A groupware system to support Collaborative Programming: design and experiences

Crescencio Bravo¹, Rafael Duque²*, Jesús Gallardo³

¹ Department of Information Systems and Technologies
University of Castilla – La Mancha
Paseo de la Universidad 4, 13071 Ciudad Real (Spain)
Crescencio.Bravo@uclm.es

² Department of Mathematics, Statistics and Computer Science
University of Cantabria
Avenida de los Castros s/n, 39005 Santander (Spain)
Rafael.Duque@unican.es

³ Department of Computer Science and System Engineering
University of Zaragoza
Ciudad Escolar s/n, 44003 Teruel (Spain)
jesus.gallardo@unizar.es

Abstract. The advances in network and collaboration technologies enable the creation of powerful environments for Collaborative Programming. One such environment is COLLECE, a groupware system to support collaborative edition, compilation and execution of programs in a synchronous distributed fashion, which includes advanced tools for communication, coordination and workspace awareness. The article analyses firstly some usability and design issues, discussing strengths and weaknesses of the system as a basis for the development of groupware tools to support Collaborative Programming. Then, the focus is on a number of experimental activities carried out. COLLECE was used to conduct a set of experimental activities about work productivity and program quality when comparing the activity of pair and solo programmers, and to analyse potential associations between ways of working and collaborating, and specific characteristics of the programs produced.

1 Introduction

The advances in network and collaboration technologies enable the creation of powerful collaborative environments from which many activities and disciplines can benefit. Computer Programming is a complex and creative task that can take advantage of such environments and thus be supported and enhanced using groupware systems and distributed architectures. The increasing complexity of software projects and expanding geographical distribution of the members of the development teams lead to increasing problems and difficulties in the programming phase. Collaborative Programming (CP) is facilitating these scenarios by allowing distributed programmers to work jointly on the same program or software application. Having a closer look at CP, it can be seen that this would be of application in two different settings: software industry and academia.

In recent years, Agile software methodologies (Williams, 2010) have attracted increasing interest within research industrial environments, with Extreme Programming (XP) being considered the most important of these methodologies (Boehm & Turner, 2004). Pair Programming (PP) (Williams & Kessler, 2002) is one of the twelve key practices of XP (Beck, 1999). In Pair Programming, the development of computer programs through the collaborative work of two programmers is proposed. These programmers solve their tasks on a single computer, by means of a process in which one of the programmers plays the role of driver and is responsible for implementing the program source code, while the other plays the role of observer and takes charge of detecting mistakes and possible improvements in the source code that is being coded by the driver. While the task is being carried out, the roles of driver and observer are interchanged by the programmers. Previous studies (Nosek,
1998; Williams & Kessler, 2000) indicate that PP does not only accelerate the problem solving processes, but it also improves the quality of the programs.

PP practices force pair programmers to work face-to-face. On the other hand, in Distributed Pair Programming (DPP) (Baheiti et al., 2002) the two members of the team collaborate synchronously on the same programming task from geographically distributed locations using generic or specific groupware tools. CP is a generalization of DPP in which the number of programmers is not limited to two people and that new rules can be necessary to support an efficient collaboration.

CP is also a promising tool to scaffold the collaborative learning of Programming, especially since Programming is a difficult subject for students to learn and for teachers to teach. This is due to the fact that students are usually capable of building a solution of a programming problem by visualizing it in their mind or expressing it in an outline, but they find it difficult to transform such a solution into a computer program using building blocks (computer instructions) that are abstract as well as unfamiliar to them (Karsten & Kparthi, 1998). Johnson (1998) points out that the process of analysing and criticising software artefacts produced by other people is a powerful method for learning programming languages, design techniques and application domains. Accordingly, we deal with the difficulties students find in learning Programming by adopting a computer-assisted learning approach and providing learner-centred settings, promoting learning-by-doing by means of collaborative problem solving activities (see, e.g., Bravo et al., 2006b).

Most PP research has been carried out without using collaborative technologies. The studies carried out to determine the advantages of PP have focused on evaluating the final result (the program produced) of the programmers’ work (Arisholm et al., 2007; McDowell et al., 2002; Müller, 2007; Vanhanen & Pekka, 2007). However, little research has been conducted into the process of collaborative work between programmers. This situation arises from the lack of tools supporting collaboration and the storing of the interactions of the users with the system (from those concerned with program implementation to those relating to coordination and communication between programmers). In addition, the few tools available (see Section 2) show limitations at usability, coordination, communication and awareness level. Awareness is a critical issue in groupware because it is aimed at allowing the users to easily perceive and gain knowledge of the interaction carried out by other people in a shared workspace (Gutwin & Greenberg, 1997).

Taking these ideas into account, this article has a twofold objective. Firstly, we discuss some design issues of CP systems. Secondly, we study the effects of using groupware tools to support DPP activities. Thus, we developed the COLLECE system as a means of dealing with these two research objectives, allowing the exploration of the benefits of CP in teaching-learning as well as professional settings. COLLECE enables users located at different workstations to create computer programs collaborating simultaneously to solve the same task. In so doing, COLLECE provides shared workspaces for the tasks of edition, compilation and execution of programs as well as advanced collaboration support. COLLECE provides an activity subsystem to organize problem solving activities and an analysis subsystem (Bravo et al., 2008) whose functionality consists of (i) storing the interactions carried out by the users and the programs built with the tools incorporated in the system in a data repository, (ii) analysing the work of users and groups in terms of high-level variables called indicators (e.g., speed, communication, coordination), and (iii) providing feedback to the users (programmers) about their activities.

Regarding the first objective, some design and usability principles are discussed and some controlled experiments are conducted for studying communication, coordination and awareness support in CP activities. In relation to the second objective, two studies are carried out for analysing both the behaviour of programmers when solving programming activities and the result of their work. In the first study, the work of some pair programmers is compared with the work of solo programmers (those who worked individually) when approaching the same programming tasks. The variables considered in this study were work productivity and program quality. In the second study, the influence of the collaboration process (collaborative programming) in the product built by the programmers (the program) is analysed. For this purpose, a number of variables referring to the process and to the product were considered.

This article contains five additional sections. Section 2 reviews the literature focusing on groupware tools to support DPP, identifying gaps in the different systems and approaches presented. The COLLECE system is presented in Section 3. Section 4 describes the aforementioned studies and Section 5 discusses the findings, including a comparison between the systems shown in Section 2. Finally, Section 6 concludes the article and draws some lines for future work.
2 Related work

Some studies indicate that PP reduces the number of errors that appear in solo programming practices (McDowell et al., 2002; Müller, 2007). Within the studies that do not use groupware tools, some have focused on the industrial sector to discuss particular aspects such as the influence of the programmers’ experience on the improvement of productivity in PP (Lui & Chan, 2006), the results obtained in PP depending on the complexity of the system to develop (Arisholm et al., 2007), or the quality factors of the software product built through PP practices (Vanhanen & Pekka, 2007). However, some authors question the benefits of PP because in certain studies they did not find an increase in the quality of the software products produced nor any improvement in productivity (Hulkko & Abrahamsson, 2005), and the same may occur with a reduction in the time required to complete the tasks that brings with it fewer errors or other advantages in the software products (Nawrocki & Wojciechowski, 2001).

