5.3.4 Peer Evaluation 3

The students have to answer another peer evaluation of how their teammates behaved during this phase.

B.5.4 Outputs

The outputs of the Iteration 2 phase are the comments that the instructor gave to students in the Requirements Presentation, Design Presentation and in the Demo Presentation.

B.5.5 Deliverables

The only deliverable from the Iteration 2 phase is the grades of this phase for all students.

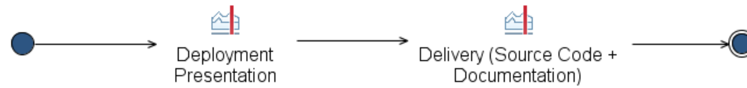


Figure B.6: EduProcess-Instructional- Deployment

B.6 Deployment

This is the last phase of our process, it could also be called the handover phase. The purpose of this phase is to install the software in the operational environment and demonstrate to the client/users all the capabilities of the software built (Figure B.6).

B.6.1 Inputs

This phase receives as input the three outputs from the Iteration 2 phase: Requirements Presentation, Design Presentation and in the Demo presentation.

B.6.2 Activities

This phase has only one activity that lasts one week. The students have to deliver to the client and to the instructor the final software up and running on the clients' environment, the documentation and the code done during the project.

B.6.3 Reviews

The instructor has to review if the outputs from the Iteration 2 phase were taken into consideration and if the software is really up and running on the clients environment.

B.6.4 Outputs

There is no output of the instructor on this phase.

B.6.5 Deliverables

The final deliverable of the instructor is the delivery grades of the project and the final grades of the project (which have to include all the prior grades of each phase).

Appendix C

Reflexive Weekly Monitoring

To deal with the student challenges mentioned in Chapter 3 (student behavior issues and students projects not being bounded by contracts) I propose the Reflexive Weekly Monitoring (RWM), which consists of a monitored weekly meeting with the team. The monitor is neither a team member, nor a Scrum master; he is responsible for warning students in case they are needed; he intends to guarantee that all team members are committed to the project success as equally as possible. During the sessions the monitor, who is not the teaching assistant or the instructor and typically is a person with professional experience in software development - observes the fulfillment of team assignments, the team members behavior, and the feasibility of reaching the project deadlines. However, the focus of each monitoring session is mainly on this last issue.

The RWM sessions take place through the whole project, although they are more useful during the first stages. The sessions should not be perceived by students as an instance of evaluation, but as a formative activity that allows them to improve the software product, project plan and teamwork capability. Therefore, the course instructor or the teaching assistants should not participate in such activity as monitors. People playing the role of monitor in academic scenarios do not necessarily count on an important professional expertise, because the projects being addressed in these courses are usually quite small and not complex. Particularly, graduate students can play this role.

The monitor's questions are simple and involve direct answers, for instance: Has everybody accomplished the assigned tasks? Is our project on schedule? Are we going to be able to reach the deadlines? Is our team working well? These questions are used to perform an initial diagnosis of the project progress, team members' commitment and quality of the teamwork. Team member must all agree on the answers given to the monitor. Finding a consensus pushes the students to analyze various opinions or points of views, and it helps them visualize the team and project current situation. The monitor can ask additional questions until building an initial diagnosis, which is usually adjusted during the next steps of the monitoring session.

Answering these questions requires reflexion and agreement among the teammates. Typically the students only understand the situation when trying to answer these questions. Therefore, this is usually very important for EduProcess. The monitors have to use their own

criteria to keep the monitoring session within certain time limits, typically between 30 and 40 minutes. This means that they have to determine which questions are the most relevant during each session, and thus prioritize them.

C.1 RWM - Structure

The RWM sessions involve three sequential steps (Figure C.1): the *monitor diagnosis*, the team *reflection* and the *closure*. In each session, the monitor asks specific questions following a script according to the course schedule in order to understand the current status of the project and the team. The questions are simple and involve direct answers. For instance: Has everybody reach the assigned tasks? Is your project on schedule? Are you going to be able to accomplish the deadlines? Is your team working well? The monitor script covers most categories involved in Jacobson's approach [68]: opportunity, stakeholders, requirements, software system, work, team, and way of working.

Typically the monitor asks specific questions to team members to evaluate the project status, and based on that, help them discover weaknesses and risks in the work they are performing. In a few cases, the monitor provides feedback and assistance about how to address these issues.

During the monitoring session the monitor ensures that every member can give his or her opinion. The monitoring questions intend to determine if team members are really conscious about the actual problems that they have to address, their importance and timing, and the involved deadlines. The feedback that the monitor provides should allow team members to make their own decisions; and provide general hints more than direct advice. The goal is not to coach team members, but just to play the "psychologist" with them, making them realize by themselves where and how they stand on their project. In Figure C.1 we present the structure of the RWM.

- *Monitor Diagnosis* - The monitoring session starts with two simple questions: "How was the week?" and "Did everybody do what they were supposed to do?" These questions are used to get some general feedback about the project status. After that, the monitor focuses on the following elements: analysis of the project/team situation (i.e. the potential problems affecting the team or the project) and the actions to be made by the team to advance the project towards its final goal.

- *Reflection* - It is set out to determine if the students are conscious of the team and project situation. In order to do that, the monitor performs a reflexive activity based on questions and answers that focus on the weaknesses and risks of the work done, project plan and team member attitudes. The main idea is to put into use the "reflection-in-action" [146] - process that allows for reshaping what the students are working on, while they are working on it. The questions must be formulated considering the monitor as part of the team; it contributes to perceive this activity as formative; e.g., *Why are we delayed? What are our main risks/weaknesses? How can we address these risks/weaknesses? How can we improve our team performance? What can we do to reach the deadlines?*. According to the Diagnosis step the monitor checks if team members are aware of the project situation (possible project risks) and in line with that

