Innovations in Education and Teaching International,
Vol. 42, No. 1, February 2005, pp. 15–25

Group projects and the computer science curriculum

Mike Joy*
University of Warwick, UK

Group projects in computer science are normally delivered with reference to good software engineering practice. The discipline of software engineering is rapidly evolving, and the application of the latest 'agile techniques' to group projects causes a potential conflict with constraints imposed by regulating bodies on the computer science curriculum. This paper explores the issues, and argues that there are strong educational reasons for modifying the computer science curriculum to accommodate these new software engineering methodologies.

Introduction

The Universities and Colleges Admissions Service (UCAS, 2003) lists 3148 courses in computer science in the UK, at undergraduate degree level and sub-degree level (such as Higher National Diploma, HND). Of those, 496 are 'single subject' computer science. The content of these courses varies greatly, with some offering a business-oriented perspective on information technology (IT), whilst others emphasize the 'science' viewpoint on the subject. The subject matter taught in each of these courses is constantly and rapidly changing, in response both to technological advances in computing, and to the economic and social environment in which graduates will make their careers. All are subject to quality assurance processes, and some are accredited by professional bodies.

Most computing courses contain one or more group projects, motivated by practical and pedagogical considerations. The delivery of such projects involves the application of good software engineering practice, a topic within the computer science curriculum which is evolving rapidly at the time of writing. There are also curriculum constraints affecting group projects, which are imposed by current quality assurance practices and accreditation requirements, and which assume a model for software engineering which is now being questioned.

Using one such course—the Computer Science degree at Warwick University in the UK—as a case study, we consider how external constraints have affected group projects within the overall

*Department of Computer Science, University of Warwick, Coventry CV4 7AL, UK.
Email: M.S.Joy@warwick.ac.uk

ISSN 1470–3297 (print)/ISSN 1470–3300 (online)/05/010015–11

DOI: 10.1080/14703290500048788

curriculum. We present an overview of the latest developments in software engineering, and argue that there are strong educational reasons for modifying the curriculum to accommodate the new software engineering methodologies.

Group work and the learning experience

The use of group work in higher education has many benefits, and there is a substantial body of knowledge on its effective deployment (Henry, 1994; Reynolds, 1994; Gibbs, 1995; Jaques, 2000). Henry (1994) identifies five principal reasons for students to undertake group work:

- *Application of knowledge*—students are able to put into practice the knowledge and theory they have assimilated in previous modules.
- *Motivation*—a suitably chosen project is likely to be of direct relevance to the student.
- *Higher cognitive skills*—students develop a deep understanding of the material they are working on, and develop corresponding deep learning skills.
- *Autonomy*—students have control over what they learn and how they learn it.
- *Assessment*—projects are effective at distinguishing the strong students from the weaker ones.

Reynolds (1994) includes a further category:

- *Ideological*—students are prepared for participation in a society which promotes collaboration and participation.

In the context of computer science, there is a cultural factor. Hughes and Cotterell (2002, p. 217) note that: 'A problem with major software projects is that they always involve working in groups, and many people attracted to software development find this difficult'.

In the Professional Practice chapter of the Computer Science Curriculum authored jointly by the Association for Computing Machinery (ACM) and the Institute of Electrical and Electronics Engineers (IEEE) in the USA (ACM/IEEE, 2001, p. 59), it is noted that: 'Learning how to work in teams is not a natural process for many students, but it is nonetheless extremely important'.

Computing students often lack basic team-working skills, and we should therefore add a further reason to include group work within the computer science curriculum:

- *Team skills*—students should learn to work effectively as part of a team, rather than as lone programmers.

External requirements

In the industrial and commercial context that most students would encounter after graduating, computer programming is normally a group activity. The UK Government is committed to 'ensuring that the education and training system responds effectively to demand from employers' (DfES, 2003, p. 58), and team working is a transferable skill desired by employers in the IT sector. In recognition of this, the accreditation requirements of the professional bodies *require* the inclusion of group work within the curriculum. We consider course accreditation and quality assurance in the UK as factors which affect course design, and for comparison purposes we will also briefly consider accreditation in the USA.

