A software quality course: the breadth approach

Petre, Luigia

Published in:
Quality of Information and Communications Technology

DOI:
10.1007/978-3-030-85347-1_38

Published: 01/01/2021

Document Version
Accepted author manuscript

Document License
Publisher rights policy

Link to publication

Please cite the original version:
Petre, L. (2021). A software quality course: the breadth approach. In A. C. R. Paiva, A. R. Cavalli, P. Ventura Martins, & R. Pérez-Castillo (Eds.), *Quality of Information and Communications Technology* (pp. 536-552). (Communications in Computer and Information Science; Vol. 1439). Springer. https://doi.org/10.1007/978-3-030-85347-1_38

A Software Quality Course: the Breadth Approach

Luigia Petre$^{1\,[0000-0002-0648-3301]}$

Åbo Akademi University, lpetre@abo.fi

Abstract. We present a Software Quality course taught in a MSc program in Computer Science and Engineering. The course takes an overview ('breadth') approach, reviewing the most important topics that contribute to the quality of software. The course has been taught traditionally as well as online; we discuss the advantages and disadvantages of both styles and point out what should be kept from the online experience. We also discuss the students' evaluation and feedback.

Keywords: Software Quality · Requirements · Formal Methods · Software Architectures · Software Metrics · Online teaching · Learning journals · Polls.

1 Introduction

The hiring policies in software industry differ substantially from other domains. A person who has not graduated the heavy medical education programs will never be hired as a doctor in a respectable hospital, nor would anyone hire lawyers who have not passed their bar exam. A software company does not follow the same protocol. The candidates to hire are certainly scrutinised by their prospective future companies, maybe subjected to coding interviews and/or other methods, but a diploma is not necessarily a requirement [8] for working as a software engineer. Persons deemed intelligent and malleable enough can be hired by a company, who then trains them in developing specific skills in demand. This practice has an immediate consequence: many students drop from their university programs once they find suitable employment, with their education only partially completed. Given the ubiquity of software in our society, this situation is rather alarming: are these software engineers able to ensure the quality of the software running our lives?

Even more worrisome, the existing university curricula do not offer much in terms of software quality education. While students are offered a rather wide selection of programming languages and paradigms to choose from, the quality of their code is not much emphasised. Some curricula can have courses on algorithmic complexity, requirements, or software architecture, but they are not as common as the programming courses. Formal methods courses are often left out completely, as are the software metrics courses. In this context, even the students who do graduate their respective programs, are not well prepared to analyse how 'good' certain software is or how to build software of good quality.

In this paper we present a course entitled ‘Software Quality’, taught in a MSc program in Computer Science and Engineering. After taking this course, the students should be able to:

- Identify and experiment with four pillars of quality software: **requirements**, **analysis (formal methods)**, **software architecture**, and **software metrics**
- Devise a **requirements document**
- Perform **basic software analysis** with formal methods
- **Distinguish several software architectures** with their advantages and disadvantages for quality
- Recognize and evaluate **different software metrics**
- Enumerate and apply different **quality management techniques**

While this is surely not the first time a course on Software Quality was taught, the novelty of our approach consists in recognising the complex nature of software quality and, hence, investigating a wide array of topics that contribute to it. For instance, formal methods are not typically considered an essential topic for software quality, unless formal methods researchers are involved. Still, formal methods are the only way to capture software ‘blueprints’ and to provide qualitative assessment of various software properties. It is also important to recognise that formal methods do not (cannot) tell the whole story of software quality, one reason being scalability.

We argue that, for students completing their software engineering education, a course like this one is beneficial, as it offers the big picture of what good software is about. Depending on the topic they will specialise and be employed in, they can later focus on certain aspects, be that requirements, formal analysis, software architecture and/or software metrics. Moreover, for students not completing their software engineering degrees, working with a skilled software quality manager may be instrumental. These no-degree employees are the stereotypical intelligent programmers, more than able to assimilate and eventually master needed topics. Knowledgeable software quality managers can pinpoint exactly the techniques needed in their particular jobs, be that requirements elicitation, feature verification, unit testing, documentation readability, etc.

In addition to the content, we also explain the teaching methods we used in the course. They essentially boil down to the Roman ‘repetitio mater studiorum’ proverb, meaning that repetition is the mother of learning. We use various techniques for this, such as weekly learning journals and regular polls and quizzes to enforce repetition. The feedback from students has been positive; apparently they are engaged by these methods rather that overwhelmed.

The online teaching aspect is also discussed. We compare the results of teaching this course traditionally vs online and point out what we believe should be kept from the online experience.

We proceed as follows. In Section 2, we present the main tenets of the course and we overview its structure. Section 3 reviews the teaching of the four pillars: requirements, formal methods, software architectures and software metrics, respectively. Section 4 explains the quality management topic. In Section 5 we explore the evaluation of students throughout the course and in Section 6 their feedback. Conclusions are presented in Section 7.

2 Basic concepts and the particulars of the course

As with many concepts related to software, terminology can be vague and mean many things. The notion of Software Quality is no exception. Expectations of a 'good' software may include code 'cleverly' built - whatever that might mean, fast software, or a good user interface, to name a few. This is the starting point for the course, exploring the terms of software, quality, and software quality for a whole 1.5-hour lecture.

