A Survey of Collaborative Tools in Software Development

Anita Sarma
Institute for Software Research
Donald Bren School of Information and Computer Sciences
University of California, Irvine
asarma@ics.uci.edu

ISR Technical Report # UCI-ISR-05-3

March 22, 2005

1 Introduction

Collaboration is at the heart of software development. Virtually all software development requires collaboration among developers within and outside their project teams, to achieve a common objective. It has in fact been shown that about 70% of a software engineer's time is spent on collaborative activities [219]. Indeed, collaboration in software development has been studied by researchers in the fields of Software Engineering and Computer Supported Cooperative Work (CSCW) since the 1980s and has produced a wide range of collaborative tools.

Enabling software developers to collaborate effectively and effortlessly is a difficult task. The collaboration needs of the team depend to a large extent on environmental factors such as, the organizational structure of the team, the domain for which the software is produced, the product structure, and individual team members. Accordingly, research in collaborative development has produced a host of tools, each typically focussing on a different aspect of collaboration. Most teams have their favorite repertoire of tools that has been built from historical use. These tools may not always be the best suited for the team, but the team still uses them nevertheless as the inertia and cost of trying out new tools surpasses the benefits.

A number of classification frameworks exist that can be used to classify collaborative tools. In addition to placing the various tools in context, developers can use these frameworks to select the right mix of tools fit for their needs. Each classification framework has a different focus: some provide a detailed taxonomy to compare tools in a particular area [44], some classify tools based on the functionality of the tools [93], some classify tools based on the high-level approach to collaboration that the tools take [218], and so on. However, currently no framework exists that classifies tools based on the user effort required to collaborate effectively. This however is also a critical component in choosing the "right" set of tools for a team.

In this survey, we take a look at collaborative tools from the perspective of user effort. For the purposes of this paper, we define user effort as the time spent in setting up the tools, monitoring the tools, and interpreting the information from the tools. While we cannot quantify the efforts required of each tool in detail, it is clear that there is a natural ordering among different groups of tools. We propose a framework that identifies these groups and highlights this ordering. Based on this framework, our survey organizes the individual tools into tiers.

Our framework is in the form of a pyramid consisting of five vertical layers and three horizontal strands. The five layers in the pyramid are: (1) `functional`, (2) `defined`, (3) `proactive`, (4) `passive`, and (5) `seamless`. Tools that are at a higher layer in the pyramid provide more sophisticated automated support, thereby reducing the user effort required in collaborating. Each level, thus, represents an improvement in the way a user is supported in their day-to-day collaborative activities. The three strands in the pyramid are: *communication*, *artifact management*, and *task management*. These three dimensions, we believe, are critical needs crosscutting all aspects of collaboration.

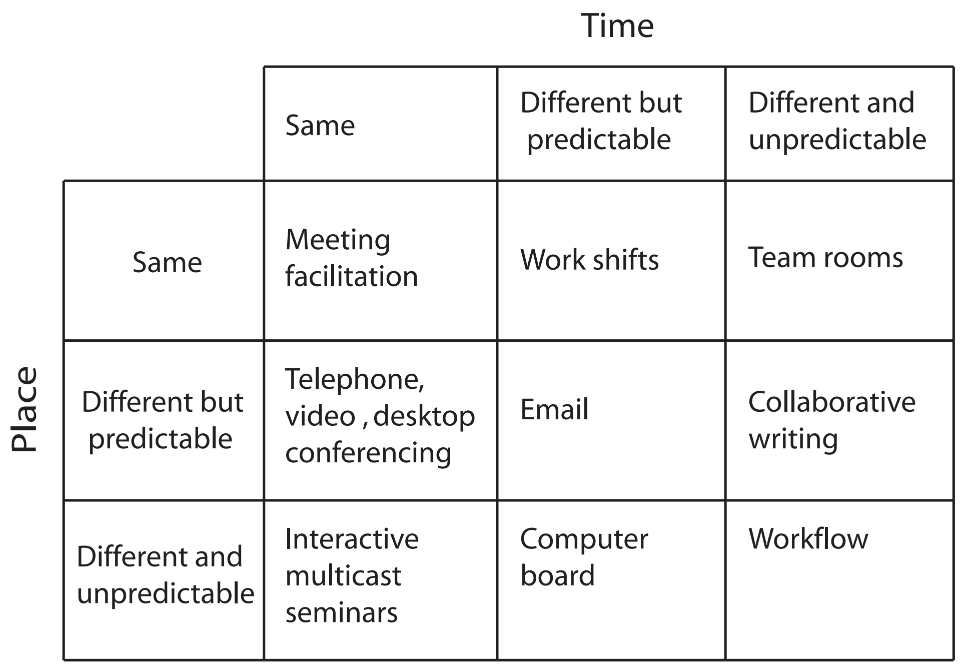
The remainder of this paper is organized as follows. In Section 2, we discuss a few existing representative classification frameworks. Section 3 presents the details of our framework. The five layers of the pyramid are discussed in Sections 4 through 8, with the `functional` layer discussed in Section 4; the `defined` layer discussed in Section 5; the `proactive` layer discussed in Section 6; the `passive` layer discussed in Section 7; and the `seamless` layer discussed in Section 8. We present our observations in Section 9 and conclude in Section 10.

