Teaching in a Software Design Studio: Implications for Modeling Education

Jon Whittle, Christopher N. Bull, Jaejoon Lee and Gerald Kotonya

School of Computing and Communications, InfoLab21, Lancaster University, Lancaster UK
{j.n.whittle, c.bull, j.lee3, g.kotonya}@lancaster.ac.uk

Abstract. This paper reflects on Lancaster University's software design studio, an approach for teaching software engineering that favours practical experimentation over traditional lectures. The studio has been running for two years at Lancaster. In the paper, we reflect on how students have used modeling as part of the studio experience. Our findings show that, given the freedom, students make heavy use of informal modeling but almost never formally model. The paper unpicks the reasons for this and discusses implications for the way modeling is taught more generally.

1 Introduction

Studio-based education is an alternative to traditional lecture-based instruction. It emphasizes reflective practice [1] as a way to develop design skills. In a studio, students are asked to reflect critically on their own and others' designs as a way to absorb design expertise. A variety of methods are used for this, such as coaching, design critiques, peer mentoring, and juried assessment. A studio course usually, but not always, is taught in a room dedicated for the purpose, i.e., the studio. The physical studio is seen as a key element because it allows students to permanently display their work, which encourages reflective practice over time. The studio also encourages frequent and informal interactions among students, which leads to peer-learning.

Historically, studios can be traced back to Medieval times when guilds would take on apprentices and learning would be "on the job" [1]. The formalized concept of a studio, as commonly understood in building architecture, for example, is closely linked to two schools, the École des Beaux-Arts and Bauhaus [2]. More recently, there have been a number of attempts to transfer the principles of studio education to software engineering (e.g., [3, 4]). Proponents of this argue that it leads to greater student engagement and satisfaction, a deeper understanding of design skills, and a stronger understanding of real-world issues. In particular, it is argued that 'design' is a tacit skill "that can't be learnt from a book" [5] but must be learnt 'by doing'.

In this paper, we reflect on our own experience of introducing a software studio at Lancaster University. The studio was instigated in 2012 and has been running continuously since. A physical, dedicated lab is made available to 2nd and 3rd year under-

graduate students. Students follow three studio-based courses in which they work on group-based projects. A traditional lecture-based course is taught in parallel to the first of these and teaches classical software design. The project work covers all phases of the software lifecycle from ideation to evaluation with real stakeholders. We have experimented with different ways of teaching design in the studio – ranging from a phased iterative/incremental approach to a sprint-based agile approach. In all cases, studio work focused on project development and avoided teaching any specific skills. These were assumed to have been either taught in other courses, or the students were expected to learn these skills independently on-demand. Students were, however, expected to follow good software engineering principles of requirements engineering, architecture, design and testing. However, the studio was not prescriptive in terms of what documentation students had to produce: the basic guidance was that a system should work, be well designed, and any documentation should clearly and concisely explain what is going on. Students were not mandated, for example, to produce UML diagrams as part of the documentation. They were simply told to use whatever form of documentation was most effective for them. As a result, the studio offers an interesting opportunity to explore what happens when students are not explicitly taught software modeling and design skills but instead pick up these skills through continuous and informal coaching and critique.

We have evaluated our studio style of teaching through a one and a half year observational study. A researcher (the second author) attended studio sessions and took field notes describing how students interacted with each other. This was supplemented with a focus group with students discussing the style of education, and how the studio courses compare to the rest of the degree. The students also frequently offered opinions and perspectives before and during each studio session. These were recorded as part of the observations. In previous work, we have described how principles of reflective practice are instilled in the studio [6]. We have also previously presented an analysis of the observational data focusing on how Lancaster's studio implements the key defining principles of studio education [7, 8]. To date, however, we have not analyzed our data to see the role that models play in studio education.

This paper revisits the qualitative data collected in the observational study and uses it to reflect on the role that models play in the design processes of studio projects. This is an interesting question to ask because students were given free rein as how to document their designs: therefore, this paper sheds some light on how useful the students found models in practice.