Software We begin with the idea that software is eating the world [2]. Software is a wonderful human development, which has increased our quality of life tremendously. It has revolutionised entire industries, with businesses replaced almost completely by software, such as in music and photography, to businesses where their products' value comes increasingly more from software, such as in the car manufacturing. It has created an immense work space and continues to be employed in aiding or solving a huge array of problems, leading to its increased complexity: it is harder to pinpoint exactly all it does. This leads to the concept of software crisis, where methods for developing small software systems kept being applied for their large counterparts, leading to famous software failures. For avoiding the latter, we need to better understand the definition and nature of software, which we do by investigating a classical reference [5].

Quality To grasp what is meant by quality, in general, we delve into a bit of history, recognising notable personalities with an influence on quality, such as Frederick Winslow Taylor (1910s), Henry Ford (1863-1947), Walter Shewhart (1920s), and W. Edwards Deming (1950s). Often, we only present the technical aspect of our topics (to save time), but explanations about the people who led to particular developments (and why) are motivating the students. We distinguish between quality assurance and quality control, explaining that quality assurance allows a shift to the 'left': we take care of this earlier in the product lifecycle.

Software Quality With an adequate understanding of the concepts of software and quality, we move on to explore what is usually understood by the concept of software quality. Based on a classical source [5], we first underline the four main software characteristics contributing to the challenge of ensuring its quality: complexity, conformity to other interfaces, changeability, and invisibility. We then emphasize four partial solutions to software quality, namely the 'buy vs build' solution, the refinement of requirements and rapid prototyping, the incremental development, and the contribution of great designers. We analyse these in some detail and emphasize how they keep being valid more than three decades after they were suggested.

We then move on to more concrete terms and topics we will explore in some detail during the course. We first distinguish between functional (what the system should do) and structural quality (how the system should do it). The concepts of fault, error and failure are also defined and placed in the context of quality assurance vs control. We point to some bodies whose mission is, in a form or another, software quality and explain their main role in quality management, hence their treatment in that module of the course. We end this part by explaining the four pillars of the course, namely Requirements Engineering, Formal Methods, Software Architectures, and Software Metrics. We also show how they correspond to the classical principles proposed by Turing award recipient Fred Brooks [5].

It always seems to be a surprise to the students to learn that a software of good quality is the one implementing its requirements (functional and structural), nothing more and nothing less.

2.1 Structure and Teaching Methods

This course has been taught several times, always in an 8-week period, both typically, in a classroom, as well as online. The components of the course are: (1) Lectures (with polls), (2) Learning Journals (mandatory), (3) Quizzes, (4) Discussion Seminars, and (5) Exam (mandatory). The components are distributed as follows during the 8 weeks:

- Lect. 1 (week 1) → The concepts of software, quality, and software quality
- Lect. 2 (week 1) → Requirements - types and elicitation

- Lect. 3 (week 2) → Requirements - specification, validation, evolution
- Discussion seminar 1 (week 2) → we exercise requirements

- Lect. 4 (week 3) → Models and TLA+: the case of Amazon Web Services
- Lect. 5 (week 3) → The case of the French metro and modelling in Event-B

- Lect. 6 (week 4) → Program verification, the case of Dafny
- Discussion seminar 2 (week 4) → we discuss examples of specifications in TLA+, Event-B and Dafny

- Lect. 7 (week 5) → Model checking, SAT and SMT solvers run our provers
- Lect. 8 (week 5) → Software Architecture is key

- Lect. 9 (week 6) → More software architecture fundamentals for quality and how could measurement and metrics for software help us
- Discussion seminar 3 (week 6) → we discuss logical properties, we ponder about some architecture questions and reflect on testing, bugs and correctness

- Lect. 10 (week 7) → Software metrics for internal product attributes (mostly size)
- Lect. 11 (week 7) → Software metrics for internal and external product attributes (structure, usability, maintainability, security)

- Lect. 12 (week 8) → Quality Management - planning, standards, ISO 9001, reviews and inspections

- Invited lecture (week 8) → Quality in Space Software

The optional components of the course are the lectures, the quizzes, and the discussion seminars. There are typically 12 regular lectures and one invited lecture at the end of the course, to exemplify how software quality is ensured in space software. There are three discussion seminars, where we typically explore the concepts of the previous lectures in more depth. During every lecture, the teacher poses several online and anonymous polls checking whether some explained concepts have been well understood. Each poll consists in 1-2 multi-choice questions relating to a concept or methodology just explained by the teacher. After the students reply online (they are given a minute usually), the teacher displays the proportion of answers on each option and discusses what is the correct answer. Since not all the students can (nor have they to) participate in the lectures, these polls are then posted as quizzes to be - optionally - taken at any time during the course. Each taken quiz is worth max one point (out of 100 for maximum grade of the course).