2 Related Work

Group collaboration among software developers has been studied by researchers in software engineering and CSCW since the 1980s. Research in these areas has produced a wide range of collaborative tools (e.g., tools that support communication, task allocation, decision making). To better understand the functionalities of these tools and how they compare with each other, a number of classification frameworks have been proposed by others. In this section, we take a brief look at some of the representative frameworks.

2.1 Space and Time Categorization

Grudin modified the DeSanctis and Gallupe space and time classification framework [58] to create a 3x3 matrix (Figure 1). The original framework was a 2x2 matrix that classified tools based on the temporality and the location of the teams (e.g., does the tool support asynchronous communication for collocated or distributed teams). Grudin improved DeSanctis and Gallupe's framework by further distinguishing the tools based on the predictability of the actions that they support [93]. Grudin's framework, then, is a 3x3 matrix that classifies tools based on the temporality of activities, location of the teams, and the predictability of the actions.



		Time		
		Same	Different but predictable	Different and unpredictable
Place	Same	Meeting facilitation	Work shifts	Team rooms
	Different but predictable	Telephone, video , desktop conferencing	Email	Collaborative writing
	Different and unpredictable	Interactive multicast seminars	Computer board	Workflow

Figure 1: Space and Time Categorization [93].

Figure 1 represents the space and time framework, each cell illustrating some representative applications for the particular space and time categorization. The rows in the matrix represent whether applications support collocated or distributed teams. The top, the middle, and the bottom row of the matrix represent activities that can be carried out at a single place, in several places that are known to the participants (email exchanges), and in numerous places not all of which are known to participants (message posted in a newsgroup), respectively. The columns in the matrix depict whether applications support synchronous or asynchronous collaboration. The left, the middle, and the right column of the matrix represent activities that can be carried out "in real time" (a meeting), at different times that are highly predictable (when one sends an email to a colleague expecting it to be read within a day or so), and at different times that are unpredictable (open-ended collaborative writing projects), respectively.

2.2 Workflow

Research in workflow advocates the use of models and systems to define the way an organization performs work [165]. A workflow system is based on a workflow model that divides the overall work procedure of an organization into discrete steps with explicit specifications of how a unit of work flows through the different steps. Workflow languages (e.g., E-net modelling [164], Information-control net (ICN) [70], PIF [134]) implement the workflow model by providing constructs for defining each

