JADE
THE JOURNAL OF ACADEMIC
DEVELOPMENT AND EDUCATION

Issue 12

Expected Publication Date: August 2020

ISSN: 2051-3593

Managing Editor
Dr. Russell Crawford

Administrator
Samantha Mottram

Telephone
+44 (0)1782 734436

Email
jade@keele.ac.uk

Web
https://www.keele.ac.uk/kiite/publications/jade/

Address
KIITE, Claus Moser Building, Keele, ST5 5BG

Article:

Foundations for Computing: Applying Pedagogy to a Foundation Year Computer Science Module

Dr Adam Wootton

Lecturer in Mathematics, Computing and Physics; Foundation Year; Keele University
a.j.wootton@keele.ac.uk

Abstract

Computers and Programming is a core module for Foundation Year students at Keele University intending to progress on to study Computer Science. This paper details a three-year reflection on how this module could be better delivered. After several areas for address were identified, it was found that changes to the structure, teaching activities and assessment of the module, in keeping with the principles of constructive alignment, led to significant improvement in student assessment performance. This work was completed as part of the reflective portfolio for Keele's MA Higher Education Practice *Design and Development in Higher Education* module.

This paper presents a critical evaluation of a Foundation Year module at Keele University, FYO-00096 *Computers and Programming*. This work was originally completed as part of a reflective portfolio for Keele's MA Higher Education Practice *Design and Development in Higher Education* module.

1 Introduction to FYO-00096

FYO-00096 *Computers and Programming* was first introduced in the 2013-14 academic year as a two-semester, 20 credit core module for Foundation Year students intending to progress on to study Computer Science. It has a very wide remit in giving the students an introduction to both computational theory and programming.

The intended learning outcomes for the module are as follows:

1. Demonstrate basic knowledge and understanding of computer architecture;
2. Describe and utilize various types of computer data, variables, programming statements, procedures and functions;
3. Define a problem with regard to design of a computer animation;
4. Break a programming problem into components in a hierarchy diagram;
5. Design an algorithm to solve a problem;
6. Represent a hierarchy diagram as Structured English and develop appropriate pseudocode;
7. Use a programming language to code an algorithm;
8. Produce a user manual for a piece of software and evaluate a piece of software;
9. Use binary arithmetic, boolean algebra and finite state machines to solve problems.

As a Foundation Year module, it also has a role in preparing students for a three-year Computer Science degree programme. Accordingly, the module must also give students a flavour of studying Computer Science. For example, the module should give a novice programmer the opportunity to determine whether or not they enjoy working on coding projects in their free time and, hence, if a full Computer Science degree programme is suitable for them.

The module is split into two distinct strands: computational theory and programming. These are taught and assessed separately, despite comprising the same module.

1.1 Module Changes, 2015 - Present

The review of the module began in the 2015-16 academic year, with one member of staff responsible for changing the delivery and assessment for the theory side of the module. In the 2017-18 academic year, the review also considered the programming side of the module, leading to further changes.

1.2 Teaching

A breakdown of the teaching structure of the module prior to2015-16 is given in Figure 1.

The module outline specifies a total of 23 hours each for lectures, programming laboratories and problem classes (i.e. one hour per week each). The theory side of the module was entirely delivered through the weekly lectures, although seven of the 23 problem classes were also given over to theory content at various points in the year. The remaining problem classes were used to teach the students about assembly language in preparation for an assembly language assignment. The 23 programming laboratories were used to teach the students about programming and program planning using the Processing language.

1.3 Assessment

The intended learning outcomes are measured by three assessments, listed below.

1. **A range of hardware and software tasks (25%):** split into two separate assignments, one theory assignment on logic gates (12.5%) and one assembly language programming assignment (12.5%).
2. **Algorithm and program design (50%):** when the module was first written, this was met by a portfolio comprising three separate elements, namely planning documents (17.5%), a user manual (7.5%) and Processing code (25%). A later iteration of the module added two formative and two assessed tasks in semester 1.
3. **2-hour examination (25%):** a two hour examination on all elements of the theory side of the module, held in semester 2.

This is summarised in Figure 2, which shows how the three assessments actually comprised eight summative assignments and two formative assignments, and that the programming side of the module was subject to a greater amount of assessment.

1.4 Structure of Paper

The remainder of this paper is organised as follows. Section 2 critically evaluates the module, identifying some areas that worked well alongside a number of possible improvements. Following this, Section 3 details changes made to the module in light of the reflection and Section 4 analyses the changes that this made to student performance. Section 5 concludes the paper.

2 Reflection on module organisation

Some aspects of the module that worked well at the start of the project. The language chosen for the programming portfolio, Processing, is particularly suitable for Foundation Year study. Firstly, the majority of introductory programming courses use Java (Major, Kyriacou and Brereton, 2012), including the Keele first year module CSC-10024 *Programming Fundamentals*. Processing is based on Java and uses much of the same syntax, meaning that *Computers and Programming* prepares students for their first year at Keele without replicating *Fundamentals of Programming*. Additionally, Processing is purposely designed as a language for first-time programmers and is based around visual, interactive media (Processing Foundation, 2019). This is important, since Cunniff, Taylor and Black (2013) and Milne and Rowe (2002) found that graphical programming languages can help beginners build a mental model of how programs work and eliminate common bugs. Finally, a typical cohort has a wide range of programming experience. As a less well-known language, Processing can level the playing field between complete novices and students who may have used some more common programming languages.

