Supporting Collaboration Awareness in Multi-developer Projects

Master's Thesis submitted to the
Faculty of Informatics of the University of Lugano
in partial fulfillment of the requirements for the degree of
Master of Science in Informatics
Software Design

presented by
Anja Guzzi

under the supervision of
Prof. Dr. Michele Lanza and Lile Hattori

June 2009

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Anja Guzzi
Lugano, June 2009

“Qualunque cosa tu possa fare, o sognare di fare, incominciala. L’audacia ha in sé genio, potere e magia. Incomincia adesso.”

Johann Wolfgang Goethe

Abstract

Teamwork is necessary to produce large software systems in a reasonable amount of time. A team of developers working on the same project must deal with concurrent development. Collaboration among team members assumes a fundamental role during the whole development process of systems.

Failing to appropriately take care of collaboration aspects, such as awareness, communication and synchronization, can result in the delay of a whole project. However, the negative consequences of uncoordinated concurrent development can be reduced with tool support for collaborative software development.

We developed Scamp, an Eclipse Plug-in conceived to support collaboration awareness through visualization. Scamp is built on top of Syde, which provides an environment for synchronous development. Relying on the underlying structure, Scamp visualizes changes in a system *as they happen* in three different ways: a distinctive mark on changed entities, a Tag Cloud and a "Buckets view".

Acknowledgements

Thanks to..

- Professor Lanza, for his passionate way of teaching and for saying "*cool!*".
- Lile Hattori, for her assistance through the development of Scamp and for her constructive comments.
- My parents and Bea, for always being there for me.
- ...all who believe in me. In particular to Dada, for her amazingly constant support (Grazie!), and to the "Dutch crew", for being so close despite the physical distance (Dank je!).

I am really grateful to all of you.

Contents

Figures

Tables

Chapter 1

Introduction

A software system, other than being intrinsically complex, is the result of a work "of multiple hands". There are many technologies which support and enable teamwork. As a consequence, a scenario where a team of developers physically de-located is cooperating to produce the same software system, is nowadays an ordinary experience. We call this process of building software systems in teams: *collaborative software development*.

The ability to widen the environment in which developers work (from a single room, to potentially the whole world) brings on some fundamental problems along with the many advantages. One of the issues concerns **collaboration**. There are three aspects of collaboration in which we are interested: awareness, communication and synchronization.

- With **awareness** we mean an understanding of the activities of others to provide a context for one's activities [DB92]. Knowing other's activities can prevent uncoordinated changes that could cause unwanted (side)effects and/or merging problems. The thesis is mostly focused on this aspect.
- Cooperation willingness can be highly compromised in absence of face-to-face **communication**. Moreover, e-mails and mailing lists can hardly fully compensate the lack of a direct verbal communication. On the other hand, phone calls, instant messaging and video conferences can overload developers, interfering with and slowing their work.
- Code **synchronization** is probably one of the most tangible aspects in a multi-developer context, which gets commonly translated into the homonymous feature provided by software repositories. However, despite the existence of such repositories, synchronization is still a bottleneck for collaborative software development. In fact, the traditional and most popular

repositories (such as CVS and SVN) employ the check in/check out model, which inevitably introduces delays in code synchronization.

Finding a way to ease collaboration between members of development teams is an issue of central importance. Providing means to improve collaboration awareness, communication and synchronization improves the quality of development and reduces project delays, in particular when the developers are geographically distributed.

We propose to support collaboration awareness through visualization. Scamp[1], the Eclipse plug-in we implemented, provides different views that inform users about ongoing changes in a software system.

1.1 Related Work

A number of academic efforts have been made in the past years toward collaborative software development.

COAST[2] [SKSH96; SSS99] is a toolkit conceived in 1996 to enhance the usability and simplify the development of synchronous groupware (a groupware is a software designed to facilitate team work). As regards awareness, COAST emphasizes the importance for users to receive up-to-date information about others' activities, identifying three main goal to achieve: data consistency, availability of *WYSIWIS*[3] views and support for different teamwork situations. The toolkit, written in Smalltalk, is based on shared documents and provides a framework for views development. The COAST views concept is to be automatically updated at any shared data object change.

In the area of web collaborative applications, another object-oriented platform has been developed to facilitate coordination. The TOP[4] groupware [GPF99] manages notifications such as users arrival and departure to a work session, messages exchange and objects sent to repositories by users. TOP does not explicitly provide any particular assistance to awareness.

Schneider *et al.* [SGPP04] developed *ProjectWatcher* with the aim to support group awareness by monitoring changes in and usage of API. Changes made during development are recorded into *shadow repository*, which is then mined. Information gathered from local snapshots has a finer granularity and is more complete than what can be mined from the more common shared software repositories (such as CVS/SVN), therefore has to be preferred in regard to

[1] *Supporting Collaboration Awareness* in M*ulti-developer* P*rojects*
[2] CO*operative Application System Toolkit*
[3] *What You See* I*s What* I *See*
[4] T*en Objects P*l*atform*

awareness support. ProjectWatcher is an Ecliplse[5] plug-in which mines and visually present information from local interaction histories in two views: activity awareness (*"What is each developer doing?"*) and proximity awareness (*"Who is working near me?"*).

CollabVS [Heg09] by Microsoft Research is a collaborative development environment in Visual Studio. The aim is to enhance the Visual Studio IDE in order to reduce the collaboration problems given by a multi-developer context (either if teams geographically distributed or not).

An interesting approach to model information mined from local history of software developers is implemented in NavTracks [SES05]. The goal is to provide an high-level conceptual understanding of the code, providing file-to-file relationships and allowing navigation into the software information space. NavTracks does not provide support for collaborative development, in that the information remains local, however the tool's concept of interaction tracks has a potential (i.e. using a client/server model) to be useful in a collaborative environment.

Noteworthy in the mining, modeling and visualization of information gathered from a single developer's development is SpyWare [RL08] by Robbes. SpyWare, implemented in Smalltalk, is an IDE plugin (it works for Squeak Smalltalk) based on a new concept of model of software evolution, at which core there is the "change" (and not the "version"). It records *semantic* transformations *as they happen*, without requiring the developer to manually "commit" anything. Monitoring all the changes, provides a very fine-grained view on the evolution of the program. On top of SpyWare model, Robbes implemented several tools. The underlying model can also be and has been exploited by other researcher to build their own tools. The different view implemented so far aim to: visualize all the coding sessions of the system (as *sparklines*), follow the evolution of one or more structural metrics, show the list of all the changes (with possibility of querying it), replying changes occurred in a development session, browsing the code at a specific date (allowing comparison between multiple version of the system), let us "grasp" the evolution of the design at a glance (with the possibility to detect design flaws), etc.

Syde [HL09a; HL09b] by Hattori, is an Eclipse plug-in which takes SpyWare change-centric approach and translates it into a collaborative context. Syde provides an environment for synchronous development. Information gathered by Syde can be represented in different views, to underline and support different aspects of collaboration.

Scamp, our prototype, is built on top of Syde.

[5] http://www.eclipse.org/

1.2 Structure of the Document

In Chapter 2 we present the problems affecting multi-developer projects, which is collaboration, and the importance of awareness. We present Syde as a possibility to enhance collaboration awareness and we motivate our thesis.

In Chapter 3 we introduce Scamp, a visualization plug-in we implemented to support collaboration awareness in multi-developer projects. We explain the concept behind Scamp, showing and explaining the offered visualizations.

In Chapter 4 we validate Scamp, by presenting its applications. We present a few multi-developer projects that have been developed with the assistance of Scamp.

We discuss the advantages and the limitations of Scamp and we propose some possibile future work in Chapter 5.

Chapter 2

Multi-developer Projects

In a multi-developer project, the team composition can have an impact on the overall development of projects. Nevertheless, dependencies among people working on the same project are unavoidable in any context [SNvdH03].

Collaborative software development is characterized by two or more people working on the same artifact. When working in teams, cooperation and communication assume a fundamental role: team members need to be aware of each other's work, in order to avoid undesired situations (changes with side-effect, code duplications, and other potentially harmfull activities).

2.1 Collaboration Awareness

Collaboration awareness is the comprehension of others' activities, followed by their contextualization with respect to ours. More clearly, it is the understanding of who is working with you, what they are doing, and how your own actions interact with theirs [GG02]. As already mentioned, awareness is an important aspect when developing software systems in teams. Lack of group awareness can bring a number of troubles, whereas increasing awareness of others' activities can greatly improve the effectiveness of developers and their development experience in general.

A situation where developers are informed of other's activities has many benefit at multiple levels. At low level (code), it can avoid concurrent modification in the same piece of code (and the consequent tedious merging task) and it can prevent circumstances where multiple developers are unaware of performing the same task (wasting working time and effort). At high level (information), knowing "who is doing what", can give us a good understanding of how the work is distributed among the team and who are the *domain experts* for the various software components. Being able to identify who has been working the most

Figure 2.1: Collaboration.

(recently and/or frequently) on a particular file, can give a good indication of whom to ask for explanations or assistance [SGPP04].

On the other hand, an awareness deficit can hinder development: if people do not know what others are doing, they might be afraid of modifying part of the system for which they are not the only responsible, with the purpose of avoiding conflicts and or incompatibilities [HL09b].

For those reasons, we believe that developers should be informed of other's actions. Team member shall be "monitored" and notified as soon as something relevant happens. For example: if two developers are working on the same file, they should know it as soon as possible (instead of "discovering" it at check-in time). This can be done with tool support for collaborative software development, specifically, with tools that both monitor and notify users during development sessions.

A few main requirements can be identified for monitoring and notification processes. First of all, neither the gathering nor the display of data should be disruptive and performance must not be affected. Secondly, to "point out" other's activity in an effective way, the delay between collection of the data and the relative notification, has to be as close as possible to real-time (considering network latency). Additionally, the collection of data has to be transparent to the user (no extra effort must be needed), yet does not have to be invasive (the user does not have to feel "spied", otherwise he/she will neglect the use of the tool). Information collected should possibly be fine-grained both in term of temporality (where capturing every action made is more precise than grouping them) and granularity of entities (where "*method x has changed*" is more precise than "*file y has changed*"). On the notification point of view, we believe that users should be informed in a natural way: not disturbing, but noticeable and clear. Views should display other's activity in an efficient way and only relevant information should be presented.

2.2 State of the Art

Collaboration support is mainly furnished by SCM[1] systems, which principally offer synchronization features. Little effort has however been put in awareness and communication assistance.

SCM repositories contain historical data about software systems, since they hold both source code and information on who committed which change. Such systems are indeed a source of information to determine aspects such as code ownership. Nevertheless their granularity is insufficient (due to the *check-in/check-out protocol*) to track back what happened inside a development session. Moreover a large majority of repositories are file-based (for the purpose of being language independent), reducing further the granularity of information. SCM are therefore a poor source of information for group awareness.

Change-based approaches are being studied and developed. Repositories based on single changes would suit best to a collaborative environment (beside keeping historical information as it is, rather than merged into "arbitrary snapshots").

2.2.1 Software Repositories

Versioning systems (repositories) store snapshots of projects in order to recover past version at any time and easily share projects.

There are two current free mainstream versioning systems, CVS and Subversion, both of which are **file-based**. CVS has been the favorite one for years, however we are experiencing a shift toward Subversion. This because it has a better underlying structure which uses a database rather than RCS files. Subversion also comes along with support for changesets and for refactoring (i.e.: it recognize if a file is moved/renamed). This kind of repositories have two great advantages: they are language-independent (i.e. one can store whatever he likes: code, documents, pictures,..) and they are open-source (their availability is thus very high and with a large community behind). Drawbacks of such approaches are however various, mainly because of their asynchronous nature. In a collaborative context, where team members are working at the same moment on the artifact, developers would have to commit changes to the system as they are made and to frequently update the system with new versions from others. Actually, "*Developers tend to spend quite a lot of time between commits, because they are not comfortable if they commit their changes every 5 minutes, fearing having too much revisions or committing code in an imperfect state (which is not a problem, if the changes committed are tagged appropriately)*" [RL05]. Moreover such a prac-

[1] software configuration management

tice (i.e. committing and updating code with hight frequency) is quite invasive and would disturb the programming session. From an evolution analysis point of view, *snapshot-based* repositories lead to information losses in the history of software systems. In fact, due to their nature of "commits", we see a snapshot as a single big change to the system, losing information about all that happened in-between one snapshot and another (for example it is not possibile to reconstruct the chronological order in which changes have been performed). Concluding, a file-based repository gives information about who committed what (which files/ lines) and at what time. Changes can be measured in number of lines (although changing a single line is seen as removing the old line and then adding a new one). Any further information needs to be mined.

Another kind of versioning system is **entity-based**, where history is kept with a finer granularity (not "files&lines", but "classes&methods"). In order to version entities at the program level, entity-based repositories are consequentially specific to a programming language.An example of such repository based on entities is the StORE repository, which is the versioning system currently in use by Cincom VisualWork Smalltalk[2]. StORE has the advantage that, while it is specific to its "language implementation", every application developed with VisualWork Smalltalk can be "committed" to StORE (creating a quite large *eco-system*). Same as file-based versioning systems, the current entity-based repositories are snapshot based, meaning the changes to the entities will be recorded only when the developer commits. In order to enforce frequent commits, StORE provides a number of tags (such as "broken", "integration-ready" and "release"); one of which has to be chosen at each commit. In this way the developer feel less frightened to commit "work in progress" versions. (Of course the tags are useful also in order to reconstruct the evolution the project, providing additional information about the snapshots.) The direct advantage of entity-based repositories is the information about the software system and the support for refactoring, merging and branching entities; nevertheless to analyze the evolution of the system, further information must be extracted.