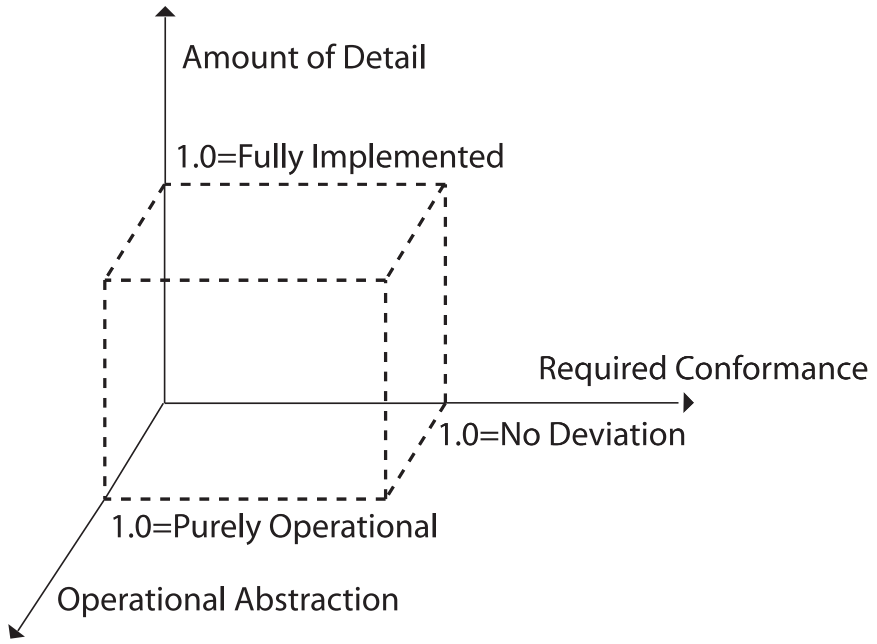


Figure 2: The Model Domain Space [165].

step (computation language) and how the unit of work flows between the steps (coordination language).

Gary Nutt [165] created a three dimensional model domain space that is based on how a workflow model models a work procedure. As illustrated in Figure 2, the three dimensions in the model space are: (1) x axis − the amount of conformance that is required by the organization for which the process is a model, (2) y axis − the level of detail of the description of the process, and (3) z axis − the operational nature of the model (whether the model describes how the process works rather than what is required from it). In this domain, models that represent only structured or explicit work [191, 31] are in the sub-space approximating $x \rightarrow 1$, $y \rightarrow 1$, and $z \rightarrow 1$, whereas systems intended to address unstructured work belong to a subspace where $x \rightarrow 0$, $y \rightarrow 0$, and $z \rightarrow 0$. All other process types fit in elsewhere in the 3D space (e.g., descriptive and analytic workflow models [70, 73] can be placed in the plane defined by $0 \leq x \leq 1$, $0 \leq y \leq 1$, and $z=1$; conventional workflow enactment systems [31, 155, 146] can be characterized by the line segment $x=1$, $y \rightarrow 1$, and $z=1$).

2.3 Interdisciplinary Theory of Coordination

Malone and Crowston [141] use coordination theory to investigate how people in other disciplines manage the dependencies that arise in collaboration. They define coordination as managing dependencies between activities and provide a framework for classifying collaborative tools by identifying the coordination processes the tools use to manage the different kinds of dependencies. Malone and Crowston take a broad outlook and study coordination at an interdisciplinary level, with the objective of finding similarities in concepts and processes in different disciplines (e.g., economics, computer science, organization theory). These similarities would then allow ideas to be transported across the discipline boundaries, which in turn would help in enriching the existing processes in each discipline. For example, the way organization theory handles resource allocation (hierarchial resource allocation, where managers at each level decide how the resources are allocated) can be modelled to handle resource allocation in software development teams.

Malone and Crowston identified the coordination processes used by different disciplines to manage dependencies between activities. They then created a taxonomy of collaborative tools, illustrated in Table 1, based on processes that the tools support in software development. Rows 1 through 4 in the table identify processes for managing typical dependencies between developers and resources in a software development team. For example, task assignment needs to ensure that tasks are assigned to developers who have the required expertise (row 1), the interdependencies between tasks should be considered while creating tasks and subtasks (row 4), and so on. In addition to the processes for managing the coordination dependencies, communication and group decision making play an important role in collaboration and have been added to the framework (rows 5 and 6). For instance, in case of

Process	Example systems
Managing shared resources (task assignment and prioritization)	Coordinator [225], Information Lens [142]
Managing producer/consumer relationships (sequencing prerequisites)	Polymer [45]
Managing simultaneity constraints (synchronizing)	Meeting scheduling tools [14]
Managing task/ subtask relationship (goal decomposition)	Polymer [45]
Group decision making	gIBIS [41], Sibyl [133], electronic meeting rooms [57, 58]
Communication	Electronic mail, computer conferencing (e.g., Lotus, 1989) electronic meeting rooms [57, 58], Information lens [142], collaborative authoring tools [78, 71]

Table 1: A Taxonomy of Collaborative Tools Based on the Process They Support [141].

shared resources, a group needs to *decide* how to allocate the resources; in managing task / subtask dependencies, a group must *decide* how to segment tasks; and so on.

