Int. J. Engng Ed. Vol. 24, No. 4, pp. 729–737, 2008
Printed in Great Britain.

0949-149X/91 $3.00+0.00
© 2008 TEMPUS Publications.

What can Software Engineering Students Learn from Studying Open Source Software?*†

D. A. CARRINGTON
School of Information Technology and Electrical Engineering, The University of Queensland, St Lucia QLD 4072, Australia. E-mail: davec@itee.uq.edu.au

There is a large gap between the scale and complexity of typical software products and examples used in software engineering education. Since complexity is considered an essential property of software systems, this gap creates a problem for software engineering students and educators. Studying open source software can provide software engineering students with realistic and challenging examples and pragmatic instances of abstract concepts such as software design patterns. For software engineering educators, the vast array of freely available software sources allows selection to suit their educational objectives and constraints. This paper reviews how open source software is used in a software engineering studio course and discusses the outcomes from the perspectives of students and educators.

Keywords: engineering education; open source community; learning environments; distributed software development

INTRODUCTION

A MAJOR CHALLENGE in teaching software engineering is helping students understand the differences between the small programs that they write as educational exercises and the large scale software products that they will deal with when they are working. This gap between education exercises and real software systems is continually widening. Brooks [3] identified complexity as one of five inherent properties of modern software systems and states that 'Many of the classic problems of developing software products derive from this essential complexity and its nonlinear increases with size'. Software engineering educators need techniques to help their students cross this gap as they transition from novices to practitioners.

The concept of sharing software source code is not new and has been common in academic and research communities since computing began. Many open source projects have started life in universities. However, the open source software movement has grown rapidly in the last decade with the expansion of the Internet and now involves many commercial and industrial participants [5].

While not a true open source project, the original distribution of UNIX to universities and research organisations in the mid 1970s included full source code and this had major consequences. Early UNIX adopters tended to share their extensions and adaptations. UNIX became widely used by students and gave rise to computer science's most famous 'suppressed' (to protect AT&T's trade secrets) publication when John Lions [9] documented the UNIX kernel source code to help teach operating system principles.

The open source software communities provide an enormous distributed repository of software artifacts that are freely available for study and experimentation. Not all artifacts are exemplars, but it is educationally valuable to expose students to a variety of programming styles and hence encourage their critical abilities. Hunt and Thomas [7] refer to this type of study as 'software archaeology', but an alternative analogy is the medical student pathology laboratory. The open source software repository continues to grow and adapt to changes in technologies. Using this repository is low cost and requires a relatively small effort by an educator. The systems are generally realistic and practical, which can be a major factor for student motivation. Using open source software also has the beneficial effect of ensuring that students are aware of the open source software movement, and opens up opportunities to discuss topics such as software piracy and ethics.

This paper discusses how open source software can be used as classroom material; in particular, it describes our experiences in a software engineering studio course focusing on software design. The

* Accepted 25 April 2008.
† Based on 'Teaching Software Design with Open Source Software' by David Carrington and Soon-Kyeong Kim, which appeared in the *Proceedings of the 33rd Annual IEEE/ASEE Frontiers in Education Conference,* Boulder, CO, November 5–8, 2003. IEEE Catalog No. 03CH37487, pp. S1C-9–S1C-14, © 2003 IEEE.

next section briefly reviews other approaches to using open source software as an educational resource. The following section explores the educational objectives that can be addressed through the use of open source software. The context for our software engineering studio course in which students are exposed to open source software, typically for the first time, is then explained. Examples of the types of open source software that we have used (a complete list is provided in the appendix) are given. Some of the results achieved by students during their practical work are reviewed, and the last section reflects on our experiences and provides suggestions for people interested in adopting our approach.

RELATED WORK

The availability of open source software is changing software development practice [13, 14, 17]. The last ten years has seen an enormous increase in software engineering research studying open source software since the artifacts are normally more readily available. As might be expected, many people are also leveraging the availability of open source software in educational contexts [2]. Most of this usage is to employ freely available software products as convenient and low-cost alternatives to commercial ones. However, the availability of source code for open source software does not appear to be as widely exploited.

