8 Continuous Co-ordination Tools and their Evaluation

A. Sarma, B. Al-Ani, E. Trainer, R. S. Silva Filho, I. da Silva, D. Redmiles, A. van der Hoek

Abstract: This chapter discusses a set of co-ordination tools (the Continuous Co-ordination (CC) tool suite that includes Ariadne, Workspace Activity Viewer (WAV), Lighthouse, Palantír, and YANCEES) and details of our evaluation framework for these tools. Specifically, we discuss how we assessed the usefulness and the usability of these tools within the context of a predefined evaluation framework called DESMET. For example, for visualization tools we evaluated the suitability of the level of abstraction and the mode of displaying information of each tool. Whereas for an infrastructure tool we evaluate the effort required to implement co-ordination tools based on the given tool. We conclude with pointers on factors to consider when evaluating co-ordination tools in general.

Keywords: Distributed and Collaborative Software Engineering, Collaborative Tools, Evaluation.

8.1 Introduction

Co-ordination has been studied in different domains and within different contexts, as any kind of group work entails co-ordination [1, 32]. For our purposes, we focus on co-ordination efforts that are required to understand interdependencies among artifacts and developers in a software project, and to take appropriate steps to produce results with minimal conflicts. We recognize that co-ordination is not a static process, but one that needs continuous adjustments. This means that concerned individuals have to have the ability to respond to ongoing changes in the project and the effects of these changes on their work. Furthermore, co-ordination efforts occur at multiple levels: among developers, between managers and their teams, among multiple teams working together, and so on. The information required by an individual strongly correlates with their role in the team and their perspective of the project. Therefore, tool support for co-ordination needs to ensure that the right information is presented to the right individual at the right time using appropriate presentation techniques. To achieve this goal, we created a suite of co-ordination tools that meets the different needs of different kinds of software development activities.

Evaluation of co-ordination tools is both critical and challenging [29]. In this chapter, we discuss the strategies we used to evaluate our co-ordination tool suite as well as results from the evaluation. In particular, we discuss our goals when evaluating the tools with respect to their usefulness and usability. Generally, the usefulness and functionality of our tool set has been largely motivated by our own ethnographical studies of multiple software development teams [11, 15].

This chapter discusses our approach to evaluate the usability, as well as, in some cases, the usefulness of each tool based on DESMET, Kitchenham et al.'s framework for evaluating software engineering tools [29]. The evaluation of each tool followed a subset of the nine evaluation types listed by DESMET, which was based on the nature/features of the tools as well as their maturity level. We found that evaluation should be iterative in nature as has been recommended for prototyping in software development [27].

The rest of the chapter is organized as follows. The next section provides a brief introduction to our approach. Section 1.2 provides a review of related work in which we discuss interdependencies and the need for co-ordination in addition to evaluation methodologies. This background section is followed by a description of the DESMET framework and our extension of this framework. We then present an outline of the Continuous Coordination (CC) principles, the origin of the CC tools, the evaluation approaches adopted for each, and the lessons learned as a result. The chapter concludes with a discussion of threats to the validity of our work and conclusions regarding the evaluation of the usefulness and usability of the CC tools.

8.2 Research Context

In software development the need for co-ordination among developers generally arises because of the underlying technical dependencies among work artifacts; as well as the structure of the development process [13, 7, 8]. Researchers in the software engineering as well as Computer-

Supported Co-operative Work (CSCW) communities have recognized this problem and created a host of tools to improve team co-ordination. However, evaluating the usability and usefulness of such tools has proven to be extremely difficult. Here we focus on different evaluation approaches that are applicable for co-ordination tools.

There exists a diverse range of approaches to evaluating collaborative tools, e.g., [41, 4, 35, 56, 18, 31, 50]. Adopting a combination of empirical evaluation approaches is perceived as means to meet the challenges typically encountered [49]. The diversity of existing tools and evaluation approaches reflect the many challenges of facilitating co-ordination in teams [24].

Further, several evaluation frameworks have been proposed to support software tool evaluation, e.g., [29, 10, 30], among others. We base our evaluations of the CC tool suite based on the DESMET framework [29]. We chose DESMET because it provides the desired level of abstraction that readily lends itself to adoption and matches our research objectives. This framework has also been successfully adopted by other researchers to evaluate software tools, e.g., [34, 36, 25].

8.3 The CC Evaluation Framework

The DESMET evaluation methodology separates evaluation approaches into two broad classes: (1) quantitative evaluations aimed at establishing measurable effects of using a tool, and (2) qualitative evaluations aimed at establishing method tool appropriateness, i.e., how well a tool fits the needs and cultures of an organization. These two methods are further subdivided into experiments, case studies, surveys, feature analyses, and screening to form nine distinct evaluation approaches. We used six of the evaluation approaches listed by Kitchenham et al. Note, we did not use all the approaches for each tool; rather a different combination of approaches was used based on the particular features, level of maturity, and the goal of the tool, i.e., usefulness or usability factors. Some of the factors that we considered when evaluating usefulness or usability were the effort that users' expended to utilize and/or understand a CC tool together with the perceived benefits. Moreover, we considered issues relating to the appropriateness of information that a tool shares with the development team (e.g., level of abstraction and mode of display).

We used the DESMET framework to determine which evaluation methodology to use per tool. Here, we present an overview of the evaluation approaches that we adopted, within the context of the framework as defined by Kitchenham et al:

1. **Qualitative screening** is defined as a feature-based evaluation done by a single individual (or cohesive group) that not only determines the features to be assessed and their rating scale, but also performs the assessment. In the initial screening, the evaluations are usually based on literature describing the software method/tools rather than actual use of the methods/tools. We conducted such a screening by surveying existing tools and their features as reported in literature. Consequently, we surveyed related work for each one of our tools.
2. **Hybrid method 1: Qualitative effects analysis** is defined as a subjective assessment of the quantitative effect of methods and tools, based on expert opinion. We have used this analysis approach repeatedly at different phases of tool development. All our tools followed iterative prototyping and at the end of each prototyping cycle, we demonstrated our tools to industry experts as well as researchers to get their feedback on both usability and usefulness.
3. **Qualitative experiment** is defined as a feature-based evaluation done by a group of potential users who are expected to try out the tools on typical tasks before evaluating them. The tasks are performed by staffs that have used the tool on a real project. We requested that participants also “think out loud” during the experiment to get an idea of which features are difficult to understand in addition to gaining insights into the reasoning behind their actions [19]. Another subcategory in this approach is the “feature analysis” experiment which is typically adopted when a tool’s impact is not directly measurable on one project and is thus evaluated across multiple projects. We conducted such experiments with mature tools.
4. **Quantitative case study** is defined as an investigation of the quantitative impact of tools organized as a case study. This mode of evaluation can be used to understand the usefulness of a tool when applied to a real project as well as the scalability of the tool. We utilized data made available in open-source software projects repositories as case study data. This data was collected from a real and ongoing large scale project.
5. **Hybrid method 2**: **Benchmarking**: is defined as a process of running a number of standard tests usually comparing one tool to alternative tools and assessing the relative performance of the tools against those tests. We selected a set of open source infrastructures to be compared with our tool as a benchmark in this instance of evaluation.

6. **Quantitative experiment** is defined as an investigation of the quantitative impact of tools organized as a formal experiment. We used a large enough sample size in our experiments to overcome the anticipated effects of individual and team differences. We typically adopted this methodology to evaluate mature tools because of the extensive effort and time required.

A detailed description of each tool is presented in the following section together with details of the evaluation approaches adopted. (Appendix A) presents a summary of the tools and the evaluation approaches we utilized within the context of DESMET.

8.4 Continuous Co-ordination (CC) Tools: Their Origin and Evaluation

Co-ordination occurs at different levels and involves different stakeholders (e.g., developers, managers, testers, clients), who may have differing co-ordination requirements. Our suite of co-ordination tools attempts to meet different requirements among different stakeholders.

The CC tool suite was designed while keeping four critical questions in mind [42]. The first involves identifying *when* the tool should provide information. Providing a constant stream of information can overwhelm users whereas infrequent sharing of information may lead to some users lacking information critical to completing their tasks. The information provided to the user depends on their role within the team. "*What* kind of information does the user need?" is the second question that guides our work. For example, a manager would typically need to be aware of team structure and work products to co-ordinate a project. A programmer, however, would generally need to be aware of changes to the design. These considerations lead us to ask, "*Who* should information be provided to?" For example, should information be provided to all programmers, to managers, or to a sub-set of these? Finally, *how* information is presented should also be considered. In general, our tools visualize graphical representations of co-ordination information because it can be more efficient and easier to understand information presented graphically than textually [52, 5].