2.4 Formal versus Informal Approach to Collaboration

The formal versus informal coordination model [218], illustrated in Figure 3, classifies tools based on their high level approach to collaboration. This framework classifies tools into three categories, namely tools that follow formal process-based approaches, tools that provide informal awareness-based coordination support, and tools that combine these two approaches.

Tools that follow formal process-based approach provide coordination by breaking the entire software development effort into discrete steps. At the end of each step, developers are required to synchronize their work to maintain consistency. In this approach the tool is responsible for the coordination protocols that the developers are required to follow (e.g., the check-in/check-out model of SCM systems, workflow systems). The chief advantages of the formal process-based approach are, that they are group centric and scalable. Their drawback is that the insulation provided by the workspaces quickly turns into isolation, as developers are not aware of the activities of others that may affect their work.

Tools that follow the informal awareness-based approach provide coordination by

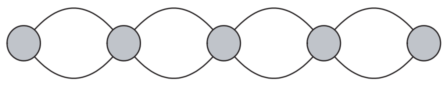
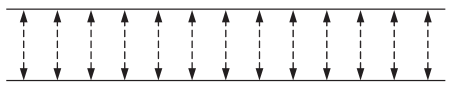
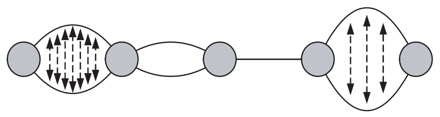
	Conceptual Visualization	*Strengths*	*Weaknesses*
Formal process based coordi-nation		Scalable; Control; Insulation from other activities; Group-centri	Resynchronization problems; Insulation becomes isolation
Informal awareness based coordination		Flexible; Promotes synergy; Raises awareness; User-centric	Not scalable; Requires extensive human intermediation
Continuous coordination		Expected to be the strengths of both formal and informal coordination	To be discovered by the current research

Figure 3: Formal vs. Informal Approach [218].

explicitly or implicitly disseminating information (e.g., artifacts that have changed, activities of other developers) to the members of a team. It is the responsibility of the members of the team to interpret this information and pro-actively self-coordinate. Usually this leads to some kind of informal agreement according to which developers plan their activities. While this approach is user-centric and gives the users control and flexibility in defining their coordination protocols, it requires extensive human intermediation and is not scalable.

Neither the formal process-based approach nor the informal awareness-based approach is completely satisfactory. The weaknesses further compound when confronted with the reality of coordination needs in distributed settings. To overcome this problem, van der Hoek et al. [218] propose an integrated approach, called continuous coordination, that supports collaborative work by combining the strengths of both the formal and informal coordination approaches. Applications that follow this approach would be highly flexible and be able to continuously adapt their coordination support to the needs of the task at hand. Research in continuous coordination, however is new and has not yet produced any prototype applications, therefore the strengths in the table are expected and the weaknesses have still to be discovered.

2.5 Summary

Each of the frameworks discussed above approaches collaboration from a different perspective: Grudin classifies collaboration tools based on whether they can support synchronous or asynchronous communication for distributed or collocated teams; Nutt classifies workflow systems based on the characteristics of the underlying workflow model; Malone and Crowston focus on coordination processes that can be shared between multiple disciplines; and van der Hoek et al. classify applications based on their high-level approach to collaboration.

What is interesting to observe, however, is that none of these frameworks classify tools based on the user's effort required to collaborate effectively. In fact, all of the frameworks look at coordination tools from a functionality point of view. For example, the space and time categorization tells us which tools can support collaboration at real time or which tools can support distributed collaboration; the formal versus informal framework informs us which tools follow the formal process-based approach and which the informal awareness-based approach. These frameworks do not specify the expected kind of user effort that is required in using a particular kind of tool for collaboration. In this survey, we introduce a new classification framework that revolves around different classes of user effort required to collaborate effectively. We discuss our framework in the next section.

