MAP-I
Programa Doutoral em Informática

Research Topics in Software Engineering

Unidade Curricular em Paradigmas da Computação
Paradigms of Computation
(UCPC)

UMinho, FEUP

July 23, 2009

Abstract

This document describes a Ph.D. level course, corresponding to a Curriculum Unit credited with 5 ECTS. It corresponds to a joint UMinho-FEUP proposal for UCPC (Paradigms of Computation) in the joint MAP-i doctoral program in Informatics, organized by three portuguese universities (Minho, Aveiro, and Porto).

LECTURING TEAM

UMinho:	João M. Fernandes, Ricardo J. Machado
FEUP:	João Pascoal Faria, Ademar Aguiar
Coordinator:	João M. Fernandes

A. Programmatic Component

1. Theme, Justification and Context

Motivation: Software Engineering

Software engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software, and the study of these approaches; that is, the application of engineering to software. It applies both computer science and engineering principles and practices to the creation, operation, and maintenance of software systems. A knowledge of programming is the main prerequisite to becoming a software engineer, but it is not sufficient. In fact, software engineering, as a scientific field, encompasses many subdisciplines:

1. Software requirements: The elicitation, analysis, specification, and validation of requirements for software.
2. Software design: The process of problem-solving and planning for a software solution. After the purpose and specifications of software are determined, software developers plan for a solution. It includes low-level component and algorithm implementation issues as well as the architectural view.
3. Software development: The construction of software through the use of programming languages.
4. Software testing: The empirical investigations conducted to provide stakeholders with information about the quality of the product or service under test, with respect to the context in which it is intended to operate.
5. Software maintenance: Software systems often have problems and need enhancements for a long time after they are first completed. This subfield deals with those problems.
6. Software configuration management: Since software systems are very complex, their configuration (such as versioning and source control) have to be managed in a standardized and structured method.
7. Software engineering management: The management of software systems borrows heavily from project management, but there are nuances encountered in software not seen in other management disciplines.
8. Software development process: The process of building software is debated among practitioners with the main paradigms being agile or waterfall.

9. Software engineering tools: The scientific application of a set of tools and methods to a software which is meant to result in high-quality, defect-free, and maintainable software products.

10. Software quality: The approaches used to measure how well software is designed (quality of design), and how well the software conforms to that design (quality of conformance)

Software engineering is also related to the disciplines of computer science, project management, and systems engineering. This diversity of skills and competences makes it mandatory to have a broad approach when teaching software engineering. This course addresses some of the challenges faced by software engineers and the software engineering field that are posed by the increasing complexity of software intensive systems (and of their development) on which our society is increasingly dependent. More specifically, in this course, we plan to tackle several subdisciplines of software engineering and address the research challenges that nowadays emerge in those subdisciplines. Additionally, we plan to present in this course the state-of-the-art of several subdisciplines of software engineering and to identify challenges that might help doctoral students on selecting a topic for developing research in Software Engineering.

Course Context

ACM Computing Classification System subjects covered:

- D. Software / D.2 Software Engineering / D.2.1 Requirements/Specifications
- D. Software / D.2 Software Engineering / D.2.2 Design Tools and Techniques
- D. Software / D.2 Software Engineering / D.2.11 Software Architectures
- D. Software Engineering / Reusable Software / D.2.13 Domain engineering
- D. Software Engineering / Reusable Software / D.2.13 Reuse models
- K. Computing Milieux / K.6 Software Management / K.6.3 Software development

2. Objectives and Learning Outcomes

This course aims to introduce the fundamental concepts underlying the fields of architecture, design, construction and integration of large-software systems. More specifically it intends to cover, both from the foundational and the methodological point of view, the construction, analysis, design, classification, animation, validation and verification of software systems at different levels of abstraction and concern. As a second objective the course aims at providing the conceptual tools for the use of models in all phases of the software process, with a particular emphasis on requirements and design. The course is not intended as an introductory survey to Software Engineering, but as an opportunity of exposing students to cutting-edge research topics in this area, although presented in a coherent and integrated way. It is placed at a similar level and cover overlapping material with advanced modules in doctoral programs at leading academic institutions.

