Teaching software-engineering concepts through industrially used tools early in the undergraduate curriculum

Dr. Temesghen Kahsai, Carnegie Mellon University

Dr. Temesghen Kahsai is a research scientist at Carnegie Mellon University Silicon Valley. His research expertise is in the intersection of formal methods and software engineering, with an emphasis on automated reasoning about software systems. He has extensive experience in the development of formal foundations and tools for the analysis of complex critical systems. He is currently leading three projects on contract-based verification of flight critical systems, analysis of heterogeneous software protocols stacks and credible auto-coding of embedded systems. These projects are funded by NASA and NSF.

Dr. Liam O’Reilly, Swansea University

Liam O’Reilly is a Senior Tutor at the department of Computer Science at Swansea University who specialises in Formal Methods, specifically verification using algebraic specification and processes algebra. Liam currently develops and delivers various courses as part of the undergraduate programme and an innovative work-based learning degree.

Prof. Markus Roggenbach, Swansea University

Associate Professor in Computer Science at Swansea University, who specialises in Formal Methods in Software Engineering. Area of expertise is verification of software designs, including tool support, specification languages, and testing. He is a member of the IFIP WG 1.3 "Foundations of System Specification”. He has significant experience of technology transfer through regular co-operations with industry, including Siemens, Rolls-Roys, and McAfee.
Teaching software-engineering concepts through industrially used tools early in the undergraduate curriculum

1 Introduction

This paper reports on an established course, running successfully for six years at Swansea University, Wales, UK. In this course carefully selected software engineering concepts (SECs) are taught to first year undergraduate students whose programming background consists of only one introductory course on programming (in Java). This course is also part of a work-based learning programme aimed at industry and is being taken by full-time employees from private companies. Due to the students’ limited programming background in both programmes, the selected SECs need to be demonstrable in a programming-in-the-small setting. Challenges to teaching SECs at such an early stage include:

**Challenge C1:** due to the students’ limited ability to think abstractly, students need hands-on experience to understand SECs.

**Challenge C2:** due to the students’ limited programming experience, students are critical of SECs and require convincing arguments that the taught SECs are relevant.

Our pedagogical approach to address these challenges is (a) to run a lab-centered course and (b) to let students see the “real thing” as often as possible.

To (a): Lectures introduce concepts and ideas that can later be experienced in carefully designed lab sessions. In the labs, we focus on SECs rather than programming by providing students with Java programs to be manipulated with tools. Topics covered include: code commenting with Javadoc, coding standards with Checkstyle, debugging in Eclipse, automated testing with JUnit, test coverage with Emma, automated GUI testing using software robots, and extreme programming applied to game development. This provides students with many concrete and accessible examples illustrating the selected SECs.

To (b): All of the tools mentioned above are used in industry: e.g., CheckStyle is standard in major companies, including the car manufacturer Daimler, Germany. The coding standard discussed is the Java Code Conventions. Extreme programming was developed to serve software needs at Chrysler, US. Lab examples are grounded in reality, e.g., testing of a “next date” function is motivated by presenting a bug report on German credit cards not working as they “can’t recognize the year 2010”. This evidence convinces students of relevance and applicability of the selected SECs.

The positive outcomes of our approach are demonstrated using various methods of assessment (MOAs) for evaluating the success of our course:

- After the course students understand formatting/commenting/documenting program code and how these help with maintenance; know how SECs are supported by tools; are able to explain extreme programming; and can develop test-suites using both white-box and black-box testing. (MOA: Exam results, see Appendix A)
The course has 100% student retention (MOA: over 5 years, no student failed the first year of the programme due solely to this course, i.e., the course is not a barrier to students. See Appendix A).

Students engage with the lab classes (MOA: ~80% of the students reliably participate in the labs, see Appendix A); enjoy the course (MOA: student feedback, interaction in the labs, see Appendix B); and are SE-tool literate (MOA: time to get new tools working in the labs reduces significantly over the course).