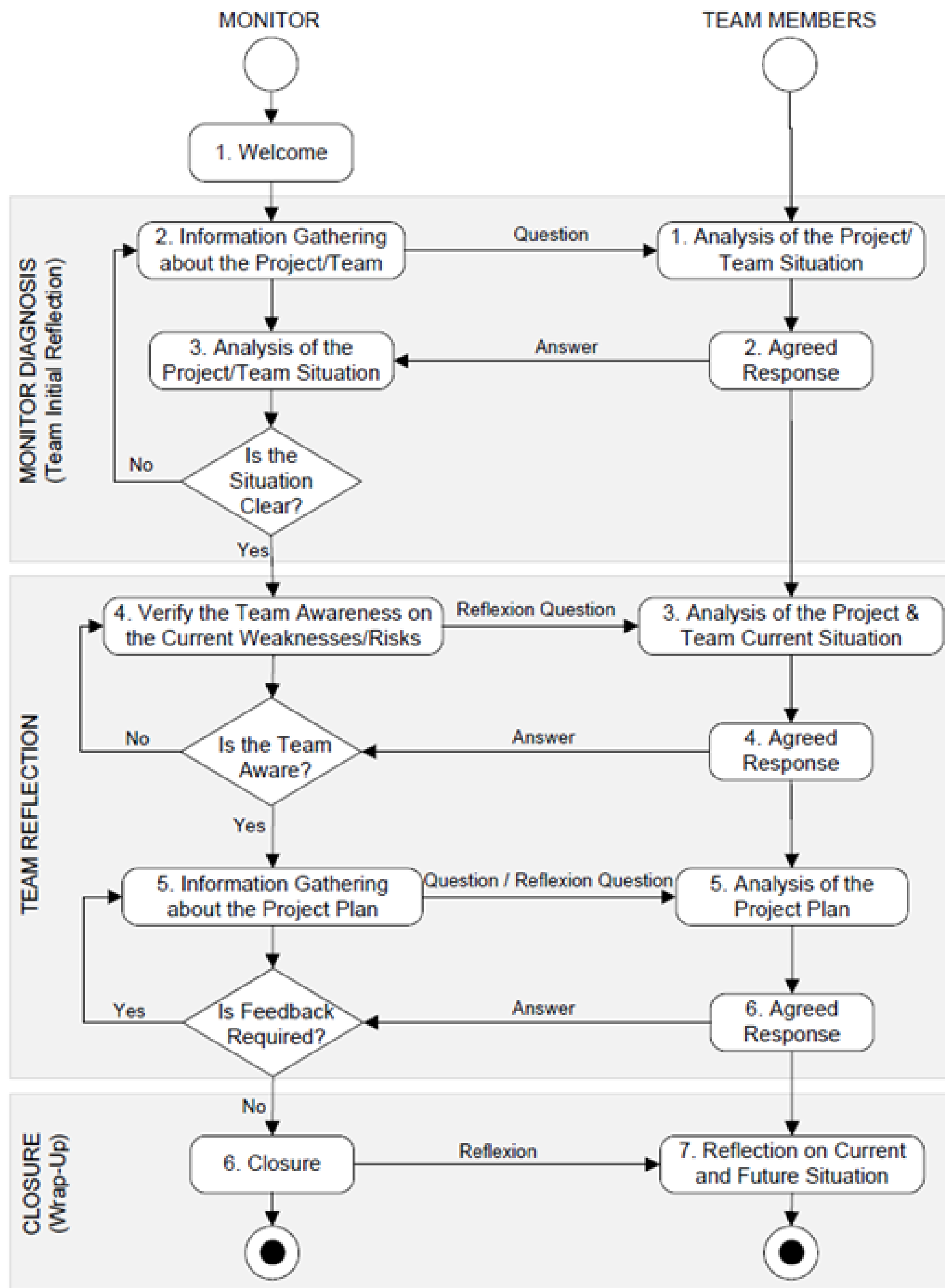


Figure C.1: Structure of a RWM Session.

the monitor makes concrete questions motivating them to think, analyze and reflect about it, i.e., a team member reports that there is some conflict in a decision about the design of something in the project; the monitor asked questions "What are the advantages of each possibility?"; "What are the costs to the project in terms of time spent with each possibility?" The monitor in these specific case suggests to students to make a short list of pros and cons where they have to analyze and evaluate the situation and the possibilities. As a result the team has to agree upon a strategy to solve the problem/situation.

At the end of this second step the monitor asks students about the project plan to

see if it is reasonably within schedule. Eventually, the monitor can offer something to think about such as, indirect feedback, which will be discussed during the following monitoring

- *Closure* - In the closure the monitor asks students about the next steps of the project to see if they are reasonable, and if the project is within schedule. The session ends with a wrap-up with the priority actions that the team needs to address within the next week consisting of a wrap-up made by the development team that highlights up to three priority actions that should be addressed for the next week. The closure acts as a trigger to create short-term tasks and assignments. If the RWM sessions are applied in long projects, the closure should also consider the priority actions for the medium term. It is recommended that the team self-organizes to address the identified tasks and determines the way they would be carried out. The self-organization requires reflexion-in and on-action [155] to determine the team next steps. This process design is inspired in the guidelines given by Marks et al. [93].

C.2 Monitoring Schedule

According to the two processes that we proposed (EduProcess and EduProcess-Instructional) and the course milestones we created a set of suggested questions that the monitor can ask students. In our process we considered eleven monitoring sessions and we present below the suggested questions that can be used according to the milestones defined. We have seven monitoring sessions that are mandatory and four monitoring sessions that are optional. The instructor should evaluate the need for the monitoring sessions according to the course schedule.

C.2.1 Conception

The questions and the amount of monitoring sessions during the conception phase should be 2 to 4. And according to this we divided the questions the monitors should ask.

- Monitoring Session 1 (mandatory)- First meeting between the team and the monitor. The main purpose of this monitoring session is to evaluate the team meetings with the client. The questions that the monitor can ask are:
  - How long was the duration of the meeting?
  - What was the objective?
  - What specific questions were asked?
  - Who were the participants present on the meeting?
  - Did anyone of the team remember to take notes?
  - Did anyone of the team managed to gather user requirements?

- Does the team already have an idea of the information flow looks like?
- Does the team have any idea of what is the client problem?
- What is the context of the problem?
- Has the team already started preparing the problem presentation?
- Were all team members present?

- Monitoring Session (optional) - The purpose of this session monitoring is to continue to evaluate client meetings. The main questions that the monitor can ask:
  - Has the development of the prototype started?
  - Can the team present something tangible of the prototype?
  - The monitor should ask the team to show him/her the prototype.
  - The team already presented the prototype to the client? The client gave feedback about it?
  - The team already have user requirements elicited?
  - The team has any ideas of possible software architecture?
  - Everyone knows what you have to develop?
  - The team already has the roles assigned?
  - Observations

- Monitoring Session 2 (mandatory) - The main purpose of this monitoring session is to evaluate prototype development. The main questions that the monitor can do:
  - Development of the prototype is already being done?
  - The team can present something tangible of the prototype?
  - Ask them to show prototype
  - The team already presented the prototype to the client? The client gave feedback about it?
  - The team already have the user requirements elicited?
  - The team has any ideas of possible software architecture?
  - Everyone knows what you have to develop?
  - The team already has the roles assigned?

C.2.2 Iteration 1

The questions and the amount of monitoring sessions during the conception phase should be 2 to 5. And according to this we divided the questions the monitors should ask.

- Monitoring Session 3 (mandatory) - The main purpose of this monitoring session is to evaluate user and system requirements . The main questions that the monitor do are:
  - The user requirements are already defined?
  - The user requirements prioritization was already negotiated with the client?
  - The team already defined what are the requirements for iteration1.
  - The team already inserted requirements in requirements manager tool?
  - The analyst already did the translation of user requirements to software requirements?
  - So far, how are the team relationship going?
- Monitoring Session 4 (mandatory) - The main purpose of this monitoring session is to evaluate the design activity. The main questions that the monitor can ask:
  - The team already have a defined idea of the design?
  - The team have a clear picture of the information flow, the type of users and the language that the software will be built?
  - How many of the team members know the development language?
  - The use cases are already done?
  - The detailed design is done?
  - And the data model?
  - Ask to take a look on what they have done so far.
  - How many team members are present?
  - As monitor can you detect or anyone is complaining about possible free riders?
- Monitoring Session (optional) - The main purpose of this monitoring session is to evaluate the test cases. The main questions that the monitor can do are:
  - The team already have defined test cases?
  - How many test cases are defined?
  - Ask to see the test cases

  - Ask to take a look on what they have done so far.
  - How many team members are present?

