Stimulating Creativity through Opportunistic Software Development

Željko Obrenović, *Technical University, Eindhoven*
Dragan Gašević, *Athabasca University*
Anton Eliëns, *Vrije Universiteit, Amsterdam*

Using opportunistic software development principles in software engineering education encourages students to be creative and to develop solutions that cross the boundaries of different technologies.

In "To Hull and Back," the 1985 Christmas special of the British sitcom *Only Fools and Horses*, the main characters Del, Rodney, and Uncle Albert decide to sail from London to Amsterdam in a hired boat. "Experienced seaman" Uncle Albert arrives to captain the boat. However, in the first serious test of his sailing skills, when the group gets lost in the North Sea, they discover that Uncle Albert has no navigational experience, despite spending years in the Royal Navy. He explains that during his days in the navy, he was a boiler maintenance man and didn't have to learn navigation because "you see, the boiler has a tendency to go wherever the ship's going."

You can find a similar story in software engineering practice and education. Software developers and students tend to be skilled in a particular technology, such as Microsoft .NET, Java, SOAP, PHP, or Flash. But when they have to navigate in a larger context and connect to systems built by others and in other technologies, they often become lost. Universities, practitioner books, and industrial training organizations contribute to this problem. They emphasize creating masterpieces of code from blank sheets of paper, ignoring technological issues or treating them in isolation, and largely overlooking the different skills required to integrate and test software that the individual didn't create and doesn't control.[1]

Here, we describe our experiences using opportunistic software development to fight the boiler-maintenance-man syndrome. We created a didactic method based on opportunistic software development and the principles of creativity support tools. We encouraged students in our courses to use this method to develop solutions that cross the boundaries of diverse technologies. By teaching students to opportunistically combine systems that were never meant to work together or even to be reused, we created a space in which they could produce many innovative ideas and solutions.

New Requirements for Software Engineering Education

To make software engineering education more practical and useful to students once they complete their studies, several computer science depart-

0740-7459/08/$25.00 © 2008 IEEE

ments started to promote design-based education (see the "Rethinking Software Engineering Education" sidebar). In design-based courses, students learn by building concrete and realistic solutions. Although these approaches use different means, they introduce three important requirements.

First, they suggest that educators provide a rich, more realistic, and engaging development context. Having deadlines, for example, teaches students to frame problems and solutions more realistically, while working with others teaches them to work in teams.

In addition, software engineering courses should let students develop their own designs through rapid prototyping. Assignments that encourage and stimulate student designs can increase students' involvement in their work.[2] Building and implementing a concrete prototype can also help students see incompleteness and inconsistencies in their ideas.[3]

Finally, such courses should use a didactic method that supports creative thinking. Students must learn not only the details of a particular technology, but also innovative ways of applying it. Collaboration and public performance, for example, let students exchange ideas and get feedback on their work.

Framework for Opportunistic Software Development Education

To support these educational requirements, we developed an educational framework based on ideas from opportunistic software development. This type of development is a good candidate for supporting new requirements for software engineering education because it emphasizes creativity, innovation, and imaginative ways of finding and gluing software to meet diverse users' needs.

Figure 1 (see the next page) shows our framework, which builds on our previous work in opportunistic software development with diverse software components.[4] The framework's tools, environments, and guidelines support the creation of a rich development context from available software, rapid prototyping, and student creativity.

A Rich Development Context

Our educational framework's main goal is to create a context in which students can focus on higher-level, innovative software composition with advanced components. To help educators create such a context from a range of diverse software components and services, we developed a pragmatic approach to software integration.[4] Our Amico (adaptable multi-interface communicator, http://amico.sourceforge.net) middleware platform supports this approach. Amico provides a common space in which users can interconnect diverse software services and components. We adopt a service-oriented approach to integrating diverse components. We run components as stand-alone

Rethinking Software Engineering Education

Software engineering educators must teach students to think creatively to find innovative solutions to real problems. Often, however, students finish their studies without ever being exposed to such problems. Having identified this gap between typical computer science education and software engineering practice, some computer science departments have used different approaches to teach students skills that are closer to software engineering practice.