A second set of studies have experimented with DPP practices where programmers were located in distributed workstations using basic technological support. Thus, general-purpose collaborative tools such as NetMeeting Instant Messaging to communicate and VNC (Virtual Network Computing) to share the desktop were used. However, there is a lack of groupware specifically built to support DPP providing suitable mechanisms for coordination and communication between the driver and the observer.

A number of studies have approached the problems of supporting CP on the Internet. RECIPE (REal-time Collaborative Interactive Programming Environment) (Shen & Sun, 2000) allows geographically distributed programmers to participate concurrently in the design, coding, testing, debugging and documentation of a program. To do this, RECIPE facilitates the conversion of single-user compilers and debuggers in collaborative applications and the integration of existent collaborative editors into the system. However, it does not offer specialized tools for communication and lacks a suitable awareness support because of its high coupling, because each user executes the same application instance that cannot be used and configured according to his/her preferences. The DPE environment (Jo & Arnold, 2003) supports collaborative edition, compilation and execution of programs, and incorporates channels for text-based and audio communication. However, the system presents limited support for task coordination and awareness mechanisms.

Some attempts (e.g., GILD, 2012) have been aimed at incorporating collaborative support (forum, chat, instant messaging, collaborative editing, etc.) in Eclipse (2012) through plug-ins, but these are still in their initial stages. The most significant work in this matter is Jazz Sangam (Devide et al., 2008). This system incorporates instant messaging and a version control system, needed due to the fact that the synchronization in the shared workspace requires that each programmer have the same program version available in the shared editor from the beginning. Systems in this category suffer from limitations at awareness level due to their high coupling and also from a lack of specialized communication and coordination tools.

Two recent systems based on cloud computing are Cloud9IDE and Kodingen. Cloud9IDE (2012) is a web-based cloud integrated development environment originally oriented to Javascript programming, although now it supports many different languages. This system puts together an online development platform for open-source code, code repositories, a deployment environment and public/private projects. Kodingen (2012) is an online development environment including code editor, cloud hosting, database administration and web-based file system. These two systems present low-range collaboration with no awareness mechanisms; their support for collaborative work is limited to store and access projects in the cloud, to create teams and to use code repositories.

On the other hand, some highly interactive synchronous collaborative editors offer advanced tools for text joint edition, including source code colouring based on syntax. One of these is Gobby (2012), which includes an IRC-like chat for communication, and another is SubEthaEdit (2012), with member panels, tele-cursors, group scrollbars and chat. However, neither of these provides support for program compilation and execution.

In the scope of collaborative programming learning, we can find JeCo (Moreno et al., 2004). JeCo is an integration of two systems: Jeliot 3 and Woven Stories. The former animates the execution of Java programs. The latter is a co-authoring tool with which users create documents (programs) and connections among them. In so doing, Woven Stories supports asynchronous collaboration, and a chat incorporated in the system for students’ discussion allows synchronous collaboration. The system lacks facilities for shared edition and animation.

Having the features and gaps of the systems above in mind, a specific groupware system, namely COLLECE, has been developed. This is described below.
3 COLLECE: a groupware system for Collaborative Programming

COLLECE (COLLaborative Edition, Compilation and Execution of programs) allows users to edit a program or code fragment, to compile it and to run it collaboratively. Up to now, the languages supported are Java and C.

Because the system is primarily used for teaching-learning purposes, two different actors are recognized: teacher and student. The teacher defines the work sessions and arranges the users participating in them by using management tools. A session is defined by means of a name, a type, a file containing the formulation of the problem to be solved and a schedule in which the session has to be carried out. The problem formulation includes a textual description of the objectives, requirements and constraints to be fulfilled with the creation of a program. When the students access the system, the session management tool is opened, showing a list with the sessions available. Some of them are public and others are private. Any user can access a public session, whereas it is necessary to be a member to access private sessions.

When a session is accessed in the scheduled time, the COLLECE workspace is opened (Fig. 1). In order to design it, we took the semi-structured model for synchronous collaborative problem solving proposed by Bravo et al. (2006a) as a base. This model proposes Scripting (Collaboration Protocols) (Wessner et al., 1999) to structure the high-level tasks, Language/Action Perspective (Winograd, 1988) to express and categorize actions for users’ coordination, and Flexible Structuring (Lund et al., 1996) for communication between users.

In order to carry out the programming tasks, an explicit collaboration protocol must be followed. First, the students create a program using the collaborative editor (Fig. 1, ①); after which they are able to compile the program, receiving a list of compilation errors (Fig. 1, ②). Finally, they can execute the program provided a compiled program is available. Iterations are possible between these three tasks. However, despite this script, the students are free to make their own decisions on when to edit, compile and execute, as well as to decide who is responsible for each task. To do so, coordination tools are available in the workspace to regulate the navigation through the collaboration protocol.

Such coordination processes are modelled with a simple protocol of actions extracted from language. In order to regulate the edition turn assignment (Fig. 1, ⑤), we identified the following acts: Request the edition turn,
Give and Don’t give. With these acts, a user can request the edition turn, and his/her fellow users can express his/her agreement or disagreement. When all the users in the group agree, the assignment is made. Similar acts are used for coordinating when to compile and when to execute the program (Fig. 1, ② and ①, respectively). These coordination tools support multiple proposals, that is to say, proposals coming from more than one user; as a result, lists are required to contain the historical proposals, enabling a user to select the proposal to which he/she wants to respond.

The communication during the tasks takes place by means of a structured chat (Fig. 1, ②). This chat is a structured one because it offers a pre-established set of communication acts, aimed at providing explicit communication acts that encourage the users’ participation, reducing the writing load and focusing the users on the task. Apart from the so-called structured messages, the chat also provides the users with free-text messages and the possibility of selecting one of the last messages sent in order to reuse it.

In the final COLLECE user interface four main areas are identified: the edition area at the top (Fig. 1, ① and ③), the console in the middle (Fig. 1, ④, ⑤ and ⑥), the chat at the bottom (Fig. 1, ⑦) and the session panel on the right (Fig. 1, ⑧). Two functions of the system allow users to consult the formulation of the problem as well as the compilation statistics. The former shows a textual description of the problem to be solved. The latter displays an ordered list of compilation errors and their frequency, so that the students are aware of their more frequently made mistakes.

