

A Plug-In Based Framework for Research on Interruption Management in Distributed 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 Street, Cambridge, MA 02138
steven_ross@us.ibm.com

Abstract

There is an inherent conflict between the deep concentration required for software development and the need of programmers to collaborate with their teams and absorb information pertaining to their work. While there are general tools for mediating incoming interruptions, there is not enough design knowledge for building tools specific to programmers.

The abundance of 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 and environments. A considerable barrier to conducting the necessary empirical studies in real settings is the high cost of developing appropriate research tools and integrating them into IDEs.

Our work addresses this problem by providing a plug-in based framework for interruption management. It strives to facilitate the rapid implementation of different interruption and awareness models, and their integration within actual development and collaboration tools. As part of the validation of our framework, we implemented a rule-based interruption management system, and integrated it with a collaboration tool for the Eclipse environment.

1. Introduction

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 [14]. Programming is a delicate and error-prone process which requires the utmost concentration. Unfortunately, developers do not operate in a vacuum. They must collaborate with their

peers, and absorb relevant information that may affect their work.

In a co-located team, social cues help determine whether a peer can be interrupted, and whether the importance of the information outweighs the cost of distraction [8]. For example, a person might wander over to a coworker's office and peek in before interrupting. If the coworker appears busy, one might decide that the urgency of the issue does not warrant interruption. The coworker may even have some peripheral awareness of the attempt, and may negotiate future contact.

Such cues are lost in the transition to distributed development teams. While synchronous collaboration tools, such as phone calls or instant messaging, facilitate ad-hoc interactions between team members, they also pose a serious problem of disruptions. Being unaware of a coworker's availability, a person seeking communications must first initiate contact, immediately disrupting the recipient's current activity. The coworker is forced to announce her availability to the initiator and might have to negotiate future contact. If such attempts become too frequent, they can negatively impact a person's ability to get their own work done. Frustration might lead team members to block their communications channels, thus reducing their ability to provide help to team members and keeping them in the dark about important developments.

1.1. Providing Awareness

One approach to this problem is providing awareness of the interruptability of users to their peers. In the simplest form, adapted by many collaboration and IM tools, users can manually set status flags and messages which are then displayed to their peers. While this is useful for planned and prolonged periods of unavailability, the need for manual set-up makes this form

inadequate for the requirements of software development. For instance, it is impractical to expect a programmer who is open to interruptions at any time except while debugging to change status prior to every debugger launch and to change it back immediately afterwards.

More complex awareness techniques strive to solve this problem by trying to automatically determine a user's cost of interruption. Promising results are shown by research which uses data from physical sensors and video analysis to build statistical models [5, 8] or to train machine learning algorithms [6, 7]. Determining the exact activities of programmers, however, is more difficult. While there is a taxonomy of activities that occur during the design process [15], developing a similar classification for actual programming tasks would require identifying a set of IDE events which are worth monitoring.

Awareness information, however, is not used by external automated agents. These agents can provide important information, but might also be a source of distractions. Some agents provide important alerts, such as a potential conflict in the source control system [16], or reports about relevant events in a bug-tracking database. Other agents provide useful context-sensitive assistance, such as proposing relevant artifacts [3], or assisting in debugging [14]. Automatic agents are also used to monitor less synchronous forms of communication, such as incoming e-mail.

1.2. Mediating Interruptions

Clearly, simply increasing awareness is not enough. It leaves control of distractions in the hands of the initiator rather than in the hands of the interrupted party, while raising a myriad of privacy issues [9]. The complementary approach involves an automated agent working on behalf of the user, much like a human personal assistant for an executive. This agent buffers between interruption sources and the user, and *mediates* incoming interruptions by deciding where, when, and how to present the information.

In his fundamental work on interruption coordination, McFarlane [12] identified four *methods of coordination*, means in which UIs can support interruptions: The *immediate* style presents the incoming interruptions instantly, without regard for a person's status; it often requires immediate action. Examples include chat windows which appear without warning, or modal message boxes which announce the arrival of new mail. A *negotiated* approach peripherally announces the existence of an interruption, but allows the user to choose when to devote attention to absorbing the details. E-

mail clients which use a small icon or audio to announce new mail employ this approach. The *mediated* style involves an agent, as described above. A *scheduled* style allows interruptions to be relayed only during specific time windows.