Fred Martin has argued for a more realistic educational context for software engineering.[1] He claims that teaching should be interactive and collaborative, noting that oversimplified toy examples represent the current state of the practice in software engineering education. Chuan-Hoo Tan and Hock-Hai Teo also argue that giving students experience developing and delivering large-scale systems under time constraints and shifting deadlines can better prepare them for future challenges.[2]

To make software development education more engaging and realistic, several universities have begun to introduce courses on games development. Kajal Claypool and Mark Claypool argue that many projects currently used in software engineering curricula lack both a fun factor to engage students and the practical realism of engineering projects that include other computer science disciplines such as networks or human-computer interaction.[3]

Ian Parberry and his colleagues explored how instructors can use game programming with art students, arguing that such an approach creates the opportunity for diverse communities of students to collaborate on joint projects.[4] Ming-Hsin Tsai and his colleagues also applied game design in the education of art and design students.[5] They report that the students made complete games, not just oversimplified exercises or simple walk-through scenes. Moreover, they gained enough fundamental programming knowledge to take intermediate programming courses.

The Rethinking CS101 Project (www.cs101.org) claims that most introductory programming courses—which typically teach computation as sequential problem solving—are outdated. Rather, such courses should emphasize interaction among processes.

References

1. F. Martin, "Toy Projects Considered Harmful," *Comm. ACM*, vol. 49, no. 7, 2006, pp. 113–116.
2. C.-H. Tan and H.-H. Teo, "Training Future Software Developers to Acquire Agile Development Skills," *Comm. ACM*, vol. 50, no. 12, 2007, pp. 97–98.
3. K. Claypool and M. Claypool, "Teaching Software Engineering through Game Design," *Proc. 10th Ann. SIGCSE Conf. Innovation and Technology in Computer Science Education* (ITiCSE 05), ACM Press, 2005, pp. 123–127.
4. I. Parberry, M.B. Kazemzadeh, and T. Roden, "The Art and Science of Game Programming," *Proc. 37th SIGCSE Technical Symp. Computer Science Education* (SIGCSE 06), ACM Press, 2006, pp. 510–514.
5. M.-H. Tsai, C. Huang, and J. Zeng, "Game Programming Courses for Nonprogrammers," *Proc. 2006 Int'l Conf. Game Research and Development*, ACM Press, 2006, pp. 219–223.

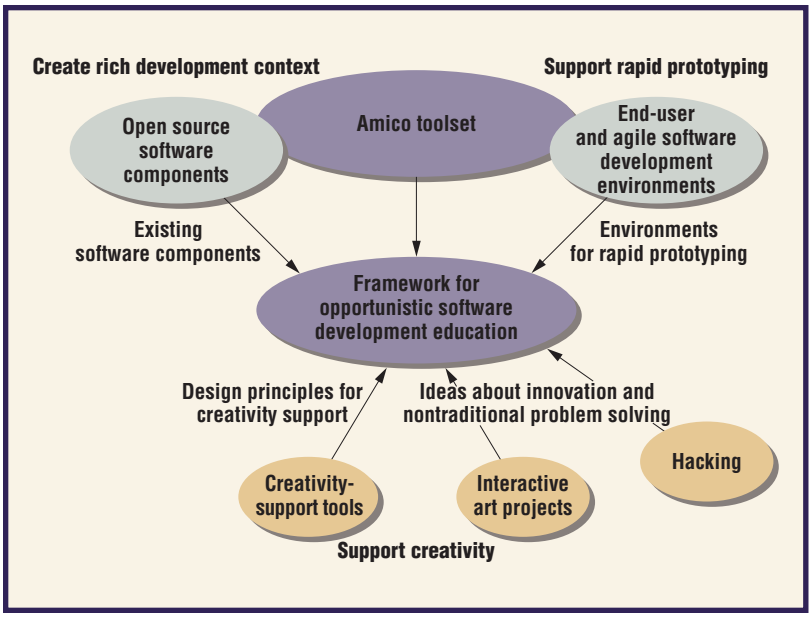


Figure 1. The framework for opportunistic software development education. Our framework consists of a set of tools, including our Amico (adaptable multi-interface communicator) middleware, and guidelines that can help educators teach students to be more creative and innovative.