The general concept of the program design assessment was also a strong point. The self-directed coding project that formed the majority of the portfolio marks would take place over an entire semester. Students would decide on a program to make and would then spend the semester determining how best to do this using the taught Processing techniques. While staff assistance would be available in the laboratory classes, a good project would need a student to spend time developing their code independently. This gives the students a valuable experience of how their future studies could develop.

While the existence of a 20-credit module that is effectively split into two separate parts may appear idiosyncratic, this does afford some benefits in terms of module delivery. For example, it allows for flexible allocation of teaching time according to the needs of each side of the module, while also allowing the students the time to develop their programming skills over two semesters, rather than one. There are also opportunities to draw common threads between aspects of programming and theory, such as the theory of boolean logic and decisions in programs.

The project did, however, identify room for improvement. By the time that I became involved in the theory side of the module (2015-16), the curriculum needed refreshing. This is not unusual in the rapidly developing field of Computer Science. The biggest issue with the theory side of the module was that it was principally delivered through a single weekly lecture, with only sporadic support from other classes. While lectures are a convenient method for delivering information, they only foster the development of the lower level cognitive skills in Bloom's taxonomy (Bloom, 1956). The provision of a single hour of lecturing per week did not allow for the students to apply, analyse, synthesise or evaluate meaningfully, meaning that their understanding of the module content was poorly developed. Furthermore, Kolb (1984) emphasises the importance of experimentation and reflection in learning, which was not amply provided by a single weekly lecture. All of this is reflected in the relatively low mean exam marks of 51.20% in 2013-14 and 59.44% in 2014-15[1].

The programming side of the module also had scope for potential improvement. Processing was taught in programming laboratory sessions, where a section of the course notes (in .doc format) would be presented to the students, who would then be given the majority of the session to 'play around' with Processing. Perkins, et al. (2013) distinguished between two types of programming students: those who enjoy experimenting and modifying code, and those who stop when confronted with a problem. In a broader review of issues in teaching programming, Robins, Rountree and Rountree (2003) noted differences between effective novices, who can learn without excessive effort, and ineffective novices, who require close support while learning. It is easy to see how the 'play around' approach might benefit the former without providing the latter with adequate support.

Similarly, giving the students information on programming but leaving them to their own initiative in applying this knowledge left the students underprepared for the program design portfolio. Davies (1993) made a distinction between declarative programming knowledge, where a student might know the purpose of a technique such as an IF statement, and strategy, where a student knows when and why to apply this knowledge.

[1][1]These statistics, and all other module performance statistics reported, are calculated so as to not include results for students who did not submit any work for one or more of the assessments, on the basis that a zero mark in the exam for a student who was (for example) withdrawn by the University during the second semester would distort the overall average.

While the students may have gained declarative knowledge, there was less opportunity to learn strategy.

All of this is reflected in a number of the comments made by students in module feedback forms. In 2015/16, a number of the suggestions for 'what could be improved' followed the same theme: that the module would have benefitted from providing a more structured learning experience to help develop programming skills.

Ultimately, the problem this is best expressed using the 2001 revision of Bloom's taxonomy (Krathwohl, 2002). Both the programming portfolio and the assembly language assignment asked the students to plan and produce an original programming project, all of which requires the students to 'create', the highest level thinking skill in the taxonomy. This is difficult when most students had little opportunity to progress beyond the lower level thinking skills, 'remember' and 'understand'. It is, therefore, not surprising that the mean student mark for the programming portfolio at this time was 58.32%.

Given the study level and number of credits available for the module, the assessment load was excessively heavy, with the three nominal assessments effectively comprising ten different assignments. According to the principles of 'constructive alignment' outlined by Biggs (1996), assessment should require students to evidence that they can match the intended learning outcomes for the module. However, in FYO-00096, learning outcomes 2, 3, 4, 5, 6, 7 and 8 are all met by the programming portfolio, rendering the assembly language assignment and assessed programming tasks redundant. Given that a typical Foundation Year student will be studying this module alongside ten other modules over the course of the year, this goes against QAA's guiding principles for assessment (The Quality Assurance Agency for Higher Education, 2018). This sort of overassessment has been associated with surface learning (George, 2009), poor attendance (Jonkman, Boer and Jagielski, 2006) and stress (Cefai and Camilleri, 2009). One final issue was that the program plans were submitted as part of the final portfolio, meaning that most students based their plans on the finished program and did not meet learning outcomes 4 and 6.