McFarlane's taxonomy gave rise to works that compared these interruption styles under different settings. His own lab experiments compared them on cognitively-simple game tasks. He concluded that the negotiated and mediated styles are usually preferable, especially when accuracy on the main task is important. Immediate interruptions were only preferable when promptness in reacting to the interrupting task was critical.

Robertson et al. [14] tried to generalize these results to the cognitively complex task of debugging. They compared the effects of immediate and negotiated interruptions from an assistance agent on the debugging of spreadsheet programs by business majors. 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 information for real development scenarios. In particular, the effect of different interruption priorities and the use of mediating agents need to be explored.

The key to successful interruption mediation lies in the prediction of a user's interruptability, by means of determining current activity and the relevance of the incoming interruption. The same techniques used for providing activity awareness are useful here.

Horvitz's *attentional user interface* and *notification platform* projects [7] 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 decision 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 be easily integrated into IDEs.

1.3. Motivation

One might ask why interruption management in software development deserves special treatment. Part of the answer lies with the unique characteristics of software development, which make coordination problems inevitable [11]. First, the scale of modern systems forces a division of labor, yet it is difficult to avoid in-

terdependence between modules. As a result, programmers must interact with their peers, but also with members of other teams. In addition, there is an uncertainty regarding requirements, specifications, and the behavior of the outside world. This uncertainty seeps into the workflow, making it difficult to predict what tasks will be performed and with whom an engineer will have to interact. While formal means, such as UML, facilitate coordination, there is still need for much informal communication. The problem of frequent coordination is further aggravated by the high cost of interruption, brought on by the error-prone nature of software development, which requires tremendous concentration.

The design space for interruption mediation tools for software development is very large and different from that of regular office and management tasks. Software development activities involve a variety of artifacts of different forms and levels of abstraction. In addition to the actual source code, these artifacts include requirement and design documents, UML diagrams, test plans, configuration management logs, bug reports, etc. Because of this variety, it is difficult to determine the exact task a user is performing and the relevancy of the incoming interruptions. To successfully determine activity, the semantic connections between all these artifacts must be explored. Another factor is the complexity of development environments, which provide a multitude of ways for presenting information and notifications.

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. At present, a significant barrier to collecting such data arises from the difficulty in developing appropriate experimentation tools and integrating them into real development environments. In particular, it is difficult to rapidly develop, modify and test different designs until an adequate solution is reached.

1.4. Our work

Our goal is to promote research on interruption management in real development settings by providing convenient means for the rapid development of research tools, such as awareness or decision models, and their integration into IDEs.

This paper presents *GateKeeper*, a plug-in based framework for context-awareness and interruption management in software development. In developing this framework, we tried to meet the following requirements:

- It should support integration with a main-

stream IDE, in order to benefit from existing tools and to ensure that experimentation is conducted in real settings.

- The framework should be able to handle interruptions and awareness information from multiple sources.
- Interruption types and sources at different levels of abstraction must be supported.
- The framework must support different awareness and decision models.
- External tools should be able to use the framework as a black box, with minimal changes to their own code and with no changes to core of the framework.
- The framework should support local and centralized decision models.
- The framework should support all of McFarelan's methods of coordination, and accommodate different notification mechanisms, including those provided by the applications in which it is integrated.
- It should allow awareness information to be derived from multiple sources, including the IDE, other collaboration tools AND other applications. Some information may be elicited from artifacts such as code, design documents and UML diagrams.
- The framework must provide internal hooks which will allow observation and logging.
- It should accommodate outside changes that occur while it is running.

By building plug-ins for this framework (our implementation uses the the *Eclipse* extensions model [4]), researchers can provide it with the specifics of interruption handling, including: interruption types, agents capable of producing interruptions, user states and means to determine them, actions to take when interruptions are accepted or denied, and the actual models used for making decisions. They can then use the framework with its contributing plug-ins as a black box for handling and presenting interruptions in research applications such as collaboration tools.

To validate the usefulness of our framework for actual research, we developed three extensions for *GateKeeper*, reminiscent of the research we described above. First, we present a sub-framework for determining context based on multiple sensor feeds, which can be used as an awareness model. Next, we develop a rule-based decision model. Finally, we demonstrate how our framework was integrated into the *Jazz* collaboration tool.