Our findings indicate that formal models are used rarely by the studio students, whereas informal models are relied on heavily. Furthermore, the pervasiveness of the informal models appears to be important: that is, the physical studio allows students to display their informal models, which, although they are generally not updated as the design changes, act as a visible reminder as to how design decisions were arrived at and therefore continue to serve as a point of reference. Based on these observations, we reflect on how software modeling is taught and make some recommendations for better modeling tools and educational approaches to modeling.

2. Software Studios

Interpretations of studio education for computer science and software engineering have been explored for over 20 years, with the earliest known computer science studio implemented at Carnegie Mellon in 1990 [3]. Over this time there have only been a relatively small number of software studios, yet project-based learning (PBL) has seen wider use. Studios share several similarities with PBL, but PBL does not determine how students should interact: for example, the culture of critique or the collocation of students in a studio is not necessarily part of PBL. Studios are often project-based but a studio is more than just a project-based course.

Carter et al. have recently presented an overview of five studios in computer science [9]. Software design studio implementations often report varying levels of success, but it has been indicated that, in certain studios, the student's content mastery is "as much or more than students in the same courses taught in the traditional way" and also that "students in studio courses have shown higher levels of motivation and engagement than students in traditional courses" [10].

Despite the apparent success of the limited number of documented studios, it is also apparent that none of them follow a shared definition of studio education – making comparisons difficult. It is challenging to succinctly define studio practice due to the inherently complex nature of studio education. As such, previous work has provided a 'studio framework' to help solidify a definition and understanding of studio practices [8]. Due to their complex nature, studios should not be considered a binary state (i.e., whether a course is a studio or not). Rather, there will be some courses which are more 'studio-like' than others.

Our framework defines ten key elements of a studio approach. Each of these defines a characteristic that is typical of a studio. However, it is recognized that some studios will focus on some characteristics at the expense of others; a studio can be a studio without having all of these traits. The first two columns of Table 1 describe a sample of these characteristics.

2.1 Lancaster's Studio

The first year of Lancaster's BSc Software Engineering degree is identical to Lancaster's BSc in Computer Science and does not use the studio model. In the second and third years, students take three studio courses. In the first of these, students work in groups of 4 and come up with their own ideas of systems to design and build. In the other courses, students work in larger groups (6 students) on progressively larger and more challenging problems, with industrial involvement in the final course. The studio modules have no lectures, and are 100% coursework-based assessment.

To give a flavour of Lancaster's studio, Table 1 compares the studio with the project-based course which was its predecessor. Based on our data analysis over a two year period and on student feedback, the studio approach appears to lead to greater student engagement as well as improved learning. For example, student feedback in official evaluations includes:

Table 1. Comparison of Project-Based and Studio-Based Courses at Lancaster University.

Aspect	**How the Aspect Manifests in a Studio Course**	**Typical Project Course**	**Lancaster's Studio Course**
Physical environment	There is a physical room (i.e., the studio) which is open and reconfigurable providing a variety of group, individual and social spaces.	Standard lab	Dedicated lab with 24 hour access, maintained by students themselves
Management of Studio	Rules regarding use of the space should not be restrictive.	Lab tightly controlled by University	24 hour access; food/drink allowed; students have admin rights
Modes of Education	Teaching staff play a coaching/mentoring role rather than being didactic.	Students given a prescriptive list of documentation to produce	Students were told to produce "as much documentation as needed". There was no prescription on what kinds of diagrams or notations to use.
Awareness	Placing work on display (as works-in-progress or final products). Visibility of work helps students see each other's work.	No special consideration	Students used mobile whiteboards in the studio to display design work, which was left up for the duration of the project
Critique	Ongoing critique is used for providing feedback and developing ideas. It should take place in multiple ways (formal and informal, group and individual, peer and staff)	Provided only as part of weekly meetings with project supervisor	Provided on a continuous basis using a variety of methods: individual and group demos/presentations, informal coaching, peer critique, critique from external assessors (e.g., companies) and formal judging.
Culture	A studio culture should be social and foster sharing, and yet should be sensitive to supporting a good work ethic.	Lack of a dedicated lab meant that students typically only met at prearranged times	Students used the studio as a home, leading to serendipitous interactions and a feeling of 'belonging' to a cohort
Inspiration	When designing, students should be encouraged to be creative in their designs and solutions.	Project specification decided a priori by academic staff	Students come up with their own project ideas in a facilitated creativity brainstorming session