In the following sections, we discuss a subset of the CC tool suite we subjected to more than one type of evaluation approach and the lessons we learned.

8.4.1 Ariadne

Ariadne is a visual tool that infers dependencies between people based on the modules they author. Our field studies led us to conclude that the management of dependencies becomes a daunting task as a project evolves and grows in the number of artifacts and contributors [14, 16]. These studies gave us insight into several types of communication and co-ordination problems, which helped us develop several representative scenarios that revealed the different types of dependency relationships managers and developers need to understand [14]. We call these relationships “socio-technical” because they involve both artifacts and the people who work on them.

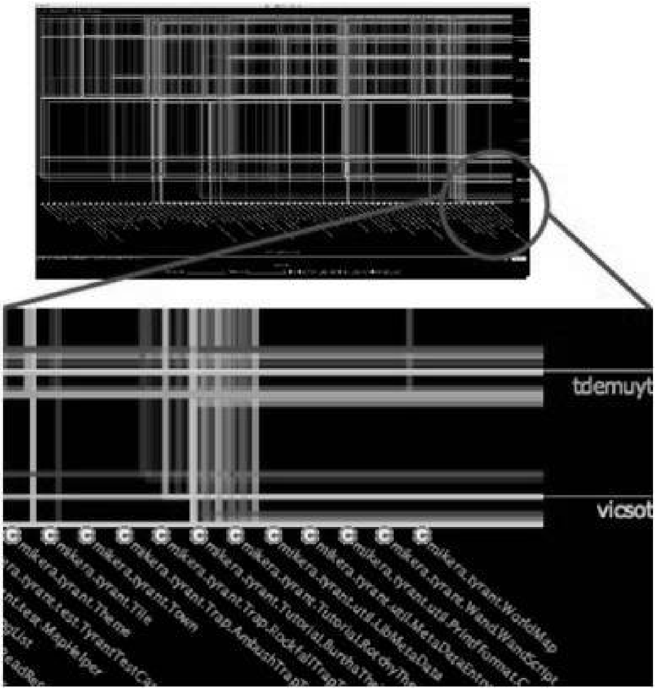

tdemuyt
vicsot
mikera.tyrant.Tile
mikera.tyrant.Town
mikera.tyrant.Trap.AmbushTrap
mikera.tyrant.Trap.RockfallTrap
mikera.tyrant.WandWandScript
mikera.tyrant.WorldMap

Fig. 8.1. An overview and zoomed in view of a project’s socio-technical dependencies using Ariadne

Ariadne visualizations allow developers and managers to identify relevant socio-technical relationships central to their co-ordination needs. First, Ariadne creates a call-graph representing dependencies between source-code modules. Second, the tool annotates this graph with authorship information by connecting to a project’s configuration management repository. Finally, Ariadne calculates a sociogram [54] representing dependencies between developers through the modules with which they work. The visualization is designed to take advantage of available screen real estate and thus occupies the entire screen.

Ariadne visualizations were designed to make the most of available screen real estate (shown in Fig. **8.1**). Ariadne lays out called code units on the horizontal axis and developers on the vertical axis. It draws connections from a dependent author to the code unit they are dependent upon and back to the author responsible for that code unit and repeats this for each code unit in the project. Further details on its visualization and its advantages have been reported elsewhere [51].

Objective of evaluation process and steps taken

Ariadne visualizes socio-technical relationships using highly abstract representations of dependency information, such as shapes, colors, and axes. As such, effort is required to learn how to use the tool to accomplish specific tasks. We thus decided upon an evaluation strategy that would allow us to evaluate this effort in early stages of the tool's design.

A survey, or qualitative screening, of literature and existing socio-technical tools revealed the general need to support awareness of dependencies and identifying developers of interest via visual interpretation. Literature in the information visualization field identified usability as one important barrier to tool adoption by end-users [2, 3, 40]. Moreover, evaluating tools in real settings and with real users (in our case, developers and managers) is expensive in terms of the effort required, especially in the early stages of design. In an effort to get usability feedback "cheaply," we applied multiple inspection usability inspection methods: Nielsen's Heuristic Evaluation [37] Lewis and Polson's Cognitive Walkthrough [55] and Thomas Green's Cognitive Dimensions of Notations [22]. In addition, we applied Edward Tufte's general principles of information presentation [52, 53]. We performed each inspection method with a team comprised of four colleagues. They had no experience using the new visualization. This unfamiliarity helped us to identify problematic design assumptions about new users' expectations and assumptions about interacting with and drawing conclusions from the visualization. Further information of our evaluations is detailed elsewhere [51].

After this qualitative inspection, we performed a case study where we selected several open-source projects from Sourceforge.net to visualize. These projects had been active for several years, and were active at the time of our evaluation. Thus they represented a test-bed from which to confirm the scalability of the visualization to real-world projects. In parallel to the previous, with the help of industry partners and open-source developers, we assessed the usefulness of current features and incorporated suggested feedback into the tool. These activities, in combination with the application of usability inspection methods, constituted a qualitative effects analysis in terms of DESMET.

Lessons learned with respect to the tool

We were able to tease out commonly occurring problems with respect to usability through the combined application of evaluation approaches. For example, the use of color to indicate individual developers and the directionality of dependencies proved to be more difficult than we originally thought, especially as we visualized larger projects. The Cognitive Walkthrough, Tufte's principles, and the Cognitive Dimensions analyses highlighted this issue. The Heuristic Evaluation and Cognitive Dimensions revealed the potential need to allow users to undo certain filtering actions in order to trace back their steps, as well as the option to view different configurations of developers, e.g., aggregating them into teams. All three methods suggested the need to improve feedback (e.g., to indicate that specific dependencies have not been created instead of displaying no search results).

Ariadne allows users to identify patterns in the way developers call different code in the system that from a general overview of a project's socio-technical dependencies. Throughout the course of applying the usability inspection methods discussed above, we realized that these patterns would heavily depend on the way the different axes were ordered. For example, a pattern generated from a temporal ordering of the code units (arranged by date last modified) might not show up if the code units were arranged in alphabetical order instead. Thus, the ordering makes a difference in the patterns that users will see, identify, and flag for future identification.

Lessons learned with respect to evaluation

The usability inspection methods we applied to Ariadne thus far have allowed us to make certain corrections to Ariadne's visualization before deploying the tool to real users in real settings. However, evaluations of this sort cannot account for organizational issues relating to adoption. This is one limitation of our evaluation strategy. Publicly exposing sensitive information normally stored in software repositories may have effects on the way developers work or even Ariadne's results. In one instance, we showed some of our early visualizations to several open source developers who commented that they would avoid "touching" certain classes to avoid breaking dependent code. To an extent, Ariadne can be used by managers and supervisors to gauge developer's progress, or lack thereof. Further, as speculated by our interviewees, individuals may “game” the tool to show an increase in their contributions, especially if they feel that a lack of activity may be used against them.

Some researchers claim that new evaluation approaches for visualizations are needed because current approaches test the wrong users and unconventional user interface components hurt user performance [2, 3]. We have described the impracticality of deploying Ariadne to our intended end-users in early design. To address the second point, the results from our evaluation indicate that usability inspection methods can be usefully applied to abstract visualizations instead of traditional interface components such as methods and drop-down menus. Moreover, despite the fact that Tufte’s principles of information are general rather than domain-specific; our work serves as one of the few examples of the application of these rules-of-thumb to novel, interactive socio-technical visualizations for software engineering. Thus, traditional evaluation approaches are still useful for incremental prototyping and iterative design of our research tools. As we continue to develop and refine Ariadne, visualization-specific evaluation heuristics like those suggested by other researchers [57] will become more useful. We expect the aforementioned evaluation to be used as a point-of-comparison for researchers evaluating socio-technical visual interfaces in early design.