> *Analyzing software evolution based on the data provided by mainstream versioning systems corresponds to watching a movie where many frames are missing. - Robbes [RL07]*

Current research, regarding versioning systems, is going toward the development of a new kind of software repository, which would overcome the mentioned drawbacks implied by the current approaches based on concepts such as file and

[2]http://www.cincomsmalltalk.com/

or snapshot. An example of research toward change-based approach is the one implemented in SpyWare by Robbes, which records all the *semantic* changes (seen as first-class entities) on a system, directly from the IDE (developers do not have to perform any commit operation). [RL06]

2.2.2 Awareness Support

If we consider the previously stated requirements for collaboration support in the area of awareness, little or no specific effort has been made. Schneider *et al.* record developers' actions and changes to software artifacts in a shadow CVS repository. "Facts" (about processes, packages, classes and metods) are then mined from the shadow repository and awareness information are visually presented to the users. ProjectWatcher, the Eclipse plug-in implementing this approach, proposes two different views: one focusing on activity awareness and one on proximity awareness. The first view visualizes team members and their current activities in a "project overview" space, where a project artifact is organized in packages, files, classes and methods that gets overlaid by awareness information. The second view allows instead to answer the question "*who is working with me*", by mapping awareness information on a dependency graph [SGPP04]. Even if edits can be auto-committed at a different time intervals, the approach still follow the *check-in*/*check-out* protocol, thus lacking of precious interaction information.

2.3 Syde

2.3.1 Synchronous Development

Hattori proposes a different approach to tackle the collaboration problem, named *synchronous development* [HL09a]. The idea is to provide a complete environment to teams, in order to assist them into their collaborative development.

Syde, the research prototype implemented by Hattori, translates SpyWare's *change-centric* approach into a multi-developers context: it automatically records *every change by every developer*, with the result of a detailed project repository from which fine-grained information can be mined. Data gets collected in a transparent way at every change and notified to a central server (Syde is a client-server application). Then, for each change that compiles successfully, notifications about *what*, *when* and by *whom* has been modified, is immediately broadcasted to clients. Hattori's approach aims to overcome the loss of information

unavoidable in a file-based *check-in/check-out* management systems. However, for now, Syde does not aim to fully replace SCM systems.

The immediate propagation of changes can have a series of positive effects on collaboration, specifically in the awareness area. Developers tend to be afraid of performing modification potentially concurrent (that is, for example, on a set of files that could be modified by others). Moreover, physical distance between team members aggravates this situation, because communication is compromised. Syde can reduce the negative distance effects on collaboration. In an environment specifically addressed to synchronous development, developers might feel more free to apply changes, because they know that they will be informed in case of potential conflicts. In the same way, changes with possible side effects can be early spotted and tedious conflict prevented.

2.3.2 Design & Implementation

Syde is implemented in Java and it is an Eclipse plug-in. Its architecture is composed of the following main components: the *Inspector*, the *Viewer* and the *Requestor* at client side and the *Collector*, the *Notifier* and the *Distributor* at server side. Those six are linked in pairs, as shown in Figure 2.2.

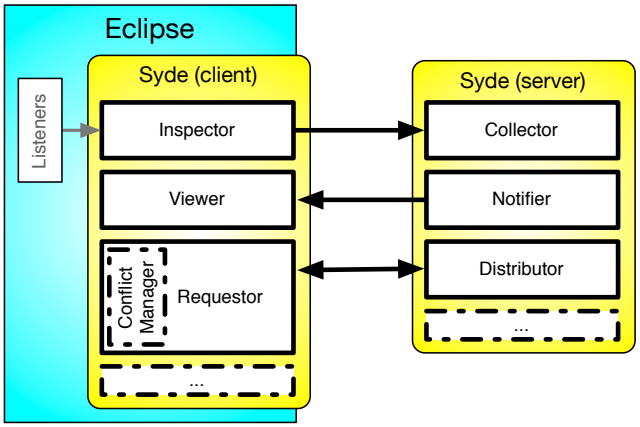

Figure 2.2: Syde Architecture

- First of all, the *Inspector* and the *Collector* are responsible for the capture of changes by developers. To do so, the inspector implements listeners on the Eclipse's workbench to collect metadata containing author's name, timestamp (of the change at client side) and status (successfully compiled or not) of the changes, along with the changes themselves. The collector

component is whereas responsible of receiving changes (sent by the inspector) and storing them into central repository.
From a software evolution point of view, the data stored on Syde's server is worthy and can be used to perform precise analysis on the various projects.

- Secondly, once changes have been recorded, the *Notifier* is responsible for the broadcast propagation of the changes (that successfully compiled) and relative metadata to all registered client (the notifier has a list of all the team members interested in receiving notification about whatever project). On the other side (the client), the *Viewer* receives the information and display them within an Eclipse view. Syde's viewer is the component that, at the end, provides awareness to the users by displaying the changes received from the notifier. Different ways of displaying information from the notifier can be implemented by Syde or by tools built on top of it. An example is Scamp, that we will describe in the next section (see Section 3). Scamp features additional views, in order to enhance awareness. Other tools could be build to cope with additional specific problems.

- Finally there are the *Requestor* and the *Distributor*, which furnishes the classical feature given by SCM systems. Once users get to know (via viewer) that a part of the system changed, if they are interested, they can ask for an update through the requestor. The requested part is then returned back by the distributor, which will take care[3] of updating the client's sources. It is worth to underline that the updated code is available despite the fact that it has been committed to the traditional SCM in use, if any. In fact, Syde is totally independent from any SCM repository.

Figure 2.3 shows a screenshot of Syde "in action". Changes are chronologically notified to team members, with the indication of who performed it, at what time and of course on which entity. Through this view, it is also possible to request for updates.

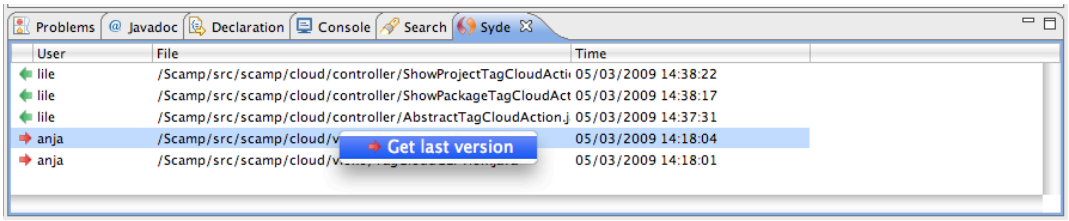

Figure 2.3: Syde

[3]A conflict manager has not yet been implemented.

For further information about Syde's concept and implementation we recommend to read [HL09b].

2.3.3 Validation & Results

A first Syde prototype has been used to get an insight into programmers' behavior while developing. Two (single-developer) projects with different characteristics have been monitored: one (X-Porter) was in development phase, while the second one (Syde itself) was under maintenance (bug fixing). Changes made to the projects, categorized into "successful compilation" and "unsuccessful compilation", were compared (see Figure 2.4). An analysis of the results shown that during a development phase the ratio of unsuccessful compilations is higher than in a project being maintained [HL09a]. This first investigation has mainly been made as a proof-of-concept.

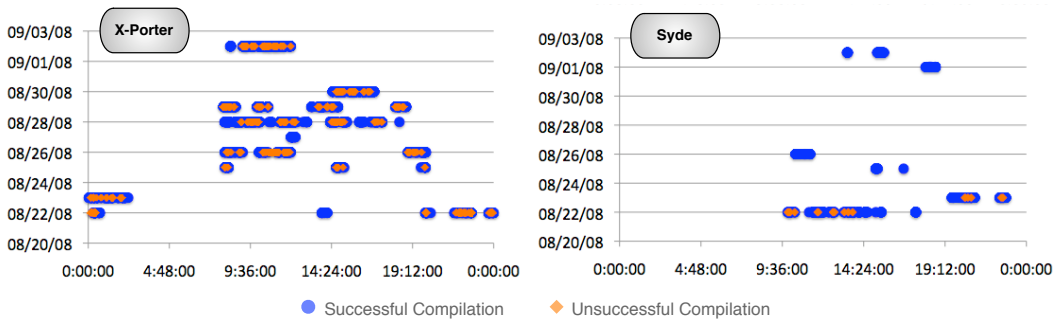

Figure 2.4: Recorded changes, categorized by successfulness of compilation.

Further analysis on data collected by Syde have been reported in [HL09b]. Several teams have been using Syde while developing their projects. Of particular interest in this analysis are two components of an industrial project, in a multi-developer context. Developers' activities have been monitored through Syde (which was embedded into the production environment of the relative software factory) for a period of 15 days, collecting around 2,500 changes overall.

- As first analysis, changes captured by Syde's have been compared to check-ins into a CVS repository, also in use by the team. The number of check-ins was 187 (7.5% with respect to the number of changes registered by Syde).

- A second analysis on the data, reveals a number of potential merging conflicts. Two changes on the same entity were marked as potentially conflicting if they where made by two different developers and within a 2 hours

span. The comparison between changes recorded by Syde and CVS commits, reveal that the developers tend to not commit changes to CVS after a development session. The longer a changes remains uncommitted, the more the chance of merging conflicts arises.

Unfortunately among the team of four developers, only two at a time where working on the project, due to time constraints and deadlines for other projects they were involved. Nevertheless, the previous annotations on the granularity of information and on detection of possibile conflicts, can already give a good idea of the validity of Syde as environment for synchronous development. In fact, using Syde, the team members are constantly up-to-date on what is going on. Conflicts can be therefore preempted or at least spotted early on (instead of being discovered days or weeks later, at check-in time).

In the same paper, Hattori indicates how it is possible to establish code ownership through Syde's information. For a first definition of *code owner*, we refer to Gîrba *et al.* [GKSD05]: *"Based on the number of lines of code added and removed extracted from the CVS log, the owner of a file is the one who owns the greater percentage of lines over the total number of lines of a file. In this case, the total number of lines of a file is approximated with information extracted from CVS."*. In order to apply the metrics by Gîrba, it is important that the frequency of the commits by team members is about the same. However, developers tend to commit changes to CVS in different ways (some performs frequent commits, while others wait longer before committing). If we, instead, use the historical data provided by Syde, it is possible to compute the above metric with higher precision, considering every change made on the artifact. Code ownership can then be redefined as: *"The owner of a file f, own_f, is the developer who has performed the greater number of small changes c on it. A developer becomes the owner of a file at the moment he performs $c+1$ changes in relation to the previous owner."*. Validation of this new definition has been done by applying both metrics to the same project, an then comparing the differences among the results. The delta between the two approaches is quite significant: following Gîrba's method, a considerable number of files (31 out of 50) remain without owner, because they were never committed within the considered time. Quite remarkable is that changes were performed on many of those, allowing Hattori's method to attribute them an owner. Additionally, around 20% of the files got assigned a different owner, when analyzing Syde's information. These considerations stress the fact that developers' behaviour (specifically the frequency of changes with respect to commits) must be considered when measuring code ownership.

Table 2.1 shows the experiment's numerical results. We remind that for this particular project under analysis, there were 187 CVS commits, while Syde

recorded 2,429 changes (over 15 days).

technique	source	classified files	user A	user B	user C	user D
Gîrba's	CVS commits	19%	53%	0%	37%	10%
Hattori's	Syde changes	100%	68%	18%	12%	2%

Table 2.1: Comparison between ownership assigned by different techniques.

Syde is still a prototype, thus it comes along with a number of limitations, which we briefly discuss. First of all, in order to collect and broadcast changes, clients must be online and connected to Syde (an auto-connect feature is available). In case the user is offline, changes will be stored in a buffer and uploaded to the server when possible. However, this feature was not yet implemented during the validation period presented above, in the course of which some developer reported to have forgotten to use Syde a few times.
Another aspect that has to be considered, is that Syde's efficacy has so far been tested only on a small set of project, most of them with a single developer.

As a conclusion on Syde, few possible extensions are already planned:

- increase the **granularity** of changes: at the moment Syde records changes at file level, but the scenario is to be able to capture changes with more details (for example, at metod level);
- implement a **conflict manager**, which would take care of occurring conflicts when updating entities;
- record the files that a developer browse during a development session (for example in order to define **dependencies between files**);
- implement an **instant messaging** service, which would allow team members to communicate without the need of additional tools external to the environment;
- following the previous point, a support for the **broadcast for help** could be introduced. Given the data at disposal of Syde and its technique to detect code ownership, the request for help could be automatically sent only to team members with the requested specific knowledge;
- implement **more views**, availing of visual metaphors, with the goal to increase developers interaction.

2.4 Thesis Motivation

We find Syde a valid tool to support collaborative development and we aim to enhance its environment with visualizations. We developed Scamp, a tool on top of Syde, which visually informs users about the changes happening in the system. The project aims to improve the quality of development in multi-developers projects, in particular by increasing the level of collaboration awareness of their developers.

Chapter 3

Scamp

3.1 Data Visualization

(A good) Visualization is a way of showing data, so that it is more accessible and easily understood. Visualizations are world-widely used as a mean of communication (maps, graphs, road signs, pictures,..).