Seen from the overall curriculum perspective, teaching (selected) SECs in the first year provides a broad view on software development early on. This is beneficial as it

- avoids a narrow-minded view on Computer Science in which programming is perceived as “the centre of the universe”;
- illustrates abstract mathematical concepts in terms of SECs: ordering relations, equivalence relations, graph theory, and propositional logic are among the concepts underlying testing;
- provides examples for data structures: testing tools need to represent graphs, the axioms of abstract data types such as a stack can guide testing; and
- supports programming education in various aspects: students become better programmers as they
  o become tool literate;
  o learn about topics as testing and debugging presented as SECs; and
  o reflect on quality of source code as well as of programs.

Such a positive view is supported, e.g., by McCauley and Jackson who write: “an early and consistent emphasis on software engineering concepts creates a student culture that values the principles and practices of software engineering”\(^{15}\). Also Liew reports on positive results\(^{14}\).

Some of the SECs and tools discussed in our course, such as code documentation with JavaDoc or coding standards with Checkstyle, could also be taught in the context of a Java course. However, only within the context of the software lifecycle, cost and consequences become adherent. A correct and fully functioning Java program can be barely documented code or fail to adhere to strict coding conventions. Other tools, such as JUnit or Emma, address the topic of testing. The design of test suites is a process orthogonal to programming. Literature and industrial test practice agree that one shall undertake two independent developments: one for constructing the system (programming) and another one to examine the system (design of test suites). The process of testing is then seen as a comparison between the outcomes of these two, independent developments.

The rest of this paper is organized as follows: in Section 2, we discuss how our course aligns to the ACM/IEEE Computer Science Curricula 2013\(^4\); then, in Section 3, we describe the learning approach applied in our course; in Section 4, we detail selected lab experiments and their encompassing lectures; finally, in Section 5 we discuss related approaches.
2 Alignment to IEEE/ACM Computer Science Curricula 2013

The Computer Science Curricula 2013 (CS2013) provides a comprehensive set of recommendations for undergraduate education in Computer Science. To this end, it identifies and recommends “The Body of Knowledge” that a Computer Science curriculum should cover. This Body of Knowledge is divided into various “Knowledge Areas”, including, e.g., the area “SE-Software Engineering”. The area Software Engineering (SE) consists then of ten knowledge units, such as Software Processes, Software Project Management, Tools and Environments, etc. Each of these units defines a number of learning outcomes. These outcomes include a classification based on Bloom’s Taxonomy, where the CS2013 uses a scale of three levels of mastery, namely Familiarity (lowest level of mastery), Usage, and Assessment (highest level of mastery).

As our course is placed early in the curriculum, learning outcomes of our course are on the levels of Familiarity and Usage only. It would be too much to ask for the highest level of mastery, i.e., Assessment, during the first year of study. Assessment requires that a “student is able to consider a concept from multiple viewpoints and/or justify the selection of a particular approach to solve a problem.”

Our course is closely, however, not solely aligned to the CS2013 unit “Tools and Environments”. Concerning the six learning outcomes defined for this unit in CS2013, our course fully covers five out of these six learning outcomes, all of which on the mastery level Usage. In three cases this mastery level exceeds the mastery recommended by CS2013. In the context of alignment, we discuss three of these six learning outcomes in more detail:

1. We refrain from addressing “Describe the difference between centralized and distributed software configuration management. [Familiarity]” due to the constraints posed by teaching in the first year: software configuration management is beyond the scope of programming-in-the-small.

2. We achieve the same mastery level on “Demonstrate the capability to use software tools in support of the development of a software product of medium size. [Usage].”

3. We exceed the recommended mastery level on “Describe how available static and dynamic test tools can be integrated into the software development environment. [Familiarity]” by letting students have hands-on experience with testing tools such as JUnit, Emma, or Abbot, i.e., this learning outcome has mastery level Usage in our course.

Overall, our course ideally supports the following statement of the CS2013 section on Software Engineering: “Software development … requires choosing the most appropriate tools … for a given development environment.”

