Supporting Creative Thinking through Opportunistic Software Development

Zeljko Obrenovic, CWI Amsterdam
Anton Eliens, VU, Amsterdam
Dragan Gasevic, Athabasca University

1. Introduction

To create an illustrative example and motivation for our article, we start with an anecdote from "To Hull and Back", a 1985 Christmas special[1] of the British sit-com "Only Fools and Horses". In this episode, main characters, Del, Rodney and Uncle Albert, decided to sail from London to Amsterdam in a hired boat. Uncle Albert, "experienced seaman", arrives to captain the boat. However, on a first serious test of his sailing skills, when they get lost during the night in the North See, it turned out that, despite spending years in the Royal Navy, Uncle Albert has no experience in navigation. His explanation is particularly interesting: during his days in the navy he was a boiler maintenance man, and he did not have to learn navigation as, "you see, the boiler has a tendency to go wherever the ship's going".

The story of Uncle Albert can often be found in software engineering practice and education. Software developers and students tend to be very skilled in particular technology, such as Microsoft.NET, Java, SOAP, PHP, or Flash, constantly following latest trends in these technologies. But, they usually face problems out of their technological space, when they have to "navigate" in a larger context and connect to systems build by others in other technologies. Universities, practitioner books and industrial training organizations, contribute to this problem, as they continue to put too much emphasis on how to create masterpieces of code from blank sheets of paper, ignoring technological issues or treating them in isolation, and largely ignore the very different skills required to integrate and test software that students did not create and do not control.

In this paper we describe our experiences in using opportunistic software development to fight the "boiler maintenance men syndrome". We created a didactic method, supported with a set of tools, where we encourage students to be creative and innovative, and develop solutions that cross boundaries of different technologies. *By teaching students to combine systems that were never meant to work together or even to be reused, we created a space where students produced many innovative ideas and solutions.*

We first describe some of the existing novel educational approaches and identify new requirements for software engineering education. We then present our framework for opportunistic software development education, and describe how it can be used to support new requirements for software engineering education. After that, we present a case study of application of our framework in the course on intelligent human-computer interaction design, where we also discuss some lessons learned.

2. Rethinking Software Engineering Education

Software engineering education needs to teach students how to face real problems and think creatively to find innovative solutions to growing users' needs. In many cases, however, students finish their studies without being exposed to such problems. Many of the computer science curriculum have identified this gap between typical computer science education and software engineering practice, and have used different novel approaches to teach students the skills that are closer to software engineering

1 http://en.wikipedia.org/wiki/To_Hull_and_Back

practice.

The need for having more realistic educational context for software engineering, has recently been argued by Fred Martin [9]. He noted that “toy examples” represent current state of the practice in software engineering education, but that teaching should be more realistic, interactive and collaborative. He proposes creating a context where students’ code matters, and where student develop solutions that has value beyond passing the exams. Chuan-Hoo Tan an Hock-Hai Teo also argue that giving students experience developing and delivering large-scale systems under time constraints and shifting deadlines can better prepare students for future challenges [10].

To make software development education more engaging and realistic, several universities have started to introduce courses on *games development*. Kajal and Mark Claypool, for example, argue that many projects currently used in Software Engineering curricula lack both the “fun factor” needed to engage students, as well as the practical realism of engineering projects that include other computer science disciplines such as Networks, or Human Computer Interaction [8]. They show that use of computer game-based projects can enhance interest and retention in existing Software Engineering curriculums. Parberry et al. explored how game programming can be used with art students, arguing that a key feature of such approach is the opportunity for diverse communities of students to collaborate on joint projects [5]. Tsai et al. also applied game design in education of art/design students. They report that the students where able to make complete games, not just oversimplified exercises or simple walk-through scenes, and they built the fundamental programming knowledge enough to take intermediate programming courses for their future careers [7].

The Rethinking CS101 Project[2] identified outdatedness of most of introductory programming courses, which usually teach computation as sequential problem-solving. To oppose this view, the project emphasis that perhaps the most fundamental idea in modern computer science is that of interactive processes. Computation is embedded in a (physical or virtual) world; its role is to interact with that world to produce desired behaviour. They state that by beginning with a decomposition in terms of interacting computational processes, we can teach the students a model of the world much closer to the one that underlies the thinking of most computer professionals. The projects provides a set of educational resources to teach interaction in introduction courses, focusing on using Java.

