CASE STUDY

Programming in Groups: developing industry-facing software development skills in the undergraduate mathematics curriculum

Matthew M. Jones, Department of Design Engineering and Mathematics, Middlesex University, London, UK. Email: m.m.jones@mdx.ac.uk.
Alison Megeney, Department of Design Engineering and Mathematics, Middlesex University, London, UK. Email: a.megeney@mdx.ac.uk.

Abstract

Programming is increasingly becoming an expected graduate skill for mathematics students. We argue in this article that programming should be given the same priority as any other graduate skill. Given the practical and philosophical constraints placed on undergraduate mathematics curricula, however, we acknowledge the difficulty in introducing, in a meaningful way, many of the core ideas of programming. We therefore present a case study of a second year course on an undergraduate mathematics programme that introduces Object Oriented Programming and aspects of software design, as well as key practical skill such as version control. We will argue that group assessment in this context is a more natural setting for students to be working and reflects more closely the experience of programming in industry; furthermore, it serves as a convenient platform to introduce students to aspects of software design and practical programming considerations. We will present an example of the type of assessment that can be used and how Version Control Systems like Git can be used to give students a more realistic experience of programming with the advantage of allowing tutors and other group members to track student work.

Keywords: Programming, Group assessment, Employability, Graduate Skills

1. Programming as a Graduate Skill

Historically, computing has been embraced by mathematicians as a tool for studying and solving problems in mathematics. The introduction of the NAG Libraries for FORTRAN in 1970 and TeX in 1979 serve as very early examples of its contributions. In different ways both of these had a significant impact on mathematics. However, they also highlight a common attitude of mathematicians to programming. According to Sangwin and O'Toole (2017) programming, as currently taught in undergraduate mathematics curricula across UK HEIs, largely reflects this natural order, often being introduced and taught in mathematics courses as a tool for solving specific problems: numerical solutions to ODEs/PDEs, numerical analysis, mathematical and statistical modelling and many other areas. This is likely the reason why the authors' findings suggest that languages such as MATLAB or R are amongst the more popular languages taught. The authors highlight a number of gaps, however, in the current offering by mathematics departments, not least the fact that programming paradigms such as Object Oriented Programming or Functional Programming may not be introduced to students in any meaningful way (p. 1145):

> *It is therefore somewhat surprising that [programming paradigms] are not currently taught and since they are at best optional, the vast majority of undergraduate students will never encounter these programming paradigms as part of their undergraduate education.*

In contrast the most popular languages for computing degrees (see Murphy *et al.*, 2017) are Java, C (and its successors C++ and C#), and Python; R is not being taught at all and MATLAB is taught in fewer than 3% of computing courses. This difference is explained by the fact that computing degrees have a clearer career progression, however it is also likely due to the relationship many professional mathematicians have with programming: it is a tool for solving specific problems or simplifying calculations. This is corroborated in Murphy *et al.* (2017) where the authors asked respondents *why* they chose their particular language. The most prevalent response, independent of the language, was its relevance in industry. Since the second most popular career choice for mathematics graduates is IT, according to Prospects (2019), and programming is increasingly becoming an important skill, we argue that it should be considered a *graduate skill* and that as mathematicians we should be mindful of this in curriculum design, even to the point of taking the lead from computing degrees. However, we also appreciate that mathematics degrees are, of course, not computing degrees and there are a number of hurdles to introducing graduate skills in mathematics curricula. Indeed, as Waldock (2011, p. 5) says,

> *There are significant barriers involved when seeking to modify Mathematics programmes to encourage the development of graduate skills. One is fundamentally philosophical, as some will wish to retain the pure, theoretical nature of their courses. Another is the practical difficulty of finding space for graduate skill development in a crowded curriculum.*

The view of students entering degree programmes in the UK has changed significantly in the last 20 years. The days when a university degree was seen as the sole route to career success have gone. In the most recent Global Learner Survey (Pearson, 2019), only 17% of UK respondents agreed with the statement that a college degree is essential to achieving a successful and prosperous career. This demonstrates a significant shift from previous studies. For example, a YouGov poll in 2012 found that 81% of respondents thought going to university was essential for them to pursue their career (Adediran, 2015). Additionally, in the Global Learner Survey, 66% of UK respondents believed a degree or certificate from a vocational college or trade school is more likely to result in a good job with career prospects than a university degree.

These changes in student attitude come at a time when the STEM skills shortage is highly publicised and a source of concern. UK government policy has in the last 10-15 years attempted to close this gap, and the extent to which universities should be responsible for addressing the shortage has been controversial in areas such as mathematics. However, with the publication of the so-called Augur report in May 2019 (Department for Education, 2019), there is a clear move to a situation where degree value is measured by graduate prospects rather than on its own merit. As a result, it is likely that graduate skills will become ever more important and will need to be transparent in the curricula of mathematics degrees in the future.