The only mandatory components of the course are the learning journals and the exam. Every week, the student has to fill in a learning journal entry, of half a page to one page (or longer if desired). In each entry, they should explain what are the main concepts they learned during that week, give some definitions and examples, and discuss the topics in any way they see fit. They are encouraged to also ask questions, to point out what topics were new to them or were seen before, as well as what they enjoyed and what they did not. The teacher then reads each entry and replies to questions, comments on the entry and awards maximum 2.5 points per lecture. In a week with two lectures, this means the learning journal can earn maximum 5 points, and in a week with one lecture and one discussion seminar - maximum 2.5 points. Hence, the learning journals can bring up to 30% of the grade. The remaining 70% is to be obtained from the exam, which reviews the concepts learned during the course.

As seen, the maximum grade can be obtained only from the mandatory components (learning journals and exam), but additional points can be obtained from quizzes (max 12 points in total).

3 Content of the course

The four main pillars of the software quality that we explore in the course are requirements, formal methods, software architecture, and software metrics. In the following we shortly overview our approach in teaching them.

3.1 Requirements

Getting the requirements wrong, even partially, is the single most devastating reason for software problems, be them only faults or errors, or full blown failures.

Although many universities (including ours) have at least one course dedicated to (functional) requirements, we dedicate two lectures to requirements in this course and emphasize their role in software quality. There were some students (up to 10%) who remarked that they knew all about this topic, but the vast majority appreciated the overview, especially in light of the 'surprising' definition of software quality - implementation of requirements. Below we discuss the topics we cover with respect to requirements.

Requirements definition and types Given the vast diversity of software, understanding what a requirement is, who formulates it and how is of high relevance. There are several ways of classifying requirements, the most important of which are descriptive vs prescriptive, user vs system, and functional vs non-functional. We discuss these in some details, with the help of examples. Understanding the levels of abstraction implicit to classes of stakeholders, the immutability of some requirements, or the necessary measurability of non-functional requirements are topics that contribute to the quality of software we develop.

Requirements Engineering Getting the requirements into a requirements document is referred to as requirement engineering. This is an iterative process, well described in standard software engineering books [12]. We investigate the following topics:

- **Requirements Elicitation** We start with the extreme approaches of business geniuses such as Steve Jobs' opinion that customers do not know what they want or Henry Ford's observation that, if queried, customers would have desired faster horses (not cars). Obtaining the requirements from customers is a complex process, involving often contradictory, incomplete or changing requirements. Several methods of elicitation are analysed, such as interviews, ethnography and stories.
- **Requirements Specification** There are several alternatives to formulating requirements, in natural language, in some structured or tabular forms, or as use cases. We review all these with examples, and then ponder on the sections of the requirements document and its users.
- **Requirements Feasibility and Validation** Not all systems can or should be built; in feasibility studies we focus on asking the right questions to determine if it is worth to develop a software. Once we determine it is, validating the requirements with the stakeholders is very useful.
- **Requirements Change** Software is infinitely malleable, as debated in the initial lecture, hence changing it and its requirements is to be accounted for. Traceability is one essential aspect that, if not handled properly, can threaten software quality. For that, we explore traceability methods especially.

3.2 Formal Methods

A formal method allows to analyse a model of the software or even the software itself, to verify it respects certain properties.

Why Formal Methods Formal Methods are not a mainstream teaching subject in today's universities. Their promoters are the researchers and, more recently, some industrial giants: Amazon [6, 9], Facebook [10], and Google [11].

Software is embedded almost in all aspects of the society, so it is necessary to be able to guarantee what it does before deploying it. This is particularly essential in critical software, such as air transportation, army, nuclear plants, medicine, etc. Mass software construction does not rely on formal methods, but on simulation and testing. While these techniques have their uses, we argue for them being insufficient: clear analysis of models is needed, and we discuss models to understand their role in formal methods.

When Formal Methods The software lifecyle with respect to quality is four-stepped: (1) we collect the relevant (user and system, functional and non-functional) requirements, (2) we draw the software architecture of the system, (3) we apply formal methods to the modules identified via software architecture, and (4) we evaluate our results with software metrics. The effective implementation of the software takes place during steps (2)-(3).

In this course we treat formal methods before software architectures, because they address the analysis of functional requirements; software architecture deals with analysing the non-functional requirements. In addition, formal methods require a special mindset that is likelier to be present in the first half of the course.

Three Formal Methods We assume we have a correct set of requirements: how do we transform them into software and how do we guarantee that all the requirements and nothing more is implemented? We explore this with three formal methods: TLA+, Event-B, and Dafny.

– **TLA+** TLA+ is reviewed with two case studies: its adoption at Amazon [9] and by following its inventor, Turing award laureate Leslie Lamport, teaching the modeling and verification of a famous logical problem: the water jug riddle. This was part of the quest in the Bruce Willis and Samuel Jackson movie Die Hard 3. At some point, the heroes need precisely 4 gallons of water and they have two jugs, a 3-gallon one and a 5-gallon one, that they can fill with water from a tap. Model checking is employed to verify properties, i.e., all states of the modeled system are explored for these properties. In particular, model checking is used to verify that there is no possible sequence of actions that can end up with the bigger jug having exactly 4 gallons of water; the TLA+ model checker returns a counterexample, meaning it found a particular sequence of actions that achieves that.
– **Event-B** Event-B is motivated by its active use at Siemens, to develop the software on driverless trains [4]. Event-B comes with a tool for proving: we first write the model and the tool verifies its correct syntax; nothing out of ordinary so far. But then, we also formulate properties that we would like our model (the future software) to respect: the tool verifies if these properties hold for the model. Moreover, we can start from a simple model capturing just the essence of the software, and then add details. The tool keeps checking if the more complex version still respects the properties and if it is indeed a correct development of the simpler model. This development strategy is called refinement.
– **Dafny** Dafny is two languages into one: an imperative language, with an executable core and a functional, specification language for annotations. Annotations describe what a program should do, and Dafny generates proofs that the annotations match the code; if it cannot, then it asks for proof hints, which might be given or might suggest a problem t0 resolve. Amazon Web Services (AWS) use Dafny and other formal tools [6] and claim that this is one of the main reasons why people move to AWS: they want formal guarantees that their data is stored correctly.

