Author Name: Neil Anderson
Contact Email: n.anderson@qub.ac.uk

STRANGE, BUT TRUE? OBJECT-ORIENTED PROGRAMMING IS BEST TAUGHT, AND LEARNT, WHILE SITTING ON THE FLOOR.

Neil Anderson, Aidan McGowan, Philip Hanna & John Busch
The Queen's University of Belfast

Abstract

There is a perception amongst some of those learning computer programming that the principles of object-oriented programming (where behaviour is often encapsulated across multiple class files) can be difficult to grasp, especially when taught through a traditional, didactic 'talk-and-chalk' method or in a lecture-based environment.

We propose a non-traditional teaching method, developed for a government-funded teaching training project delivered by Queen's University, we call it *bigCode*. In this scenario, learners are provided with many printed, poster-sized fragments of code (in this case either Java or C#). The learners sit on the floor in groups and assemble these fragments into the many classes which make-up an object-oriented program.

Early trials indicate that *bigCode* is an effective method for teaching object-orientation. The requirement to physically organise the code fragments imitates closely the thought processes of a good software developer when developing object-oriented code.

Furthermore, in addition to teaching the principles involved in object-orientation, *bigCode* is also an extremely useful technique for teaching learners the organisation and structure of individual classes in Java or C# (as well as the organisation of procedural code). The mechanics of organising fragments of code into complete, correct computer programs give the users first-hand practice of this important skill, and as a result they subsequently find it much easier to develop well-structured code on a computer.

Yet, open questions remain. Is *bigCode* successful only because we have unknowingly targeted predominantly kinesthetic learners? Is *bigCode* also an effective teaching approach for other forms of learners, such as visual learners? How scalable is *bigCode*: in its current form can it be used with large class sizes, or outside the classroom?

Introduction

In 2012, The Royal Society in London published a report, '*Shut down or restart?*' (The Royal Society, 2012). This seminal investigation, which details the way forward for computing in United Kingdom (UK) schools, assesses the current provision for Computing (as an academic subject) in schools in the UK. It also seeks to understand what, if any, preparation, a subject called ICT (Information and Communications Technology) affords students as they exit secondary education to commence Computing disciplines at university or in the workplace.

The subject of ICT was originally designed as a broad-brush introduction to the aspects of applied computing, a suitable preparation for those students who would be expected to be proficient in the use of computer programs in a workplace environment. ICT was not designed to provide preparation for those students that would eventually be designing, developing and deploying computer systems and software projects in a professional capacity. Indeed, the first main finding of the report from the Royal Society report says:

> The current delivery of Computing education in many UK schools is highly unsatisfactory. Although existing curricula for Information and Communication Technology (ICT) are broad and allow scope for teachers to inspire pupils and help them develop interests in Computing, many pupils are not inspired by what they are taught and gain nothing beyond basic digital literacy skills such as how to use a word-processor or a database.

Author Name: Neil Anderson
Contact Email: n.anderson@qub.ac.uk

The report goes on to state, that:

> There is a need for augmentation and coordination of current Enhancement and Enrichment activities to support the study of Computing.

Change came in short order. In late 2012, Ofqual (The Office of Qualifications and Examinations Regulation) which regulates qualifications, examinations and assessments in England and vocational qualifications in Northern Ireland, initiated reform of AS and A level qualifications (Ofqual 2015). This included reform of Computer Science and ICT subjects at AS and A level. The timeline for delivery of these revised subjects was set for the years 2015 to 2019. At present all of the English examination boards have produced revised A level Computer Science qualifications for delivery from 2015.

In 2013, the local Northern Ireland examining board, CCEA (Council for the Curriculum, Examinations & Assessment), introduced a new A level specification entitled Software Systems Development (SSD). CCEA, encouraged by universities and industry alike, developed new curriculum to teach and assess the fundamental concepts of computer programming, including Object Oriented Programming (OOP) and Database Design.

While SSD is only in its infancy, the level of participation is worrying. In 2014 a total of 37 students were enrolled in the SSD AS-Level course by schools across Northern Ireland (AS level is an Advanced Subsidiary level, the first part of the current A level qualification offered by educational bodies in the UK). In contrast, there were 1886 students enrolled in the ICT AS level course. In the following academic year, Queen's University Belfast admitted 422 students to Computer Science, or Computer Science-related, courses where each student is required to undertake and pass a module in computer programming including OOP. At tertiary level education there is good demand for students with strong OOP skills.

There is also strong demand from industry for graduates skilled in computing (this area of industry is sometimes known as the 'tech sector'). The latest '*Tech Monitor*' report (KPMG, 2015) from KPMG (a global network of professional services firms) outlines the demand from the tech industry in the final quarter of 2014. The report says:

> Tech sector job creation and new business trends also exceeded UK-wide benchmarks by substantial margins in the final quarter of last year, with firms citing a wave of new product launches and greater investment spending.

Cleary the lack of participation in SSD at AS-Level is not caused by a lack of demand for students with programming skills from university or industry. Instead, the problem originates from the lack of teaching staff with the corresponding skills and experience needed to teach OOP. These teachers are in chronic shortage in Northern Ireland. In response to this skills gap, the Department of Education (DE) in Northern Ireland funded a three-year 'up skilling' course to provide teachers with the skills they need to teach OOP in secondary education. The course is delivered by Queen's University Belfast, and the *bigCode* teaching method described herein is an initiative developed as part of this course.