In this climate it is, therefore, becoming necessary for subjects like mathematics to reaffirm their position as career-facing subjects and, we would suggest, challenge the complacency that mathematics is, by some measure, top-of-the-pile in terms of its employability status. It is with this in mind that we have reconsidered how we teach programming on undergraduate mathematics degrees at Middlesex University, aiming to include specific, industry standard skills training that students can highlight to potential employers. And we have done this in a way that minimises the encroachment into the standard curriculum.

2. Context

The course we discuss in this case study is a second year undergraduate course on the BSc Mathematics programme. Students learn either R or Python in their first year and are introduced to

Java in their second-year. As is recommended in Sangwin and O'Toole (2017) this design means programming is taught throughout the first two years of the students' degree rather than in isolated courses, and remains optional in their third year. The course in question is a skills-based course, *Problem Solving Methods*, that introduces students to a wide range of techniques in applied mathematics as well as techniques to develop mathematical problem solving skills in pure mathematics (see Jones and Megeney, 2018). Workshops are inquiry-led, sometimes employing the Moore method (see Parker, 2005), to encourage students to develop their problem-solving skills and confidence. The content of the module ranges from areas of applied mathematics including optimisation, mathematical modelling, numerical methods and analysis, to areas of pure mathematics including number theory and real analysis. Students work weekly on different problems, developing strategies to solve abstract and unfamiliar problems, building a set of robust, internalised tools for enquiry. Programming is used as one such tool for examining problems and conjecturing solutions, and students are encouraged to see it as one of many avenues of progress. The structure of the workshops is heavily influenced by Pólya (1957), although expanded to include, as tools for examining problems, the use of software or programming. Whereas when Pólya wrote his work on solving problems he wrote about examining examples to get a better understanding of a problem, we encourage students to do the same using computers. The use of programming thus becomes one of the many integral tools available to students to study problems.

Students arrive in their second year with a good grounding in basic procedural programming and have developed some appreciation and experience of algorithm design. Introduction to a new language is therefore a matter of learning a new syntax (although further specific differences must also be mastered such as might be expected when learning a compiled, statically typed language).

Although the course content is taught in an informal workshop setting, many of the initial programming laboratories are taught more traditionally. Topics are introduced by the tutor and students work in pairs using the driver/navigator model, as described in Hannay *et al.* (2009) and Brown and Wilson (2018). There is an emphasis on teaching students many of the formal concepts from computer science that are necessary to implement object oriented design principles. We do not aim to teach aspects of functional programming; although Java does incorporate this paradigm in some sense, the course team does not believe it is in the interest of the students to confuse object oriented programming and functional programming. The taxonomy outlined in Figure 1, influenced by Selby (2015) and our own experience, is used as reference; it models the cognitive journey and, especially, our aspirations for where they will reach. The Aesthetics alluded to in the figure are not taught explicitly – instead students see aspects of them in the problems they solve and the assessment. The assessment of the course consists of individual coursework and group coursework; it is the latter that we wish to discuss here.

Students become accustomed to working in teams and presenting their work in class. This helps alleviate some of the issues common in group work as discussed, for example, in MacBean *et al.* (2004).

3. Structure of the group assessment

The philosophy of object-oriented programming lends itself naturally to group work, compartmentalisation of code allows group members to work independently of one another whilst still being part of a team. Indeed, aspects of high-level design such as design patterns, abstraction and inheritance are given a heightened importance – students must *design* the structure of the programme *before* they start coding in order to maintain compatibility.