How Formal Methods prove An extra lecture delving into the proving techniques in formal methods is necessary at this point. We investigate mainly model checking and satisfiability (SAT and SMT) checkers. With this, the students are shown the breadth of the topic of formal methods as well as the richness of its aspects. It equips the students with knowledge that - given the scarcity of higher education in this field - can offer them a big advantage compared to their peers. The relatively brief excursion into formal methods (4 lectures) is not exhaustive by far, but with it, the students know the main coordinates as well as where to look for more.

3.3 Software Architectures

We now address non-functional requirements, after the treatment of functionality with formal methods. Non-functional requirements are a direct consequence of the unavoidable complexity of software, hence the overall structure of the software system becomes relevant. In software architecture, we refer to non-functional requirements as **software qualities** and to design decisions for achieving them as **tactics**.

- **Software Qualities** We focus on availability, modifiability, performance and security. Availability is concerned with failures, modifiability with changes, performance with timing, while security combines availability (providing services to legitimate users) with resisting unauthorised users.
- **Tactics** For each quality, there are tactics that favour it and tactics that do not. This is one of the cornerstones of understanding software architecture: there are many ways to implement a certain functionality, some better for certain goals. It is impossible to satisfy all possible qualities, hence defining the most important ones for a system allows a trade-off with the various stakeholders [3].
- **Architectural styles** A collection of tactics forms a particular software architecture. People have devised generic styles that promote or inhibit certain software qualities. We analyse several styles and focus on cloud architectures, since the concepts are familiar to students; they have an easier time to depict the main architectural issues of interest and then to generalise them to other styles [12].
- **Views and Documentation** Another aspect of complexity inherent to software is that we can decompose a system based on various criteria: functionality, concurrency, physical allocation, etc. These are structures of a system that can be viewed in different ways, for instance in certain templates or diagrams. This is important for building the system, but also for documenting it.

3.4 Software Metrics

In this module we describe how we can measure the quality of software [7].

- **Software measurement** We start by clarifying what we want to measure, for instance the likelihood of product failure or delay, of losing key personnel, of bankruptcy, etc. In general, the ideal is of getting a 'big picture' indicator of software 'goodness' during development or maintenance. We clarify that there are several entities we can measure, such as products, processes, or resources, and for each we can have several attributes we are interested in, both internal (say, size for code) or external (say, reliability for code). The problem in software measurement is that we want to control external attributes such as reliability of code (a product), productivity for personnel (a resource), or cost-effectiveness for design (a process). However, we can only measure internal attributes, for instance code, team or design size.
- **Software size** Size is an interesting internal attribute that is used widely to predict many other measures. We learn to express size in the number of lines of code, differentiating between more and less important lines; Halstead's length, vocabulary and volume; the number of function points. We learn to use size to normalize measures of other attributes, quantify the amount of reuse, and measure attributes related to software testing.
- **Software structure** Software structure manifests as structure of data (trail of data items as created or handled by the program) and structure of control (sequence in which instructions are executed in a program). This can indicate issues impossible to see with the number of lines of code, for instance, the iterative and looping nature of the program. Since there is a link between structure and quality of software, we investigate structure in some detail. Structure is usually detected by studying flowgraphs, so we quickly review them. We discuss structured programming, prime decomposition, cyclomatic complexity, and as an example, their applicability in testing. Flowgraphs illustrate the inner structure of software modules, but there are also attributes for measuring the overall structure of a software system and its inter-module relations. We study here modularity, morphology, tree impurity and internal reuse.
- **External attributes** We are interested in measuring the quality of software code, especially its external attributes that depend on users. What we can measure, however, are the internal attributes, that can well be predictors of external attributes and that are available for measurement early in the software lifecycle. In contrast, the external attributes depend on the user's interaction with the (ready) system, so much later in the software lifecycle. We discuss in some detail the measuring of defects, usability, maintainability and security.

4 Quality Management

Should software quality be the job of the software engineers, should some administrators take care of it, or both? We argue for the last option and explain what it means to have both a software engineering and a software quality team [12].

Quality processes and culture While small and large software projects have different quality assurance processes, we can apply a quality process to any software project. A specific quality management team (different from the software engineering team) needs to ensure the realisation of such a process. The expectation is that a 'good' process will lead to a 'good' software. The quality and the project managers can agree on a particular quality assurance process, with a certain number of milestones. The process can follow standards for quality that are specific to the organisation and/or to the specific product being developed. The quality manager should report to higher management, while the project manager should have the freedom to ratify what quality procedures are relevant to the current project.