Background

The purpose of computer programming is to write a sequence of instructions that will allow a computer to automatically perform a specific task or to solve a given problem. There are a number of programming paradigms (ways of building the structure and elements of computer programs), which provide for fundamentally different styles of computer programming.

Author Name: Neil Anderson
Contact Email: n.anderson@qub.ac.uk

The procedural paradigm and the object-oriented paradigm are two of the most important paradigms, and while the two are not mutually exclusive, the distinctions between them are significant enough to warrant attention.

Procedural Programming

Procedural programming is intuitive. A programmer writes the set of instructions in the order in which they want to the computer to execute these instructions. Indeed, procedural languages are often referred to as 'top-down' languages. This is a reference to the fact that the programming language follows Western reading order of left to right, top to bottom.

In procedural programming it is also possible to write routines. These portions of code, which may be called and executed from anywhere in the program, usually contain step-by-step instructions for commonly-performed tasks, such as printing text to the screen. Again, the instructions contained in these routines are executed in the same 'top-down' manner.

Object-oriented Programming

OOP takes a fundamentally different approach to procedural programming (Kölling, 1999). In OOP, the programmer logically divides the problem they are trying to solve into small components, each of which is considered an object. A typical object-oriented program comprises many objects; each of these basic units can store information, perform certain behaviors and interact with other elements of the program.

Most importantly, the objects often relate to real-world entities. For example, consider an animal. In OOP, we say that an animal is an object. We would expect that an animal would have some attributes, say for instance, a name. In OOP this is called a property of the object. We would also expect that the object should be able to undertake some actions, for instance, an animal will eat and sleep. We call these actions the methods of the object.

When the program runs on a computer, each object is created from a class file. A class is often defined as a blueprint or a template for an object. The class contains the definitions for the behaviours and properties for a single object.

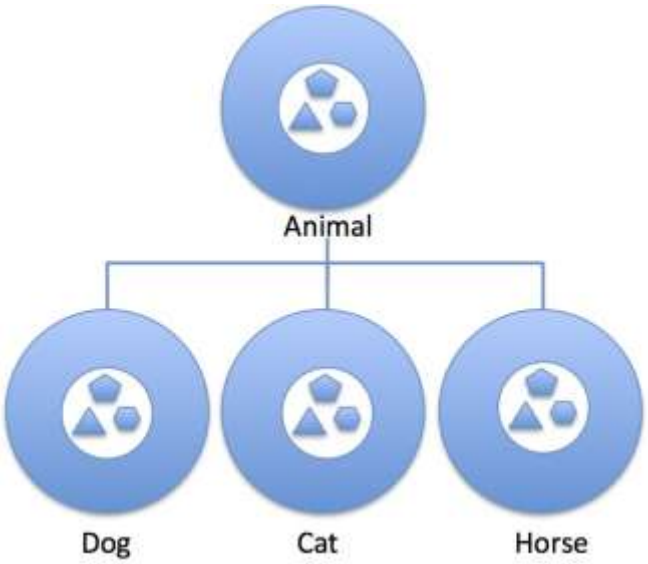

Figure 1. A diagram depicting a simple inheritance hierarchy for some animals

Author Name: Neil Anderson
Contact Email: n.anderson@qub.ac.uk

Objects often have a certain amount in common with each other. The embodiment of this concept in OOP is called inheritance. Inheritance allows us to arrange classes in a hierarchy where we can define an "is-a" relationship between these classes. Dogs, cats and horses, for example, both share the characteristics of animals (they sleep and they eat). Yet each also defines these features in a way that make them different: dogs eat anything; cats eat chicken and horses eat hay.

Object-orientation is intrinsic to languages such as Java and C#. Skills in both of these languages are in high demand by software development companies. Consequently, there is great pressure on secondary schools to teach OOP so that their students are adequately prepared for university Computer Science courses.

Teaching Object-oriented Programming

Learning to program a computer is difficult, and novice programmers experience a wide range of difficulties while learning their craft (Robins et al, 2003).

For many students, aged 16 or 17 and learning OOP in secondary school, this is their first experience of computer programming. (There is, at present, a drive to introduce computer programming earlier in the UK National Curriculum, from as early as age five in primary school.) Accordingly, much time is spent teaching these students the fundamentals of computer programming, such as syntax, variables, data structures, loops and conditional statements to name but a few. Consequently, there is not always a great deal of time remaining to cover, in a meaningful manner, concepts such as objects, classes and inheritance.

In general, programming suffers from the paradox of attempting to teach a practical subject using predominantly traditional theoretical methods. Indeed, there are also a number reasons why, in a traditional setting, it is difficult to teach these concepts well. We examine each of these in turn.

Practical Laboratory Sessions

Practical laboratory sessions are invaluable for students when learning to program. They offer students the opportunity to practice writing code in a controlled environment. Modern Integrated Development Environments (IDE), such as Eclipse also play a supportive role. Features such as code completion, automatic formatting and the relevant colour-coding of reserved words in the code can all help a student write better computer code.

IDEs are primarily industrial tools, which we use in order to expose our students to industry practices. However, the very features that are intended to support a professional programmer are often inhibiting to the learning process for a novice programmer. A student can very quickly learn to rely on the features discussed above, rather than learning the fundamental principles of programming for themselves.