While focussed on “Tools and Environments”, our course supports nearly all of the listed knowledge units. It is reasonable not to support the knowledge unit “Software Design”. This is demonstrated, e.g., by its learning outcome “Articulate design principles including separation of concerns, information hiding, coupling and cohesion, and encapsulation.”, which is beyond the scope of programming-in-the-small. On the other hand, our course covers “Software Processes”. Here, one learning outcome is: “Explain the concept of a
software lifecycle and provide an example, illustrating its phases including the deliverables that are produced. [Familiarity]". To this end, our course includes, e.g., lab-classes on Extreme Programming that bring together all SECs and tools seen, i.e., this learning outcome is supported on the mastery level Usage. This lab-class additionally addresses the overall aim “the best way to learn to apply software engineering theory and knowledge is in the practical environment of a project”.

In combination with a subsequent course on Software Engineering in the second year, which includes a group project, and a substantial, individual software project in the third year, the Computer Science curriculum at our university fully covers the area of SE as suggested by CS2013.

3 A learning approach to Software Engineering

For effective learning of software engineering, we suggest a learning approach that

- appeals to the plug-and-play mindset of a student generation who loves to play with gadgets of all kinds,
- illustrates (selected) concepts of the discipline with hands-on experience, and
- is relevant to industry.

This eases teaching on the conceptual level, as – thanks to the above listed points – students engage with the course and buy-in into the concepts presented. Further, our playful approach provides students with a sense of achievement: students discover and use SECs with a concrete, visible effect on their screen – rather than being lost in, or even frustrated by, some intricate technical detail.

Our Computer Science programme considers guidance by an Industrial Advisor Board, which provides feedback from an industrial perspective. Frequently, company representatives state that their software development processes use many different tools (frequently more than 10). Furthermore, they state that a common problem in recruiting fresh graduate students is that considerable training is required before new employees are comfortable using a moderate number of tools. Industry requires employees who are tool literate. To this end, our course allows students to gain significant experience in the use of tools, allowing the students to be adaptable to using new tools they encounter.

To allow students to make meaningful experiments with tools supporting SECs in the lab classes, the lectures of our course provide the SE context within which these SECs live. In the following we discuss four principles underlying these lectures.

3.1 Learning SECs in the overall context of software lifecycle models

Fundamental to SE is the notion of the software lifecycle (SLC). The SLC divides software development into distinct phases with the intent of supporting planning, management, deployment, and maintenance of software products. The selected SECs belong to one or more phases of the SLC. Consequently, the lectures in our course discuss various SLC models (such as the Waterfall, V, Spiral, Incremental, and Iterative models) and point out typical
phases such as design, implementation, and testing. They provide an overview on typical languages supporting these phases. For specification, various concrete examples are shown in, e.g., natural language, UML, propositional logic, formal specification languages. The students are pointed to a number of programming languages. For testing, the course covers JUnit and gives a pointer to TTCN-3 (Testing and Test Control Notation version 3), a test language widely used in industry. The mastery level for these topics is Familiarity.

A characteristic for SE is that developments follow prescribed processes. In the lectures, our course discusses such processes for various SECs. A typical process is the development of a white-box test suite:

- **Activity 1:** Transform a program P to a program graph PG.
- **Activity 2:** Select a (minimal) set of runs R that provide certain coverage of the program graph PG.
- **Activity 3:** Select a set of inputs I that trigger the set of runs R. (There might not always be an input for each run \( r \in R \).)
- **Activity 4:** Derive from the specification SP the outputs O for the inputs I.
- **Activity 5:** Compile a test suite T consisting of inputs I and outputs O.

Typical for such a process is that it involves numerous artefacts of the SLC (here: the program P, the program graph PG, the specification SP, and the test suite T). The single activities can involve complex analysis steps, e.g., finding a minimal coverage for a program graph can be a challenge. Furthermore, these activities might partially fail. For instance, in Activity 3, it might be impossible to find an input triggering a specific run of the program. In this case, the process requires a documentation of the exception. The mastery level of this process is Usage.

Learning about such processes in the context of programming-in-the-small prepares students to cope with the far more complex processes of SE that they will encounter later in their studies.

### 3.2 Extreme Programming (XP) as instance of Agile Methods

In contrast to the traditional SLC models, our course includes sessions on XP as a concrete example of an agile method. Here, we present XP on the whole, covering notions such as the different roles in an XP team, the notion of a story, test cases as specification, 10-minute-build, weekly deployment, fortnightly sprint, quarterly reflection, and pair programming. This includes an open discussion of how much XP serves as a philosophy, consultancy method, or engineering discipline. In the lab classes, the students then apply XP within a mini project (see Section 4.6 for details). Thus, the XP component’s mastery level is Usage.