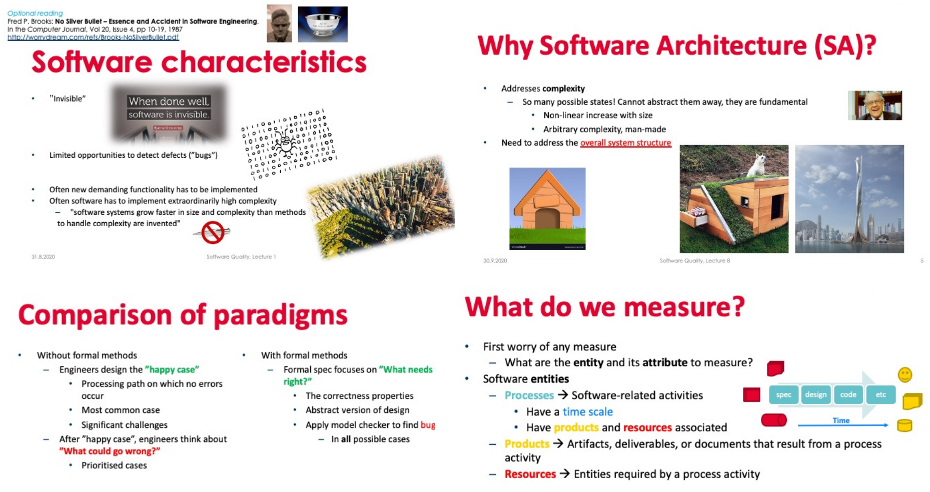
Standards Software standards are important for several reasons. They can be a reservoir of best practices, used to avoid past mistakes; they can clarify the organisation's view on quality or even the user expectations; they can provide for smooth adaptation and/or continuity for new employees, who understand the organisation through the prism of its standards. The quality and project managers can define both process and product standards. Both types should avoid being over-prescriptive and requiring excessive clerical work. A method of convincing software engineers on the worth of following standards is to include them in the process of devising them. There are numerous international and national standards as well, and certain companies require to work only with other companies who have followed particular software standards (e.g., ISO 9001). However, such standards typically confirm the presence of a particular process that has been followed in creating certain products; the quality of said products is a totally different issue.

Reviews and Inspections The most typical type of quality 'assurance' in companies is that of organising reviews and inspections. The reviews can 'sign off' a certain product, to move into the next phase and should be organised with several participants well in advance. Program inspections can be simpler, such as peer reviews where the most common mistakes are checked, e.g., 'have all variables been initialised?' or 'is each loop certain to terminate?' or 'has space been allocated correctly?', etc. The existence of a quality culture in the organisation is of utmost interest here, since blame pointing is really unhelpful and can lead to people hiding and refusing to share their code.

5 Course Evaluation

As can be seen from the described course content, the topics are modular and we delve into each of them only to some level of detail. Ideally, a university should have more courses to offer on each of the topics (requirements, formal methods, software architectures, software metrics, and quality management). In a software quality course we can only have an overview (breadth-approach) of these topics and study how they contribute to improving the software products.

This breadth-approach required a certain approach to teaching, based on repetition, to better cement the certainly new concepts. First off, the lectures were made as interesting as possible, with each slide well-thought of in terms of content and pictures. Some examples of slides are shown in Fig. 1. Then, to interrupt the monotony of the teacher talking, we introduced several polls per lecture. These polls are very easy to implement in Zoom, for instance, but can be certainly kept when returning to traditional teaching. One thing we can safely assume about our students is that they all have a mobile device with them, and so they can type some tiny-url addresses that can take them to the anonymous polls. DirectPoll [1] is one such resource. The results of their voting can be then shown on the big screen. This feature is entertaining both for teachers and students, makes the students more willing to focus on what is taught, knowing there will be a check-up very soon, and contributes to the repetition of concepts, as argued in Section 1. It is also very friendly: it is anonymous, the teacher explains the correct choice immediately after, and can also discuss the probable reasons some other options were chosen as well as why they were not correct. If some students really do not have a mobile device with them, then the polls are available as quizzes during the whole duration of the course; this also helps people who do not take part in the lectures. An example of a poll is shown in Fig. 2 and its quiz counterpart in Fig. 3.


When done well,
software is invisible.
spec
design
code
etc
Time

Optional reading
Fred P. Brooks: **No Silver Bullet – Essence and Accident in Software Engineering.**
In the *Computer Journal*, Vol 20, Issue 4, pp 10-19, 1987
http://worrydream.com/refs/Brooks-NoSilverBullet.pdf

Software characteristics

- "Invisible"
- Limited opportunities to detect defects ("bugs")
- Often new demanding functionality has to be implemented
- Often software has to implement extraordinarily high complexity
  - "software systems grow faster in size and complexity than methods to handle complexity are invented"

31.8.2020 Software Quality, Lecture 1

Why Software Architecture (SA)?