A more generic issue facing Computer Scientists is the disparity in the numbers of men and women enrolled as students in the subject (Sax, et al., 2017). In the 2017/18 academic year, 12,885 female students (15% of total) and 71,125 male students (85% of total) enrolled on undergraduate Computer Science degree programs in the UK. This is in direct contrast to gender divide found elsewhere in UK higher education, where female students comprised 55.89% of all students enrolling on undergraduate degree programs (Higher Education Statistics Agency, 2019). The gender gap in FYO-00096 is broadly in line with that seen nationally, with Table 1 showing how female students have never made up more than 20% of a cohort.

Increasing female recruitment goes beyond the scope of this study, but providing an inclusive student experience is a key part of Keele's equality objectives for 2018 - 2022 (Keele University, 2018). It is vital to ensure that Computer Science classes provide a safe, accessible and inclusive environment for all students.

3 Methods for Innovation

3.1 Changes to theory delivery

As mentioned in the previous section, lectures deliver information to students, but only develop the lower level cognitive skills without additional support. Students must be given concrete experiences to reflect on in order to truly learn (Kolb, 1984). The lectures were changed to combine the delivery of information with regular example questions. For topics where example questions are not applicable, other methods are used to give students concrete experiences. For example, the lecture on internal hardware of a computer combines lecture slides with a live dissection of a computer, where students could handle individual components. Most importantly, these lectures were backed up with weekly problem classes devoted to the topic of that week's lecture. Work for these classes is released immediately after the lecture for students to attempt before class. Students are now able to have concrete experiences in the lectures, reflect on these between the lecture and problem class, and to then attempt active experimentation when presented with the problems.

These additional classes were supplemented with learning materials to be used outside of taught sessions, catering for students with different learning methods. For example, an interactive binary calculator was written in Excel, giving students visual solutions to conversions between base numbers, as well as fixed and floating point binary numbers. For those students with a less visual approach, the underlying formulae could be accessed and deconstructed. Since the examination covered two semesters worth of work, the students were also given an 'Exam Survival Guide' at the end of the second semester. This 39-page document covered every examinable topic, required methods and exam technique.

3.2 Programming teaching and assessment

One of the biggest changes made to the programming side of the module was the removal of the assembly language assignment, due to the intended learning outcomes being covered elsewhere, the relative irrelevance of low-level languages and the amount of assessment. However, assembly language was listed amongst the indicative content in the module outline, and so was not removed from the module entirely. It was instead incorporated into the theory side of the module as a part of the finite state machines topic.

The programming side of the module was completely restructured. Processing was retained, but the first semester is now devoted to learning different transferable techniques for coding in Processing. Each of these techniques is then assessed in the final portfolio. The semester culminates in a lecture on program planning, with the program planning assignment completed over the Christmas vacation. This feeds into the second semester, where the lab sessions are used to develop a game based on the plan. This is an example of the kind of problem-based learning that has previously been used successfully to teach programming (O'Kelly and Gibson, 2006).

There are several advantages to this. Firstly, having the program plans submitted prior to the second semester, rather than as part of the final portfolio, means that the students think ahead about breaking down a problem into steps, rather than completing it based on the final program design. This is more in keeping with intended learning outcomes 3, 4 and 6. Secondly, the portfolio assesses how well the students have met learning outcomes 2, 3, 4, 5, 6 and 7, meaning that the programming side of the module is constructively aligned (Biggs, 1996). Finally, by giving the students the opportunity to direct their own learning in the second semester, the students gain independent study skills that will benefit them on a Computer Science degree.

The changes to the programming labs drew on a range of literature. They now use a scaffolding approach, where students are given support from staff when beginning to learn new concepts before the level of support is gradually reduced over time (Wood, Bruner and Ross, 1976), and a spiral curriculum, which begins with a relatively simple concept that is then built upon until mastery of the subject is achieved (Bruner, 1996). Tan, Ting and Ling (2009) found that programming students respond poorly to lecturing and prefer to work from interactive examples, while Jenkins (2001) indicated that teaching programming is not about transmitting information but motivating students to solve problems and develop skills. In line with this, the lecturing time in the lab is kept to a minimum and the students are given directed tasks. Each session now has a ten minute lecture to introduce a topic, after which the students are given a lab task to complete by midnight on the following Sunday evening. This sharply contrasts with the 'play around with it' approach used previously, and was intended to help bridge the gap between effective and ineffective novices (Robins, Rountree and Rountree, 2003). For example, the final task introduces the students to arrays, but also requires that they use primitives, variables, animation, loops, decisions and user input. The repetition of these topics as part of new work allows the students to develop mastery. The fact that the majority of the lab classes are given over to working on these tasks means that the students are able to begin their work in a supportive environment with multiple staff members on hand to assist, allowing them to address any problems before potentially completing the work at home.

While these lab tasks were introduced as a part of the Range of Hardware and Software Tasks assessment and contributed 10% of the overall module mark, they were marked purely on engagement. There was one mark available for each section, and this would be awarded if the student had made an honest attempt at completing that part. This encourages engagement without deterring any ineffective novices.

