Four Generic Issues for Tools-as-Plugins Illustrated by the Distributed Editor Saros

Lutz Prechelt
Freie Universität Berlin, Institut f. Informatik
Berlin, Germany
prechelt@inf.fu-berlin.de

Karl Beecher
Freie Universität Berlin, Institut f. Informatik
Berlin, Germany
karl.beecher@fu-berlin.de

ABSTRACT

Saros is an Eclipse plugin for multi-writer, real-time, distributed collaborative text editing that also includes VoIP, chat, whiteboard, and screen sharing functionality. We present four problematic issues we encountered in the development of Saros: Providing portability, choosing a metaphor, handling clashes in display markups, and attributing incompatibilities correctly to their source. These issues will apply to many other plugins similarly. For three of them, no generic solution approach yet exists but should be worked out.

Categories and Subject Descriptors

D.2.6 [**Software Eng.**]: Programming Environments—*Integrated Environments, Eclipse*

General Terms

Design

Keywords

Saros, portability, metaphor, conflict, incompatibility

1. INTRODUCTION

Saros (www.saros-project.org) is an Open Source Eclipse plugin for distributed collaborative text editing and viewing. This means that two or more participants of a Saros session have an identical copy of all files of a project (using any set of textual languages) and any change made by any participant of the session will be reproduced in real-time in the corresponding files (and, if applicable, on the screen) of all other participants. Saros shows at any time where each participant is working by annotations on the package explorer file icons, a remote quasi-scrollbar, and other means as shown in Figure 1. It shows what were the last few changes made by each remote participant by highlighting in the text. Rather than making changes, one can set Saros to follow the view seen by another participant automatically in order to watch what somebody else is doing or showing. All navigation and editing functions of Eclipse (even including refactorings) are still available. A VoIP connection, a whiteboard/sketching function, and optional screen sharing (for non-editing collaboration steps) complete the collaboration scenario.

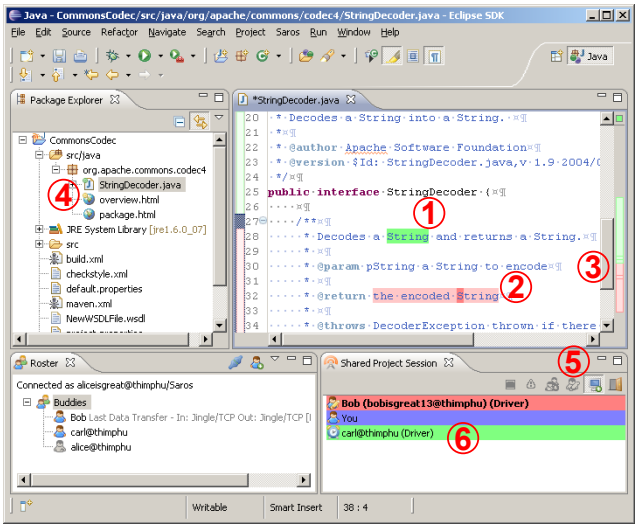

Figure 1: Various awareness features used in Saros such as (1) selection, (2) text edits, and (3) viewports highlighted in each users' color; (4) opened and active files by current drivers, (5) button for activating the follow-mode, and (6) icons showing for which users Eclipse is currently the active window.

Among the many issues encountered in developing such a tool, we have identified four that specifically stem from or relate to the fact that Saros is a plug-in and that will apply in some form to many tools designed as plug-ins.

We present each of these in a separate section and describe first the origin and form of the issue as it appears in Saros and then its generic core.

2. ISSUE: PROVIDING PORTABILITY

2.1 Saros Issue

In principle, Saros is for extending any text editor with collaborative features; the dependencies on any other specific Eclipse functionality (besides pure text editing) are either incidental or optional, but never fundamental. Saros therefore ought to be portable to other popular IDEs, at

*TOPI'*11, May 28, 2011, Waikiki, Honolulu, HI, USA

least those that are based on Java, such as Netbeans or IntelliJ.

However, when we started developing Saros, we had our hands full with other things: We had to design and implement the basic mechanism for putting and then keeping local resources in sync, had to develop a suitable mental model for explaining this to the user (see Section 3), had to design UI functionality for making the user aware of what was going on on the remote side (see Section 4), and had to implement a non-trivial networking infrastructure. Frankly, we did not spend even a minute thinking about portability to other platforms than Eclipse.