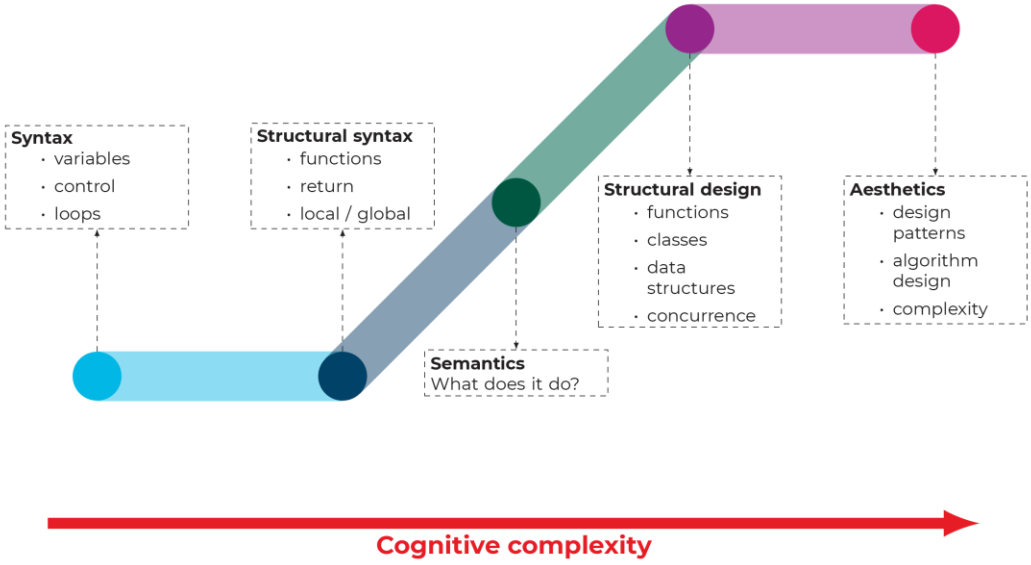

Figure 1: Programming taxonomy

In software design, design patterns are pre-packaged solutions to common problems, a classic reference to these is the so-called Gang-of-Four (Gamma *et al.*, 1995). In our opinion one of the most accessible design pattern for teaching is the Factory Design pattern (and, to a lesser extent, the Abstract Factory Design). We focus on this in our design of suitable assessment since it serves as a useful platform to further enquiry into design patterns. However it should also be noted that this is only our personal preference and other design patterns might also lend themselves naturally to group work.

The coding for the group assessment must be of the following form:

1. Decomposable into smaller problems
2. Each smaller problem should be solvable independently of the others
3. There should be an interface that ensures compatibility of code
4. The solution should lend itself to the Factory Design pattern (Figure 2).

Groups are arranged with a lead who will be responsible for the interface and acts as a client for the software – i.e. queries, runs and presents output. Other members of the group are responsible for solving the smaller parts of the problem.

As an example we might have groups write software that solves numerically an ordinary differential equations. Groups would normally be expected to use different numerical techniques such as Huen's method, or various other levels of precision of Runge-Kutta to find solutions. Individual group members can then take responsibility for each of the techniques used and the lead takes responsibility for the overall design.

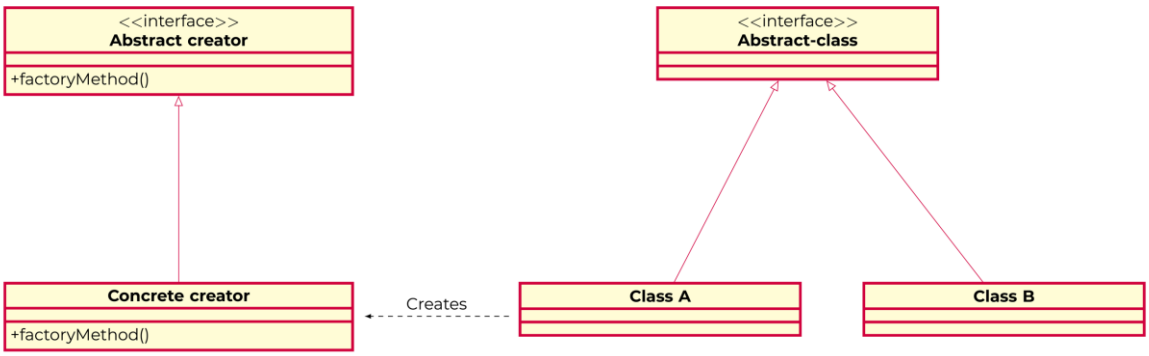

Figure 2: UML diagram for the factory design pattern

The advantage of structuring assessment in this form is that, with appropriate feedback, students will naturally discover that something approximating a factory design pattern must be used. Tutors then introduce students to Design Patterns or at least the Factory Design Pattern. Note at this stage students will have already encountered many design patterns in the course of their learning of object oriented programming, although may not have recognised them as such. For example the Iterator Design is built in to Java, and students will have seen the Adapter pattern when implementing data structures.

An important aspect of designing assessment that is decomposable like this is that students need to address the issue of version control – how does one know one is working on the current version of the code, or how does one avoid clashing with other work done. Version Control Systems are numerous, but the most common is Git. The university uses a local Git repository provided by GitLab. We do not advise students to use their own GitHub accounts to complete group assessment since elsewhere we promote GitHub as a convenient place for students to document a portfolio of work, so only final versions of software are made available on GitHub. Students are trained in the use of Git to maintain software, and the log from individual Git forks are used to ensure comparability of work effort in the final submission, thereby mitigating against the problem of 'coasting'.

4. Reflection and Concluding Remarks

Our approach to group assessment in programming discussed in this article is still in its initial stages with only two cohorts having been assessed this way. In a future article we expect to be able to evaluate the effectiveness using longitudinal data. However we are not yet at this stage.

Initially our main concern was that mathematics students would not feel comfortable learning technology like Git, especially those that were not aiming to go on to careers in software development. However, we have found that students have reacted positively to the opportunity of learning it. The following student's response summarises views on the usage of Git:

> *Being introduced to Git within the [undergraduate] degree is also helpful as the student can set up their own GitHub profile, load up their coursework and use this as a portfolio which is amazing for employability.*

In our experience students look favourably on opportunities to develop outward-facing exhibits of their work for employers and so even those not thinking of careers at this point see the advantage of using a tool like this.