- Monitoring Session (optional) - The main purpose of this monitoring session is to evaluate the overall progress of the project. The main questions that the monitor ask are:
  - How the project is going in general?
  - What is the relationship between the client and the team?
  - What is the relationship among team members?
  - What are the biggest risks that team see to the project?
  - The team believes they will finish what they committed on time?
  - All the team members were present?

- Monitoring Session (optional) - The main purpose of this monitoring session is to evaluate to evaluate the Iteration 1 presentation progress. The main questions that the monitor can do are:
  - Is the demo for Iteration 1 ready?
  - The demo was validated with the client?
  - The team delivered everything that was committed to the Iteration 1?
  - What were the biggest problems the team had in this iteration?
  - All team members were involved with the project in time?

C.2.3 Iteration 2

The questions and the amount of monitoring sessions during the conception phase should be 3 to 5. And according to this we divided the questions the monitors should ask.

- Monitoring Session 5 (mandatory) - The main purpose of this monitoring session is to evaluate user and software requirements. The main questions that the monitor can do are:
  - User requirements for the second iteration are already defined?
  - The negotiation with the client of the user requirements prioritization was done?
  - The team already defined what the core requirements will be for iteration 2?
  - The team already put some requirements in the software?

  - The analyst already updated all the requirements of the first iteration?
  - If the client came up with new requirements the team was able to negotiate?

- Monitoring Session 6 (mandatory) - The main purpose of this monitoring session is to evaluate design. The main questions that the monitor can do are:
  - The design of what was done is the design presented on iteration 1?
  - What changed in the design?
  - The data model is reflecting what was being done?
  - What are the main challenges that the team has to finish the project?
  - The team believes they will be able to finish everything committed?
  - How many team members are present?

- Monitoring Session 7 (mandatory) -The main purpose of this monitoring session is to evaluate the Iteration 2 presentation. The main questions that the monitor can do are:
  - Is the demo for Iteration 2 ready?
  - The demo was validated with the client?
  - The team delivered everything that was committed to the project?
  - What were the biggest problems the team had in the project?
  - All team members delivered their tasks in time?
  - How team members feel about the experience of working in teams?

Appendix D

SSP versus EduProcess

In this Appendix we briefly present SSP (Simple Software Process) [115] and a comparison with EduProcess.

D.1 SSP

The first version of the Simple Software Process (SSP) was created in 1998 to support software development projects in the academia. SSP has two increments: Core and Complement and each one of this increments has four phases as shown in Figure D.1. The first increment involves about 70% of user requirements and 100% of quality requirements. The second increment addresses the residual user requirements, which usually are not clear by the time the project starts.

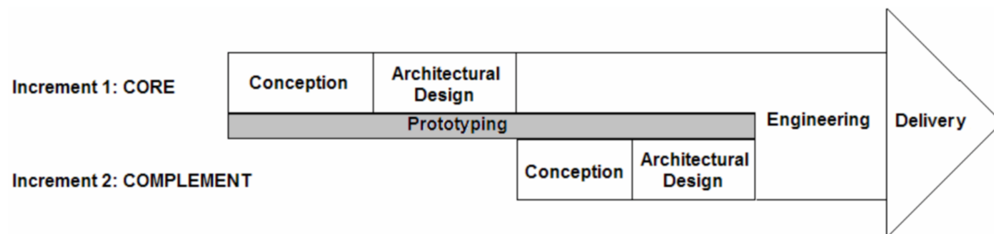


Figure D.1: Structure of SSP

D.1.1 Phases

SSP works in a parallel model with low interactions among team members which is fundamental definition in SSP.

- *Conception* - This phase has two goals: to define the project viability, and to specify the user requirements which will guide the development process.

- *Architectural Design* - In this phase the team has to define the product structure in terms of subsystems, relationship among subsystems, information structure, system

navigation, basic look-and-feel and also specifies the operational environment of the system.

- *Engineering* - This is the construction phase. During the development of the first increment, the programmers implement as much as possible in order to reduce the risks and to validate the usability of the software. During the second increment, the additional functionality is implemented.

- *Delivery* - The delivery phase focuses on installing the product in the user/client environment.

D.1.2 Roles and Context

SSP has six roles to assign to team members: *project manager, analyst, designer, programmer, tester* and *user/client*. The context where SSP was used were courses taken by advanced undergraduate and graduate students of computer science. Students were grouped in teams of 4-6 people and a real project was assigned to each team. The projects involved participation of real clients and users. Each team had 16 weeks to develop and deliver the final product.

It has three defined main check points for each project upon finishing the conception phase during first increment, upon ending the conception phase during the second increment, and upon completing the engineering phase during both increments (core and complement).

In Section 2.2 of the thesis we presented Paula [123], a set of "minimal" requirements for a software process for use in an academic scenario. We presented a comparison of EduProcess with these requirements in Chapter 4. The same analysis was applied to SSP:

- *Process architecture* - All the educational process is not clearly defined and it is not specified.

- *Team orientation* - SSP was designed to be used by small software teams, from four to six students.

- *Project cycle time* - The cycle time duration of SSP is of to sixteen weeks.

- *Standards and practices* - SSP uses SE standards and best practices; i.e.; detailed design, requirements and others.

- *Student support* - The process does not offer any student support.

- *Instructor support* - SSP does not offer support to instructors.

D.2 Comparison

In Table D.1 we present a comparison between SSP, EduProcess and the set of "minimal" requirements for an academic software process

Table D.1: Comparison of SSP and EduProcess with the Set of "Minimal" Requirements for an Academic Software Process

"Minimal" Requirement Item	**SSP**	**EduProcess**
Process architecture	All the educational process is not clearly defined and it is not specified.	All the educational process is clearly defined with its inputs, activities, outputs, deliverables and milestones. EduProcess is specified in detail and is available online.
Team orientation	SSP was designed to be used by small software teams, from four to six students.	EduProcess was designed to be used by small software teams, from four to seven students.
Project cycle time	The cycle time is sixteen weeks.	The cycle time can be from seven to fifteen weeks.
Standards and practices	It uses SE standards and best practices; i.e.; detailed design, requirements and others.	Uses SE standards and best practices; i.e.; requirements, design, testing and others.
Student support	The process does not offer any student support.	The process has guidelines and templates to help students to understand what they have to accomplish.
Instructor support	It does not offer support to instructors.	It has a specific track that deals with the need of instructors support.