The lab tasks allowed for the module assessment to be streamlined. Assembly language is now assessed as part of the exam, rendering the assembly language assignment unnecessary. Instead, the Range of Hardware and Software Tasks now consists of the logic assignment (15% of the overall module mark) and the lab tasks (10% of the overall module mark). The two formative and summative programming assignments were also redundant and, hence, removed. This all meant that it was possible to make the assignment more challenging by including a ‘functionality’ criterion that assessed how well the final program fulfilled its purpose. The updated breakdown of the module assessment is given in Figure 3. Overall, the number of assignments was reduced from ten to six.

3.3 Inclusivity

The reasons for lower female retention in Computer Science are complex, with Beyer, et al. (2003) citing the availability of same-sex peer support, characteristics of the faculty and the community environment as key factors. While the first two items go beyond the scope of this work, the classroom environment in FYO-00096 does not. The ideas of problems with the environment, including perceived stereotypes, isolation from peers outside of computing, sexual harrassment and a sense of belonging, have been echoed in numerous past studies (Cheryan, et al., 2019; Giannakos, et al., 2017; Master, Cheryan and Meltzoff, 2016; Michell, et al., 2017). Beyer, Rynes and Haller (2004), Malik and Al-Emran (2018) and Sax, et al. (2017) also emphasised a need to show that computing can help people, as female students were more motivated by being able to improve lives through their work than by their future job prospects.

Amongst issues such as a perception of computing being incompatible with communal goals and a relative lack of experience, Beyer (2017) identified one key problem as a lack of appealing role models. Hence, one of the simplest means of creating a more inclusive environment was to have a five minute ‘heroes of computing’ section at the start of each theory lecture. Each week, a relevant figure from the world of computing would be introduced to the group with a short, humorous biography. While many of the key pioneers from this time period come from a similar background (such as Boole, de Morgan and Babbage, or Shockley, Brattain, Shannon and others from 1940 - 1980), this gives ample scope to discuss important female computer scientists such as Margaret Hamilton, Sr. Mary Kenneth Keller and Grace Hopper. There is also cultural diversity, with the history of the binary number system covering Egypt, China, India and Mangareva. This all emphasises that contributions have been made by both men and women of varied backgrounds, which helps build a supportive environment. Additionally, humour and storytelling have been shown to have a positive effect on learning (Garner, 2006; Nasiri and Mafakheri, 2015; Papadimitriou, 2003; Short and Martin, 2011).

It is not easy to give a sense of Computing being used to help others during some of the earlier topics such as base numbers and logic, but the inclusion of two lectures on big data allowed for the presentation of case studies showing Computer Science in action, such as helping to prevent influenza outbreaks and combat online extremism.

4 Results

Tables 2, 3 and 4 show how the mean mark for each assessment changed, and in each case there is a clear improvement. Indeed, in all but one case, the increase was equivalent to one or even two grade boundaries. Table 5, meanwhile, demonstrates the significance of these changes, with particularly low P values seen for the Range of Hardware and Software Tasks and Program Design assignment. It should also be noted that this change comes in the face of rapidly increasing Foundation Year student numbers and decreasing UCAS tariffs (88 UCAS points in 2015 compared to 64 UCAS points in 2018). Collectively, the data show that the changes to the module improved student learning quantitatively.

Given some of the negative student feedback that the programming side of the module had previously received, it should be noted that in the last two academic years, not a single respondent has selected the ‘dissatisfied’ or ‘totally dissatisfied’ options for any part of the module[2].

5 Summary and Future Work

This critical reflection examined FYO-00096 and identified some positive aspects a number of areas for improvement. It was then shown how these areas were remodelled while the positive aspects were retained. A brief look at the module results showed that these changes made a significant difference to student performance in the face of increasing cohort sizes with lower entry requirements.

Despite this, there is still room to continue developing the module. One interesting possibility would be to take a flipped classroom approach to the programming labs (Lage, Platt and Treglia, 2000). By studying the lecture notes at home, the students would maximise the time spent applying them in the labs. For the theory side, further resources will be developed for learning outside of lectures. While recent research indicates that conventional lecture capture may have a detrimental effect on studies (Edwards and Clinton, 2019; Lyon, 2018), five minute video summaries of the key points from lectures should provide support without discouraging attendance. Conventional problem sheets will be replaced with homework, focusing problem classes on troubleshooting.

A repeated theme in the literature surrounding female Computer Science students is that classes should strive to include a greater emphasis on collaboration and helping others. With this in mind, the 2019/20 academic year will see the introduction of a

[2]31 Total Respondents

Foundation Year Computer Science module based around collaborative design under the instruction of a client.

References

Beyer, S., 2017. Women in Computer Science: Deterrents. In: Laplante, P. A. *Encyclopedia of Computer Science and Technology Volume II*. 2nd ed. Boca Raton: CRC Press. Boca Raton.

Beyer, S., Rynes, K. and Haller, S., 2004. Deterrents to women taking computer science courses. *IEEE Technology and Society Magazine*, 23(1), pp. 21–28. ISSN: 0278-0097. https://doi.org/10.1109/ MTAS.2004.1273468.