3.1 Design and implementation issues

When implementing COLLECE, we worked with three Programming teachers in the participative design of the user interface. First, some paper prototypes were designed and evaluated by these teachers. A software prototype was then elaborated and put to the test in a formative evaluation with these teachers as well as some software engineers. In so doing, a final design was implemented (see Fig. 1) in accordance with the following main design principles:

− The three main tasks (edition, compilation and execution) are contained in a single window (the main user interface).
− The edition area consists of a shared text editor whose floor control is regulated by a turn-taking mechanism based on a coordination tool.
− Specific coordination tools also have to be available for deciding when to compile or execute the program.
− The console area shows both the compilation errors and the execution outcome.
− The edition and console areas are not fully WYSIWIS (What You See Is What I See), since each user can decide which part of these areas to visualize. In other words, there is a significant degree of disacoupling.
− Awareness is provided mainly in the form of tele-pointers, state labels, lists of past interactions and beeps.
− The user interface also contains a structured chat and a session panel.

Following its implementation, a number of teachers and students used the system to carry out some CP activities with the purpose of evaluating it. Thus, some important conclusions were drawn regarding the user interface effectiveness and some refinements were made. Among them, we highlight the following improvements:

− The inclusion of a semaphore in the coordination tools, so that the users can easily perceive (with a green light) when and where there are proposals of other users still pending an answer.
− The addition of a visual indicator to the edition scroll bar of the remote users to indicate where the editor user is working.
− The refinement of the set of pre-defined messages included in the structured chat.

Besides coordination and communication support, awareness support is also available. Herbsleb & Grinter (1999) see the lack of awareness as one of the major problems in distributed software development. COLLECE deals with the problems of awareness by providing a number of techniques to inform about people, their state and their actions. Specifically, COLLECE’s awareness is made possible by means of a number of elements: (i) session panel (Fig. 1, b); (ii) global state (editing, compiling or executing) (Fig. 1, k) and individual state (Fig. 1, i); (iii) tele-pointers, in the form of a coloured rectangle drawn around the source code line (Fig. 1, b); (iv) lists of interactions (Fig. 1, e); (v) semaphores (Fig. 1, f); (vi) beeps, when actions occur; (vii) users’ position in the source code (Fig. 1, c); and (viii) other mechanisms (Fig. 1, a, d, g and j) such as rectangles and labels to highlight the user and show some elements of information.

The system, developed using Java technology, operates on client/server architecture. The data management as well as the synchronization services for implementing the synchronous collaboration are centralized on a server, whereas the distributed clients (the users executing the system) access the system from a web page using the

3.2 The analysis model of COLLECE

COLLECE incorporates a subsystem for collaboration and interaction analysis built using the process-solution analysis framework proposed by Bravo et al. (2008) as a basis. This analysis framework is based on the concept of analysis indicator. An analysis indicator says something about the characteristics of the individual activity, the form of collaboration or the quality of the collaborative product. The indicators referring to the process, in this case the collaborative work of programming are Communication, Coordination, Work, Velocity, Experimentation, Willingness and Collaboration; the indicators referring to the product, which is a program which solves a problem, are described in Section 4.5.

This way, COLLECE is a complement to the traditional theoretical lessons of Programming. Using COLLECE, the students can practice the building of programs in groups using a problem solving approach, following a learning protocol defined by the teacher in terms of group organization, problem complexity and solving order. During this task, the students can receive feedback about their activity in the form of recommendations and visualizations of sets of analysis indicators, which would provide extra scaffolding to the added benefits of doing DPP.

4 Studies

In this section we present a number of studies in which the COLLECE system was used (Bravo et al., 2007; Duque & Bravo, 2008). Students enrolled in the first and fifth years of Computer Science studies (BSc) of Computer Science Engineering at the University of Castilla – La Mancha (Spain) as well as employees of a software factory took part in the studies.

4.1 Description and method

In order to conduct the study we considered the following research questions:

Q1. Is the designed structured chat tool a useful tool in CP?
Q2. Are the designed coordination tools to regulate the use of the different programming workspaces suitable tools?
Q3. Is the awareness support useful?
Q4. What is the effect of using CP tools in contrast with single-user tools in terms of productivity and program quality?
Q5. Are there significant correlations between ways of working (collaborating) and specific characteristics of the programs built?

The first three questions fall in the sphere of usability because a suitable communication, coordination and awareness support in a groupware tool contributes to the tool’s usability. This support should constitute an essential component of CP systems, which however was not found in many of the systems proposed in the literature. Within the Human-Computer Interaction discipline, usability plays a crucial role in assuring the quality and success of any interactive system (ISO, 2011). In collaborative systems, usability is especially important, and different evaluations are required to guarantee the best usability of a product. To give an answer to such questions, we define different programming activities as follows.

For questions Q1, Q2 and Q3, 51 five-year students were organized in 17 pairs (34 pair programmers) and 17 solo programmers. It is worth noting that all the students participating had a similar level of knowledge and skills as they were all in the same academic year, had at least one year’s experience with the Java programming language and received a training course in COLLECE. The students were randomly arranged in groups and designated as distributed pair programmers or solo programmers, irrespective of their age, sex, knowledge, etc. Then, they were asked to create a program for sorting ten previously read numbers and calculating the median, an activity that we designated A5. The time available for this activity was 45 minutes. The use of the system was logged for later analysis and some questionnaires were prepared to collect information from the users. Although the solo programmers worked individually, they also used the COLLECE system to facilitate the analysis of their work, because, as in the case of pair programmers, the COLLECE analysis subsystem stores user activity (time used, interactions, etc.) in a database.
The two last questions are aimed at studying real benefits of supporting CP. Thus, for questions Q4 and Q5, a different sample of 38 first-year students, organized randomly in pairs, approached three different activities about recursivity consisting respectively in calculating whether a sentence is a palindrome, implementing the Euclidean algorithm or implementing the Fibonacci function, to which we refer as AP, AE and AF. In the first and second problem we took into account 19 pairs, but in the third activity only 17 pairs were considered because two groups did not produce any solution. In these activities the time available was 40 minutes. As in the previous case, the users’ activity was logged for analysis purposes.

Below, we present some results in sections 4.2 to 4.4 corresponding to questions Q1, Q2 and Q3 respectively and to activity AS. Sections 4.5 and 4.6 refer respectively to questions Q4 and Q5, and are related to three activities about recursivity: AP, AE and AF. Section 4.7 includes a study of a more complex and real programming activity in a software factory with experienced programmers, which deals with questions Q4 and Q5 as well. Later, we will discuss all the results in Section 5.

4.2 Use of the chat

As in real life, people need to communicate when carrying out a complex task as programming in a group is. The sentence-openers approach has proved to be a potentially effective mechanism for structuring communication in a number of systems (e.g., McManus & Aiken, 1995). As a result, we chose to incorporate a chat with sentence openers in COLLECE. The study described here aims to evaluate the suitability of such a chat.

To derive a set of chat messages (sentence openers) for use in the Programming domain, we first met with some Programming teachers in order to identify potentially interesting chat messages. From this, 19 messages were initially formulated. They were filtered when considering the messages exchanged between the users during some preliminary tests of the formative evaluation (see Section 3.1). Finally, the teachers were interviewed again to validate a final set of 10 messages, 6 of which propose a sentence to be completed by typing text. The texts are shown in Fig. 2.