3 Classification Framework

Our classification framework is based on two principal characteristics of collaborative tools, namely: (1) the level of coordination support provided to users, and (2) the focus of a tool on one of the three essential elements of collaboration: communication, artifact management, and task management. Our classification framework combines these two characteristics to form a pyramid, as illustrated in Figure 4. We distinguish five levels of coordination support and three different foci of tools. The five levels of coordination support are organized vertically and we call them "layers" from here on. The foci of the tools are organized horizontally, and we call them "strands" from now on.

Layers are based on the "level of coordination support" provided by the tools. By this, we mean that tools at a higher layer provide better automated support (and therefore less user effort) than tools in layers below it. Each layer, thus, represents an improvement in the way a user is supported in their day-to-day collaborative activities. We identify five layers: (1) `functional`, (2) `defined`, (3) `proactive`, (4) `passive`, and (5) `seamless`.

Strands in the pyramid represent the three elements that we consider intrinsic to collaboration, namely: (1) communication among team members, (2) artifact management, and (3) task management. Research in collaboration has typically focused on one of the strands at a time. By including all three strands in our pyramid, we are able to create a common ground for classifying the tools that stem from different approaches.

As we move up the layers in the pyramid, the level of support provided by collaboration tools increases while at the same time the user effort in enabling collaboration decreases. Tools at the higher layers in the pyramid provide advanced automated support to users, can handle large and complex team structures, and reduce action and information overload on users as compared to tools at the lower layers. We envision that the layers are not isolated, but functionally build upon each other. Layers become stronger than when functioning in isolation, i.e., when combined with the functionalities of the layers below it.

As we progress up the pyramid, the distinction among the strands becomes increasingly blurred. The communication strand slowly but surely moves over into the territory of the artifact management strand. So does the task management strand, until all three strands merge in the highest layer of the pyramid. This merging represents insights from ethnographic studies [92, 124] which found that, to coordinate their activities, users combine different cues and resources from the environment in which they operate. Researchers and tool builders alike have recognized this, and have broadened their focus to encompass support for more than one strand in their tools.