Computer Science, and other Information Technology-related degree courses in the UK are regulated by the British Computer Society (BCS) or the Institute of Electrical Engineers (IEE), or both. These two organizations are nominated bodies of the Engineering Council (EC) that are allowed to award professional qualifications including chartered engineer status (CEng) to appropriately qualified members. The relationship between the EC and its nominated bodies has developed over time, but is essentially pyramidal in structure. The majority of the work of the EC is devolved to the individual organizations with the EC issuing binding guidance on limited aspects of an organization's processes. In the case of the BCS, one specific effect is that membership of the BCS is a prerequisite for CEng status, but not a sufficient one. Membership of the BCS is open to a wider pool of professionals than those who aspire to be chartered engineers.

In order for IT professionals to become members of the BCS (designated by the letters MBCS after their name), they must fulfil two prerequisites: they must have sufficient academic background, together with appropriate industrial or commercial experience. The BCS administers its own examinations, but the majority of members have a qualification which exempts them from the BCS examinations, and which is typically a three-year undergraduate honours degree. The process for applying for exemption is described in detail in the BCS guidelines (1998). The academic prerequisite for CEng is more onerous, and requires either four years of undergraduate study, or a three-year honours degree plus a one-year 'matching section', which is typically a one-year MSc.

The Quality Assurance Agency (QAA) has also been involved in the specification of standards in computing, and in 2000 the 'subject benchmark statement' for computing was published (QAA, 2000). The purpose of the document is to provide an outline of the curriculum expected of a computing honours degree, and its intended audience includes, in addition to quality assurance reviewers, potential students, external examiners and the individual academics within computing departments. When drafts of the benchmark were distributed during the consultation phase, they met with a high degree of acceptance, and there seems to be a high level of confidence within computing departments that its contents are accurate and representative.

In its preamble, the benchmark states that it 'provides general guidance for articulating the learning outcomes … but is not a specification of a detailed curriculum'. The extent to which the benchmark is in active use by individual departments in their course specification process is as yet unclear, but the QAA will use it in future quality assurance exercises, and the BCS has already incorporated reference to the QAA benchmark in its exemption and accreditation guidelines (BCS, 1998).

The system in the USA is somewhat different, and there are a number of national organizations involved in the accreditation process. The ACM is the direct equivalent of the BCS, and the IEEE mirrors the role of the IEE. Accreditation of individual courses is handled by the Computer Accreditation Commission (CAC)—a committee of the Accreditation Board for Engineering Technology (ABET). The CAC is advised by the Computer Science Accreditation Board (CSAB)—a professional federation with the ACM and IEEE amongst others as members. Every 10 years the ACM and the IEEE Computer Society jointly agree a model computer science curriculum for use by the CSAB, the most recent having been published in 2001 (ACM/IEEE, 2001). Distinct curricula for computer engineering, software engineering and Information Systems are also in preparation.

The ACM/IEEE curriculum is highly prescriptive, and specifies in depth both a core body of knowledge and several 'implementation strategies' for delivering that knowledge. As a consequence, accredited courses tend to have similar modules, with identical names and module codes, and there is a large choice of textbooks each targeted at each named module. Although most UK universities do not normally wish to seek USA accreditation, the ACM/IEEE approach taken to group projects within the CS curriculum is not dissimilar from the BCS in the UK (and explicitly draws on the QAA benchmark), and it will be helpful to us to refer to the ACM/IEEE curriculum.

It is interesting to compare the BCS (or IEE) with other EC nominated bodies in the UK. For example, CEng status is often a required qualification for senior jobs within the civil engineering profession. In contrast, in May 2002 the BCS had less than 17,000 full members (BCS, 2003), only a minority of IT professionals, and few undergraduates join the BCS as Associate Members. There is minimal regulation of the IT industry in the UK, and the professional bodies do not have the same influence as in other engineering disciplines. This mirrors practice in most countries, the one exception being the State of Texas (2001), where use of the phrase 'software engineer' is controlled by law.