![Structured messages](image)

**Fig. 2.** Use of the structured chat in AS.

In AS, the most used message type was the free message (92 %), and the least used was the structured message (3 %). The remaining 5 % of messages were messages already sent (reused). Analyzing the structured messages (see Fig. 2), “I think that…” was the most used (17 times, 37 %), followed by “This already works!” (12 times, 26 %), and “In the loop …” was not used.

4.3 Use of the coordination tools

Table 1 shows the number of proposals (requests of edition turn, compilation proposals and execution proposals) and answers made with the coordination tools. There are two unexpected results. Firstly, the number of agreements is less than expected with respect to the number of proposals. Secondly, the number of disagreements is very low. The difference between the number of proposals and of total responses (agreements and disagreements) means that some proposals were not answered.
### Table 1. Number of coordination actions in Aₜ.

<table>
<thead>
<tr>
<th></th>
<th>Proposals</th>
<th>Agreements</th>
<th>Disagreements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Edition</td>
<td>205</td>
<td>99</td>
<td>3</td>
</tr>
<tr>
<td>Compilation</td>
<td>320</td>
<td>202</td>
<td>9</td>
</tr>
<tr>
<td>Execution</td>
<td>190</td>
<td>103</td>
<td>4</td>
</tr>
</tbody>
</table>

#### 4.4 Students’ evaluation of usability and communication and awareness mechanisms

The students of Aₜ were requested to fill in a questionnaire after the working sessions. This questionnaire contained both open and closed questions to register the students’ opinion about different issues of the system, such as its weaknesses, facility of use and of learning, and messages of the chat.

Many students identified several strengths of the system. However, our interest lies more in the weaknesses. At user interface level, the students indicated that there are too many areas in which to interact, and therefore that they had to pay attention to too many things simultaneously. They felt the need for a better editor to edit the source code, which included the number of each line and had syntax colouring. They pointed out that the requests for edition turn, compilation and execution as well as their possible answers should be better highlighted. In regard to the structured chat, they would have liked shortcuts to select the structured messages using the keyboard, but they also suggested that structured messages are not useful because in most cases they preferred to write their own messages. The students identified some structured messages as being of little use whilst others were more useful, something that will be considered in the future. Another issue mentioned was the CP approach: some of the students believed that a more concurrent programming would be a promising advance, and others thought it might be more productive if the users were able to work separately and afterwards integrate or compare their work. However, these approaches follow a different protocol to which the system is presently based on.

Two closed questions of the questionnaire were aimed at indirectly evaluating the system usability. They were “Do you think that the user interface is easy to use?” and “Do you think that the user interface is easy to learn?”. 92.68 % of the students replied positively to the first question, and 97.56 % thought that the user interface is easy to learn.

The students were also requested to evaluate the quality of the awareness information that some specific components of the user interface provide. Table 2 shows the averages scores using a five-point scale. The remote tele-pointer was the highest scoring mechanism, and the semaphore was the lowest. The session panel also scored highly, while the lists of proposals were not considered to be an efficient way of providing awareness information.

#### Table 2. Students’ scores for the awareness tools and mechanisms of COLLECE (M: mean; SD: standard deviation).

<table>
<thead>
<tr>
<th>Category</th>
<th>M</th>
<th>SD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program with which users are working (Fig. 1-a)</td>
<td>3.3</td>
<td>0.7</td>
</tr>
<tr>
<td>Remote user’s tele-pointer in the edition area (Fig. 1-b)</td>
<td>3.9</td>
<td>0.8</td>
</tr>
<tr>
<td>Remote user’s tele-pointer in the editor’s scroll bar (1-c)</td>
<td>3.4</td>
<td>0.8</td>
</tr>
<tr>
<td>Line number in which the editor user is editing (Fig. 1-d)</td>
<td>3.5</td>
<td>1.1</td>
</tr>
<tr>
<td>Lists of proposals, agreements and disagreements (Fig. 1-e)</td>
<td>3.3</td>
<td>1.0</td>
</tr>
<tr>
<td>Semaphore indicating proposals pending answer (Fig. 1-f)</td>
<td>3.0</td>
<td>1.1</td>
</tr>
<tr>
<td>Information about who is the editor user (Fig. 1-g)</td>
<td>3.4</td>
<td>0.9</td>
</tr>
<tr>
<td>Session panel (Fig. 1-h)</td>
<td>3.8</td>
<td>0.7</td>
</tr>
<tr>
<td>Session panel, individual state (Fig. 1-i)</td>
<td>3.5</td>
<td>0.7</td>
</tr>
<tr>
<td>Session panel, who are you? (Fig. 1-j)</td>
<td>3.3</td>
<td>0.8</td>
</tr>
<tr>
<td>Global state (Fig. 1-k)</td>
<td>3.4</td>
<td>1.0</td>
</tr>
</tbody>
</table>

#### 4.5 Analysing work productivity and program quality

The analysis of the work productivity of the programmers and the quality of the resulting programs when using groupware with support for the creation of programs in a distributed way, together with the number of compilation errors, could give an insight into the process of building programs in a CP approach, specifically with the COLLECE system.
Accordingly, we carried out the present study looking for an answer to Q4. The specific objective of the study was to compare the activity of distributed pair programmers to solo programmers. For this purpose we considered the following variables:

- **Productivity**: This quantifies the programmers’ workload per unit of time (Kitchenham & Mendes, 2000). Thus, *Productivity* is computed as a function of the following variables:
  - The time spent on completing the task.
  - The number of edition interactions (e.g., insert character) (NI_edic).
  - The number of interactions aimed at executing the program (NI_exec).
  - The number of communication interactions (chat messages) (NI_com). In the case of solo programmers this value is zero.
  - The number of coordination interactions (NI_coor). These interactions refer to the changing of roles between the driver and the observer, and the synchronization of the compilation and execution processes. In the case of solo programmers this value is zero.

- **Quality**: The programs built by the users were subjected to a process where some experts evaluated different components of the quality. The *Quality* variable is quantified with a natural number belonging to the set \{1, 2, 3, 4, 5\} (1: very low quality, 2: low, 3: intermediate, 4: high, and 5: very high) and is computed using the following analysis indicators (Bravo et al. 2008), which made up our quality model for computer programs:
  - **Subjective assessment**: evaluation of the final solution (program) to the problem by some experts.
  - **Accuracy**: degree to which the program fulfils the problem’s requirements, that is to say, reaches the final objective of solving the problem.
  - **Cost**: size of the program built, computed as the sum of the number of lines of code and the number of instructions.
  - **Validity**: computed from two other indicators:
    - **Satisfaction of constraints**: degree to which the problem’s constraints are satisfied. In the activities under study these constraints were: (i) using a recursive function, (ii) not using a loop, and (iii) reading the inputs by keyboard and writing the outputs by screen.
    - **Well-formed**: degree to which the final solution is well-formed, that is to say, it does not contain syntactical and semantic errors, computed as the number of compilation errors.
  - **Testability**: degree to which the program was tested suitably in the different runnings.

