MISSIVE

Programming Environments
Computer Science 233, Fall 2010
Steven P. Reiss

1.0 Overview

The goal of this course this semester is to understand the problems inherent in providing next-generation programming tools as part of an integrated development environment or a programming environment (I'll use these terms interchangably.)

To achieve this goal we are going to start by considering the purpose of a programming environment, the problems of today's environments, and what an ideal environment should do. Then we will consider different areas in which programming tools and environments can be helpful. For each of these areas we will look at the state of the art and what can and should be done.

2.0 Course Mechanics

The course will be broken down by topic. After a few initial classes where we will attempt to set the framework for further discussions, we will move to individual topics that are of interest to the students in the class. For each topic, we will consider the state of the art in terms of what can be done, what tools currently do, and how these tools are integrated into current environments.

We will spend two to three classes on each topic. This will consist of a presentation of two or more related papers. The presentations should provide an overview of the topic and then describe the current state of the art. Following these presentations we will have a class discussion of what should or could be done related to this topic as part of an integrated development environment. This discussion should cover the strengths and weaknesses of the presented papers, other approaches that have been taken or that could be taken, and user experience with the tools.

Nominally, each student will be in charge of a topic and hence a week's worth of class. However, students are encouraged to work in teams so that a team is responsible for two separate topics and the load is split accordingly. Being in charge of a topic will include giving two interactive talks and leading the follow-up discussion. It also means assigning a paper that may be one of the papers being presented but doesn't have to be. For example, the assigned paper might be overview of the problem or that plus an earlier solution — something that would provide a lead-in to the presented papers. Students who are not doing the presentation are responsible for reading this paper, asking questions, and generally interacting with the presentation. This means that the presenters can assume that the students have read (and possibly even understood) the assigned paper.

In addition, those taking the course for credit are expected to produce a prototype software tool that addresses one of the topics. This can be done either as a standalone tool, as an Eclipse plug-in or RCP, or as a Code Bubbles plug-in. Students will be expected to do two formal presentations on the project; the first, about a third of the way through the course, will be an initial design presentation where the student will describe what they propose and will solicit feedback from the rest of the class. The second presentation will be a final presentation at the end of the course showing the tool in action.

3.0 Possible Topics

(These are given in no particular order).

Collaboration

More and more software is being developed by teams of programmers, teams that are often globally distributed. How can and should environments support this.

Papers

Jazzing up Eclipse with Collaborative Tools (Cheng, Hupfer, Ross, Patterson)

How a good software practice thwarts collaboration: the multiple roles of APIs in software development (de Souze, Redmiles, Cheng, Millen, Patterson)

Configuration management

Configuration management has not progressed that much since SCCS in the 70s. RCS, CVS, SVN, git, etc. are all minor improvements (or are they). Make and ant aren't much better. What should be done here?

Papers

Operation-based conflict detection and resolution (Koegel, Helming, Seyboth)

Safe-commit analysis to facilitate team software development (Wloka, Ryder, Tip, Ren)

Debugging

Debugging technology is rooted in the 60's with minor enhancements for handling threads and some tools for memory debugging. What should debugging actually look and feel like. What would be helpful.

Papers

Debugging reinvented: asking and answering why and why not questions about program behavior (Ko, Myers)

Selective capture and replay of program executions (Orso, Kennedy)

Isolating cause-effect chains from computer programs (Zeller)

A dataflow language for scriptable debugging (Marceau, Cooper, Krishnamurthi, Reiss)

Design and Model-Driven Development

There is a movement to use UML as a high-level language for programming by having the system generate code from the UML model and having programming be done by refining the model and regenerating code. Is this a viable option? Is UML, which is widely hated by programmers, a suitable representation? Is anything better.

Papers

Model-driven development (Mellor, Clark, Futagami)

User Interface Building

One of the more difficult aspects of writing code today involves designing and implementing user interfaces. Some programming environments provide tools to help the programmer do this, but these are generally difficult to use and don't

handle particularly well the cases where the user interface depends on the state of the program. What should user interface tools look like?

Papers

Building user interfaces by direct manipulation (Cardelli)

Authoring sensor-based interactions by demonstration with direct manipulation and pattern recognition (Hartmann, Abdulla, Mittal, Klemmer)

The state of the art in automating usability evaluation of user interface (Ivory, Hearst)

Automatic generation of personalizable user interfaces with Supple (Gajos, Weld, Wobbrock)

Testing

Agile development has spurred an interest in testing, although testing is a long-standing topic of interest in program development. Testing raises several issues that are relevant to programming environments: how to generate test cases, how and when to run test cases, and how to interpret the results of test cases.

Papers

Continuous testing in Eclipse (Saff, Ernst)

Visualization of test information to assist fault localization (Jones, Harrold, Stasko)

On test suite composition and cost-effective regression testing (Rothermel, Elbaum, Malishevsky, Kallakuri, Qui)

Test generation through programming in UDITA (Gligoric, Gvero, Jagannath, Khurshid, Kuncak, Marinov)

Random test data generation for Java classes annotated with JML specifications (Cheon, Rubio-Medrano)

Tool-assisted unit-test generation and selection based on operation abstractions (Xie, Notkin)

Scripting Languages

Most programming environments are designed to handle traditional programs with some sort of write-compile-run cycle. What is or should be different in dealing with scripting languages such as Scheme, MatLab, Python, etc.