This raises the obvious question of why an academic degree programme would wish to be approved for BCS exemption or EC accreditation. A compelling reason is course marketing, where accreditation may be seen by potential students as a seal of quality. This is particularly important at present where a significant proportion of students are overseas, and who scrutinize quality indicators carefully when selecting their course.

Group projects in computer science

Team work within computer science typically refers to a group project, where several programmers work together to write an item of software. Raymond (2001, p. 52), in a discussion of the historical development of the Linux operating system, applies Kropotkin's 'severe effort of many converging wills' to the successful completion of a collaborative software development project. A group project requires hard work, commitment and enthusiasm. There is a substantial body of knowledge on the application of group work within this context, including a section of Fincher *et al.*'s (2001) good practice guide for projects in computer science, and Hughes and Cotterell's (2002) guide to software project management.

The professional bodies all include group work within their framework for a computing degree course. The QAA (2000, p. 6) specifies as a 'computing-related practical ability': 'The ability to work as a member of a development team, recognising the different roles within a team and different ways of organising teams'.

The BCS requires that graduates of accredited courses should have the ability to 'work in multidisciplinary teams' (BCS, 1998, p. 18) . Furthermore, each accredited course must contain a substantial group project, which must fulfil (*inter alia*) the following criteria (BCS, 1998, p. 22):

- 'it should exhibit a structured approach to information systems practice, involving a number of stages in the life cycle;
- each student must clearly identify their contribution to the overall project; … and the assessment must clearly identify each individual's personal contribution;

- the product must exhibit the attributes of quality, reliability, timeliness and maintainability'.

Furthermore, the project must generate a report which should include (BCS, 1998, p. 21):

- 'a clear description of the stages of the life cycle undertaken;
- a description of how verification and validation were applied at all stages;
- appendices—technical documentation'.

The ACM/IEEE curriculum (2001) refers to 'teamwork with individual accountability', and requires the incorporation of a 'significant team project that encompasses both design and implementation'. It suggests a model for this as a fourth-level 'Capstone Project' (a project which brings together the knowledge and skills assimilated during the course of study) being a 'team-oriented, software-engineering effort'.

There seems, therefore, to be a general understanding that the nature of the programming activity makes group work a desirable and necessary component of the computer science curriculum. However, this does not allow academics to use the full range of team activities described in the literature. The necessity of access to equipment and associated software tools makes inappropriate many structured activities which might be used in the humanities and social sciences, and the accreditation requirements further circumscribe the content of the project. More important, from a practitioner's standpoint, are the theoretical foundations for the discipline known as 'software engineering', which effectively prescribe the type of activity that can be undertaken.

Software engineering

As we have noted above, computer science is an Engineering discipline, and a major topic within the subject which might be seen as deserving such a characterization is that of software engineering.

Software engineering is 'a systematic approach to the analysis, design, implementation and maintenance of software' (FOLDOC, 2003). Ghezzi *et al.* (2003, p. 1) describe it as the 'building of software systems which are so large or so complex that they are built by a team or teams of engineers'. The discipline requires the approach to be defined accurately, and group work is fundamental to the process. A group project in computer science and a software engineering project are to most intents and purposes synonymous. A successful group project would necessarily apply good software engineering practice, whereas a software engineering project would not be a solitary activity.

Although there are many approaches to systematizing the software development process (Schach, 2002), the standard view of the process is a variant of the 'waterfall model' (see Figure 1).

This process involves the following phases (the 'software life cycle'):

1. the requirements of the 'customer' are ascertained;
2. a specification (a detailed—possibly formal—description of the intended product) is written;
3. the software specification is then analysed, the software designed, and the specification is mapped to a collection of structures appropriate to the programming language to be used;
4. the software is implemented (i.e. coded) in a programming language; and finally
5. the software is then tested (to remove any errors).