Outline: The rest of this paper is organized as follows: Our framework is presented in Section 2. Section 3 discusses the multiple-sensor framework. Section 4 describes the rule-based decision model. Integration with the *Jazz* collaboration environment is described in Section 5. We conclude in Section 6, where we summarize how the *GateKeeper* system fulfills its requirements, and present directions for further research.

2. The framework

This section describes the *GateKeeper* framework, and the extension points it makes available to specific implementations. It then explains how applications can use this framework to manage interruptions.

GateKeeper is implemented in the Java language in order to benefit from its runtime class-loading capabilities. It uses the *Eclipse* plug-in model [4] and its accompanying SWT and JFace GUI toolkits. Nevertheless, it is relatively straightforward to decouple it from these toolkits, and to use alternative plug-in models.

One advantage of *Eclipse*, in addition to its popularity and open-source nature, is the abundance of extensions that provide useful artifacts from other stages of the development process. For example, *Rational Rose XDE* [13] contributes UML modelling capabilities to Eclipse, while *Websphere Studio* [10] adds J2EE and web development tools. The artifacts obtained from such extensions can be used to elicit awareness information and improve collaborations in the team.

At the core of the framework is a singleton *interruption manager* object. It is used by extensions that supply specific implementations, and by client programs which submit interruption requests and receive decisions with an appropriate course of action. In addition, this object manages an *interruption management configuration*.

2.1. Configuration features

By itself, the *GateKeeper* framework does not provide specifics which allow flexible control of interruptions. Instead, it relies on additional plug-ins to contribute *configuration features*¹ which build up to an *interruption management configuration*. We now proceed to describe these configuration features in detail.

Scheme: The prominent feature in the configuration is the *interruption management scheme*, a *strategy object* which serves as our decision model. It receives

¹Not to be confused with the notion of *features* from the *Eclipse* plug-in model.

incoming *interruption request events* and returns *decision events*. The scheme can be very simplistic, blocking or allowing all interruptions through. In reality it will be more complex, relying on user-defined rules, statistical models, or machine-learning algorithms with cost-of-interruption heuristics. Although only one of the registered schemes is active at any time, it is possible to create an aggregate scheme which runs multiple schemes concurrently. Such schemes are useful for comparative research purposes or in order to make a decision based on several strategies.

Interruption type: Every interruption request has an associated *interruption type*, which is selected from a hierarchy of registered types. The tree structure, in which internal nodes are considered abstract, allows the use of schemes that make fine-grained decisions. A scheme can, for example, block all chat requests, or specifically block certain kinds of chats, such as video conferencing. To ensure consistency, selecting a parent implies the selection of all its descendants. If all the descendants of a particular node are selected, that node becomes selected as well.

Agents: Every interruption request also has a *source agent* and a *target agent*. The registered agents form a semi-lattice, a multiple hierarchy with **anyone** as its single root. Concrete agents, such as individuals, form the bottom layer of the lattice, while teams form the abstract internal nodes. The use of multiple hierarchy accommodates overlapping between teams. Of course, agents need not be human, and may include automated agents such as monitors and assistants. To support deployment in a centralized interruption management server, every interruption request has an associated target agent. This information is hidden in local deployments, and an implicit **self** target agent is used.

Context states: *GateKeeper* represents the context of a user at any given time as a subset of the registered *context states*. Each state is a *predicate* object, which can be queried by scheme objects when deciding whether to immediately accept a request, or watched for specific changes while awaiting a better timing for interruption. Like agents, context states are also arranged in a semi-lattice, with the state **doing anything** as its root. In this case, however, implication is bottom-up: if at least one descendent of a state is active, then that state is active as well. Thus, if a state like **debugging in Eclipse** is active, the states **working in Eclipse** and **doing anything** are also active.

The collections of features which constitute the configuration are populated when the interruption manager object is first created. For each kind of feature

described above, the framework first obtains all the *suppliers* for such features, and then asks each of them to provide the objects as well as their placement in the hierarchy. In our implementation of the framework, which relies on the *Eclipse* plug-in model, there is an *extension point* for each type of supplier. These suppliers are obtained by asking for all the extensions of a specified point. Of course, a particular plug-in can provide suppliers to multiple configuration features by extending several points.

Although the majority of features are loaded when the configuration is first created, they can also be added and removed at run time. This allows the system to adjust to changes such as programmers leaving and joining the team. Listeners are notified when such configuration changes occur.