3. New Requirements for Software Engineering Education

Described approaches identified significant gap between software engineering education and practice. To reduce this gap, they promote a design-based education, where students learn through building concrete and more realistic solutions, going beyond toy examples. Although presented educational approaches use different means, we can identify three important requirements that such approaches introduce:

- Providing more realistic and engaging development context, i.e. providing the context and development experience where student programming matters, exposing students to real issues in project and team management. By solving real problems, and developing code that can be used by someone else, students can get more realistic motivation for their work. Having deadlines and working with others, teach students to frame problems and solutions more realistically and work in teams.
- Enabling students to develop their own design, encouraging rapid prototyping. There is always a balance between providing assignments that are focused on particular curricular topics and ones that are more open-ended. Assignments that support student design can be particularly

2 http://www.cs101.org/

successful in encouraging students to become more invested in their work [9]. Developing their own design, using rapid prototyping techniques, can help students to understand the ideas they are working with.

- Usage of didactic method that supports creative thinking. Students have to learn not only details of particular technology, but also creative ways of applying this technology. A benefit of design-based projects is that students are welcome to take advantage of whatever resources they can marshal, including using code located on the Internet. (As non-commercial users, even more code will be available to them.) By definition, their design is unique, and they will get further by leverage published code and other work. Encouraging students to be creative in reframing their problem based on their developing insights along the way. Collaboration and public performance are also often introduced to enable students to exchange ideas, and get feedback on their work.

4. A Framework for Opportunistic Software Development Education

In order to support identified educational requirements, we have started to build an educational framework based on ideas from opportunistic software development. Opportunistic software systems development (OSSD) is a very good candidate for supporting new requirements for software engineering education, as it emphasis creativity, innovation, and imaginative ways of finding and gluing software to meet diverse users' needs [ref.]. In this way, students can get useful practical experience in agile and non-conventional problem solving methods [ref.].

Our framework consists of a set of tools and guidelines that can help educators to teach students to be more creative and innovative. We develop this framework building on our previous work in opportunistic software development with open-source software [Obrenovic07]. One of our goals is to promote broader reuse of open-source software (OSS) in education. OSS provides a rich context for opportunistic software development and education, as we can reuse thousands of freely available and functional software components from numerous domains. The diversity of functionalities, technologies, and development approaches used by OSS community, provide a ground for creating advanced and innovative solutions. The main challenge is to solve interoperability problems that such diversity introduces. We provide a pragmatic solutions that can improve the interoperability of OSS, and enable rapid prototyping with diverse OSS components [Obrenovic07, also see http://amico.sourceforge.net].

