Chair of
Software Engineering

Monitors: Keeping Informed on Code Changes

Independent Research Study

By: Christopher Dentel

Supervised by: Christian Estler
Dr. Martin Nordio
Prof. Dr. Bertrand Meyer

Student Number: 11-909-868

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

inf | Informatik
Computer Science

ABSTRACT

In small, localized teams information proliferation about ongoing code changes is a natural consequence of the immediate proximity of the developers. However, larger projects and especially distributed projects face challenges where developers may not be aware of all changes that are occurring within a specific project. This report introduces the concept of Monitors, the ability to keep an eye on those changes which are important to a developer. A "monitor" is added to a particular "aspect" of code (library, file, class, feature, pre/post condition, invariant, etc.). When the monitor is added, a "shadow" of the current state of the monitored aspect is archived. The degree to which the aspect should be monitored is specified within a "comparator," which continuously compares the current state of the aspect as well as the shadow to determine if a "violation" has occurred. If a violation has occurred, the monitor informs the developer through one or more specified communication means. This report also documents the implementation of monitors into the Integrated Development Environment and toolset provided by Cloudstudio.

Table of Contents

Cloudstudio

Motivation

For most large projects and teams, using an Integrated Development Environment (IDE) is a necessity of software development. But traditional desktop software IDEs do not seek to accommodate teams who are spread out across multiple countries, time zones, languages, and cultures. These are the challenges faced by those organizations which pursue global software development (GSD). The challenges that GSD poses are not necessarily new, and approaches to alleviate the difficulties that come with GSD have been previously investigated with many different approaches. Some investigations have focused on comparing different project management approaches, including agile vs. structured development [1]. Carmel and Agarwal [2] investigated means to reducing the "distance" between teams (national, organizational, cultural, and temporal distances) and reducing collaboration. Several other investigations have not sought to avoid collaboration but instead focused on how to better facilitate collaboration across time-zones [3] [4] [5]. One such study [6] utilized the Distributed and Outsourced Software Engineering course (DOSE), [7] [8] using some of the technologies presented in this paper.

Existing IDEs rely on configuration management that leads to disparities in information awareness, such as discovering at commit that two major refactorings have occurred simultaneously. While this style of configuration management seeks to isolate developers from the changes that other developers are making, Cloudstudio[a] seeks to share information in real-time. As online document collaboration websites are eliminating the need to email documents of varying revisions back and forth, Cloudstudio seeks to allow developers to work simultaneously on a project, sharing information between themselves as they like, while also maintaining the isolation that traditional configuration management affords.

Features

Cloudstudio [9] is a web based IDE, allowing developers to access their projects at any time from any machine. This move to the web eliminates the need to maintain and update different versions of software on local machines, while also allowing developers to work where they want, when they want. But Cloudstudio is not just a web-app clone of an existing IDE. Cloudstudio seeks specifically to meet the needs of developing in distributed environments through smarter configuration management and tool integration. While still only a web-app, Cloudstudio integrates development tools, collaboration tools, and verification tools. Some such tools are listed below.

Development:

- *Languages* – Cloudstudio supports projects in Eiffel, Java, C#, and JavaScript.
- *Configuration management* – Cloudstudio's configuration management system encourages developers to share early and share often. Developers commit to their own private branches and chose to share their changes when they wish.

[a] Try out Cloudstudio at: http://www.cloudstudio.ethz.ch

- *External Development* – Cloudstudio's configuration management system allows developers who do not wish to use Cloudstudio to still contribute to projects. As Git is the underlying backend for the project, developers can directly connect with the repository and work without being bound to the IDE.
- *Monitors* – Monitors provide an early warning system for developers and allow them to keep track of the changes occurring in a project that are important to them. This was developed and implemented in this report.

Collaboration:

- *Chat / Skype* – Cloudstudio allows developers to see what other developers are currently working on the same project, and provides access to both chat and Skype from the IDE.
- *Code Reviews* – Code reviews are fully integrated into the IDE, allowing developers to invite their team members to discuss changes without leaving the IDE.
- *Notifications* – All tools have access to a news / notification system, which keeps developers up to date on what is going on in their project [10]. The system is highly customizable, and will be discussed in more detail later in this report.
- *Document Sharing (In progress)* – Teams will be able to collaborate on non-code documents, and see each other's results in real-time.

Verification and Testing:

- Auto Proof – Auto Proof is a static verification tool for Eiffel which allows for proving Eiffel programs in the browser without the need for any additional specifications [11] [12]. Postconditions are tested against possible preconditions to determine if there are cases in which satisfactory preconditions yield unsatisfactory post conditions.
- *Auto Test* – An entirely automated unit-testing suite which infers tests based off contracts [13] [14]. Developers select how long they wish to run the suite for, and Auto Test exercises the classes to test the bounds of the contract.
- *Auto Fix (integration with Cloudstudio in progress)* – While Auto Test tests the bounds of the contracts of a given class and reports failures, Auto Fix will attempt to generate fixes for the errors found [15] [16]. It uses a combination of both static and dynamic analysis to generate fixes, and then regression tests the fixes to determine if they are a suitable candidate to fix the error found.

Details of Implementation

Cloudstudio is developed using Google Web Toolkit and is deployable as an app-engine app. Cloudstudio's editor is an Eiffel program which has been compiled to JavaScript via an Eiffel to JavaScript compiler developed by Alexandru Dima [17]. For back-end data storage, Cloudstudio uses MySQL.