- **CE**: The number of compilation errors that appeared along the whole programming work process.

Fig. 3 shows the inference graph for calculating *Quality*. A + sign on the arc indicates that the relationship is directly proportional. A - sign indicates that the relationship is inversely proportional. The thickness of the arc stroke indicates whether the weight of the antecedent indicator in the calculation of the consequent indicator is very significant (thick stroke) or not (thin stroke).

Table 3 shows the average values of *Quality* and CE for the programs built in the activities A_P, A_E and A_F, only considering those programs that reasonably solved each problem.

---

![Fig. 3. Graph describing how Quality is inferred from other indicators.](image-url)
Table 3. Variables that evaluate Quality and CE in the three activities.

<table>
<thead>
<tr>
<th></th>
<th>AP (Palindrome)</th>
<th></th>
<th>AE (Euclidean)</th>
<th></th>
<th>AF (Fibonacci)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Solo programmers</td>
<td>Pair programmers</td>
<td>Solo programmers</td>
<td>Pair programmers</td>
<td>Solo programmers</td>
<td>Pair programmers</td>
</tr>
<tr>
<td></td>
<td>M</td>
<td>SD</td>
<td>M</td>
<td>SD</td>
<td>M</td>
<td>SD</td>
</tr>
<tr>
<td>Quality</td>
<td>2.60</td>
<td>1.20</td>
<td>2.46</td>
<td>0.82</td>
<td>3.37</td>
<td>1.07</td>
</tr>
<tr>
<td>CE</td>
<td>2.26</td>
<td>2.12</td>
<td>1.60</td>
<td>1.51</td>
<td>0.80</td>
<td>1.17</td>
</tr>
</tbody>
</table>

In AP, the quality of the programs was slightly higher in the case of solo programmers (Quality=2.60) with respect to pair programmers (Quality=2.47). However, distributed pair programmers made fewer errors (CE=1.67) than solo programmers (CE=2.27) during the work process.

In AE, we can see a difference with the data collected in AP. In this case, the quality of the programs created by the distributed pair programmers is higher than that of solo programmers. In addition, in most cases the work process of pair programmers was better from a syntactic point of view, that is to say, they made fewer compilation errors.

The data for the distributed pair programmers in AP are better than for the solo programmers: the program quality is 0.34 points higher. In the work process of pair programmers fewer syntactical errors appeared, as in AE.

Table 4 shows the average values of the productivity variables for the working groups of AP, AE and AF. It can be seen in AP how the users who resolved the activity using PP needed more time (13.30 % longer). This is due to the number of interactions carried out to coordinate actions and to exchange messages, because the distributed pair programmers generally needed around 50 % fewer edition interactions with respect to solo programmers.

Table 4. Variables that evaluate Productivity in the three activities.

<table>
<thead>
<tr>
<th></th>
<th>AP (Palindrome)</th>
<th></th>
<th>AE (Euclidean)</th>
<th></th>
<th>AF (Fibonacci)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Solo programmers</td>
<td>Pair programmers</td>
<td>Solo programmers</td>
<td>Pair programmers</td>
<td>Solo programmers</td>
<td>Pair programmers</td>
</tr>
<tr>
<td></td>
<td>M</td>
<td>SD</td>
<td>M</td>
<td>SD</td>
<td>M</td>
<td>SD</td>
</tr>
<tr>
<td>Time (s)</td>
<td>2560.53</td>
<td>581.69</td>
<td>2901.26</td>
<td>469.84</td>
<td>2300.75</td>
<td>395.06</td>
</tr>
<tr>
<td>NI_edic</td>
<td>1052.26</td>
<td>293.81</td>
<td>446.86</td>
<td>134.71</td>
<td>843.00</td>
<td>257.54</td>
</tr>
<tr>
<td>NI_exec</td>
<td>5.20</td>
<td>5.86</td>
<td>1.73</td>
<td>2.04</td>
<td>8.81</td>
<td>10.36</td>
</tr>
<tr>
<td>NI_coor</td>
<td>-</td>
<td>-</td>
<td>18.40</td>
<td>12.93</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>NI_com</td>
<td>-</td>
<td>-</td>
<td>26.03</td>
<td>15.51</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

In AE we can see that solo programmers spent less time (Time=2300.75 s) than distributed pair programmers (Time=2495.62 s) as in AP. The pattern observed in AP whereby distributed pair programmers carried out around 50 % fewer edition interactions than the solo programmers is repeated in AE. Once again, the coordination and communication interactions involve additional work reflected by the time spent solving the activity.

In AF, the time spent by solo programmers in solving the activity is again lower than that which was spent by distributed programmers, but the effort to implement the final source code requires a higher number of edition interactions.

An overall analysis of the results obtained in the three activities comprising the experiment shows that when the distributed pair programmers have enough experience in the use of the groupware system (for which we consider the order in which the activities are approached) and when they work collaboratively with their partner, the quality of the programs built is better than those built by solo programmers (Fig. 4, left). This situation where experience is a factor which directly impacts the results of PP-based practices coincides with the conclusions obtained in similar studies (Lui & Chan, 2006).

On the other hand, an overview of the data collected from the three activities reveals that in all the activities, the number of mistakes (CE) made by distributed pair programmers is always lower than those made by solo programmers (Fig. 4, right). This result is also corroborated in some studies (McDowell et al., 2002; Müller, 2007), but in this case the novelty is the use of a groupware system.
The data related to productivity reflect a similar pattern of behaviour in each activity. The distributed pair programmers took more time to complete their activities (Fig. 5, left). This is due to the fact that pair programmers had to carry out additional work in order to coordinate and communicate in a distributed collaborative synchronous environment. However, this additional coordination and communication effort is offset by fewer edition interactions (Fig. 5, right) (which means less work on coding) in order to implement a higher quality program.

4.6 The influence of the collaborative programming process on the program

Taking advantage of the aforementioned functionalities for collaboration and interaction analysis incorporated in COLLECE, we quantified the degree to which a process indicator is associated with a solution indicator. For example, it would be interesting to determine whether the group has communicated and the individual participants have had positive attitudes when they have built correct and valid solutions together. We dealt with Q5 in this study.

Table 5 depicts the relations between a process indicator and a solution indicator, which are calculated, according to the algorithm of the aforementioned analysis framework, using a nonparametric statistic as the Kendall’s tau-b measure of association (Siegel & Castellan, 1988), since the indicators (variables), which take value in the set {1, 2, 3, 4, 5}, are nominal and ordinal. We focus on associations between variables with a significance level of $p \leq 0.05$. 
Table 5. Associations between process indicators and solution indicators.