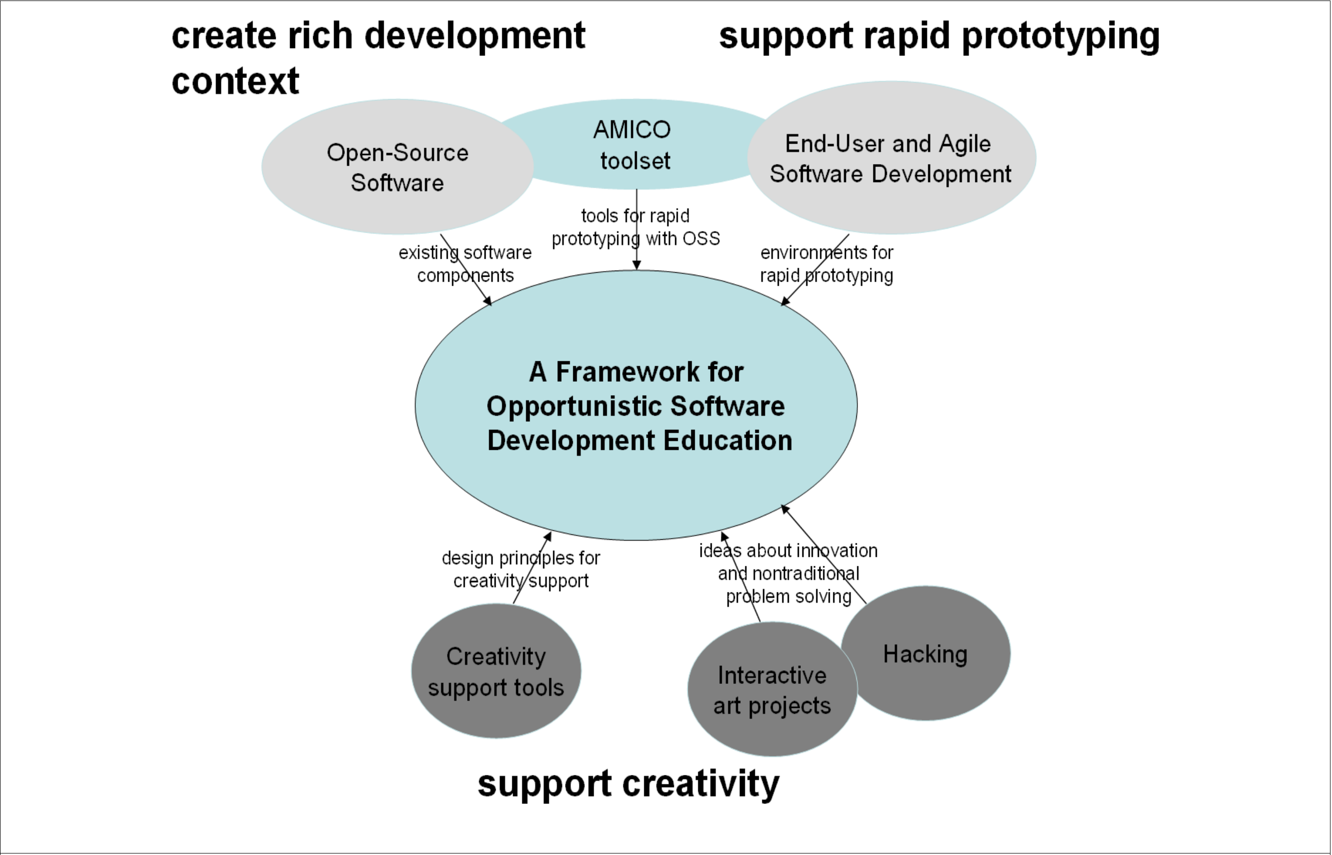

create rich development context
support rapid prototyping
Open-Source Software
AMICO toolset
End-User and Agile Software Development
existing software components
tools for rapid prototyping with OSS
environments for rapid prototyping
A Framework for Opportunistic Software Development Education
design principles for creativity support
ideas about innovation and nontraditional problem solving
Creativity support tools
Interactive art projects
Hacking
support creativity

Figure 1. *A framework for opportunistic software development education.*

As part of our educational framework, we provide:

- A set of tools to facilitate educators and students in creating a rich and engaging opportunistic software development context from a range of diverse OSS components,
- Several development environments to support rapid prototyping in such context, including spreadsheets, Web browser extensions, and declarative languages, aimed at students with different levels of expertise,
- Guidelines to support creative thinking.

In the rest of this section, we describe how our educational framework supports new requirements for software engineering education.

4.1 Creating a Rich Development Context from OSS

The main goal of our educational framework is in creating a context where students can focus on higher-level and innovative software composition with advanced components. To enable creating of rich opportunistic software development context from range of diverse OSS components, we have developed a pragmatic approach to OSS integration, supported with a middle ware platform that can facilitate interconnections among diverse technologies [Obrenovic07].

We have adopted a service-oriented approach to OSS integration: we run OSS components as standalone applications that offer their functionality through open communication interfaces. Building, installing and running standalone programs is usually a straightforward activity, even for users who are not familiar with the technology used by the component (i.e., even though you do not know Python or

Java, it is still relatively easy to install Python or Java interpreters, and build and run their applications). In this way applications written in one languages can easier use component written in another language (e.g. you can expose the functions of your component with Python OSC server library, which can be accessed by other components through Java or C++ OSC client libraries). Turning OSS component to standalone services does not require changing the basic functionality of the component, but adding the code that offers component functionality through any of the open communication interfaces, such as XML-RPC, OSC, SOAP, or some application specific TCP or UDP interface. As a starting point for these adaptations, we have often used examples and demo programs that come with OSS distributions.