Cloudstudio is being developed at ETH Zürich by the Chair of Software Engineering. Cloudstudio's principal members include Professor Bertrand Meyer, Dr. Martin Nordio, and

Christian Estler. Over 13 masters and bachelors students from several universities have also been involved in implementing Cloudstudio.[b]

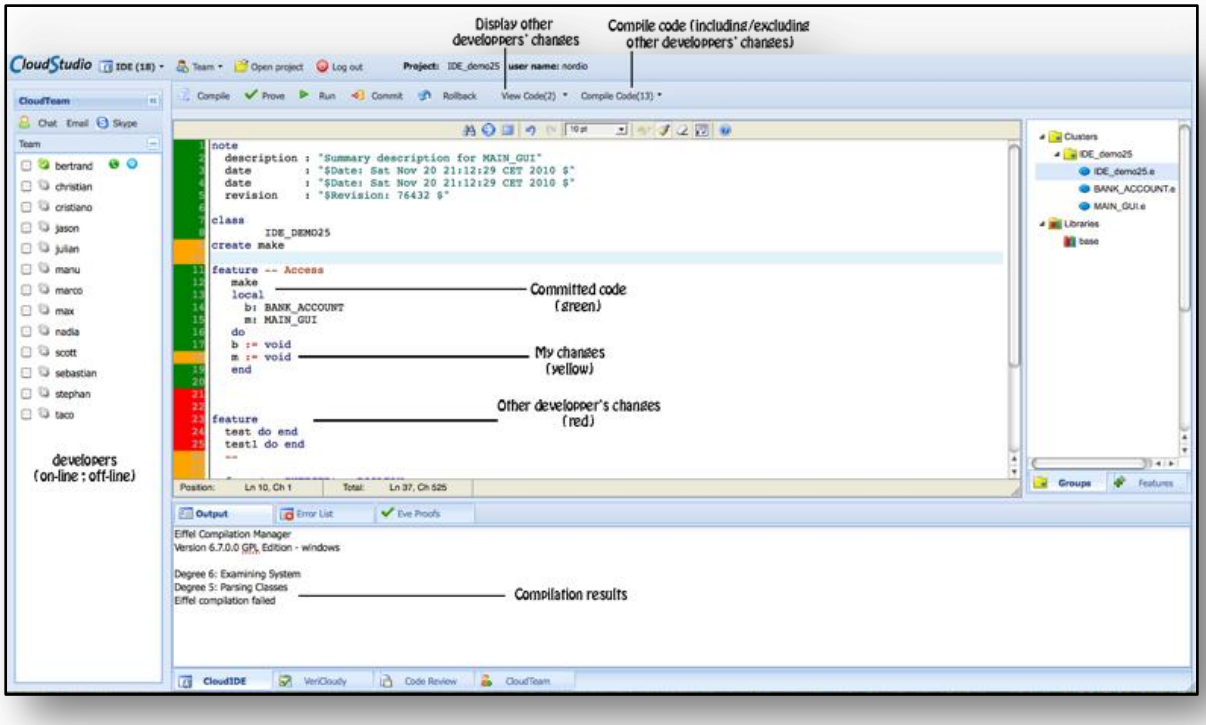

Figure 1: Cloudstudio IDE view.[c]

As is evidenced from the previously mentioned features, Cloudstudio seeks to expand upon the functionalities which are necessary for development while also adding in the features that promote the collaboration necessary to be successful in a distributed development environment. Tools like chat integration, integrated code-reviews, notifications, and monitors bridge the gap that occurs when knowledge cannot naturally circulate through teams that are centrally located. The features in Cloudstudio are tightly coupled yet also flexible, allowing developers to take advantage of the features that benefit them, while not shackling them to the entire suite of tools.

[b] To learn more about the Cloudstudio development effort, visit: http://se.inf.ethz.ch/research/cloudstudio/
[c] Graphic from: http://se.inf.ethz.ch/research/Cloudstudio/

Monitors

Motivation

In small, localized teams information proliferation about ongoing code changes is a natural consequence of the immediate proximity of the developers. However, larger projects and especially distributed projects face challenges where developers may not be aware of all changes that are occurring within a specific project. Espinosa et al. [18] study this phenomenon in the context of a shared mental model which exists across team-members. Their research suggested that having a shared mental model across team-members and familiarity of each other's work yields a positive effect on distributed development. A large, distributed project is not conducive to developing shared mental models due to the lack of implicit interaction. In addition distribution makes it less likely that members will be familiar with their other members' work. These deficiencies lead to issues such as the revision of existing code by one developer causing a breaking change for another developer who is a client of such code. If a change does not result in compilation errors or the failure of an automated testing suite, this breaking change may be released without either developer becoming aware of any potential issue. While the change may have been recorded in the change-log of the project's configuration management system, it is largely impractical to expect every developer to scour all changes for those which may impact their past and present projects.

At this point, it is useful to introduce two relationships a developer may have with a piece of code. A developer may be an author or owner of the piece of code, and the extent to which other developers may change that piece of code will vary with the team's code ownership practices. However, even in the most agile teams with collective code ownership, developers tend to gain a sphere of expertise in a particular region of code. These developers will be most apt to understand possible complications that could arise from change. Subsequently, these individuals will be referred to as a piece of code's steward, regardless of whether they are the author, owner, or developer with the best understanding of the code's working.

Developers working on a team also inevitably become clients of other developers' code. These clients are reliant on their suppliers, and a bug introduced by a supplier can cripple the client's code. A revision or extension by a supplier in order to facilitate the release of a new feature may cripple already existing features. While automated and thorough integration testing would hopefully reveal any defects introduced in such a revision, it would still be useful for a client developer to remain informed about effects that may occur to their past or present projects.

In both the situation of the steward and the client, better dissemination of information about changes occurring to the code base would be beneficial in multiple ways. Once informed a steward can provide their opinion about how changes may have unintentional consequences or why a library or piece of code is written the way that it is. An informed client may be able to find issues in a supplier revision that could affect all clients of the revised code. In both these cases, greater awareness of change leads to avoiding bugs as well as sharing expertise between developers. This benefit can only be reaped if there exists a way for developers to remain informed.

What is a Monitor?

Currently, there are many sources through which developers keep informed of changes being made by other developers. Code reviews provide the opportunity for team members to see changes being made by other developers and to voice their opinion on changes being implemented. However, in most organizations, it would be impractical for all developers to participate in such a review. In addition, there is no way for a developer to guarantee that they will be invited to a code review if a section of code which interests them is modified. As mentioned earlier, developers could also comb through configuration management change logs to look for changes which may affect them. Both of these solutions do not allow for automated "monitoring" for changes which interest them.