In the course, students gain experience with pair programming on two levels: throughout the lab classes, even before XP has been introduced, students are required to solve their lab tasks in groups of two; furthermore, they follow the concept of pair programming as part of the XP lab-classes. Based on this extensive experience, students are then asked to write a brief essay assessing pair programming, where they are also pointed to selected scientific literature. It is
at this stage, that our course involves an element towards mastery level Assessment. In our perception first year students find it challenging to take a stand in favour or against pair programming.

3.3 Software Quality

In the beginnings of Computer Science, having a program that just about does the job, might have been adequate. Nowadays, software products need to be developed with consideration of the long-term support requirements. Furthermore, standards such as the DO-178-C require integrity levels that are established by applying standard procedures such as code-inspections, walk-throughs, or testing.

The lectures of our course discuss the existence of industrial standards and their respective certification processes. We focus on two quality aspects, namely quality of source code as well as quality of programs. Concerning source code, we introduce two SECs: formatting to increase the readability of code, commenting to make code easier to understand, e.g., by explaining coding tricks. Both of these support maintenance, as – according to standard literature – code understanding takes more than 50% of time in maintenance. Testing is the most adopted dynamic technique in quality assurance in order to provide evidence that a program conforms to a given specification. The mastery level for these quality related SECs is Usage.

3.4 Tool Literacy

At Industrial Advisory Board meetings for curriculum development, industry regularly names “tool literacy” as a key skill that is high in demand. Software development in industry involves a number of tools, be it editors (textual or graphical), linkers, interpreters, compilers, code generators, debuggers, testing tools (documentation, test data derivation, test evaluation), static analysers, GUI tools, user interface management systems, configuration management tools, tracking tools, reverse and re-engineering tools, process support tools, management tools (Gantt and Pert Charts, cost estimation), to name just a few and also to ignore bespoke tools. Though a university curriculum can cover only a small number of such kinds of tools - and these only in selected instances – a Computer Science curriculum should train students to be flexible with regards to tools and to easily adapt to new tools.

This training includes building up a certain self-confidence needed to play with tools. For instance, under the Eclipse IDE, the installation of a number of tools requires the user to go through a number of menus in order to change settings. At first, students are reluctant to follow such procedures, as the risk of making a mistake appears to be too high. Another challenge is to cope with different user interfaces. While one tool offers a GUI with buttons for the different activities, another tool requires annotations in the programming text, while a third one needs compiler pragmas on the command line.

The lab classes of our course challenge the students to work with a new tool each week (on average). For each tool, the lab tasks range over tool installation, getting the tool to work on a simple example, and finally to solve a challenge with the tool’s help. Here, we observe that
the time to install tools and to get new tools working in the labs reduces significantly over the course. The time to solve the challenges depends on the difficulty of the problem at hand and is not part of tool literacy. The mastery level for the tools in the lab classes is Usage.

4 Lab experiments and their encompassing lectures

The overall format of the course is that there are lectures and lab classes on a two to one ratio. As described above, the lectures focus on SE in principle, introduce the SECs to be explored in the labs, and place the SECs into their SE context. Additionally, they prepare for the subsequent lab and consolidate the previous lab. Thus, each lab is embedded in the following way:

<table>
<thead>
<tr>
<th>Lecture</th>
<th>Lecture</th>
<th>Lab</th>
<th>Lecture</th>
<th>Lecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cons</td>
<td>SECs in SE Context</td>
<td>Prep</td>
<td>Tool Centered</td>
<td>Cons</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SECs in SE Context</td>
<td>Prep</td>
</tr>
</tbody>
</table>

First, the SEC idea is introduced and placed into the overall SE context (SECs in SE Context). On average, there are two 50 minute lectures scheduled for this. If need be, the second of these lectures includes a short, technical preparation phase (Prep). Then, the students explore the SEC by a plug-and-play approach in a tool centred, 50 minute lab. Finally, the next lecture starts with a brief conceptual and technical consolidation (Cons) of this lab. I.e., students are beforehand prepared for technical hurdles, and afterwards the larger picture is emphasized again.