Three additional kinds of features: *request generators*, *actions* and *handlers* participate in the interruption management process, which we shall now describe.

2.2. Interruption management process

We describe the process of submitting and handling interruptions when *GateKeeper* is operating on the recipient's machine. For our running example, the application making use of our framework is *Jazz* [2], an *Eclipse*-based collaborative development environment. It will handle instant messaging interruptions from team members. As we will show in Section 5, a plug-in has already contributed *Jazz*-specific configuration features, including an interruption type representing instant messages, and source agents representing every team member.

Jazz has a thread which listens for incoming chat requests from the server. When such a request is received, 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. It also attaches the contents of the first message to the request object, using a properties bag. 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.²

Upon receiving the request, the manager verifies the submitter against its list of pre-registered *interruption request generators*, and ensures that it had not previously turned blocked this specific submitter. This verification mechanism is intended to consolidate the lo-

cations from which requests can be submitted, as well as to allow the manager to disable faulty or malicious interruption sources. If the submitter is valid, the request is sent to the current interruption management scheme, which is treated 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, on the other hand, the request was denied, the thread does not open the window, and instead sends an instant message to the other party with an automated response informing of the user's unavailability.

Rather than fulfill accepted requests by themselves, client programs can also register *interruption handlers* for specific types. This allows, for instance, for different monitor agents to submit one-line alerts, counting on a single handler to display them to the user once they are allowed through. Such a design decouples agent programs which provide relevant information from the code responsible for presenting them to the user.

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. For example, *Jazz* registers itself as a listener to the manager, and decorates the icons of involved parties when an interruption is denied. 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* using schemes that deny interruptions and use actions as alternative means of notifications, or by postponing decisions until the cost of interruption is low enough. *Scheduled interruptions* are easy to implement by delaying decisions until the appropriate time, or by using handlers which wait before presenting the information to the user.

3. A sensor-based awareness framework

Successful interruption mediation depends on accurate determination of a user's context and interruptibility. To this end, it is important to identify the states or activities that may occur during the devel-

²We also support asynchronous submissions using callbacks.

opment process, and find means to determine whether the user is in a certain state at a given time.

Identifying relevant states, such as *debugging* or *writing new code*, is relatively straightforward. Determining their semantics, however, is an open research problem. For example, what does it mean that a person is debugging? If the user is currently stepping through a program using *Eclipse*'s debugger, the debugging state is probably active. However, debugging sometimes consists of running the application in standard mode, monitoring the output for specific strings. How do we accommodate that, and still differentiate between debugging and testing? Similarly, how do we distinguish between writing new code while extending the system, and modifying existing code for maintenance purposes? There might well be a different cost of interruption associated with each of these activities.

The current approach to determining activity in general office work or computing settings relies on the use of machine-learning or statistical techniques for a multitude of sensory inputs [5, 7, 8]. It is likely that the approach for software development settings will be similar, although much of the sensor data will have to come from within the IDE. In preparation for such work, we implemented a sub-framework for sensor-based context states as a plug-in to *GateKeeper*.

At the heart of this sub-framework is a plug-in for *GateKeeper* which provides it with a singleton *context supplier*. Remember that the supplier provides context state objects, which can then be queried for activity. This plug-in publishes two extension points and uses them to obtain and maintain collections of *atomic context information templates* and context states. Each template, identified by a unique tag, can be thought of as a simple sensor. The template is used to instantiate values, which can be boolean, selections from a predefined list, or free-form strings. Context states contributed via this framework can request the values for specific sensors, and use them to determine their own activity value.

For example, one plug-in which contributes to this framework provides atomic information templates for the *Eclipse* environment. These include the names of the current perspective and active project, boolean indications of whether there are any unsaved files, and the numbers of processes currently under debug mode and under run mode. This plug-in also provides several context states which rely on these templates, such as *debugging in Eclipse*. Based on our arbitrary decision, the activity of this state is evaluated as active if we are either in the debug perspective with at least one program in run or debug mode, or if we are in another perspective with at least one program under

debug mode.

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 choosing to use this plug-in, in spite of its privacy implications, can benefit from context states such as: *writing an email*, *using instant messaging*, etc. Such states are not displayed to others, but can be used by the user to fine-tune interruption preferences.