<table>
<thead>
<tr>
<th>Process indicator</th>
<th>Solution indicator</th>
<th>tau-b</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>Communication</td>
<td>Well_Formed</td>
<td>0.202</td>
<td>0.049</td>
</tr>
<tr>
<td>Work</td>
<td>Validity</td>
<td>0.299</td>
<td>0.006</td>
</tr>
<tr>
<td>Work</td>
<td>Well_Formed</td>
<td>0.316</td>
<td>0.005</td>
</tr>
<tr>
<td>Velocity</td>
<td>Well_Formed</td>
<td>0.221</td>
<td>0.036</td>
</tr>
<tr>
<td>Experimentation_Willingness</td>
<td>Well_Formed</td>
<td>0.499</td>
<td>0.000</td>
</tr>
<tr>
<td>Experimentation_Willingness</td>
<td>Validity</td>
<td>0.507</td>
<td>0.000</td>
</tr>
<tr>
<td>Experimentation_Willingness</td>
<td>Testability</td>
<td>0.549</td>
<td>0.000</td>
</tr>
<tr>
<td>Experimentation_Willingness</td>
<td>Quality</td>
<td>0.406</td>
<td>0.000</td>
</tr>
<tr>
<td>Experimentation_Willingness</td>
<td>Accuracy</td>
<td>0.256</td>
<td>0.013</td>
</tr>
</tbody>
</table>

The Communication indicator is associated with the degree to which the program (built as a solution to the problem) is well-formed, that is to say, is syntactically and semantically correct. In other words, rich communication (the students exchange many messages through the chat) is a guarantee of well-formed programs. The reverse is also true: poor communication would imply not well-formed solutions. Similarly, the Work is positively correlated to Well_Formed and Validity, which means that the better the group work, the better the degree to which the solution is well-formed and the better is its validity. Validity here means the degree to which a solution satisfies the problem constraints. Examples of constraints are the obligation of using a recursive function for solving the problem or reading the date from the keyboard. Experimentation_Willingness is associated with a number of solution indicators. This way, a group’s willingness to experiment with the solution (by compilations and executions) would produce solutions with good validity, testability, quality, accuracy and degree of being well-formed; on the other hand, a poor group’s willingness would result in poor solutions. The Testability indicator measures the degree to which one can experiment easily and productively with a solution; in Programming, experimenting consists of executing the program and analyzing its outcome. The accuracy of a program is the degree to which the program correctly solves the proposed problem. Finally, Velocity is related with Well_Formed in the sense that the users who work quickly (which could be a skill of good programmers) usually built well-formed programs.

4.7 A study in a software factory

Indra\(^1\) is a global technology and innovation company on the cutting edge of high value-added solutions and services for several sectors. The global delivery model of Indra, based on a network of Software Labs, is made up of 20 centres and more than 33000 professionals throughout the world. One of the Indra Software Labs is located in Ciudad Real (Spain), which we refer to as ISL-CR. As a complement to the previous studies, and in order to check whether the results are applicable to industrial settings, we prepared a collaborative programming activity using COLLECE with employees of this software factory. We first met the general manager and two team managers working for the public administration sector were then selected. Fifteen programmers from their teams with homogeneous skills in Java programming were recruited for participation in the study.

The programming activity consisted of processing two XML files containing a CD catalogue, selecting discs from 1990 and later, and showing an ordered list of those with a price greater than $9. As additional technological requirements, the programmers had to use a SAX (Simple Api for XML) analyser and follow a 3-tier (presentation, logic and data) design pattern. As a constraint, the use of a list or vector data structure was required. This problem formulation was elaborated in cooperation with the team managers, and is part of a real project but simplified to be completed within a set time limit.

The programmers were randomly arranged in five pairs and five solo programmers, who focused on the activity during a 90-minute session. When the activity finished, two experienced computer science engineers external to the factory assessed different properties of the program and the overall work of the programmers by scoring them between 1 (very low) and 5 (very high). The results of this study are shown below in a similar structure as the studies in the academic setting and taking the answers for questions Q1-Q4 into consideration.

The structured chat was configured in this case to only support structured messages. The effect was that the programmers used the first message (“I think that...”) to say what they wanted as if it were a free-text chat, not observing the purpose of the structured messages. Only the message “Why...?” was used a significant but small number of times.

---

\(^1\) [http://www.indracompany.com/en](http://www.indracompany.com/en)
Regarding the coordination tools, again the number of disagreements was very low. However, in this activity not many proposals came without an answer: 26 %, 16 % and 32 % for turn requests, compilation proposals and executions proposals. At awareness level, the programmers evaluated the remote tele-pointer in the edition area (M=4.0) and editor’s scroll bar (M=3.7), and the session panel (M=3.8) as strengths of the awareness support, and the program name (M=3.4), the line number label (M=3.1) and again the lists of proposals (M=3.3) as inefficient awareness mechanisms.

We focus now on analysing the average quality, compilation errors, programming time, number of interactions and other new measures of the programs and programming tasks under study (Table 6), such as the number of compilations and the evaluation of the design to be in accordance with the 3-tier architecture. We designate A_SWF to the activity carried out in ISL-CR.

Table 6. Indicators that evaluate Quality, CE, Productivity and other measures in A_SWF.

<table>
<thead>
<tr>
<th></th>
<th>A_SWF</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Solo programmers</td>
<td>Pair programmers</td>
<td></td>
</tr>
<tr>
<td><strong>Quality</strong></td>
<td>2.80</td>
<td>0.64</td>
<td>3.00</td>
</tr>
<tr>
<td><strong>CE</strong></td>
<td>167.60</td>
<td>87.92</td>
<td>53.80</td>
</tr>
<tr>
<td><strong>Number of compilations (NC)</strong></td>
<td>36.40</td>
<td>4.60</td>
<td>8.80</td>
</tr>
<tr>
<td><strong>Time (s)</strong></td>
<td>5854.80</td>
<td>447.84</td>
<td>5811.20</td>
</tr>
<tr>
<td><strong>NI Edic</strong></td>
<td>2420.80</td>
<td>655.04</td>
<td>1388.00</td>
</tr>
<tr>
<td><strong>NIExec</strong></td>
<td>12.40</td>
<td>7.68</td>
<td>3.30</td>
</tr>
<tr>
<td><strong>NI Coor</strong></td>
<td>-</td>
<td>-</td>
<td>27.50</td>
</tr>
<tr>
<td><strong>NI Com</strong></td>
<td>-</td>
<td>-</td>
<td>32.30</td>
</tr>
<tr>
<td><strong>Accordance with the 3-tier architecture</strong></td>
<td>2.80</td>
<td>0.80</td>
<td>3.40</td>
</tr>
</tbody>
</table>

The quality of the programs of the pair programmers (Quality=3.00) was slightly better than that of solo programmers (Quality=2.80), in line with the results obtained in the academic setting. It needs to be clarified that most of the pairs and solo programmers did not solve the problem correctly, even when they used all the time available (90 minutes). The quality was computed as explained in Section 4.5.

The number of compilation errors (CE) is higher in this activity. This is due to the complexity of the problem: the manipulation of the XML files, the work with several java files and classes, the use of a specific design pattern, the need of using a list data structure, the sorting of the data and the time limit. The solo programmers made around three times the number of errors (CE=167.60) than that of pair programmers (CE=53.80) and also made many more compilations (NC=36.40), highlighting the benefits of working together to build programs with fewer errors and requiring fewer compilations. However, the average number of errors per compilation is smaller in solo programmers. Similarly to compilations, the solo programmers execute the program more times.