In the following, we present for a number of SECs, the tools which we choose to demonstrate them, and the content of the labs with their surrounding lectures. Thanks to the hands-on experience in the lab, the mastery level for each of these SECs is Usage.

We illustrate in two figures the learning material from the idea and from the plug-and-play phase for the SEC code documentation, see Section 4.2. Figure 1 shows a sample slide from the lectures that explains how to write useful comments in the source code. Figure 2 shows an excerpt from the lab book, referring to the rule book seen in Figure 1.

4.1 SEC: Code documentation, Tool: Javadoc

Idea Ease maintenance / reduce maintenance costs as maintenance dominates the SLC. Literature indicates that time spent in the process of understanding the code is the biggest single issue in maintenance: documentation eases understanding. Present guidelines for code-documentation.

Technical preparation Demo of Javadoc

Labwork Use Javadoc to document a given program according to the presented guidelines. Check the result in a web-browser. Tangible lab-output: Commented Java file and HTML documentation of the supplied code.

Consolidation Awareness of the various tools used such as terminal, Java compiler, editor, Javadoc compiler, and web-browser. Insight: software development involves many
supporting tools. An IDE integrates such tools under one roof. Not all commenting concepts are supported with constructs in Javadoc, e.g., documentary comments such as the creation date don’t have a special Javadoc tag. This mismatch between concepts and constructs will appear regularly when discussing SECs and their underlying tool support.

4.2 SEC: Coding conventions, Tool: Checkstyle

Idea Large software projects utilise teams of coders. The resulting code must be readable by everyone. Coding conventions, such as the Java Code Conventions are heavily used in industrial practice.

Technical preparation Demo of Checkstyle and discussion of selected error messages.

Labwork The students shall find out what a provided, deliberately badly formatted, short program does. The students reformat this program manually in an editor according to the Java Code Conventions. After installing Checkstyle in Eclipse, the students reformat a long Java program until Checkstyle confirms it follows the coding convention. Tangible lab-output: two perfectly formatted Java programs.

Consolidation Awareness of coding conventions and an understanding of the benefits they bring to a team of programmers. Insight: tools can be used to enforce and improve code quality.

MR’s rule book: branching structures

Rule: Branching structures
State the purpose of the whole branching structure, explain what happens in each of the cases.

Example:
/* add polynomials only if they are of the same degree */
if degree(p) = degree(q)
then /* add the polynomials */
   {...}
else /* error case */
   {...}

Figure 1: Sample slide from the lectures.
4.3 SEC: Fault analysis, Tool: Eclipse debugger

Idea Explanation of the testing lifecycle, summarised by one prominent tester as “the first phase (coding) is putting bugs in, the testing phase is finding bugs, and the last three phases (fault classification, fault isolation, fault resolution) are getting bugs out”\textsuperscript{11}. Explanation of the concept of control flow.

Technical preparation None.

Labwork Students explore the Eclipse debugger in a playful way. Students debug two given Java programs, a short one and a longer, more complex one. Tangible lab-output: two bug free programs.

Consolidation A debugger, among other features, allows to pause and resume the control flow of a program and to inspect the current program state.

4.4 SEC: Black-box testing (two labs), Tool: JUnit\textsuperscript{12}

Idea Explanation and demonstration of black-box testing techniques, including Boundary Value Testing and Equivalence Class Testing.

Technical preparation Demonstration of JUnit (using Eclipse).
Labwork Lab 1: Students configure JUnit in Eclipse. Students encode a given test suite (derived with Boundary Value Testing) where each test case becomes one JUnit test. With this encoding they test a given Java program. **Tangible lab-output:** A test suite encoded in JUnit, test documentation of the automated test run.

Labwork Lab 2: Students encode a given test suite (derived with Equivalence Class Testing) where tests are represented in a dedicated data structure. With this encoding they test a given, buggy Java program. They debug this program till it passes the encoded test suite. **Tangible lab-output:** A test suite encoded in JUnit, a debugged program, and test documentation of the automated, passing test run.

