Eclipse as a Platform for Research on Interruption Management in Software Development

Uri Dekel
ISRI, School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue, Pittsburgh, PA 15213
udekel@cs.cmu.edu

Steven Ross
Collaborative User Experience Group
IBM Research Cambridge
1 Rogers St., Cambridge, MA 02138
steven_ross@us.ibm.com

Abstract

Automated tools for mediating incoming interruptions are necessary in order to balance the concentration required for software development with the need to collaborate and absorb information. At present, there is no design knowledge for building such tools for programmers. The abundant literature on the general problem of interruptions and awareness does not address the unique characteristics of software development, and the few studies which do are restricted to simplified tasks or environments. We attribute this scarcity to difficulties in conducting empirical studies in real settings, because of the need to implement appropriate research tools.

Eclipse is poised as an ideal platform for such research thanks to its popularity, plug-in model, and observation hooks. This paper presents GateKeeper, a plug-in based framework for managing interruptions, allowing the rapid implementation of different interruption and awareness models, and their integration within actual collaboration tools. To validate our framework, we implemented a rule-based interruption management system, and integrated it with *Jazz*, an *Eclipse*-based collaboration tool.

1 Introduction

1.1 The problem of interruptions

Distractions incur mental context switches which slow and introduce errors into cognitively-complex tasks [1], a phenomenon which appears to apply to software development as well [8]. Programming is a delicate and error-prone process, but developers do not operate in a vacuum: they must receive information and alerts about their project, and collaborate with their peers. Ideally, distractions should occur only if the importance of the information outweighs the cost.

In a co-located team, social cues help determine whether a peer can be interrupted [5]; these cues are missing in distributed teams. Synchronous means of communications, such as instant messaging, facilitate interaction between remote team members. However, since they provide no awareness about the recipient's activity, the initiator must first attempt the communication, causing an immediate distraction, and only then negotiate the interruption. If such attempts become too frequent, they can negatively impact a person's ability to get their own work done. Frustrated users might block incoming communications, reducing their value to the team and possibly missing important developments.

One approach to this problem is providing awareness to peers. In the simplest form, adapted by most IM tools, users manually set their status. While useful for planned and prolonged periods of unavailability, the need for manual set-up makes this inadequate for the rapid context-switching of software development. For example, it is unacceptable for a programmer who only wants to block interruptions while debugging to manually set this status prior to every launch. More advanced awareness tools use physical sensors, video or voice analysis to determine interruptability [4, 5]. However, while there is a taxonomy of activities that occur during the design process [9], a similar classification for actual programming would require identifying relevant events in the IDE.

Clearly, simply increasing awareness is not enough because it leaves control of distractions in the hands of others, while raising a myriad of privacy issues [6]. Awareness also fails to affect automated external agents, such as context-sensitive

assistants [3, 8], source control monitors [10] and bug-database monitors. The complimentary approach involves an automated agent working on behalf of the user. This agent buffers between interruption sources and the user, and *mediates* incoming interruptions by deciding where, when, and how to present the information.

1.2 Previous research

In his fundamental work on interruption coordination, McFarlane [7] identified four *methods of coordination*, means in which UIs can support interruptions: The *immediate* style presents the incoming interruption instantly, without regard for a person's status; it often requires immediate action. The *negotiated* approach peripherally announces the existence of an interruption, but allows the user to choose when to devote attention to absorbing the details. A *mediated* style involves the agent described above. The *scheduled* style allows interruptions only during specific time windows.

McFarlane's taxonomy gave rise to works that compared these interruption styles. His own lab experiments with cognitively-simple game tasks showed that negotiated (and mediated) interruptions are usually preferable, especially when accuracy on the primary task is important. The immediate style was only useful when promptness in responding to the interrupting task was necessary.

Robertson et al. [8] strove to obtain design knowledge for programming environments. They compared the effects of immediate and negotiated interruptions from an assistance agent on the debugging of spreadsheet programs. Their experiments showed that the negotiated style is always preferable, although they did not have interruptions which required prompt response and did not attempt to mediate interruptions. Clearly, more experiments are necessary to obtain design knowledge for real development scenarios.

The key to successfully mediating interruptions lies in balancing the user's current activity and the relevance of the information. The techniques used to obtain awareness information can be used by mediating agents. Horvitz's *attentional user interface* and *notification platform* projects [4] developed a generic framework for mediated interruptions. Incoming interruptions are handled by a notification manager which decides how to handle them using a *decision model* capable of interrupting through a variety of mediums. This model uses an *attention model* to determine the interruptability of the user and the preferable interruption medium by relying on a variety of sensors which report to a *context server*. Their published implementation, however, does not address the specifics of software development, nor can it be easily integrated into IDEs.