Beyer, S., Rynes, K., Perrault, J., Hay, K. and Haller, S., 2003. Gender Differences in Computer Science Students. In: *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education*. SIGCSE '03. Reno, Navada, USA: ACM, pp. 49–53. ISBN: 1-58113-648-X.

Biggs, J., 1996. Enhancing teaching through constructive alignment. *Higher Education*, 32(3), pp. 347–364.ISSN: 1573-174X.

Bloom, B. S., 1956. *Taxonomy of Educational Objectives: The Classification of Educational Goals*. Cognitive Domain. New York: David McKay Company.

Bruner, J. S., 1996. *The Culture of Education*. Cambridge: Harvard University Press.

Cefai, C. and Camilleri, L., 2009. *Healthy students healthy lives: the health of Maltese university students*. Malta: University of Malta European Centre for Educational Resilience and Socio-Emotional Health. Malta.

Cheryan, S., Lombard, E. J., Hudson, L., Louis, K., Plaut, V. C. and Murphy, M. C., 2019. Double isolation: Identity expression threat predicts greater gender disparities in computer science. *Self and Identity*, 0(0), pp. 1–23.

Cunniff, N., Taylor, R. P. and Black, J. B., 2013. Does Programming Language Affect the Type of Conceptual Bugs in Beginners' Programs? A Comparison of FPL and Pascal. In: Soloway, E. and Spohrer, J. C. *Studying the Novice Programmer*. New Jersey: Psychology Press, pp. 419–430.

Davies, S. P., 1993. Models and theories of programming strategy. *International Journal of Man-Machine Studies*, 39(2), pp. 237 –267. ISSN: 0020-7373.

Edwards, M. R. and Clinton, M. E., 2019. A study exploring the impact of lecture capture availability and lecture capture usage on student attendance and attainment. *Higher Education*, 77(3), pp. 403–421. ISSN: 1573-174X.

Garner, R. L., 2006. Humor in Pedagogy: How Ha-Ha can Lead to Aha! *College Teaching*, 54(1), pp. 177–180.

George, J. W., 2009. Classical Curriculum Design. *Arts and Humanities in Higher Education*, 8(2), pp. 160–179.

Giannakos, M. N., Pappas, I. O., Jaccheri, L. and Sampson, D. G., 2017. Understanding student retention in computer science education: The role of environment, gains, barriers and usefulness. *Education and Information Technologies*, 22(5), pp. 2365–2382. ISSN: 1573-7608.

Higher Education Statistics Agency, 2019. *Table 9 - HE student enrolments by subject of study 2014/15 to 2017/18*. Available at: <https://www.hesa.ac.uk/data-and-analysis/students/table-9>.

Jenkins, T., 2001. Teaching Programming - A Journey from Teacher to Motivator. In: *Proceedings of 2nd Annual conference of the LSTN Centre for Information and Computer Science*.

Jonkman, M, Boer, F. G. de and Jagielski, J., 2006. Are We Over-assessing Our Students? The Students' View. In: *Proceedings of the 17th Annual Conference of the Australasian Association for Engineering Education*.

Keele University, 2018. *Keele University Equality, Diversity and Inclusion Strategy 2018 - 2022*. Keele University. Available at: <https://www.keele.ac.uk/media/keeleuniversity/equaldiversity/ Equality % 20and % 20Diversity % 20Strategy % 202015 - 2020 % 20FINAL . pdf> [Accessed 29th May 2019].

Kolb, D. A., 1984. *Experiential learning: experience as the source of learning and development*. Englewood Cliffs, NJ: Prentice Hall. Available at: <http : / / www . learningfromexperience . com / images / uploads/process-of-experiential-learning.pdf>.

Krathwohl, D. R., 2002. A Revision of Bloom's Taxonomy: An Overview. *Theory into Practice*, 41, 4, pp. 212–218.

Lage, M. J., Platt, G. J. and Treglia, M., 2000. Inverting the Classroom: A Gateway to Creating an Inclusive Learning Environment. *The Journal of Economic Education*, 31(1), pp. 30–43. ISSN: 00220485, 21524068.

Lyon, S. D., 2018. Relationship between attendance, academic performance, and lecture-capture among veterinary students. MA. Oklahoma State University.

Major, L., Kyriacou, T. and Brereton, O., 2012. Systematic literature review: teaching novices programming using robots. English. *IET Software*, 6, 6, 502–513(11). ISSN: 1751-8806.

Malik, S. and Al-Emran, M., 2018. Social Factors Influence on Career Choices for Female Computer Science Students. *International Journal of Emerging Technologies in Learning (iJET)*, 13(05), pp. 56–70. ISSN: 1863-0383. Available at: <https://online-journals.org/index.php/i-jet/article/view/ 8231>.

Master, A, Cheryan, S. and Meltzoff, A. N., 2016. Computing Whether She Belongs: Stereotypes Undermine Girls' Interest and Sense of Belonging in Computer Science. *Journal of Educational Psychology*, 108, 3.

Michell, D., Szorenyi, A., Falkner, K. and Szabo, C., 2017. Broadening participation not border protection: how universities can support women in computer science. *Journal of Higher Education Policy and Management*, 39(4), pp. 406–422.