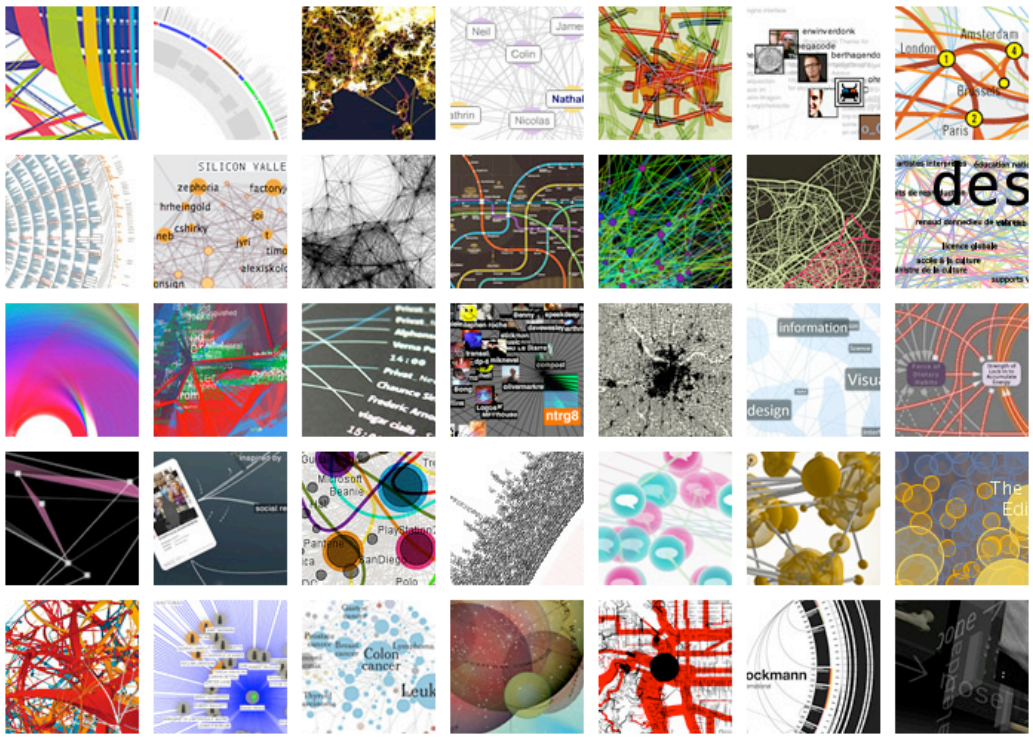

Figure 3.1: Visualization potpourri.

Visual perception, with around 70% of the body's sense receptors dedicated to it, is the most developed and efficient of our senses, making *vision* the most powerful and efficient channel for transmitting information. Following some perception-based rules, we can display our data so that important and informative patterns stand out. It is therefore very useful to understand how we can present data in the most efficient way: following some perception-based rules, we can display our data so that important and informative patterns stand out.

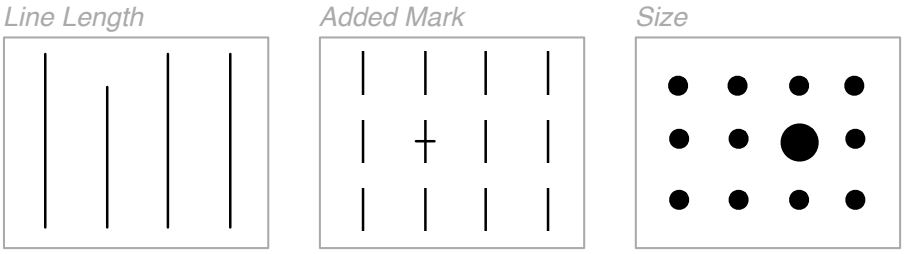

Figure 3.2: Example of preattentive attributes of visual perception.

Our brain processes visual information with three distinct memories: *iconic*, *short-term* and *long-term*. Iconic memory can be considered as a "buffer", which holds information for a very short period of time (less than a second). The cognitive process that happens at this stage, called *preattentive processing*, is automatic and unconscious. Only visual data that "looks interesting" at this early stage is considered for further processing. There is a only limited number of visual attributes that can be detected preattentively (such as those in Figure 3.2 or *color*).These preattentive attributes is what we can exploit to present our data in the most effective way.

> *Visualization is any technique for creating images, diagrams, or animations to communicate a message. Visualization through visual imagery has been an effective way to communicate both abstract and concrete ideas since the dawn of man. - From Wikipedia [Wik09]*

Visualizations are particularly useful in a context where there is a lot of information, because they can scale on both the number and type of information: the data to visualize can potentially be a huge set, and can even be abstract. Visualization can show data in particular and organized ways, which allows to stress relevant facts, thus facilitates considerations about the data.

We believe that visualization is a good means in the context of collaboration awareness, because it can offer information easily interpreted by developers.

3.2 Scamp Plug-in

As described in Section 2.3, Syde's notifier broadcasts changes to clients, along with relevant information. We use the same data as information source to visually inform developers about changes in a Java project, *as they happen*. Scamp's main functionality (collaboration awareness) is supported by the tool's reliability and usability. Moreover, once developers are aware of changes, they can decide what action to take. They can access the local code easily and request an update from the server directly through Scamp (for this feature, we avail ourselves of Syde's underlying structure), or they can use other means. Scamp, as Syde, is thought to be complementary to SCM systems, therefore developers could also choose to commit/update their code. Another scenario is that developers choose to get in touch with each other (i.e. via phone call or instant messaging) to communicate directly.

Scamp analyzes one project at a time. A project is seen as a collection of units. A unit is any file under the project directory (java file, xml, text,..). Changes received from Syde are bound to the corresponding unit and to the developer who made it. Since Scamp visualizes *all* the changed units, it will show also those that are not yet in the user's local working copy of the project. Scamp provides three kinds of visualization: a **Tag Cloud**, a **Buckets** one and a **Decoration** on files. The first two are available directly on the plug-in view, while the decoration is visible on the files in the default Eclipse's *Package Explorer* view.

> *The visual system has its own rules. We can easily see patterns presented in certain ways, but if they are presented in other ways, they become invisible. - Colin Ware*

The visualizations we propose are multivariate and multiscale, which means that they embed all of the data received from Syde: file, author, and time for any change (we recall that only successfully compiled changes get notified to the clients). We decided to provide three different perspectives, each of them made to inform about different aspects. The Tag Cloud view focuses more on the temporal aspect by highlighting the most recent changed units, the "bucket" one aims to remember to the developers how much others are working in the same project/unit by showing all the changes, while the decoration points out every file that has changed by adding to its icon a distinctive mark and indications about the last change made. Scamp's visualizations show the last changes happened during a given period of time, which can correspond to a day, a week

or a month. This allow both to have different overall views on how the system evolves and to give some flexibility for different kinds of project or phase they are in (for example when the team is in an intensive development phase, a span time of one day is preferred, while if the project gets changed now and then, a weekly or monthly history of change suits best). Currently this time span corresponds to the most recent one (i.e. "today"): the possibility to choose the starting date has not been implemented yet.

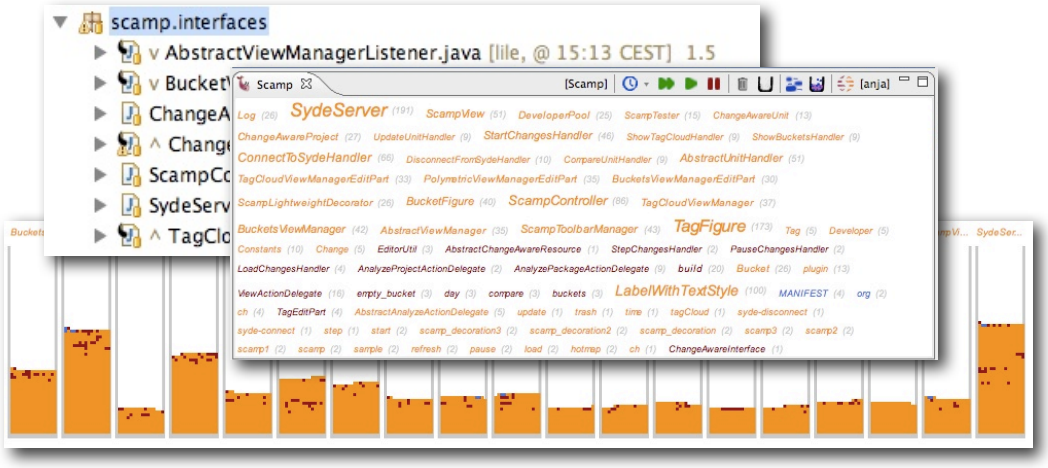

Figure 3.3: Scamp's visualizations in a nutshell.

Since we want to focus on helping developers to be aware of others in their context, we assign each developer a different **color** (which remains consistent between the different views). The color attribute in a visualization has a fundamental role as it can be applied to almost any visual element. In fact, color can be easily assigned to text, figures, lines, etc., while, other attributes may be more difficult to assign (think of assigning a shape to a textual element). The disadvantage of using colors is that there is only a limited amount of them (around 12) that are distinct enough to use simultaneously; additionally, the human eye can process only about 5 of them at a time. This drawback might be an issue in projects with a large number of developers. Moreover a small percentage of people suffer from colorblindness. We provide support for that by indicating the name of the developer either explicitly (in the decoration) or in a tooltip.

Another aspect of Scamp's visualization approach that enforces awareness is that views are automatically updated, causing the visualization to *move* if any change happened. We exploit this preattentive attribute (*motion*) to make the new information stand out in the visualization. The motion is really fast and it is unique for any change (i.e. it does not blink nor have a transitional

time), however it is enough for the iconic memory (which is the early and subconscious stage of visual perception) to process the visual information. Being automatically updated, Scamp views actually **tell a story** about the system being developed and this, other than inform, should keep developers "curious" (so that they will check now and then what is going on).

As we will describe more in Section 3.3, Scamp visualizations do not show a complete model of the project. Only units with the most recent changes are visible (with the possibility of changing the period of time visualized). This makes the relevant information more clear, since there is no extra information, and also increases performance. Moreover, since we do not model the whole project, Scamp should **scale up** to industrial-size projects. As mentioned, Scamp only shows the most recently changed unit, currently giving the option to choose between three different time spans: a day, a week or a month. We made this choice because, for the purpose of collaboration awareness, we want to inform the developer of changes *as they happen*. Moreover information about old changes ages with time and becomes less useful as the system evolves. Therefore the oldest changes are not relevant for our objective.

During the development of Scamp we took benefit from Hattori's collaboration. Her kind assistance gave us the opportunity to fully exploit Syde's change-centric model. Another great advantage of the direct cooperation with Syde's developer has been the possibility to request additional services, such as the possibility to obtain past changes by giving a period's start and ending dates (which was not originally available). Moreover, the usability of Scamp was improved by moving the implementation of the attribution of colors to developers at Syde server side. In fact, this guarantees consistency among different Scamp's instances: all the team members will see the same color representing the same person.

Even though Scamp is conceived to be used during the development process, it can also be employed in a research context. Analysis on the visualization can be valuable in terms of software engineering. For example the Tag Cloud can indicate where the current developing effort is located, while the "bucket" view can be useful to get an insight about code ownership. We will explain these and other phenomena more in the next section (3.3) and propose some concrete example in Section 4, where we validate our tool. Scamp can also be used in an educational environment, where professors can "check" the status of the development of students' projects.With the use of Scamp, it is possibile to follow the development effort (that could be measured in number and frequency of changes) of students, day by day. With regard of code ownership, Scamp would allow to easily identify who contributed in which part of the project.

3.2.1 "User Manual"

In this Section we will present and show Scamp's interface and how to interact with the plug-in. (To start off, the plug-in needs to be in the plugin folder of Eclipse, along with Syde and GEF, a graphical framework.)

To analyze a project with Scamp, it must be in the list of available projects in the Eclipse workspace. When right clicking a project, in the menu there will be an "Monitor project" entry (see **A** in Figure 3.4). By selecting "Analyze Project", the plug-in view will eventually open (only one project can be inspected at the same time). As the view opens, there will be indicated which project is currently under analysis: in this example the Scamp plugin itself (see Figure 3.5). Scamp allows to "focus" on a package too; in that case, the package path will be shown. By clicking on the Syde icon to the right side, you will be able to connect to Syde to submit and receive information. Users will be asked to enter their Syde user name in order to connect (if they were previously connected to Syde, the user name used there will be remembered). Once connected, the view will be filled with the changes of the past week (in the Tag Cloud visualization). This might take 5-6 seconds, depending on the number of changes performed on the system.

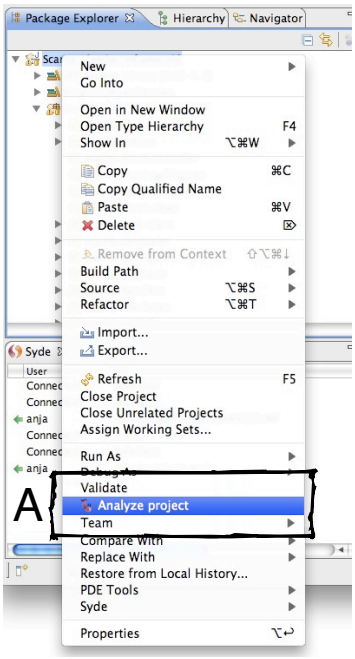

Figure 3.4: Eclipse menu.