Upon successful completion of this curricular unit, students should be able:

- to define what type of procedures the requirements engineering team is supposed to execute at the development process, by identifying the formal involvement of the stakeholders;
- to define the way requirements are to be elicited and the techniques to use to correctly gather requirements from all the sources;
- to promote the assessment of software process and to monitor, in collaboration with software engineers, the software process improvement efforts;
- to identify the positive and negative aspects of the software process, through the acquisition, analysis, and interpretation of quantitative data;
- to explain the need for describing software systems with models, as a way to abstract from the system's complexity and to reason about its properties;
- to use models for the activities (analysis, design, implementation, testing, maintenance) associated with the development of large software systems;
- to idealize different alternative architectures to solve the same problem and evaluate (justifying) which is the best one in terms of design quality;
- to recognize and understand several architectural and design patterns.

3. Course Contents

1. **REQUIREMENTS ENGINEERING & MANAGEMENT**
   This unit focuses on the software requirements knowledge area as a critical domain of software engineering, as outlined in the IEEE Computer Society's Software Engineering Body of Knowledge (SWEBOK). The area of software requirements deals with the acquisition, analysis, specification, validation, and maintenance of software requirements. Requirements are the properties that a given system (still in project) will exhibit when its development is finished. This area is

recognized as being extremely important for industry, since its activities have a great impact on the development process. This unit focuses on the following topics:

- Definition of requirement;
- Distinction among different types of requirements (user requirements vs. system requirements; functional requirements vs. non-functional requirements);
- Requirements process and its associated activities;
- Elicitation techniques;
- Requirements prioritization and negotiation.

2. **SOFTWARE PROCESS ENGINEERING**
The software engineering process can be considered at two distinct levels: (1) the activities related to the acquisition, development, and maintenance of software; (2) the activities related to the definition, implementation, measurement, and improvement the software process itself. In this context, this unit focuses on the techniques and methods devoted to: (1) the definition of software processes at its relation to the software artifacts' lifecycles; (2) the configuration of best practices of software referential design processes to support the development of large-scale software solutions. The unit is structured into the following topics:

   - Software process fundamentals (software lifecycles and notatations for process definition).
   - Software processes for large enterprizes (RUP, EUP, EABOK, TOGAF).
   - Software improvement and maturity models (ISO standards, SEI reference models).

3. **MODEL-DRIVEN APPROACHES**
This unit's purpose is to study several modeling frameworks in Software Engineering, with a particular focus in approaches based on the Unified Model Language (UML). As the OMG specification states, UML is "a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system". Although in itself it does not specify any methodological or design process, its role as a (collection of inter-related) semi-formal notations in supporting software development, from business processes or global architectures down to database schema, and reusable software components, became more and more fundamental, almost a de facto standard, to the whole Software Engineering discipline. This course unit also covers the fundamental issue of model transformation within the two basic types of models considered: visual (like UML) and formal (like VDM, Petri nets or ASM). Particular emphasis will be placed on the following topics:

- Making UML models precise and executable, either through the use of UML related languages (OCL and UML action semantics), or through the integration with formal specification languages (like VDM++, Spec#, ASM, Petri nets or Alloy).
- Definition of domain specific languages and UML profiles, using the UML extensibility mechanisms and the UML meta-model.
- Model analysis (with model-checking), simulation and testing.
- Model refinement and transformation (from analysis into design models, making explicit the envisaged software architecture, and from platform independent into platform specific models).
- Code generation from design models, especially for behavior-intensive systems, and its current limitations.
- Model-based testing (i.e., automatic generation of conformance test cases from models), especially for interactive systems, and its current limitations.
- Adaptation of high-maturity processes (like the Personal Software Process and Team Software Process) for model-driven software engineering.