Of course, the evaluation of state activity does not have to be a simple condition on the atomic information values. Instead, we can use more advanced techniques, such as relying on historical data, or employing learning techniques.

Note that the sensor-based framework can also be configured to trap changes to atomic values. This enables the implementation of management schemes which provide mediated interruptions by delaying an interruption until the associated cost drops below a certain threshold.

4. A rule-based interruption management scheme

As part of the validation of the *GateKeeper* framework, we wanted to implement a nontrivial interruption management scheme which would automatically transform some interruptions from the distracting immediate style to the less-distracting negotiated style.

People have varying preferences for receiving interruptions under different circumstance. Our main design principle was therefore to provide a flexible and customizable scheme, allowing users to specify the conditions under which an interruption is permissible, and the form that different interruptions may take under different circumstances. In order not to overload users, we strove to minimize user interaction and not force them to configure the scheme. The degree of control over interruptions becomes proportional to the effort invested in customization.

To this end, we adopted the approach familiar from the filtering rules of e-mail clients, and implemented a rule-based scheme for controlling interruptions. It is important to note that we are not advocating a rule-based approach as the best solution to mediating interruptions. We would prefer, of course, a scheme that can automatically deduce the preferences of the user. Since building such a learning-scheme is not the focus of our work, we chose the rule-based approach as a compromise that adequately demonstrates the capabilities and flexibility of the *GateKeeper* framework. In addition to being intuitive to programmers, the rule-based approach also has the benefit of *accountability*, allow-

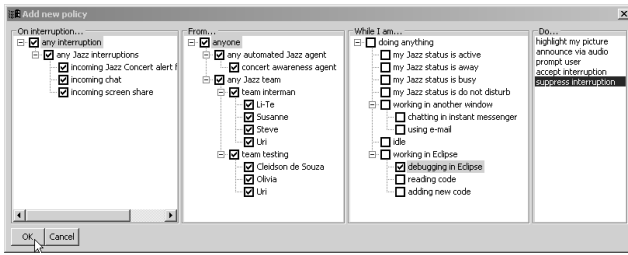


Figure 1. Adding a policy

ing the scheme to justify why it made certain decisions, making it easier to rectify problems.

Our scheme maintains a collection 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.

4.1 Scenario

To demonstrate how this rule-based system can be beneficial under real conditions, let us consider a scenario in which we are using the collaborative *Jazz* environment with interruption management support.

Suppose that during this morning’s group meeting, we were tasked with fixing a critical bug, and we wish not to be interrupted while doing so. We open the scheme editor, and choose to add a new policy. In the policy editor, shown in Figure 1, we choose to suppress all types of interruptions from all sources while we are debugging. Note that the editor is aware of implications, so we only need to check one box per category and the others become checked automatically. While the policy itself consists of all the selected nodes, textual descriptions of the policy use only the ancestor node names. Thus, this policy is described as: “*on any interruption from anyone while I am debugging in Eclipse, suppress interruption*”.

We soon realize that we are no longer alerted when coworkers are modifying their local versions of files which we are currently editing. The *Jazz concert awareness* component watches for such changes, and can alert us if a the potential for conflict arises. However, the distracting pop-up alerts which it usually generates are now blocked. We therefore add a second policy, instructing *Jazz* to sound an alarm when a resource alert arrives from the concert agent while we are debugging. This negotiated style of interruption without forcing us to deal with it immediately; we can examine

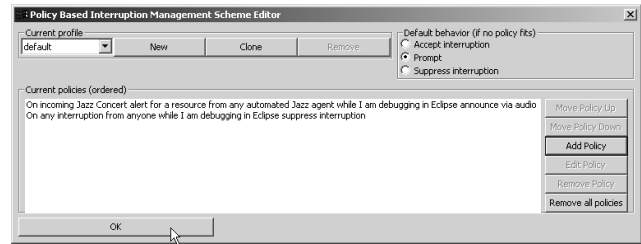


Figure 2. The scheme editor

the specifics of the alert at our convenience.

As we can see in Figure 2, the system has determined that the new policy is more specific than the original one, and thus placed it first in the list.