Despite the difference in the requirements there are other differences between SSP and EduProcess. The most important ones in relevance are:

- EduProcess is a specified process with guidelines and examples that can be used by other instructors; SSP is not specified.
- Conception phase with a non-functional prototype that needs to be validated and agreed upon between the team and the client; SSP does not have it.
- Reflexive Weekly Monitoring sessions that help students coordinate and analyze their project; SSP does not have it.
- Peer-assessment used as feedback tool to support teamwork; SSP has a small version and it was not mandatory.
- Client meetings, in EduProcess the client has to meet the team at least once a week; in SSP it was not mandatory.
- Evaluation, in EduProcess the evaluations are done according to the role of the student

in the review process and there are more evaluations. In SSP there are only three evaluations and the team was equally evaluated.

- Roles, in EduProcess there are fewer roles than in SSP.

Appendix E

A Systematic Mapping Study on Practical Approaches to Teaching Software Engineering - FIE 2014

A Systematic Mapping Study on Practical Approaches to Teaching Software Engineering

Maíra R. Marques
Computer Science Department
Universidad de Chile
Santiago, Chile
mmarques@dcc.uchile.cl

Alcides Quispe
Computer Science Department
Universidad de Chile
Santiago, Chile
aquispe@dcc.uchile.cl

Sergio F. Ochoa
Computer Science Department
Universidad de Chile
Santiago, Chile
sochoa@dcc.uchile.cl

***Abstract*— Background: Software engineering is a core subject in computing education. Today, there seems to be a consensus that teaching software engineering requires students to perform practical experiences that simulate the work in the software industry. This represents a challenge for universities and instructors, because these experiences are complex to setup and involve considerable time and effort. Although there are several experiences and proposals reported in the literature, there is no clear solution to address this challenge. Aim: Being knowledgeable about the several approaches reported in the literature for dealing with this challenge is the first step to proposing a new solution. Counting on this knowledge allows instructors to reuse lessons learned from other universities. In order to address this challenge, we conducted a systematic mapping study that intends to answer the following questions: What are the main approaches used to address the practical experiences in software engineering education? Is there an emerging tendency to address this challenge? Which software process models are used to support the practical experiences in software engineering courses? Have the universities changed the way of conducting these experiences over the years? What are the main forums to seek information on practical approaches for teaching software engineering? Method: We used a systematic mapping study to identify and classify available research papers that report the use of practical experiences in software engineering education.**

Results: There were 173 papers selected, analyzed and classified. The results indicate that universities have realized the value of including practical experiences as part of the software engineering teaching process. However, few proposals indicate how to address that challenge. The practical approaches identified in this study were game learning, case studies, simulation, inverted classrooms, maintenance projects, service learning, and open source development. Only one recent report on the use of traditional approaches (i.e., teaching using expositive lectures) was found. The use of a development process to support these practical experiences seems not to be a concern for software engineering instructors. Only 40% of these studies report the use of a development process to guide the process experience. The reported processes are mainly agile methods. Conferences are the most used forum to publish studies in this area (72%). One third of these studies have been published over the last five years.

Conclusion: There is a clear concern for teaching software engineering involving practical experiences, and there are several initiatives exploring how to do it. The map gives us an overview of the different proposals to address this challenge, and also allows us to make some preliminary conclusions about the preferred approaches.

Keywords—software engineering education; systematic mapping.

I. Introduction

Software engineering is a discipline designed to apply systematic, disciplined, quantifiable approaches to the development, operation and maintenance of software [1]. As any engineering discipline, it is tightly linked to the industry. Consequently, there is a need to prepare students for the world of industrial software development [2].

Preparing software engineers for the industry is a complex task that most universities are striving to accomplish. Such a task must not only transfer knowledge on core-computing concepts, but also allow students to become lifelong learners who are able to keep pace with innovations in the discipline [3]. The approaches to teaching the theoretical aspects of software engineering have been well supported by the universities. However, software engineering needs a more practical teaching approach than just expositive classes.

Moore and Potts [4] state "*Software engineering cannot be taught exclusively in the classroom. Because software engineering is a competence, not just a body of knowledge, any presentation of principles and experience that is not backed up by active and regular participation by the students in real projects is sure to miss the essence of what the student needs to learn*". Practice-based software engineering seems to be the best instructional approach to deal with the contemporary software engineering education [5]. Unfortunately, not all computer science and software engineering programs offer experimental courses on software development that allow students to participate in a real life software development experience.

Integrating these experiences into the courses is a huge challenge for universities. This demands significant effort in terms of time and human resources to support the instructional process, such as utilizing coaches). These experiences also need instructors with expertise in software development, and they frequently require real clients that act as counterparts in

these projects. These restrictions make this challenge difficult, if not impossible to address for most universities.

There is currently no predominant approach, nor right or wrong way to conduct these practical experiences, but there are certainly approaches that are more effective than others depending on the teaching context. For these reasons we have conducted a systematic mapping study to determine which approaches are available, and which of them seem to be preferred or are more effective. Our study was conducted following the procedure proposed by Petersen et al. [6]. Systematic mapping studies are intended to give an overview of a specific topic; in our case, it is the practical approaches for teaching software engineering. A systematic mapping study is typically classified as a secondary study and indicates the quality and quantity of work already done by the research community in the area. The papers used in the mapping study are selected and classified according to the research questions proposed in indexed scientific databases, and the papers selected are called primary studies.

One of the main goals of a systematic mapping study is to categorize an area of knowledge providing evidence that there is information and possible clusters where further research could be done.

The rest of the paper is organized as follows. Section 2 describes the paper's selection process used in this work and it explains the process conducted to perform the mapping. Section 3 reports the obtained results. Section 4 discusses the threats to validity of this systematic mapping. Finally, Section 5 presents the conclusions of this study.

II. Study Research Selection

In this study we followed the process proposed by Petersen et al. [6] (Figure 1). Next, we describe in detail how the process steps were performed.

A. Definition of Research Questions

The research questions to be answered in this systematic mapping study are the following:

RQ1: *What are the main approaches used to address the practical experiences in software engineering education?* The idea was to create a comprehensive map with the main instructional approaches, and identify those most commonly used.

RQ2: *Is there an emerging tendency to address this challenge?* This question tries to determine if the lessons learned from past experiences have generated consensus about the best ways to teach software engineering from a practical point of view.

RQ3: *Which software process models are used to support the practical experiences in software engineering courses?* The use of a particular process model involves certain challenges and usage efforts for the instructors. This research question intends to determine which models can be afforded by the instructors during a course, considering the models complexity, and also the time and effort required for using them.

RQ4: *Have the universities changed the way of conducting these experiences over the years?* This question intends to determine the evolution of the software engineering teaching approaches, in order to identify promising trends that help researchers and instructors to define new solutions to address the stated challenge.

RQ5: *What are the main forums to seek information on practical approaches for teaching software engineering?* By identifying the main information sources on this topic, we intend to ease the searching and retrieving of new proposals on this research topic.