Milne, I. and Rowe, G., 2002. Difficulties in Learning and Teaching Programming—Views of Students and Tutors. *Education and Information Technologies*, 7(1), pp. 55–66. ISSN: 1573-7608.

Nasiri, F. and Mafakheri, F., 2015. Higher Education Lecturing and Humor: From Perspectives to Strategies. *Higher Education Studies*, 5(5), pp. 26 –31.

O'Kelly, J. and Gibson, J. P., 2006. RoboCode & Problem-based Learning: A Non-prescriptive Approach to Teaching Programming. In: *Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*. ITICSE '06. Bologna, Italy: ACM, pp. 217–221. ISBN: 1-59593-055-8.

Papadimitriou, C. H., 2003. Mythematics: Storytelling in the Teaching of Computer Science and Mathematics. In: *Proceedings of the 8th Annual Conference on Innovation and Technology in Computer Science Education*. ITiCSE '03. Thessaloniki, Greece: ACM, pp. 1–1. ISBN: 1-58113-672-2.

Perkins, D. N., Hancock, C., Hobbs, R., Martin, F. and Simmons, R., 2013. Conditions of Learning in Novice Programmers. In: Soloway, E. and Spohrer, J. C. *Studying the Novice Programmer*. New Jersey: Psychology Press, pp. 261–280.

Processing Foundation, 2019. *Overview. A short introduction to the Processing software and projects from the community*. Available at: <https://processing.org/overview/> [Accessed 22nd May 2019].

Robins, A., Rountree, J. and Rountree, N., 2003. Learning and Teaching Programming: A Review and Discussion. *Computer Science Education*, 13(2), pp. 137–172.

Sax, L. J., Lehman, K. J., Jacobs, J. A., Kanny, M. A., Lim, G., Monje-Paulson, L. and Zimmerman, H. B., 2017. Anatomy of an Enduring Gender Gap: The Evolution of Women's Participation in Computer Science. *The Journal of Higher Education*, 88(2), pp. 258–293.

Short, F. and Martin, J., 2011. Presentation vs. Performance: Effects of Lecturing Style in Higher Education on Student Preference and Student Learning. *Psychology Teaching Review*, 17(2), pp. 71 –82.

Tan, P., Ting, C. and Ling, S., 2009. Learning Difficulties in Programming Courses: Undergraduates' Perspective and Perception. In: *2009 International Conference on Computer Technology and Development*. Vol. 1, pp. 42–46.

The Quality Assurance Agency for Higher Education, 2018. *UK Quality Code for Higher Educaton: Advice and Guidance - Assessment*. Available at: <https://www.qaa.ac.uk/docs/qaa/quality-code/advice-and-guidance-assessment.pdf?sfvrsn=ca29c181_4> [Accessed 30th May 2019].

Wood, D., Bruner, J. S. and Ross, G., 1976. The Role of Tutoring in Problem Solving. *Journal of Child Psychology and Psychiatry*, 17(2), pp. 89–100. ISSN: 1469-7610.

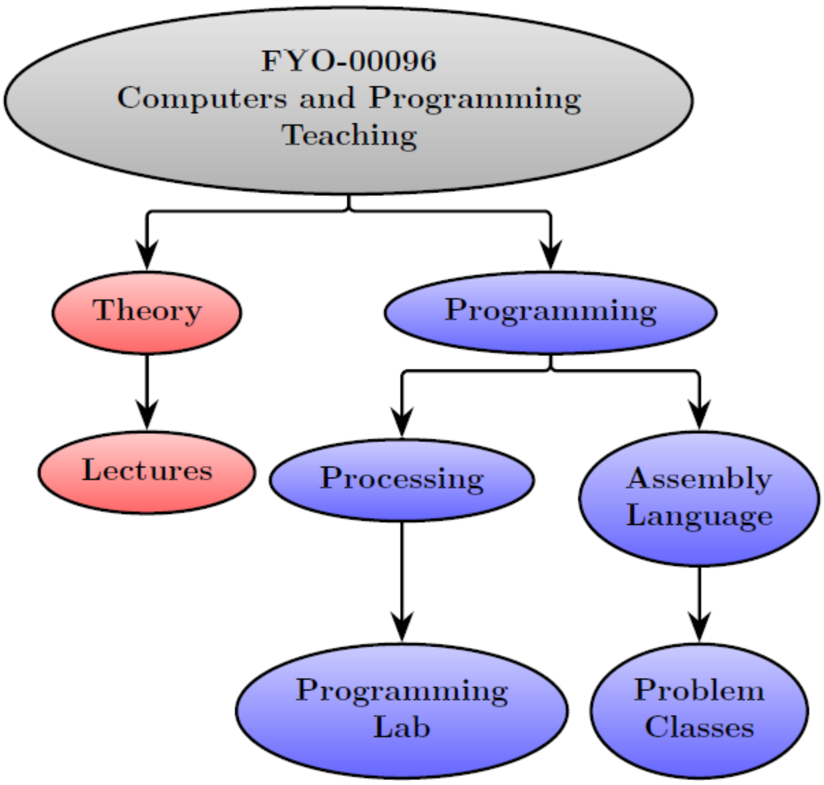

Figure 1: The division of taught sessions between theory and programming prior to the 2015-16 academic year.