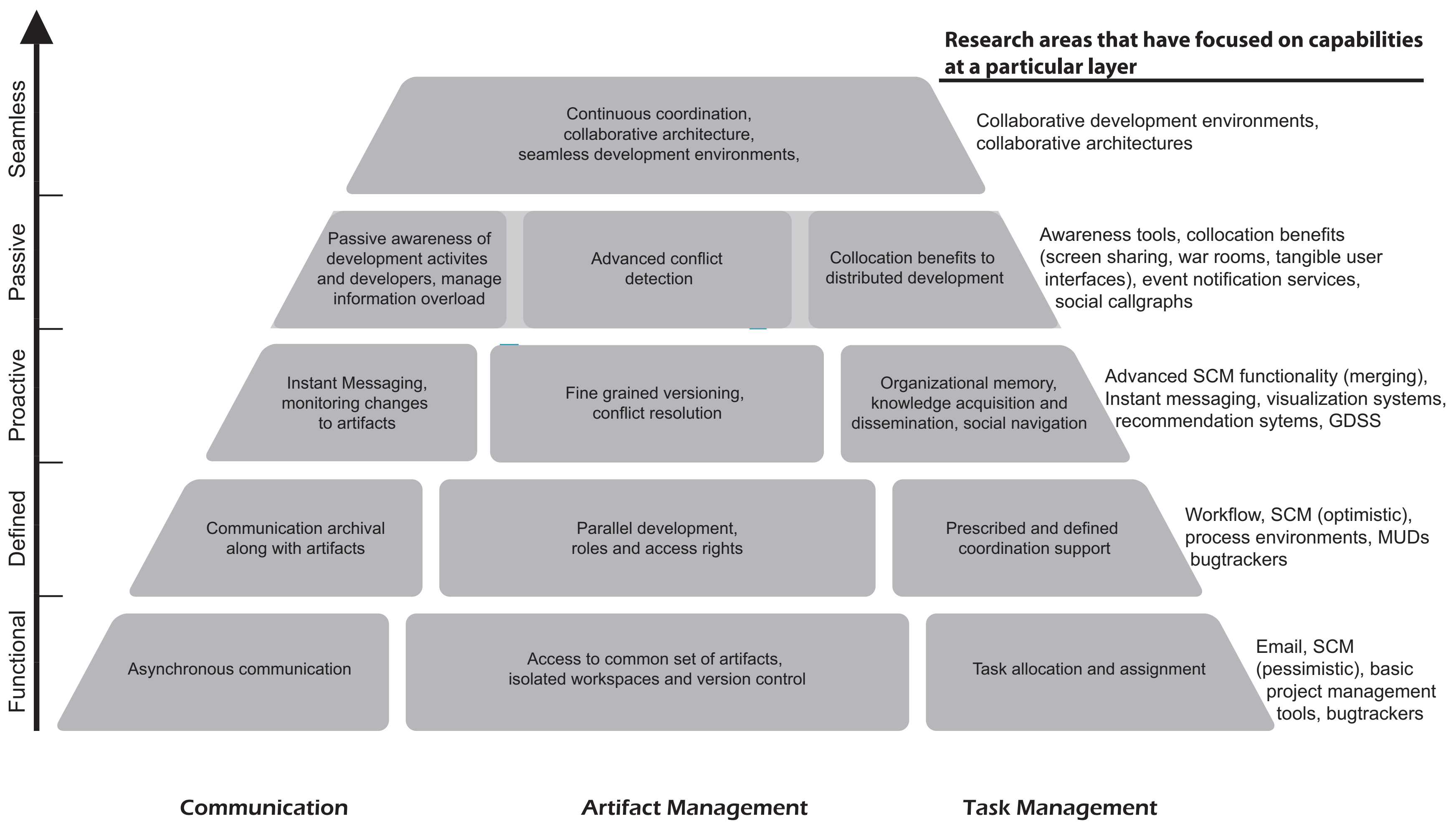

Research areas that have focused on capabilities at a particular layer
Seamless
Passive
Proactive
Defined
Functional
Continuous coordination, collaborative architecture, seamless development environments,
Collaborative development environments, collaborative architectures
Passive awareness of development activites and developers, manage information overload
Advanced conflict detection
Collocation benefits to distributed development
Awareness tools, collocation benefits (screen sharing, war rooms, tangible user interfaces), event notification services, social callgraphs
Instant Messaging, monitoring changes to artifacts
Fine grained versioning, conflict resolution
Organizational memory, knowledge acquisition and dissemination, social navigation
Advanced SCM functionality (merging), Instant messaging, visualization systems, recommendation sytems, GDSS
Communication archival along with artifacts
Parallel development, roles and access rights
Prescribed and defined coordination support
Workflow, SCM (optimistic), process environments, MUDs bugtrackers
Asynchronous communication
Access to common set of artifacts, isolated workspaces and version control
Task allocation and assignment
Email, SCM (pessimistic), basic project management tools, bugtrackers
Communication
Artifact Management
Task Management

Figure 4: Classification Framework.

At this point, the pyramid is not complete. We have left the top of the pyramid open to signify further research. We do not know if the `seamless` layer will be the last layer in the pyramid or if it will split into additional layers. This is subject to future research as we have barely begun to scratch the surface of this layer.

3.1 Layers

Tools in the `functional` layer enable collaborative development, but do so with minimal technical support. Development teams at this level are small and work with the bare minimum in tool support; developers can get by in collaboration, but much manual effort is still required. For example, generally tools at this level allow different developers to access the same set of artifacts or communicate using email, but developers at this level are chiefly responsible for the actual coordination activity of who changed which artifacts and at what time. Teams relying on tools at this level depend on the developers' knowledge of the product structure, of which developer has been and is currently working on which changes, and of how the various changes relate to each other.