There is also another fundamental limitation concerning the use of an IDE in a teaching environment. As shown in Figure 2, the ancillary features of the IDE, such as the Package Explorer and the Console, for example, consume valuable screen real-estate. As a consequence the screen area for the program code is reduced. It would, of course, be trivial to pop-out the program code for each class into separate windows; however, even a simple program can often contain numerous classes. Our animal example, as shown in Figure 1, has five different classes. It quickly becomes challenging for a novice student programmer to maintain, in their head, a conceptual object-oriented model of all five classes, as well as the corresponding relationships between each class. This is especially difficult when you consider that the student is, at the same time, just beginning to learn the fundamental principles of objected-oriented design.

Author Name: Neil Anderson
Contact Email: n.anderson@qub.ac.uk

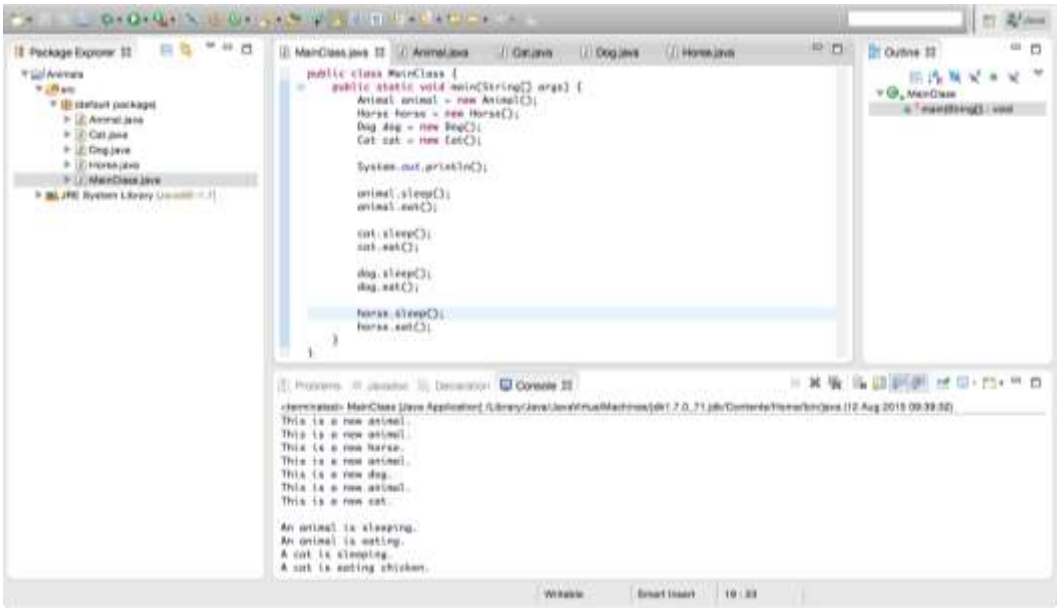

Figure 2. A screenshot of the Eclipse IDE being used for Java development

Traditional Classroom/Lecture Sessions

More recently, it has become common for tutors to interlace their theory-based material with live code demonstrations (Rubin, M. J. 2013; Samuel, S. 2015). These 'code demos' have proved immensely popular among students as they give the students an opportunity to learn a piece of theory and immediately see the same theory in practice. However, useful as they are, live code demonstrations are still limited.

First, the tutor, rather than each student controls the pace of the demonstration. Of course, the tutor is usually sympathetic to student requests to revisit a particular aspect of the demonstration, but class time is ultimately finite, so there is a limit to the time available for a live demonstration.

Second, the tutor is bound by the same screen real-estate limitations that affect students in the laboratory sessions. It is nearly impossible for the tutor to fit all of the relevant program code from each class onto a single screen, even for a relatively trivial object-oriented program.

An ever-increasing number of tutors are now recording their live code demonstrations and publishing these to video sharing platforms such as YouTube (Holliman and Scanlon, 2004; McGowan, A., Hanna, P., & Anderson, N, in press). These recorded demonstrations instantly solve the issue with control of pace. They firmly put the learner in control of the demonstration, allowing them to fast-forward, re-wind and re-watch as often as they like. Unfortunately the recordings obviously do not provide any additional screen real-estate.

Eureka!

In 2014 a number of Computer Science lectures were scheduled in a newly refurbished lecture theatre. As part of the refurbishment, the room, which is unusually wide, was fitted with two data projectors and two screens, sited side-by-side. In most circumstances, both screens simply mirror the presenter's computer. However, crucially, it is possible to use two computers to present independently on each screen.

Author Name: Neil Anderson
Contact Email: n.anderson@qub.ac.uk

This configuration of a teaching room will provide for vast improvement in the ability to teach OOP. At last, students can now see the 'full picture'. No longer will a lecturer have to constantly switch between different tabs in the Eclipse IDE in order to demonstrate the relationship between the different classes in an object-oriented program. Instead they can project the code from one class, such as the *Animal* class on one screen, and the code from another class, such as the *Dog*, *Cat* or *Horse* class on the second screen. Allowing the students to simultaneously view code from two classes in an object-oriented software program will encourage them to establish a meaningful cognitive relationship between the different classes in the program.

In total Queen's University has over 200 teaching rooms and lecture theaters fitted with data projectors, only three of which are fitted with dual screen technology. Clearly, the dual screen setup is the exception rather than the rule.