"The best part […] was that we got to learn first hand from our own mistakes"
"Having a dedicated studio for our own use gave me a quiet place where I could focus on my studies"
"Help was available from staff when needed. Freedom to apply our own creative ideas into the project rather than follow strict guidelines of what we can do."
"The most valuable parts of the module were the open nature of the module giving you the freedom to stretch out and do what you want extending far from the basic requirements, the lecturers were always on hand to help"

Module evaluations have been very strong: students rated the quality of teaching on the three studio modules as good or very good (overall 88% very good) and the last studio course was given a perfect 5/5 overall score from 100% of students. The studio approach won a Pilkington Teaching Award in 2014.

3 Models in Studio Education

Our studio enables us to explore an interesting question. Given that our students had been formally trained in software modeling (through the parallel lecture-based software design course), but they were not mandated to use formal modeling notations in the studio projects, would the students decide for themselves to use models? If they did, what benefits did they gain from modeling? And if not, why not? In essence, this is a kind of "vote with your feet" approach to modeling: students had already been given the skills to model, so the big question is whether they would *choose* to model on a project comparable in size and complexity to a real-world problem for a small project team. This section unpicks what we have learned about this question.

3.1 Formal Modeling

Our first finding is that students did very little, if any, formal modeling. By formal modeling, we mean the use of a formalized modeling notation, such as UML, where the notation was used in a precise manner using the correct syntax. This need not be in a modeling tool. In contrast, we will use the term informal modeling to describe the use of either an ad-hoc notation (e.g., drawing shapes on a whiteboard, pen and paper, or a digital sketching tool) or a loose interpretation of a recognized modeling language (e.g., drawing a UML class diagram but without caring about correct syntax).

In general, we saw many instances where students used formal modeling notations such as UML use cases and class diagrams, but used them informally without caring about syntax and without using a formal modeling tool. Perhaps surprisingly, we saw very few cases where these informal models were converted into formal models (e.g., using a UML tool) so that they could serve as formal documentation. There were some cases where students tried to do this, but they quickly gave up because the process of transcribing the model into a formal notation began to impede discussion.

In contrast, informal modeling was used widely and often. In Lancaster's studio, this was facilitated by a number of mobile whiteboards. The students used these extensively in the studio for brainstorming designs, to collaboratively make design deci-

sions, and as a way of providing context in group discussions – students often had discussions around "their" whiteboard and, although they might not directly interact with the models on the whiteboard, the models appeared to serve the purpose of an implicit reminder or contextual cues for their ongoing deliberations.

The students were reluctant to transfer the informal models from the whiteboards into a formal modeling tool. There appear to be a number of reasons for this:

1. It was not required by the course. There were no explicit marks for producing UML diagrams. Rather, the students were graded according to whether their documentation was "appropriate and understandable." In practice, this meant that students often chose other ways to document their designs than UML.
2. Students interacted mainly through small group discussions. Computers seemed to inhibit fruitful discussions and so were often not used. Standard size screens are not easily visible to all members in a small group discussion. In some cases, one group member would act as the "scribe" and document UML models formally during the discussion. However, this practice subsided over time as it made it difficult for the scribe to fully participate.
3. Interestingly, some groups used a computer screen to display information relevant to the task at hand (e.g., slides from a lecture). This meant that the computer was not available for creating models. In some cases, students compensated for this by using laptops to create material, whereas the desktop computer was used to consume material.
4. Formal models served no purpose for the students. As this was not a course on model-driven engineering, there was no need to produce formal models for code generation. Students manually coded things and, as long as they had a shared understanding of the system, the informal models were sufficient.