applications that offer their functionality through open communication interfaces. Building, installing, and running stand-alone programs is usually a straightforward activity, even for users unfamiliar with the component's technology. (For example, even if you don't know Python or Java, it's still relatively easy to install interpreters for these languages and build and run their applications.) Therefore, applications written in one language can more easily use components written in another language. (For example, you can expose a component's functions with the Python Open Sound Control (OSC) server library, which other components can access through Java or C++ OSC client libraries.)

Turning software components into stand-alone services doesn't require changing their basic functionality. You need only add the code offering the functionality through any open communication interface, such as XML-RPC, OSC, SOAP, or an application-specific TCP or UDP (User Datagram Protocol) interface.

Tools for Rapid Prototyping

We support diverse development environments on top of our middleware framework. We don't limit students in their choice of development technology, and we provide several solutions, including

- *spreadsheets* for students with limited or no programming skills;
- *Web browser scripting* for students with Web development experience, including asynchronous JavaScript and XML (Ajax), applet support, and browser extensions;
- *declarative programming mashups* for students with experience in declarative programming languages, including XML-based configuration files and Prolog; and
- *programming libraries* for students with experience in procedural and object-oriented languages.

Users can start with a basic and simple development environment (such as a spreadsheet), switching to a more advanced mashup interface as their expertise develops and they need more complex functionality. For example, in our Intelligent Multimedia Technology course (which we describe later), students initially used spreadsheets to quickly sketch, discuss, and evaluate interactive system prototypes. They then switched to declarative and procedural programming and Web browser extensions to create more complex solutions.

Various tools follow this design philosophy. Many video games have dozens of layers, most search engines have novice and advanced layers (for example, Google and Yahoo), and many art and video tools have three or more workspaces (for example, Apple Final Cut Pro and Adobe Premiere).

Creativity Support Principles as a Didactic Method

Software construction requires approaches that empower and liberate the creative mind.[3] Opportunistic software development offers many possibilities; however, conventional approaches can't easily solve problems such as black-box components that lack detailed API descriptions and interoperability. Dealing with such problems requires creativity. Many design-based software engineering courses implicitly support creative thinking. We wanted, however, a more structured set of guidelines to support creative and innovative thinking in dealing with opportunistic software development challenges.

After reviewing other researchers' experiences, we chose to reuse the design principles defined by Mitchel Resnick and his colleagues.[5] By reusing existing principles, we hoped to put opportunistic software development into a broader context of existing creativity-support tools.[5–7] Resnick and colleagues define the following design principles:

- *Support exploration*. Let users try many alternatives before settling on a final design.
- *Support low thresholds, high ceilings, and*

wide walls. Make it easy for beginners to start, but also let experts work on more-complicated projects, and support a range of explorations.
- *Support many paths and styles*. Assist learners with different styles and approaches.
- *Support collaboration*. Encourage teamwork.
- *Support open interchange*. Diverse tools that support creative work should be interoperable.
- *Make it as simple as possible—and maybe even simpler*. Avoid making tools too complex by adding unnecessary features.
- *Choose black boxes carefully*. Carefully select the primitives that users will manipulate.
- *Invent things that you would want to use*. Use your own experience in creative work.
- *Balance user suggestions with observation and participatory processes*. Involve end users in the design process.
- *Iterate, iterate—then iterate again*. Support iterative design using prototypes.
- *Design for designers*. Build tools that let others design.
- *Evaluate your tools*. Use empirical testing methods; don't rely on intuition.