1.3 Our work

If we could collect experimental results in real development settings, we would have design knowledge for building mediating agents customized to the special needs of programmers. We need to identify the activities and states that occur in programming, and discover the best ways to relay information from different sources in each state. The scarcity in data likely arises from the difficulty in developing appropriate experimentation tools and integrating them into real development environments.

Eclipse is well poised as a research platform for collecting such data. Its popularity and platform-independence allows experiments to be conducted within the developer's native environment. Its robust plug-in mechanism allows smooth integration of tools into the environment, as well as integration between different tools. In addition, *Eclipse* provides an extensive set of hooks, allowing fine-grained monitoring of the user's interaction with the IDE.

In this paper we describe *GateKeeper*, a plug-in based framework for context-awareness and interruption management in *Eclipse*. We aim to promote research on interruption handling in real development settings by providing convenient means for the rapid development and integration of research tools such as awareness or decision models. To validate the usefulness of this framework we developed three extensions for *GateKeeper*, reminiscent of the research we described above: a sub-framework for determing context based on multiple sensor feeds, a rule-based decision model, and integration of our framework into the *Jazz* collaboration tool [2].

2 The framework

The core of the *GateKeeper* framework does not deal with the specifics of interruption management. Instead, it provides extension points and relies on additional plug-ins to contribute *configuration features* which build up to an *interruption management configuration*.

The fundamental configuration feature is the *interruption management scheme*, a *strategy object* which serves as our *decision model*. It receives *incoming interruption request events* and returns *decision events*. A scheme can be as simple as blocking or allowing all interruptions, or as complex as a

rule-based or learning-based strategy. Only one of the registered schemes is active at any time.

Every interruption request has an associated *interruption type*, selected from a hierarchy of registered types. A scheme can, for example, block all chat requests, or block specific ones, such as voice chats. Every request also has a *source agent*, selected from a single-rooted multiple-hierarchy of registered agents. Concrete agents, such as individuals or automated agents, form the bottom layer of this lattice, while teams form the abstract internal nodes. The use of multiple hierarchy accommodates overlapping between teams.

User context in *GateKeeper* is represented by a subset of the registered *context states*, serving as *predicate* objects which schemes can query or watch. The registered states are also arranged in a semi-lattice.

The collections of features constituting the configuration are populated when the interruption manager object is first created. The framework publishes extension points for each kind of configuration feature, using them to obtain *suppliers* that provide the objects and their placement in the hierarchies. Features can also be added and removed at run time, allowing the system to adjust to changes such as programmers leaving and joining the team. Listeners are notified when such configuration changes occur.

We now describe the process of submitting and handling interruptions when *GateKeeper* is operating on the recipient's machine. We use *Jazz* as a running example, demonstrating the handling of IM interruptions. The *Jazz* framework has already contributed the necessary configuration features, including an interruption type for IMs, agents representing teams and individuals, and appropriate context states.

Jazz has a thread which intercepts incoming chat requests from the server. When such a request arrives, it starts another thread which would normally open a chat window displaying the first message. With *GateKeeper*, however, the thread first requests the singleton interruption manager object. It then creates an *interruption request event*, specifying IM as the type and the sending user as the source agent, and attaches the contents of the first message to the request. Since it is an independent thread, it now makes a synchronous submission of the event to the interruption manager, blocking until a decision is made.[1]

The manager sends the request to the scheme, which it treats as a black box. The decision is returned in the form of an *interruption decision event*, which is classified as either an acceptance or a denial, and is relayed back to the submitting code. In our example, if an acceptance decision is made, the thread will go on to open the chat window on the recipient's side. If the request is denied, the thread does not open the window and instead sends an instant message to the original sender party with an automated response informing of the recipient's unavailability.

Our framework is not limited to deciding between distracting *immediate-style interruptions* and complete blocking. We provide inherent support for *negotiated interruptions* by storing the suppressed interruptions and presenting them when requested, using the attached messages. *Jazz* registers itself as a listener to the manager, and decorates the pictures of the involved parties in the *Jazz* band (an iconic "buddy list") with an envelope icon when an interruption request is denied; the framework collects these requests and allows users to view the waiting messages at their convenience.

Decision events also have associated *actions*, executed by the manager before returning decisions to the client program. These actions can peripherally notify the user about blocked interruptions, for example by playing a sound, showing a temporary pop-up, or, as done in *Jazz*, briefly flashing the image of the user.