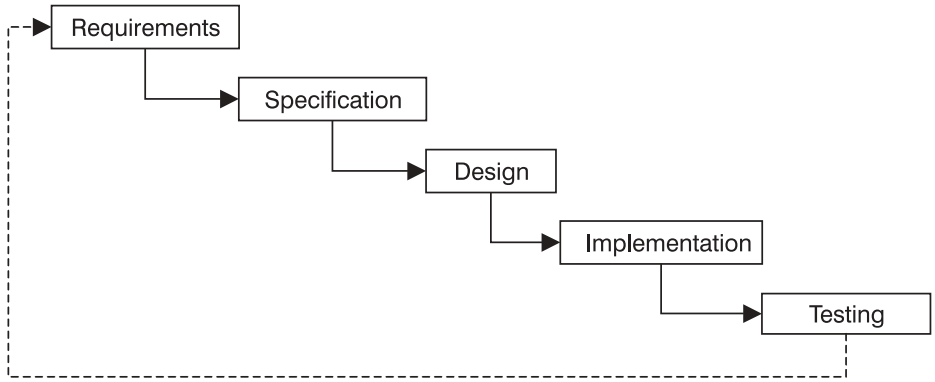

Figure 1. The 'waterfall model' of software development

At the end of the process, there may be further iterations. For example, experience implementing the software may reveal an implicit flaw in the specification, or the customer may modify his/her requirements. This model of the software engineering process bears a strong resemblance to that of other engineering disciplines—for 'programming language' read 'construction materials', for 'coded' read 'constructed', and the process is simply mapped to the civil engineering equivalent.

Within this broad framework, there is a variety of software engineering 'methodologies' which further guide the programmer, requiring for example specific documentation to be generated during the process, and indicating the use of supporting software tools to assist in correctly relating design, code and documentation together.

The organization of a team within such a software engineering project would typically be hierarchical, with a team leader delegating specific well-defined tasks to individual team members.

Only application of knowledge and team-working skills, of the six motivating factors for group work listed above, appear to be directly and unambiguously relevant. Whilst we would hope that students would be motivated, and autonomously develop higher-order cognitive skills, and that we would measure effectively individual students' contributions, those outcomes would depend on the details of the project to be undertaken, and the structure and internal dynamics of the individual group. The ideological perspective might even be adversely affected by excessively authoritarian management of a group.

This 'classical' process, however, does map well on to the criteria specified by the external agencies. The software life cycle is catered for, individual contributions of students can be ascertained and the production of suitable documentation is implicit.

Agile programming

The standard model of the software engineering process is open to criticism. The process can be very inflexible, and the burden of documentation is often heavy. Furthermore, many customers do not actually know what they want at the start of the software development process, and classical processes do not easily allow for changes of specification part-way during the development (Martin, 2003).

There is now a trend towards what is known as *agile programming* (Martin, 2003), a paradigm for software development which eschews classical wisdom on the subject, and is instead guided by the lessons learned by the pragmatics of the team programming activity.

The most well-known agile process is XP—an acronym for 'Extreme Programming' (Beck, 1999)—though others are used, such as SCRUM (ADM Inc., 2003) and Crystal (Cockburn *et al.*, 2003). These processes are lightweight, and perhaps best suited to small project development. They are iterative, and require the active involvement of the customer throughout the development process. So, instead of the software life cycle, the software development process is subdivided—in the case of XP the divisions are 'releases' lasting typically several months, themselves subdivided into 'iterations' of up to three weeks (see Figure 2).

At the start of each iteration a planning meeting is held with the customer, and the programming team is present, to decide what work is to be done during that iteration. This is an exercise in the customer listing desired features ('user stories'). These are short documents (often written on index cards), in contrast to the lengthy usage scenarios found in classical software engineering.

During each iteration the coding is shared, and is dynamically allocated. Each programmer selects the tasks he/she wishes to attempt, but during the iteration may spend time assisting other programmers. In XP, 'pair programming' is expected, where two programmers use each computer terminal at the same time, so that only one of them can actually be entering code whilst the other keeps a watchful eye on the machine, and this strategy is known to be an effective strategy to deliver good code (Williams & Upchurch, 2001; McDowell *et al.*, 2002).