Papers

DrScheme: A Programming Environment for Scheme (Findler, Clements, Flanagan, Flatt, Krishnamurthi, Steckler, Felleisen)

Web Applications

More programs today are being written as web applications meaning that they consist of different pieces, often written in different languages, and designed to be run in different environments (e.g. web browser, tomcat, web server, ...). What can or should a programming environment do to simplify and support the development of such applications.

Papers

TBD

Code Search

There are over 1 billion lines of code available in open-source repositories today. Given this, it seems that almost anytime a programmer wants to write some code, that code (or something like it) has probably already been written by someone else. How can a programming environment help locate and adapt such code to let programmers avoid duplicating other's efforts.

Papers

Semantics-based code search (Reiss)

Lowering the barrier to reuse through test-driven search (Janjic, Stoll, Bostan, Atkinson)

Static Analysis

One of the major improvements in programming tools over the last decade has been the advent of static analyzers that can find real or potential bugs without having to execute the program. The wide range and applicability of these tools is impressive and is growing.

Papers

ESC/Java2: uniting ESC/Java and JML (Cok, Kiniry)

The spec# programming language: an overview (Barnett, Leino, Schulte)

Semantic essence of AsmL (Gurivich, Rossman, Shulte)

Alloy: a lightweight object modeling notation (Jackson)

Model checking Java programs using Java pathfinder (Havelund, Pressburger)

An efficient inclusion-based points-to analysis for strictly typed languages (Whaley, Lam)

A practical flow-sensitive and context-sensitive C and C++ memory leak detector (Heine, Lam)

Dynamic Analysis

Static analysis is limited in that most interesting problems are unsolvable or at best exponential. A viable alternative is to do some sort of dynamic analysis where the program is checked as it is run. Dynamic analysis can be used to study very complex systems and to look at issues beyond simple correctness or bugs such as performance issues.

Papers

Applying "design by contract" (Meyer)

The paradyn parallel performance measurement tools (Miller, Callaghan, et al.)

X-Trace: a pervasive network tracing framework (Fonseca, Porter, Katz, Shenker, Stoica)

Dynamic instrumentation of production systems (Cantrill, Shapiro, Leventhal)

The Daikon system for dynamic detection of likely invariants (Ernst, Perkins, et al)

Teaching Environments

Most programming environments are designed to handle large systems written by sophisticated programmers. These environments are often much too complex to be used effectively by students in introductory programming courses or by stu-

dents learning a new language. A teaching environment might provide students with better errors, additional help, intelligent completions, etc.

Papers

Software visualization in teaching at Brown University (Bazik, Tamassia, Reiss, van Dam)

Working sets

Programmers typically work on one particular task at a time. This task is often cross-cutting, involving multiple functions and fields spread out over multiple files. Today's programming environments provide little support for such task-based development. What can be done?

Papers

How effective developers investigate source code: an exploratory study (Robillard, Coelho, Murphy)

Using task context to improve programmer productivity (Kersten, Murphy)

Code bubbles: rethinking the user interface paradigm of integrated development environments (Bragdon, Reiss, et al.)

Multi-core programming

Parallel or multiple core programming is inherently more complex than simple, single-threaded coding. What can programming environments do to simplify the task.

Papers

Eraser: a dynamic data trace detector for multithreaded programs (Savage, Burrows, Nelson, Sobalvarro, Anderson)

Static deadlock detection for Java libraries (Williams, Thies, Ernst)

4.0 Syllabus

Week 1: (9/1, 9/3)

Course introduction
Assignment of students to topics
Discussion of the problems with today's environment
Discussion of what the ideal tools would be

Week 2: (9/6, 9/8, 9/10)

History of programming environments
Current work at Brown

Week 3: (9/13, 9/15, 9/17)

First Topic of discussion

Week 4: (9/20, 9/22, 9/24)

No class Monday/Friday
Wednesday: Project proposals

Week 5: (9/27, 9/29, 10/1)

Second Topic of discussion

Week 6: (10/4, 10/6, 10/8)

Third Topic of discussion

Week 7: (10/11, 10/13, 10/15)

No class Monday
Fourth topic

Week 8: (10/18, 10/20, 10/22)

Fifth topic

Week 9: (10/25, 10/27, 10/29)

(Possibly no class Monday)
Sixth topic

Week 10: (11/1, 11/3, 11/5)

Seventh topic

Week 11: (11/8, 11/10, 11/12)

(Possibly no class Monday)
Eighth topic

Week 12: (11/15, 11/17, 11/19)

Ninth topic

Week 13: (11/22, 11/24, 11/26)

No class Wednesday, Friday.
Monday: TBD

Week 14: (11/29, 12/1, 12/3)

Tenth topic

Week 15: (12/6, 12/8)

Project Presentations