The solo programmers also carried out many more edition interactions (NI Edic=2420.80) than pair programmers (NI Edic=1388.00). This does not mean that their programs are bigger, but they write and delete more excerpts of source code. We have calculated the average size of the programs as the number of instructions plus the lines of code and the programs of the pairs are 13% smaller.

Finally, the experts evaluated (from 1 to 5) to which degree the programmers built their programs following a 3-tier architecture for software systems. The solo programmers received a score of 2.80 while the pair programmers scored 3.40. This result strengthens the result of the quality indicator, which is better for the pairs.

Fig. 6 summarises some of the aforementioned indicators, showing that pair programmers achieve better results than solo programmers in program quality, in following a 3-tier architecture, in the number of edition interactions and in the program size.
After the development of the activity, we met some of the programmers and the two team managers, who were present in the experiment, so that they could give us their view about COLLECE and its approach. They found COLLECE especially useful in tutoring, so that an expert programmer can monitor the work of novice programmers or beginners in a specific technology, by checking, for instance, that the programs accomplish the directives for source code specified in the project. In their opinion, the approach is applicable in projects where a high level of code supervision is required for multiple reasons: performance, security issues and client demand.

They found that the source code editor is a poor tool. They use the Eclipse platform and its powerful editor in their projects and it was a challenge for them to use the one of COLLECE. Similarly, the programmers of ISL-CR became accustomed to using free-text chats and audio conferencing to exchange opinions during programming. Therefore, they did not like the structured chat, which limits their communication needs. Regarding the coordination panels, they thought that the request of permission to compile is unnecessary; on the contrary, they missed similar protocols for storing programs in the repository or other actions requiring the agreement of the aforementioned expert programmer.

5 Discussion

In some way or another, the different questions stated can now be answered. Let’s start with the communication, coordination and awareness support of COLLECE.

Structured chat tools are frequently incorporated in synchronous collaborative systems. Despite providing some benefits in specific systems and situations, we did not find them particularly useful in our approach. The frequencies of both structured and reused messages were too low in the academic setting. This may have been due to us having included the possibility to write free-text messages in the chat, which were preferred by the users. Accordingly, in the industry setting we removed the functionality of free-text messages with the aim of analysing to what extent it would be advisable to have a chat with only structured messages, thus making the users focus on the programming task and advising them with the information contained in specific messages. Again, the structured messages were not used as expected, and the inclusion of an audio (and even video) channel should be further researched.

We found significant data about the coordination tools that show that students and industry programmers did not use them properly. The low number of disagreements suggests considering the use of other coordination models that do not require two possible answers to a proposal (OK / Not OK), such as a request-release model that assigns resources when requested if they are not in use, and that are released by the owner user when he/she wants. In the specific case of the edition turn, an interesting possibility in teaching-learning settings is the mediation of the system to assign the turn.

In collaborative systems, awareness support is a component of the system’s usability. Usability is related to awareness; our hypothesis is that by improving the awareness support the usability is increased. From the beginning we recognized the importance of awareness in the design of COLLECE. Thus, we incorporated a significant number of mechanisms in the system. However, the students participating in the experiments were more critical of the awareness mechanisms, as reflected in the questionnaires. Apart from anything else, they said that the system was easy to use and to learn, although during the experiments they had problems using specific components such as the coordination tools. The factory programmers were a little more positive about
awareness. Taking together the opinion of the two kinds of programmers, we highlight the remote user’s tele-
pointers in both the edition area and the editor’s scroll bar, and the session panel as the best mechanisms for
providing awareness information.

When analysing the data from the academic studies aimed at comparing pair and solo programmers, we
concluded that distributed pair programmers usually spend more time completing their activities because of the
time spent on communication and coordination interactions. However, in the case of distributed pair
programmers, there is a very significant reduction in the number of interactions required to edit the source code
of the program under construction. Secondly, the analysis of the programs built by distributed pair programmers
shows that they make fewer syntactical errors than solo programmers during the work process in all cases. This
result can be directly extrapolated to the industry setting. The global quality of the programs built was also
analysed, which led us to the conclusion that the quality of the programs built by distributed pair programmers is
higher than that of solo programmers, once the pair programmers have acquired enough experience in the use of
collaborative tools. This also took place with the professionals of ISL-CR.

The last study in the academic setting allowed us to identify significant associations between specific
behaviours when solving the programming problems and specific types of solutions. This is a very interesting
result for reflection for both teachers and students because positive behaviours are highlighted, and confirms the
added value of using groupware tools for programming. In general, as the study illustrates, to communicate, work
and be willing to experiment are programmers’ behaviours that guarantee the production of well-formed,
valid and accurate programs. However, it is surprising that the Coordination indicator is not associated with any
characteristic of the program (validity, accuracy, etc.).

5.1. Comparative evaluation

In Table 7 a comparison of the systems described in Section 2 is given, evaluating their support for
collaboration. Thus, in the table we collect the time-space classification of each system, according to the
classical taxonomy proposed by Johansen et al (1991), the support for communication, coordination and
awareness, the tools available in relation to collaborative programming, the support for the learning of
programming, and the support for collaboration analysis. We evaluate each characteristic on a scale from very
poor or inexistent support to very rich support.

The essential difference between collaborative systems and multi-user systems is that the former make
notifications to the users of the work carried out. This way, mechanisms for changes notification and awareness
have a central role in collaborative systems. Ellis et al. (1991) stated that there is not a clear line that easily
divides groupware and non-groupware systems, and that is more appropriate to classify systems in terms of a
spectrum in which the different systems can be placed. This spectrum is based on two dimensions: the common
task and shared environment dimensions. Thus, a system is more collaborative when scores are high in these
dimensions.

The common task dimension measures the degree to which the users jointly carry out a goal-oriented activity.
Systems that score low in this dimension are RECIPE and Kodingen, while DPE, Jazz Sangam, Gobby,
SubEthaEdit and COLLECE have a high degree.

The shared environment dimension refers to what extent the system provides information about the
participants, the state of the tasks and the social activity, mainly by means of awareness and notification
mechanisms. Accordingly, RECIPE, DPE, Jeco, Cloud9IDE and Kodingen have a low degree of shared
environment, and Gobby, SubEthaEdit and COLLECE scores high in this dimension, although it is worth noting
that Gobby and SubEthaEdit are systems that do not support collaborative programming.

