The case for an integrated editor

Joop Ringelberg 26-04-22 Version: 1

Introduction

Perspectives is composed of several components:

- a conceptual system whose primary notions are context, role and perspective;
- a graphical language to model real world situations in those terms;
- a plugin for Enterprise Architect to help create a graphical model;
- a written language to express a model (called ARC);
- a distributed runtime system (the PDR) that interprets such models into an application with end user screens.
- a container program (InPlace) that allows one to actually use the PDR in the browser.

Finally, we seek to support the modeller who uses the written language. Originally, we planned support for ARC in two IDE's: vscode and Atom. In this text we discuss the relative merits of an alternative, an integrated (and thus online) editor.

We'll first discuss our online editor of choice, to be able to compare the IDE solution with the integrated solution.

We will then compare both approaches on the following dimensions:

- source management
- architectural integration with the PDR
- editing support
- ease of use
- intended audience

Online editors: alternatives

A number of professional open source text editors geared towards programming have been around for years. One of the first is the ACE editor (https://ace.c9.io), which has been around since 2010. We (Joop Ringelberg) have integrated this editor in a previous product.

A more modern alternative is the Monaco editor (https://github.com/Microsoft/monaco-editor), that is the core of the vscode IDE. It can be used in the browser just as well as in the desktop application vscode (although a different tokenising engine has to be used).

Where Monaco comes with a hefty 5 Mb footprint, the recent new implementation of CodeMirror steps in lightly with just over 1 Mb. Moreover, CodeMirror 6 (https://codemirror.net/6/) is very well modularised and supports the Language Server Protocol, making it ideal for our purposes. CodeMirror has good performance, is very customisable and gets excellent reviews.

See https://blog.replit.com/code-editors for an in-depth review of these alternativs.

We opt for CodeMirror 6.

Architectural integration with the PDR

The PDR is a javascript program that relates to screens in a way similar to a server program. The PDR 'serves' content to multiple clients, where each client is a separate tab or window in a browser program. Nevertheless, the PDR runs client side: it runs in a shared web client. It is a javascript process that is separate from the one running in a tab or window.

IDE's, however, run as native OS programs. They must receive support for a particular language through an (http) server that speaks the Language Server Protocol (LSP). For Perspectives, this server must be the PDR or it must be able to use a PDR.

In principle, the PDR can run as a Node process outside of the browser environment. However, several practical difficulties have to be overcome before this can be a reality:

- the node process will have to handle the (cookie) authentication mechanisms of the browser. The browser version uses fetch or XHR. While an XHR-emulating module exists for Node (minimising the required changes), it does not support cookies. We have maintained a fork that added that functionality, but have dropped it, as browser developers continually improve security on these aspects and publish them overnight, while corresponding Node modules lag behind;
- the node process will have to handle CORS (Cross Domain requests).

Both are not trivial and make PDR-as-Node-Server prone to breaking.

In other words, these issues complicate language support for Perspectives in vscode and Atom.

To evade these problems, we have devised an architecture where the PDR *in the shared webworker* functions as a 'server' for an intermediate OS process that

- on the one hand speaks LSP to Atom and vscode (and thus functions as a server)
- on the other hand connects to the PDR in the browser as a client, where the PDR listens continuously (e.g. on a WebSocket) to this client.

Because these connections are local, they can rely on plain http (at least do not have to deal with CORS).

This will work, but is not ideal, not in the last place because the modeller will have to make sure that there actually **is** a browser process that serves his IDE. The more connections, the more security issues arise, the less robust the system.

Contrast this to the *much simpler* architecture where an editor running in the browser itself must connect to the PDR:

- either directly (and then we'll have to extend the PDR API with LSP)

- or through an intermediate process, running in another webworker, that speaks LSP and connects to the PDR.

In conclusion we can say that an integrated editor is architecturally superior to IDE support.