Although there are several proposals on how to teach software engineering, there is no clear consensus on whether or not there is a better approach to do it. The literature shows a multitude of anecdotal information indicating what is best and what everyone does. However, there is no empirical evidence supporting these suggestions. Therefore, this systematic mapping study intends to provide a more comprehensive and objective point of view on the practices used by universities to address the practical experiences in software engineering courses.

B. Literature Search

We started our mapping study by identifying keywords and search strings that would be useful in finding relevant articles. We followed the suggestions of Kitchenham et al. [7] who recommend: (1) breaking down the research questions into concepts, (2) finding synonyms, abbreviations and alternative spellings, (3) considering subject headings of relevant journals and works in the area, (4) using Boolean operators in the search strings.

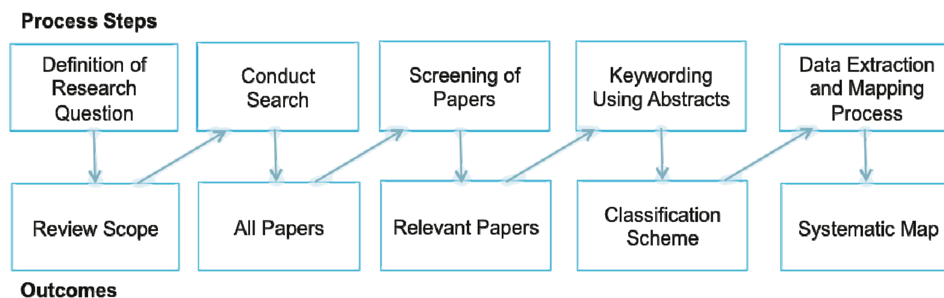


Figure 1 - Systematic Mapping Process by Petersen et al. [6]

Table 1 shows the search strings that were used to identify relevant papers. In such a process we matched these strings against the papers title, abstract and keywords, when possible. The following scientific databases were used in this searching process: IEEEXplore, ACM Digital Library, Web of Knowledge (former ISI Web of Science), Science Direct (Elsevier), SpringerLink and Wiley International. These databases are relevant sources, indexing the major journals and conference proceedings in the software engineering area. They also allow automatic searching.

Table 1 - Strings Used in the Literature Search

Search Strings
"software engineering education" **OR**
"software development course" **OR**
"software development capstone" **OR**
"software development education" **OR**
"software engineering capstone" **OR**
"software engineering course" **OR**
"teaching software engineering" **OR**
"software engineering instruction" **OR**
"teaching software development" **OR**
"software practice education" **OR**
"software engineering projects"

The search was done from October 1st to 30rh ,2013. A total of 7,517 documents were found. Table 2 shows the number of articles retrieved from each database. Provided that many papers are indexed by more than one digital library, we removed the duplicated articles and thus obtained a corpus of 3,725 documents related to the research topic. This corpus of literature represents the input used for the next phase of this study (i.e. the screening). Figure 2 shows the percentages of papers used from each literature database.

Table 2 - Amount of Articles Retrieved by Source

Source	Retrieved
ACM Digital Library (http://dl.acm.org)	2,100
IEEEXplore (http://ieeexplore.ieee.org/Xplore/home.jsp)	996
Web of Knowledge (http://www.webofknowledge.com/)	2,279
SpringerLink (http://www.springer.com)	934
Science Direct (http://www.sciencedirect.com)	2,075
Wiley International (http://onlinelibrary.wiley.com)	1,212
TOTAL:	**7517**

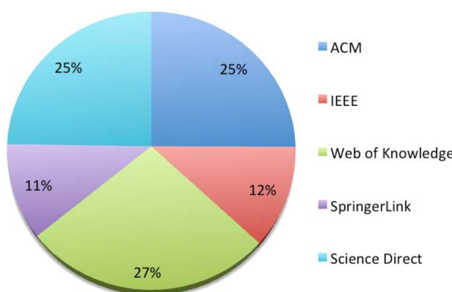


Figure 2 – Representativeness of the literature sources

C. Screening of Papers and Keywording

The main goal of the screening stage is to select studies that address the stated research questions. In our screening process, two inclusions criteria and eight exclusions criteria were used and presented in the following list:

Exclusion Criteria:

1. Documents whose full text is not available.
2. Documents that are only available as an abstract.
3. Documents describing a panel.
4. Documents describing a workshop.
5. Documents that are a letter from the editor.
6. Documents in which the context is not related to teaching software engineering.
7. Documents that report the use of software engineering as a means of teaching a specific programming language or framework.
8. Documents that report opinion papers and philosophical theories.

Inclusion Criteria:

1. Documents reporting teaching/learning software engineering experiences.
2. Documents presented as full papers; short papers and works in progress are counted if they are within the research scope.

The authors individually applied the inclusion and exclusion filters, and then examined the title, abstract and keywords of each selected paper. The results were later compared and the conflicts were discussed between the authors. A total of 247 papers were selected after this first filter. Then, a skimming through the introduction and conclusion sections of each paper was conducted in order to select the primary studies that help answer the research questions. As the result of this skimming, 173 papers were selected as primary studies, which were used for the keywording stage.

According to Petersen et al. [6], "*keywording is a way to reduce the time needed in developing the classification scheme and ensuring that the scheme takes the existing studies into account".* This stage involves two steps: (1) look for keywords that reflect the concepts and contribution of the paper, and (2) combine the keywords to classify and understand the contribution of the research. Following this proposal, we defined three criteria for classifying our primary studies:

Research type – It identifies the research approach used in the studies reported in the papers. Table 3 shows a description of the research type categories used to classify the articles. These categories are based on those proposed by Petersen et al. [6] and Nacimento et al. [8].

Teaching approach – It identifies the pedagogical teaching approaches used in these practical experiences. Table 4

presents the categories of teaching approaches and their description.

Software process model – It indicates the type of software process used to guide these experiences. Software development processes are "*used as guidelines or frameworks to organize and structure how software development activities should be performed, and in what order*" [9]. Table 5 shows the categories of software processes used to classify the experiences, and also their description of each category.

Table 3 – Research Types

Category	Description
Experience report	Author´s personal experience describing what and how something was done. This usually includes a lesson-learned section.
Case study	Report of investigation of a real-life phenomenon. It may include quantitative evidence, relies on multiple sources of evidence, and also may report one or several cases.
Action Research	Problem solving actions implemented in a collaborative context with data-driven analysis or research, with the intent to enlighten underlying causes enabling future trend prediction.
Experiment/Quasi-Experiment	Manipulation and controlled testing for causal analysis. Normally, one or more variables are manipulated to determine their effect on the dependent variable.
Solution Proposal	A solution is proposed and the results of using the proposed solution are reported. An example or a line of argumentation must show the applicability of the solution.