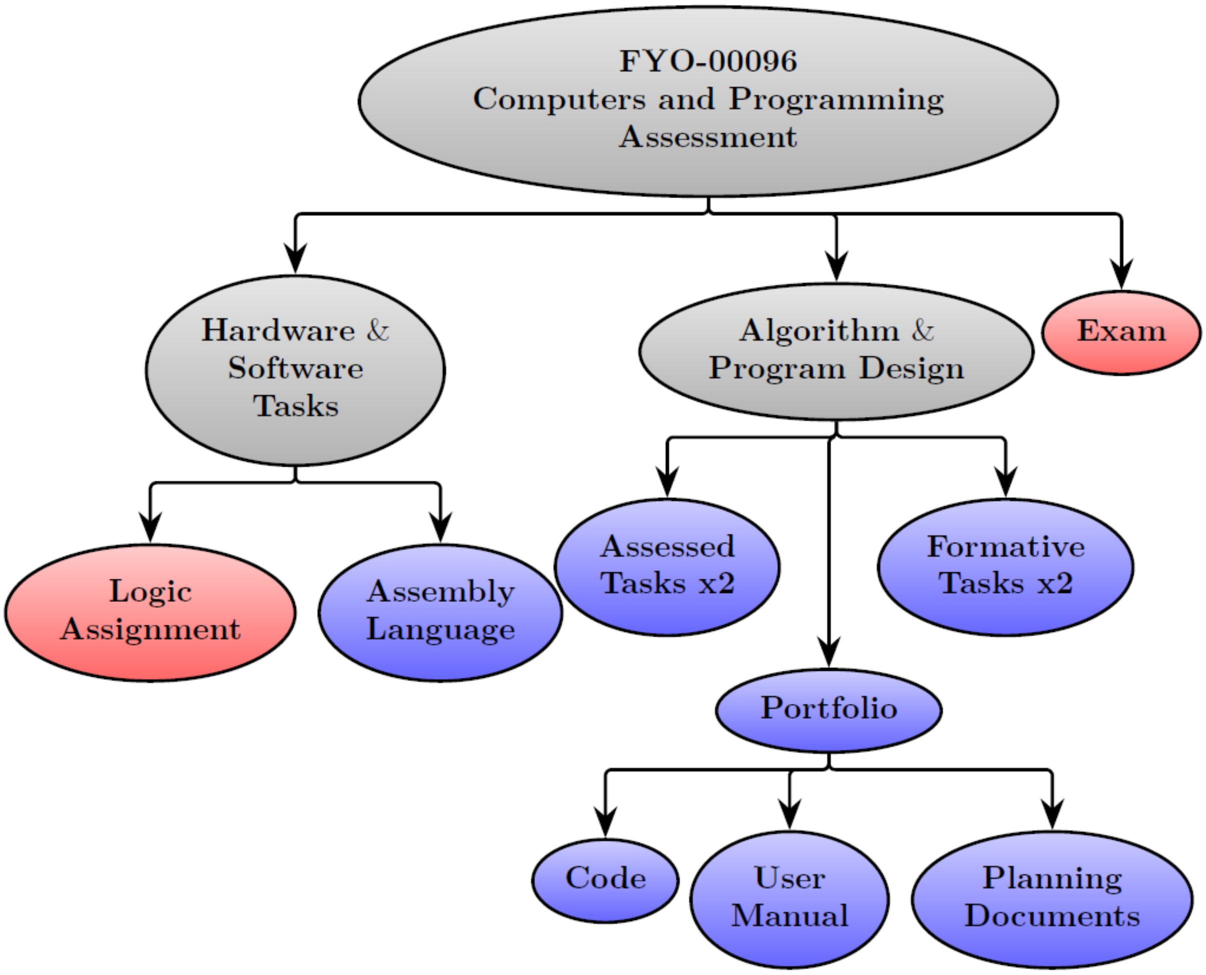

Figure 2: The assignments for FYO-00096 prior to my involvement, group according to the assessment that they contributed to. Assignments relating to theory are coloured in red, while those relating to programming are coloured in blue.