Software engineering educators are beginning to discuss how they are using open source software to attain educational goals. Andrews and Lutfiyya [1] used some GNU software products in a senior software engineering course focusing on software maintenance. O'Hara and Kay [12] provide an overview that includes a description of some of the different open source licences. Nelson and Ng [11] describe a course on computer networking that relied on multiple open source packages. At the SIGCSE conference in 2002, a panel reviewed the social and ethical responsibilities associated with using open source software [19]. Hawthorne and Perry [6] consider the consequences for software engineering education of distributed development of software systems using open source software and COTS components. Liu [10] presents an iterative approach for developing open source software using multiple cohorts of students with the customers being local community organisations.

Shockey and Cabrera [15] describe the results from the SNAP Development Center within the Inter American University of Puerto Rico. The Center employs undergraduate student researchers developing open source software with the primary goal of transitioning them from trainees to software engineering practitioners. Toth [18] discusses how the availability of open source software engineering tools allows him to get students to evaluate existing software engineering tools, select a candidate and then extend that tool. The paper provides an insightful evaluation of the associated challenges and lists multiple 'lessons learned'.

Ellis et al. [4] are using an open source disaster management system as a basis for their software engineering program with undergraduate students developing software modules that are incorporated into the current releases. Ellis et al. provide several guidelines for using open source projects in software engineering education. Jaccheri and Østerlie [8] have developed an approach for teaching master's level students using principles from action research. Their students act as both developers and researchers in open source software projects. The case study in their paper refers to the Netbeans project.

EDUCATIONAL OBJECTIVES

Many aspects of software engineering can be illustrated through open source software. The most obvious feature is that such software is developed to solve real-world needs by software practitioners. For educational purposes, this feature can be extremely important. Open source software examples are more realistic than examples found in textbooks or developed specifically by teaching staff. This is not to say that all open source software is exemplary. Studying open source software reveals both good and bad examples, which can be a revelation to students who sometimes assume that all non-student software is high quality.

One important aspect of realism is the sheer size of real software systems. Textbook and classroom exercises are normally carefully chosen to be understandable after a few minutes of study, isolating the topic of interest from everything else. For students, this can create a false impression that all software is similarly small and manageable. It is important that students get an introduction to strategies for approaching a large and unknown body of code [16].

Another key lesson for students is the value of appropriate documentation. Novice programmers often struggle to appreciate why and how to document their code. When they write software, it seems clear to them that the code is 'self-documenting' and rarely do they need to revisit their code weeks or months later. However, the experience of studying source code written by another person normally makes the need for documentation clear. A particularly effective demonstration occurs if the student is supplied with a program written in a foreign language (identifiers and comments).

Because students tend to work on small software assignments whose lifetime is a matter of days or weeks, the problems associated with software evolution are usually not experienced. Again by inheriting an existing project and having to understand it and then modify it, they get a taste for the effort involved in software evolution/maintenance. Reverse-engineering a design from code without documentation can become a meaningful task rather than just a topic talked about in lectures. Refactoring code techniques can be applied to real examples where there is no text-book answer. All of these activities help students to understand how software engineering is practised and to develop appropriate skills.

LEARNING CONTEXT

The Software Engineering Studio course is a core element of the second year program for both the software engineering (4 year) and the information technology (3 year) degrees offered by the School of Information Technology and Electrical Engineering at the University of Queensland. Incoming students have previously completed at least one programming course using Java and are typically studying a data structures and algorithms course in parallel. The goals for the course are to:

1. demonstrate the concepts and practice of software design and testing, the UML notation, software design patterns, code refactoring and configuration management;
2. extend students' programming experience, particularly in terms of program size, but also in the use of additional program structures and development methods;
3. provide students with positive experiences of collaborative learning and some appreciation of the need for life-long learning skills.
4. expose students to open source software and real-world code written by other developers.

The one semester (13 teaching weeks) course is normally structured with a two-hour lecture and a two hour laboratory class each week. The practical work in the course requires students to study and modify one from a nominated set of open source software systems, in teams of three or four students (students are encouraged to suggest extra software systems to be added to the initial set). Each lecture is for the full class, while laboratory classes and tutorials are subdivided into groups of up to twenty-eight students (up to eight teams). Members of each student team are constrained to attend the same laboratory class so they can work as a team during this time. Team presentations are also required in some laboratory classes.

Assessment for the course includes three team assignments based on open source software. In the first assignment, students install and use their chosen open source system so they can give a five minute presentation to their peers explaining:

- the purpose of the system,
- the installation process, and
- how to use the system, preferably with a simple scenario.