Phabricator[d], an open source suite of development tools developed by Evan Priestly while at Facebook, contains a small feature which is a starting point for automated monitoring. Phabricator contains a tool within it called Herald which "allows you to write processing rules that take effect when objects are created or updated.[e]" Herald rules are similar to mail processing rules, in that you want to do an action when a commit that matches a rule occurs. Rules can be used to match class names, authors, or a regular expression match within the commit. Actions typically include automatically adding yourself to a code review. The idea of monitors presented in this paper builds upon this but in a more general way, and introduces several ideas to make monitoring smarter.

At this point, the components of a monitor will be introduced. A monitor contains:

- An owner
- An aspect being monitored
- A comparator (which specifies what changes to the aspect would "violate" the monitor)
- A shadow (archive of content when monitor is introduced)
- A set of notifications to be sent when the comparator detects a violation.

Each component will be explored in further depth.

Owner: This is the person (or group of people) whom the monitor will notify upon detection of a violation.

Aspect: The aspect being monitored can be of any number of granularities. A monitor could be placed at the level of a library, source file, class, feature, pre/post condition, invariant, or other granularity. The aspect only defines the scope of code which will be monitored, not the particular changes which would violate it, and aspects are hierarchical. For instance, the class aspect may be composed of the following other aspects:

- Class
    a. Inheritance Clause(s)
    b. Feature(s)

[d] To learn more about phabricator, visit phabricator.org
[e] A discussion of how to use herald rules:
http://phabricator.com/docs/phabricator/article/Herald_User_Guide.html

    i. Pre-condition(s)
    ii. Body
    iii. Post-condition(s)
c. Invariant(s)

When choosing to monitor a class, monitors could be applied to any combination of aspects which will define that class.

Comparator: The comparator is what determines if a monitored aspect has been violated. While the aspect determines only the relevant code section, the comparator is the set of rules which would constitute a violation. This process is determined by comparing the original state of the aspect (shadow) to its present state. Comparators can range from a simple diff, such as monitoring whether or not the contents of an entire file have changed, or be more complex and involve static analysis. We can imagine the comparator of a precondition aspect analyzing the precondition to see if it has been strengthened, and reporting a violation only in this event.

Shadow: The shadow is the encapsulation of the state at the time at which the monitor is created (or recreated). What is contained in a shadow depends on both the monitored aspect and the comparator. The comparator will use the shadow and the present state to perform its analysis. In the previously mentioned example of detecting any modifications to a file, the shadow could be the old version of the file.

Notification Set: In the event that the comparator does detect a violation, the owner should be informed of the violation. Notifications could be dispatched through any variety of mechanisms, from an email to an automated text message. Mailing lists could also be the target of such a notification, or perhaps even a code review service which automatically initializes a code review of the violated aspect.

In summary: a "monitor" is added to a particular "aspect" of code (library, file, class, feature, pre/post condition, invariant, etc.). When the monitor is added, a "shadow" of the current state of the monitored aspect is archived. The degree to which the aspect should be monitored is specified within a "comparator," which continuously compares the current state of the aspect as well as the shadow to determine if a "violation" has occurred. If a violation has occurred, the monitor informs the developer through one or more specified communication means.

Monitors in Cloudstudio

Final Product

Cloudstudio proved to be an excellent environment to explore the usability as well as the feasibility of implementing monitors as part of an integrated suite of development tools. While this paper comprehensively defines monitors and explores their role in a distributed environment, the final implementation of monitors in Cloudstudio focused on a smaller set of features.

The 5 previously defined aspects of a monitor as they exist in Cloudstudio are outlined below:

- **Owner**: Monitors are owned by one and only one developer. Developers are responsible to place their monitors on those classes important to them, and monitors are not shared across developers.
- **Aspect**: Monitors in Cloudstudio are done at the file level. Any text file under configuration management in the project can be monitored.
- **Comparator**: There exist four possible comparators in the implementation for this report. The comparators are the product of two different monitoring options: Monitoring live changes and ignoring whitespace and comments. With neither of these options selected, the current version of the file in the repository is compared against the shadow. If any changes are detected, then the file is considered to be violated. When whitespace and comments are ignored, any insertions, deletions, or modifications to whitespace or comments (including comments on the same line as non-comment code) are not considered to be violations. The final feature, monitoring live changes, allows developers to detect violations that are occurring on branches that have not yet been merged back into the original repository. This can be combined with ignoring whitespace and comments
- **Shadow**: For the shadow, the committed version of the text was extracted from the configuration management system and saved in the datastore.
- **Notification**: The monitors in Cloudstudio make use of Cloudstudio's integrated notification system [10]. Developers receive a notification immediately when the violation is detected (when another developer either shares or commits their work to the repository, depending on the monitor's comparator). Developers receive monitor violation notifications when they first log in for any monitors that were violated since they last were logged into Cloudstudio.

While this monitoring system is sufficient to detect monitors, an interface had to be developed which would allow developers to interface with the monitoring system. The additional features developed in this report are described below

- **Adding Monitors:** Developers may add multiple monitors at a time through a modal dialogue which presents to them all unmonitored file in the project. Developers select those files they wish to monitor with checkboxes. At the time of selecting what files to monitor, developers also customize the comparator that will be used to detect violations (choosing whether or not monitor live changes and to monitor whitespace and comments).

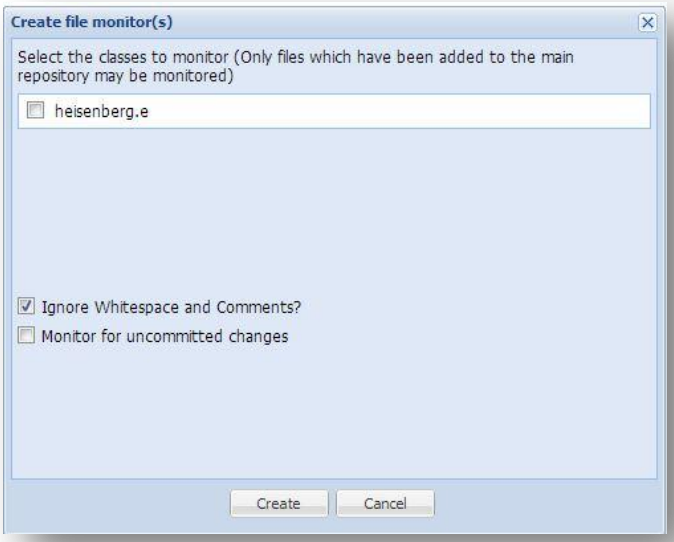

Figure 2: Modal add dialogue

- **Removing monitors:** Should a developer decide that they no longer need to monitor a file, they may remove the monitor through a modal dialog similar to the add dialog
- **Dismissing monitors:** When a monitor is violated, the monitoring developer may determine that the changes made to file do not require a roll-back and are satisfactory. In this case, the developer would like to set a new baseline for monitoring. Dismissing the monitor takes a current snapshot of the state of the monitored aspect, and uses this as the new shadow going forward. Optionally a developer may determine that all violated monitors are dismissible and do so as one action.
- **Monitored Files View:** This window displays what files are being monitored as well as the current status. Violated monitors appear in red, and unviolated monitors in green (See Figure 3).