Furthermore, while this technical presentation setup is rare in tertiary education, it is all but unheard of in a primary or secondary education setting. It is obvious that we are in need a low-tech solution that will allow us to offer the same learning experience to students in primary, secondary and tertiary education.

Kinesthetic Leaning in Programming

The development of suitable kinesthetic-based teaching approaches in preference to the traditional 'talk and chalk' teaching styles adopted by many schools has long been an area of interest (Bonwell, C. C., & Eison, J. A. 1991; Wolfman, S. A., & Bates, R. A. 2005). More recently, kinesthetic-based approaches suitable for the teaching of computer programming have received attention (Pollard, S., & Duvall, R. C. 2006). These are normally cast as a game-based interaction, where students are taught that writing a computer program is really a problem-solving exercise. However, more extreme approaches have also been trialed. In one such approach Poon (2000) proposes that the use of physical props in teaching programming language is beneficial to students in constructing mental models of the abstract programming concepts. In another approach Fleury (1997) goes a step further and advocates the physical participation of students where each person acts out a part of a computer program. For example, a number of students move round a round physically enacting the process of traversing a Binary Search Tree.

Our Solution

We developed our solution in an attempt to solve some of the difficulties of teaching programming to novice programmers. We appreciate that programming is a practical subject and our approach accordingly makes extensive use of kinesthetic-based teaching approaches: a core part of the learning process is based on a practical activity. We also ask students to work in groups. This has the added benefit that it develops team-based relationships and promotes peer learning and problem solving. These skills are as highly prized in industry as they are in education.

We describe our approach, which is called *bigCode*, thus. Novice programmers are provided with numerous printed, poster-sized fragments of program code. Together the fragments contain the code necessary to complete an object-oriented software program, usually containing five or six classes. The programmers are also given a small class diagram for the program, similar to that shown in Figure 1. The programmers sit on the floor in groups and assemble these fragments into the classes which make-up the program. They must ensure that each fragment is placed in the correct class file and that the fragments in each class file are presented in the correct order. As shown in Figure 3, the fragments are provided in a jumbled state, but *bigCode* only contains the fragments required to create a working program, it does not contain any false fragments or distractors. *bigCode* is available for both *Java* and

Author Name: Neil Anderson
Contact Email: n.anderson@qub.ac.uk

C#, both of which are object-oriented languages.

Benefits of *bigCode*

The kinesthetic process of arranging the fragments into completed code helps the novice programmers develop a number of different skills crucial to good programming practices as well as skills such a effective team working and peer programming which are both crucial for success in the software industry.

Learning to Recognise Good Code

Due to the screen real-estate constraints of an IDE it is often very difficult to have all of the code in a class visible on screen at once. All IDEs, of course, have vertical and horizontal scroll controls, but even so it is difficult for a novice programmer to become familiar with the appearance of a correctly formatted and fully functional computer program.