As an aside, our experiences in the Lancaster studio may provide some insights for those working on so-called 'flexible modeling tools' [11]. A recent trend in modeling tool research is for tools that allow the flexibility of informal sketching but then provide seamless transitions to more formal models within the same tool. FlexiSketch, for example, is a tool for creating domain-specific modeling languages on-the-fly [12]. Similarly, Calico [13], a collaborative sketching tool, has some capabilities for executing sketch models (e.g., a sketched state machine). Whilst these tools are interesting, our experience in the studio seems to suggest that there is not always a need to translate informal models into formal ones. So the use cases for such tools may be limited. This observation agrees with work by Petre, which shows that professional software designers make heavy use of informal models, including 'ephemeral' models which developers dispose of quickly [14]. In a separate study, Petre found limited uptake of formal UML modeling [15].

The lack of formal models in the Lancaster studio could mean a number of things. One interpretation is that formal models just are not useful to the students; they can accomplish what they need to accomplish without formal models.

Another interpretation is that since formal models were not mandated (and assessed) in the course, students simply ignored them. This might suggest that the studio courses should give credit for (e.g.) formal UML models and that, by doing so, the

students would produce better software. However, it is debatable whether formal UML models lead to better software. Either way, *forcing* students to produce UML models goes against the ethos of the studio. In the studio, students learn largely through reflective practice. They are constantly encouraged to think about their design decisions and their designs are constantly critiqued both by peers and teaching staff. There is an open question whether formal modeling could or does encourage reflective practice. On the one hand, one could argue that formalization forces students to think about the ramifications of their design. On the other, formalization could act as a barrier because reflective practice relies on constant iterations of a design, which is not necessarily easy or well-supported with current model tools.

A third possible interpretation is the small group size – perhaps a larger group would have had greater need to formalize the models. Whilst this may be true, we feel that introducing formal models explicitly in the studio would therefore be a bad idea: it would be introducing a method for large groups into a small group context.

3.2 Informal Modeling

Whilst students in the studio did little formal modeling, they regularly came up with informal models. These included both UML models (typically, use cases and class diagrams), sketches (e.g., a high-level architecture, UI design sketches), and process-based models (e.g., burn-down charts as used in agile development).

There are two aspects of this informal modeling which are worth comment. Firstly, the students essentially modeled "on demand". That is, they did as little modeling as they needed, and only when they needed it. This is very much an "agile" way of working (cf. Ambler's agile modeling, for example [16]). The students did not attempt to produce fully worked out models for a feature they were about to build. Models were mainly used for brainstorming and for thinking through a design. Once the design was thought through enough to allow the team to move on to the next stage of development, the models were left as they were, and no further detail was added. This is not to say, however, that the models were then discarded. Quite the contrary. As previously mentioned, students made heavy use of mobile whiteboards for informal modeling. The second aspect worth noting is that the whiteboards and the use of a dedicated space, which meant that the students could happily leave design work on the whiteboards permanently, meant that the informal models became a pervasive artefact. Rather than erasing the models once design decisions had been taken, the models were left on the whiteboards for weeks or months, even when the models had outlasted their purpose. It seems the models came to serve a contextual purpose – students tended to use the models either as quick reminders of what they were doing, or as background 'noise' which somehow helped them to work as a group.

3.3 On the Value of Modeling

Despite the lack of formal models in the studio courses, there is evidence that the students have learned a lot about modeling and, in particular, have come to appreciate the value of modeling through the studio course. Indeed, it appears that the students

like (informal) modeling because in the studio it is a highly collaborative and creative activity. One of the problems that we have experienced in teaching modeling in more traditional lecture-based courses is that students often do not like or engage with modeling. They don't easily see the point of modeling, since they want to build code that runs. And in a traditional top-down approach to teaching modeling – where requirements are taught, then architecture, then low-level design, then coding – the models produced are often low quality because the students don't yet understand enough about the system they are building to properly model it.