Figure 3: Monitored File Menu

- **Monitor Diff View:** This feature (Figure 4) dominates the screen when using the monitor tool. It displays side by side the shadow version of the file with one of the current versions of the file. If the monitoring developer has opted to monitor live changes, then there will exist more than one current version. In this case, the developer

selects from a drop down box, which displays both the version they will be comparing against the shadow as well as the status of that version (violated or not).

The diff view itself shows the two versions of the file side by side and shows what changes have occurred in the file. One column shows a mark representing the type of change (+ for insertion, – for deletion, and ! for changes). In addition, the diff view provides different colorings to easily identify the changes. Insertions are marked in green, deletions in red, and changes in yellow.

The diff view also provides syntax highlighting.

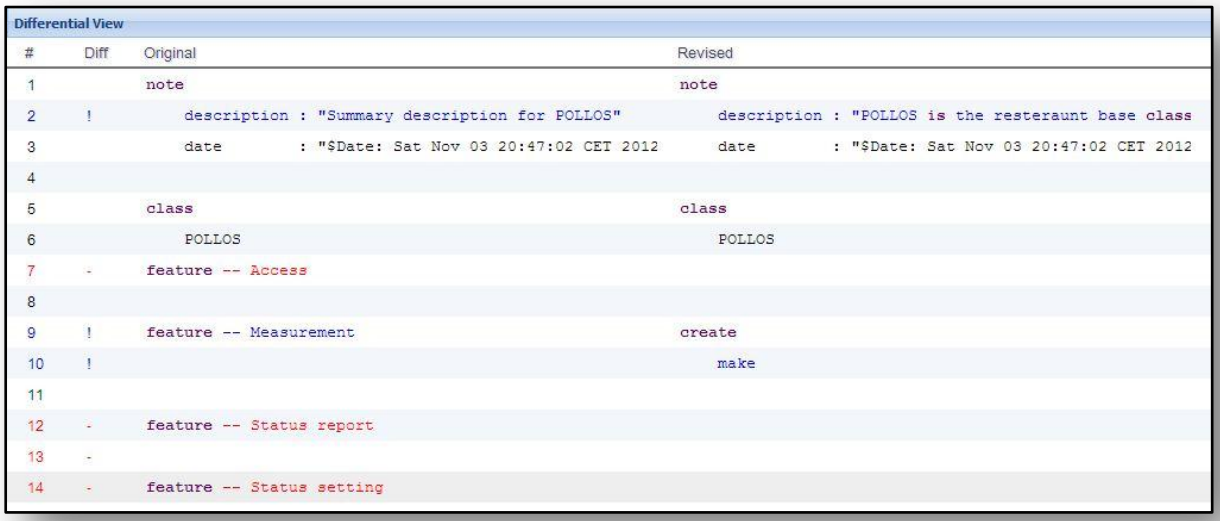

Figure 4: Diff View

Analysis / Effectiveness

The implementation previously described serves as an excellent proof of concept and exploratory device. While more advanced monitors will provide a higher granularity of specificity, this report's implementation would still be useful in a large portion of development cases.

Almost as important as the implementation of the monitors was the implementation of the tools that facilitate the use of the monitors. The diff layout view is monitor agnostic (as it could be used to diff any two versions of a file), but serves its purpose very well for inspecting what changes were made in a violated file.

The comparators implemented are both useful in different ways. Ignoring whitespace and comments provides an example of picking and choosing within a file what components will be monitored and what will be ignored. However, the ability to monitor live changes is perhaps the most useful feature of this implementation. Monitoring live changes can help alleviate the agony of merge conflicts, as developers can monitor those classes with which they are working to determine if any other work is going on simultaneously. Being able to detect situations that will lead to a merge conflict early on could greatly improve developer productivity [19]. In addition, with monitoring live changes developers can keep an eye on those pieces of code that they are stewards of. Instead of having to wait until a code review is requested, the steward can initiate a conversation before too much time has been sunk into changes which will ultimately not be approved.

While being able to preempt merge conflicts and changes that will ultimately be rejected is useful, being able to "spy" on other developers' private branches could have a detrimental effect on a project. Developers may not wish to step out of their own team's code for fear that someone will confront them the moment they make a change to the file. Also, the monitoring developer may end up seeing changes before they are complete, which does not give the modifying developer a chance to refine their work and present it once it is finished. In addition to this, if a developer is only temporarily modifying a file (perhaps to debug another file) monitors could be violated causing the steward of the monitored code to get involved. While these situations are unfortunate consequences of monitoring live changes, the detriment they contribute is largely outweighed by the detect issues in their infancy. In the "Further Work" portion of this paper, a way to alleviate the deficiencies of monitoring live changes is presented.

One of the limitations of the monitor system implemented is the lack of more granular aspect control. While monitoring at the file level is surely beneficial in some cases, being able to monitor specific methods or entire packages also have their use cases as well. An ideal implementation of monitors would allow monitoring at all levels of granularity. However adding levels of granularity has diminishing returns. Less granular levels reduce the amount of monitors that a developer must place, and finer granularity levels reduce the likelihood of false-violations by allowing the developer to target more specific segments. In other words, having only file monitors provides all of the functionality that other granularities of monitors would allow in terms of detecting violations, but it requires more work on the part of the developer. This is detrimental, as tools which are difficult to use are often not used at all. In order to determine

those granularities that are most useful, further study would need to be done, which is recommended in the further work portion of this report.