In the second assignment, students undertake a detailed investigation of the source code. Using any existing documentation as a base, the team constructs a design guide for their system. The guide should incorporate an overview of the system's structure, including a complete list of classes and explain any structuring into packages. If the system relies on external libraries or other systems, these should be noted.

Since some systems are much larger than others, teams are encouraged to focus their detailed description on a coherent subset of the overall source code. A rationale for the selection of the subset is required. As a rough guide to the size of a subset for this assignment, we suggest focusing on about twelve to sixteen classes or about 2000 to 3000 lines of code (not counting comments).

UML class diagrams are to be constructed to capture the classes and the relationships between them. Not all attributes and methods need to be included for each class—inclusion should be on the basis of assisting understanding. UML interaction diagrams (collaboration or sequence) are not required in this assignment, but they can be used where they assist the documentation goal. No source modifications are required for this assignment.

A plan for the next stage of investigation is also required. While this plan is not binding on the team, it represents an opportunity to prepare for the final assignment. The plan should propose some extensions, improvements or refactorings to the software that the team feels capable of implementing. These enhancements are to be described as precisely as possible without getting into implementation details (i.e., what is to be extended, improved or refactored and why, but not how this is to be achieved). A test plan is also required for those parts of the system to be affected by the proposed enhancements. The test plan is to describe the class(es) to be tested, the proposed test process including any required test scaffolding, and the general types of tests that the team proposes to execute.

The final assignment requires each team to enhance their system, either by adding functionality or by refactoring the existing code. The deliverables are

1. source code changes and accompanying descriptions;
2. UML diagrams (class and sequence) illustrating the source changes;
3. testing information describing how the changes have been verified, and
4. a ten- to fifteen-minute presentation in the final laboratory class summarising the assignment outcomes.

OPEN SOURCE SOFTWARE EXAMPLES

The Internet has provided an amazing mechanism for world-wide information sharing and distribution. One application has been for programmers everywhere to establish repositories of executable systems and source code for others to use. SourceForge (http://sourceforge.net) is probably the best known of these repositories. Given that SourceForge itself currently has over 100 000 projects of which over 20 000 use the Java programming language, there is a selection problem, that is, what criteria can (and should) be used to select open source software examples for educational purposes.

The main criteria we have applied are:

- *Relevant:* The product should be potentially relevant in some way to the students in their software engineering studies. This criteria has been used to exclude software for computer-based games, a decision that is not strictly necessary but that avoids distracting students in their studies.
- *Understandable:* The application domain of the product needs to be comprehensible to the students otherwise they are likely to take an excessive amount of time researching the problem domain.
- *Java:* The software needs to be implemented in Java since Java is the programming language that students entering this course can be assumed to know.
- *Stand-alone:* The software needs to be able to be compiled and run in our student laboratories, so it must not rely on other systems that need to be installed, such as databases or web servers. For obvious reasons, the laboratory environment is controlled to avoid security problems.
- *Modest size:* This criterion was not applied when the course was first run in 2002. The effect of asking students to investigate Eclipse or NetBeans (even if they only had to study a small part) was completely underestimated. Since then, the size of the source has been considered a key factor. Deciding what is reasonable is not straightforward, but having thousands of source lines is generally necessary to be challenging, but having hundreds of thousands of source lines is excessive. For some products, part of the source code can be auto-generated by systems like Antlr (http://www.antlr.org/). Such code is generally considered to be out of scope within this course.

To illustrate how these selection criteria are applied, here are some types of products found to be effective and some examples used within this course. More details on the examples are provided in the appendix.

Editors and IDEs: These interactive tools easily satisfy the relevant and understandable criteria and are generally popular choices. Examples include: JEdit, NetBeans, Eclipse, DrJava, Jaxe, JOE, Jipe and RText.

Development tools: These tools assist the software development process. Examples include:

- Abbot (GUI testing framework)
- Ant (build tool)
- JarWizy (GUI for archive files)
- Java GUI Builder
- jCVS (GUI client for CVS)
- JIP (Java Interactive Profiler)
- JODE (Java decompiler and optimiser)
- JRefactory (refactoring tool)
- JUnit (testing framework)
- Process Dashboard (PSP/TSP support tool)

Ant and JUnit has subsequently been incorporated into the toolset used by all students within the course. JUnit is now also used as an exemplar during lectures on software design patterns.