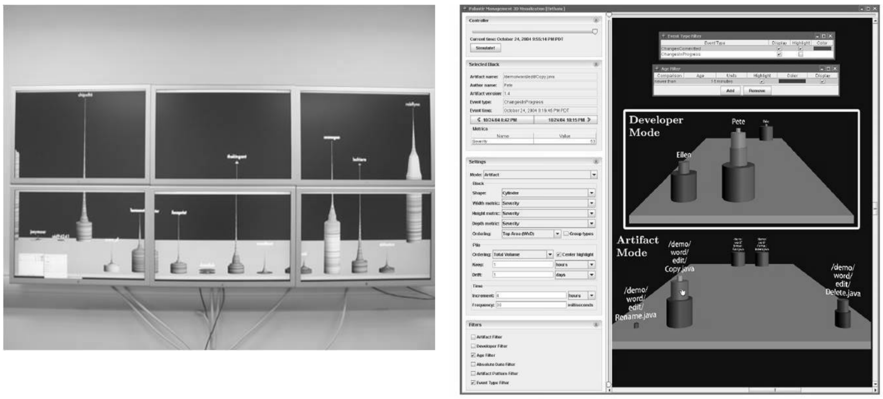
8.4.2 Workspace Activity Viewer

Workspace Activity Viewer (WAV) provides a highly scalable view of all ongoing parallel development activities in a software project [43]. WAV visualizes information in 3D to illustrate changes to a software project over time, the types and sizes of the changes, and provides various filters to examine aspects of workspace activities in more detail. WAV reveals social evolution via a movie-like playback of the state of the project, showing what developers are active when, and to which types of artifacts they contribute (Fig. **8.2.**). As such, WAV can benefit both developers and managers, and provides two different views: artifact-centric and developer-centric, accordingly. Both views use a cylinder metaphor to represent workspace changes, where the width of the cylinder represents the size of a change. In the artifact-centric view, cylinders represent artifacts, with each segment of a cylinder denoting a developer that has made changes to that artifact. In the developer-centric view, cylinders represent developers, with each segment of a cylinder denoting an artifact that developer has touched. As stacks (artifacts or developers) become dormant, the associated stack of cylinders slowly moves to the back of the display. A more detailed account of the tool is reported elsewhere [43].

Objective of evaluation process and steps taken

The objective of our evaluation of WAV was to confirm the accuracy of the tool's playback of the activities occurring in real software development projects and to test the visualization's capacity to scale to large software projects [43]. In terms of display technique, we wanted to see if all relevant workspace events could be clearly visualized using the screen real-estate WAV requires. As we have seen in the case of Ariadne, deploying tools in real settings is a difficult challenge, especially in early prototyping. Thus, we decided to evaluate WAV through a case study and report results to project managers and developers.

We applied WAV to five open-source projects: ArgoUML, GAIM, Freemind, jEdit, and Scarab. In addition, we analyzed project data from a local company that collaborates with our research group. Since we used archived data for our case study, we did not gather information of real-time workspace edits. To overcome this problem, we simulated workspace data based on CVS change metadata (e.g., who checked the file in, when they did it, and how much changed). This metadata allowed us to establish known states for each artifact and subsequently generate events correlating to workspace activity before the commit occurred. The evaluation we performed constitutes a qualitative-effects analysis and a quantitative case study.



Controller

Current time: October 24, 2004 9:55:14 PM PDT

Selected Block

Artifact name: /demo/word/edit/Copy.java

Author name: Pete

Artifact version: 1.4

Event type: ChangesInProgress

Event time: October 24, 2004 9:18:46 PM PDT

< 10/24/04 8:42 PM | 10/24/04 10:15 PM >

Metrics

Name	Value
Severity	53

Settings

Mode: Artifact

Block

Shape: Cylinder

Width metric: Severity

Height metric: Severity

Depth metric: Severity

Ordering: Top Area (WxD) □ Group types

Pile

Ordering: Total Volume ☑ Center highlight

Keep: 1 hours

Drift: 1 days

Time

Increment: 4 hours

Frequency: 30 milliseconds

Filters

□ Artifact Filter

□ Developer Filter

☑ Age Filter

□ Absolute Date Filter

□ Artifact Pattern Filter

☑ Event Type Filter

Event Type Filter

EventType	Display	Highlight	Color
ChangesCommitted	☑	☑	■
ChangesInProgress	☑	□	

Age Filter

Comparison	Age	Units	Highlight	Color	Display
newer than	15	minutes	☑	■	☑

Palantír Management 3D Visualization [Default]
Simulate!
Add
Remove
Developer Mode
Pete
Ellen
Artifact Mode
/demo/word/edit/Copy.java
/demo/word/edit/Rename.java
/demo/word/edit/Delete.java

Fig. 8.2. Developer-centric mode on six monitors (left) and artifact-centric mode with user-definable filters on the right

Lessons learned with respect to the tool

Visualizing the collective activity in a project can allow managers to choose and identify patterns that may lead to co-ordination breakdowns. For example, the movie-like playback feature of WAV allows one to see periods of stagnation which may indicate insufficient progress. Whereas spurts of activity as artifacts and developers' piles expand upward and move to the front may indicate conflicts. These patterns can then be used as potential “red flags” to indicate the possibility of problems over the lifecycle of a project.

An important concern is the visualization’s ability to scale to large software projects caused by the amount of workspace events captured [43]. Over this range, the filters available on WAV’s interface and the ability to rotate the visualization’s axes provided sufficient support to manage the problem of scalability, as reported by the managers to whom we showed the data. The evaluation method we chose for WAV allowed us to validate the accuracy of the events captured by the tool by correlating them with actual events over the course of development. It was further validated by a project manager who confirmed our observations.

Lessons learned with respect to evaluation methods

Our evaluations are not a substitute for assessment in real settings. However, they come close by looking at real project data from real development teams. Unlike costly evaluation approaches such as talk-aloud methods or human subjects tests, case-study data can be collected relatively cheaply from existing, (often) publicly available project repositories. While we were not able to gain access to real workspace activities, we were able to simulate them based on randomizations of the patterns between known check-ins and check-outs. As such, we could still make observations about the evolution of the projects. The most expensive part of the process is reflecting findings back to the original participants.

One aspect that evaluations of this type leave out is usability for the end-user, which is typically one of the main barriers to visualization adoption [2, 3, 40]. Usability is especially important in the context of the work discussed here because of the upfront costs associated with human subjects testing. Future WAV evaluations involves the application of usability inspection methods such as those applied to Ariadne [51]. These evaluations can reveal patterns of interest and compare activity between both developers and artifacts.

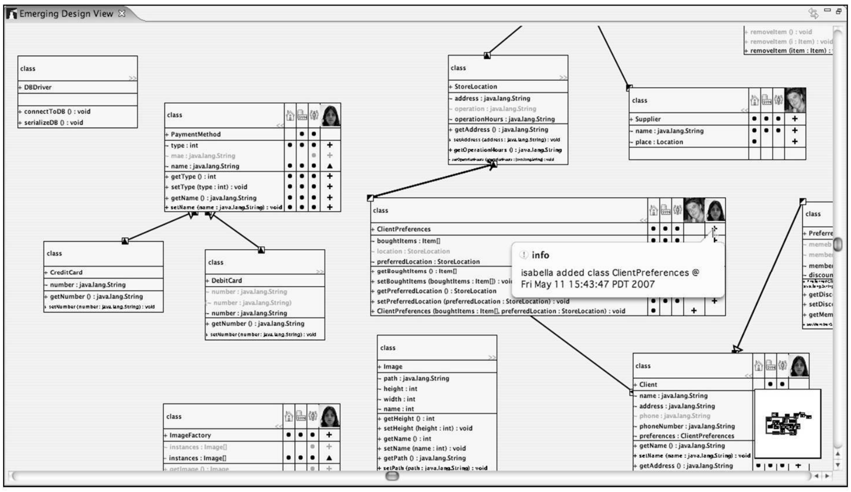
8.4.3 Lighthouse

Lighthouse is an awareness tool that supports team co-ordination by providing each developer with information of ongoing activities in the project [9]. The goal of the tool is to improve a developer's understanding about others' activities and how one's own activities affect the others. The tool builds an Emerging Design diagram, an always up-to-date abstraction of the source code components, dependencies, authorship and current changes. The diagram consists of a UML-like class representation of the code as it exists on the developers' workspaces (Fig. **8.1.**). All information about changes made to the code is collected automatically by Lighthouse from the IDE and the CM system and is propagated immediately to all project members.

Lighthouse visualization supports early detection of design decay by allowing users to identify unintended design changes. Problems like conflicting changes in shared artifacts and duplicate work can also be spotted as soon as they surface. A detailed account of the tool's features and the nature of the support it provides is reported elsewhere [9].

Objective of evaluation process and steps taken

Lighthouse has been evaluated both via qualitative effects analysis and qualitative experiments. We demonstrated Lighthouse to various industry experts and academic researchers, obtained and incorporated their feedbacks. Later we evaluated Lighthouse via a qualitative observational study to investigate its usefulness in warning participants of emerging conflicts, as well as the effort required by an individual to investigate and resolve conflicts.


