Opportunistic Programming: Rapid Ideation and Prototyping in Practice

Joel Brandt, Joel Lewenstein, William Choi, Philip Guo, Scott Klemmer

What is it?
Non-trivial software systems are constructed with little to no upfront planning about implementation details, and ease and speed of development are prioritized over code robustness and maintainability.

This is not simply “sloppy programming”; instead, this approach enables prototyping, ideation, and discovery, tasks that benefit significantly from being done rapidly, and are often best accomplished by building a functional piece of software.

Examples
- A user interface researcher rushes to create as many quick-and-dirty prototypes as possible
- A museum exhibit designer explores many different directions to convey a scientific concept.
- A physicist writes a complex real-time system to collect massive amounts of data during an experiment that she can only run once
- A professional C++ programmer decides to create a simple web application in his spare time to share bookmarks with his friends.

Characteristics of Opportunistic Programming

Build From Scratch
Use of high-level tools that map closely to the task at hand and build their systems from scratch by “gluing” these tools together.

Consider Code Impermanent
Often code will be used only once or never. Also, code is used to ideate and explore the design space when prototyping, a kind of “breadth-first” programming where many ideas are thrown away early. Can lead to “code satisficing.”

Iterate Rapidly
Short edit-debug cycles:

Copy-and-Paste Coding
Writing code by iteratively searching for, copying, and modifying short blocks of code (~ 30 lines) with desired functionality.

Unique Debugging Challenges
By employing a federation of languages, programmers often cannot make effective use of sophisticated debugging tools intended for a single language. Also, because there is little or no upfront design, pieces of the system often do not have clean interfaces.

Tools for Opportunistic Programming

1. Interactively define a function, getting real-time execution feedback on each line, inside an IDE
2. Better integration of copied code into existing code
3. Visually and interactively follow code execution flow and state changes

Histogram of per-subject edit-debug cycle times in our laboratory study. Total number of edit-debug cycles for each subject are given by the black number on each bar, and bar length is normalized across subjects. A black line separates cycles of less than and greater than 5 minutes in length.

96 93 28 98 74 73 56 72 78 76 52 128 39 72 81 33 246 65 43 97 0% 20% 40% 60% 80% 100% < 10 sec 10-30 sec 30-60 sec 1-2 min 2-5 min 5-10 min 10-30 min > 30 min

Stanford University HCI Group
http://hci.stanford.edu/opportunistic/