As part of our framework, we include our AMICO middleware platform (see amico.sourceforge.net) which provides a common space where diverse services can be interconnected. The main innovation of AMICO is in using a great number of communication interfaces, such as low-level TCP and UDP interfaces, Bluetooth, as well as many higher-level interfaces such as HTTP GET/POST, XML-RPC, OSC, SOAP**,** and many application specific interfaces. Our platform is extensible and allows for addition of new service interfaces. AMICO enables much easier reuse of existing services and components, as they do not have to be adapted to a common interface, i.e. adaptation can be made using the technology of the component, and developers and students can chose the interface they are most familiar with [2].

By turning OSS components into services, and connecting them to our middleware, we create a rich context, where you can inter-connect many diverse components.

4.2 Tools for Rapid Prototyping with OSS

On top of our middleware, we have supported several development environments that can be used to support diverse users in development with OSS components. We do not limit students in their choice of development technology, and we provide several solutions, including:

- *Spreadsheets*, aimed at students without or with limited programming skills,
- *Web browser scripting*, aimed at students with Web development experience, including AJAX and applet support, and browser extensions
- *Declarative programming mashups*, aimed at students with experiences in declarative programming languages, including XML based configuration files, and Prolog.
- *Programming libraries,* aimed at students with experiences in procedural and object-oriented languages.

The motivation behind having multiple development environments is similar to a multi-layered user interfaces philosophy [5]: *enabling users to start with a basic and simple development environments, such as spreadsheets, and "move up" by switching to more advanced mashup interface as their expertise develops and when more complex functionality is needed*. For example, in our course, students were using spreadsheets in the beginning to quickly sketch, discuss and evaluate prototypes of interactive systems, and then switched to declarative and procedural programming and Web browser extensions to create more complex solutions. There are also examples of various tools that follow this design philosophy: many video games have dozens of layers, most search engines have novice and advanced layers (Google, Yahoo), many art and video tools have three or more workspaces (Apple Final Cut Pro, Adobe Premiere), and some tools have as many as eight layers to accommodate a wide range of expertise and ambition.

4.3 Principles of Creativity Support Tools as a Didactic Method

Many of the design-based courses implicitly support creativity thinking. We wanted, however, to have a more structured set of guidelines to support creative and innovative thinking of students. After reviewing the experience of others, we decided to reuse the design principles for tools that support creativity thinking defined by Resnick et al (see Sidebar 1). In section 4, we describe how we have applied these principles in our course on intelligent multimedia technology.

We also promote usage of examples form interactive art projects and hacking [12], to give students ideas about innovation and nontraditional problem solving. As a source for this examples, we used electronic art conferences, such as, ARS Electronica (www.aec.at), DEAF (www.deaf07.nl), and ACM Multimedia Interactive Arts track, as well as hacking conferences, such as, Blackhat (www.blackhat.com), or Chaos Computer Conference (www.ccc.de).

4.4 How to Apply the Framework

Our aim is to create a generic environment that can be adapted to different courses in different domains. In order to adapt this framework to a concrete course, we propose following steps:

- *Defining rich development context*, by selecting representative OSS projects and services, and adapting them, and connecting them to our infrastructure according to proposed service-oriented approach.
- *Create illustrative examples*, in each of the development environments that you plan to use during the course, illustrating how each component can be used, and how components can be interconnected. Make all components and examples open-source and available online.
- *Define course objectives and assignments*, using the design principles for creativity support tools as a guideline, and create necessary resources to support these objectives, including, mailing lists, links to inspiring projects, and web site where students can share their design and notes.

Sidebar 1: Design Principles for Tools to Support Creative Thinking

While experience across domains is diverse, several authors have identified underlying principles to guide design of creativity support tools [1]. What distinguishes these principles from other user interface principles is that they emphasize easy exploration, rapid experimentation, and fortuitous combinations that lead to innovations.