Emerging Design View
class
+ DBDriver
+ connectToDB () : void
+ serializeDB () : void
class
+ PaymentMethod
- type : int
- mae : java.lang.String
- name : java.lang.String
+ getType () : int
+ setType (type : int) : void
+ getName () : java.lang.String
+ setName (name : java.lang.String) : void
class
+ CreditCard
- number : java.lang.String
+ getNumber () : java.lang.String
+ setNumber (number: java.lang.String) : void
class
+ DebitCard
- number : java.lang.String
- number : java.lang.String)
- number : java.lang.String
+ getNumber () : java.lang.String
+ setNumber (number: java.lang.String) : void
class
+ ImageFactory
- instances : Image[]
- instances : Image[]
+ getImage () : Image
class
+ StoreLocation
- address : java.lang.String
- operation : java.lang.String
- operationHours : java.lang.String
+ getAddress () : java.lang.String
+ setAddress (address : java.lang.String) : void
+ getOperationHours () : java.lang.String
class
+ Supplier
- name : java.lang.String
- place : Location
+ removeItem () : void
+ removeItem (i : Item) : void
+ removeItem (item : Item) :
class
+ ClientPreferences
- boughtItems : Item[]
- location : StoreLocation
- preferredLocation : StoreLocation
+ getBoughtItems () : Item[]
+ setBoughtItems (boughtItems : Item[]) : voi
+ getPreferredLocation () : StoreLocation
+ setPreferredLocation (preferredLocation : StoreLocation) : void
+ ClientPreferences (boughtItems : Item[], preferredLocation : StoreLocation) : void
class
+ Image
- path : java.lang.String
- height : int
- width : int
- name : int
+ getHeight () : int
+ setHeight (height : int) : void
+ getName () : int
+ setName (name : int) : void
+ getPath () : java.lang.String
+ setPath (path : java.lang.String) : void
class
+ Client
- name : java.lang.String
- address : java.lang.String
- phone : java.lang.String
- phoneNumber : java.lang.String
- preferences : ClientPreferences
+ getName () : java.lang.String
+ setName (name : java.lang.String) : void
+ getAddress () : java.lang.String
class
+ Preferre
- memeb
- member
- member
- discoun
+ getDisc
+ setDisc
+ getMem

info

isabella added class ClientPreferences @
Fri May 11 15:43:47 PDT 2007

Fig. 8.1. Lighthouse emerging design

This study recruited four graduate student volunteers who had sufficient knowledge about the Java programming language, the Eclipse IDE, and the software configuration management (SCM) tool (preferably Subversion). These volunteers used the prototype to execute small programming tasks on a simulated software development team. More specifically, participants were told that they would be joining a pre-existing team, substituting a developer who recently left the project. They were also informed that the rest of the team was distributed and available for communication solely by Instant Messaging (IM). Each participant was asked about their background, given a brief tutorial on Lighthouse, assigned a set of five programming tasks involving online store software, and asked to fill out an exit questionnaire. In reality, each participant was working by themselves; the other two team members being virtual entities (confederates) that were controlled by the experimenters [44]. The confederate's programming tasks were simulated with automated scripts that introduced changes in the software source code at pre-defined time intervals. Some of these tasks introduced conflicts in the source code that were supposed to be detected and dealt with by the participants. The experimenter also controlled the communication via IM between participant and confederates. The use of confederates allowed for control over the number of conflicts and coordination opportunities introduced in the experiment which facilitated the comparison of results across experiments.

Lessons learned with respect to the tool

For the experiment, we introduced two direct conflicts (concurrent changes to the same artifact) and two indirect conflicts (conflicting changes to dependent artifacts). We observed that the timing of conflict introduction was a decisive factor on detecting direct conflicts; developers who had already started coding a task before the confederate created the duplicated effort did not detect the conflict. We also observed that changes made by confederates were either noticed as soon as they surfaced or not until the end of the task, when participants faced merge problems because of the CM system. All changes detected on time, though, were quickly and appropriately addressed. When indirect conflicts were introduced during the experiment, only half of the participants recognized the conflict in one task and none could complete the other task in the given time.

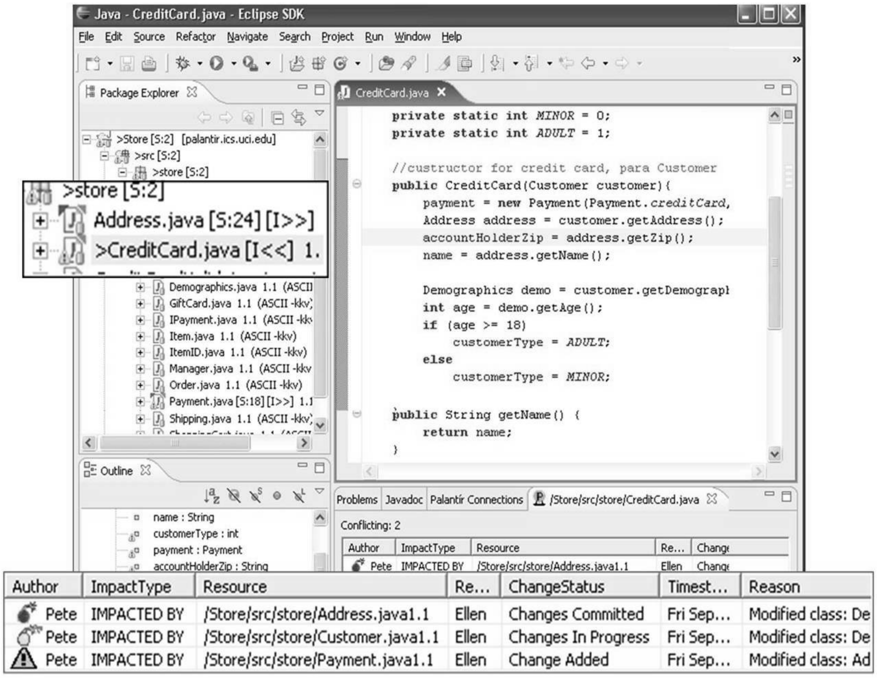
We designed the experiment to understand the role of “emerging design” is helping participant’s co-ordinate their work. At the end of the study participants reported that they found the emerging design served as a reference for understanding the software structure, which were corroborated by our observations on how participants explored the diagram during the study. We also found that participants by using filters that highlighted recent changes to the emerging diagram were able to use the diagram as a way of identifying ongoing changes in the project. Finally, in many cases the emerging design stimulated communication in a team. For example, when trying to contact a confederate to resolve a conflict, participants always first looked for the author of conflicting changes using the emerging design diagram. In all cases, participants contacted the most adequate confederate to address the issue. Further, changes that were unrelated to the tasks being performed were correctly ignored, thereby showing that Lighthouse streamlines communications in a project. However, we observed that participants were sometimes confused regarding which changes were local and which remote. Consequently, this usability problem might hinder users from responding to remote emerging conflicts. Our future work will provide means to differentiate between local and remote changes, which will help overcome this problem.

Lessons learned with respect to the evaluation of tool

Our study suffered from threats to validity common for user experiments. The total time of one hour was insufficient for subjects to complete all the tasks and a simple walkthrough of Lighthouse’s features was insufficient for them to correctly appropriately learn all the tool features. We found that the complexity of Lighthouse’s different interactive features meant participants required more time to learn how to use them. Further, to understand how the software code was evolving and its effect on the given tasks required a much longer experiment involving a more complicated code base. Such an experiment would allow independent changes made in different parts of the code to interact and create more intricate conflicts. Finally, the pressure of having to complete all the tasks within a limited period of time might have made participants spend less time observing and understanding the emerging design. We plan to follow this study with a more detailed in situ study of real developers working on their projects.

8.4.4 Palantír

Palantír is a workspace awareness tool that automatically and unobtrusively intercepts local edits as well as all CM operations in a workspace and transmits these events across relevant workspaces to inform developers of ongoing changes in the project [46]. Each workspace summarizes the events it receives and communicates these to a developer via subtle awareness cues.