Software engineering educators can also use examples from interactive art projects and hacking to demonstrate innovation and nontraditional problem solving.[8] To find these examples, we looked at electronic-art conferences, such as Ars Electronica (www.aec.at), the Dutch Electronic Arts Festival (DEAF, www.deaf07.nl), and ACM's Multimedia Interactive Arts track. We also looked at hacking conferences, such as Blackhat (www.blackhat.com) and the Chaos Computer Congresses (www.ccc.de).

Applying the Framework

We aimed to create a generic environment that educators could use to promote opportunistic software development in different courses and domains. To adapt this framework to a concrete course, we propose the following steps:

- *Define a rich development context*. Select representative open source software or commercial projects and services, adapt them, and optionally connect them to our infrastructure using a service-oriented approach.
- *Create illustrative examples*. In each development environment that you plan to use during the course, show students how they can use and interconnect the different components. If possible, make components and examples open source and available online.
- *Define course objectives and assignments*. Using the creativity-support tool design principles as a guideline, create resources necessary to support these objectives, including mailing lists, links to inspiring projects, and Web sites where students can share their designs and notes.

We've followed these steps in applying our framework and tools to a course on integrated multimedia technology.

Case Study

Our Intelligent Multimedia Technology course, subtitled "Everything You Always Wanted to Develop …," sought to teach students to organize intelligent dialogues between users and complex systems, such as virtual environments and multimedia Web applications. The course went beyond direct control and conventional mouse- and keyboard-based interaction, introducing additional interaction modalities such as speech and camera-based user sensing (see http://amico.sourceforge.net/amico-demos.html for an illustration). The course's main focus was practical work and student creativity. It consisted of lectures, student presentations, and individual work in a laboratory and at home. Our lectures focused on integration patterns that showed how opportunistic software development can enable different technologies to work together, while students explored and presented particular technologies and combined them to build new solutions.

The class consisted of 32 undergraduate students (third and fourth year) from various departments, including cognitive systems, information systems, computer sciences, and artificial intelligence. The class also included several exchange students. Most of the students knew some computing basics, but their programming knowledge varied from beginner to experienced developers. Because of the huge diversity of technologies and student backgrounds, the course provided a good environment for applying and evaluating our educational framework.

Students proposed several innovative solutions, combining a huge range of technologies. Figure 2 shows some of their work, which combines various components with available Web services. Most students learned and used the technologies for the first time during the course.

Students ranked the course positively, with an average mark of 4 (on a scale from 1 to 5, where 1 = very bad, 2 = bad, 3 = neutral, 4 = good, and 5 = very good), making the course one of the top-ranked in the department. One encouraging piece

Software engineering educators can use interactive art projects and hacking to demonstrate innovation.

Figure 2. Example student projects and proposals: (a) The World Time mashup combines the Microsoft Silverlight plugin, Google Maps Ajax component, and Earthtools.org Web service. (b) The More than a Song project combines Flex with Amazon, Flickr, and YouTube Web services. (c) The Mobile Device Interaction with Maps project uses an Amico back end to connect a Flex interface with the OpenStreet Web service and Java 2 Micron Edition MIDlets (Java programs for embedded devices). (d) The Wii device adapter lets players use gestures in virtual and game environments.

of feedback was students' desire for similar courses more often. The main negative feedback was the lack of course material and documentation.

In organizing the course, we learned some lessons in terms of defining the development context, adapting tools, and using guidelines that support creative thinking.

Defining the Development Context

To give students ideas about what they might build, we selected several open source components and services from the interactive domain, reusing some components from our previous projects.[4] Components included

- several text-to-speech engines,
- a speech recognizer,
- a camera-based face detector and motion detector,
- Java 2 Micro Edition modules for interacting with mobile-device vibrations,
- a messaging system and GPS sensors,
- semantic services such as WordNet,
- extensions for the Firefox Web browser, and
- interfaces to several Web services, including the Google search service and spell checker, translation services, the Alexa statistics service, and news services.