1. **Support exploration.** An important requirement for creativity is to be able to try out many different alternatives. Almost by definition, creative work means that the final design is not necessarily known at the outset, so users must be encouraged to explore the space.
2. **Low threshold, high ceiling, and wide walls.** Effective tool designs should make it easy for novices to get started (low threshold) but also possible for experts to work on increasingly sophisticated projects (high ceiling) [Myers 2000], and support and suggest a wide range of explorations (wide-walls).
3. **Support many paths and many styles.** This principles put a high priority on supporting learners of all different styles and approaches, and on paying special attention to make sure that technologies and activities are accessible and appealing to the diverse users and students.
4. **Support collaboration.** In projects, in schools, and in the "real world," most creative work is done in teams. An important implication is the need to provide support for collaboration in the tools.
5. **Support open interchange.** The creative process will not usually be supported by a single tool, but rather will require that the user orchestrate a variety of tools each of which supports part of the task**.** Creativity support tools should seamlessly interoperate with other tools. This includes the ability to easily import and export data from conventional tools such as spreadsheets, word processors and data analysis tools, and also with other creativity support tools. This requires that the data formats in the files be open and well-defined.
6. **Make it as simple as possible—and maybe even simpler.** Technology-based products have become more and more complex. One reason is "creeping featurism": advances in technology make it possible to add new features, so each new generation of products has more and more features. This trend is reinforced by the belief among marketing professionals that it's quite hard to sell a product as "simpler," but much easier to sell it as "containing more features."
7. **Choose black boxes carefully.** In designing creativity support tools, one of the most important decisions is the choice of the "primitive elements" that users will manipulate. This choice determines, to a large extent, what ideas users can explore with the tool – and what ideas remain hidden from view.
8. **Invent things that you would want to use yourself.** At first blush, this design principle might seem incredibly egocentric. And, indeed, there is a danger of over-generalizing from your own personal tastes and interests. But we have found that we do a much better job as designers of creativity support tools when we ourselves really enjoy using the tools that we are building. We feel that this approach is, ultimately, more respectful to users of the technology. Why should we impose on users systems that we don't enjoy using ourselves?
9. **Balance user suggestions with observation and participatory processes.** Most successful designers seek to understand their users, in order to design products well-matched to the needs of their users. They invest considerable time observing and interviewing users, talking with focus groups, asking users for suggestions and feedback on features, and inviting users to participate in design processes.
10. **Iterate, iterate—then iterate again.** Another standard principle of user interface design that we would like to re-emphasize for creativity support tools is the importance of iterative design using prototypes. In designing creativity support tools, we put a high priority on "tinkerability" – we want to encourage users to mess with the materials, to try out multiple alternatives, to shift directions in the middle of the process, to take things apart and create new versions.
11. **Design for designers.** By creating you become creative. In designing new creativity support tools, it is important to design for designers – that is, design tools that enable others to design, create, and invent things (see also this report's section titled "Creativity Support Tools for and by the New Media Arts Community" for a further discussion)
12. **Evaluate your tools.** While the rigor of controlled studies makes them the traditional method of scientific research, longitudinal studies with active users for weeks or months seem a valid method to gain deep insights about what is helpful (and why) to creative individuals.

References:

1. M. Resnick, B. Myers, K. Nakakoji, B. Shneiderman, R. Pausch, T. Selker, M. Eisenberg, "Design Principles for Tools to Support Creative Thinking", http://www.cs.umd.edu/hcil/CST/Papers/designprinciples.pdf, October 30, 2005.

5. Case Study: Everything you always wanted to develop...

We applied our framework and tools in the course on intelligent multimedia technology[3] With the subtitle “Everything you always wanted to develop...”, the course was teaching students how to organize intelligent dialogues between the user and complex systems, such as virtual environments and multimedia Web applications. The course went beyond direct control and conventional interaction based on the mouse and keyboard, and introduced additional interaction modalities such as speech input and output, and camera based user sensing[4]. The main focus was on the practical work of the students, where at the end of class, they had to delivery application and write report about it.

The course consisted of our lectures, students presentations of technologies and ideas, and student individual work in a laboratory and at home. Our lectures focused on integration patterns that show how different technologies can work together, while students explored and presented particular technologies. The course was realized with 32 undergraduate students (third and fourth year), from various departments, including cognitive systems, information systems, computer sciences, artificial intelligence, and several exchange students.