Java - CreditCard.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help
Package Explorer
>Store [S:2] [palantir.ics.uci.edu]
>src [S:2]
>store [S:2]
Address.java [S:24] [I>>]
>CreditCard.java [I<<] 1.
Demographics.java 1.1 (ASCII
GiftCard.java 1.1 (ASCII -kkv)
IPayment.java 1.1 (ASCII -kk
Item.java 1.1 (ASCII -kkv)
ItemID.java 1.1 (ASCII -kkv)
Manager.java 1.1 (ASCII -kkv
Order.java 1.1 (ASCII -kkv)
Payment.java [S:18] [I>>] 1.1
Shipping.java 1.1 (ASCII -kkv)
Outline
name : String
customerType : int
payment : Payment
accountHolderZip : String
CreditCard.java
Problems
Javadoc
Palantír Connections
/Store/src/store/CreditCard.java
Conflicting: 2

```
    private static int MINOR = 0;
    private static int ADULT = 1;

    //custructor for credit card, para Customer
    public CreditCard(Customer customer){
        payment = new Payment(Payment.creditCard,
        Address address = customer.getAddress();
        accountHolderZip = address.getZip();
        name = address.getName();

        Demographics demo = customer.getDemograpl
        int age = demo.getAge();
        if (age >= 18)
            customerType = ADULT;
        else
            customerType = MINOR;

    public String getName() {
        return name;
    }
```

Author	ImpactType	Resource	Re…	ChangeStatus	Timest…	Reason
Pete	IMPACTED BY	/Store/src/store/Address.java1.1	Ellen	Changes Committed	Fri Sep…	Modified class: De
Pete	IMPACTED BY	/Store/src/store/Customer.java1.1	Ellen	Changes In Progress	Fri Sep…	Modified class: De
⚠ Pete	IMPACTED BY	/Store/src/store/Payment.java1.1	Ellen	Change Added	Fri Sep…	Modified class: Ad

Fig. 8.2. Palantír workspace awareness

The purpose of these cues is to unobtrusively draw the user's attention to emerging conflicts, both direct and indirect, without undue distractions or overwhelming the user with too much information (Fig. **8.2.**). Palantír currently detects indirect conflicts that arise because of changes to public methods and variables [45]. Palantír was integrated into the Eclipse development environment such that annotations in the package explorer view inform developers of activities in other workspaces (top inset in Fig. **8.2.**) and a new Eclipse view, the conflict view, allows users to obtain further details of changes causing conflicts (bottom inset in Fig. **8.2.**). The goal is for the textual annotations to warn developers of impending conflicts and when the user needs further information, they can investigate the conflict via the Palantír conflict view, where various kinds of icons provide additional information about the state of a conflict.

Objective of evaluation process and steps taken

Palantír is one of the more mature prototypes in the CC tool suite. Therefore it has iteratively undergone several evaluation approaches. Qualitative screening by surveying other tools via literature survey and iterative qualitative effects analysis, to get feedback from experts, helped us determine its specific awareness and display features early on in the project. We then validated the feasibility of our approach via feature analysis experiments, where we integrated Palantír with three CM systems – CVS, RCS, and Subversion. We subsequently performed initial qualitative experiments to validate and obtain feedback on our experimental setup before performing our quantitative user experiments. These experiments were designed to test the usefulness of Palantír in enabling participants discover potential conflicts and test its ease of use and the effort required by participants to notice, investigate, and resolve conflicts in their tasks.

The experiments were specifically designed to observe a participant making edits in a group setting with (and without) using Palantír to coordinate their changes. Particular individual differences that concern our experiment are differences in how a team member interacts in the group and a programmer's technical skills. We controlled for differences in group interaction by using confederate based design, similar to Lighthouse evaluations, where a participant could interact with the two other team members via IM.

We controlled individual differences that stem from technical skills by conducting stratified random assignment. Further, we benchmarked the non-programming tasks evaluations with our results from an analogous experiment with programming tasks. In "textual" experiment, we chose a sample text that was neither too complex nor too interesting to overwhelm or distract the participants. The text reflected some key properties of software, primarily modularity and dependency. Modularity was attained by using text which was comprised of separate files (chapters). Whereas, dependency was simulated by text containing references that linked text across modules and which had to be kept consistent. The textual experiment was followed by a "Java" experiment to evaluate Palantír in the programming domain. This experiment sought to confirm results from the first experiment. However, here we sought to takes into account the limitation of the programmer's individual differences becoming visible, especially in the time it takes for them to complete change tasks.

Lessons learned with respect to the tool

The evaluation of Palantír sought answers to three principle questions regarding the tool's usefulness and usability. Firstly, *does workspace awareness help users in their ability to identify and resolve a larger number of conflicts?* We found with statistical significance that participants in the Experiment group detected and resolved a larger number of conflicts for both conflict types (direct and indirect). We found that participants typically noticed information provided by Palantír before embarking on their task or right after finishing it. Secondly, *does workspace awareness affect the time–to-completion for tasks with conflicts?* An obvious effect of workspace awareness tools is the fact that they incur some extra overhead as developers must spend time and effort to monitor the information that is provided to them. Further, if they suspect a conflict then they spend time and effort to investigate and resolve it. We examine this overhead by comparing the average time, which includes the time to detect, investigate, coordinate, and resolve a conflict that participants in each of the treatment groups took to complete tasks. We found that on average participants using Palantír detected a larger number of conflicts without significant overheads. Finally, *does workspace awareness promote co-ordination?* We observed that on detecting a conflict participants generally took one of the following actions: synchronize, update, chat, skip the particular task, or implement the task by using a placeholder. In general, we saw a comparable number of co-ordination actions for direct conflicts between the control and experiment groups, but a sharp increase in the number of co-ordination actions for indirect conflicts for the experiment group.

Lessons learned with respect to the evaluation of tool

Our experiments led us to conclude that evaluating co-ordination tools that require a group of people to understand and use the information provided to co-ordinate with each other is extremely complex. While we took great care to control individual differences between participants we still found large enough variances in the time to completion of tasks. Another way of controlling individual differences would have been to perform a between subject test, i.e., test the same participant in both the control and experiment conditions using two very similar projects. Additionally, in our experiment we seeded the same type of conflicts in the same order. It is possible that participants may learn from past conflicts and change their behavior with how they react to new conflicts; therefore, changing the order in which we introduced the direct and indirect conflicts may produce different results. Finally, in the Java experiment, participants were not required to integrate their changes and build the entire project. Therefore, nearly all participants in the control group and some in the experiment group did not detect the conflicts remaining in the code base. This fact combined with the fact that we did not penalize the task with unresolved conflicts precluded us from quantifying the benefits of workspace awareness with respect to the time and effort saved in co-ordination. While this experiment design decision was disadvantageous, finding the perfect balance between the amounts of time required for participants to learn about the tool, complete tasks, and the complexity of the project is not trivial.

8.4.5 YANCEES

Notification servers (or publish/subscribe infrastructures) support the continuous co-ordination requirements of disseminating information from distributed information producers to different information consumers in a timely fashion [39]. They provide mechanisms for publishing, routing, filtering and disseminating information in the form of events. As such, publish/subscribe infrastructures have been used in support of different event-driven applications [12, 17, 26, 46]. Whenever a new event-driven application is conceived, developers face two alternatives: build a publish/subscribe infrastructure from scratch, or reuse one of many existing research and industrial systems. A qualitative screening of existing publish/subscribe infrastructures revealed different architectural patterns adopted by industrial and research publish/subscribe infrastructures in the support of the evolving and heterogeneous requirements of different application domains [48]. For example: minimal core, one-size-fits-all, co-ordination languages and compositional models. Most of these patterns are neither extensible nor configurable in the set of features they provide, making their adaptation and reuse a difficult endeavor. This observation motivated the development of YANCEES, which is an extensible and configurable publish/subscribe infrastructure based on plug-ins [47]. As such, our goal in the development of YANCEES was twofold. First, from the infrastructure developers' perspective, we sought the reduction of the development effort. Second, from the point of view of infrastructure consumers, we sought an infrastructure that can reduce the development effort of event-driven applications. In order to evaluate these goals, we designed the following evaluation.

Objective of evaluation process and steps taken

Our evaluation had three major objectives. First, we sought to assess the usefulness of YANCEES i.e., its ability to support the performance and application-specific requirements of different application domains. Second, we sought to evaluate its usability, which is measured as the development effort of both infrastructure developers and consumers. Finally, our evaluation compares these measures with existing approach in both the literature and industry. We took the following steps to achieve these goals:

1. We performed qualitative screening of industrial and research infrastructures with the goal of identifying major architectural patterns adopted by these tools in the support of different application domains requirements. The screening revealed four new alternatives which included: (a) employ generalization in the construction of minimal APIs; (b) support extensibility through the use of co-ordination languages; (c) employ variation in the construction of one-size-fits-all infrastructures; (d) or support flexibility by the use of component frameworks as is the case with YANCEES.
2. We selected a set of open source infrastructures, one for each category to be compared in a benchmark. These included Siena [6] representing generalized minimal APIs; Sun JavaSpaces [20] representing co-ordination languages; CORBA Notification Service (or CORBA-NS) [38] representing one-size-fits-all infrastructures, and YANCEES [47] representing flexible compositional infrastructures
3. We selected three feature-rich event-driven application domains as the source of requirements for our study. These were usability monitoring represented by EDEM, awareness represented by CASSIUS and collaborative environments represented by Impromptu [17, 28]. These infrastructures were selected first for their diversity of requirements, and second, for the previous experience of the authors in their development,

which provides both access to the source code, and expertise in their set of requirements.