Looking at Saros today, we find that it is intertwined with Eclipse in quite a number of places: editor input handling, editor output control, color markup in the edited text, color markup in the annotation bar alongside the text, color markup in our separate remote quasi-scrollbar (remote viewport indication), annotation on the icons in the package explorer, reading and storing Saros preferences (options, settings), editing Saros preferences, reading Eclipse preferences regarding files to be ignored in Saros session synchronization (such as generated files or version management metadata), and so on.

We find the idea of having Saros in other IDEs attractive, but have not analyzed it closely yet. However, the thought of untangling all of the above from the Eclipse API and realizing it via a yet-to-be-defined Saros API in such a way that this API can be straightforwardly adapter-connected to the APIs of multiple other IDEs feels intimidating.

2.2 Generic Issue

Unfortunately, we know of no set of instructions that help achieving this untangling What would be needed to do it was a point-by-point overview of most aspects of the various IDEs' APIs that describes
(1) which features of platform A look how on platform B,
(2) how their common denominator could be represented,
(3) what are the pros, cons, and limitations of each of the different options for such representation,
(4) which parts of A cannot be achieved in B at all,
(5) which parts can be achieved but only in such a different manner that a common representation appears hopeless.
Given the partially sketchy state of documentation of even the best IDEs' APIs, we do not find this likely to appear any time soon, though.

3. ISSUE: CHOOSING METAPHOR

3.1 Saros Issue

When we design the original version of Saros, we developed what looked like a neat way (from the user's point of view) of integrating it into Eclipse: User A would right-click any existing Eclipse project in the package explorer and choose "Share project...". This would open an invitation dialog in which A (called the "host") could select one other user B and invite her. For this, both A and B must be previously logged into an XMPP server. Saros would then copy all files of the respective project over from A's Eclipse workspace to B's and the shared editing would start. Only one user had write privilege at any time and the host could assign it back and forth as needed, resulting in a strict close-collaboration work mode as in pair programming.

Since then, we have added a lot of functionality into Saros: There is now text chat, VoIP, screen sharing, a whiteboard for sketching, meaningful coupling of version management operations, an unlimited number of users in the session, and all users can have concurrent write privilege. All members of the session will inhabit the chat room and whiteboard, which are automatically created with the session, and can create subsessions for screen sharing and VoIP as needed.

The multi-writer feature turns out to be the crucial one: Today, we find that our former metaphor of a "shared project" as the central UI concept of Saros does no longer work well. Rather, we are now considering sessions in which a larger number of participants (say, five) work in a common session all day, but closely collaborate only part of the time, a work mode that is inspired by Alistair Cockburns Side-by-Side Programming [1, Section 3.T8] and that we term Distributed Party Programming. In such a setting, one would like to have:

- A session that is independent of a host and can exist for an unlimited time.
- A common chat room for asking short questions and for arranging close collaborations using other media.
- Chat rooms, VoIP conversations, screen sharings, and whiteboards that are available for arbitrary subgroups (often two people, possibly several subgroups at once) and are possibly short-lived, to be used on a case-by-case basis during times of close collaboration.
- The ability to share more than one project.
- The ability to share a project with an arbitrary subgroup of the session participants and to unshare it again.

So what is the issue here? We found that this move from *shared project* to *session* as the central metaphor is technically very hard for us. The reason for this is that we failed to keep *shared project* and *session* clearly apart in our design structure when we started building Saros. The reason for that is that *shared project* appeared so natural initially that it never occured to us the two concepts might *not* be one and the same.

3.2 Generic Issue

So why did we make this design mistake? And what has all this to do with Saros being a plugin?

We believe that the deeper issue is the influence of the base platform, Eclipse in our case, on the choice of metaphor and other core concepts. It is tempting to reuse existing concepts from the platform because this appears to make things easier both for the user and for the designer. It appears to save learning effort; one expects to obtain a lot of understanding essentially for free.

The truth is, however, that such a choice might be misleading. We believe that, had we designed Saros as a standalone tool, we would have recognized and used *session* as the central concept rather than *shared project* and would not now have as much hassle with "liberating" the individual Saros services as we do.

Conclusion: Being a plugin can seduce tool designers into less-than-optimal choices for core concepts.

