An Open Source Culture in the Undergraduate Computer Science Curriculum

John David N. Dionisio, PhD
Assistant Professor, Computer Science
Loyola Marymount University

Partial support for this work was provided by the National Science Foundation's Course, Curriculum, and Laboratory Improvement Program, Award No. 0511732

Outline

- Background and Motivation
- Characteristics of an "Open Source Culture"
- Teaching Techniques
- Hardware & Software Infrastructure
- Early Returns

Impedance Mismatch!

Disconnect between undergraduate computer science training and expectations/skill sets required in industry

Undergraduate Training	Industry Expectation
Work alone	Work in a team
"Toy" programs and algorithms	Large, modular project
Throwaway code	Code longevity (for better or worse)

Historically Open

- Software as scholarly work, freely shared and published — a "golden age"
- Software became intellectual property, product of commerce, or trade secret

Returning to our Roots

Open source culture

New development approaches and practices; better development tools, programming languages, programming environments

Increased software complexity, decreased software quality

State of the Industry

- Commercial software remains (and probably won't go away — not a bad thing in and of itself IMHO)
- Open source software gaining broader exposure and acceptance (including new business models that embrace it)
- Continuing improvement in software engineering methods, techniques, tools

So how can we take this to school?

Official Open Source Definition (version 1.9)

Free redistribution

Source code

Derived works

Integrity of the author's source code

No discrimination against persons or groups

No discrimination against fields of endeavor

Distribution of license

License must not be specific to a product

License must not restrict other software

License must be technology-neutral

"Open Source Teaching Framework"

- Adaptation, not adoption
- Shift in goals: learning computer science, not software development
- Curriculum-wide scope vs. individual courses

Curriculum Progression

Capstone projects
(team, individual)

Term-length, focused projects

Concise functionality
with unit tests

Examine sample code, test
existing code, fix bugs

General Principles: Source Code

- All code — by faculty or students — resides in a centralized, public repository
- As much as possible, everyone's code is visible to everyone else — sometimes in the classroom, for code review or team fixing
- No code is thrown away — it remains available to future "generations"

General Principles: Quality & Community

- Documentation, inline and online
- Automated tests
- Constructive code review — give & take
- Form collaborative communities among faculty, students, classes, and projects

General Principles: Rights & Responsibilities

- Credit where credit is due
- Appropriate access and authorization
- Acknowledge, understand, and respect privacy, confidentiality, and license

Sample Code Bazaar

- Faculty members maintain live, organized, searchable, student-accessible sample code libraries for their courses
- Students access the sample code using the same tools and processes that they would encounter outside — no handouts!
- "Derived works:" exercises involve extending existing code

The Cyclic Life of Code

- Student programming projects live beyond the original term and course
- Focus on existing code and not new, throwaway fragments
  - Add/improve unit tests
  - Find/fix bugs
  - Improve/refactor designs

Test Infection

1. Instructor specifies required functionality
2. Students submit unit tests
3. Instructor runs tests against a library of implementations — some good, some bad
4. Students submit code
5. Students' unit tests are run against each other's code

Release Early, Release Often, Release Open

- For courses with term-long individual programming projects
- Project milestones throughout the term
- In-class code review
- Code sharing for common functions
- Projects feed back to lower division work

"CourseForge"

- A hardware + software infrastructure for supporting the teaching framework
- Certain teaching elements are impractical without some degree of automation
- Derived from open source software, delivered as open source software — the system will use itself

Department-Wide Source Code Repository

- Multifaceted organization: faculty, students, courses, projects, homework, research
- Tags, versions, and branches
- Release mechanism for "official" submissions
- Read-only views over the Web

Automated Testing & Reporting

- Scripted actions upon source code submission: configurable per instructor, per project, per course, or per assignment
- Web and e-mail feedback
- Summary/reports for instructors, team leaders, individual students

Compile Farm

- Build, test, and run against multiple architectures and operating systems
- Isolate specific build fixtures for particular software requirements: computer graphics, database management systems
- Comprehensive build reports

Wiki

- Parallels structure of source code repository, with appropriate authorizations
- Integrated links to other Web-accessible information: source code repository, test and build reports, blogs/forums/community

Blogs, Forums, Community

- Several tiers, ranging from individuals to groups and projects
- Searchable archives, e-mail access

Initial Experiment

- Actual exercise in a programming languages course this semester
- Done "manually" — no CourseForge yet
- Work in progress!
- More to come

Assignment

- Use JavaCC to implement a parser for a simple language
- Provide a test suite for the parser: accept strings that belong to the language, reject strings that don't
  - Differentiate between lexical and syntactic errors in "bad" strings

Initial "Test Shakedown"

- Against instructor's "control" implementation
  - Should eventually include more than one implementation, both correct and incorrect
- Four total test errors caught

0	1	2	3	4	5	6
	ab	acd	b			

		Tests						
		0	1	2	3	4	5	6
Parsers	0							
	1				a			
	2	bbc	bd	b	bbbc	bb	bbe	b
	3							
	4							
	5	c			ac	c		
	6		f			f		

Findings

- Precision is key — need to ensure consistent understanding of what is required
  - Parallels requirements analysis
- Tests must cover both commission and omission errors
- Tests must be very fine-grained — otherwise failed test may mask additional failures

Student Responses

- Overall more enjoyable than a conventional programming assignment
- Increased motivation to get it right
- Satisfaction with working as a team to get everyone's code to "go green"

Under Consideration

- Grading and assessment
  - Working code
  - Comprehensive tests
  - Team participation
- License for student projects — particularly after they have graduated
- Replicability in other institutions

Conclusion

Adaptation of an open source culture at the curriculum level may improve undergraduate computer science education

- Better computer scientists
- Better match with industry expectations
- Increased individual satisfaction, sense of accomplishment, and real-world impact