Our framework also supports the implementation of mediated interruptions. If the time is not appropriate for an interruption, the scheme can postpone making the decision. It can also demote an immediate interruption into a less distracting one by denying the request and executing an action for the alternative mode of presentation. Scheduled interruptions can be implemented by delaying decisions until the appropriate time.

Finally, note that client programs can also register *interruption handlers*, rather than handle accepted requests by themselves. This allows, for instance, for different monitor agents to send one-line alerts, counting on a single handler to display them to the user once they are allowed through.

3 A sensor-based context management framework

The *GateKeeper* framework represents the current context of the user as a subset of the registered *context states*. While it is straightforward to identify relevant states, such as `debugging`, determining their semantics is an open research problem. For example, how do we distinguish between debugging and testing? The solution is likely to

[1] We also support asynchronous submissions using callbacks.

rely on identifying patterns in data captured from the IDE. In preparation for such research, we added a sub-framework for sensor-based context states.

At the heart of this framework is a plug-in for *GateKeeper* which provides it with a singleton *context supplier*. This supplier obtains *atomic context information templates* and context states by publishing two extension points. Each template can be thought of as a simple sensor, and is used to instantiate values, typically by hooking into the IDE. For example, one contributing plug-in tracks the current perspective and project, unsaved files and processes under debug. It then provides a *debug* context state which uses a complex condition on these atomic values. Similarly, we have a plug-in which uses native methods of the Windows API to find which programs and windows are currently active. A user who chooses to use this plug-in, in spite of its privacy implications, can benefit from context states that involve non-programming activities. Of course, more complex algorithms can be used for determining state from atomic information.

4 A rule-based interruption management scheme

As part of our validation of the framework, we wanted to implement an interruption management scheme which would mediate some interruptions from the distracting immediate style to the less distracting negotiated style. We strove to support customization while minimizing user interaction, so that programmers can incrementally set their preferences. A rule-based scheme, reminiscent of the filtering rules of e-mail clients, was therefore developed. While we are not advocating it as the best approach, it does demonstrate the capabilities of our framework. It also has the advantage of being intuitive to programmers and of giving a measure of *accountability*, allowing it to justify why it reached a certain decision, making it easier to rectify problems.

Our scheme maintains a set of *profiles*, of which exactly one is active at any time, Each profile consists of an ordered list of *policies* (rules), and a default action. When an interruption request arrives, it is tested against each policy in order until the first match is found, and the decision is made according to the action associated with that policy. If no match is found in the current profile, the default action takes place.

We demonstrate the use of this scheme on a scenario using *Jazz*. Suppose that during this morning's group meeting we were tasked with fixing a critical bug, and wish not to be interrupted while doing so. We open the scheme editor and add a new policy, choosing to suppress all interruptions from everyone while we are debugging.

We soon realize that we are no longer alerted when coworkers modify the files we are working on and create potential conflicts. The *Jazz concert awareness* component which watches the CVS repository for changes is set to generate distracting pop-up alerts, but is now blocked. We therefore add a second policy, asking the system to beep when a concert alert from the concert agent occurs while we are debugging; we can view the actual messages later. The system places this new policy before the previous one, because it is more specific.

Later, our department manager instructs us to provide immediate assistance, if requested, to a team with a pending deadline. We add a third policy, stating that all interruptions from that team shall be accepted automatically. The system realizes that this new policy conflicts with the existing policies when we are debugging, and asks for clarifications which help it position it in the list.

We have finished debugging and started writing new code. A dialog appears a few minutes later, asking whether to accept an incoming chat request. Puzzled as to why this interruption was not deferred, we ask the system to justify its decision and learn that none of the policies apply to our peer when we are not debugging. We can now decide whether to accept this request, and can also create a new policy on the spot.

The default action for each profile takes place when no policy matches a request. The two obvious actions, *accept* and *reject*, can be used to easily adopt a blacklist or whitelist approach, respectively. The prompting option, however, helps to gently introduce new users to the interruption manager. Upon receiving the first interruption, they get a chance to define a policy. This can be used to instruct the system to accept or reject all interruptions, or to start constructing a more elaborate set of policies.

Note that the scheme listens to configuration changes, such as additions of new team member. It uses a variety of principles to adapt the existing policies to the new state. The policies also persist across activations of Eclipse, and are adjusted if the configuration changes.