Static analysers: These tools take source or byte code as input and generate derived information. Examples include:

- Checkstyle (checks against a coding standard)
- Classycle (analyses class and package dependencies)
- JavaNCSS (measures non-comment source lines and cyclomatic complexity)
- JDepend (generates design quality metrics)
- PMD (identifies problems like duplicate or dead code or potential defects)

Personal calendar/organisers: These programs can help students manage their lives. Examples include:

- Borg (calendar & task tracker)
- ConsultComm (time tracker for projects)
- Essential Budget (personal finance manager)
- FreeMind (mind-map editor)
- JreePad (personal organiser)
- Memoranda (diary manager)
- JCycleData (training log/diary)

OUTCOMES

To appreciate what is possible for second-year undergraduate students studying large open source software products, three of the more successful projects are reviewed. Successful teams generally become absorbed in understanding and modifying their software product. Not all student teams achieve the same results for a variety of reasons, including the skill and aptitude of individual team members, team cohesion and organisation, and interest in the course and the software under study. The time available is limited, and of course all the students are taking a variety of other courses that typically reach their peak workload at about the same time (the end of semester).

In 2003, one of the software products to be studied was FreeMind (version 0.6.1), an editor for mindmaps. The source contained over 70 classes organised into four packages. One of the teams investigating FreeMind recorded in their second assignment their impression of the source code structure:

At first when looking at the source code of FreeMind, it is quite obvious that the developers of this program were not writing their code to make it any easier for other people to understand how the program actually works. This of course made it extremely difficult to generate a source code description for this program. The almost complete lack of general commenting, proper code indentation, the absence of proper Javadoc comments, and any other documentation in general, was extremely frustrating when attempting to comprehend the composition and dynamics of this program.

In their final report, this team reported making the following changes to FreeMind.

1. The mouse listeners were revised so that the mouse-over effect of selecting classes was disabled, replaced by a one-click method for selecting nodes and a double-click method for folding/unfolding nodes. This functionality change was justified in terms of conforming to common user interface conventions.
2. The very long constructor method FreeMind() in the Main package was refactored by introducing ten new methods. Similar changes were made to the FreeMindApplet class.
3. The Controller class was also refactored by removing inner classes to improve readability.
4. The java.awt.robot class used to excellent effect for GUI-level testing (it was particularly effective in the final presentation—'look no hands!'). The Ant build script was also modified to automatically run the batch tests.

In 2004, several teams investigated Java Wizard (JWiz) (Version 1.0.4), a Java source code checker that identifies some common mistakes made by Java programmers. This tool was developed around 1997 for JDK1.3.1 at the Collaborative Software Development Laboratory at the University of Hawaii and has not been updated since 1999. JWiz uses the JavaCC compiler compiler to generate a parse tree for the input. Since the JavaCC EBNF input was not available, the parsing part of JWiz was regarded as a black box. One of the teams investigating JWiz made the following changes.

1. New checks were introduced to identify:
   a. lack of an update clause in a for loop;
   b. a loop control variable being modified in the loop body;
   c. methods named equal, hashcode or tostring;
   d. methods throwing overly general exceptions such as Exception or Throwable.
2. A configuration file facility was introduced to specify which checks were to be performed (rather than always perform all checks). This required removing the hard-coded call to checks.
3. The Factory pattern was introduced to initialise the JWiz checks.
4. The collection of warnings in a string buffer was replaced by an implementation of the collecting parameter pattern to simplify the code and allow additional operations on the list of warnings.
5. The program was enhanced so it could be used as an Ant task.

Essential Budget (Version 0.8rc2), a graphical personal finance manager designed for tracking home finances, was adopted in 2006 by several teams as their open source project. One of the Essential Budget teams undertook an extensive redevelopment of the product that required removing three classes, adding four new classes and over 1200 lines of new code. The changes included:

1. Changing the user interface to remove duplicated mechanisms for invoking functionality, for example, menu items such as 'Accounts', 'Categories' and 'Budgets' appear in a menu bar at the top of the window, also at the bottom of the window and in a drop-down menu under the 'View' tab. The reclaimed user-interface space was then used for new functionality.
2. Adding the ability to load and save named files so that multiple budgets could be active on the same computer.
3. Adding functionality to perform reconciliation between bank statements and Essential Budget information.
4. Providing printing facilities.
5. Linking help information by creating a browser window onto a new HTML help file.

All teams are required to demonstrate via tests that the changes to their product have been successfully implemented and are encouraged to develop unit tests (preferably using JUnit) for all modified classes.