The inefficiencies of the configuration management system used in Cloudstudio resulted in duplication when archiving shadows. The configuration management system, presented in Section 4.2 of "Collaborative Software Development on the Web" [9], does not keep a history of diffs, but instead stored a semi-raw file in the MySQL database. When file monitors were implemented for this configuration management system, a physical copy of the shadow had to be saved, as the configuration management did not keep track of any previous state. In addition all of the developers' file versions were stored together, which made for tedious reconstruction to check for violations of all of the monitored versions of the files. However work by Sandra Weber [20], addressed these deficiencies, and the migration of the file system presented here to this new configuration management system is addressed in another report [21].

In addition to the difficulties faced in detecting violations, the configuration management system also posed problems during version rollbacks. As all versions of a line were stored together, it became impossible at rollback time to determine what lines to revert. As a result, a rollback issued by one developer also destroys the current uncommitted progress of all other developers for that class.

As the development and implementation of the monitors progressed, it became apparent that meta-information would be necessary to supplement the essential monitor components. The version presented in this report does not provide any access mechanism for developers to uncover when and why they placed a monitor. In addition, the comparators that the developer selects are not modifiable after initiation. This does not allow developers to be able to fine tune their monitoring preferences over time. These deficiencies are rectified and documented in another report [21].

Finally, one of the best aspects of monitor system developed is its integration with the other tools. Monitors are automatically evaluated for violation when a developer makes a commit from the IDE, and notifications of violations are presented. This allows developers to immediately know when a violation has occurred. With the first implementation that was done, violations were not brought to the surface, and a developer would have to navigate to the monitor tool to check to see if any of their monitors had been violated. Should a developer not check their monitors frequently, the whole benefit of monitors being able to detect potentially problematic changes early is lost. The concurrently developed notification system [10] was provides an answer to this and the need for other tools to be able to share information. Using it, the monitor system is able to effectively broadcast when violations have occurred.

While the existing implementation does introduce monitors to Cloudstudio, it is obvious that there is much more that could bring monitors to the vision presented at the start of this report. Many of these ideas will be presented later in the "Further Work" portion of this paper. Most of these suggestions are additional aspects, additional comparators, or deeper integrations with existing Cloudstudio tools.

Implementation

As with most of Cloudstudio, file monitors are implemented in Java for Google Web Toolkit, a suite of technologies which allow developers to program web apps entirely in Java.[f] Code is packaged into client, shared, or server code packages. Client and shared code is compiled to JavaScript and displayable in the browser. However, only a subset of the Java library may be compiled in this way. There is no restriction on what code may execute on the server.

SQL storage

The storage footprint of file monitors is rather high, mostly due to the fact that the shadow is a copy of the full text of the file. All of the monitoring data is contained in a single table with 10 columns. These columns track:

- The file path, name, and contents at the time that the monitor was placed or last updated,
- The user who is monitoring the file,
- The description that the user has associated with the monitor,
- Comparator options (Boolean options for whether or not to monitor whitespace and comments and to monitor live changes),
- A timestamp of when the revision hash was last updated.

Access to the database is controlled via a database table *FileMonitorTable*, which controlled insertion, retrieval and updating of values. This database table primarily interfaces with the service layer via *FileMonitorModel* instances.

Models

FileMonitorModel and *FileMonitorModelWithDiffs* provide the interface between the datastore, the service layer, and the application layer. These classes are serializable and transmitted via RPC to the frontend where they are then presented.

FileMonitorModel is a very simple mutable encapsulation of the datastore fields, providing getters for all fields and some setters. In addition to the data stored in the datastore, a *FileMonitorModel* also contains one more piece of information: whether or not the model that this model represents is violated. This model does not specify what exactly is violated about this monitor, but rather only if it is violated.

FileMonitorModelWithDiffs, on the other hand, is the more detailed child of *FileMonitorModel*. It is a sub-type of *FileMonitorModel*, and adds additional information about how a monitor is violated. A *FileMonitorModelWithDiffs* does satisfy the Liskov Substitution Principle, but for performance reasons it should not always be used. A *FileMonitorModelWithDiffs* may contain a *FileMonitorDiff* for every single other developer on the project as well as a *FileMonitorDiff* for the committed version. As will be explained next, each of these diffs contains two different versions of the file, which is why *FileMonitorModel*s should be the preferable way of transmitting information about monitors from the datastore to the frontend.

[f] To learn more, visit: https://developers.google.com/web-toolkit/overview

FileMonitorDiff encapsulates the differences between two different versions of a file. One of these versions will always be the shadow, and the other version is either a developer's current version or the master branches version. These are differentiated, as every file monitor tracks the master branches version, but only those monitoring live changes will be concerned with other developer's changes. *FileMonitorDiff* provide several getter functions that are also available in *FileMonitorModel* and *FileMonitorModelWithDiffs*, such as retrieving the file path and the monitoring user id. *FileMonitorDiffs* are used on the server side to determine if *FileMonitorModel*s are violated, as they contain the actual diff. This diff is represented as a list of *CodeLineDiff*s, each of which is responsible for the difference that has occurred in a single line of code. The comparator logic is executed over these *CodeLineDiff*s to determine if a violation has occurred.

```
private boolean processForViolation() {
        boolean violated = false;
        for (CodeLineDiff cld : data) {
                if (cld.getDiffMark().equals(CodeLineDiff.EQUAL)) {
                        continue;
                } else if (!cld.getDiffMark().equals(CodeLineDiff.EQUAL) && monitorWhitespaceAndComments) {
                        violated = true;
                        break;
                } else if (cld.getDiffMark().equals(CodeLineDiff.DELETED)) {
                        // If the line was not EITHER all blank OR a comment, deleting it is a violation
                        if (!cld.original.trim().isEmpty() || !cld.original.trim().startsWith("--")) {
                                violated = true;
                                break;
                        }
                } else if (cld.getDiffMark().equals(CodeLineDiff.INSERTED)) {
                        // If the new line is not EITHER empty OR starts with "--", then it is a violation
                        if (!cld.revised.trim().isEmpty() || !cld.revised.trim().startsWith("--")) {
                                violated = true;
                                break;
                        }
                } else if (cld.getDiffMark().equals(CodeLineDiff.CHANGED)) {
                        // If it was not EITHER a whitespace change OR the change of an end line comment
                        if(!cld.revised.trim().equals(cld.original.trim()) ||
                        (!cld.original.split("--")[0].trim().equals(cld.revised.split("--")[0].trim()))) {
                                violated = true;
                                break;
                        }
                }

        }
        return violated;
}
```

Figure 5 Selected algorithm showing comparator logic for ignoring whitespace and comments

*CodeLineDiff*s are the lowest level representation of the diff and contain the information that the front end will eventually display in the *DiffMonitorView*. The object consists of the content of the original line, the content of the new version of the line, what diff tag to associate with the line { +, –, !}, and the line number of the file.