Due to a huge diversity of used technologies and students backgrounds, the course provided a very good environment for applying and evaluation of our educational framework.

5.1 Defining the Development Context

We selected a number of open-source components and services from the interactive domain, reusing some of the components from our previous projects [Obrenovic07]. The list of components includes several text-to-speech engines, speech recognizer, a camera based face detector and motion detector, J2ME modules for interaction with mobile device vibrations, messaging system and GPS sensors, semantic services such as WordNet, extensions for Firefox web browser, and interfaces toward several Web services, including Google search service and spelling checker, translation services, Alexa statistic service, and news services. We created a simple service interfaces and documentation for these components.

5.2 Adapting the tools, and creating examples

We created simple examples that illustrate how individual components they can be used, as well as a number of examples that illustrate how several of these services can be interconnected. Several examples demonstrated usage of speech in interaction, for instance, to control of Google maps, and or interaction with VRML scenes. Other examples illustrated how you could use face or motion detector to interact with multimedia content on the Web. We also created examples that combine Google search service and spelling checker, with WordNet definition service, translation service, and text-to speech engine. With Web browser extensions, we created an example that shows how you can select a text from a Web page, call translation service, and hear (through TTS engine), translation of it. These examples mixed very diverse technologies, and students could use them to learn service usage, and to build on.

5.3 Supporting Creativity

We organized our course and assignment according to the 12 design principles of creativity support. These principles proved to be a useful guidance, and here we describe some lessons learned.

3 http://www.cwi.nl/~obrenovi/teaching/imt/

4 See http://amico.sourceforge.net/amico-demos.html for illustration.

Low threshold, high ceiling, and wide walls. In order to create a lower-threshold, we simplified installation procedures for our tools and used OSS components, and provided simple documentation and example material. We were still giving technology that can be used for building much more then "hello world" applications, with dozens of complex components and examples (high ceiling). We provided no strict limitation about the task or technology (wide walls). Creating low threshold was the most challenging and critical for students with less experience in development, for which even setting system variables was a new task.

Support exploration. Students where encouraged to look for new technologies, and explore its possibilities. As one of the assignments students had to write a 2-3 page report about chosen technology, and present it to the class. Most of the students were very eager to do such explorations, and their presentations often provoked very interesting discussions. We reduced the time of our lectures, to give more time for such presentations.

Support many paths and many styles. Students could chose any of the development environments, suited for their previous knowledge, including, spreadsheets (for less experienced students), scripting, declarative programming, or programming libraries. Due to 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 most appropriate for them.

Support collaboration. We encouraged students to present and discuss with others their explorations and ideas. In some cases, presentation of technology discovered by one student, inspired the others to use this technology for their final assignment. Students were also encouraged to do their final assignments in groups of two or three. We also encouraged students to put everything online, and explore each others works.

Support open interchange. Even though we did not limit students in the technologies they want to use, we encouraged them to make their solutions open, and easy to integration with others. This resulted in several useful modules, that we will be able to reuse in future courses.

Make it as simple as possible—and maybe even simpler. We tried to maximally simplify usage of our tools and materials. However, we failed to make our technology simple enough for all the students. For examples, most of the comments at early stage of the course were about setting system variables, which we assumed that the students will be able to do without problems. Our observation is that if students where not able to install and test examples first time they attempt to do it, they will be much less enthusiastic and late in their assignments.

Choose black boxes carefully. Our tools use simple data structures (untyped variables), that are easy to understand, and easy to map to most of development environments. For most of the examples build by students, these structures were sufficient and easy to use. The problem appeared when students wanted to combine dozens of complex services, as they then had to work with hundreds of variables.

Invent things that you would want to use yourself. All of the tools that we introduced during the course were used and built by us. Students appreciate lectures about technologies you are enthusiastic about. Also, during the course, we had lots of (unexpected) questions and challenges from students, and it is much easier to handle them if you are very familiar with the tools you present.

Balance user suggestions with observation and participatory processes. We encouraged students to think about users of their systems, i.e. to describe why their application is useful, and to discusses it with others. We provided some examples in area of accessibility. However, it remains an open problem how to involve more real user issues, although that was not focus of our course.

