Computers in Human Behavior 48 (2015) 245–254

Contents lists available at ScienceDirect

Computers in Human Behavior

journal homepage: www.elsevier.com/locate/comphumbeh

Putting engineering into software engineering: Upholding software engineering principles in the classroom

Fairouz Tchier [a], Latifa Ben Arfa Rabai [b,*], Ali Mili [c]

[a] King Saud University, Riyadh, Saudi Arabia
[b] ISG, Bardo 2000, Tunisia
[c] NJIT, University Heights, Newark, NJ 07102-1982, United States

ARTICLE INFO

Article history:

Keywords:
Software engineering
Software engineering education
Software engineering principles
Classroom
Teaching practice

ABSTRACT

Ever since it emerged in the late (nineteen) sixties, the discipline of software engineering has set itself apart from other engineering disciplines in a number of ways, including: the pervasiveness of its products; the complexity of its products and processes; the criticality of its applications; the difficulty of managing its processes and estimating its costs; the volatility of its workforce; the intractability of its process lifecycles; etc. A number of principles have emerged from recent software engineering research, that have the potential to bring a measure of control to the practice of this discipline; but they have not made it into routine practice in industry. We argue that the classroom is a good place to start acquainting students with these principles, and to start getting them into the habit of adhering to them as a matter of routine practice.

1. Software engineering education

1.1. A profession in high demand

In its 2012 survey of best jobs in America, the Wall Street Journal (http://www.wsj.com/) lists software engineers at number 1, using data from the (US) Bureau of Labor Statistics compiled by the job search site CareerCast (http://www.careercast.com/). This assessment is echoed by CNN Money, which lists Software Architects and software developers among the top ten jobs in America, citing their growth opportunities and outstanding prospects. Many of the criteria used in this classification have global significance, i.e. are not specific to the United States; hence their conclusions apply equally well on a global scale. Software engineering jobs are not only attractive, they are also in abundant supply: The TechServeAlliance (http://www.techservealliance.org/), a collaboration of Information Technologies (IT) services firms, clients, consultants and suppliers, finds, on the basis of statistics published by the (US) Bureau of Labor Statistics, that in November 2012 the IT sector provided 4,185,800 jobs, which represents a 2.66% increase over a year earlier (November 2011), and a 24.24% increase over a decade earlier (November 2002). Also, the job search site Indeed (http://www.indeed.com/) collects statistics on the number of job openings by sector, and finds that in October 2012, the IT sector has 284 397 job openings, which ranks it as 3rd out of 13 sectors, after healthcare and retail. The combination of a plentiful job market and attractive working conditions for software engineers creates a demand for academic degrees in the field: In its May 2012 online edition, the Wall Street Journal discusses the shortage of IT engineers, and the efforts made by companies to find qualified talent. Echoing this finding, the July 2012 online edition of US News and World Report (which is responsible for the yearly ranking of academic institutions in the US) writes about the explosive growth of computer science majors on US campuses, and the strain this places on CS departments struggling to meet the demand; in particular, this articles cites a 25% increase of enrolment in CS at Stanford with respect to the 2000–2001 enrolment figure, at the height of the dot-com boom.

With increased demand for computer science and software engineering professionals in the job market, and increased popularity of computer science and software engineering majors on campus, comes increased responsibility for computer science and software engineering educators to review/reassess their options and to refine their decisions. As educators, we are under conflicting pressures to fulfil two criteria that may appear to be irreconcilable: the short term goal of making students operational on their first day on the job; and the long term goal of enabling them to adapt to future technological evolution.

* Corresponding author.
E-mail addresses: ftchier@ksu.edu.sa (F. Tchier), Latifa.rabai@isg.rnu.tn (L.B.A. Rabai), mili@oak.njit.edu (A. Mili).

http://dx.doi.org/10.1016/j.chb.2015.01.054
0747-5632/

1.2. A unique engineering discipline

In addition to being the youngest field of engineering, software engineering differs from other engineering disciplines in many ways, which we briefly discuss below:

- *The pervasiveness of its products*: Software pervades all sectors of modern economies, and all aspects of modern lifestyles. With the emergence of social media, computers now pervade the most personal aspects of people's lives, especially for millennials; this trend is expected to accelerate, with the advent of wearable computing.
- *The complexity of its products and processes*: A recent study by the Software Engineering Institute (Northrop et al., 2006) predicts that future software systems will have a size in excess of a billion lines of code, and will be characterized by massive heterogeneity along several dimensions.
- *The criticality of its applications*: Software is used in many safety–critical and mission-critical applications, making it imperative that we control its quality attributes; because the software development process is very labor-intensive and involves extensive human intervention, it is very difficult to ensure its quality through process controls, hence most quality assurance must depend on product controls instead.
- *The difficulty of managing its processes and estimating its costs*: While the bulk of the cost of an engineering product is typically due to manufacturing (rather than design) costs, the cost of a software product is almost totally accounted for by design rather than manufacturing. Also, whereas testing is a minor lifecycle cost in all engineering disciplines, it is a major cost component in large and complex software projects, often reaching or exceeding 50% of lifecycle costs. Also, whereas the cost of an engineering project can typically be estimated on the basis of project parameters, the cost of a software engineering project is often dependent on estimates of projected line of code count, projected function points, or projected object points, all of which can only be obtained through expert opinion, reasoning by analogy, and other informal analysis (Boehm et al., 1995).
- *The volatility of its workforce*: Empirical studies show consistently that programmers distinguish themselves from other workers by specific motivation patterns, distinct goal structures, and distinct productivity metrics.
- *The intractability of its process lifecycles*: Unlike the project lifecycles of other engineering disciplines, software process lifecycles do not lend themselves to simple modeling, are highly iterative (prone to backtrack), and suffer from poor visibility (in the sense that it is hard to tell, at any point during the project, what portion of a project has been completed and what portion remains).

Software engineering research of the last decades has enabled us to gain some insights into the nature of this engineering discipline, and to derive sound principles of how to manage software projects. In this paper, we discuss how we propose to integrate some of these principles into the computer science/software engineering curriculum, at the undergraduate and graduate levels; also, we discuss why we feel that the classroom is an adequate environment for integrating these principles into the routine practices of software engineering students.

1.3. Principled software engineering education

Traditional engineering disciplines (chemical engineering, civil engineering, electrical engineering, mechanical engineering, nuclear engineering, etc.) are based on theoretical laws that have matured through decades, sometimes centuries, of research, and have made their way into routine practice and into routine educational materials. The youth of the software engineering discipline, the complexity of software processes and artefacts, and the massive market pressures placed on the software industry have undermined this natural flow, and have caused the proliferation of improvised ad-hoc solutions whose foundations are not carefully analyzed and understood.

We argue that a long term solution to this situation involves the integration of software engineering principles into the educational curriculum. Whereas software engineering is usually divided into three broad areas, dealing with technical aspects, economic aspects, and organizational aspects, we focus in this paper on technical aspects, and we select four principles therein. These principles deal with the four main phases of the software lifecycle, namely: requirements specification, software design, programming, and testing. We present them below:

- *Formal Specifications*: We argue that formal specifications are an indispensable basis for any sound analysis of the functional properties of software artefacts. The art of identifying relevant stakeholders, eliciting relevant requirements, representing these requirements and combining them into a comprehensive specification are an integral part of a software engineering discipline.
- *Modular Design*: Modular programming (Parnas, 1972) is a discipline of bottom-up program design that is based on the principle of information hiding, whereby each component/module in a software product exports its specification but hides its design and implementation. This approach offers significant advantages to software developers, in terms of enhanced programmer productivity, and enhanced program maintainability, testability, reliability, and reusability. We argue that it is an integral part of a software engineering discipline.
- *Verification-based Programming*: Computers are precise machines that hold us rigorously to every word/every symbol we write in a program; we cannot work with such machines unless we can match their precision and rigor with equally precise methods of program construction. In the same way that a civil engineer designs a bridge by solving equations pertaining to load requirements and terrain characteristics, and in the same way that a mechanical engineer designs an engine by solving equations pertaining to power requirements and vehicle characteristics, so a software engineer ought to design programs by calculation from software requirements and machine characteristics, rather than by obscure trial and error approaches.
- *Goal Oriented Testing*: We argue that software testing should be conducted in a systematic manner, by following a rigorous process that includes the following steps (Mili & Tchier, 2015): defining a precise goal for the test (unit testing, integration testing, acceptance testing, certification testing, reliability testing, regression testing, etc.); defining precise hypotheses under which the test is conducted (what are we assuming to be correct, what are we checking); defining an oracle that determines whether any given test was successful; defining a criterion for test data selection; defining a criterion for when we can consider that the goal of the testing activity has been achieved; defining how the test results are to be analyzed; defining the claim that we can make about the software artefact once the test has completed and its results have been analyzed.

These four principles are not routine in current industrial practice, as confirmed repeatedly by our students, many of whom hold positions in industry. In this paper, we discuss the design and delivery of two courses where we make a special effort to adhere to these principles, and report on our experience. We feel that