The diffs are computed using Java-Diff-Utils, which was developed by Dmitry Naumenko.[g]

Service Layer

A fairly light service layer interfaces between the front end and the datastore to provide access via RPC. *MonitorServiceImpl* is this layer, and it is a Google-Web-Toolkit

[g] Available online under GNU Public license at: http://code.google.com/p/java-diff-utils/

RemoteServiceServlet.[h] This class implements an interface which the front-end accesses asynchronously. It provides a wrapper around *FileMonitorTable* and performs additional processes and lookups in other data tables before invoking requests at the data table level.

Front end

The front end is implemented in 3 packages, **presenter**, **view**, and **event**. The presenters contain most all of the application logic necessary for user interaction. The view package contains classes which are displayable components that the presenter presents. The event package contains several events which are used to communication across the various presenters.

The view components are implemented by composing Google Web-Toolkit's existing design assets as well as through using Sencha's Java UI component library for Google Web-Toolkit (GXT)[i]. GXT provides more powerful tools that were necessary for some of the more complex views, like the *DiffMonitorView*.

The user interface developed uses some events to help facilitate communication between the different display widgets; however, there is significant coupling which makes it difficult to easily add new functionality. This coupling is eliminated as part of one of the refinements presented in a subsequent report [21].

The highest level components which control the monitor system are the *MonitorView* and its corresponding *MonitorPresenter*. *MonitorView* creates and lays out all of its children. *MonitorPresenter* is the heart of the client-side application logic. It creates all sub-presenters (presenters which are associated with the widgets that *MonitorView* is composed of.

In addition to the dialog views, there exist three non-trivial views and presenters which compose the front end of the monitor system. While the components are non-trivial, they will not all be detailed here. For an example of the interactions across the presenters, refer to Figure 6.

[h] Documentation available at: http://google-web -toolkit.googlecode.com/svn/javadoc/1.5/com/google/gwt/user/server/rpc/ RemoteServiceServlet.html

[i] A gallery of widgets available in gxt is available here: http://www.sencha.com/products/gxt/examples/

```
public void handleEvent(FileModelOpenEvent event) {
        fileModelOpen = event.model;
        FileMonitorDiff diffToOpen;
        // If no diff currently open, just open the committed one
        if (diffOpen == null) {
                diffToOpen = locateViolatedDiff(event.model);
        // Else if there is a diff open and it matches the model open and is a diff against the commit,
        open that
        } else if (diffOpen.getFilePath().equals(event.model.getFilePath()) &&
        diffOpen.type.equals(FileMonitorDiffType.COMMITTED)) {
                diffToOpen = event.model.getDiffAgainstCommitted();
        // Else if there is a diff open AND it matches the model open AND it is a user diff AND the model
        still has a diff for that user
        } else if (diffOpen.getFilePath().equals(event.model.getFilePath()) &&
        event.model.monitoringLiveChanges() &&
        event.model.getUsersWhoHaveMadeLiveChanges().contains(diffOpen.getUserId())) {
                diffToOpen = event.model.getLiveDiffAgainstUser(diffOpen.getUserId());
        // Else the diff that is open no longer exists or is a different file
        } else {
                diffToOpen = locateViolatedDiff(event.model);
        }

        // Open the model and diff in the toolbar
        diffToolbarPresenter.open(event.model, diffToOpen);
        eventBus.fireEvent(new FileMonitorOpenDiffEvent(diffToOpen));
        diffOpen = diffToOpen;
}
```

Figure 6 Sample of event-driven presentation code

The most substantial view and presenter were those of the *DiffMonitorView* and *DiffMonitorPresenter*. *DiffMonitorView* has it is core a GWT component (*DataGrid*)[i], which helps layout the data. The syntax highlighting was also implemented through regular expression pattern matching on the client side. The syntax highlighting only matches based on words, and does not take into account string literals or comments.

Detection of Violations

The server side algorithms which detect violations in the source code must know when changes have occurred. To do this, an Event Handler was placed in Cloudstudio's *NotificationArbiter* [10]. When a user commits their code or shares their changes, an *IdeCommitEvent* is fired, which this handler handles. In the handler, an RPC call is made to the servlet, asking it to register notifications for all monitors that are now violated.

This is slightly problematic, as the datastore representation of monitors does not store whether or not the monitor is violated. In addition, at the time of implementation the *IdeCommitEvent* did not contain information about which files were modified during the commit. This leads to having to examine every single monitor in the datastore for violation. When a violation is detected, there is no way of knowing whether or not a notification of violation has already been sent to the user. This actually led to expanding the functionality of the notification system, which is described in a separate report [10].

To detect a violation, the servlet computes all relevant *FileMonitorDiff*s for the monitor (depending on whether or not monitoring live changes), and if one or more are violated, then the monitor is considered to be violated. However, to compute the *FileMonitorDiff*s, the servlet first

[i] Javadoc available at:
http://google-web-toolkit.googlecode.com/svn/javadoc/latest/com/google/gwt/user/cellview/client/DataGrid.html

extracts several versioned files from the configuration management system, rebuilding the files line-by-line.

Issues

Several issues had to be addressed throughout the implementation of file monitors into Cloudstudio, ranging from technical obstacles to performance issues.