LESSONS LEARNT

As already mentioned in the section above, 'Open source software examples', choosing an appropriately sized open source software product is important if the educational objectives are to be achieved. It is also useful to pre-test the product to ensure that it compiles and runs within the laboratory environment to be used by students. Otherwise, it can be frustrating for students and seriously impede their progress. For one of the early classes, we identified an interesting product that ran in our environment. However, one of the required libraries was no longer available, making it impossible to build from source. The most obvious things to check are any dependencies on hardware or (more usually) other software. Checking for the need to install servers (such as the Apache HTTP server) or other system-level software is important as this type of software may conflict with limitations on student-installed software in laboratories or organisational security policies.

It is critical to motivate the students and establish realistic expectations. Many students find it less motivating to study existing software and prefer to develop something new from scratch, perhaps not appreciating that software maintenance and evolution is typical of industry practice. There is often a major hurdle with the need to abstract from all the details of a software product that is usually several orders of magnitude larger than anything they have studied before. Abstraction is a non-trivial skill to develop and requires a significant effort in this context where the detail is initially unknown.

I believe that the use of student teams is an essential factor in the success of this course. While teams were introduced for other educational reasons, they have multiple benefits for the investigation of open source software. Since studying a large software product is a novel experience for almost every student in the course, the team provides a set of interdependent peers who share the same goals and experiences. The collective expertise of the team provides a context for discussion and resilience in the face of difficulty.

For instructors there are assessment challenges, even beyond assessing group-work. It is infeasible for instructors to have detailed knowledge of all the open source software products being investigated. As a consequence, assessment is performed through student presentations and reports. The presentations offer an opportunity for the instructors to ask questions to check on the depth of understanding and knowledge. Since the course typically has about 100 students, we run multiple tutorial classes (with about 20–24 students in each class). For variety in the student presentations, the teams in each tutorial class are constrained to select different open source products. However for consistency, all teams investigating the same open source product are assessed by the same instructors. Normally, two instructors assess each team assignment submission. While this appears to double the assessment load, the effect helps improve consistency across instructors and allows each instructor to focus on particular aspects of the submission. A key challenge in the assessment is realising that there is no 'right' answer, so assessment needs to be done against specified criteria.

Teaching staff need to be capable of helping students if they initially fail to engage with the task. This is best done face-to-face with each team separately, and may involve the product they are studying or another example. Showing students how the task might be approached and then making concrete suggestions on how they could proceed further usually seems to help.

In general, the better students (as assessed by grades in courses other than this one) seem to like the open-ended nature of their task while the weaker students struggle. Some students find it difficult to imagine that they might be able to extend or improve the work of other people:

> 'It's hard for us to say that our design could possibly be better than the original programmers.'

CONCLUSIONS

The open source software movement provides wonderful resources for teaching software engineering. Our experience using open source Java systems in this course on software design and testing has been positive, although there is still room for improvement. Students generally appreciate the opportunity to use and explore real world software and to study its internal construction. While this paper has focused on a single course, studying open source software is applicable throughout the software engineering and computer science curriculum. Courses on compilers, operating systems, networks, middleware, and so on can all benefit from having students investigate real examples of relevant software products.

For teaching staff, the challenge is to provide adequate support and scaffolding for open source learning since they are unlikely to be familiar with all the details of the systems. They need to be able to demonstrate how to reverse engineer software by exploring and extracting critical information from the source code.

There are some open issues which we have not yet resolved:

- Can students' work contribute to the open source community through feedback, software changes or documentation?
- Can the open source community interact with the students other than by providing their software?

Acknowledgements—I would like to thank Ian Hayes, Daniel Jarrott, Soon-Kyeong Kim, Erica Mealy and Alan Whiteside as additional teaching staff involved in developing and presenting the Software Engineering Studio course over the past six years. I also wish to thank the numerous tutors who provided face-to-face assistance and assessment for this course. In addition, I want to acknowledge the hundreds of students who accepted the challenge of studying large and complex software systems.

REFERENCES

1. J. H. Andrews and H. L. Lutfiyya, Experiences with a software maintenance project course, *IEEE Transactions on Education*, **43**(4), (2000), pp. 383–388.
2. Graham Attwell, What is the significance of Open Source Software for the education and training community?, *Proceedings of the First International Conference on Open Source Systems*, (2005) pp. 353–358.