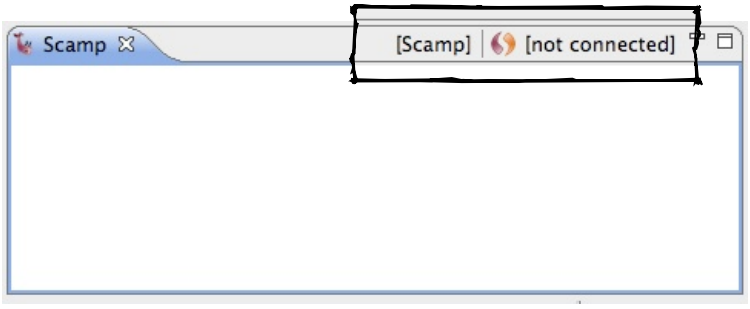

Figure 3.5: Scamp initial empty view, with zoom on the toolbar.

When a user is connected the Syde's toolbar will look like in Figure 3.6. From the toolbar it is possible to load the past changes (last day, week or month), to start/pause receiving new changes, to clear all the changes so far and to switch between the two available visualizations. Moreover, it is possible to disconnect from Syde.

Figure 3.6: Scamp toolbar.

Toolbar's icons are thought to be symbolic and easy to understand. For the sake of unambiguity, a tooltip indicating their function will appear on mouse over. As can be seen in Figure 3.5 and 3.6, the toolbar is divided into chunks, two of which are always visible: the first item, at the start, displays the name of the project or package currently being monitored, while the part at the end, related to Syde, has a connect/disconnect action and a label with the developer's username (if logged in). Once logged into Syde, more actions are visible: those relative to changes, the ones to empty the views and those to trigger the different visualizations. Following, there is an explanation of each one of the actions directly available from the toolbar.

This action is actually a menu, which lets the users set the changes **time window**, that is the period of time of the visible changes. This can be currently set to **1 day**, **1 week** or **1 month**. Setting the time window will cause the last changes to be loaded and displayed.

Load the past changes, according to the time window's span of time. The result is displayed only after the loading.

Play / **Pause** for the changes. Clicking play will let Scamp receive new changes and display them as they will arrive. Pause will stop that. Pressing Play after having stopped, will cause all the past "queued" changes to "flow".

Clear all the changes (trash bin icon) & **empty buckets**. Those buttons let the users delete all the past changes or only those into the buckets visualization. Although changes can be easily recovered by clicking the loading button or by changing the time window, the user will be asked for a confirmation before removing the changes.

TagCloud visualization, where all the changes are seen as a Tag Cloud, and **Bucket visualization**, where units are seen as buckets, and changes are seen as little squares that fill them. The views will be described in Section 3.3.

The **Syde** icon allows to disconnect from (respectively connect to) Syde. If the user is already logged in, the name he in using to commit changes will be shown in square brackets.

Logging to Syde through the right most button in the toolbar is the only operation that the user has to perform in order to "activate" Scamp. Once connected, first the last changes will be loaded, and then the tool will automatically start receiving changes. To interact with the views, users can pause and restart (play) the flow of changes. This is particularly useful to see what happens if the user needs to go away from the keyboard for a short period of time (i.e. stop when we leave, and play as we get back, to see what happened). If during the period of absence, the other team members performed a large number of changes, we suggest to use the load function, which will recover the changes all at once. If the user logs out during the development session, Scamp's views will be still available to see, but they will not be updated until he logs in back.

Scamp also provides the possibility to request for the update of a changed file. Before deciding to do so, users can choose to compare their local copy with the last version on the Syde server (a proficient file comparator is not yet implemented). By right clicking a changed file (distinguishable by the decoration) in the Eclipse's *Package Explorer* view, a "Scamp" entry will be available in the popup menu, making the compare and update options available. The menu entry will appear only on those units that have changes.

3.3 Visualizations

3.3.1 TagCloud View

A *tag cloud* is a list of tags (usually words), which are weighted, colored and sorted according to some metric (not necessarily the same). The tag cloud is a visual technique that has born on the web (introduced the first time by the popular photo sharing website Flickr[1]) to describe the content of a website. Figure 3.7 shows a tag cloud about all the terms used in Scamp, where the size and

[1] http://www.flickr.com/

color of the tags are correlated to the frequency of the word in the tag. The cloud is generated by TAGete (a plug-in by an UROP student).

Figure 3.7: A Tag Cloud on Scamp's vocabulary (by TAGete).

We transfer the concept of the tag cloud to Scamp, depicting the units on which at least a change has been performed.

Figure 3.8: A Tag Cloud by Scamp on Scamp itself.

In Scamp's tag cloud (see an example in Figure 3.8) each tag represents a different unit (i.e. a java class or a documentation file), in which size is proportional to the number of changes made to it: the larger the amount of changes, the bigger the tag is. Most recently changed units are shown at the beginning of the cloud (top left), while units that have not been changed recently will have their tag at the end (bottom right) of the cloud. The color of the tag is associated to the developer who made the last (most recent) change to the unit it represents.

Shortly after a change is made to a unit, it will be visible on the tag cloud, which gets constantly updated. We decided to keep tags sorted chronologically because in a collaborative environment we are interested in the parts of a software system that get modified. The most frequently changed entities will be at the beginning of the cloud, which is the first spot where most people's eye will naturally look. In this way the users can always know with a glimpse which are the files being modified at the moment. New changes are noticeable because the tag corresponding to the modified unit will "move" at the beginning of the tag cloud and the color will change to the color of the developer that has made the change. In this way the user will be aware that something happened in the system and he will know what changed and by whom. Additionally, it is possible (thanks to our iconic memory) to remember where the tag was previously located and its old color, which can give additional information about what is going on. Moreover, the size of the tag, which gradually increase as the unit evolves, aims to give an insight about the kind of unit and its importance in that development phase. For example, it is likely that central or problematic components of a system will have a higher number of changes, thus a bigger tag. When a new entity is created by a developer, the other team members will see the corresponding tag displayed at the beginning of their tag cloud, as with changes on units already in their local working copy of the project. To obtain the source of the new entity, the team will need to update their local copies from the repositories, once it gets committed.

The tag claud view is the default view in Scamp, because it is the one that is more focused on collaboration awareness. The fact that it changes when there are new changes, gives the idea that the system is *alive*. We calibrated the tool, so that the visualization is refreshed at an appropriate frequency: a too rapid refresh would be disturbing, while a too slow one would fail in its purpose of showing changes in (close to) real-time.

The benefits of this view are multiple. The information given by the sorting of tags is precious when team members work on the same files: if the particular entity that a developer wants to modify is among the top ones, he might want to check out the new version of the file before editing it, respectively to wait for the other to be done working on that entity. On the other hand, if the file is not among the most recently modified ones, developers should feel more free to modify it, without fearing concurrent development.

Moreover, a developer can detect a possible conflict by noticing that the color of a tag has changed from "his" color to another one in a short time, meaning that someone else changed the entity probably without being aware of his code modification. Team members do not have to constantly look at the view, because

anyway their mind will spot new changes and, only if it is relevant, process further the information. Another phenomenon that could be detected by looking at Scamp's Tag Cloud view is the refactoring involving two different files. In this case, we will see that the two files repeatedly alternate each other in taking the spot at the beginning of the cloud. Detecting such facts is useful to developers, because they become aware of these particular situations as they happen.

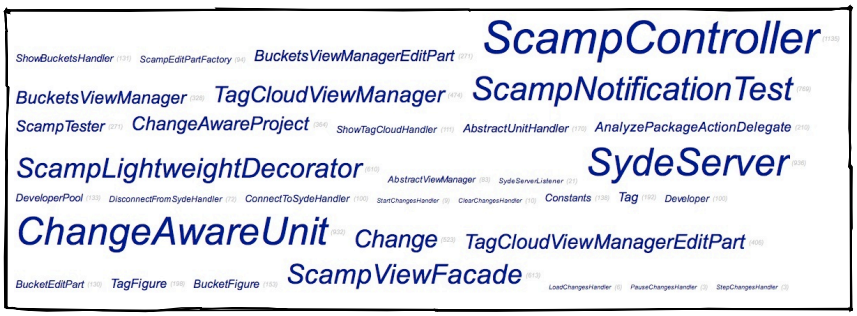

Figure 3.9: Another example of Tag Cloud on Scamp itself.

Interesting files, namely those most recently modified and the ones with most changes (during the displayed period of time) can not only be detected more easily thanks to the visualization, but they can also be accessed in a faster way. In fact, by clicking on a tag, the corresponding unit will be opened in the default editor (if it is already in the user's local copy). It is also easy, with the colors, to recognize and select only those files that have been last touched by a particular developer.

With regard to evolution analysis, observing the evolving of the tag cloud during the development of the system can be particularly interesting. Depending on the selected time span, different conclusion can be drawn about the ongoing development by analyzing the tag cloud as a whole. If at the end of the day the general picture (in term of size of tags) is similar for any time span, it can mean that the development is homogeneous (entities are modified with the same frequency through the development). If the scene of the modifications drastically differ from a time span to the other, it can mean that the system is in a maintenance phase: in a bug fixing phase it is possible that many un-related files are modified. Additionally, considering a time span suitable to the project's "development style", size and color of the tags can be revealing too. God Classes could be spotted by the large and highly frequent number of changes made to them, likely from many developers, whereas "activity" of developers can be spotted by looking at tags' colors.

We investigate possible patterns in Section 4, where we validate our tool.

3.3.2 Buckets View

In the previous section (Section 3.3.1) we described in details the Scamp's Tag Cloud view and we enumerated the various advantages it has. Although the Tag Cloud is great for drawing the attention of team members on collaboration, it looses information about changes made in the near past. While it holds historical facts in the size of the tag and in the chronological order of units' last change, information about their authors is lost. We developed a second visualization, that we call *buckets*, which retains knowledge about who contributed to the entities.

The visualization's name comes from the fact that units are seen as "buckets". Those containers get "filled" by little squares, each of which represents one single change. The color of each change maps to the developer that did it. As the plug-in gets activated and the user connects to Syde, the buckets in the view will be filled with the past changes (if any). Oldest changes will be at the bottom of the bucket, newest changes will *appear* at the top of all the other changes already present (if any) in the bucket. Changes will remain there as long as they happened in the given time span. The result, as can be see in Figure 3.11, is a sequence of buckets, which differ from each other at different extents in the number and color of contained items. Each bucket will have at the top a label with its corresponding unit's name, which color changes according to the developer who contributed most to that unit (and who, following Hattori's definition, which says that the owner of a file *is the developer who has performed the greater number of small changes c on it*, is thus the unit's current owner).

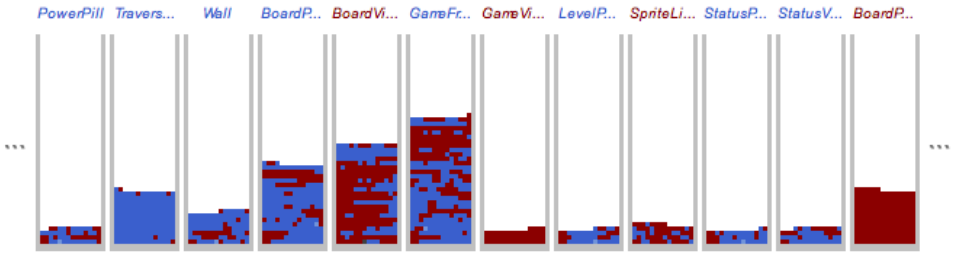

Figure 3.10: Buckets showing a month of activities in a two-developers project.

This kind of visualization brings along some considerable scalability issues, which needed to be tackled both from a conceptual and a practical point of view. In fact we have a limited space at disposal, where only a finite (small) number

of buckets can be always visible, while the systems to be monitored and analyzed are potentially very large. Although there is the possibility to horizontally scroll the view and see all the buckets, we find impractical to actually browse a very large number of entities. A solution is to sort the buckets, so that the most relevant ones are visible. There are various metrics that can be adopted to sort buckets: following chronological order (according to the date and time of the last change), by number of changes (most changed units first), by frequency of changes (weighting differently old and new changes), alphabetically, and others. However, the fact that the visualization gets updated as there are new changes, led us to choose the alphabetical sorting. We noticed that by sorting them according to a metric that frequently changes (such as chronologically), results in distracting the user and interfering with their observation: the view keeps on "readjusting" and it becomes difficult to follow the units' evolution. When applying chronological sorting the buckets keep on changing order and when sorting by number of changes we frequently noticed cases in which two or more buckets were keeping on swapping between each other; with the result that in both cases the visualization lose its value. Another solution, the one currently in use, is to compromise between the number of displayed unit and the way they are sorted. To decide how many units to display, we applied the *Pareto principle*, which which states that a low percentage of variables causes a high percentage of effects. This principle can generally be observed in any kind of large complex system (population&wealth, traffic&roads, etc.) and is widely exploited to decide how and where distribute resources. We approximate that 80% of a changes involves 20% of the units, therefore the Buckets view will show only the 20% most changed units (or a minimum of 15 units). We alphabetically sort those selected buckets, so that their position will be kept more or less constant (depending on the set of most modified unit) and we chose to sort them according to the unit's file path, so that units will also be conceptually sorted (units in the same package will be near each other). Although this choices are good enough for our current version of the prototype, we plan to investigate more these aspects. Finding other ways to sort the buckets and other percentages to choose them, might lead to a more relevant visualization.