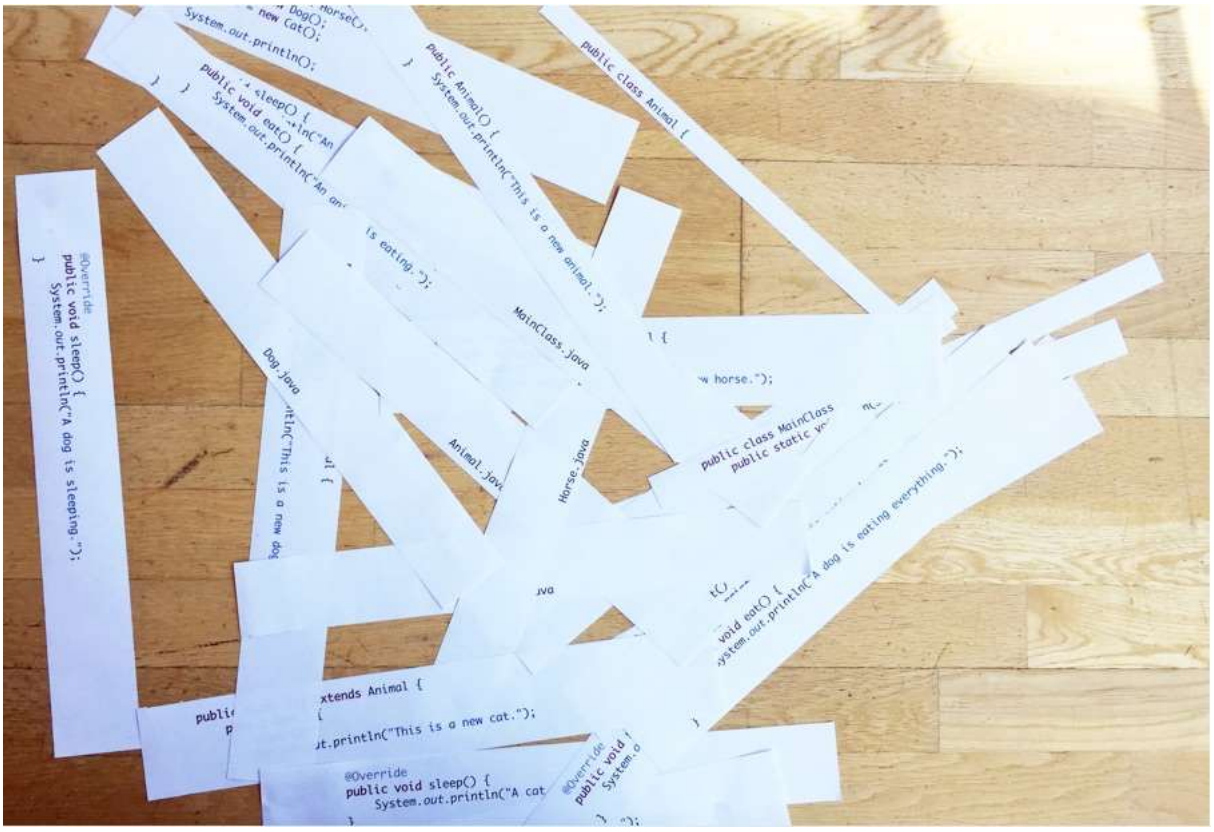

Figure 3. The many code fragments of *bigCode.*

Author Name: Neil Anderson
Contact Email: n.anderson@qub.ac.uk

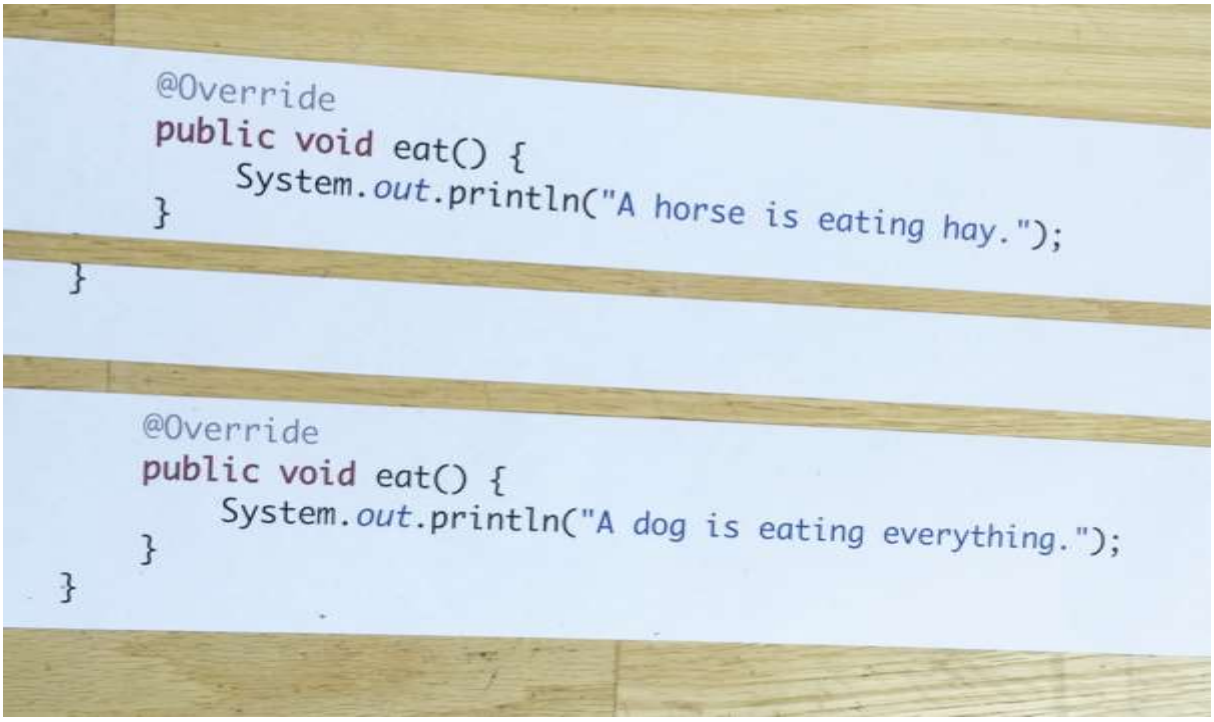

Figure 4. Alternative slicing strategies for *bigCode* fragments.

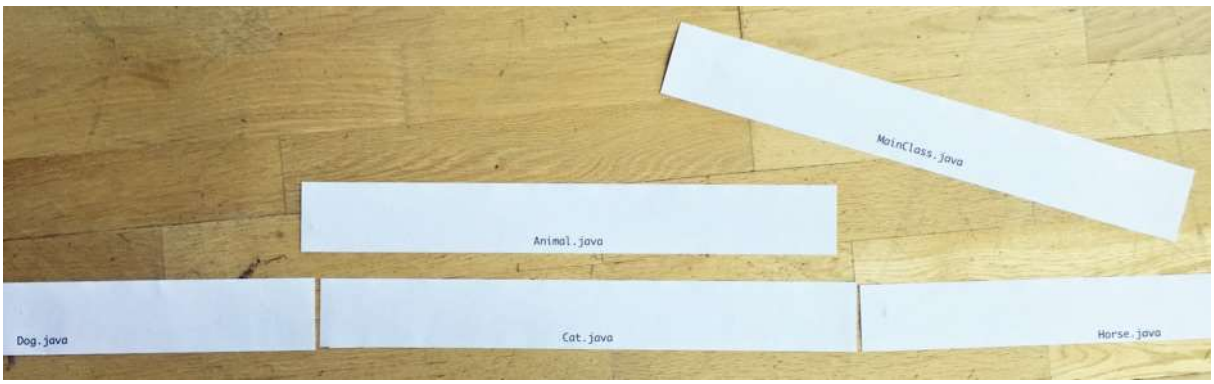

Figure 5. Class file headers arranged to mimic the program hierarchy.