We created simple service interfaces on top of these components but provided documentation for only a few components.

Adapting the Tools

We created some simple examples illustrating how students can use individual components and how they can interconnect several components built using diverse technologies. Several examples demonstrated the use of speech in interaction—for instance, to control Google maps or interact with Virtual Reality Modeling Language (VRML) scenes. Other examples illustrated how students could use face or motion detectors to interact with multimedia content on the Web. We also created examples that combine the Google search service and a spelling checker, the WordNet definition service, a translation service, and a text-to-speech engine. Using Web browser extensions, we created an example demonstrating how to select text from a Web page, call a translation service, and hear (through a text-to-speech engine) the text's translation.

Students found our opportunistic software development tools to be useful. They used the tools to explore and learn a particular technology as

well as to connect their components. For example, one group developed a solution in Adobe Flex that lets users interact with geographic maps through mobile devices (see Figure 2c). In addition, students used the tools as a rich test context, in which they connected their solutions to other components. One student group developed a Wii adapter and connected it to Amico to demonstrate the use of gestures in virtual and game environments already connected to Amico (see Figure 2d).

Creativity and Opportunistic Software Development

The creativity-support design principles proved to be useful guidance for organizing the course. We wanted to emphasize the importance of creativity when combining diverse technologies. Here, we describe some lessons learned, organized according to the 12 design principles.

Support exploration. We encouraged students to discover and explore new technologies. One assignment asked students to write and present a two- to three-page report about their chosen technology. Most students were eager to do such explorations, and their presentations often provoked interesting discussions. We reduced lecture time to allocate more time to student presentations. Through their explorations, students found appropriate pieces of software and discovered the functionality of poorly documented components and APIs. They generally acknowledged that the exploration extended their understanding of current software technologies' possibilities.

Low threshold, high ceiling, and wide walls. To lower the threshold, we simplified installation procedures for our tools and software components and provided examples. We still presented sophisticated technologies (that is, capable of supporting much more then "hello world" applications) with dozens of complex components and examples (high ceiling). We gave no strict limitations on the task or technology (wide walls). Creating a low threshold was our most challenging task, and it was critical for students with less development experience—for whom even setting system variables was a new task.

Support many paths and styles. Students could choose the development environment most suited to their previous knowledge—for example, less-experienced students might choose spreadsheets, whereas more-advanced students might choose scripting, declarative programming, or programming libraries. Because of the huge diversity of students' background, having environments suited for various skills was crucial in enabling all the students to produce practical results. We also encouraged students to think about novel environments that would be most appropriate for them.

Support collaboration. We encouraged students to discuss their explorations and ideas with others. We also encouraged them to work in groups of two or three on their final assignments. Most groups consisted of students with similar backgrounds. In future courses, we'll try to group students with complementary backgrounds and skills.

We encouraged students to post their work online and explore each other's work, as well as to participate in discussions on open forums dedicated to their chosen technology.

Support open interchange. Although we didn't limit students' technology choices, we encouraged them to make their solutions open and easy to integrate. This resulted in several useful modules that we'll reuse in future courses.

Make it as simple as possible—and maybe even simpler. We tried to maximally simplify use of the tools and materials. However, we failed to make our technology simple enough for all students. For example, early in the course, most comments were about setting system variables, which we had assumed the students could do without any problem. We observed that if students couldn't install and test examples the first time they tried, they would be much less enthusiastic about future assignments.

Choose black boxes carefully. Our tools use simple abstractions and data structures (untyped variables) that are easy to understand and map to most development environments. For most student examples, these structures were sufficient. However, when students wanted to combine dozens of complex services, the number of variables increased into the hundreds. They therefore had to limit the number of services they wanted to combine.