Later, our manager instructs us to provide immediate assistance, if required, to the *testing* team which has a pending deadline. We add a third policy, stating that all interruptions from that team should be accepted automatically. The system realizes that this new policy might conflict with the existing policies when we are debugging. Rather than arbitrarily placing it, the system asks for clarifications which will help decide its placement. It presents the situation where an interruption from the testing team arrives while we are debugging, and asks whether we want to completely suppress the interruption, as our original policy dictated, or whether we want to accept the interruption. We choose to accept the interruption, indicating that the new policy will be placed before the original policy. There is no conflict with the policy for the concert alert, and therefore the system places the new policy between the two.

We have finished debugging and have started writing new code. After a few minutes, a dialog pops up; it informs us of an incoming message from Steve, and asks whether to accept it. Puzzled as to why this interruption was not deferred, we ask the system to justify its decision. As we can see in Figure 3, the second policy did not apply to Steve, and the others only apply while we are debugging. The system has realized that we are no longer debugging, and therefore the request was not matched by any policy, and the default prompting action took place. In addition to making a decision, we can create a new policy directly from this dialog, making the system remember our decision.

Remember that 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: In the former, we add policies specifying the requests to block, letting the rest through. In the latter, we specify only those requests which should be accepted.

The *prompt* option, on the other hand, helps gently

Policy	Covers Type	Covers Source	Covers States
On incoming Jazz Concert alert for a resource from any automated Jazz agent while I am debugging in Eclipse announce ...	NO	NO	NO
On any interruption from team testing while I am doing anything except interruption	yes	NO	yes
On any interruption from anyone while I am debugging in Eclipse suppress interruption	yes	yes	NO

Figure 3. Justifying why an interruption request was not matched by the existing policies

introduce new users to the interruption management system. Upon receiving the first interruption, users get a chance to define a policy. This policy can be used to instruct the system to accept or reject all interruptions, so that the user never has to interact with *GateKeeper*. Preferably, however, it will be used as a first step towards constructing an elaborate set of policies.

4.2 Adapting to changes

Before concluding our discussion of the policy-based scheme, we demonstrate how interruption management schemes can handle configuration changes. Such changes are likely to occur not only between activations of the system, but also while it is running. For example, suppose that a new user joins the testing team. Since we have a policy that refers to this team and lists its members, it needs to be adjusted to include the new member. Another possible scenario is the addition of a new state, such as `working on project X`. We need to allow new policies to refer to this state, but also to adjust existing policies if it is ever removed.

Since it is unacceptable to simply erase all the policies every time the configuration changes, our implementation tries to resolve all problems automatically, at the risk of making some wrong decisions. The scheme is registered as a listener to configuration changes in the manager, and invokes an update sequence when it is notified of changes. This sequence is also triggered if configuration changes are noted when the policies are loaded from persistent storage when *Eclipse* is reactivated.

An update sequence consists of iterating over every profile, and adjusting every policy according to the rules which we shall now describe. It is important to note that these rules are arbitrary, although we believe they present an acceptable trade-off. When a new element is added as a child to an existing node, its siblings determine whether it will be selected: If all the siblings are selected, the new node is selected as well, but if at least one is not selected, then the new item is not selected either. The underlying reasoning is that if all the siblings are selected (and thus also their parent), it is likely that the policy focuses on the parent and does not

care about specific child nodes. If, on the other hand, there is at least one unselected sibling, it means that choices were made regarding specific child nodes, and we cannot infer anything regarding this new node. The adjustment which follows the removal of child nodes is simpler: If after the removal of a node all its siblings are selected, we select the parent as well.

Note that as a result of adjustments, certain policies may become identical to existing ones. In such cases, the system will ask for clarifications (if the actions are different) or will remove the duplicates.

5. Integration with a collaboration suite

So far we presented our framework and an implementation of an interruption management scheme. We now show how *GateKeeper* can be used in existing applications with minimal changes. This is demonstrated on *Jazz*, an eclipse-based collaboration tool.

5.1. The Jazz framework

IBM's *Jazz* project [2] enhances *Eclipse* with extensible collaboration tools for small teams of developers. Its client-side implementation consists of a set of plug-ins which interact with other *Jazz* clients via a shared-objects server. Figure 4 depicts the *Jazz*-enhanced *Eclipse* workbench, highlighting important features.