Author Name: Neil Anderson
Contact Email: n.anderson@qub.ac.uk

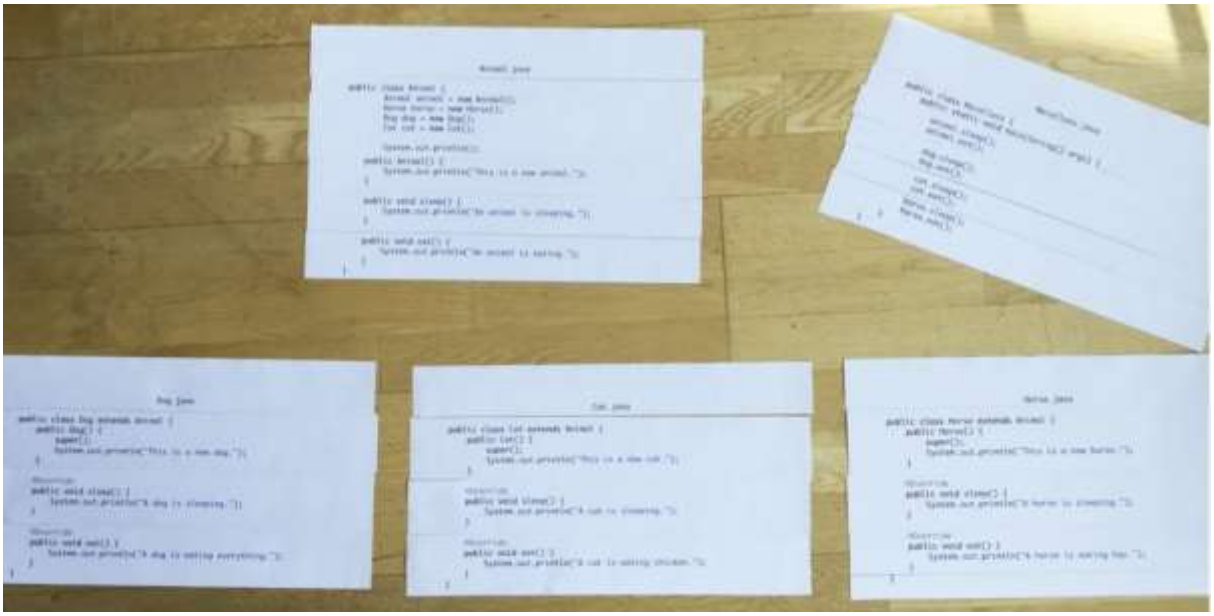

Figure 6. bigCode fragments arranged into completed code.

By piecing together the code fragments provided for each class, *bigCode* allows a novice programmer to see the all of the source code of a class in its entirety. During the process of putting the code fragments together, students learn to recognise the silhouette of a correctly laid-out and fully-functioning computer program.

For example, the code fragments are cut from the code in each class in an inconsistent manner. This approach is a deliberate to force the students to use their problem solving skills when rebuilding the fragments into completed program code. For example, as shown in Figure 4, the *eat()* method is contained in a single fragment, but this fragment also includes the closing curly brace '}' for the Cat class. In this case, the student must place this fragment at the bottom of the program code for the Cat class. Failure to do so will mean that the class structure will be incomplete as the closing curly brace for the class will be in the wrong place.

Contrast the Cat class to that of the Horse class. As shown in the upper two fragments in Figure 4, the corresponding fragment containing the *eat()* method for a horse does not include the closing curly brace for the Horse class. In the Horse class the student can safely put code fragments containing the *eat()* and *sleep()* in any order in the class, as long as they are placed above the fragment containing the closing curly brace for the class. However, as shown in the lower fragment in Figure 4, the *eat()* method for a dog does include the closing curly brace for the class. This fragment, therefore, must be placed at the bottom of the dog class.

Working through this process will give the students great experience of the layout and structure of computer programs. This will, in turn, save them a vast amount of time when working in an IDE. Students that have worked with *bigCode* will know instinctively when the formatting of a program is wrong. In short, they will know 'just by looking' at the shape of the code, if a computer program is wrong. Clearly, *bigCode* is a tool, which benefits those with a preference for visual learning approaches as well as those with a preference for a kinesthetic-based leaning approaches.

Author Name: Neil Anderson
Contact Email: n.anderson@qub.ac.uk

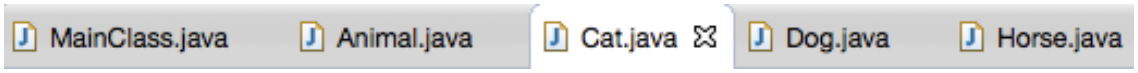

Figure 7. Tabs for each class in the Eclipse IDE

Class Relationships in OOP

Figure 1 depicts the hierarchal arrangement of a relatively simple object-oriented program with inheritance. However, this basic program with its simple hierarchy is, when implemented, made up of five different class files. This means that in their IDE, a novice programmer must keep track of five different files, each one containing the code for a different class. As shown in Figure 7, the five class files are arranged, in a linear fashion, as five tabs across the top of the screen. This is a sensible and practical arrangement for experienced programmers, however, for a novice programmer it presents a number of difficulties. First, by design, the code from only one tab can be shown at once. This means that a novice programmer must 'hop' between each of the five tabs in order to read all of the code. Second, the liner organisation of the tabs does not mirror the hierarchical organisation of the program. For a novice programmer, these two factors promote a disconnection between the organisational theory of OOP and the development of the code in an IDE.

bigCode, on the other hand, was developed to deliberately promote a cognitive connection between the organisation of the code fragments into class files and the hierarchical organization of an object-oriented program. For example, as shown in Figure 5, a student has deliberately setout the header fragments from each class file in a manner that replicates physically the hierarchal organisation of the program.