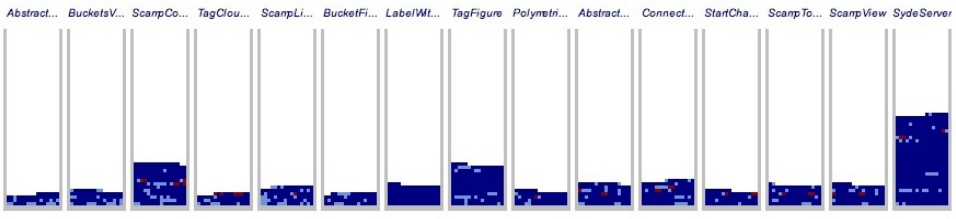

Figure 3.11: Buckets view "in action" on Scamp itself.

The focus of this visualization is mainly to see how each developer in the team is contributing to the project and to keep into developers' mind that they might not be the only person in the team working on some component. For this reason, information about the most relevant buckets is directly embedded into the view (i.e. only those are visible), and chronological order is maintained both by the appearing of new changes and by the order in which those are displayed. This gives the possibility to exploit visualization features (i.e. color) to present other aspects in an effective way.

Since we want to "drive" the user's attention to other aspects, such as patterns in the colors of the changes, the *motion* component has been given less importance in this view than in the Tag Cloud. Changes that "appear" will still be noticed by the user's iconic memory, giving the indication that the unit is evolving, however the attention that they capture is "low".

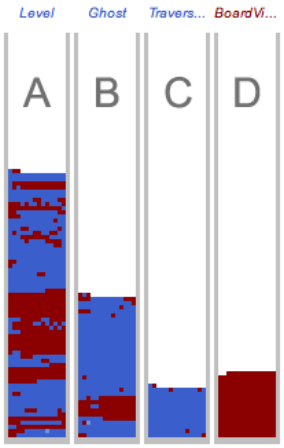

Figure 3.12: Example of patterns in buckets.

Figure 3.12 shows an example of buckets from a real-case two-developers project. Changes in the buckets come from the last month of development, which roughly correspond to the duration of the whole project. We can see that in the most noticeable element when watching a single bucket is its changes' colors, while if we watch more than a bucket at a time, we also notice (and compare) their different state of fullness. In the particular example is possibile to catch in a glimpse that the bucket labeled **A** has twice, respectively four time, as much changes than the other three. Once noticed this fact, the user attention is driven to this particular entity, and the focus moves to the colors and their pattern(s).

We can recognize very different pattern in the colors of the buckets:

- Bucket **A** has been co-developed by both developers, more or less in alternating each other. We can spot that in the middle of the unit's development there has been a phase in which it has mainly been modified by only one of the developers (the one with color *red*). We can see that during this middle phase, which we assume to be continuous, there are a few changes by the other developer (i.e. color *blue*): this could possibly have led to conflicts.

- The second bucket, **B**, has initially been developed by both team members. We can see two distinct blocs at the bottom of the bucket, where the two developers have been alternating each other: first *blue* and then *red*. After

this first phase, the unit has been modified by developer *blue*, with very few changes from the other team member, most of which are at the end of the development (maybe for some bug fixes).

- **C** has mainly been developed by one developer (*blue*), however its development partner (*red*) did some changes at the start and at the end.

- Unit **D** has entirely been developed by one developer: *red*.

We saw, when singularly analyzing each bucket, that **B** and **C** have both been mainly modified by *blue* with a few changes by *red* and noticed that the majority of those changes can be found at the end of their development. We can interpret this in a number of ways: maybe at the end of the development the two developers had sessions of pair-programming, where *red* was working thus committing changes, or perhaps *red* has been responsible for checking and/or adding documentation's comments to units. From the example in Figure 3.12, we can also see how *code ownership* can be spotted with a quick-look. While for bucket **A** it can be difficult to determine which author contributed most, we clearly see that buckets **B** and **C** are owned by *blue*, whereas **C** by *red*. By checking the color of units' name at the top of each bucket, we can see that the first three have a blue label, while the last one is written in red, therefore *blue* can be considered the owner of **A**. We find, anyway, that the labels' color has only a complementary value when determining code ownership (or expertise) and that looking at the buckets is important. Bucket **A** is a good example showing that, while the unit's ownership is assigned to *blue*, both developers can be considered *experts* about it. Concluding, the Buckets view can help in determining to whom ask questions about entities in a system. In Section 4, where we discuss the validity our tool, we will present more bucket's patterns, with relative conclusions and comments.

When checking the Buckets view, users should always keep in mind that a little square (i.e. a single change) is added each time some code is either *modified*, *added* or *removed* to the unit. It is therefore quite important to remember that the level of changes in a bucket is not directly proportional to the "length" of the entities, for example measured in number of lines of code. The Buckets view aims to give the idea that entities are evolving, which is represented by the growing amount of changes. We find that our approach can give a better insight into the effort taken to develop an unit, with respect to the information given by comparing two unit's versions through CVS or Subversion. For example, the case of a removal of code is considered as any other change, while in a SCM it mostly result in a negative difference in the number of file's lines, which might give the false impression of being necessarily something unwanted. The

same example is valid in the more general case of code modification of bug fixing (the result of comparing the two versions from a SCM repository shows the number of lines added and removed, flattening the real number of changes the unit undergo since the last snapshot and reducing the user's perception about the file's evolution). Exploiting this, Scamp provides the possibility to *empty* the buckets. A user can empty the buckets to follow the evolution of the system from that point in time on. If needed, past changes can be recovered at any time.

While tags in the Tag Cloud has a tooltip indicating the name of the last developer that modified it, in the Buckets the name of the developers are not reported anywhere (we are still investigating how to do it in a non-disruptive way). A third view visualizing developers, currently under development, will be integrated in the plug-in to overcome this issue (see Section 3.3.4). For now, we propose to switch to the Tag Cloud view to check out which color is assigned to which developer (this method should work fine in a project with up to five developers, because our memory can memorize around that number of developer-color pairs and retain the information long enough to switch back to the Buckets view and transfer it to the visualization colors).

As final remark: each bucket has a tooltip, which indicates the full path of the corresponding unit and the exact number of changes it underwent during the visualized time span.

3.3.3 Decoration

The Scamp plug-in, other than the two already mentioned views, also provides a **decoration** (little annotations on the Package Explorer) for the files in the project. Project's files that changed will be marked, so that users will know that "something is going on". Scamp's Decoration is composed of three main elements: an *overlaid icon*, an *arrow* and an *annotation*, respectly signed as **A**, **B** and **C** in Figure 3.13.

Figure 3.13: Decoration added by Scamp to modified file.

Additionally, a fourth element (an annotation with "[Scamp Eyes]"), is added after the name of the project (and, if any, package) that is being monitored.

A - A black icon is overlaid to the icon of the files (i.e. units) which have changed. The icon is a slim black stylized Scamp's logo, which is very visible and integrates well with the Package Explorer view. We decided to put this element of decoration in the top left corner of a file's icon, trying to avoid conflicts with decorations added by other plug-ins. As an example, CVS and Subversion add their decorations to the right of the icon (respectively at the top and at the bottom).

The goal of this decoration is to mark the changed units, so that they are easily identified.

B - The second element of Scamp's decoration is an arrow added right after the icon, in front of the name of the file itself. The arrow is going up ($\wedge$), if the changed has been changed by the user himself, or down ($\vee$), if the last changing the unit is another developer.

The arrow concisely indicates who made the last change, distinguishing by the user of Scamp and the other developers of the team. Moreover it reinforces the visibility of the changed units by shifting their name to the right, causing misalignment in the list of files, which makes them "stand out".

C - If "someone else" changed a file/unit as last, an annotation will be visible after the name of the file. This decoration contains the indication of who changed it (i.e. Syde username) and when; showing only the time, omitting the date, if the change has been made in the same day (i.e. "today").

The annotation aims to give complementary interesting information in an unobtrusive way.

The three decoration's components have different goals, yet they coexist and form an homogeneous Scamp's feature, which informs once more the user that the system is changing. The decorations are added to files only as some change happens and can be cleared by deleting all the changes from the Scamp's toolbar. Decorations are not added when loading past changes, because this might result in marking the majority of files, radically reducing the value of the decoration.

The advantages of the decoration are multiple. First of all it takes little space and can be seen as long as the user is connected to Syde, even if the Scamp's view is not visible. Secondly it visually distinguish changed files from the others in the project (while the other views we provide do not contemplate non-modified

units). Third, once the unit is modified, the arrow and the annotation decorations will be automatically updated according to new changes. Additionally, it potentially scales up with any project, as long as it can be imported in eclipse.

Compared to the other views, the Decoration is much more lightweight. Users that find the visualizations too distracting can still benefit from Scamp by only looking at the decorations. Collaboration awareness won't be therefore entirely compromised, in the case that Scamp's main view is hidden.

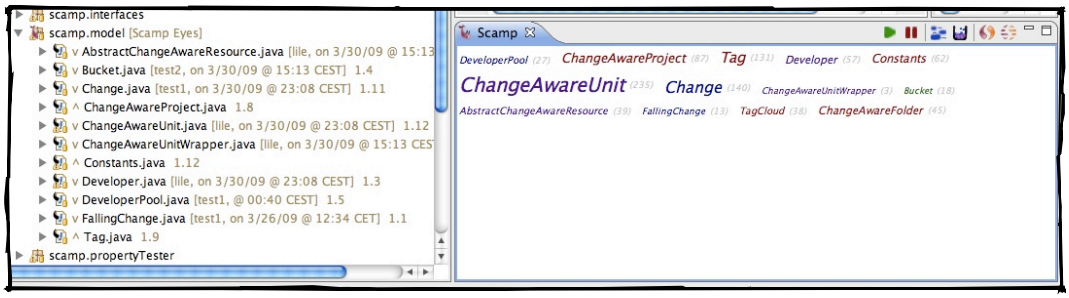

Figure 3.14: Decoration and Tag Cloud showing the same information (on Scamp).

Figure 3.14 shows the matching between the Decoration and the Tag Cloud views (example on the Scamp project itself). Users that for some reason are not interested in the main view, will still be informed through the decorations. Although changes might not be spotted as they happen, developers will eventually notice that a file has been modified by seeing the decoration at the moment they open a it for editing. If the file is open in the editor, the decoration will be visible also in the Eclipse's Outline view (see Figure 3.15).

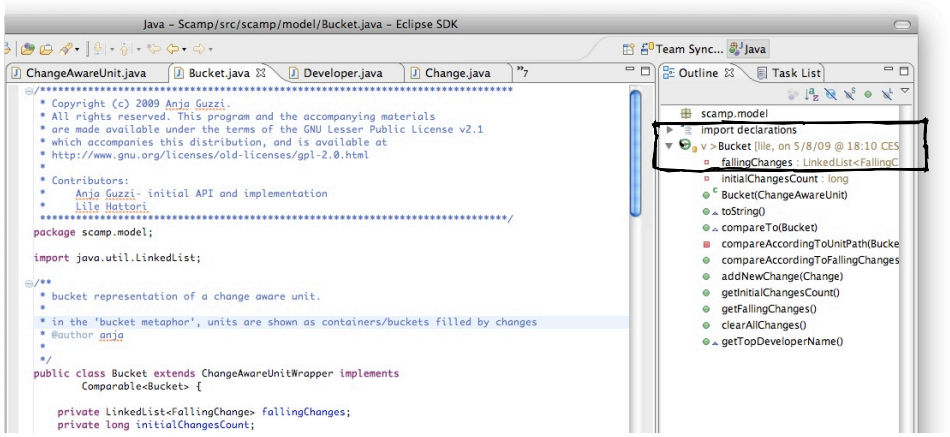

Figure 3.15: Decoration in Eclipse's Outline view.

3.3.4 Developers View

In this section we describe a third view, the *Developers* view, which is currently under development and therefore is not yet integrated part of the Scamp plug-in. The goal of this view is to visualize all the team members that took part in the project's development, giving information about their assigned colors and if they are currently connected to Syde. Moreover we plan to embed indications about how much they contributed. There are a few ideas about how to consistently display all the information in unique view:

- A possible idea is to "reuse" the concept of the **tag cloud**. Each tag would represent a developer and encode both his user name and color in its text, with size mapped to the number of changes he made (always in the given time span, which is common for all the views). The by looking at the tag cloud, users will spot either the color of the developer's name they are interested it and know the matching in a second. Tags can be separated in online and offline users and then sorted, for example, by number of changes made or alphabetically. Sorting them by number of changes can help the look up of both a color or developer's name, because users already have an insight of the amount of his contribution if they previously looked at the Buckets view.

- Another alternative is to implement a rich **buddy list**, like those of the well known IMs, where each entry in the list would correspond to a team member. The user name can be colored, or the developer's color can be otherwise added as an additional icon in each entry. This kind of view would be particularly useful in the case where the status of team members has a central role, because the status can either be displayed as additional icon/text in the entry, or can be mapped to the text's color (for example distinguish offline developers by writing their username in light gray). Moreover, if the status is mapped to an icon, there is the possibility to add other kinds of status (such as "idle", "developing" or "away"), which could be automatically detected by the plug-in and then broadcasted to the other team members. With regard to this, a **twitter**-like status message with additional information, such as date and time of the last change(s) made to the system, could also be automatically assigned by Scamp, informing others about the last few units modified by a developer and making this visualization a rich pool of information about developers. Team members could also have the possibility to keep others updated by adding themselves information about what features they are implementing or which bug they are trying to fix.

- Extending the previous point, a small **chat** could be developed to serve the purpose of visualizing developers, informing the user about who else of the team is logged on Syde and easing communication among team members. A chat would allow users to exchange messages between each other or broadcast a request for help.