Iterate, iterate—then iterate again. We supported rapid prototyping, encouraged students to work in small steps and ask questions before they invest significant amount of time in implementation. When students send questions by email, it is important to give answer as soon as possible and encourage them, to keep their creative momentum going.

Design for designers. Emphasis was on building practical projects, and for their final assignment students have to propose and build a working prototype. Opportunistic software development provided a very nice context for enabling students of very diverse backgrounds to build practical and complex interactive systems, as they can chose from huge range of development environments and available components.

Evaluate your tools. We announced to the students that the course is new and experimental, and that we want their feedback and participation. We asked them to write assignment about their experience in using our tools, and we used this feedback to fix the bugs, and improve the course material.

5.4 Results

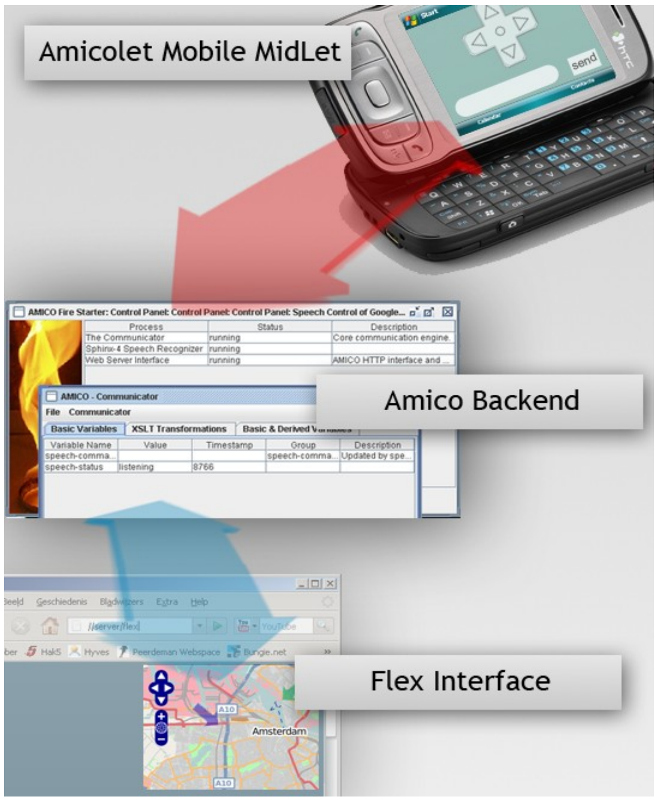
We were positively surprised with student participation and results. Students demonstrated their ability to creativity combine very different technologies. They proposed a number of very innovative solutions, exploring huge range of technologies, and their proposals usually combined several of them. For examples, one project implemented World Time Mashup, combining MS Silverlight plugin, Google Maps AJAX component, and Earthtools.org web service (Figure 2b). Most of the students learned and first time used the technologies during the course.

Our opportunistic software development tools have also been useful for most of the students. In general students used them in three ways:

- *As a exploration tool*, to explore and learn particular technology.
- *As a main platform for their solutions*, connecting components they have build. For example, one group developed a solution that enables interaction through mobile devices with geographic maps in Flex, where they have used AMICO to connect Adobe Flex component, Openstreet mapdata Web services, and J2ME mobile midlets (Figure 2a).
- *As a rich test context*, where their solutions could be connected with other components. For example, one student group developed a Wii adapter (Figure 2c), and connected it to AMICO demonstrating it usage in virtual and game environments already connected to AMICO.

Students positively ranked the course, with average mark “rather good”[5]. One of the encouraging feedbacks is that they would like to have similar courses more often. The main negative feedback was about the lack of course material, as we were partially building it and testing it during the course, so online page about the course, for example, was sometime outdated.