4. ISSUE: HANDLING MARKUP CLASHES

4.1 Saros Issue

Being a collaboration tool, an important aspect of Saros is the manner in which Saros shows a user what is going on on

the remote side(s). For this purpose, six different channels are used:

- The existing file icons in the package explorer are extended by a yellow dot in the top left corner of the icon of each file that is opened by some member of the current Saros session. If the file is currently visible in an editor, the dot is green (area 4 in Figure 1).

- Each user within a session has a personal color, these colors are shown in each user's the session view (area 6 in Figure 1). The annotation bar to the left of the editor text shows this color at each line that user can currently see in her viewport.

- The left edge of the session view also shows an icon for each user indicating whether the user's Eclipse window is currently focused ("present") or not ("away").

- A remote quasi-scrollbar shows the scrollbar of each remote user to the right of the local user's real scrollbar (area 3 in Figure 1), so that the local user can understand which part (if any) of this file the remote users are viewing even when those lines are not in the local viewport.

- Within the editor pane, the current text selection (if one exists) of a remote user is shown in that user's color (area 1 in Figure 1).

- The few dozen characters affected by the most recent edits of a remote user are also highlighted in that user's color in the editor pane (area 2 in Figure 1).

The markup of areas 3 and 6 appears in regions of the screen that are private to Saros. The markup in the package explorer icons, annotation bar and in particular in the text, however, can collide with markup provided by other plugins. In the case of Saros, we find that in practice this is a problem only rarely. Some programming language plugins provide similar markup in the text area but the problems resulting from these and other clashes are usually modest.

4.2 Generic Issue

However, this need not always be the case. The markup of the next plugin a user wants to combine with Saros may clash with Saros' so violently as to render either of these markups and perhaps even the plugins themselves useless. There are no conventions (let alone mechanisms) in Eclipse that establish a regime in which a plugin could systematically step out of harm's way when such clashes occur.

The first step in this direction could be a technique by which a plugin could automatically detect impending markup clashes. The second step could be markup conventions that define a small set of "channels" of markup, consisting of a color palette, rules for symbol palettes, and placement rules. With such channels, any well-behaved plugin might use only one of them and could switch from channel (one set of markups) to another in order to avoid markup clashes with other plugins.

Currently, the only way for plugin authors of coping with markup clashes is making the plugin's markup highly user-configurable. There are currently not even guidelines for structuring such configuration options so that users may have to modify dozens of settings individually.

5. ISSUE: DETECTING SOURCES OF INCOMPATIBILITY

5.1 Saros Issue

While markup clashes are annoying, they will only rarely actually *break* the functionality of a plugin. The same is not necessarily true for other types of incompatibility.

For Saros, it is vitally important that it be informed (as required by the Eclipse programming guidelines) about any changes made to any text file in the current session, because Saros has to propagate those changes to the other participants. If some other plug-in is not well-behaved in this respect, the consequence is brutal: Saros has a consistency watchdog that periodically compares checksums of all files. If a file is found to be out-of-sync with the version present at the session's host, it will be flagged as inconsistent and the user's only options are to either ignore the problem or to overwrite the file with the host's version.

5.2 Generic Issue

The deeper problem behind this is misallocation of trust. A user can work with a misbehaving plugin for a long time and never detect the misbehavior at all.

Along comes the yet unknown Saros plugin and suddenly things go wrong: The new plugin speaks of "inconsistency" and if the user did not read the prompt carefully, s/he may overwrite an important change. Saros (or some other plugin in a similar position) will receive the blame although in fact the other plugin is the one at fault.

However, Eclipse provides no general mechanism by which the plugin that has detected the existence of the problem can also detect the problem's origin and attribute it to the actual culprit.

6. CONCLUSION

We have used the Eclipse-based distributed editor Saros as an example to present four problematic issues that will affect many tools developed as plugins:

- "Providing portability" pertains to the coexistence of several competing plugin platforms such as (in the Java case) Eclipse, IntelliJ, and NetBeans.
- "Choosing metaphor" pertains to the influence onto the particular plugin of the basic platform's concepts, which may affect UI design as well as technical architecture negatively.
- "Handling markup clashes" pertains to the coexistence of the particular plugin with other plugins if both of these add markup to the default display of existing UI elements.
- "Detecting sources of incompatibility" pertains to the coexistence of the particular plugin X with other plugins if the other plugins misbehave in a way that ruins X's ability to function properly.

7. REFERENCES

[1] Alistair Cockburn. *Crystal Clear: A Human-Powered Methodology for Small Teams.* Addison-Wesley Longman, 2004.