Conversation Starter: Imagining Autotelic IDEs

Kate Compton
Department of Computer Science
Northwestern University

Abstract

We know what to expect from an IDE, right? We will type some code, the code will be executed, we will observe the output, and iterate as many times as needed. The IDE will assist us in that goal with reliable syntax highlighting, code completion, and debugging tools, in order to be the most demure and unobtrusive possible partner in our work. That's exactly what we want when we want to get work done, but are there IDEs that are themselves autotelic[1] experiences? Can we imagine systems that extend or subvert traditional IDE abilities, creating IDEs that are playfully over-responsive, mysterious, generative, or improvisational?

We have many IDEs that are great for performing useful work and writing good, reliable, productive code. What happens if we let an IDE break free from productivity? Can we consider an IDE as an autotelic experience in itself, rather than as a means to create useful code? This paper is a provocation to the community to consider not only IDEs *for* entertainment, but IDEs *as* entertainment:

- What does is mean to make a non-productive IDE?
- What goals could an IDE have *outside* of productivity?
- What are the common features of an IDE that support productivity goals?
- How can a non-productive/autotelic framing let us imagine warping those tools in ways that were unthinkable in a productivity-focused framing?

What do we expect from an IDE?

An IDE (integrated development environment) is a bundle of utilities that are useful for programming. Usually at a bare minimum, there is a place to write and edit code, and a way to run the code and view its output. They may have additional features to specify how a program is compiled or executed, and features that can help organize and annotate the raw source code.

[1]from the Greek "for its own purpose"

When compared to writing code in a non-specialized text editor and running it in a separate program (or via a command line), some IDEs can be more approachable for novice users as they provide a single entry point to start programming. A novice user can download Processing[Reas and Fry2006], type some rudimentary Java, hit the play button, and see the results of their code, without needing to understand what a compiler or executable JAR files are. Processing's interface, on starting up, contains primarily a text editor, a play and pause button, and a console output window, which gives the user a clear hint that their primary activity loop will be typing code, pressing play, and viewing the output.

Noticing that coding has an edit-execute-observe loop allows us to examine particular moments [Hutchins, Hollan, and Norman1985] in the loop where there may be problems or opportunities, or even consider the whole loop cycle as a way to 'grok' the possibility space of the program, or 'grokloop' [Compton2019]. For example, Hutchins, Hollan, and Norman identify the "gulf of execution" (the moment where you must know both *what* to input into your system, and *how* to input it) and "gulf of evaluation" (the moment you observe what the system has responded with, and must understand what that output means). The faster we go around this loop, the faster we can iterate on our programs, so many IDEs have features that shorten one or the other of these gulfs, such as code completion and syntax highlighting (gulf of execution) and error logging with line numbers and variable tracking (gulf of evaluation). REPLs (read–eval–print loop) are also available: these IDEs (often web-based IDEs) can re-run code each time the user edits the code. If we can sufficiently shorten the two gulfs and the time it takes to re-execute the code, then the core loop of the IDE can start to feel like a continuous manipulation of the code.

Speeding up the turtle

LOGO [Papert2020] is a full programming language best remembered for drawing generative spirals with turtles. When you program in LOGO (or many of its copycats) you watch the turtle painstakingly walk through its path. Papert thought that one could more easily understand the code by "playing

Turtle", either watching it move or physically walking the path yourself. But the gulf of evaluation for that paradigm is painfully slow. Nicky Case's "Joy.js" (https://ncase.me/joy/) re-imagines LOGO as a continuously running REPL, but as the user edits the code, the turtle's path redraws itself in an instant. Not content with bridging the gulf of evaluation, Joy.js also shortens the gulf of execution by replacing plain-text code with sliders for numbers. The result: a user can continuously update the program a dozen times in a second by sliding a slider and seeing the changes, which the side claims makes Joy.js "more creative, more alive" and "lets you improvize, and discover 'happy little accidents'".