In classical software engineering, the testing phase occurs at the end of the life cycle. In XP, tests must be written at the *start* of each iteration. In consultation with the customer, the team specifies tests for each user story which will be used to ascertain whether the user story has been

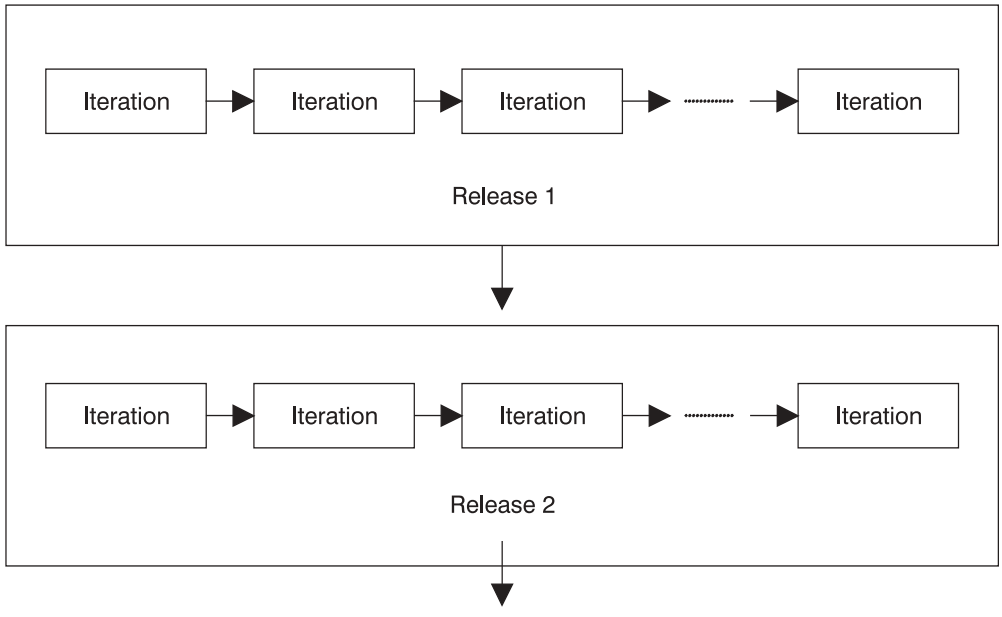

Figure 2. Cycles within an agile process

implemented correctly. Thus tests are used to drive the software coding, rather than being a validation check at the end of the process.

Another innovation is the approach to documentation. Classical processes generate large volumes of documents, but the agile view is 'Produce no document unless its need is immediate and significant' (Martin, 2003, p. 5).

The new curriculum

The computer science curriculum, as viewed by the BCS and ACM/IEEE, contains references to the software life cycle as fundamental to a group project, and requires the production of technical documentation. Each team member must be able to identify clearly their own contribution and the quality and maintainability must be measurable. How does this fit with agile programming?

The classical software life cycle does not exist within agile programming. The production of technical documentation is explicitly discouraged. The structure of a team mitigates against any member 'owning' part of the process or the code. Quality and maintainability are not issues which are explicitly part of the process, unless they are customer's user stories.

Another issue is the role of the lecturer. In the classical case, the project is presented to the group by the lecturer, acting as a customer, and the group 'delivers' a product at the end of the process. The agile paradigm involves the customer throughout, leaving the lecturer either to be deeply involved with the group, or for the customer to be a different person.

Agile programming has existed as an identifiable phenomenon since 1999, and is already popular and appears to be academically respectable. The pedagogic aims of having a group project seem to be met by using an agile approach. Indeed, the emphasis on close co-operation and team integration would appear to address all seven of the factors identified above, and there is no conflict with the QAA requirement. However, the curriculum requirement for a group project to demonstrate the use of the software life cycle appears to preclude the use of an agile programming process.

We must ask ourselves what is it that students are learning when they undertake a group project, and what are we measuring. If our focus is on effective group co-operation in order to deliver a demonstrably good-quality product, then we should not prevent the use of a process which supports this aim.