- A different approach could be to visualize the developers and the or "correlations" among them. This "**Developer Pool**" view will show developers, maybe displayed as little colored sketched humas (or with a customizable avatar), more or less close to each other, depending on the relationship's strength. Connections between developers can be measured with the amount of units they developed together to a certain degree (a relevant metric should be conceived and tested). The resulting view is a graph-like visualization, which can reveal groups of developer. This can be useful for a user to limit the number of colors he has to remember, and for example to which he has to pay attention when watching the Tag Cloud.

Visualizing developers in a dedicated view has a dual value: at one side it assists the other two visualizations, by providing an explicit and clear matching between colors and people; while from another point of view, it is complementary, enhancing awareness by notifying which team member is online and working on the project at the moment.

Chapter 4

Validation

In this chapter we will present 3 real-case projects, that have been developed with the assistance of Scamp. Two of the projects are two are students' projects developed in pairs, while the third one is Scamp itself, which have mainly been developed by only one person, We will describe the users' experience, show screenshots and comment them in order to validate the ideas behind our prototype that we described in Chapter 3. Because of the nature of the data collected and visualized by Scamp, the validation will also have a strong evolution analysis component.

4.1 PF II Projects

Four USI student have been at disposal to use our prototype during the development of their programming fundamental class final semester project. The students have been given a tutorial about Scamp's basics (with informations very similar to those that can be found in Section 3.2.1). Their projects lasted for about 4-5 weeks, during which we were refining and testing Scamp for its first release. The students provided us with some intermediate feedback, reporting any problem they encountered.

At the end of their semester project we collected a final feedback from each one of the involved student, by means of a small questionary about their experience using Scamp.

4.1.1 Questionnaire

To validate our prototype, we are interested in knowing a few fact from its users. In particular we first wanted to check if it has been used and was appreciated and secondly if it served its purpose of enhancing collaboration awareness.

Because the targets of our "interview" were students, we decided to send them by e-mail a few (optional) questions, which they could answer with no pressure. Among the requests we made to get an insight into their experience with Scamp, we asked them to explain in few words what Scamp was, what is its use is and we invited them to report its positive aspects and what could be, otherwise, ameliorate.

4.1.2 PF II Project - Group 1

The first experience we collected is the one of Stefano and Thomas, which developed **jArk**. jArk is an Arkanoid/Breakout implementation in Java. After 5 weeks of development, jArk counts approximately 7,200 lines of code in 83 classes (82 files), divided among 12 packages and has been committed almost 300 times. The students have been using goggle code, which provide Subversion code hosting.

From Stefano's experience, Scamp "*lets the user see in real time other people changes in the working project*" and would be particularly useful in a context with a large number of developer working on the same software system. Stefano and Thomas have mostly been programming at the university, therefore they mainly spent time working in the same place and speaking to each other. However, Stefano reports that "*it happened that seeing a file was beeing modified, I didn't commit my changes but waited for the other.*". Thomas reported that Scamp was not so useful for the awareness purpose, since they almost always worked together, however we can understand from his answers that Scamp's view was almost always visible and active and that it was not distracting or disturbing their development.

> *I like the fact that you can actually see who did how much.*
> *- Stefano*

The experience with Scamp lived by jArk's developers was overall nice: even if the awareness information where not always useful in their context, they both liked the fact that with Scamp it is possibile to actually see how much each person contributed to the project and how much each file has been modified (by looking at the size of the tags). As suggestions for future works, both students report that it would be useful if Scamp would provide the possibly to automatically connect and the possibility to scroll through the Tag Cloud to see all the units. In fact, although at the moment Scamp remembers the user's login name, it does not automatically log. Another suggestion made is that "*it could be intresting to have a preview of the changes being made, instead of only the files.*".

A basic answer to the “request” is given by the simple comparator provided by the current Scamp prototype, however, at the moment of the tutorial to the students, this feature was not yet implemented. Nevertheless, to implement such feature in a cleaner and more efficient way, Scamp will need finer-grained data from Syde.

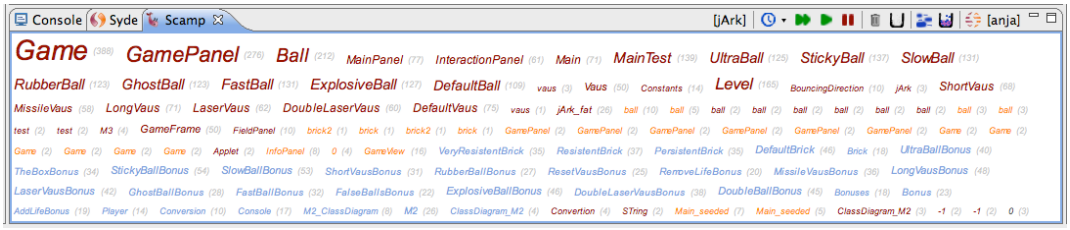

Figure 4.1: jArk’s Tag Cloud in April.

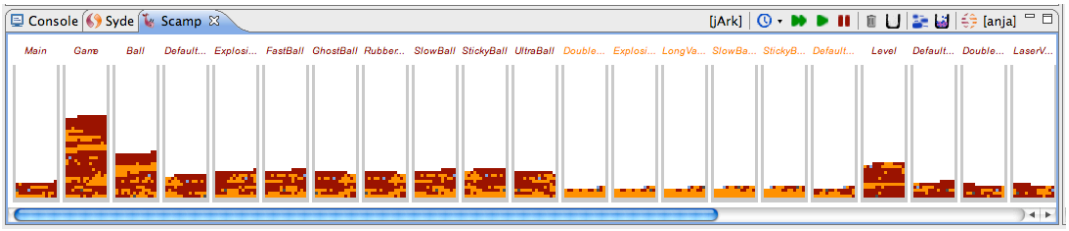

Figure 4.2: jArk’s Buckets in April.

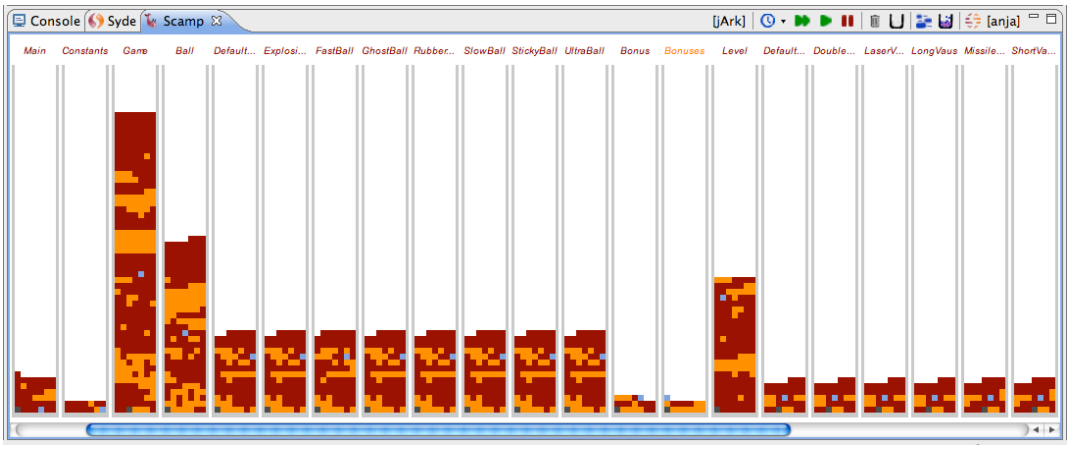

Figure 4.3: jArk’s Buckets in May.

Figure 4.1, 4.2 and 4.3 show screenshots of jArk's changes: the first two pictures are related to April, while the last one depicts changes done in May. From the Tag Cloud we can that *Game*, *GamePanel* and *Ball* tags are more noticeable than others. Knowing the game and talking to the developers we can confirm that. More interesting is the fact that from the Buckets we can see how files (i.e. classes) modeling the different kind of balls implemented in the game (i.e. *DefaultBall*, *ExplosiveBall*, *FastBall*, *GostBall*,..) have the same pattern of changes. Relate to that, a very similar pattern can also noticed in the changes of the *Ball* class, which is most probably a superclass of the above, even though it has approximately twice as much changes as its subclasses.

From the statistics about the project, we can also see that, for this project, Scamp's units correspond almost 1-to-1 to classes (jArk has 83 classes and 81 files), therefore observations about the visualized "units", applies to the program's Java classes.

4.1.3 PF II Project - Group 2

The second team that participated in Scamp's validation was composed by Mark and Luca (however, only Mark decided to answer our questions, therefore we will report here only his experience). They developed a Java version of Pacman in 5-6 weeks. The project counts 5 package, 59 classes, 382 metods, for a total of 3,978 lines of code and has been committed 170 times to Subversion.

Pacman has been mainly developed in pair-programming, however some parts of the project assigned to the one or the other student. They went through an initial experimental phase, where they gained confidence with the technology, and then they started from scratch some part of the project.

> *The useful part of this plugin was the possibility to see live if my other team mate was modifying a file I was working on too, so that I could ask him "What are you doing?".*
> *- Mark*

Mark particularly liked the possibility to see who did the last change on each file, while he found a bit disturbing the fact that any kind of file (in particular he was concerned about images) are shown by Scamp, because that "filled" the Tag Cloud view with tags not relevant to him. Moreover, he also suggested to provide an "auto connect" feature.

Figure 4.4 shows the Tag Cloud view by Scamp on Pacman. We find this view very interesting from an engineering point of view, in fact from the image can be spot a few classes: *Level*, *Board*. *Ghost*, *Pacman*, *Player*, *Creature*, *Controller*,.. we asked to the developers if those classes have a central role in the project and they confirmed, however Mark pointed out that (due to some refactoring) entities such as *GameFrame*, which can also be spotted in the cloud, do not exits anymore. As a remark: these "false positive" gets automatically eliminated through time, since those units do not get updated anymore, they will eventually exit the visualized time span.

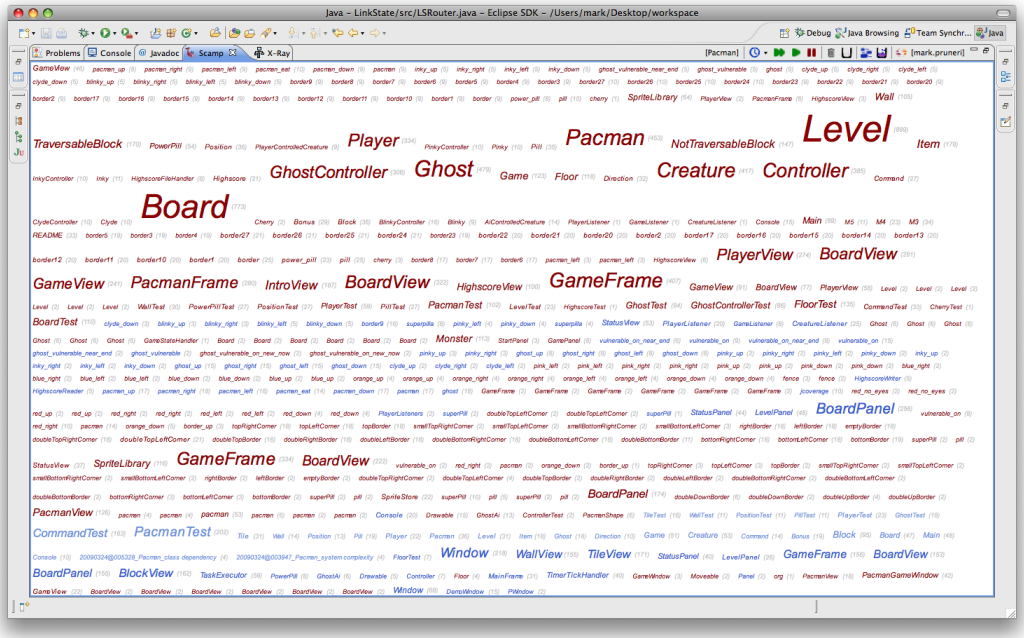

Figure 4.4: Pacman's Tag Cloud (last month).

Figure 4.5 and 4.6 show two snapshots of the corresponding Buckets view. From the Buckets, it is possible to spot some patterns. We can identify the units that have been mainly pair-programmed and those that have been entirely developed by only one of the two developers. Mark explained that classes relatives to the model and the testing classes have been pair-programmed, while other independent components (such as the view) have been assigned to one of the two developers (the view elements can be spotted in Figure 4.6, on the right side, because they all have the same color).

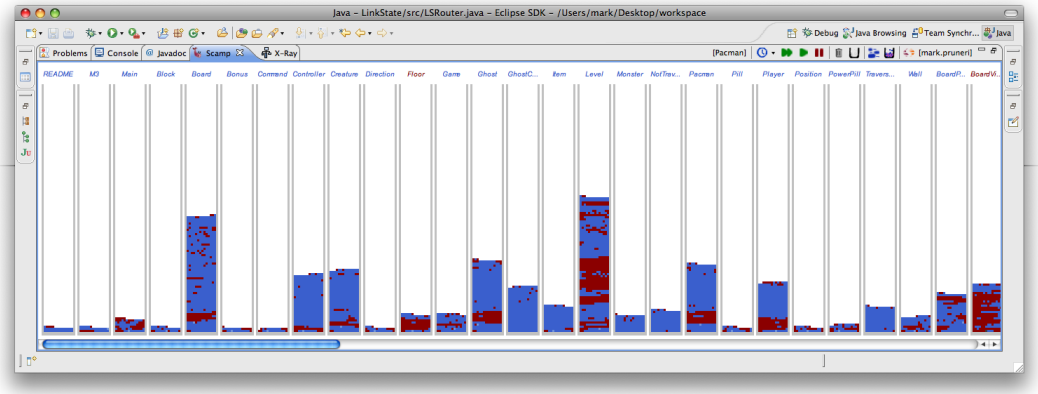

Figure 4.5: Pacman's Buckets (last month).