Joy.js is one of many continuously-updating REPLs available, which also includes Tidal[2] for generative audiovisual experiences, Sonic PI[3] for music and Shader.place[4] for WebGL shader code. All these IDEs focus on emergent generative programming. One notable feature of generativity is that the eventual pattern that will emerge from an instruction is hard to predict from the instruction itself. By shortening the grokloop, these IDEs allow the emergent pattern to change simultaneously with the code, without waiting, so that the user can playfully explore the space and enjoy the emergent patterns, or even do so as a live performance ('algorave').

In the case of Shader.place, the code is also live-synced to multiple other players in a virtual room, who can all simultaneously edit the shader together. Though the code often breaks (shader code is very fragile) the code will recompile as all the users are typing, and will continue to recompile until it runs again. Continuous execution makes the chaos of a multiperson shader-editor viable and even enjoyable, rather than frustrating.

IDEs with agency

So far we have just talked about IDEs that assist the user to edit and execute code. But many IDEs will also play an active role in annotating or even writing code themselves. Common ways that IDEs take *an active role* when coding include:

- detecting the syntax of parts of the code and coloring or formatting it (syntax highlighting, auto-indentation)
- alerting the user to malformed, nonstandard code (linting)
- tracking how identifiers or variable names are used across code to create clickable links, indexes of identifiers, or one-click renaming UIs
- identifying possible identifiers that could fit in the current context (code completion)

There are systems where an advanced AI autocompletes user's code while the users is writing code in a productivity-focused environment[5], and systems where an advanced AI generates emergent new code that the user can inspect in an autotelic tool[Kreminski et al.2020]. There seems to be potential for autotelic systems to generate not only code, but also annotations, formatting, links and UI features. Are there currently any systems where the IDE itself is given enough agency to be surprising, or to have a character? Can an IDE have enough agency to feel like it is improvising with the user, rather than acting as a transparent tool?

[2]https://tidalcycles.org

[3]https://sonic-pi.net/

[4]https://www.shader.place/

[5]https://copilot.github.com/

IDEs with mystery

Finally, there is a potential to also experiment with IDEs that oppose or challenge the user. An example of this in a productive framing would be teaching IDEs that provide a set of tests that the user must write code to pass, as in the online teaching apps Check.IO[6] and CodeCombat[7]. Normally, the user of an IDE can program anything within the space of the language. These IDEs subvert that by restricting what the user can program and when, or which syntax is accessible. If we consider an IDE as a way that we feed instructions to a computer, we can consider games with antagonistic or mysterious UIs like "The Gostak" (Muckenhoupt, 2001) and "Mu Cartographer" (Millet, 2017) as playful IDEs.

As we look forward towards a future filled with more machine-learning than we might want, we can also look at interacting with machine-learned models as a form of IDE. To "program" with these models we must find a shared syntax that both model and the humans understand, referred to as 'prompt engineering' [Gwern2020] or 'prompt hacking'[8]. This trial-and-error conversational programming often resembles an adventure game like "The Gostak" more than a traditional IDE.

In conclusion, we will still need productivity-focused IDEs with which to write games and digital entertainment. But let us imagine also a world where we make IDEs that are generative, playful, and even mysteriously alien.

References

[Compton2019] Compton, K. 2019. *Casual creators: Defining a genre of autotelic creativity support systems*. University of California, Santa Cruz.

[Gwern2020] Gwern, B. 2020. Gpt-3 creative fiction, https://www.gwern.net/gpt-3prompts-as-programming.

[Hutchins, Hollan, and Norman1985] Hutchins, E. L.; Hollan, J. D.; and Norman, D. A. 1985. Direct manipulation interfaces. *Human–computer interaction* 1(4):311–338.

[Kreminski et al.2020] Kreminski, M.; Dickinson, M.; Osborn, J.; Summerville, A.; Mateas, M.; and Wardrip-Fruin, N. 2020. Germinate: A mixed-initiative casual creator for rhetorical games. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 16, 102–108.

[Papert2020] Papert, S. A. 2020. *Mindstorms: Children, computers, and powerful ideas*. Basic books.

[Reas and Fry2006] Reas, C., and Fry, B. 2006. Processing: programming for the media arts. *Ai & Society* 20(4):526–538.

[6]https://checkio.org/

[7]https://codecombat.com/

[8]https://twitter.com/ArYoMo/status/1399801016669835268