3. Frederick P. Brooks, Jr., No silver bullet: essence and accidents of software engineering, *IEEE Computer*, **20**(4), (1987), pp. 10–19.
4. Heidi J. C. Ellis, Ralph A. Morelli, Trishan R. de Lanerolle and Gregory W. Hislop, Holistic software engineering education based on a humanitarian open source project, *Proceedings of the 20th Conference on Software Engineering Education & Training*, (2007), pp. 327–335.
5. Joseph Feller, Brian Fitzgerald and Eric S. Raymond, *Understanding Open Source Software Development*, Addison-Wesley, (2001).
6. Matthew J. Hawthorne and Dewayne E. Perry, Software engineering education in the era of outsourcing, distributed development, and open source software: challenges and opportunities, *Proceedings of the 27th International Conference on Software Engineering*, (2005) pp. 643–644.
7. A. Hunt and D. Thomas, Software archaeology, *IEEE Software*, **19**(2), (2002), pp. 20–22.
8. Letizia Jaccheri and Thomas Østerlie, Open source software: a source of possibilities for software engineering education and empirical software engineering, *Proceedings of First International Workshop on Emerging Trends in FLOSS Research and Development,* (2007).
9. J. Lions, *Lions' Commentary on UNIX 6th Edition, with Source Code*, Peer-to-Peer Communications, (1996).
10. Chang Liu, Enriching software engineering courses with service-learning projects and the open-source approach, *Proceedings of the 27th International Conference on Software Engineering ICSE '05*, (2005) pp. 613–614.
11. D. Nelson and Y. M. Ng, Teaching computer networking using open source software, *Proceedings of the ITiCSE 2000 conference*, (2000) pp. 13–16.
12. K. J. O'Hara and J. S. Kay, Open source software and computer science education, *Journal of Computing Sciences in Colleges*, **18**(3), (2000), pp. 1–7.
13. Eric S. Raymond, *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*, O'Reilly Media, (2001).
14. M. Shaw, Software engineering education: a roadmap, *Proceedings of the Conference on the Future of Software Engineering*, ACM Press, (2000) pp. 371–380.
15. Kevin Shockey and P. Cabrera, Using open source to enhance learning, *Proceedings of the 6th International Conference on Information Technology Based Higher Education and Training*, (2005) pp. 7–12.
16. Diomidis Spinellis, *Code Reading: The Open Source Perspective*, Addison-Wesley, (2003).
17. Diomidis Spinellis and Clemens Szyperski, How is open source affecting software development?, *IEEE Software*, **21**(1), (2004), pp. 28–33.
18. Kal Toth, Experiences with open source software engineering tools, *IEEE Software*, **23**(6), (2006), pp. 44–52.
19. M. J. Wolf (Moderator), Open source software: intellectual challenges to the status quo, *Proceedings of the 33rd SIGCSE Conference*, (2002) pp. 317–318.

APPENDIX

This appendix provides an annotated list of the open source software used in the Software Engineering Studio course over the past six years.

2002	
Program name and web reference	**Description**
Ant ant.apache.org	Apache Ant is a Java-based build tool.
ArgoUML argouml.tigris.org	ArgoUML is a UML modelling tool with support for UML 1.4.
Eclipse www.eclipse.org	Eclipse is known as a Java IDE, but is much more.
jEdit www.jedit.org	jEdit is a programmer's text editor.
JUnit www.junit.org	JUnit is a regression testing framework for implementing unit tests in Java.
NetBeans www.netbeans.org	NetBeans is an IDE for software developers.
PMD pmd.sourceforge.net	PMD scans Java source code and looks for potential problems.
ProcessDashboard processdash.sourceforge.net	Process Dashboard is a PSP/ TSP support tool.
JRefactory jrefactory.sourceforge.net	JRefactory is a tool to refactor Java source code.

2003

Program name and web reference	Description
Classycle classycle.sourceforge.net	Classycle analyses Java class and package dependencies.
FreeMind freemind.sourceforge.net	FreeMind is an editor for mind-maps.
ICEMail www.icemail.org	ICEMail is an email client.
JarWizy sourceforge.net/projects/jarwizy	JarWizy is a GUI for archived file formats (jar, zip, tar).
Java2D java.sun.com/products/java-media/2D/	Java2D is part of the Sun Java distribution and includes a set of sample programs.
JavaNCSS www.kclee.de/clemens/java/javancss	JavaNCSS is a utility for measuring two standard source code metrics for the Java programming language.
jCVS www.jcvs.org	JCVS is a GUI client for CVS.
JDepend clarkware.com/software/JDepend.html	JDepend generates design quality metrics for Java.
LOCC csdl.ics.hawaii.edu/Tools/LOCC	LOCC measures the size of work products.
SwingSet2 *Part of the Sun Java 1.4 distribution*	Program demonstrating features of the SwingSet library.