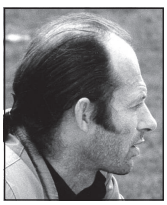
Invent things that you'd want to use. We built and used all the tools we introduced during the course. Students appreciate lectures about technologies their instructors are enthusiastic about—in our case, Amico and service composition. During the course, students asked many (unexpected) questions, which are much easier to handle if you're familiar with the tools you present.

In our course, we wanted to emphasize the importance of creativity when combining diverse technologies.

About the Authors

Željko Obrenović is an assistant professor in the Industrial Design Department of the Technical University in Eindhoven. His research interests include human-computer interaction and software engineering. Obrenović received his PhD in computer science from the University of Belgrade. He completed the work presented in this article while he worked at Centruum Wiskunde & Informatica, Amsterdam. Contact him at z.obrenovic@tue.nl.

Dragan Gašević is an assistant professor in the School of Computing and Information Systems at Athabasca University. His research interests include the Semantic Web, model-driven software engineering, knowledge management, service-oriented architectures, and learning technologies. Gašević received his PhD in computer science from the University of Belgrade. He recently received Alberta Ingenuity's 2008 New Faculty Award. Contact him at dgasevic@acm.org.



Anton Eliëns is a lecturer at Vrije Universiteit Amsterdam, where he teaches multimedia courses and coordinates the Master Multimedia for Computer Science program. He received his PhD in computer sciences from Vrije Universiteit. Contact him at eliens@cs.vu.nl.

Balance user suggestions with observation and participatory processes. We encouraged students to think about their systems' users and how their system would be useful to them. We provided some examples of accessibility. How to involve a greater number of real user issues remains an open problem, but this wasn't the course's focus.

Iterate, iterate—then iterate again. We supported rapid prototyping and encouraged students to work in small steps and to ask questions before they invested a lot of time in implementation. When students emailed questions, we responded as quickly as possible to keep their creative momentum going.

Design for designers. For the final assignment, we asked students to propose and build a working prototype. Opportunistic software development offers a huge range of development environments and components, enabling students of diverse backgrounds to build practical and complex interactive systems.

Evaluate your tools. We told the students that the course was new and experimental and that we wanted their feedback. We asked them to write about their experience using our tools, and we used this feedback to fix the bugs and improve the course material.

Although our initial results are encouraging, they're only a first step toward a more synergic mix of opportunistic software development and creativity-support tools. To encourage further development and research, we've made our course materials and software freely available and reusable by others. In the future, we'll work on applications of our framework in other courses, including introductory computer programming courses, and address problems such as providing a uniform debugging environment and documentation of diverse software resources.

References

1. C. Ncube, P. Oberndorf, and A.W. Kark, "Opportunistic Software System Development: Making Systems from What's Available," *IEEE Software*, vol. 25, no. 6, 2008, pp. 38–41.
2. F. Martin, "Toy Projects Considered Harmful," *Comm. ACM*, vol. 49, no. 7, 2006, pp. 113–116.
3. F.P. Brooks Jr., *The Mythical Man-Month: Essays on Software Engineering*, 20th anniversary ed., Addison-Wesley, 1995.
4. Ž. Obrenović and D. Gašević, "Open Source Software: All You Do Is Put It Together," *IEEE Software*, vol. 24, no. 5, 2007, pp. 86–95.
5. M. Resnick et al., "Design Principles for Tools to Support Creative Thinking," *Proc. Workshop Creativity Support Tools*, 2005; www.cs.umd.edu/hcil/CST/Papers/designprinciples.pdf.
6. B. Shneiderman, "Creativity Support Tools: Accelerating Discovery and Innovation," *Comm. ACM*, vol. 50, no. 12, 2007, pp. 20–32.
7. B. Shneiderman, "Creating Creativity: User Interfaces for Supporting Innovation," *ACM Trans. Computer–Human Interaction*, vol. 7, no. 1, 2000, pp. 114–138.
8. G. Conti, "Hacking and Innovation: Introduction," *Comm. ACM*, vol. 49, no. 6, 2006, pp. 32–36.