Pedagogy of agile programming

We have indicated that the use of agile programming in the software development process is of interest, and that there is a potential conflict between agile programming methodologies and the current requirements for a computer science curriculum. If we can demonstrate that the pedagogic benefits of such a methodology suggest that its use in group projects is desirable, then we can argue that the software engineering requirements for a group project should be interpreted so as to include agile programming.

We have taken as a case study a single group of four students enrolled on the four-year Master of Engineering (MEng) course in Computer Science at Warwick University. This course contains two group projects, one in the second year and one in the fourth. The four students worked together as a team in both projects, the second-year project delivered using a

'traditional' methodology, but for the fourth-year project they used the XP agile methodology. We interviewed all four students to ascertain how they perceived the differences in the two methodologies affected their learning experiences, and we summarize the results under seven headings, the seven reasons for students to undertake group work we introduced above.

Application of knowledge

The skills used in the two cases (which we will refer to as 'traditional' and 'XP') were perceived as being very different. Traditional methodologies require diagrammatic methods, whereas XP demands negotiating skills (for communicating with the customer), for example. Depending on the role of a given team member in a traditional project, that student may only use a part of the knowledge he/she has been taught, whereas in XP, since there is no 'ownership' of code or process, the student must practise a greater variety of skills. The point was also made that in any such project (whether traditional or XP) the knowledge a student brings is rarely sufficient, and that part of the educational experience during a project involves the assimilation of new skills which have not formed part of the taught syllabus.

The constraints of an assessed project were felt to be an issue for the traditional methodology. Typically, the instructor will provide the students with the specification of the software to be written, and students therefore are unable to apply that part of the methodology which refers to the specification phase. However, it would be in practice very difficult to provide a task for a group to perform without a specification. In XP, the evolution of the customer requirements and the less formal nature of the specification mean that this is not an issue.

Motivation

The students were very clear that in XP the motivation is consistent throughout the project, due to the regular requirement for iterations to be completed involving all members of the team. This contrasts with the 'bursty' (non-constant) nature of the traditional process, where a team member might be very heavily involved at specific parts of the process and lightly involved at other times (e.g. the student in charge of the design phase would have his/her work skewed to the start of the project).

Another point made was that with a traditional methodology there may be a temptation for students to delay their work until the last minute, perhaps producing documentation which obscures their failure to adhere to the time dependencies of the components of the software life cycle. For example, code might be written in haste, the testing phase ignored and the design documents retrospectively written. This would be impossible with XP.

Perhaps the most persuasive argument in favour of XP motivating students is best summed up in the words of one of the students: 'I think it's just more fun, because it seems more suited to programming'.

Higher cognitive skills

The iterative nature of XP affects this also. Since a traditional methodology normally demands only one iteration, the opportunity for students to reflect on and evaluate their work cannot

comfortably take place until near the end of the project, whereas in XP this can happen after each iteration.

With a traditional methodology, the necessity that the design phase be completed well in order for the coding to be successful requires the students to think deeply about their work at the start of the process. The subsequent implementation and testing should then be a relatively straightforward process. In XP, each iteration requires careful thought about the design of the software.

Autonomy

In the traditional process, members of a team are allocated tasks, so a student's autonomy is restricted by that allocation. The opportunities to choose a task would depend on the composition of each individual group. In XP, an individual can—and, indeed, must—be able to contribute to different tasks throughout the project. So it would appear that a student would enjoy greater autonomy when using XP. However, this observation ignores the input of the customer, who may constrain the flexibility of the group to take informed decisions.

Assessment

This is a topic which the students felt was generally independent of the methodology employed. However, the point was made that sometimes the individual contributions of individuals are significantly unequal when a traditional methodology is used, whereas there is perhaps a tendency in XP for a levelling process to occur. This can partly be explained by individuals having ownership of specified parts of the process in a non-agile environment, and those components being of unequal size.