- Addresses **complexity**
  - So many possible states! Cannot abstract them away, they are fundamental
    - Non-linear increase with size
    - Arbitrary complexity, man-made
- Need to address the overall system structure

30.9.2020 Software Quality, Lecture 8 5

Comparison of paradigms

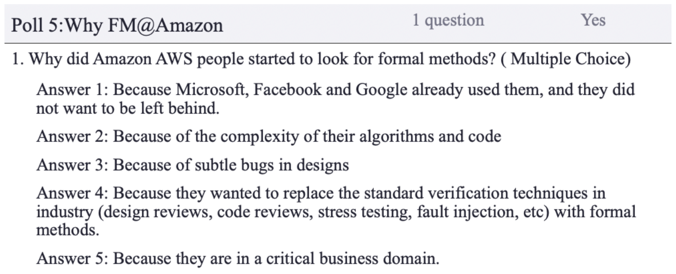
- Without formal methods
  - Engineers design the **"happy case"**
    - Processing path on which no errors occur
    - Most common case
    - Significant challenges
  - After "happy case", engineers think about **"What could go wrong?"**
    - Prioritised cases
- With formal methods
  - Formal spec focuses on **"What needs right?"**
    - The correctness properties
    - Abstract version of design
    - Apply model checker to find bug
      - In **all** possible cases

What do we measure?

- First worry of any measure
  - What are the **entity** and its **attribute** to measure?
- Software **entities**
  - Processes → Software-related activities
    - Have a time scale
    - Have **products** and **resources** associated
  - **Products** → Artifacts, deliverables, or documents that result from a process activity
  - **Resources** → Entities required by a process activity

Fig. 1. Some Software Quality slides

The second way of implementing repetition to favour learning is by using weekly learning journals. They are rather modern and many courses implement them; the difference from the approaches in many other courses is that, in the presented course, the teacher really reads all entries and provides feedback, even if short. While a student listens to a lecture, certain questions and comments arise, but they are maybe unexpressed due to a variety of reasons (shyness, not


Poll 5:Why FM@Amazon 1 question Yes

1. Why did Amazon AWS people started to look for formal methods? (Multiple Choice)

Answer 1: Because Microsoft, Facebook and Google already used them, and they did not want to be left behind.

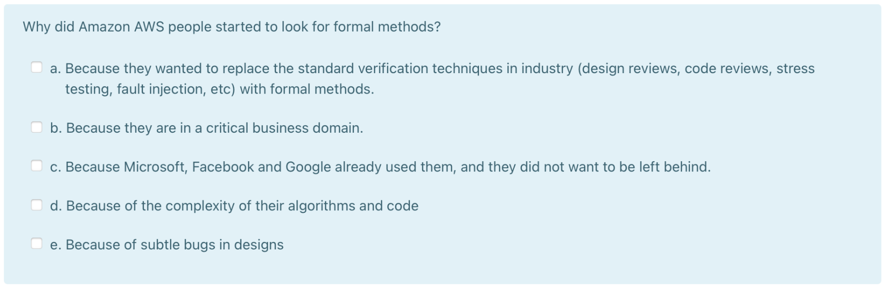
Answer 2: Because of the complexity of their algorithms and code

Answer 3: Because of subtle bugs in designs

Answer 4: Because they wanted to replace the standard verification techniques in industry (design reviews, code reviews, stress testing, fault injection, etc) with formal methods.

Answer 5: Because they are in a critical business domain.

Fig. 2. A poll


Why did Amazon AWS people started to look for formal methods?

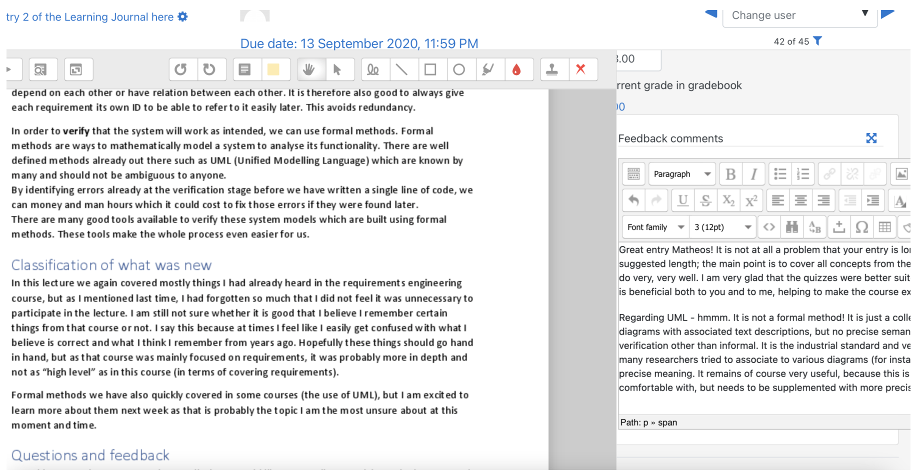
- a. Because they wanted to replace the standard verification techniques in industry (design reviews, code reviews, stress testing, fault injection, etc) with formal methods.
- b. Because they are in a critical business domain.
- c. Because Microsoft, Facebook and Google already used them, and they did not want to be left behind.
- d. Because of the complexity of their algorithms and code
- e. Because of subtle bugs in designs

Fig. 3. A quiz question

wanting to interrupt the teacher, what will other people think if I ask this, etc). The learning journals are the place to ask everything; they offer regular slices of personalised teaching, each learning journal entry being a private discussion between the teacher and the student. It is in fact very enriching for both parties. Comments like ‘more polls’, ‘less slides’, ‘go slower’, ‘can we study cloud architectures too?’ are only some examples of feedback to the teaching style. The most interesting and thought provoking questions and comments are those related to content, because the students understand the material in their own unique way, influenced also by their background and sometimes work experience. An example of a learning journal entry is shown in Fig. 4.