As teams become larger in size, the bare bones coordination support offered at the `functional` layer is insufficient. Developers in large teams often must make changes to the same artifacts in parallel, because it is more difficult to make non-overlapping task allocations in large teams. Tools at the `defined` layer provide exactly this kind of support by guiding users with a well-defined set of prescribed steps. For instance, tools in office automation or workflow help divide the development process into discrete steps, help specify which developer should change which artifacts, and help in directing what the changes to an artifact should be. These systems are "good", as developers can now rely on the system to support their coordination activities, (e.g., the system automatically routes the final checked-in code to the "test" team), but at the same time they are "bad" as developers have to strictly follow the prescribed steps and have little flexibility.

Tools at the `proactive` layer allow developers more control over the coordination steps. Specifically, developers can be proactive in obtaining the information with which they can fine tune the coordination steps to suit their project. For example, using tools such as CVS-watch [20] and Coven [37], users can monitor changes to artifacts of interest in order to avoid potential conflicts. Once the tools inform users of such changes, they can either merge the changes using automated merge tools [151] or not take any action at the moment. Developers with tools at this layer clearly have control over the overall coordination process, but in order to achieve this flexibility they have to be actively involved in the process.

Tools at the `passive` layer reduce the effort required to obtain the information necessary to tailor their development process. At this layer, users can configure the tools to view relevant, timely information. Some of the tools provide default modes requiring little to no configuration, yet provide important information at opportune times. Typically, the information is peripherally displayed in an unobtrusive manner, to create a subtle "awareness" of what activities are occurring in parallel. For instance, tools such as Palantír [194], JAZZ [36], and others [97, 109, 170] provide passive workspace awareness by displaying which developers are changing which artifacts. Developers can now realize which changes would affect them and with whom they should coordinate their activities. Tools at the `passive` layer allow developers to concentrate on their current development activity, knowing that they will be notified if there are pertinent changes. However, a drawback of the tools at this layer is that users have to typically run multiple applications, because each of these tools provides a different set of information (process support mechanism, awareness mechanism, conflict resolution mechanism, and so on) and each typically is a stand alone tool.

At the final layer in our pyramid, the `seamless` layer, developers no longer have to switch contexts while accessing different sets of information, as tools at this layer are built such that they can be integrated seamlessly to provide continuous coordination [218]. Research in environments such as Oz [19] and Serendipity [95] are investigating exactly these kinds of integration, providing a single tool for seamless task management, artifact management and communication (see Section 3.2 for each of these strands). Research in this area is still very much in progress, but the hope is that eventually there will be environments that allow developers to seamlessly use all necessary collaboration facilities within their development environment.

As we move up the hierarchy, tools change from supporting the minimal needs that barely enable coordination to providing full-featured, seamless coordination support that places a minimal burden onto the user. We note that this change has gradually occurred over time, but that layer development has not been strictly historical. Sometimes, research has jumped a level (as in the case of Portholes [64], see section 7.1.1) and sometimes research has returned to a lower level to spark evolution at a higher level (as in research in social navigation, see section 6.3.3, where research on email allowed subsequent development of recommendation systems [163, 166]).

We recognize that environmental factors impact research in collaboration. For example, a change in the problem domain or the team structure often creates different coordination needs. We consider two environmental factors, namely product structure and organizational structure, as factors that have shaped the collaboration capabilities of tools at each layer. An example of a change in product structure leading to new capabilities was the fact that programming evolved to include interfaces (promoting stronger separation of concerns), which has allowed easier impact analysis. An example of a change in organizational structure was the emergence of distributed teams that are geographically separated. This created a need for workspace and presence awareness tools that operate across time and distances. We describe the effect of these factors and the context as we introduce each layer.

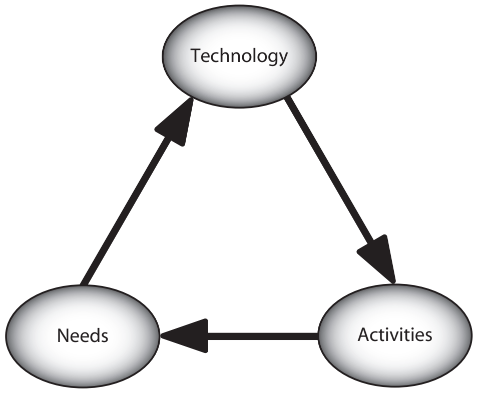


Figure 5: Three level interaction.