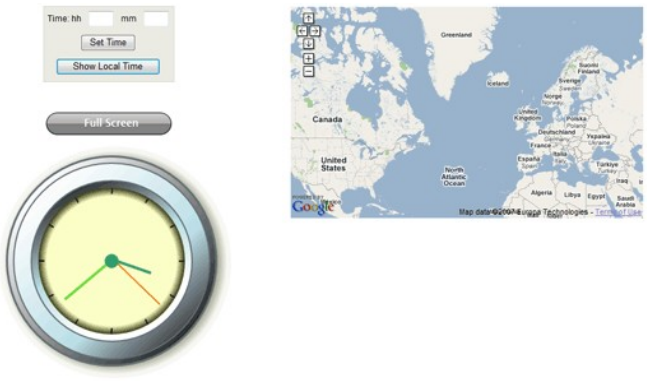
5 On a scale from 1 to 5: 1 – very bad, 2 – fairly bad, 3 – neutral, 4 – rather good, 5 – very good



Process	Status	Description
The Communicator	running	Core communication engine.
Sphinx-4 Speech Recognizer	running	
Web Server Interface	running	AMICO HTTP interface and ...

Variable Name	Value	Timestamp	Group	Description
speech-comma...			speech-comma...	Updated by spe...
speech-status	listening	8766		

(a)



(b)

(c)

Figure 2. *Examples of students' projects: (a) mobile device interaction with maps, (b) world time mashup, and (c) Wii device adapter.*

6. Conclusion

With more people using computers and computers being more diverse and pervasive, software development requires more and more innovative approaches. We need to teach students to be creative in exploring the space of possible solutions, and finding right solution. Opportunistic software development can provide an incredibly rich environment for stimulating creativity and innovation.

In addition to supporting educators with practical tools, our solutions can support diverse forms of rapid application development, and enable broader reuse of OSS components. To support our open-source policy, all course materials and software are freely available, and may be reused by others.

In our future work we will work on applications of our framework in other courses, including introductory courses.

References

1. Shneiderman, B. 2007. Creativity support tools: accelerating discovery and innovation. *Commun. ACM* 50, 12 (Dec. 2007), 20-32.

2. Shneiderman, B. 2000. Creating creativity: user interfaces for supporting innovation. *ACM Trans. Comput.-Hum. Interact.* 7, 1 (Mar. 2000), 114-138.

3. Smith, D. K., Paradice, D. B., and Smith, S. M. 2000. Prepare your mind for creativity. *Commun. ACM* 43, 7 (Jul. 2000), 110-116.

4. Knuth, D. E. 1974. Computer programming as an art. *Commun. ACM* 17, 12 (Dec. 1974), 667-673.

5. Parberry, I., Kazemzadeh, M. B., and Roden, T. 2006. The art and science of game programming. In Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education (Houston, Texas, USA, March 03 - 05, 2006). SIGCSE '06. ACM, New York, NY, 510-514.

6. Ursyn, A., Scott, T., Hobgood, B. R., and Mill, L. 1997. Combining art skills with programming in teaching computer art graphics. *SIGGRAPH Comput. Graph.* 31, 3 (Aug. 1997), 60-61.

7. Tsai, M., Huang, C., and Zeng, J. 2006. Game programming courses for non programmers. In *Proceedings of the 2006 international Conference on Game Research and Development* (Perth, Australia, December 04 - 06, 2006). ACM International Conference Proceeding Series, vol. 223. Murdoch University, Murdoch University, Australia, 219-223.

8. Claypool, K. and Claypool, M. 2005. Teaching software engineering through game design. In *Proceedings of the 10th Annual SIGCSE Conference on innovation and Technology in Computer Science Education* (Caparica, Portugal, June 27 - 29, 2005). ITiCSE '05. ACM, New York, NY, 123-127.

9. Martin, F. 2006. Toy projects considered harmful. *Commun. ACM* 49, 7 (Jul. 2006), 113-116.

10. Tan, C. and Teo, H. 2007. Training future software developers to acquire agile development skills. Commun. ACM 50, 12 (Dec. 2007), 97-98.

11. Fischer, G., Giaccardi, E., Ye, Y., Sutcliffe, A. G., and Mehandjiev, N. 2004. Meta-design: a manifesto for end-user development. *Commun. ACM* 47, 9 (Sep. 2004), 33-37.

12. Conti, G. 2006. Hacking and Innovation: Introduction. *Commun. ACM* 49, 6 (Jun. 2006), 32-36.