A major limitation of developing this project in Google Web-Toolkit is the restrictions that are imposed on all client code and shared code (code which will be presented in the browser once it has been compiled into JavaScript). Because of how the development mode in GWT works, some of these issues are not detected until the app is deployed. Originally the *FileMonitorDiff*s (described in Implementation) utilized the previously mentioned Java-Diff-Utils library directly; however this jar contains some code in it which cannot be compiled to JavaScript (namely the Regular Expression classes). This was detected late into development, but luckily creating a *FileMonitorDiffFactory* which exists in a server package and interfaces with the Java-Diff-Util jar mitigated this problem.

As a personal development issue, programming, debugging, and testing with Google Web-Toolkit proved to be very difficult. In the developer mode, the client-side JavaScript is generated on the fly, which leads to a very slow web application for slower machines. Often times this interpretation was so slow that the app assumed that its connection with the server had been lost. Testing new features was incredibly tedious, and sometimes performing the simplest of tasks could be nearly impossible.

Future Work

Cloudstudio, being a cloud-based IDE designed to support distributed software projects is an excellent candidate for implementing monitors. There are many additional projects which could be developed that will further demonstrate the usefulness of monitors in the development life-cycle. Some of this future work may not be implementable for all programming languages that Cloudstudio currently supports. Projects are listed in order of benefit for estimated effort.

Other Language Support

The principal of monitoring is language agnostic and very little of the implementation developed for Cloudstudio is bound to the language for which the monitors were developed (Eiffel). However, some features like syntactic highlighting in the view for comparing a shadow version with the current version were implemented only for Eiffel. As Cloudstudio continues to expand its language palette it would benefit to extend the monitoring system to support them. With simple text comparators, this should be fairly trivial.

Aspect Granularity

While the monitors developed for Cloudstudio in this report only monitor at the file level, much work can be done to improve the granularity of monitored aspects. Providing more useful comparators is discussed later, but the following aspect granularities would allow for more control by developers and ultimately a lower signal to noise ratio. In order to be effective, broad monitors need to be used sparingly, whereas very fine grained monitors (like invariant monitoring mentioned below) could be applied liberally with a lower chance of experiencing false positives that surface irrelevant information.

- **Class aspect monitoring**: The notion of class would be useful for monitoring many object oriented languages, especially languages which allow multiple class definitions per file.
- **Invariant monitoring**: As invariant changes in a stable project could have unexpected and far reaching effects, the ability to monitor only an invariant is likely to be useful when applied to clusters/packages or the entire project.
- **Full feature monitoring**: Developers may only wish to monitor a select group of features within a class. With fully developed feature monitoring, class monitoring could be implemented as a composite of feature and invariant monitoring, and file monitoring as a set of class monitors
- **Contract monitoring**: While monitoring features does provide more granularity over class or file, monitoring contracts of Eiffel (and to a lesser extent other languages) features could be sufficient in many cases, especially for developers who are clients to other developers' APIs. Monitoring of preconditions and post conditions for any changes would already be very useful, but this monitoring aspect becomes even more useful when combined with some of the more advanced comparators mentioned later. In addition to monitoring the contracts for individual features, the feature contracts for an entire class along with its invariant could be monitored for class contract monitoring.
- **Method signature Monitoring**: For languages which do not have formal preconditions and postconditions as a language feature, monitoring the method signature could still be a viable aspect to monitor. For a language like Java, monitoring visibility, synchronicity, return type, name, and parameters could provide an early heads up on what could be a breaking change.
- **Inheritance monitoring:** For subclasses, a change of the inheritance clause of the class could radically redefine the behavior of the class. Changes in inheritance clauses would be highly useful for languages like Eiffel which allow for multiple inheritance, visibility rescoping, and renaming. This could also be used in other object-oriented languages supported by Cloudstudio.

Contract Violation Monitoring

For languages like Eiffel which provide contracts and invariants, there exist some very exciting and interesting possibilities for monitor. Previously mentioned in this section is the ability to monitor on the aspect of class contract. With this aspect, an interesting comparator would be that of comparing the shadow with the present version to determine if the present version is a valid Liskov substitution of the shadow. A violation would be caused by any method either strengthening its precondition or weakening its postcondition or by the class weakening its invariant. An implementation could be as simple as detecting whether clauses have been added or removed, or perform more complex contract analysis, analyzing each Hoare clause to determine how it has changed. This would tie together contract analysis and monitoring together in a way that could produce a highly useful monitor which could be used liberally with little fear of generating false positives.

Usage study

How are monitors used? When are they most beneficial? These are questions that have not yet been answered due to the fact that no usage study has been completed. Providing several

different types of projects to users and observing the effects that monitoring has on the development process is key to understanding the way in which monitors should be further developed. This usage study could possibly be carried out through coursework projects at universities, especially in the Distributed and Outsourced Software Engineering Course at ETH Zürich[k].

Facilitating violation resolution

If a developer has a monitor which has detected a violation and the developer has determined that the violation is noteworthy, it would be useful to have an easy course of action to take. Integrating with existing Cloudstudio tools and possibly other tools could provide a pathway that the developer can begin towards resolving the violation.

- **Send a message:** At the very least, it would be beneficial to be able to send a brief message to the developer, either via email of through the notification system.
- **Initiate a code review:** Should the developer feel that the implementation that has violated their monitor is something that is worthy of discussion, they could begin a code review of the violated class. In the event that the developer is monitoring changes that have not been committed yet, this code review could be deferred until the point in which the user decides to share their changes, or could even be a prerequisite before the code can be shared or pushed.
- **File a Bug:** Currently Cloudstudio does not have a bug tracker, but were one to be developed, a pathway for monitor violation resolution could be to file a bug with the author of the violating code. This would be more applicable for situations where the code correctness is not necessarily a question (such as a breaking change to an existing interface).