4. The requirements of each application were then abstracted into a set of reference APIs representing ideal features that a publish/subscribe infrastructure must support in each domain. We implemented each one of these tree reference APIs using the four selected infrastructures. We also implemented each API from scratch, as a control implementation.
5. Finally, we performed a quantitative evaluation of the resulting implementations, measuring their average responsiveness and the total development effort of each. The development effort is calculated as the product of the number of lines of code (LOC) and the McCabe Cyclomatic Complexity (or McCabeCC) of the code required to adapt each infrastructure in the implemnetatoin of each API [33]. The goal of the performance benchmark in our study is to determine the usefulness of the infrastructure, in serving its purpose within the requirements of each application domain.

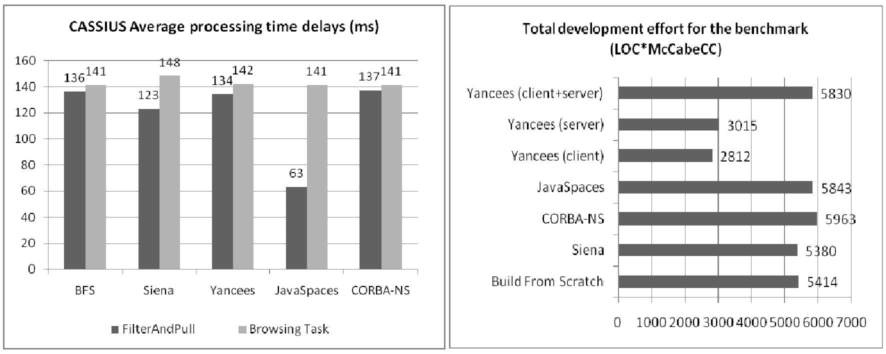


Fig. 8.3. YANCEES Performance Benchmark (left) and Comparative Development Effort (right)

Lessons learned with respect to the tool

In our performance benchmarks, we compared responsiveness of an infrastructure implemented with YANCEES with the same infrastructure implemented reusing the other infrastructures. The results of one of the three benchmarks are shown in Fig. **8.3.** (left). The results show that YANCEES performance is comparable to that obtained by reusing existing infrastructures or even to the cases where the APIs are building from scratch.

This demonstrates YANCEES ability to support the requirements of different application domains and its usefulness in supporting the development of application-specific infrastructures, with no significant performance penalty.

We also compared the total development effort (measures at the product of LOC and McCabeCC) to determine the usability of YANCEES when the other infrastructures are used to support the three application domains Fig. **8.3.** (right). It is important to note that infrastructures (e.g., Siena, CORBA-NS, etc.,) are reused as black-boxes. They are extended “from the outside”, by building the required functionality around their provided APIs. YANCEES, on the other hand, is configurable and extensible “from the inside”, allowing the modification of the set of features its supports. This fundamental difference is reflected in the graphs of (Fig. **8.3.**) where both client and server side development efforts are shown, together with combined effort (client + sever) in a separate bar.

Fig. **8.3.** (right) demonstrates that while the total cost of reuse of YANCEES in all the three scenarios (client + server) is comparable with existing approaches, its ability to separate client and server-side development has two important advantages. First, it allows the separation between publish/subscribe infrastructures producers and consumers, dividing the development effort (the two bars: YANCEES client and YANCEES server in Fig. **8.3.**). Second it reduces the application development effort, since the infrastructure can be configured and extended to support the exact application-specific set of features required by the application domain. This is made evident by the lower YANCEES client effort (Fig. **8.3.**). Contrary to our expectations, the total (server + client) side development effort when using YANCEES was not significantly lower than the other approaches. This can be the consequence of the additional effort devoted to configuration and extension of the infrastructure.

Lessons learned with respect to the evaluation

When comparing different software infrastructures, developed with different original goals, it is important to strive for a fair evaluation process. Different strategies were adopted in the design of our benchmark to increase equitable comparison between the different approaches. First, we chose to implement the benchmark ourselves to eliminate the variance that may come by the use of different developers at different levels of expertise. Second, we adopted best of breed design practices in all implementations [21] and modularized common features into components that were reused throughout the different implementations. We also adopted the same algorithms used by the original applications (EDEM, CASSIUS, Impromptu) we emulated. Finally, we aligned the different implementations to follow the same task structure. This facilitates our data collection and analysis.

These strategies collectively increase the likelihood that code style, algorithms and overall software architecture were similar throughout our experiments. Finally, the benchmark tests were conducted in the same set of machines (one client and one server), connected via a 100 Mbps local Ethernet, thus providing a constant environment.

While the overall comparison of different infrastructures reusability based on the number of LOC and McCabeCC allows the comparison of the total development effort of these infrastructures, they do not reveal important details about the individual concerns and costs involved in each approach. For example, the costs of adaptation, extension and configuration. In order to investigate these costs in more details, we are currently conducting a finer-grained analysis of the code uses in our benchmark.

8.5 Discussion

Our goal was to evaluate the usability and usefulness of different co-ordination tools constituting our CC tool suite. Our tools were motivated by findings from a set of ethnographic studies on co-ordination in software teams [14, 16] and a qualitative screening of existing co-ordination tools. While these studies formed the basis on which we determined the usefulness of the tool features, each tool's usefulness and usability was further evaluated using the DESMET evaluation framework. The particular approach used for a particular tool was determined based on its functionality, the specific aspect that was being evaluated (usefulness or usability), and the maturity level of the tool.

The majority of our tools strive to provide appropriate information of ongoing project activities to the user, therefore, a primary goal of our evaluations was to study the usefulness of the tools based on whether a tool achieved an appropriate level of abstraction. Depending on the desired functionality of a co-ordination tool and the target audience, different levels of data abstraction are required, which can then be visualized via text, tables, charts, or other visualization metaphors. Most of our CC tools have a visualization component. These components vary, from being completely unobtrusive and subtle, such as information display as extensions to the development editor, or more intensive displays requiring separate stand alone visualizations that work best in auxiliary display units (second monitor or ambient devices) or as large scale visualization that acts as a command control center. Thus, a key evaluation criterion was to assess the usefulness of a tool's display technique. In particular, we investigated the tradeoff between the amounts of information that was displayed and the obtrusiveness of the display. Towards this goal, we observed that *qualitative effects* analysis and *usability inspections* served as a good first level of analysis to obtain user feedback. Further, most software projects are large, which requires that our tools can scale well to large data sets. Towards assessing the scalability of our tools, we used *quantitative case studies*, namely, using our tools to visualize large scale open source projects and then interviewing developers or managers from those projects to obtain their feedback.

The next important criterion for our evaluation was to test the usability of our tools. We primarily evaluated the usability of a tool by investigating the trade-offs between the efforts users are willing to expend in operating and/or learning a tool, versus the estimated benefits gained. Moreover, since many visualization tools rely on novel metaphors to help users interpret and navigate the vast information space generated by software, it is important to evaluate the time and effort it takes users to understand visualizations. Therefore, we also evaluated the effort expended by individuals to understand the information provided by a tool via user experiments (both qualitative and quantitative).

We found that two challenges are typically encountered when evaluating co-ordination tools: (1) differences in outcome because of differences in the technical aptitude of participants and (2) differences in how a group reacts to tasks and conflicts. Through our experiments, we sought to control for both these differences. We controlled for differences in technical aptitude by stratifying our participants based on their background and then randomly selecting participants from each stratum. Further, we benchmarked our results first by using non-programming tasks and then confirming these results in a programming domain. We controlled for differences in group interactions by using confederate based design, which ensured consistency in the kind and timing of conflicts, as well as group interactions via IM.

While we took special care to control external factors to be able to test specifically the usability of our tools, our study suffers from the common external threats to research validity that arise in user experiments. For example, selecting students as participants in several of our evaluation threatens the ability to generalize from our results. We sought to recruit different participants each time with varying levels of expertise (i.e., graduate and post-graduate students) to limit this threat. We used confederates to achieve consistency in our experiments. As such, we realize that results can differ if events are introduced closer to the completion of the task or at random intervals, as may happen in practice. Thus, the controlled introduction of an event at a specific time can also threaten the generality of our results.