Table 4 - Teaching Approaches

Category	Description
Case Studies	Students develop skills in analytical thinking by reading and discussing complex real-life problems.
Learning by doing	There is no formal reported approach stated in the paper, but report the experiences on using an approach where students take on a project and have to address the clash of software engineering theory and practice.
Game based learning	A game development is proposed as a fun way to teach software engineering and catch the students' attention.
Maintenance	Students have to learn how to deal with software created by other developers, and to participate in its evolution or correction.
Open Source	Students have to learn software engineering by participating actively in an open source community or project.
Problem/Outcome Based Learning (PBL/OBL)	These experiences address practical problems as way to drive the learning process. They can also be focused on getting certain outcomes as a way to emphasize a more functional knowledge than a declarative one. Students learn about a subject through the experience of solving a problem, looking for results and by working in groups.
Simulation	Students work on projects that try to simulate a part of a real life project. For instance they have to play a role, deal with a client, address a problem, and work in an environment similar to industrial scenarios.
Traditional	Involves a lecturer standing in front of students, with the course content divided into a number of topical lectures. Typically, a set of practical assignments is used to help students apply the theoretical concepts to the professional work.
Service Learning	Students develop and use their academic skills to address real life problems within their own environment.
Inverted Classroom	The traditional lecture is placed online for students to utilize during their study time. Their classroom time is used for other activities, such as workshops, interactive work time with the instructor, or demonstrations.

Table 5 - Software Process Models

Category	Description
Agile	The agile models are product-driven processes that propose a software development approach based on customer collaboration, iterative development and technology adaptable to change [10]. Little or no bureaucracy is involved in these processes.
CMM	CMM (Capability Maturity Model) is a general recommendation that identifies key process areas, and also different levels of work maturity for those areas. These experiences involve planning, performing or validating software process improvement using this recommendation as a guideline.
Mbase	Mbase (Model Based Architecting and Software Engineering) is a particular software process model, which is focused on managing the traceability of the project deliverables.
RUP (based)	Works that report the use of RUP (Rational Unified Process) or a variation of it to guide the development of software projects. Some processes derived from RUP are UP (Unified Process) or UPEDU (Unified Process for Education).

TSP/PSP	Papers that report on the use of TSP (Team Software Process) or PSP (Personal Software Process) [11]. These processes are focused on self-measuring and managing the development effort during a project.
Traditional	Works that report the use of waterfall, incremental or any other structured approach for software development. Typically these processes are opposite to agile methods in terms of bureaucracy required to obtain a product.
Various	Papers that report the use of more than one process approach simultaneously. For instance, using a structured model for developing a particular software. We however use an agile approach to address each process stage.
Ad-hoc (stated)	Reports that use a specific process approach, which is described in that work, and that do not match with previous categories.
Not specified	Authors do not state the process approach used to address the work.

D. Data Extraction

The data extraction process conducted in this study was particularly designed to answer five research questions. In order to do that, each paper that passed the screening process was read to determine the correct classification of the work according to the defined criteria. Moreover, we retrieved the following information from those papers: title, year, venue and author(s). The information required to classify the articles were in some cases stated in the article, and in other cases, they were manually classified by the authors considering the papers' content. In this work, none of the papers were classified in more than one category according to a same criterion.

III. Obtained Results

The details of the 173 papers selected as primary studies are available in [12]. The results are presented according to the stated research questions. Figure 3 gives us an overview, a map of the approaches reported in the primary studies. The bubbles indicate the number of papers classified according to the already defined criteria; i.e., research type, software process model and teaching approach. Next, we present the obtained results for each research question.

***RQ1**: What are the main approaches used to address the practical experiences in software engineering education?*

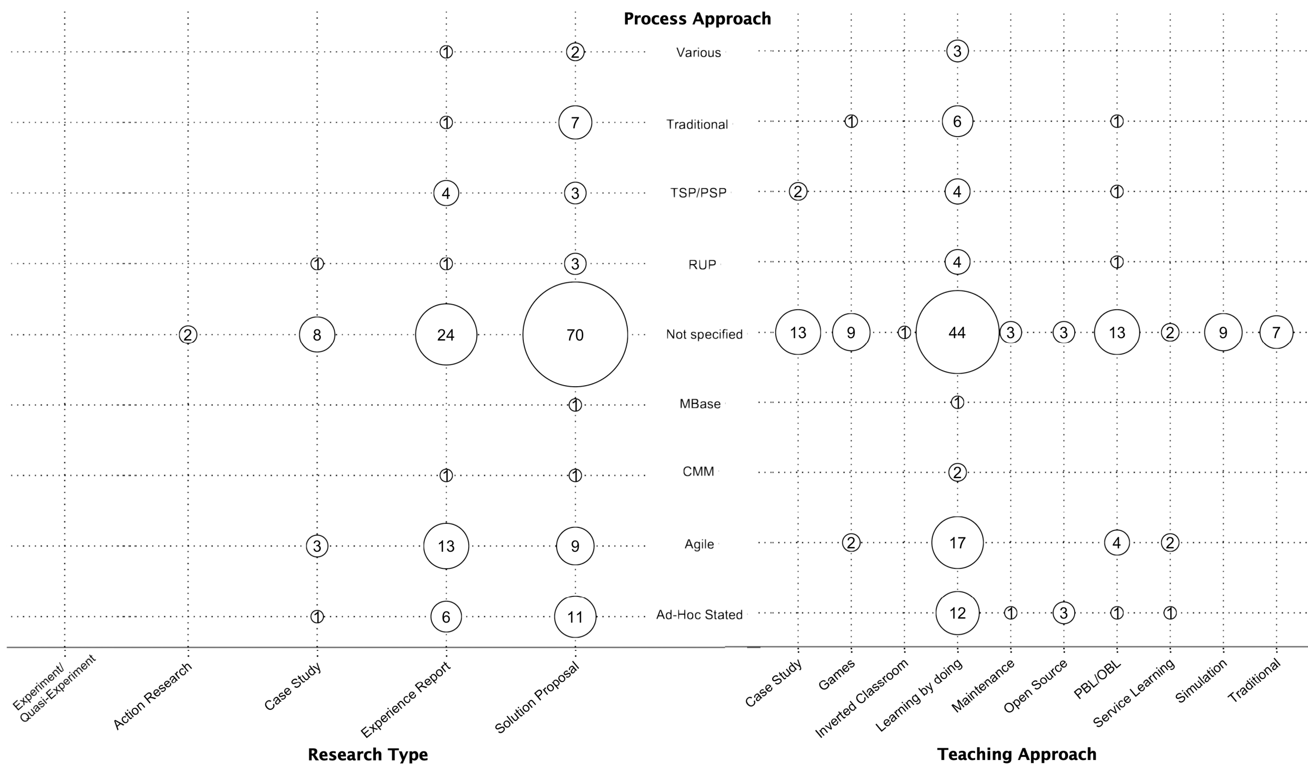


Figure 3 - Research Type versus Process Approach versus Teaching Approach

The literature reports ten instructional approaches to addressing these practical experiences. Clearly, the variety of alternatives is small.