At present, *bigCode* is entirely paper-based, and as such does not offer any facility for automatic feedback or student self-assessment. However, tasks such as the organisation of fragments into classes is ideally suited to a digital environment. Students could organise the fragments on-screen and received targeted and meaningful feedback when they make a mistake and again when they complete the task. This is an area for further development.

Inspiration from Mixed Modality Presentations

Moreno and Mayer (1999) describe a style of teaching presentation, which they term as 'mixed modality'. This style of presentation can be employed when giving a multimedia explanation or live demonstration (such as a live code demonstration).

According to the modality principle, words should be presented auditorily rather than visually. In the content of a presentation, this means that words should be presented as an auditory narration rather than presented as text on screen.

They found this approach to be superior to process of presenting both the media that is to be explained on screen alongside text that is intended to explain it.

Further, they found mixed modality to be consistent with Paivio's (1986) theory that "when learners can concurrently hold words in auditory working memory and pictures in visual working memory, they are better able to devote attentional resources to building connections between them."

We drew inspiration from the modality principle and extended *bigCode* such that the process includes a 'Code Walkthrough' stage. A Code Walkthrough (IEEE, 1993), which is a process used extensively in the software development industry, is a form of peer review in which, "a designer or programmer leads members of the development team and other interested parties through a software product, and the participants ask questions and make comments about possible errors, violation of development standards, and other problems"

Author Name: Neil Anderson
Contact Email: n.anderson@qub.ac.uk

Once a group of students has finished arranging the *bigCode* fragments into a complete program one student from the group is asked to lead a walkthrough of the code. The remaining students in the group act as members of the development team: they ask questions and makes comments about possible errors. The students are expected to explain to each other how the object-oriented relationships between the different classes operate. This process promotes peer-learning and is a valuable final step in the *bigCode* process as it also allows students to develop peer-programming skills which they will use extensively in industry.

Form-Factor Considerations

Great care was taken to establish the perfect form factor for *bigCode.* A number of different variants were trialed with students, each with varying degrees of success.

Initial Prototype

The first prototype of *bigCode* was produced using A0 paper. However, this meant that the code fragments, when printed in landscape, we almost 1.2 meters wide. This made *bigCode* almost impossible for the learners to work with. In fact, an area equivalent in size to one quarter of a tennis court was required simply to allow a group of five people adequate space to set out code for a multi-class C# program.

A revised prototype of *bigCode* was produced at A2 size instead. In this revised version the code fragments were much more manageable, and the project required much less floor space. Crucially, however, A2 paper is still large enough to provide the learners with the 'wow-factor' when *bigCode* is first presented to them.

Figure 8. bigCode fragments contained in their presentation wallet.

Author Name: Neil Anderson
Contact Email: n.anderson@qub.ac.uk

Smaller Version

Reducing the size of *bigCode* has some clear benefits. First, the code fragments are easier to handle and considerably cheaper to produce. Most schools and universities already have access to A3 printers, which means no specialist equipment is required and therefore a smaller version of *bigCode* can be produced in-house (the A2 version of *bigCode* is simply scaled to fit A3 by the printing software). It is also much easier to distribute and store the code fragments in A3 format as they easily fit into a presentation wallet, as shown in Figure 8.

Moreover, smaller code fragments can also be used in more restricted settings. For instance, code fragments produced from A4, or even A5 paper, can be used in a classroom or lecture theatre environment. This allows tutors to introduce much sought after interactions into existing theory-based teaching sessions. Clearly some impact will be lost with a reduction in fragment size; however, in a given scenario this may be more than made up for by the flexibility offered by a smaller form factor.

Conclusion

Students' comments indicate that they like the process of setting out the fragments of code into a completed object-oriented program. The vast majority of the comments centre on the opportunity that they get to actually get to see all of the code, from many different classes, at once. A substantial number of students went on to comment that the process of sorting the code fragments into different classes allowed them to develop an appreciation, in their mind, of how the relationships between the different classes in an object-oriented program really work. Finally, a number of students also commented that they had learnt the importance of 'clarity in coding' when writing a computer program. This is a technique where-by programmers for the structural appearance of their code using white space, tabbed spaces and new lines to clearly and neatly layout the code in their programs. All of the code contained in the *bigCode* fragments has been presented in a clean and clear manner, such that the silhouette of the code in the correct program has a distinctive shape and clarity. The students that commented on the importance of code clarity all directly attributed their appreciation of this style of development to the appearance of the code silhouette in *bigCode.*

Our plans for future work fall into two distinct strands. First, we intend to undertake an empirical study such that we can fully understand the full pedagogic impact that *bigCode* has on computing students in secondary education. Second, we also intent to extend *bigCode* by developing a digital edition, allowing us to provide *bigCode* to a much greater number of students.