The third way of implementing repetition is by introducing three discussion seminars, paced throughout the course, one after three lectures. There we discuss ideally all that students would like to delve deeper into. In practice, students are again shy and expect the teacher to come up with topics of discussion. We set up little problems and exercises (sometimes posed a few days before the seminar) to sparkle up the discussion. This works better in the traditional setting than online, but overall, people like to be poked and prodded a little and asked to explain their opinion.

All in all, these three methods (polls, learning journals and discussion seminars) contribute to exercise all the topics we discuss about in the lectures. They reinforce the concepts, clarify them, and keep the students engaged and interested. The most rewarding aspect is that the students appreciate the effort (for



Due date: 13 September 2020, 11:59 PM

depend on each other or have relation between each other. It is therefore also good to always give each requirement its own ID to be able to refer to it easily later. This avoids redundancy.

In order to **verify** that the system will work as intended, we can use formal methods. Formal methods are ways to mathematically model a system to analyse its functionality. There are well defined methods already out there such as UML (Unified Modelling Language) which are known by many and should not be ambiguous to anyone.
By identifying errors already at the verification stage before we have written a single line of code, we can money and man hours which it could cost to fix those errors if they were found later.
There are many good tools available to verify these system models which are built using formal methods. These tools make the whole process even easier for us.

Classification of what was new

In this lecture we again covered mostly things I had already heard in the requirements engineering course, but as I mentioned last time, I had forgotten so much that I did not feel it was unnecessary to participate in the lecture. I am still not sure whether it is good that I believe I remember certain things from that course or not. I say this because at times I feel like I easily get confused with what I believe is correct and what I think I remember from years ago. Hopefully these things should go hand in hand, but as that course was mainly focused on requirements, it was probably more in depth and not as "high level" as in this course (in terms of covering requirements).

Formal methods we have also quickly covered in some courses (the use of UML), but I am excited to learn more about them next week as that is probably the topic I am the most unsure about at this moment and time.

Questions and feedback

.00

rrent grade in gradebook

)0

Feedback comments

Great entry Matheos! It is not at all a problem that your entry is lo
suggested length; the main point is to cover all concepts from the
do very, very well. I am very glad that the quizzes were better suit
is beneficial both to you and to me, helping to make the course ex

Regarding UML - hmmm. It is not a formal method! It is just a colle
diagrams with associated text descriptions, but no precise seman
verification other than informal. It is the industrial standard and ve
many researchers tried to associate to various diagrams (for insta
precise meaning. It remains of course very useful, because this is
comfortable with, but needs to be supplemented with more precis

Path: p » span

Fig. 4. A learning journal entry

instance to read and reply to the learning journals questions) and so it all seems like a good idea worth continuing with.

6 Students' feedback

In our university, the students' feedback is not mandatory nor recompensed, hence we have rather low rates of participation, about 30%. With this is mind, we explored the following questions for student feedback:

1. "I believe that the content of the course was, overall, according to its learning objectives." (4)
2. "The structure of the course helped me to achieve its learning objectives" (4)
3. "The course materials helped me achieve its learning objectives." (3.93)
4. "The teacher's pedagogical skills supported my learning." (3.73)
5. "I have actively participated in the course activities." (3.47)
6. "I feel that the course supported my education and/or future work." (3.8)
7. "I feel that the course has worked well in digital form." (4)
8. "Free comments on questions 1-7 and the course in general. What worked well and what less well?"
9. "The experience of writing a learning journal helped." (4)
10. "Could you please explain in a few words how was your experience of learning the whole course online? Was it lonely? Was it better? Did you miss anything?"
11. "Do you feel like you understand software quality better after this course?"

For questions 1.-7. and 9., students must choose between grades 1-4, 4 representing 'totally of the same opinion' and 1 meaning 'totally of the opposite opinion'. The scores obtained are indicated in parentheses. The comments for question 8. are below:

1. "Some lectures had a tad too much info, otherwise the format was good."
2. "Great material and engaging polls during the lectures."
3. "The course was quite interesting and thanks to the teacher, she never made the course boring."
4. "It was a very helpful course. The only thing which was quite hard for me is size of lecture slides, in my opinion there were too many slides."
5. "I feel good about the course content and the teaching method. Big thank to the lecturer!"
6. "For a remote course all went really well in my opinion"
7. "There's been only one issue with the course. I think it would have been better if students were expected to gain a certain number of points from their learning journal entries instead of having to submit all of them. Otherwise the course was very informative and the streaming format worked very well. I'm also glad that the teacher provided feedback to our learning journal."

8. "Very thorough course. I learned several concepts during this course. I like that it is in digital form."
9. "The course is designed, structured and executed well."