We note that the evaluation of a co-ordination infrastructure tool such as YANCEES, require different evaluation methods than other frontend co-ordination tools. Therefore, its evaluations follow a slightly different format, although they still fall within the DESMET evaluation framework. We tested the usefulness of YANCEES mainly through qualitative screening; and the usability and robustness of YANCEES by implementing three feature rich applications using YANCEES and three other competing event notification services. A quantitative evaluation of the resulting implementations was performed that assessed the average responsiveness and total development effort required per implementation.

Finally, we encountered internal threats in the form of bias that may have been introduced during our qualitative screening evaluations. The potential for bias also exists in the feedback participants provided because tool developers typically conducted the experiments and were direct recipients of the feedback. We strove to minimize the impact of these threats by conducting a combination of different evaluation approaches for each tool.

8.6 Conclusions

In this paper, we described a set of co-ordination tools as known the CC tool suite. The focus of this chapter was to describe in detail the different evaluation methodologies that we followed for assessing the usefulness and usability of our tools. In conclusion, we maintain there is no one evaluation method for a tool; rather, tools should be iteratively evaluated using multiple evaluation methods to obtain well rounded evaluation results. We found that a different evaluation methodology is often needed to assess usefulness or usability aspects of a tool. Finally, the experience acquired while researching continuous co-ordination has led us to conclude that we need to consider the co-ordination information in terms of *what, how*, *when*, and *who* shares it, which means that the evaluation of these tools would benefit from evaluating whether these aspects of the tools addressed developers' needs.

Future plans for each tool were specified in their respective sections. However, we have specific tasks ahead of us that hold true for most of the CC tools at both the individual and organizational level. For example, at the individual level we need to evaluate the impact the order of events has on the outcome of our evaluations and the possible co-ordination patterns that can emerge. CC tools typically share potentially sensitive information thus it would be beneficial to investigate the issues relating to individual privacy and data confidentiality. Both are important issues that need to be carefully assessed to design usable co-ordination tools. Co-ordination tools can fail if individuals perceive that the tool is used as a managerial performance metric or used by their competitors [23]. We also need to evaluate the use of CC tools within an organizational context. A tool that requires changes to the typical workflow in an organization will generally encounter more resistance because potential users do not readily change their work processes to adopt a new tool.

Acknowledgements

This research was supported by the U.S. National Science Foundation under grants 0534775, 0326105, 0093489, and 0205724, by the Intel Corporation, by two IBM Eclipse Technology Exchange grants, and an IBM Technology Fellowship.

References

[1] Amrit C (2005) Co-ordination in software development: the problem of task allocation, In Proceedings of the 2005 workshop on Human and social factors of software engineering (ACM) St. Louis, Missouri, pp 1 - 7

[2] Andrews K (2006) Evaluating Information Visualizations, In AVI workshop on beyond time and errors: novel evaluation methods for information visualization (ACM) Venice, Italy, pp 1-5

[3] Ardito C, Buono P, Costabile MF, Lanzilotti R (2006) Systematic inspection of information visualization systems, In AVI workshop on beyond time and errors: novel evaluation methods for information visualization (ACM) Venice, Italy

[4] Barkhuus L, Rode JA (2007) From Mice to Men - 24 Years of Evaluation in CHI, In CHI 2007 http://www.viktoria.se/altchi/

[5] Card SK, Mackinlay JD, Shneiderman B (1999) *Readings in Information Visualization: Using Vision to Think.* Morgan Kaufmann San Francisco, CA

[6] Carzaniga A, Rosenblum DS, Wolf AL (2001) *Design and Evaluation of a Wide-Area Event Notification Service.* ACM Transactions on Computer Systems, (19) 3:332-383

[7] Cataldo M, Wagstrom PA, Herbsleb JD, Carley KM (2006) Identification of Co-ordination Requirements: Implications for the Design of Collaboration and Awareness Tools, In *Computer Supported Co-operative Work*. Banff, Alberta, Canada, pp 353 - 362

[8] Crowston K (1997) *A Co-ordination Theory Approach to Organizational Process Design.* Organization Science, (8) 2:157-175

[9] da Silva I, Chen P, Van der Westhuizen C, Ripley R, Hoek Avd (2006) Lighthouse: Co-ordination through Emerging Design. In: *OOPSLA workshop on Eclipse Technology eXchange*, Portland, Oregon, pp 11-15

[10] Damianos L, Hirschman L, Kozierok R, Kurtz J, Greenberg A, Walls K, Laskowski S, Scholtz J (1999) *Evaluation for collaborative systems.* ACM Computing Surveys, Vol. 31, Issue 2es, Article No. 15

[11] de Souza CRB (2005) On the Relationship between Software Dependencies and Co-ordination: Field Studies and Tool Support, In *Ph.D. dissertation, Donald Bren School of Information and Computer Sciences*, UC, Irvine, CA

[12] de Souza CRB, Basaveswara SD, Redmiles D (2002) Lessons Learned Using Notification Servers to Support Application Awareness, In: *Meeting of the Human Computer Interaction Consortium,* Frasier, CO

[13] de Souza CRB, Froehlich J, Dourish P (2005) Seeking the source: software source code as a social and technical artifact, In *ACM SIGGROUP Conference On Supporting Group Work*, Sanibel Island, Florida, USA, November 06 - 09 2005, pp 197-206

[14] de Souza CRB, Quirk S, Trainer E, Redmiles DF (2007) Supporting Collaborative Software Development Through the Visualization of Socio-technical Dependencies, In 2007 International ACM Conference on Supporting Group Work (ACM) Sanibel Island, Florida, USA, pp 147-156

[15] de Souza CRB, Redmiles D, Cheng LT, Millen D, Patterson J (2004) How a Good Software Practice Thwarts Collaboration - The Multiple roles of APIs in Software Development. In: *Foundations of Software Engineering,* Newport Beach, CA, October 31 to Nov 5 2004, (ACM Press) pp 221 - 230

[16] de Souza CRB, Redmiles D, Dourish P (2003) "Breaking the code", moving between private and public work in collaborative software development, In *ACM SIGGROUP Conference on Supporting Group Work (ACM)*, Sanibel Island, Florida, USA, November 9-12 2003, pp 105 - 114

[17] DePaula R, Ding X, Dourish P, Nies K, Pillet B, Redmiles D, Ren J, Rode J, Silva Filho RS (2005) *In the Eye of the Beholder: A Visualization-based Approach to Information System Security*. International Journal of Human-Computer Studies - Special Issue on HCI Research in Privacy and Security, Vol. 63, Issue 1-2, pp 5-24

[18] Ellis JB, Wahid S, Danis C, Kellogg WA (2007) Task and social visualization in software development: evaluation of a prototype, In *SIGCHI conference on Human factors in computing systems,* San Jose, CA, Apr 28 - May 03 2007, pp 577-586

[19] Ericsson KA, Simon HA (1993) *Protocol Analysis - Rev'd Edition: Verbal Reports as Data* (MIT Press Cambridge, MA) p 496

[20] Freeman E, Hupfer S, Arnold K (1999) *Java Spaces Principles, Patterns, and Practice*. The Jini Technology Series (Book News, Inc.,)

[21] Gamma E, Helm R, Johnson R, Vlissides J (1995) *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Professional Computing Series (Addison-Wesley Publishing Company)

[22] Green TRG (1989) Cognitive Dimensions of Notations, In *Fifth Conference of the British Computer Society, Human-Computer Interaction Specialist*

Group on People and Computers (Cambridge University Press) Univ. of Nottingham, pp 433-460

[23]Grudin J (1988) Why CSCW applications fail: problems in the design and evaluation of organization of organizational interfaces. In: *Computer Supported Co-operative Work,* Portland, Oregon, pp 85-93

[24]Grudin J (1994) *Groupware and Social Dynamics: Eight Challenges for Developers.* Communications of ACM, (37) 1:92-105

[25]Hedberg H, Lappalainen J (2005) A Preliminary Evaluation of Software Inspection Tools, with the DESMET Method, In *International Conference on Quality Software* (IEEE Computer Society) 19-20 Sept 2005, pp 45-54

[26]Hilbert D, Redmiles D (1998) An Approach to Large-scale Collection of Application Usage Data over the Internet. In: *20th International Conference on Software Engineering (ICSE '98),* Kyoto, Japan. April 19-25 1998, (IEEE Computer Society Press) pp 136-45