The Jazz band, marked *A* in Figure 4, is an iconic analogue of the *buddy lists* familiar from instant messaging clients. It shows the image of the user, followed by images of coworkers, grouped by teams. The images of disconnected users are grayed out, while those of connected users are decorated with icons and status messages that provide some awareness of the coworker's activity. Communications with peers can be initiated by clicking their images in the band. Supported forms of communication include an instant messaging session (marked *B*), voice chat, and screen sharing. Collaborations can also be started from within editors: one can select a particular code section and "chat around it"; the chat transcripts can then be saved along with

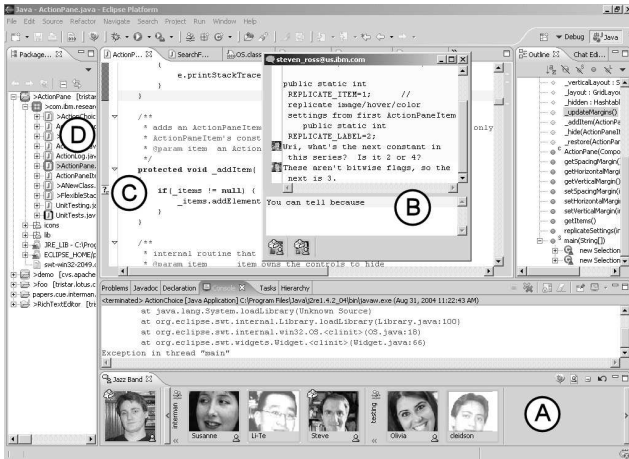


Figure 4. A Jazz-enhanced Eclipse Workbench

the code (marked *C*), serving as a “writing on the wall” for future readers.

Like other collaboration tools, *Jazz* also provides integration with source-control systems using its *concert awareness* module. Resources being modified by other team members are highlighted according to their risk for merge conflicts (*D*). It is also possible to set alerts for changes to particular files.

5.2. Adding interruption management capabilities to Jazz

Like most *Eclipse*-based frameworks, *Jazz* consists of a central core plug-in which defines extension points, interfaces, and an API, and from multiple plug-ins which extend these points and provide concrete feature implementations. The core object has skeleton methods for many *Jazz* features, but relies on extensions for actually implementing them. For example, the method receiving incoming chat events creates a new thread which instantiates a chat client from an extension and starts it.

In accordance with this design, we strove to minimize changes to existing *Jazz* code, limiting them to the skeleton methods described above. We needed to allow the normal operation of *Jazz* without requiring the installation of interruption management, and to allow *GateKeeper* to work even if fellow *Jazz* users do not have it installed.

To this end, we added an *interruption management* extension point to the *Jazz* core, with a boolean method for each relevant interruption kind in the associated interface. We then proceeded to change the code of the skeleton methods: instead of immediately instantiating an appropriate extension and instructing it to handle the incoming event, the new code first at-

tempts to find an interruption management extension and, if found, ask it whether to allow the interruption through. We described this process for the example of chats in Section 2.2. No further changes were necessary for the existing *Jazz* code and plug-ins.

The *Jazz* and *GateKeeper* frameworks are independent of one another: The interruption management extension point, which we just described, buffers *Jazz* from *GateKeeper*. *GateKeeper*, by itself, does not have *Jazz*-specific features. The features which connect both frameworks are provided by an additional *Jazz connectivity* plug-in.

First, the connectivity plug-in extends *GateKeeper* with several configuration features: An interruption type is contributed for every kind of interruption supported by *Jazz*. A sub-hierarchy of agents is contributed to the existing hierarchy, organized by teams and then by users. A new action-on-interruption object allows the images of users to be flashed when an interruption request has been declined.

Next, the plug-in implements the new interruption management extension point of *Jazz*. For each *Jazz* feature, the corresponding method creates an appropriate interruption request event object, and submits it to the *GateKeeper* interruption manager for a decision. It also registers itself as a listener for decision events, so that when a request is denied, it decorates the pictures of the user and of the coworker who sent the request with an icon indicating that there are waiting messages. Finally, it contributes to the *Jazz* band pop-up menu, adding options for changing profiles, editing the scheme, and viewing incoming messages.

In addition to the main *Jazz* connectivity plug-in, two additional plug-ins tie *Jazz* and *GateKeeper*. One extends the sensor-based context framework of Sec. 3 with sensors that tracks the *Jazz* status of the user, and context states representing each of the predefined *Jazz* user states. The other plug-in extends the *Jazz* user study plug-in, which logs different *Jazz* events for a preliminary user study being carried out at IBM. This extension, registered as a listener to the interruption manager, contributes interruption management events and configuration changes to the study log for future analysis.