Ideology

Close contact with the customer in XP, pair programming and the joint ownership of code were factors which were considered to be beneficial in preparing students for future participation in society. One student remarked that his XP experience with 'customer contact' had been very helpful when he was applying for jobs, since his potential employers regarded it as a valuable skill. In contrast, a traditional approach might allow a 'loner' to complete his/her allocated task with minimal contact with the rest of the group.

Team skills

Both types of methodology were felt to enhance team skills, provided that the team members were engaged and enthusiastic.

Overall, we find that the experience of this specific group of students would seem to favour the use of XP on educational grounds. Motivation and student autonomy appear to be significantly enhanced, and deep learning is consistently promoted throughout. Issues relating to assessment, application of knowledge, ideology and team skills are less differentiated, but none was considered to favour the traditional approach to software development.

Conclusion

We have considered the use of group projects within the computer science curriculum, and remarked that a typical curriculum meeting the accreditation requirements of the professional bodies requires the use of a 'traditional' software engineering methodology. We have also noted the popularity within industry of a new type of software engineering methodology, so-called 'agile programming'. A small case study has indicated that there may be positive pedagogical benefits to be gained by using agile programming rather than a traditional methodology in computer science group projects.

Notes on contributor

Mike Joy is a senior lecturer in Computer Science at the University of Warwick. His research interests include educational technology in computer science education, programming languages and agent-based systems.

References

ACM/IEEE-CS Joint Curriculum Task Force (2001) *Computing curricula 2001.* Available online at: www.acm.org/sigcse/cc2001/ (accessed 10 April 2003).

ADM Inc. (2003) *SCRUM.* Available online at: www.controlchaos.com (accessed 23 April 2003).

BCS (1998) *Guidelines on course exemption and accreditation* (Swindon, British Computer Society).

BCS (2003) *Review 2003.* Available online at: www.bcs.org.uk (accessed 21 April 2003).

Beck, K. (1999) *Extreme programming explained: embracing change* (Reading, MA, Addison-Wesley).

Cockburn, A., Highsmith, J., Johansen, K. & Jones, M. (2003) *Crystal.* Available online at: www.crystal-methodologies.org (accessed 23 April 2003).

DfES (2003) *The future of higher education* (London, The Stationary Office).

Fincher, S., Petre, M. & Clark, M. (Eds) (2001) *Computer science project work: principles and pragmatics* (London, Springer).

FOLDOC (2003) *The free online dictionary of computing.* Available online at: www.foldoc.org (accessed 21 April 2003).

Ghezzi, C., Jazayeri, M. & Mandrioli, D. (2003) *Fundamentals of software engineering* (2nd edn) (New York, Prentice-Hall).

Gibbs, G. (1995) *Learning in teams: a tutor guide* (Oxford, Oxford Centre for Staff Development).

Henry, J. (1994) *Teaching through projects* (London, Kogan Page).

Hughes, R. & Cotterell, M. (2002) *Software project management* (3rd edn) (London, McGraw-Hill).

Jaques, D. (2000) *Learning in groups* (3rd edn) (London, Kogan Page).

QAA (2000) *Academic standards—computing* (Gloucester, Quality Assurance Agency for Higher Education).

Martin, R. C. (2003) *Agile software development: principles, patterns and practices* (New York, Prentice-Hall).

McDowell, C., Werner, L., Bullock, H. & Fernald, J. (2002) The effects of pair-programming on performance in an introductory programming course, *ACM SIGCSE Bulletin,* 34(1), 38–42.

Raymond, E. (2001) *The cathedral and the bazaar* (Sebastopol, CA, O'Reilly).

Reynolds, M. (1994) *Groupwork in education and training* (London, Kogan Page).

Schach, S. R. (2002) *Object-oriented and classical software engineering* (5th edn) (New York, McGraw-Hill).

State of Texas (2001) *Texas engineering practice act* (Austin, TX, Texas Board of Professional Engineers).

UCAS (2003) *Course search.* Available online at: www.ucas.ac.uk/ucc (accessed 10 April 2003).

Williams, L. & Upchurch, R. L. (2001) In support of student pair-programming, *ACM SIGCSE Bulletin,* 33(1), 327–331.