[27]Huang EM, Mynatt ED, Russell DM, Sue AE (2006) *Secrets to Success and Fatal Flaws: The Design of Large-Display Groupware.* IEEE Computer Graphics and Applications, (26) 1:37-45

[28]Kantor M, Redmiles D (2001) Creating an Infrastructure for Ubiquitous Awareness. In: *Eighth IFIP TC 13 Conference on Human-Computer Interaction (INTERACT 2001),* Tokyo, Japan, pp 431-438

[29]Kitchenham BA (1996) Evaluating Software Engineering Methods and Tool Part 1: The Evaluation Context and Evaluation Methods. SIGSOFT Software Engineering Notes, (21) 1:11-14

[30]Lethbridge TC, Sim SE, Singer J (2005) *Studying Software Engineers: Data Collection Techniques for Software Field Studies.* Empirical Software Engineering, (10) 3:311-341

[31]Lungu M, Lanza M, Girba T, Heeck R (2007) Reverse Engineering Super-Repositories, In Proceedings of the 14th Working Conference on Reverse Engineering (IEEE Computer Society) pp 120-129

[32]Malone TW, Crowston K (1994) *The Interdisciplinary Study of Coordination.* ACM Computing Surveys, (26) 1:87-119

[33]McCabe TJ (1976) A Complexity Measure, In IEEE Transactions on Software Engineering, December 1976, pp 308-320

[34]Mealy E, Strooper P (2006) Evaluating Software Refactoring Tool Support, In Proceedings of the Australian Software Engineering Conference (IEEE Computer Society) pp 331-340

[35]Michelis GD, Loregian M, Martini P (2006) Directional Interaction with Large Displays Using Mobile Phones, In Proceedings of the 4th annual IEEE international conference on Pervasive Computing and Communications Workshops (IEEE Computer Society) 5:13-17

[36]Morera D (2002) COTS Evaluation Using DESMET Methodology & Analytic Hierarchy Process (AHP), In 4th International Conference on Product Focused Software Process Improvement (Springer-Verlag) pp 485-493

[37]Nielsen JK (1994) *Heuristic Evaluation.* Wiley, New York

[38]OMG (2004) CORBAcos Notification Service version 1.1 formal/04-10-13, (Object Management Group) p 229

[39] Patterson JF, Day M, Kucan J (1996) Notification servers for synchronous groupware, In: *ACM conference on Computer supported co-operative work (CSCW'96),* Boston, Massachusetts, pp 122-129

[40] Plaisant C (2004) The Challenge of Information Visualization Evaluation, In Working Conference on Advanced Visual Interfaces (ACM) Gallipoli, Italy, pp 109-116

[41] Ramage M (1999) The Learning Way: Evaluating Co-operative Systems, PhD Dissertation. In: *Department of Computer Science*. Lancaster, UK, p 142

[42] Redmiles D, Hoek Avd, Al-Ani B, Hildenbrand T, Quirk S, Sarma A, Silva Filho R, de Souza CRB, Trainer E (2007) *Continuous Co-ordination: A New Paradigm to Support Globally Distributed Software Development Projects.* Wirtschaftsinformatik, Vol. 49, pp 28-38

[43] Ripley R, Sarma A, Hoek Avd (2007) A Visualization for Software Project Awareness and Evolution. In: *Workshop on Visualizing Software for Understanding and Analysis,* Alberta, Canada, pp 137-144

[44] Russell J, Roberts C (2001) *Angles on Psychological Research* (Nelson Thornes Ltd. p 256

[45] Sarma A, Bortis G, Hoek Avd (2007) Towards Supporting Awareness of Indirect Conflicts across Software Configuration Management Workspaces. In: *Conference on Automated Software Engineering,* Atlanta, USA, pp 94-103

[46] Sarma A, Noroozi Z, Hoek Avd (2003) Palantír: Raising Awareness among Configuration Management Workspaces. In: *Twenty-fifth International Conference on Software Engineering,* Portland, Oregon, USA, pp 444-454

[47] Silva Filho RS, Redmiles D (2005) Striving for Versatility in Publish/Subscribe Infrastructures. In: *5th International Workshop on Software Engineering and Middleware (SEM'05),* Lisbon, Portugal, 5-6 September 2005, (ACM Press) pp 17 - 24

[48] Silva Filho RS, Redmiles DF (2005) A Survey on Versatility for Publish/Subscribe Infrastructures. Technical Report UCI-ISR-05-8, (Institute for Software Research) Irvine, CA, pp 1-77

[49] Sjoberg DIK, Dyba T, Jorgensen M (2007) The Future of Empirical Methods in Software Engineering Research. In: *Future of Software Engineering, 2007. FOSE '07,* pp 358-378

[50] Storey MA, Cheng LT, Bull I, Rigby P (2006) Shared waypoints and social tagging to support collaboration in software development, In Proceedings of the 2006 20th anniversary conference on Computer supported co-operative work (ACM) Banff, Alberta, Canada, pp 195-198

[51] Trainer E, Quirk S, de Souza CRB, Redmiles DF (2008) Analyzing a Socio-Technical Visualization Tool Using Usability Inspection Methods, In *IEEE Symposium on Visual Languages and Human Centric Computing*, Washington, DC, pp 78-81

[52] Tufte E (1990) *Envisioning Information* (Graphics Press Cheshire, CT) p 126

[53] Tufte E (2006) *Beautiful Evidence* (Graphics Press Cheshire, CT) p 213

[54] Wasserman S, Faust K (1994) *Social Network Analysis: Methods and Applications* (Cambridge University Press) Cambridge, UK

[55]Wharton C, Rieman J, Lewis C, Polson P (1994) The Cognitive Walkthrough Method: A Practitioner's Guide, In: *Usability inspection methods*, (Eds.) Wiley, pp 105-140

[56]Xiaojun B, Yuanchun S, Xiaojie C (2006) uPen: A Smart Pen-liked Device for Facilitating Interaction on Large Displays. In: *First IEEE International Workshop on Horizontal Interactive Human-Computer Systems*, Adelaide, South Australia. 5-7 January 2006

[57]Zuk T, Schlesier L, Neumann P, Hancock MS, Carpendale S (2006) Heuristics for Information Visualization Evaluation, In *AVI Workshop on Beyond time and errors: novel evaluation methods for information visualization.* (ACM) Venice, Italy

Appendix A

Table 14.1 Summary of evaluation approaches for each CC tool

Tool	**Purpose of tool**	**Purpose of Evaluation**	**DESMET**	**Evaluation Approach**

Ariadne	Allows developers to explore and analyze socio-technical dependency information.	Identify usability issues with Ariadne's visual interface.	Qualitative screening;
			Qualitative-effects analysis;
			Quantitative case-study.
				Usability inspection methods:
				Heuristic Evaluation
				Cognitive Walkthrough
				Cognitive Dimensions of Notations
				Tufte's principles of information presentation.

WAV	Provides a 3d view of all parallel workspace activities and supports playback over time.	Demonstrate accuracy of data collected and to test scalability of the visualization.	Qualitative screening;
			Qualitative-effects analysis;
			Quantitative case-study.
				Post-mortem analysis (e.g. case study) of existing open-source projects and validation of results with project members.

Lighthouse	Creates the Emerging Design, an always up-to-date abstraction of the software code.	Observe the usefulness of Lighthouse in helping users understand ongoing project activities, detect emerging conflicts, and communicate with team members.	Qualitative screening;
			Qualitative-effects analysis;
			Qualitative experiments.
				Informal user experiment involving
				Observation by experimenters
				Think aloud techniques
				Exit survey

Palantír	Promote workspace awareness by transmitting information of ongoing project activities to detect emerging direct and indirect conflicts at real time	Statistically determine the usefulness of Palantír in detecting emerging conflicts and promoting coordination to resolve conflicts	Qualitative screening;
			Qualitative-effects analysis;
			Qualitative experiments (user experiment; interoperability);
			Quantitative experiment;
			Benchmarking of experiment results
				Formal sser experiment (Benchmark: use text data to control for individual difference arising due to difference in technical skills;
				Confederate design: control the type, number, and timing of conflicts to overcome variances in group interaction)
				Observation by experimenters
				Think aloud techniques
				Exit survey

YANCEES	Improve the support for heterogeneous set of requirements of Continuous Coordination tools	Assess the reusability and performance of YANCEES, comparing the results with existing approaches	Qualitative screening to determine existing approaches. Quantitative experiment: reusability Benchmarking: performance	Implement three APIs based on selected infrastructures, measuring the development effort. Benchmark: evaluate the performance of the resulting implementations