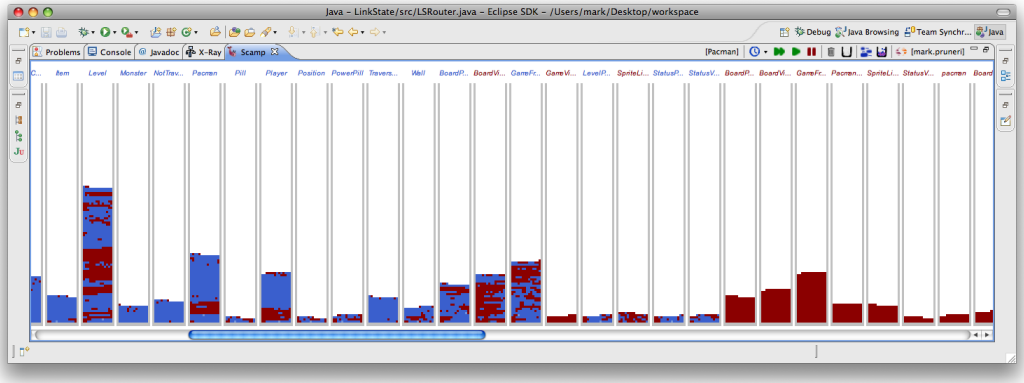

Figure 4.6: Pacman's Buckets (last month).

4.1.4 Conclusions on the PF II Projects experience

Two students' projects have been constantly monitored by Scamp through their development. Some patterns emerge from the resulting visualizations: the Tag Cloud enlighten the central classes of the project, while the Buckets reveals patterns between the contribution of developers. The Decoration has not been mentioned by any of the three students that answered our questionary (premising that we ourselves did not mention any of our visualizations in the questionary). The reasons behind this omission can be a few, the first of which is that the decoration needs in some cases to be triggered from Eclipse's preferences (thus, maybe the students did not had it visible at all). While, in the case the decorations were visible, it is possible that the Decoration is seen by the user as "an extra and additional feature", while identifying Scamp with its two main views.

In a context where developers work close to each other (i.e. in the case of pair-programming), the additional value added by Scamp to the project development process result minor with respect to its potential value in a project where developers do not interact with each other and works geographically distributed.

4.2 Scamp itself

Scamp has been developed over a period of about 3 months: during approximately half of the project, only one person was working on Scamp, while later in the development a second one joned, cooperating in some parts. The first weeks were focused on "exploring" APIs and in starting up with a first basic visualization, the following 4-6 weeks have been the core of the development, while the last month has mainly been dedicated to ameliorate the plug-in in order to have a final stable release.

We had the possibility to use Syde to capture all the changes we made to Scamp from project creation, which gave us the opportunity to test our plug-in "in action" with known data since the very beginning of its development. Having the possibility to test the plug-in visualizing our own ongoing development has been of great advantage: the developer knew exactly what she was doing and could check if the visualizations were matching her expectations. In this way, we could constantly test and calibrate our views as Scamp was taking form. Being ourselves the very first users of Scamp has for sure helped to make it better, improving its views, and to enhance its usability.

To overcome the fact that Scamp has mainly been developed by only one person, we decided to modify its code so that changes were resulting as coming from a pool of developers, so that we could roughly simulate a multi developer project. This "hack" has been useful throughout the development of Scamp and has been removed in the last phases of the project, when Hattori joined the development for some part of the plug-in and to test it. Besides, because we introduced the additional developers at Scamp side (and not by connecting to Syde with multiple different names), all the changes we made to Scamp have been faithfully recorded.

Figures 4.7, 4.8 and 4.9 shows Scamp's Buckets view for each one of the months during which Scamp has been developed, while Figures 4.11, 4.12 and 4.13 show the corresponding Tag Cloud views. We can see from the Buckets that during the first month (March), only one developer (color *blue*) has been developing, as at the start of April (it can be guessed looking at the bottom of the buckets). While from a collaborative point of view, the visualizations seem useless in a system being developed only by one developer, recording the changes

can be useful from an evolutionary point of view. The information collected by Syde can be useful in term of reverse engineering and Scamp's views can help detecting pattern in the data: for example the size of tags can indicate the importance of an entity.

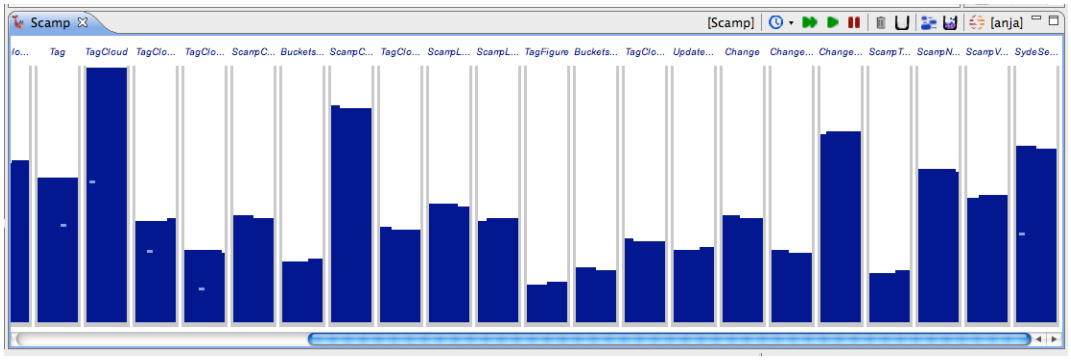

Figure 4.7: Scamp's Buckets in March.

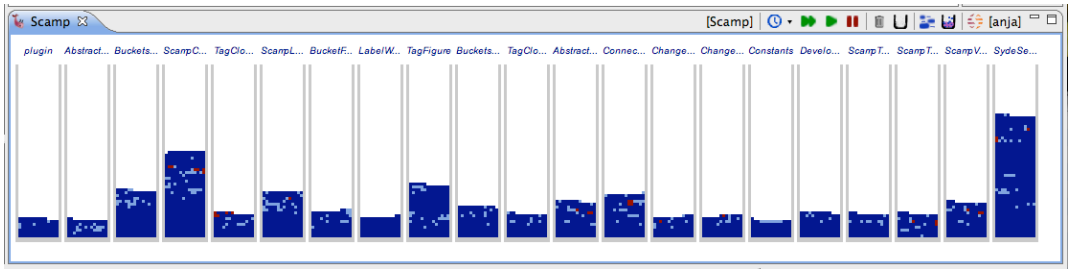

Figure 4.8: Scamp's Buckets in April.

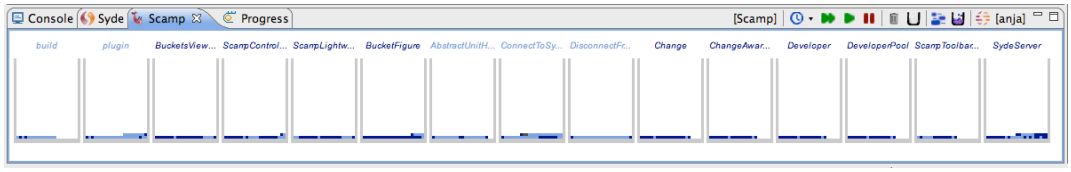

Figure 4.9: Scamp's Buckets in May.

In April, as we can see in Figure 4.8, a second person (color *azure*) joined the development. From an awareness point of view, we can see that during April the changes have different color, indicating that two person collaborated. This can also be seen by looking at the corresponding Tag Cloud view, in Figure

4.12. However conclusions about the development effort made by a developers, should be drawn after looking at the buckets. As an example, in Figure 4.10 we can compare a Tag Cloud view, where all the tags have the same color, to the corresponding Buckets view, where we can see that the units are actually being modified also by others developers.

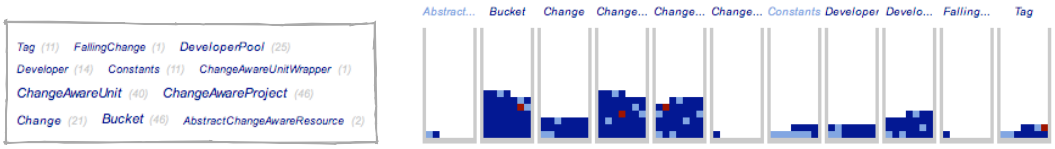

Figure 4.10: Contributions by developers: Tag Cloud vs. Buckets.

Comparing the screenshots it is possible to notice that the three Tag Cloud views differ from each other, however some units (such as *SydeServer*) can be spotted in each month. Those unit are the central ones in the system, thus have constantly been evolving through the development, for example to extend them because new features were added or to fix some issue.

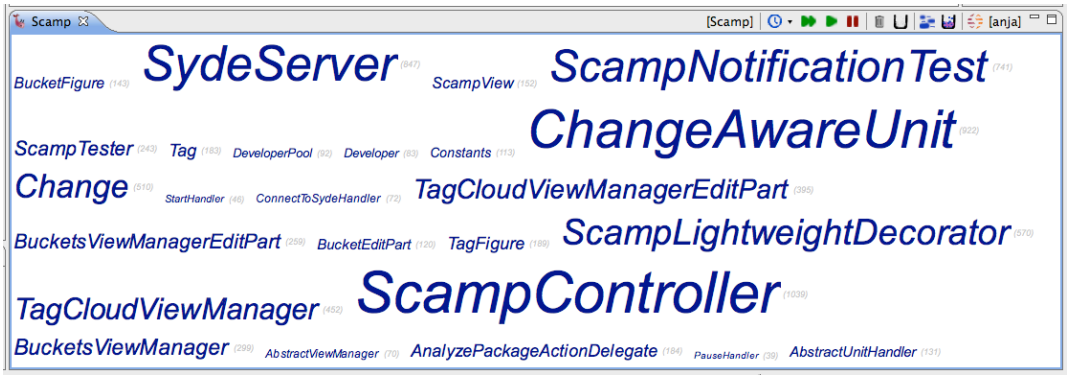

Figure 4.11: Scamp's Tag Cloud in March.

Table 4.1 presents the number of changes monthly made to the project as a whole and to three of the most modifies units, while Table 4.2 summarizes the total number of changes for each one of them, compared to the number of CVS commits for the corresponding file. We can see that the number of quantity made is remarkably larger with respect to the commits to the repository. However, this could also be argumented with the fact that in a one-developer project, the repositories serves as backup storage, rather than as a synchronization point.

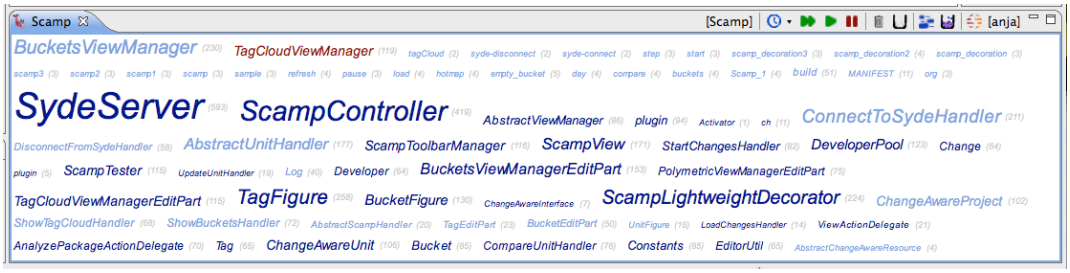

Figure 4.12: Scamp's Tag Cloud in April.

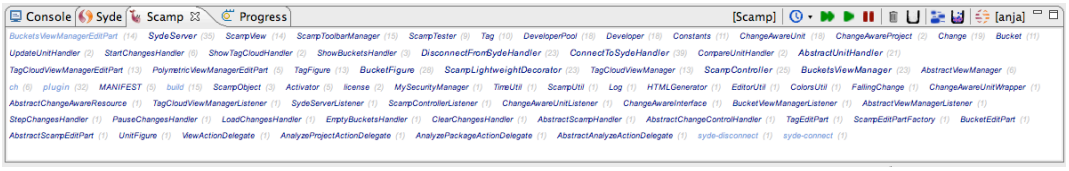

Figure 4.13: Scamp's Tag Cloud in May.

	whole project	*ScampController*	*SydeServer*	*ChangeAwareUnit*
march	25,458	1,039	847	922
april	17,091	419	593	106
may	2,564	25	35	18
total	45,113	1,483	1,475	1,046

Table 4.1: Changes made on Scamp as a whole and on three central units.

	changes	*CVS commits*
ScampController	1,483	32
SydeServer	1,475	52
ChangeAwareUnit	1,046	23

Table 4.2: Number of changes vs. number of CVS commits for the three most changed units of Scamp.

Chapter 5

Conclusions

5.1 Summary

We developed Scamp to visualize information about ongoing development in multi-developer projects. Our tool informs its users about changes made to the system under analysis. We developed three kinds of visualization, each one dedicated to a different aspect of the data.