The map shows that the *learning by doing* approach is the most used one, with 93 studies (54%), followed by the *PBL/OBL* approach with 21 studies (12%), *case studies* with 15 studies (9%), *games* with 12 studies (7%), and then *simulation* (5%), *traditional* (4%), *open source* (3%), *service learning* (2%) and *inverted classroom* with only one study.

Learning by doing and also PBL/OBL are flexible strategies that can be adjusted to take advantage of the capabilities and interests of the instructors and students. These approaches do not impose important constraints or require special capabilities of the instructors. This could be a reason why these alternatives seem to be frequently used to support these practical experiences.

RQ2*: Is there an emerging tendency to address this challenge?*

The tendency seems to be defined by several criteria, for instance, the flexibility and ease of using the approaches, their suitability for being used by most instructors, and the effort, restrictions and skills involved in the use of these approaches. The map shows that approaches involving real projects or clients (that are most of the alternatives) are not particularly attractive for software engineering instructors. This is probably because these approaches are more complex to apply and also involve significant effort from instructors, teaching assistants, and sometimes also from clients and users.

We suppose that the traditional approaches (i.e., those that compliment theoretical lectures with practical assignments) are the most used, because it is not easy to change the instructional process used by the lecturers, and also because most computer science and engineering courses follow that pattern. However, there are only a few reports on the use of traditional approaches in the literature. This is likely due to such an approach being not as novel and it is rarely effective at delivering knowledge in this area.

RQ3*: Which software process models are used to support the practical experiences in software engineering courses?*

Viewing the map and analyzing the software processes used to guide these practical experiences, we see that more than half of the studies do not report the process used in these experiences (104 studies – 60%). This result is not surprising, because many software engineering practical experiences tend to simulate the work scenario of small or very small software enterprises, due that will probably be the first professional niche addressed by the new software engineers. According to a study performed by Quispe et al. [13], the software process models used by these enterprises are typically informal, immature and guided by the deadlines. Therefore, if the experiences in the academia simulate this part of the software industry (that is the most representative one), clearly these experiences do not have a process to be reported in the papers. In summary, it is highly probable that the process used in the academia to guide the practical experiences is informal and guided by the deadlines.

Concerning the reporting of articles, most of the processes used in these experiences indicate the use of an agile process. Agile processes are usually not too formal and guided by the deadlines, which support the previous hypothesis. Twenty-five studies reported the use of agile processes (14%), followed by ad-hoc process (stated in the study) with 18 studies (10%), traditional processes with 8 studies (5%), TSP/PSP with 7 studies (4%), RUP with 5 studies (3%), various other process with 3 studies (2%), CMM with 2 studies and MBase with 1 study. Supporting the hypothesis that the complexity and effort of conducting these experiences establishes the usage tendency. Few articles report on the use of RUP, TSP/PSP, MBase and CMM. The traditional processes represent an interim point because they are a bit bureaucratic (e.g. waterfall or incremental), but easy to use. Their usage does not involve significant effort by the instructor. Clearly, the preferred are the agile processes because they are easier to use.

RQ4*: Have the universities changed the way of conducting these experiences over the years?*

Figure 4 shows a timeline with the reported teaching approaches used in these practical experiences. Initially the traditional approach and learning by doing were used; however, during the last ten years several other alternatives have been proposed and used indicating that this issue is still open.

Nowadays, the learning by doing approach is still in use and the traditional one has lost ground. Maintenance was used at some point, but did not have continuity of use. This could be because today the development of new solutions is much more frequent than the extension of the existing ones. Other approaches such as games, open source, PBL/OBL have started to be reported on over the last ten years and continue being used as an attempt to find new ways of addressing the stated challenge.

RQ5*: What are the main forums to seek information on practical approaches for* teaching s*oftware* e*ngineering?*

The majority of the primary studies selected were published in conference proceedings (125 studies – 72%) and only 48 studies (28%) were journal publications. The main forums on the theme were the Conference on Software Engineering Education and Training - CSEE&T (40), Frontiers in Education – FIE (18), International Conference on Software Engineering – ICSE (7), ACM Technical Symposium on Computer Science Education (6) and Conference on Innovation and Technology in Computer Science – ITiCSE (5) and the rest of the articles were published in 36 different conferences.

Concerning the articles reported in journals, 12 studies (25%) were published in the ACM SIGCSE Bulletin, 11 studies (23%) in the Journal of Computing Sciences in College, 8 studies (17%) in IEEE Transactions on Education, 3 studies (6%) in the Journal of Systems and Software and 2 studies (4%) in the Software Engineering Journal. The rest of the studies (12 studies – 25%) were published in several journals, which are not particularly focused on software engineering or engineering education.

IV. THREATS TO VALIDITY

Construct validity reflects to what extent the phenomenon under study really represents what the researchers have in mind, and what is investigated according to the research questions. The number of papers found indicates that the terms used in the search process are well defined. Only the data available before October 30, 2013 was considered in this study.

Data reliability focuses on the papers collected and analyzed in order to see if these processes were conducted in a way that they can be repeated. The search terms were defined and the procedures applied in this study followed the previously described guidelines. The corpus of literature used as a primary study is reported in [12], therefore other researchers can replicate this study. The extracted information could also be a source of reliability concern, but the authors made the data extraction in an individual and parallel way. In case of differences in the authors' opinion, a clarification process was conducted until reaching consensus.

The results found could be subject to limitations of the automated search engines used in this work. The primary studies considered here were the ones that entered the inclusion criteria and were not rejected by the exclusion criteria. The classification of the studies according the predefined criteria was not always straightforward (some classifications were implied by the authors). The classification of the studies was done mainly by reading the title, abstract and in a few cases, the whole article.

Internal validity is more related to analysis of the collected data. A mapping study does not have any statistical assumption; therefore there are no threats. External validity is concerned with generalization. Since the systematic mapping studies in general do not state conclusions, the external validity threats are not applicable.

V. CONCLUSIONS

This paper presents a systematic mapping study that characterizes the approaches available to address the practical experiences in software engineering education. We used a corpus of literature of 173 primary studies that were published in the main conferences and journals of the area. The results show that the academia is conscious of the need to teach software engineering in a practical way, and it is also conscious of the effort that this challenge represents for instructors, teaching assistants and universities.