We can conclude that COLLECE has a high degree in both dimensions and, consequently, is a complete
groupware system for collaborative programming, with full support for awareness, communication and
coordination. The system was conceived for teaching-learning purposes and, thus, it follows an approach based
on problem solving. Moreover, the system includes a subsystem for collaboration analysis, which is of interest
for evaluating collaborative work as well as collaborative learning.
Table 7. Comparative evaluation of different systems for CP (++): very rich support; +: rich support; =: not significant support; -: poor support; --: very poor or inexistent support)

<table>
<thead>
<tr>
<th>System</th>
<th>Time-space classification</th>
<th>Communication tools</th>
<th>Coordination tools</th>
<th>Awareness support</th>
<th>Collaborative programming</th>
<th>Programming learning</th>
<th>Collaboration analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>RECIPE</td>
<td>Synchronous distributed</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>++ Collaborative coding, compilation, running, debugging</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>DPE</td>
<td>Synchronous distributed</td>
<td>++ Text-based, audio and video</td>
<td>--</td>
<td>--</td>
<td>++ Collaborative coding, compilation, running</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>Jazz Sangam</td>
<td>Synchronous distributed</td>
<td>+ Chat</td>
<td>--</td>
<td>= Changes notification</td>
<td>Non-synchronized collaborative coding</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>Gobby</td>
<td>Synchronous distributed</td>
<td>+ Chat</td>
<td>--</td>
<td>+ Member panel, tele-cursors, code user’s code colouring</td>
<td>Program editing</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>SubEthaEdit</td>
<td>Synchronous distributed</td>
<td>+ Chat</td>
<td>+ Invitation and access control tools</td>
<td>++ Member panel, tele-cursors, group scrollbar, user’s code colouring</td>
<td>Program editing</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>JeCo</td>
<td>Synchronous/asynchronous distributed</td>
<td>+ Chat</td>
<td>--</td>
<td>--</td>
<td>- Users chatting</td>
<td>Program sharing, execution animation</td>
<td>+ Collaborative learning based on program animation</td>
</tr>
<tr>
<td>Cloud9IDE</td>
<td>Asynchronous distributed</td>
<td>+ Chat</td>
<td>--</td>
<td>--</td>
<td>Users chatting</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>Kodingen</td>
<td>Asynchronous distributed</td>
<td>= Forums</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>COLLECE</td>
<td>Synchronous distributed</td>
<td>+ Structured chat</td>
<td>++ Coordination tools</td>
<td>++ Member panel, tele-cursors, group scrollbar, user’s and group’s state labels</td>
<td>++ Collaborative coding, compilation, running</td>
<td>++ Collaborative learning in a problem solving approach</td>
<td>++ Process and product analysis</td>
</tr>
</tbody>
</table>

6 Conclusions

In this article we have approached Collaborative Programming. We firstly presented COLLECE, a synchronous distributed collaborative system we developed for supporting programming tasks (edit, compile and run).

COLLECE has been put to the test in teaching-learning and industry settings through some studies. Some of the specific tools incorporated or some design decisions have not provided the expected benefits. A structured chat does not appear to be the best choice as a communication tool. Structured messages or the reuse facility are not very popular with the programmers. However, we think that the set of ten messages considered constitutes a good representation of the range of possible messages to be exchanged among programmers.
A voting or coordination tool is required to synchronize the joint work of the users. Our proposal-based coordination tools never showed themselves to be effective, and their characteristic of supporting multiple proposals seemed complex to operate for the users.

Considering the comments and results above, we suggest providing groupware tools for CP with:

- Coordination tools based on a request-release policy, including semaphores and suitable awareness mechanisms to give information about the request and release operations (the students recommended better highlighting the proposals of the colleagues).
- A free-text chat or, instead, an audio tool. Only in the case that it would be interesting to categorize the communication for subsequent analysis would it be recommendable to use a structured chat although, in this case, always without the choice of using free-text messages.
- A wide awareness support with a session panel, tele-pointers, configurable beeps, state labels and many other mechanisms so that the users are easily aware of who they are working with and what, where and when they are doing, among other things.

Thanks to the COLLECE analysis subsystem, we have been able to study some effects of doing CP with a groupware tool. Working in pairs has a positive effect on the quality of the programs under construction, and the communication among programmers and an intensive collaborative work have positive effects on the program produced.

Finally, it is necessary to reflect on the use of the COLLECE system for professional purposes. We believe that its general approach is suitable for industry since it can support XP and particularly DPP and CP. The COLLECE’s approach could be used in real projects for program editing, shared error correction, discussion of solutions or explanation of programs. Such a system would have an impact not only on the immediate productivity or quality of programming but also on the control and analysis of the process.

The team managers of ISL-CR found the system quite useful to deal with their problems of technological niches in the future. Software companies need high levels of deep knowledge in specific technologies. For one of these expert programmers, it would be easier to share his/her knowledge using collaborative systems such as COLLECE, as well as to supervise critical issues of the software and its programming cycle and to analyse the programming performance. However, it must be taken into account that in software factories distributed synchronous work could be a limitation when it was not possible for all the participants to work at the same time due to their different working conditions due to their different countries/regions, timetables, etc.

COLLECE is presently being used by university Computer Science students enrolled in the subject matters of Programming Fundamentals and Data Structures for finding evidence that confirms the benefits of its awareness support and specifically for exploring radar views as a means of improving the understanding of the users’ work.

All the lessons learned are obviously a starting point for the new versions of the system. Nevertheless, the future of COLLECE is clearly on the Eclipse platform (Eclipse, 2012). We are working in transforming the different collaboration tools in plug-ins, using ECF (Eclipse Communication Framework) technology, and the new version of the system will also work as a plug-in (see Bravo et al., 2009). In the questionnaires, the students recommended better source code editor; they will have it with Eclipse. Some of them also suggested the availability of a shared editor allowing multiple programmers to work on the source code simultaneously. However, this would require a different approach that will necessitate further work.

7 Acknowledgements

This research is supported by the Comunidad Autónoma de Castilla-La Mancha (Spain) in the PAC07-0020-5702 project and by the Ministerio de Educación y Ciencia (Spain) under project TIN2011-29542-C02-02. We gratefully acknowledge the students and teachers of the University of Castilla – La Mancha and the team managers and programmers of Indra Software Labs (Ciudad Real, Spain) for their participation in the various studies.

8 References


Crescencio Bravo received his MSc in Computer Science (1996) from Universidad de Sevilla (Spain) and his PhD (2002) from the Universidad de Castilla – La Mancha (Spain). He joined the Computer Science Engineering Faculty of the Universidad de Castilla – La Mancha in 1998. His research interests include computer-support for Collaborative Learning and Cooperative Work and Computer-Human Interaction.

Rafael Duque received his MSc in Computer Science (2005) and his PhD (2010) from the Universidad de Castilla-La Mancha (Spain). He has been an assistant professor at that University since 2010, and is now a professor at the University of Cantabria (Spain). His research interests include CSCW/CSCL and analysis of collaboration and interaction in groupware systems.

Jesús Gallardo received his BS, MS and PhD from the Universidad de Castilla-La Mancha (Spain). He has been an assistant professor at that University since 2011, and is now a professor at the Universidad de Zaragoza (Spain). He has been a member of the CHICO research group at the Universidad de Castilla-La Mancha since 2006. His research interests include CSCW, groupware development and the application of model-driven engineering to such fields.