6. Conclusions and future research

In this paper we presented the need for research on interruption management in software development, and described the *GateKeeper* framework which strives to remove a significant barrier for the conduction of such research.

Our discussion showed how the framework meets the

design requirements which we presented in the introduction: *GateKeeper* is implemented within the popular *Eclipse* environment, although it could easily be converted to the extension models used by proprietary IDEs. The plug-in model and the use of internal hierarchies allows us to aggregate interruptions and awareness information from multiple sources, and manage interruptions at different levels of abstraction. The rule-based scheme demonstrated how the framework accommodates changes that occur at runtime, while the *Jazz* integration demonstrated different notification mechanisms and the use of observation hooks.

This paper demonstrated how awareness and decision models can be contributed, and how an existing application can use *GateKeeper* with minimal changes. Even though we described a fully-local decision model, it is straightforward to implement a simple local model which defers to a centralized decision model running on a server. Furthermore, in some applications it might be possible to route all interruptions through a centralized coordination server so that blocked interruptions never reach the user.

It is our hope that the capabilities of the *GateKeeper* framework will assist in research for obtaining the design knowledge needed for the creation of better mediation tools for software developers.

Our research proceeds in four directions: First, we are collecting and analyzing data from the *Jazz* user study to learn how programmers collaborate and use the interruption management capabilities which we implemented. Second, we are collecting data from multiple sources within *Eclipse*, in an attempt to characterize programmer activities. Third, we are investigating the semantic connections between artifacts of the software development process, in order to improve the accuracy of interruption mediation. Finally, we are considering the problem of centralized interruption management: We are investigating whether an automated agent with knowledge of all the users and their activities might be able to orchestrate collaboration and mediate interruptions better than a local agent working on behalf of an individual.

References

- [1] I. Burmistrov and A. Leonova. Do interrupted users work faster or slower? In *Proceedings of the 10th International Conference on Human-Computer Interaction*, pages 621–625, Crete, Greece, June 2003.
- [2] L.-T. Cheng, C. R. de Souza, S. Hupfer, J. Patterson, and S. Ross. Building collaboration into ides. *Queue*, 1(9):40–50, 2004.
- [3] D. Cubranic and G. C. Murphy. Hipikat: recommending pertinent software development artifacts. In *Proceedings of the 25th international conference on Software engineering*, pages 408–418. IEEE Computer Society, 2003.
- [4] Eclipse project homepage. <http://www.eclipse.org>.
- [5] J. Fogarty, S. E. Hudson, and J. Lai. Examining the robustness of sensor-based statistical models of human interruptibility. In *Proceedings of the 2004 conference on Human factors in computing systems*, pages 207–214. ACM Press, 2004.
- [6] E. Horvitz and J. Apacible. Learning and reasoning about interruption. In *Proceedings of the 5th international conference on Multimodal interfaces*, pages 20–27. ACM Press, 2003.
- [7] 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.
- [8] 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 *Proceedings of the conference on Human factors in computing systems*, pages 257–264. ACM Press, 2003.
- [9] S. E. Hudson and I. Smith. Techniques for addressing fundamental privacy and disruption tradeoffs in awareness support systems. In *Proceedings of the 1996 ACM conference on Computer supported cooperative work*, pages 248–257. ACM Press, 1996.
- [10] IBM Websphere Studio Application Developer homepage. <http://www.ibm.com/software/awdtools/studioappdev>.
- [11] R. E. Kraut and L. A. Streeter. Coordination in software development. *Commun. ACM*, 38(3):69–81, 1995.
- [12] 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.
- [13] Rational Rose XDE homepage. <http://www.ibm.com/software/awdtools/developer/rosexde>.
- [14] 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 *Proceedings of the 2004 conference on Human factors in computing systems*, pages 287–294. ACM Press, 2004.
- [15] P. N. Robillard, P. d’Astous, F. Detienne, and W. Visser. Measuring cognitive activities in software engineering. In *Proceedings of the 20th international conference on Software engineering*, pages 292–299. IEEE Computer Society, 1998.
- [16] A. Sarma, Z. Noroozi, and A. van der Hoek. Palantir: raising awareness among configuration management workspaces. In *Proceedings of the 25th international conference on Software engineering*, pages 444–454. IEEE Computer Society, 2003.