In our studio courses, we have seen evidence that students gain a greater appreciation for modeling than in traditional lecture-based courses. Our students are given free rein as to what and when they should model. Hence, they tend to model when they get some value from it. As a result, they appreciate that models do have value. Students have told us, for example, that the studio course gave them a better understanding of the motivations for modeling. This appeared to be particularly true for behavioural models.

4 Discussion

A year and a half long observation of Lancaster's software studio has revealed that, in this style of education, students make little use of formal modeling notations but heavy use of informal model sketches. Since there is strong evidence that the studio model is an effective way of teaching software engineering, we argue that these observations have implications for the way modeling is currently taught as well as how it might be taught differently. We discuss these implications below.

A typical approach is to teach UML as part of a software engineering course. In this approach, software engineering is taught in a top-down fashion. Students are first taught about the importance of software requirements, and then how to refine those requirements into a software architecture, a low-level design, and finally an implementation. UML is often used as the lingua franca across these lifecycle phases and heavy emphasis is placed on the students producing high quality, syntactically and semantically correct UML diagrams.

Our experience in the studio questions whether this is the best way to teach modeling, and software engineering more broadly. In our experience, we have noted that students do not engage well with this kind of approach. This is largely because they cannot immediately appreciate the benefits of this kind of more formal modeling. Formal modeling means that a working system is delayed and the benefits of formal modeling are not necessarily experienced in a typical software engineering course, which does not include the important activities of software maintenance and evolution. In the studio, in contrast, the educational emphasis is on reflective practice; students learn by doing and by reflecting on what they have done. Reflective practice can of course be applied to teach formal modeling. However, the purpose of the studio was not to teach formal modeling; rather, it was to teach software engineering. In this context, reflective practice naturally focuses on those activities which are most critical to teach software engineering. Since these activities were not necessarily defined a-

priori, but were introduced on-demand as and when they were needed by the students, it is interesting to note that formal modeling was generally not needed. At best, this suggests that formal models are not necessarily appropriate in small group projects. At worst, it questions the value of formal models more generally.

In either case, the studio approach suggests an alternative way of teaching modeling. Largely, there is a tendency within software engineering education to teach "the right way" to do things and then grading the students on whether they have done things the "right way". This is fine if the learning objective is about teaching a specific skill-set. However, it is problematic when teaching in project-based courses, such as a studio, because there is considerable controversy over what is the "right way". Some argue for formal modeling, for example, whereas others argue for agile methods. The danger of choosing a particular method and then teaching this method in the "right way" in a project course is that the students will not see the benefit of that particular method and so will come away disillusioned about the method.

Therefore, we argue for a different approach. We believe that students should be given exposure to a range of methods, apply these methods in practice, and through reflective practice, be encouraged to learn about the positives and negatives of these approaches for themselves. Indeed, one could argue that the most important thing to teach is the culture of reflective practice itself. If students learn how to reflect, they will become reflective practitioners and can apply those skills to any new method or approach which they are faced with in their future careers.

We maintain therefore that, as a community, we might consider de-emphasizing teaching the ins and outs of UML notation and be less dogmatic about teaching a step-by-step approach to modeling. Rather, let the students discover modeling for themselves. Give them projects to work on and encourage the use of models where it makes sense; where it doesn't, don't force it. Use techniques of coaching and critique so that when modeling does appear, students can be coached in creating useful and usable models. And use coaching and similar methods to get students to think beyond simple model sketches and ask them if they would benefit from more formal models. If they would, encourage them. However, crucially, don't force the students to produce formal models where there is no benefit in doing so. Also, let students make mistakes. If they don't see a benefit in doing things in a particular way, don't mandate it, but find ways of letting students fail gracefully and then discussing with them afterwards how the "right way" might have avoided those mistakes. This is clearly easier to do in small project courses, like the studio, than in large classes.