5 Integration with *Jazz*

IBM's *Jazz* project [2] enhances *Eclipse* with extensible collaboration tools for small teams which interact via a shared-objects server. Its features

include a decorated contacts list, IM and VOIP chats, chats around source code, screen sharing, etc. A *concert awareness* module highlights resources based on the risk of merge conflicts.

Jazz consists of a core plug-in which defines extension points and of plug-ins which extend them and implement the *Jazz* features. We thus strove to minimize changes to the existing code, limiting them to the core plug-in. We also needed to allow *Jazz* to compile and run even if interruption management capabilities are not installed.

To this end, we added an *interruption management* extension point to the *Jazz* core, with a method for each relevant interruption kind. We then changed certain methods in the core object, so that for each event we first try to find an interruption management extension and, if found, ask it whether to allow the interruption. No further changes were necessary for the existing *Jazz* code and plug-ins.

We now created the *Jazz connectivity* plug-in, which requires and extends points of both frameworks. First, it contributes *GateKeeper* configuration features, including interruption types, a hierarchy of agents, and peripheral notification actions. Next, the plug-in implements the new *Jazz* extension point, creating *GateKeeper* requests from the *Jazz* objects, and returning the decision to *Jazz*. It also listens for decision events in order to decorate user images with an indication of waiting messages. Finally, it contributes to the *Jazz* band pop-up menu, adding options for changing profiles, editing the scheme, and viewing incoming messages.

An additional plug-in extends the sensor-based context framework with templates for tracking the status of the *Jazz* user, and context states representing each of the predefined *Jazz* user states. The *Jazz* user study plug-in, used for logging data in an internal IBM user study is also extended, to record interruption management events for future analysis.

6 Conclusions

This paper presented the *GateKeeper* framework, and demonstrated its use by a rule-based interruption management system which was integrated into *Jazz*. It is our hope that the capabilities of this framework and *Eclipse* will assist in research for obtaining the design knowledge which will enable the development of better mediation tools for software developers.

Our current research directions include collecting and analyzing in the *Jazz* user study to learn how users collaborate and use interruption management. We are also collecting data from multiple sources in *Eclipse*, in an attempt to characterize programmer activities.

About the Authors

Uri Dekel is a doctoral student of software engineering at the School of Computer Science in Carnegie Mellon University. He holds BS and MS degrees from the Technion in Israel, and has worked for Intel and IBM Research.

Steven Ross is a member of the Collaborative User Experience group at IBM Research in Cambridge, MA. He holds BS and MS degrees from MIT. He was chief architect of the Cambridge Speech Initiative and an architect on Lotus 1-2-3.

References

[1] I. Burmistrov and A. Leonova. Do interrupted users work faster or slower? In *10th Int. Conf. on HCI*, pages 621–625, June 2003.

[2] L.-T. Cheng, S. Hupfer, S. Ross, and J. Patterson. Jazzing up eclipse with collaborative tools. In *Proceedings of the 2003 OOPSLA workshop on eclipse technology eXchange*, pages 45–49, 2003.

[3] D. Cubranic and G. C. Murphy. Hipikat: recommending pertinent software development artifacts. In *ICSE 2003*, pages 408–418.

[4] E. Horvitz, C. Kadie, T. Paek, and D. Hovel. Models of attention in computing and communication: from principles to applications. *Commun. ACM*, 46(3):52–59, 2003.

[5] S. Hudson, J. Fogarty, C. Atkeson, D. Avrahami, J. Forlizzi, S. Kiesler, J. Lee, and J. Yang. Predicting human interruptibility with sensors: a wizard of oz feasibility study. In *Proc. conf. on Human factors in computing systems*, pages 257–264, 2003.

[6] S. E. Hudson and I. Smith. Techniques for addressing fundamental privacy and disruption tradeoffs in awareness support systems. In *CSCW 1996*, pages 248–257, 1996.

[7] D. C. McFarlane. Comparison of four primary methods for coordinating the interruption of people in human-computer interaction. *Human-Computer Interaction*, 17(1):63–139, 2002.

[8] T. J. Robertson, S. Prabhakararao, M. Burnett, C. Cook, J. R. Ruthruff, L. Beckwith, and A. Phalgune. Impact of interruption style on end-user debugging. In *Proc. 2004 conf. on Human factors in computing systems*, pages 287–294.

[9] P. N. Robillard, P. d'Astous, F. Detienne, and W. Visser. Measuring cognitive activities in software engineering. In *ICSE 1998*, pages 292–299.

[10] A. Sarma, Z. Noroozi, and A. van der Hoek. Palantir: raising awareness among configuration management workspaces. In *ICSE 2003*, pages 444–454, 2003.