We were also concerned that students would not be able to make the link between the structure of the assessment and the factory design pattern at all. Instead, in all cases, groups naturally designed their code with some of the notions of these patterns embedded. This helped improve confidence in their coding significantly and when, during formative feedback sessions, students were introduced to the formal factory design pattern it was evident that they made a significant connection to what can be an abstract idea. In some cases students went on to research more about design patterns and algorithm design.

Group work can be fraught with problems such as coasting, and a perceived increase in plagiarism. Indeed, there is growing scepticism amongst undergraduate students that it is worth the effort. Whereas plagiarism can be mitigated to some extent by assessment design, the coasting effect is certainly still an issue for some students using the approach described in this article. We prefer a proactive approach to dealing with these problems, intervening when needed. Communicating the use of Git to measure mutual effort has been useful, but we have not yet taken the approach of weighting group members' marks based on this. Given the approach described in this article is still evolving, we may well need to take a more authoritarian approach to this if necessary in the future.

In conclusion, considered as a pilot, our approach to introducing more advanced programming techniques in a group setting has been successful. In our experience students are well-suited to independently discover practical aspects of programming. It should be noted however that we have not yet encountered a situation where students have not independently discovered the ideas we have intended them to, and this will need to be considered in future.

5. Acknowledgments

This paper was presented at the "Programming in the Undergraduate Mathematics Curriculum" workshop held at Middlesex University on 27th June 2019. We would like to thank the sponsors of the event: IMA and Middlesex University.

6. References

Adediran, M., 2015. *Students value university education over costs*. Available at: https://yougov.co.uk/topics/politics/articles-reports/2015/05/01/students-value-university-education-over-costs [Accessed 30th January 2020].

Brown, N.C.C. and Wilson, G., 2018. *Ten quick tips for teaching programming*. PLoS Computational Biology, 14(4). https://doi.org/10.1371/journal.pcbi.1006023.

Department for Education, 2019. *Independent panel report to the Review of Post-18 Education and Funding*. Available at: https://assets.publishing.service.gov.uk [Accessed 30th January 2020].

Gamma, E., Helm, R., Johnson, R. and Vlissides, J., 1995. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley.

Hannay, J.E., Dybå, T., Arisholm, E., Sjøberg, D.I.K., 2009. The effectiveness of pair programming: a meta-analysis. *Information and Software Technology*, 51(7), pp. 1110-1122. https://doi.org/10.1016/j.infsof.2009.02.001.

Jones, M. and Megeney, A., 2018. Problem solving methods in undergraduate mathematics. In: *CETL-MSOR Conference 2018 Evidencing Excellence, 05-06 Sept 2018, University of Glasgow, Scotland*.

MacBean, J., Graham, T. and Sangwin, C., 2004. Group work in mathematics: a survey of students' experiences and attitudes. *Teaching Mathematics and its Applications*, 23(2), pp. 49-68. https://doi.org/10.1093/teamat/23.2.49.

Murphy, E., Crick, T. and Davenport, J.H., 2017. An Analysis of Introductory Programming Courses at UK Universities. *The Art, Science, and Engineering of Programming*, 1(2). https://doi.org/10.22152/programming-journal.org/2017/1/18.

Parker, J., 2005. *R. L. Moore: Mathematician and Teacher.* Mathematical Association of America.

Pearson, 2019. *The Global Learner Survey*. Available at: https://www.pearson.com/content/dam/global-store/global/resources/Pearson_Global_Learner_Survey_2019.pdf [Accessed 30th January 2020].

Pólya, G., 1957. *How to Solve It*. Princeton, NJ: Princeton University Press.

Prospects, 2019. *What do graduates do?* Available at: https://luminate.prospects.ac.uk/tag/reports [Accessed 30th January 2020].

Sangwin, C.J. and O'Toole, C., 2017. Computer programming in the UK mathematics curriculum. *International Journal of Mathematical Education in Science and Technology*, 48(8), pp.1133-1152. https://doi.org/10.1080/0020739X.2017.1315186.

Selby, C., 2015. Relationships: computational thinking, pedagogy of programming, and Bloom's Taxonomy. *The 10th Workshop in Primary and Secondary Computing Education, United Kingdom.* pp. 80-87. https://doi.org/10.1145/2818314.2818315.

Waldock, J., 2011. *Developing Graduate Skills in HE Mathematics Programmes - Case Studies of Successful Practice*. Birmingham: Maths, Stats and OR Network. Available at: http://www.mathcentre.ac.uk/resources/uploaded/gradskills.pdf [Accessed 30th January 2020].