Consolidation It is possible to keep the system under test (SUT) and test cases in separate files within one project. This allows for testing of the SUT as it stands, without any modifications. **Insight:** Students understand that JUnit automatizes test execution and test evaluation. Students learn that test suites are a data structure and that one can program in a testing language such as JUnit. Students also appreciate that Equivalence Class Testing can provide better quality assurance than Boundary Value Testing.

4.5 SEC: White-box testing, Tool: Emma

Idea Explanation and demonstration of white-box testing techniques, including Path Testing for the Test Coverage Matrix $C_0$ (every statement), $C_P$ (every predicate to each outcome), $C_k$ (every program path that contains up to $k$ repetitions of a loop).

Technical preparation None.

Labwork After installing Emma in Eclipse, the students are provided with a Java program and test suite (encoded in JUnit). The students explore how Emma reports on test coverage by removing test cases. The students are given a decision rich program and shall develop and encode a test suite that satisfies 100% coverage in terms of Emma. Students are free to follow a trial and error approach or to systematically develop the test suite according to the program graph as discussed in the lectures. **Tangible lab-output:** A test suite encoded in JUnit, test documentation of the automated test run and its coverage report.

Consolidation Emma reports on coverage at all code levels (method, class, package). This challenges students with the question for which code level Emma actually reports the coverage. Students learn to question and interpret tool output. **Insight:** proving coverage is complex and requires tool support for any non-trivial program.

4.6 SEC: Extreme Programming (two labs), Tool: Abbot Java GUI testing framework

Idea Students are taught on XP and pair programming (see above in Section 3.2).

Technical Preparation A simple computer game is demonstrated. Programming a software robot for GUI testing is explained.
**Labwork** The students are presented with three stories. For each story they first develop and implement a test suite in JUnit using the Abbot Java GUI testing Framework. They then implement the story; finally they test and debug their implementation. To this end, students receive a skeleton JUnit test file demonstrating the usage of Abbot. They further receive a skeleton program including the method declarations needed to implement the stories. The skeleton programs represent the results of previous sprints that the students now shall take forward with new stories towards the final product. *Tangible lab-output: A working interactive, graphical, fully tested computer game.*

**Consolidation** Students see how software robots allow for the automated testing of GUls. Students reflect on pair programming (see above in Section 3.2).

### 4.7 SEC: Version Control Systems, Tool: Subversion

**Idea:** Version control systems are vital tools that allow many programmers to work on the same code base simultaneously, manage releases of software and provide a mechanism to reliably track the development.

**Technical Preparation** None.

**Labwork** Each pair of students is given a pre-setup Subversion repository. They are given a strict series of steps, the purpose of which is to allow the students to see and resolve various situations in the repository such as a basic conflict or a merge operation. *Tangible lab-output: A small working Java program where both students have contributed to the source code.*

**Consolidation** Students gain an appreciation of how version control systems can support various stages of the SLC and ease the burden of version tracking within SE.

### 5 Related work

Several authors, including McCauley and Jackson\(^\text{15}\), as well as Liew\(^\text{14}\), share our view that SECs should be taught early on in a Computer Science curriculum. It is also a common view that students need to be exposed to current tools and techniques for software development and ideally gain experience in a hands-on manner, see, e.g., Teel et al.\(^\text{18}\).

Including lab classes and projects for SE within Computer Science programmes at later stages of the curriculum, where students have already learnt about advanced topics such as operating systems, databases, compilers, and computer networks, and are also fluent in programming including constructs supporting programming-in-the-large, is standard at most universities. The speciality of our course is that it provides students with genuine hands-on experience of a number of tools whilst their exposure to the Body of Knowledge\(^\text{4}\) of Computer Science is still limited.

Running lab intensive courses has become more and more standard in all fields of computer science. Guo et al. report in their paper “Learning Mobile Security with Android Security Labware”\(^\text{9}\) how to set up lab sessions and experiments for a course on mobile security. Deiters et al.\(^\text{5}\) discuss how to design practical exercises for global software engineering in a
lab-centered course. Designing and implementing lab-based courses such as ours becomes easier thanks to such experience reports.