Making monitors public and sharable with other developers

Making a developer's monitors public would benefit both those whose code is being monitored and the developers who are monitoring others' code. As monitors intrinsically allow other developers to keep tabs on each other, there seems to be no disadvantage in developers being able to see the monitors of their co-developers. Making monitors public and sharable could:

- **Enable notification of teams:** Allow for notifying a team to the target of a monitor violation notification. If a team is responsible for maintaining a cluster / package, it is commonplace that they participate in code-reviews for any changes. In the event that a change is being made, it may be only necessary for one member of a team to take a look at the violation and determine if it is truly an issue. In the event that it is not, this developer could dismiss the notification for the entire team.
- **Share wealth:** Once one developer has setup monitors for an aspect, it would be beneficial for other developers to be able to easily add the same monitors to their set of monitored aspects.
- **Provide context for violating developers:** In the event that a co-developer has violated a monitor that is in place, it would be beneficial for them to be able to be able to see the

[k] For information about this course, visit: http://se.inf.ethz.ch/research/dose/

same violation details as the developer who originally monitored the file. This would lead to better violation resolution, as the violated monitor could provide a rough acceptance test for what changes are acceptable to the monitoring development.

- **Facilitate stakeholder involvement:** Discussed earlier were some detriments that being able to monitor live changes cause. Developers may be discouraged from venturing out of tier teams code for fear that others will be immediately warned. But by knowing who is monitoring a piece of code, these ambitious developers could begin a conversation with the monitoring developers. Most developers will only monitor aspects that they feel strongly about, and being able to begin a discussion with all concerned members before beginning development would not only avoid being pounced on like was mentioned earlier, but could also allow those monitoring developers to participate in the evolution of that aspect.

Default monitoring level

For large projects involving multiple teams, the previously discussed roles of authors and stewards are very useful for sharing expertise across a project. Allowing developers to set default monitors for the classes that they create or modify would save the developers from having to remember to manually add a monitor for every class that they create. Such monitors could even be inferred from author metadata in the file. Phabricator supports notifying a user for all commits that match a regular expression. Such a feature in the monitoring system could automatically monitor all files which match a regular expression.

Table of Figures

References

[1] H.-C. Estler, M. Nordio, C. A. Furia, B. Meyer and J. Schneider, "Agile vs. Structured Distributed Software Development: A Case Study," in *7th International Conference on Global Software Engineering*, IEEE, 2012.

[2] E. Carmel and R. Agarwal, "Tactical Approaches for Alleviating Distance in Global Software Development," *IEEE Softw.,* vol. 18, no. 2, pp. 22-29, March 2001.

[3] E. Carmel, Global software teams: collaborating across borders and time zones, Upper Saddle River, NJ, USA: Prentice Hall PTR, 1999.

[4] M. Nordio, R. Mitin, B. Meyer, C. Ghezzi, E. Di Nitto and G. Tamburrelli, "The Role of Contracts in Distributed Development," in *Proceedings of Software Engineering Approaches for Offshore and Outsourced Development*, 2009.

[5] J. A. Espinosa, K. Nan and E. Carmel, "Do Gradations of Time Zone Separation Make a Difference in Performance? A First Laboratory Study," in *Proceedings of the IEEE International Conference on Global Software Engineering (ICGSE 2007)*, IEEE, 2007, pp. 12-22.

[6] M. Nordio, H.-C. Estler, B. Meyer, Ghezzi, C. Ghezzi and E. Di Nitto, "How do Distribution and Time Zones affect Software Development? A Case Study on Communication," in *Proceedings of the IEEE International Conference on Global Software Engineering (ICGSE 2011)*, IEEE, 2011.

[7] M. Nordio, R. Mitin and B. Meyer, "Advanced Hands-on Training for Distributed and Outsourced Software Engineering," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE)*, 2010.

[8] M. Nordio, C. Ghezzi, B. Meyer, E. Di Nitto, G. Tamburrelli, J. Tschannen, N. Aguirre and V. Kulkarni, "Teaching Software Engineering using Globally Distributed Projects: the DOSE course," in *Collaborative Teaching of Globally Distributed Software Development - Community Building Workshop (CTGDSD)*, ACM, 2011.

[9] M. Nordio, H.-C. Estler, C. A. Furia and B. Meyer, "Collaborative Software Development on the Web," 2011.

[10] C. Dentel, "News and Notification: Propagating Relevant Changes to Developers," Software Engineering Laboratory: Open Source Eiffel Studio, ETH Zürich, 2012.

[11] M. Nordio, C. Calcagno, B. Meyer, P. Müller and J. Tschannen, "Reasoning About Function Objects," in *TOOLS-Europe*, J. Vitek, Ed., Springer-Verlag, 2010.

[12] J. Tschannen, C. A. Furia, M. Nordio and B. Meyer, "Verifying Eiffel Programs With Boogie," in *First International Workshop on Intermediate Verification Languages (BOOGIE 2011)*,

2011.

[13] Y. Wei, H. Roth, C. A. Furia, Y. Pei, A. Horton, M. Steindorfer, M. Nordio and B. Meyer, "Stateful Testing: Finding More Errors in Code and Contracts," in *26th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2011.

[14] B. Meyer, A. Fiva, I. Ciupa, A. Leitner, Y. Wei and E. Stapf, "Programs That Test Themselves," *IEEE Computer,* vol. 42, no. 9, pp. 56-55, 2009.

[15] Y. Pei, Y. Wei, C. A. Furia, M. Nordio and B. Meyer, "Code-Based Automated Program Fixing," in *26th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2011.

[16] J. Tschannen, C. A. Furia, M. Nordio and B. Meyer, "Usable Verification of Object-Oriented Programs by Combining Static and Dynamic Techniques," in *Proceedings of the 9th International Conference on Software Engineering and Formal Methods*, 2011.

[17] A. Dima, "Developing JavaScript Applications in Eiffel," Masters Thesis, ETH Zürich, 2011.

[18] J. A. Espinosa, R. E. Kraut, S. A. Slaughter, J. F. Lerch, J. D. Herbsleb and A. Mockus, "Shared Mental Models, Familiarity and Coordination: A Mulit- Method Study of Distributed Software Teams," in *International Conference for Information Systems*, 2001.

[19] Y. Brun, R. Holmes, M. Ernst and D. Notkin, "Speculative Identification of Merge Conflicts and Non-Conflicts," University of Washington, Seattle, 2010.

[20] S. Weber, "Automatic Version Control System for Distributed Software Development," Masters Thesis, ETH Zürich, 2012.

[21] C. Dentel, "Refinements and Git Integration with Notifications and Monitoring," Software Engineering Laboratory: Open Source Eiffel Studio, ETH Zürich, 2012.