There is a *Tag Cloud view*, which focus on notifying developers about changes *as they happen*, allowing developers to follow all the changes made by others in real time. This visualization displays the system as being alive and constantly modified, moreover it gives indications about the most modified entities in the system. Following, there is the *Buckets view*, which shows *all the changes* made to the system, with indication of who made each change. The main goal of this second view is to see how the project is evolving, based on what each developer in the team is working on. The *Decoration* is the third visualization kind offered by Scamp. It allows to see what changed, when and by whom directly from the Package Explorer, in a more lightweight fashion with respect to the other two proposed visualizations. The decorations added by Scamp will eventually be seen by users at the moment they open a file, informing them about the evolution of the project, even if the main view is not visible.

Scamp can potentially reduce merge conflicts, by notifying users about every change that happens. A developer, once is aware that some other team member is modifying an entity, can decide to wait before performing some change to it. Our prototype also allows to know whom to ask in case of questions or problems relative to some part of the system, in fact with Scamp code owners can be easily identified. Additionally, Scamp can be useful in terms of software evolution analysis, because it depicts all the changes made to the system, giving an insight into the development efforts made to develop it. In the specific, the Tag Cloud

gives information about the total effort made into each unit's development, in the context of the project (the information is contextualized, because the its relevance is determined by comparing sizes between one tag and the other in the same project); whereas the Buckets view gives information about how the team members worked on the project, revealing, for example, which entities have collaboratively developed and which have otherwise been mainly under the responsibility of only one developer.

Besides using and testing our prototype extensively, while monitoring our own ongoing development, we had the possibility to argument the validity of Scamp by having four students using it during the development of their semester's projects. However, we also plan to apply Scamp in the context of larger (both in term of size and number of developers) projects. We identify a number of future improvement and extension to the plug-in, which we enumerated in Section 5.3.

5.2 Discussion

With its Tag Cloud, Buckets and Decoration, Scamp helps to keep track of the evolution of a project, and is particularly useful for teams of developers working de-located, because it informs all the team members about who is touching and changing each entity (i.e. file) in the project. We think that using Scamp, every developer is constantly aware that he/she is part of a team. It is also possible that Scamp enhances communication among team members, because once they are informed about some ongoing change they might be interested in, they can contact their colleagues to obtain further information.

The advantages of being integrated into a development environment are multiple. For example, Scamp can is at disposal of a large community of Java developers without having them to switch development environment. However this also limit the kind of visualization that can be offered (for example, offering the possibility to visualize the development of a system in a movie like manner, with the possibility to go back in time and replay changes, it is still unrealistic in the context of Eclipse). Additionally, the plug-ins architecture needs the plug-ins to be lightweight and use a limited amount of resources, whereas animated views (such as as those provided by Scamp) needs to be constantly refreshed. Scamp's view are therefore "heavy" in terms of needed resources (such as memory). Moreover, due to the large amount of data about changes, the visualizations can filled only when the user is online (even if he wants to see past changes).

With regards of its validation, Scamp needs further investigation. We find that observing the users could also give more information about its validity.

5.3 Future Work

As previously mentioned in the document, Scamp can be both improved and extended. In this section we will list some of the improvements and give details about what could be added in the future releases of the plug-in. Scamp's future works can be categorized in the following subsets:

- features to enhance usability
- improvements for the current visualizations
- additional views
- stability and performance

5.3.1 Features to enhance usability

Scamp is prototype and many features could be added to enhance it and ameliorate the user's experience. We are listing here a few future works that could be implemented.

- The main feature we plan to add in the near future to enhance the plug-in usability is the possibility to change the starting time for the visualized time span. This could be implemented by means of a **calendar** that would let users choose a date. Additionally Scamp could provide the possibility to choose a custom time span.
- Filters are another features that can be added across visualizations. Scamp provides the possibility to limit the monitoring to a particular selected package, however the possibility to **filter the visualized units** in a custom way is missing. In particular for the Buckets view, it could be useful to limit the set of visualized buckets to some in which the user is interested. Scamp could furnish developers with a set of predefined filters, such as *by developer(s)* (which would show only units that have been modified by the given developer or by all the specified developers), *by workspace's files* (which would show only those units for which a corresponding file exists in the user local copy of the project) or *by viewed file* (which would show only those buckets relative to the files currently opened in the user's Eclipse editor).
- The scale used for the **size** of tags in the Tag Cloud and for the changes in the Buckets should be tested and calibrated accordingly. At the moment the views are adjusted for rather small projects developed by 1-2 people.

Further investigation about larger projects should be done in order to get an insight of how many changes per day, week and month there are on average and, to then resize tags/changes according to the time span visualized. The views could also be adjusted automatically given the project statistics (computed automatically), however this would reduce the consistency between views over time. Moreover, a **zoom-in/zoom-out** feature to scale sizes in a custom way could be implemented to allow users to adjust the views in a custom way.

- Future work include the possibility to **automatically connect** to Syde as the user open Eclipse.
- We plan to implement a **file comparator** in the next releases, giving users the possibility to compare their local copy to the one on Syde server. The comparator would also help in case of conflicts, allowing to accept of refuse new changes.
- Scamp, with special support from Syde, could also implement **hierarchies among developers**, which will be reported when notifying changes. This can be particularly useful in large teams, where developers have different roles and power of action in the project. For example, the decoration on changed entities could be of different shape, and/or color, indicating the role in the team that the person who did the change has.

5.3.2 Improvement for the Buckets visualization

We consider a number of possible improvements for the Buckets visualization. In particular, further investigation should be done regarding the number of visualized buckets and their sorting order. The following ideas should be implemented and tested in a prototype:

- a metric to select the most **relevant buckets**. We propose to select the most frequently changed, weighing past and recent changes differently.
- A different way to display more buckets in the view. We propose to implement the view as if it has **multiple "pages"**. The top most relevant buckets will then be in the first "page", while the less relevant in the last one. Users could browse the different pages, looking at buckets from the most to the least relevant ones. This would avoid the problem of the current view, where some relevant units results at the "far end" of the view.
- The possibility to distinguish the **type of changes** in *modifications*, *additions* and *removal* of code. Different shapes of the change's representation

could be used (to keep the information about the developer as color). Otherwise provide a way to switch the mapping on the color from the developers to the type and back. Visualizing the type of changes in the Bucket view could reveal different pattens, giving more information in particular about buckets with many changes.

5.3.3 Additional views

Additionally to the *Developers* view, that we already described in details in Section 3.3.4 and which is under development, more views can be added to Scamp to both improve collaboration awareness and to help evolution analysis. The plug-in is designed so that additional visualizations can be added easily, from a technical point of view.

A few ideas about additional visualizations include:

- **heat maps**
- **system complexity**
- **charts** (sparklines)

Heat maps could be exploited to argument collaboration awareness in a number of ways. A first possibility is to design a visualization in which the whole system is represented (for example by means of a **system complexity**) and where each unit changes color with an heat map fashion. New incoming changes would make the color of the relative unit go toward red (or darker), while as the time goes by without modifications, it would change toward blue (or lighter). The visualization can then be enriched further with information about developers. For example, making the size of the unit bigger or smaller according to the number of developers working on it in certain period of time, giving both a "tag cloud" and a heat map effect on the visualization (the first one emphasizing the number of developers working on a unit, whereas the second one points out the number of changes to units). This view would therefore show (in real time), where the development efforts are distributed over the system and make some patterns distinguishable at sight.

The **system complexity** view can also be transposed to a collaborative context in a number of different ways, other than overlying it with a "changes heat map", for example adding a distinct marks for each developer that modified it or by coloring each unit according to its "owner". The advantage of such visualization is that it would contextualize changes in the system as a whole.

Charts could also be added to Scamp, in particular with the goal of giving information about the exact chronological sequence of changes, for example depicting them on a time line. Each developer could be represented by a

"serie" in the chart, which would embed its color. Charts can make different patterns stand out, stressing out one or the other variable (such as frequency of changes), however scalability must be carefully considered. Collaboration data is multivariate and multiscale, potentially leading to huge clumsy charts. Figure 5.1 and 5.2 show an example of a possibile visualization: a **Sparkline**. Each sparkline would represent a single unit on a time line, showing all the changes relatives to it by means of a marker placed whenever a change has been made. The marker's color matches the color of the developer that made it. Additionally, Sparklines could show information about the type of the change, by using different markers. This view could show every unit (or a subset of them, like in the current Buckets view), or it could visualize only selected and requested units (for example, Scamp could give the possibility to "inspect" a unit from any other view, which would show its relative Sparkline). The sparkline has the advantage of being a compact visualization, which can give many information at a glance.

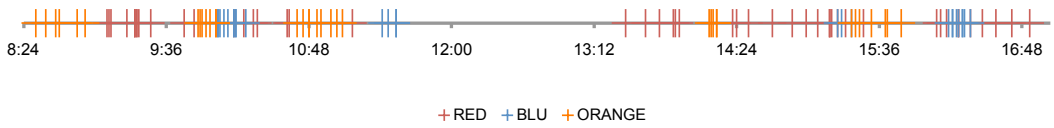

Figure 5.1: Hypothetical Sparkline visualization: unit changed by three developers.

Figure 5.2: Hypothetical Sparkline visualization: unit changed by one developer.

5.3.4 Stability and performance

Scamp's performance could be greatly increased: in fact, to avoid to deal with memory problem due to caching at the early stage of our prototype, we decided to load changes when the time span or the project vary, with consequent increased waiting time for the user and increased network loading. Caching of changes could be introduced to speed up the loading of past changes and the switching from a time span to the other, however in addition to the usual memory issues caching needs to be thoroughly thought for the particular environment Scamp is in. In particular it needs to be taken into account that some user could be offline at the moment they make some changes (logging to Syde only later),

in this case Scamp should be able to detect and retrieve those “old” changes to avoid losses of informations and consequence inconsistency between views.

Scamp needs to be improved in term of stability too. In particular, it does have some memory issues with the Buckets view when there are lots of changes (i.e. when visualizing a whole month).

Bibliography

[DB92] P. Dourish and V. Bellotti. Awareness and coordination in shared workspaces. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW'92)*, pages 107–114, Toronto, Ontario, 1992. ACM Press.

[GG02] Carl Gutwin and Saul Greenberg. A descriptive framework of workspace awareness for real-time groupware. *Computer Supported Cooperative Work (CSCW)*, 11(3 - 4):411–446, 2002.

[GKSD05] Tudor Gîrba, Adrian Kuhn, Mauricio Seeberger, and Stéphane Ducasse. How developers drive software evolution. In *Proceedings of International Workshop on Principles of Software Evolution (IWPSE 2005)*, pages 113–122. IEEE Computer Society Press, 2005.

[GPF99] Luis A. Guerrero, Roberto C. Portugal, and David A. Fuller. Top: A platform for the development of web interfaces and collaborative applications. *CLEI Electron. J.*, 2(2), 1999.

[Heg09] Rajesh Hegde. Collaborative development environment using visual studio. `http://research.microsoft.com/en-us/projects/collabvs/default.aspx`, June 2009.

[HL09a] Lile Hattori and Michele Lanza. An environment for synchronous software development. In *Proceedings of ICSE 2009 (31st ACM/IEEE International Conference on Software Engineering - New Ideas and Emerging Results Track)*, pages 223–226. IEEE CS Press, 2009.

[HL09b] Lile Hattori and Michele Lanza. Mining the history of synchronous changes to refine code ownership. In *Proceedings of MSR 2009 (6th IEEE Working Conference on Mining Software Repositories)*, pages 141–150. IEEE CS Press, 2009.

[RL05] Romain Robbes and Michele Lanza. Versioning systems for evolution research. In *Proceedings of IWPSE 2005 (8th International*

Workshop on Principles of Software Evolution), pages 155–164. IEEE Computer Society, 2005.

[RL06] Romain Robbes and Michele Lanza. Change-based software evolution. In *Proceedings of EVOL 2006 (1st International ERCIM Workshop on Challenges in Software Evolution)*, pages 159–164, 2006.

[RL07] Romain Robbes and Michele Lanza. Towards change-aware development tools. Technical Report 6, Faculty of Informatics, Università della Svizzerra Italiana, Lugano, Switzerland, may 2007.

[RL08] Romain Robbes and Michele Lanza. Spyware: a change-aware development toolset. In *ICSE*, pages 847–850, 2008.

[SES05] Janice Singer, Robert Elves, and Margaret-Anne Storey. Navtracks: Supporting navigation in software maintenance. In *International Conference on Software Maintenance (ICSM'05)*, pages 325–335, sep 2005.

[SGPP04] Kevin A. Schneider, Carl Gutwin, Reagan Penner, and David Paquette. Mining a software developerÕs local interaction history. In *In Proceedings of the International Workshop on Mining Software Repositories*, pages 106–110, 2004.

[SKSH96] Christian Schuckmann, Lutz Kirchner, Jan Schümmer, and Jörg M. Haake. Designing object-oriented synchronous groupware with coast. In *CSCW '96: Proceedings of the 1996 ACM conference on Computer supported cooperative work*, pages 30–38, New York, NY, USA, 1996. ACM.

[SNvdH03] A. Sarma, Z. Noroozi, and A. van der Hoek. Palantir: raising awareness among configuration management workspaces. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 444–454, 2003.

[SSS99] Jan Sch§mmer, Christian Schuckmann, and Till Sch§mmer. Coast in action: An efficient way to build complex groupware, 1999.

[Wik09] Wikipedia. Visualization (computer graphics). `http://en.wikipedia.org/wiki/Visualization_(computer_graphics)`, May 2009.