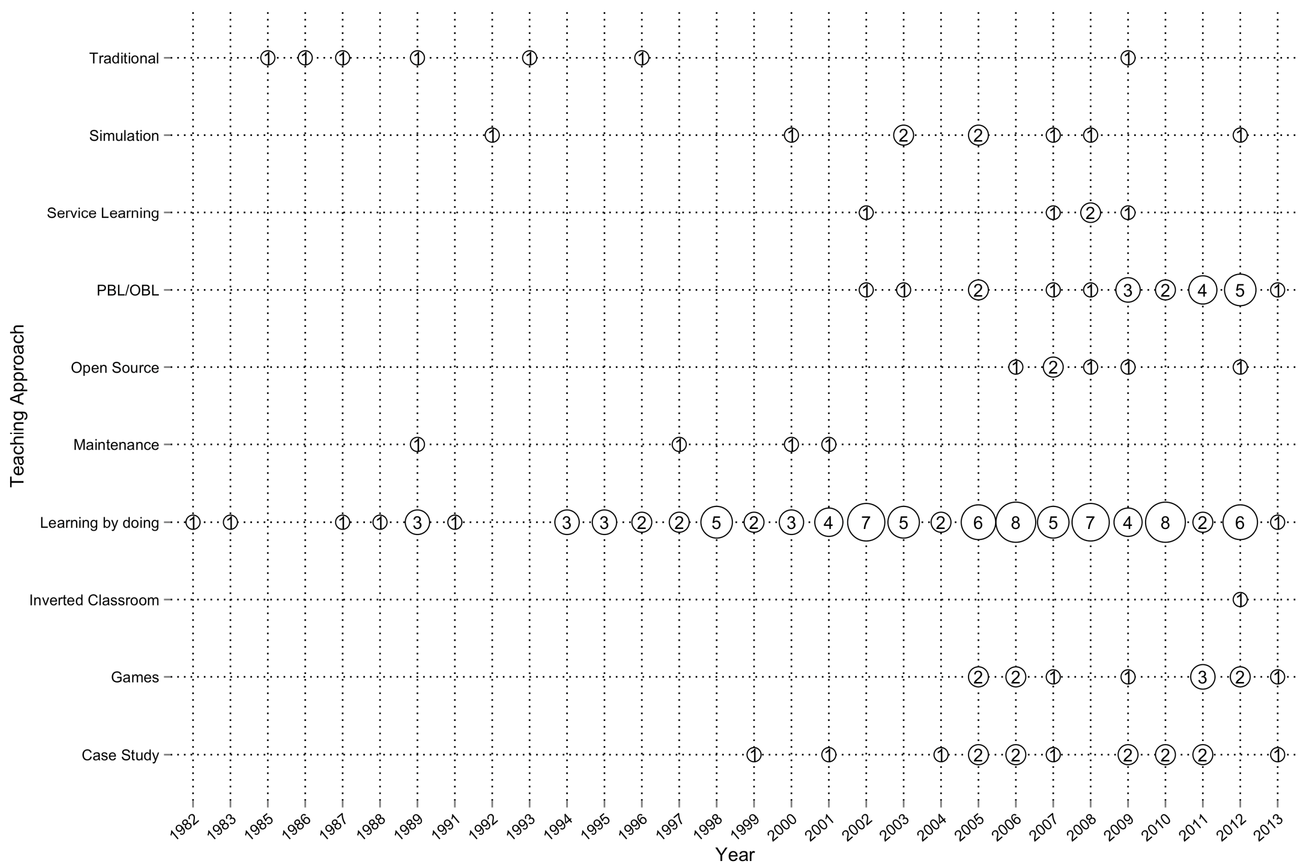


Figure 4 - Timeline of the teaching approaches

Although several approaches have been proposed to address this challenge, the preferred ones seem to be those that are easy to use, and represent minimum effort for the instructors. As there is no comparison of the effectiveness of these approaches, we therefore cannot say anything about it.

Concerning the software process models used to guide these experiences; the preferred ones seems also be those requiring less effort and complexity in their use. In that sense, agile methods were preferred considering the articles that reported this feature of the practical experiences. Most studies do not indicate the software process used in the experiences. If we draw a parallel between these experiences and the development scenarios of very small and small software enterprises, we can hypothesize that no process was used in the practical experiences in the academia. The process will probably be informal and guided by the deadlines assigned to the project; therefore there is no process to report.

This result is not surprising because teaching software engineering in a practical way is still an open issue, and represents an important challenge for any university. More research and experimentation is required in this area in order to find new proposals that address this challenge more effectively.

The map presented in this study shows some trends and issues in software engineering education. This gives us some directions for future research. We plan to perform a more in-depth analysis of the proposed teaching approaches and software processes in order to determine their effectiveness in the instructional process.

Acknowledgment

The work of Maíra Marques Samary was supported by the PhD Scholarship Program of Conicyt Chile (CONICYT-PCHA/Doctorado Nacional/2012-21120544). This work was also partially supported by the Project Fondef Idea, grant: IT13I20010.

References

[1] IEEE Standards Collection: Software Engineering. IEEE Standard 610.12-1990, (1993).

[2] Chen, J., H. Lu, L. An, and Y. Zhou. "Exploring Teaching Methods in Software Engineering Education". Proceedings of the 4th International Conference on Computer Science & Education (ICSE). IEEE Press (2009): 1733-1738.

[3] Begel, A., and B. Simon. "Novice Software Developers, All Over Again". Proceedings of the 4th International Workshop on Computing Education Research. ACM Press (2008): 3-14.

[4] Moore, M., and C. Potts. "Learning by Doing: Goals and Experiences of two Software Engineering Project Courses". Lecture Notes in Software Engineering Education, Vol. 750. (1994): 151-164.

[5] Giraldo, F., C. Collazos, S.F. Ochoa, S. Zapata, and G. Clunie. "Teaching Software Engineering from a Collaborative Perspective: Some Latin-American Experiences". Proceedings of the IEEE Workshop on Database and Expert Systems Applications, IEEE Press (2010). 97-101.

[6] Petersen, F., R. Feldt, S. Mujtaba, and M. Mattsson, "Systematic Mapping Studies in Software Engineering". Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering, (2008), pp. 71-80.

[7] Kitchenham, B.A. "Guidelines for performing systematic literature reviews in software engineering version 2.3". Technical Report S.o.C.S.a.M. Software Engineering Group, Keele University and Department of Computer Science, University of Durham, (2007).

[8] Nascimento, D.M., K. Cox, R.A. Bittencourt, R. Souza, and C. Chavez, "Using Open Source Projects in Software Engineering Education: A Systematic Mapping Study". Proceeding of the Frontiers in Education Conference, (2013).

[9] Scacchi, W. Process Models in Software Engineering. 2nd Edition. New York: John Wiley and Sons, Inc., (2001).

[10] Cockburn, A. A Human-Powered Methodology for Small Teams. Boston: Addison Wesley, (2004).

[11] Humphrey, W. "Using a Defined and Measured Personal Software Process". IEEE Software, (1996), pp. 77-88.

[12] Marques, M., A. Quispe, and S.F. Ochoa. "Corpus of Literature for a Systematic Mapping Study on Practical Approaches for Teaching Software Engineering". Technical Report No. DCC-20140425-002. Computer Science Department, University of Chile. (2014). Available at: http://www.dcc.uchile.cl/TR/2014/TR_DCC-20140425-002.pdf

[13] Quispe, A., M. Marques, L. Silvestre, S.F. Ochoa, and R. Robbes. "Requirements Engineering Practices in Very Small Software Enterprises: A Diagnostic Study". Proceedings of the XXIX International Conference of the Chilean Computer Science Society, pp. 81-87. IEEE Press. Antofagasta, Chile, Nov. 15-19, 2010.