4. **SOFTWARE DESIGN AND PATTERNS**
The architecture of a software system describes the global structure in terms of its components, external properties and its interrelations. As software systems grow in scale and complexity, it becomes increasingly more important to understand them at many abstraction levels other than algorithms, functions, objects or components, and by different kinds of people, such as procurers, acquirers, producers, integrators, trainers, and users. Architectural models for such large-scale systems must be tailored to allow the dynamic construction and allocation of customized applications to heterogeneous computing devices, with different computational or interface capabilities. In this unit, many design and architectural challenges for highly complex and large-scale software systems are addressed. Many of these are software engineering challenges that must take into consideration aspects not only related with individual computing devices, but also with the entire system obtained from the cooperation of diverse, dispersed, integrated or mobile computing devices that in conjunction contribute to the achievement of the overall system objectives. In particular, the following topics will be considered:

   - Software design: fundamental concepts and principles.
   - Software architecture: definitions, concepts, components, connectors, views, quality attributes.
   - architectural styles, reference models and reference architectures.
   - Architectural styles: pipes and filters, data abstraction, object-orientation, event-based systems, layered systems, repositories, interpreters, process-control systems.

- Software patterns: origins, notion of patterns and pattern languages, kinds of patterns (architectural, design, others).
- Thematic catalogs of patterns: patterns of enterprise application architecture, patterns for enterprise integration.

4. Teaching Methods and Student Assessment

The best way to understand and master software design and software architecture is to experience it. In the educational setting, this means:

- learning the fundamental concepts and principles;
- knowing and understanding the solutions and practices proven to be the best, through the exploration of specific examples from the past, so-called case studies; and
- applying the knowledge acquired by imitating and adapting known solutions to a specific problem through hands-on development of a software system, in an individual project.

No textbook adequately covers the course's range of topics, so a diversity of bibliographic elements (books, journals and conference proceedings) will be used.

Readings

All reading assignments come from journals and conference proceedings. Each week, the students must read papers or some few supplemental readings provided. This exposes many students to extensive readings from the research literature for the first time. To help them with their reading, we require them to write a brief summary for each paper, submitted via email at the beginning of the week. We also ask them to submit a list of questions about the readings, which we try to work into the lecture if possible. During the last few weeks of the course, we no longer require reading summaries, to give students more time to focus on the project.

Classes

The class meetings are meant to be conversational, and we encourage students to ask questions and make comments. Consequently, the discussion may follow tangents to the prepared lecture, but they should be fruitful, informative, and thought provoking.

Individual research project

For the individual research project, we base the grade on an oral presentation (for a more methodological project) or a demonstration (for a more technological project), and a final written report. A few weeks into the course, we hand out descriptions of possible projects. The students have a week to look over the project descriptions before chosing one of them. No two students can work on the same project. We make the project descriptions intentionally vague, since it gives them considerable leeway in making design decisions. Having too specific descriptions would force students down a design path that they might not choose on their own. Once students complete their project, they must demonstrate it, make an oral presentation, and submit a final written report. The report has two major pieces: first, the discussion of the project?s major design decisions and trade-offs; second, one section entitled "If I could do it all over again. . ." describing what they would do differently if they could have a second chance to start from the beginning.

5. Basic Bibliographic References

- Ambler SW, Nalbone J, Vizdos MJ. *The Enterprise Unified Process: Extending the Rational Unified Process*, Addison-Wesley, 2008.
- Buschmann F, Meunier R, Rohnert H, Sommerlad P, Stal P. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*, John Wiley & Sons, 1996.
- Gamma E, Helm R, Johnson R, Vlissides J. *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
- Gomaa H. *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*, Addison-Wesley, 2005.
- Kleppe A, Warmer J, Bast W. *MDA Explained: The Model Driven Architecture – Practice and Promise*, Addison-Wesley, 2003.
- Mellor SJ, Balcer MJ. *Executable UML: A Foundation for Model-Driven Architecture*, Addison-Wesley, 2002.
- Robertson S, Robertson J. *Mastering the Requirements Process*, Addison Wesley, 2nd edition, 2006.
- Shaw M, Garlan D. *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996.
- Utting M, Legeard M. *Practical Model-Based Testing: A Tools Approach*, Morgan Kaufmann, 2007.