The answers to question 10. are below:

1. "Was better for me as i work besides school so was nice that the lectures were recorded so I could watch them later."
2. "It was well planned and went smoothly for me. In fact, online course is better than the traditional one as learning at own pace and own comfort zone is easier."
3. "It was ok, no problems."
4. "Face-to-face courses are always better in my opinion, but the teacher handled the distance courses very well. Being able to watch again the lectures is a big plus."
5. "It was great, easier to focus."
6. "I missed the face to face interaction with fellow students, but also the teacher. I can't speak to if the course was "better" when it was held online this year. But I can say that the course lectures did not suffer because of it. Maybe even better, because of the Zoom polls."
7. "Specifically for this software quality, it was quite good, not lonely at all. I like it."
8. "It worked perfectly with my situation at the moment, online studies and work are a perfect fit, almost"
9. "My online learning experience is good in general. Some major benefits: I can arrange my time to watch the recorded lessons later. I can target the interesting parts of the course. I can skip some parts that I already knew."
10. "For me remote learning is preferred, since I have to actively work. I could tackle immediate work tasks when needed (once during this course) without losing time to move around the campus."
11. "I actually prefer it to normal lectures, because I can always go back to recorded videos. I can also watch materials whenever it fits my schedule. I can manage my time more efficiently and so I learn better. Distant learning has been a bit lonely, but I think it works well with courses that consist mostly of lectures."
12. "My experience of this course is very good. I was learning it in my own time and I didn't feel much pressure."
13. " It was challenging to always try to keep focused when you are at home with so many distractions."
14. "I am used to the online mode of learning now, and it was designed engagingly with the discussion seminars and quizzes."

The replies to question 11. were different shades of 'yes'; one answer stood out: "Even if I have years of experience working as a programmer, the course is still very helpful to connect and fulfil my understanding of software quality, especially regarding the software quality practices from big companies."

7 Conclusions

Given freedom in the job of teaching a course whose content is not totally standard, there is a big temptation to choose one's favourite topic related to the course name. In the present case, formal methods were briefly considered to form the (only) body of the course, also given their relative under-representation in Computer Science and Engineering curricula. But formal methods require a certain background and a certain openness in the local industry - not everyone lives in Seattle to be hired by Amazon. More importantly, they do not tell the whole story of Software Quality. Getting the requirements well is certainly the starting point. The huge complexity of contemporary software makes formal methods inapplicable on the whole system in all details, requiring software architecture techniques to reach modules of functionality. Finally, we need a way to measure what we produce; logical properties again do not tell the whole story and need to be supplemented by measuring a variety of attributes (e.g., structure).

We taught this course in both settings - traditional and online. The traditional setting has the advantage of 'reading' the students much better. Compare discussing our topics with and in front of our students and getting immediate body language feedback on the quality of our discourses with looking at cameras and talking in front of our laptops, with tens of black rectangles representing the students, neither seen nor heard. The online setting has made it possible for more students to take the course (51 compared to 19), since the barriers of being in the classroom were eliminated. We recorded the lectures and saved the videos in the Moodle page of the course; we (a digital assistant) even split each main video of a lecture into several smaller videos addressing certain topics and saved those too in the Moodle page. These are likely to remain useful regardless of the teaching style. A hybrid style is in fact very probable, in which the traditional teaching is resumed, but the lectures are still recorded for those who cannot make it to class.

Finally, we found especially the learning journals and the polls to be extremely useful for learning. They seemed to also be entertaining for both teachers and students, and so learning becomes almost a game. These ideas can certainly be re-purposed for other courses as well, especially those courses with abstract and harder to grasp topics.

References

1. Direct poll. https://www.directpoll.com, Last accessed April 20, 2021.
2. M. Andreessen. Why software is eating the world. *Wall Street Journal*, 2011. Last accessed April 9, 2021.
3. L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 3rd edition, 2013.
4. P. Behm, P. Benoit, A. Faivre, and J.-M. Meynadier. Meteor: A successful application of b in a large project. In *International Symposium on Formal Methods*, volume 1708 of *Lecture Notes in Computer Science*, pages 369–387. Springer, 1999.
5. F.P. Brooks. No silver bullet – essence and accident in software engineering. *Computer Journal*, 20(4):10–19, 1987. Last accessed April 9, 2021.
6. B. Cook. Formal reasoning about the security of amazon web services. In *Computer Aided Verification*, volume 10981 of *Lecture Notes in Computer Science*, pages 38–47. Springer, 2018.
7. N. Fenton and J. Bieman. *Software Metrics: A Rigorous and Practical Approach*. CRC Press, 3rd edition, 2015.
8. J. Milord. No degree? no problem. here are the jobs at top companies you can land without one. *LinkedInNews, April 8*, 2019. Last accessed April 20, 2021.
9. C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff. How amazon web services uses formal methods. *CACM*, 58(4):66–73, 2015.
10. P. O'Hearn. Continuous reasoning: Scaling the impact of formal methods. In *Logic in Computer Science*, pages 13–25. ACM, 2018.
11. C. Sadowski, J. van Gogh, C. Jaspan, E. Söderberg, and C. Winter. Tricorder: Building a program analysis ecosystem. In *International Conference on Software Engineering*, pages 598–608. IEEE, 2015.
12. I. Sommerville. *Software Engineering*. Pearson Education, 10th edition, 2016. Last accessed April 20, 2021.