Our studio experience also has implications for modeling tools. A lot has been written recently about the effectiveness and/or issues of software modeling tools (e.g., [17,18]). Less has been written, however, about modeling tools that support software engineering education. Alfert et al. argue that industrial modeling tools are not well suited to teaching modeling [19]. There is surprisingly little work on modeling tools to support student project teams. Exceptions include the work on Calico [12] and there has, of course, been a wealth of research on the collaborative aspects of software designers more generally (see [20] for a recent example). However, there is not much that focuses on the requirements of modeling tools to support project courses.

References

1. Schön, Donald A. (1983). *The reflective practitioner: How professionals think in action*. London: Temple Smith. ISBN: 9780465068784.
2. Salama, Ashraf (1995). *New Trends in Architectural Education: Designing the Design Studio*. Raleigh, NC, USA: Tailored Text. ISBN: 9780964795006.
3. Tomayko, James E. (1991). Teaching software development in a studio environment. In: *ACM SIGCSE Bulletin* 23(1), pp. 300–303.
4. Kuhn, Sarah (1998). The software design studio: An exploration. In: *IEEE Software* 15(2), pp. 65–71.
5. Broadfoot, Ouita and Rick Bennett (2003). Design Studios: Online? Comparing traditional face-to-face design studio education with modern Internet-based design studios. In: *Apple University Consortium*, pp. 1–13.
6. Bull, Christopher and Whittle, Jon (2014). Supporting Reflective Practice in Software Engineering Education through a Studio-Based Approach. *IEEE Software* 31(4): 44-50
7. Bull, Christopher and Whittle, Jon (2014). Observations of a Software Engineering Studio: Reflecting with the Studio Framework, *CSEE&T*.
8. Bull, Christopher, Whittle, Jon, and Cruickshank, Leon (2013). Studios in software engineering education: towards an evaluable model. *ICSE*, pp. 1063-1072
9. Carter, Adam S. and Christopher D. Hundhausen (2011). A review of studio-based learning in computer science. In: *Journal of Computing Sciences in Colleges* 27(1), pp. 105–111.
10. Narayanan, N. Hari et al. (2012). Transforming the CS classroom with studio-based learning. In: *Proceedings of the 43rd ACM technical symposium on Computer Science Education* (SIGCSE '12), pp. 165–166.
11. Ossher, Harold et al. (2011). ICSE 2011 Workshop on Flexible Modeling Tools, Honolulu, Hawaii.
12. Wüest, Dustin, Seyff, Norbert, and Glinz, Martin (2013). Semi-automatic Generation of Metamodels from Model Sketches. In: *IEEE/ACM International Conference on Automated Software Engineering*.
13. Mangano, Nicolas and van der Hoek, Andre (2012). The design and evaluation of a tool to support software designers at the whiteboard. In *Automated Software Engineering*, 19(4), pp. 381–421.
14. Petre, Marian (2009). Representations for Idea Capture in Early Software and Hardware Development, Technical Report No. 2009/12, Department of Computing, The Open University.
15. Petre, Marian (2013). UML in practice. *ICSE 2013*, pp.722-731
16. Ambler, Scott (2001). Agile Modeling: A Brief Overview. pUML 2001, pp. 7-11
17. Whittle, Jon, Hutchinson, John, Rouncefield, Mark, Burden, Håkan and Heldal, Rogardt (2013). Industrial Adoption of Model-Driven Engineering: Are the Tools Really the Problem? *MODELS 2013*, pp. 1-17
18. Paige, Richard and Varró, Dániel (2012). Lessons learned from building model-driven development tools. *Software and System Modeling* 11(4), pp. 527-539
19. Alfert, Klaus, Plemann, Jörg and Schröder, Jens (2004). Software Engineering Education Needs Adequate Modeling Tools. In *17th Conference on Software Engineering Education and Training (CSEET '04).* IEEE Computer Society, Washington, DC, USA, pp. 72-77.
20. Baker, Alex, van der Hoek, Andre, Ossher, Harold and Petre, Marian (2012). Guest Editors' Introduction: Studying Professional Software Design. *IEEE Software 29(1)*, pp. 28-33.