Source management

One of the boons of an IDE is source management, through a combination of

- file system support and
- source control system integration (e.g. Git).

With a browser-integrated editor we lose this facility. On closer inspection however, IDE source management for Perspectives is itself a little problematic. This is because model artefacts have to be stored online, in repositories, to be of use in the Perspectives Universe. The consequence is that a model has not one but at least *two* locations where it is stored:

- in the Perspectives repository;
- in the modellers file system;

And complicating things even further, usually local source management systems connect with remote ones for duplication and sharing (think Github). These functionalities, however, are taken care of quite differently in the Perspectives Universe (with private but shared contexts).

Multiple storage of an artefact is always problematic. The browser-integrated editor does not have this problem.

What about *version control*? Couchdb (the storage backend of choice for Perspectives) supports version control on the level of a single document out of the box. We can strive towards unlocking this functionality in the integrated editor.

Single-document version control is a far cry from the multiple-artefact version control offered by (e.g.) Git. But we have good reason to suppose that future functionality in the Perspectives Universe will not be constructed by combining tenths or hundreds of files, but instead will be more a monolithic modelling effort (not excluding import of existing models). We do not expect Perspectives model development to be a large, coordinated effort involving teams of workers. In short, an 'application' equals one model, equals one file.

Finally, IDE's make it easy to compare one version of a document with another. Javascript-based versions of the diff tool that run in the browser, exist. In other words, it is in principle possible to compare model versions in the browser, too.

Concluding: with an integrated editor we lose out on (integrated) refined source control, but this seems less of an issue for Perspectives than for traditional programming languages. We also lose the ability to compare model versions (but it is possible to reclaim this functionality later on). We win, however, in terms of clarity of where models live.

Editing support

Here we can be brief and unequivocal: editing support in the browser and in an IDE can be the same. This is easily seen in the case of Monaco, as it actually is the editor integrated in vscode. Our editor of choice - CodeMirror 6 - compares very well with vscode when it comes to editing support and experience.

Ease of use

Building code is not just about editing. How would the rest of the user experience be, when we support her with an integrated editor rather than an IDE that should be installed? Let's explore:

1. Setup: atom and vscode need to be installed, whereas an integrated editor does not. Neither does the integrated editor require extension packages to support ARC.
2. File selection. It is not yet quite clear how files to be edited, should be opened from within an IDE. In principle, the files stored in the Perspectives Universe are leading. However, with the integrated editor things become very natural indeed: from within a context that holds a model file (and our current models represent a model as a role, with the source text as a property), the modeller can open the editor right away.
3. As stated before, the modeller will have to make sure a browser with the PDR is running, in order to tap into the Perspectives Universe. Obviously this is not an extra step if editing commences from within a context.

To sum up: the integrated editor wins when it comes to ease of use.

Intended audience

A big advantage of IDE's would be familiarity. Many programmers use either vscode or Atom, so they would get a flying start if it were possible to edit ARC files in their favourite environment.

However, we have some reason to believe that it will not be per se *programmers* who will lead the way when it comes to building InPlace functionality. Perspectives builds on a completely new conceptual framework and our (limited) experience shows that conventional programmers are reluctant and sceptical. We expect that modelling will be taken up by people who are closer to end users (or to 'the business'). For them, an IDE might be an overwhelming barrier, while an integrated editor is just a single click away.

Conclusion

Summing up in terms of pros and cons:

PRO: an integrated editor

- is architecturally superior to the IDE solution;
- provides a much clearer concept of where models are;
- is easier to use or start using than an IDE;
- aligns better with the intended audience.

CON: an integrated editor

- does not support version control of *collections* of sources - but this seems to be far less important for Perspectives than for traditional programming languages;
- does not offer file version comparison out of the box (but this can be added later to InPlace);

Most important, however, is that the editing experience itself does not significantly differ between the two approaches. We don’t sacrifice editing support by moving to an integrated editor.

All in all we are in favour for the integrated approach.