Our approach nicely links in with concepts from Problem Based Learning (PBL)\textsuperscript{17} with respect to the hands-on approach, however, differs through the preparation and consolidation phases where students receive more guidance than in PBL. From our experience, these phases are important to ensure support for the entire cohort of students, where a too high percentage would be overwhelmed when asked to explore SECs on their own.

A course similar to ours can be found in CS2013\textsuperscript{4}. CS2013 documents in its appendix the course “CS169: Software Engineering, University of California, Berkeley”. CS169 appears also to cater for first year students. It covers similar areas of SE as our course, however, our tool involvement is far higher. Thus, our course delivers on a higher mastery level, namely Usage. Furthermore, thanks to the large number of tools involved, our students become tool literate (see Section 3.4 above).

6 Conclusion and Future Development

We explained the fundamental ideas underlying our SE course, which presents carefully selected software engineering concepts (SECs) to first year undergraduate students whose programming background consists of only one introductory course on programming (in Java). We also discussed, in some detail, how we designed lab classes for SECs that appeal to the plug-and-play mindset of a student generation who loves to play with gadgets of all kinds. These lab classes are surrounded by lectures that focus on SE in principle, introduce the SECs to be explored in the labs, and place the SECs into their SE context. Our course is aligned with the CS2013 guidelines, teaching experience and outcomes are positive.

Future developments shall cover elements of formal methods, e.g., on model checking based on the Java PathFinder\textsuperscript{19} tool or on program verification using the Dafny language and verifier\textsuperscript{20}. Both of these techniques have gained such maturity over the past few years, that we believe them now to belong to the syllabus of such a course.

Acknowledgements The authors wish to kindly thank all Swansea students of the courses CS-144, CS-135, and CSF-104, who – thanks to their successful participation and to their constructive critique over the years – helped us to fine-tune our teaching concept. We thank Geoff Dawkins for his help in adapting the course material for the work-based learning programme. We further thank the teaching assistants for their helpful support in the lab-classes, especially Phillip James, Mike Smith, and Josh Taylor. Another big thank you goes to Faron G Moller for his generous help with funding our participation at the 2015 ASEE Annual Conference and Exposition. Finally, we would like to thank Erwin R. Catesbeiana (Jr) for continuously challenging our views on SECs from a frog’s perspective.
References


Appendix A: Assessment and Attendance data

The overall course results for the last three years are:

<table>
<thead>
<tr>
<th>Students’ Average Mark</th>
<th>2011/12</th>
<th>2012/13</th>
<th>2013/14</th>
</tr>
</thead>
<tbody>
<tr>
<td>Students’ with a Good Outcome (Mark of 60%+)</td>
<td>68%</td>
<td>64%</td>
<td>65%</td>
</tr>
<tr>
<td>Failing Students (Mark between 0% - 29%)</td>
<td>75%</td>
<td>70%</td>
<td>73%</td>
</tr>
</tbody>
</table>

These results include lab-work, course-work, and an exam (weighted as 20%, 20%, and 60% respectively).

In 2013/14, the average attendance was 77% for the lab classes and the average mark was also 77%. This level of engagement and achievement is what the course has been designed for: students experience success with tools.

Appendix B: Student Course Evaluation

Comments by students in the course evaluation run by the university without any involvement of lecturing staff:

Under the heading “best things about this course”, students wrote:

- “Labs were very useful and they also linked to other courses.”
- “The lab sessions really helped me understand the work more clearly.”
- “The lab sessions were very interesting.”
- “This course enhanced my understanding of system testing and pair programming and also the importance of teamwork.”
- “The course was mainly practical – unlike other courses.”
- “Interesting Practical labs every week, which helped greatly with understanding the theory.”

Under the heading “what would you like to change about the course”, students suggested:

- “Better signposting for what topics we were covering to help with revision organization”.
- “More hands in the lab”.


Survey results on the course are typically of the following form:

Students rate courses on a scale from 1 to 5, where 1 represents “strongly disagree” and 5 represents “strongly agree”. The red, dashed line shows the result of this course, the blue, dotted line presents the average of all courses taught in the department.

The above graph shows that – in the view of the students – the module is of high quality and is well received.