2004

Program name and web reference	Description
Borg borg-calendar.sourceforge.net	BORG is a combination calendar and task tracking system.
ConsultComm consultcomm.sourceforge.net	ConsultComm keeps track of time spent on projects.
DrJava www.drjava.org	DrJava is a Java development environment, designed primarily for students.
JMT jmt.tigris.org	JMT (Java-Measurement-Tool) measures and judges Java code.
Jreepad jreepad.sourceforge.net	Jreepad is a personal organizer, information manager and text editor.
JWiz csdl.ics.hawaii.edu/Research/ JWiz/JWiz.html	JWiz is an automated code checker for Java programs.
Memoranda memoranda.sourceforge.net	Memoranda is a diary manager and a tool for scheduling personal projects.
UMLet www.umlet.com	UMLet is a simple UML drawing tool.

2005

Program name and web reference	Description
Abbot abbot.sourceforge.net	Abbot is a framework for unit and functional testing of Java GUIs.
Checkstyle checkstyle.sourceforge.net	Checkstyle is a tool to help programmers adhere to a coding standard.
Jaxe jaxe.sourceforge.net	Jaxe is a Java XML editor.
Java Gui Builder jgb.sourceforge.net	The Java Gui Builder decouples the GUI building code from the rest of the application using an XML file.
jGnash jgnash.sourceforge.net	jGnash is a personal finance manager.
JODE jode.sourceforge.net	JODE is a java decompiler and an optimizer.
Mars Simulation Project mars-sim.sourceforge.net	The Mars Simulation Project is a simulation of human settlement on the planet Mars.
NFC Chat nfcchat.sourceforge.net	NFC is a chat server and client.
Phosphor phosphor.sourceforge.net	Phosphor is a peer-to-peer file sharing program.

2006

Program name and web reference	**Description**
Essential Budget sourceforge.net/projects/essentialbudget	Essential Budget is a personal finance manager.
Java Outline Editor (JOE) outliner.sourceforge.net	The Java Outline Editor (JOE) is a folding editor.
JCycleData jcycledata.sourceforge.net	JCycleData is a training log / diary.
Jipe jipe.sourceforge.net	Jipe is a small Java IDE.
RText rtext.sourceforge.net	RText is a programmer's text editor.
Search and Whatever searchnwhatever.sourceforge.net	"Search and Whatever" is a utility to assist performing search operations.

2007

Program name and web reference	**Description**
Classroom scheduler sourceforge.net/projects/cr-scheduler	An application used to help avoid conflicts when scheduling courses and professors into classrooms.
CyVis cyvis.sourceforge.net	CyVis is a software metrics collection, analysis and visualisation tool for java based software.
DocSearcher docsearcher.henschelsoft.de	DocSearcher provides searching capabilities for text, HTML, MS Word, MS Excel, RTF, PDF, Open Office (and Star Office) Documents.
Factory projectfactory.sourceforge.net	Factory is a project management tool, but it should be considered first as a personal organizer.
Google Web Toolkit (GWT) code.google.com/webtoolkit/	GWT is a software development framework that makes writing AJAX applications easy for developers.
JIP jiprof.sourceforge.net	JIP is a code profiling tool much like the hprof tool that ships with the JDK.
Rachota rachota.sourceforge.net/en/index.html	Rachota is an application for time-tracking multiple projects. It displays time data in diagram form and creates HTML reports.
Timmon timmon.sourceforge.net	Timmon is a time tracking tool helping to keep track of time spent on different projects.

David Carrington is an Associate Professor in the School of Information Technology and Electrical Engineering at the University of Queensland, Australia and is the program director of the Software Engineering program. David has a broad range of research interests in the areas of software development, user interfaces and processes, including techniques and tools for formal specification, refinement techniques, design methods, programming environments and specification-based testing methods and tools. He was the Knowledge Area Specialist for the Software Engineering Body of Knowledge Project (SWEBOK) in the area of software engineering infrastructure (tools and methods).