Measuring Pedagogic Impact

In the forthcoming academic year we will undertake an empirical study to measure the effectiveness of *bigCode* as a teaching approach. In this study, we will create two groups of novice computer programmers; both groups will have members, which are of a similar age, background and programming ability. Each group will have a similar number of members and a similar gender balance. We will use both groups of novice programmers to establish a baseline understanding of the abilities of each group to produce object-oriented code without any formal training.

We then teach object-oriented programming to the first group using the *bigCode* technique. We use a traditional lecture-based technique to teach object-oriented programming to teach the second group. This study seeks to establish if the first group, having been taught using the *bigCode* technique, shows an enhanced level of ability to produce object-oriented code when compared to the control group.

Author Name: Neil Anderson
Contact Email: n.anderson@qub.ac.uk

bigCode Digital Edition

Our intention is to create a digital edition of *bigCode*. It is anticipated that this edition will make use of the HTML 5 canvas element, which is a container for graphics, allowing them to be manipulated and redrawn on the fly using JavaScript drag-and-drop controls. HTML 5 will also allow us to provide *bigCode* on the widest possible range of platforms and devices. For instance, the application will work equally well on touch screen devices, such as the iPad and Android tablets, as well as on an interactive white-board in a school classroom.

Furthermore, the digital edition will allow us to provide automatic contextual feedback. If the user places the code fragment in the correct class, but in the wrong place in that class, then the will get a different feedback massage than if they has placed the code fragment into the wrong class.

Finally, our intention is to develop *bigCode* Digital Edition in such a manner that it can be delivered as a multi-screen tablet-based application, with full communication between each of the screens during operation. This will allow the students to completely mimic the experience of the paper-based *bigCode* which will be the true test of the success of *bigCode* Digital Edition.

Author Name: Neil Anderson
Contact Email: n.anderson@qub.ac.uk

References

A. E. Fleury (1997). Acting out algorithms: how and why it works. *The Journal of Computing in Small Colleges*, 13(2):83–90. doi: 10.1080/08993400500056563

Bonwell, C. C., & Eison, J. A. (1991). Active Learning: Creating Excitement in the Classroom. ASHE-ERIC Higher Education Reports. *ERIC Clearinghouse on Higher Education*, Washington, DC.

Coffield, F., Moseley, D., Hall, E., & Ecclestone, K. (2004). *Learning styles and pedagogy in post 16 learning: a systematic and critical review*. The Learning and Skills Research Centre.

Holliman, R., & Scanlon, E. (2013). *Mediating science learning through information and communications technology*. London, UK: Routledge.

IEEE (1993). *IEEE Standard for Software Reviews.* Retrieved from: https://standards.ieee.org/findstds/standard/1028-1988.html

Kölling, M. The Problem of Teaching Object-Oriented Programming, Part 1: Languages. *Journal of Object-Oriented Programming*, 11(8): 8-15. doi:10.1.1.39.8492

KPMG. (2015). Tech Monitor. Retrieved from: https://www.kpmg.com/UK/en/IssuesAndInsights/ArticlesPublications/Pages/techmonitoruk.aspx

Moreno, R., & Mayer, R. E. (1999). Cognitive principles of multimedia learning: The role of modality and contiguity. *Journal of educational psychology*, 91(2), 358. doi:10.1.1.458.4719

McGowan, A., Hanna, P., & Anderson, N. (in press). How Video Lecture Capture affects Student Engagement in a University Computer Programming Course: Attendance, Video Viewing Behaviours and Student Attitudes. *Paper presented at The European Conference on Educational Research 2015*, Budapest.

Ofqual (2015). Reform of AS and A level qualifications. Retrieved from: https://www.gov.uk/government/collections/reform-of-as-and-a-level-qualifications-by-ofqual

Paivio, A. (1986). Mental representations: A dual coding approach. *Oxford Psychology Series, 9.*

Pollard, S., & Duvall, R. C. (2006). Everything I needed to know about teaching I learned in kindergarten: bringing elementary education techniques to undergraduate computer science classes. *ACM SIGCSE Bulletin* (Vol. 38, No. 1, pp. 224-228). doi:10.1145/1124706.1121411

Poon, J. (2000). Java meets teletubbies: an interaction between program codes and physical props. In *Proceedings of the Australasian conference on Computing education* (pp. 195-202). Retrieved from: https://royalsociety.org/~/media/education/computing-in-schools/2012-01-12-computing-in-schools.pdf

Robins et al. (2003). Learning and teaching programming: A review and discussion. *Computer Science Education.* 13(2), 137-172. doi:10.1076/csed.13.2.137.14200

Rubin, M. J. (2013). The effectiveness of live-coding to teach introductory programming. In *Proceeding of the 44th ACM technical symposium on Computer science education* (pp. 651-656).

Author Name: Neil Anderson
Contact Email: n.anderson@qub.ac.uk

Samuel, S. (2015). Teaching Programming Subjects with Emphasis on Programming Paradigms. *2014 International Conference on Advances in Education Technology (ICEAT-14)*. Atlantis Press.

The Royal Society. (2012). Shut down or restart? The way forward for computing in UK schools.

Wolfman, S. A., & Bates, R. A. (2005). Kinesthetic learning in the classroom. *Journal of Computing Sciences in Colleges*, *21*(1), 203-206.