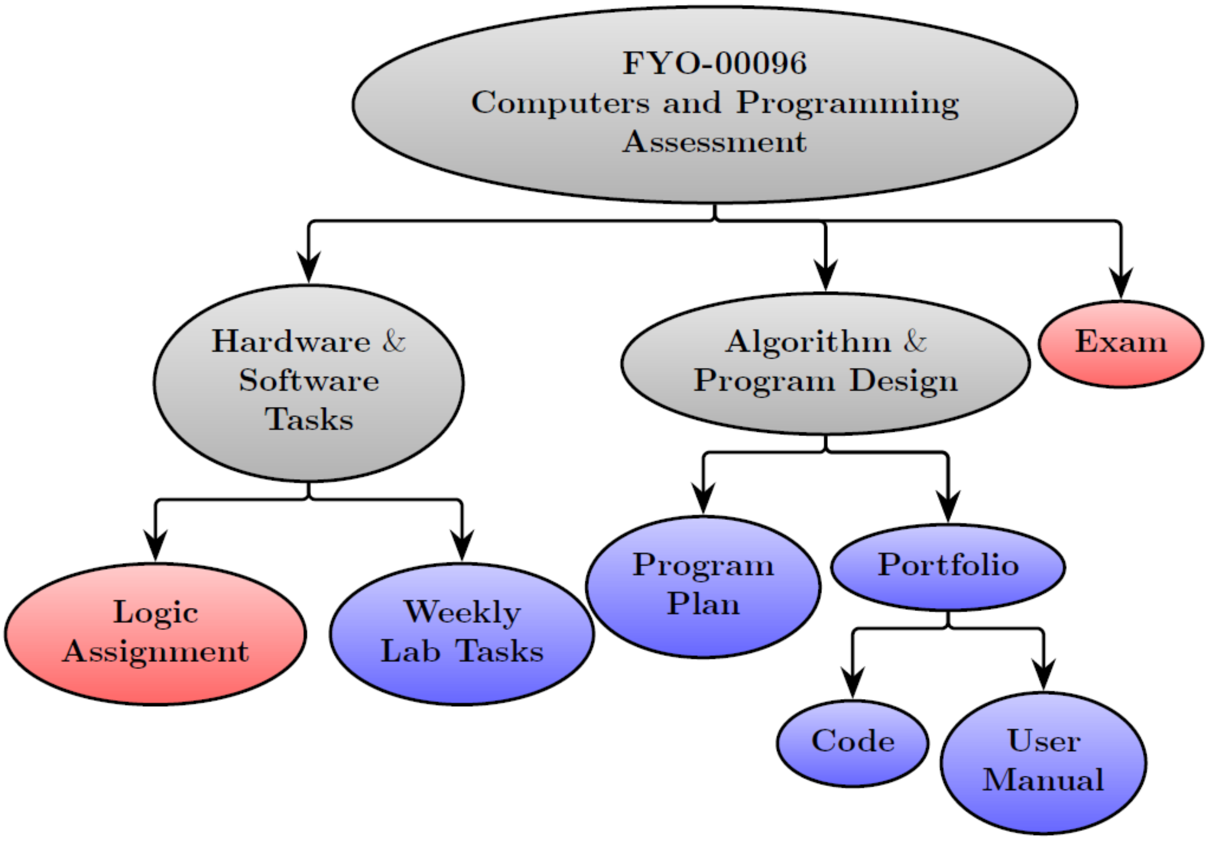

Figure 3: The revised breakdown of assessments for FYO-00096. Assignments relating to theory are coloured in red, while those relating to programming are coloured in blue.

Year	Male	Female
2013-14	85.71%	14.29%
2014-15	100.00%	0.00%
2015-16	80.65%	19.35%
2016-17	93.10%	6.90%
2017-18	80.00%	20.00%
2018-19	92.86%	7.14%

Table 1: The percentage of FYO-00096 students identifying as male compared to the percentage of students identifying as female, 2013 - 2019.

Year	Mean Mark (%)	Change from Before
2013-16 (Before)	57.02	N/A
2017-18 (After)	74.98	17.96
2018-19 (After)	77.29	20.28

Table 2: The effects of the changes to the module on the student marks for the ‘Range of Hardware and Software Tasks’ assignment. Note that the results for the 2016-17 cohort are not included in ‘before’ due to changes in the assessment caused by unexpected staff absence that year. Results for students who did not engage with one or more assignments are not included in the analysis.

Year	Mean Mark (%)	Change from Before
2013-17 (Before)	58.32	N/A
2017-18 (After)	72.39	14.08
2018-19 (After)	77.56	19.24

Table 3: The effects of the changes to the module on the student marks for the ‘Program Design’ assignment. Results for students who did not engage with one or more assignments are not included in the analysis.

Year	Mean Mark (%)	Change from Before
2013-15 (Before)	58.32	N/A
2015-16 (After)	69.37	13.72
2016-17 (After)	65.29	9.65
2017-18 (After)	71.01	15.37

Table 4: The effects of the changes to the module on the student marks for the exam. Results for students who did not engage with one or more assignments are not included in the analysis.

Assignment	N Before	N After	Mean Before	Mean After	P Value
H&S Tasks	34	43	57.02	76.38	0.0002
Program Design	57	35	58.32	75.05	0.0001
Exam	13	61	55.64	68.33	0.0175

Table 5: The number of samples N, mean marks and P values obtained for each assessment before and after the module changes using an unpaired t test. Note that the ‘Range of Hardware and Software Tasks’ results for the 2016-17 cohort are not included in ‘before’ due to changes in the assessment caused by unexpected staff absence that year. Results for students who did not engage with one or more assignments are not included in the analysis. Results for 2018-19 were only available for the ‘Range of Hardware and Software Tasks’ and ‘Program Design’.