While creating the hierarchy of layers, we also consider experiences from tool usage that have influenced the needs for the tools at each layer. For example, when users become experienced with their current tools they feel the need for advanced features. By the same token, the approach taken by the tools in solving a particular need creates a newer set of needs that have to be addressed. This is illustrated by the relationship between needs, technology, and usage of technology in Figure 5[1] . Typically, researchers identify a set of needs and then create the technology that addresses those needs. Oftentimes the technological solution or the way users interact with the technology creates a newer set of needs. A new generation of tools is then created to address these additional set of needs, continuing the cycle.

For example, to address the coordination needs of a large software development team, SCM tools created workspaces [20, 44, 51]. These workspaces allow developers to make changes in private, and after completing their tasks, synchronize their changes with the repository. On the one hand, the isolation created by workspaces is “good”, because it allows developers to make changes without being affected by others. On the other hand, this isolation is “bad”, since it inhibits developers from being aware of their co-developers’ activities. This was confirmed in a study of a software development organization, which revealed that developers planned their activities based on the activities of their colleagues [89, 90]. The users gained this information by querying the SCM repository to find out which artifacts had been checked out by whom. Another study [55] investigated the informal convention of using email that augmented the formal development process already in use. These activities around the repository clearly imply the need for workspace awareness. Such research is now in progress [194, 36, 157].

[1]Ddiscussion with Paul Dourish, professor, Informatics, UCI, Nov 2004.

3.2 Strands

The Cambridge dictionary defines collaboration as: "two or more people working together to create or achieve the same thing" [179]. In the domain of software development, collaboration involves the coordination of developers to create and maintain software artifacts. Broadly speaking, collaborative support in software tools must involve support for: (1) communication among team members, (2) artifact management, and (3) task management. We use these three aspects, which we call "strands", throughout the remainder of the paper as vertical slices of the pyramid.

Communication: Teams use communication to keep each other up to date with the tasks that have been completed, to communicate changes in schedules, to ask questions or provide solutions to problems, to schedule meetings, and numerous other purposes. When an organization does not have a good communication infrastructure, developers might hesitate to ask questions to colleagues whom they do not know, not communicate changes to an artifact that others might depend on, not be able to keep track of past communication, and so on. This lack of good communication in teams often leads to project delays.

Artifact Management: Artifacts in software systems are highly interdependent on each other. This interdependence implies that changes to software artifacts, be it code or other artifacts produced during the life cycle (e.g., a requirements specification, a design), need to be managed to ensure the correct behavior of the program. In large projects, it becomes impossible for developers to manage changes to software artifacts on their own. Developers therefore increasingly depend on tool support to version their code, coordinate parallel development, resolve conflicts, and integrate code, among other things.

We place artifact management as the central strand in our framework, since collaboration in software engineering primarily involves coordination of development activities around a set of software artifacts. As we move up the pyramid, we note that the distinction among the strands becomes blurred and that the other two strands slowly but surely intrude into the territory of artifact management. As we stated before, this is an explicit choice, namely to illustrate the change in research focus wherein research is increasingly drawing on all three strands to provide tool support. For example, in the `proactive` layer of the pyramid, developers can monitor changes to artifacts. In the event of a change, the system notifies developers who had registered interest on the artifact. Here, we see that "when" the communication is triggered or "who" the communication is sent to, depends on the artifacts in question.

Task Management: Task management in software development involves, among others, decomposing the project into smaller units, identifying developers with expertise, assigning tasks to developers, and creating a development schedule. In large projects, task management is a time-consuming and difficult task. Managers of large teams typically rely on project management tools to assign and monitor tasks. For instance, workflow systems split the development process into discrete steps and specify which developer should make what change to which artifact. Task allocation in large projects is seldom clean and managers rely on automated tool support to continuously coordinate the activities of developers with overlapping task responsibilities. For example, concurrent changes to the same artifact have to merged before they can be placed in the repository.

In the following sections, we discuss each layer and each of the three strands crosscutting a given layer. We begin with the `functional` layer and its communication, artifact management, and task management strands. This pattern is repeated through the first four layers of the pyramid (Sections 4 through 7).

4 The Functional Support Layer

The `functional` layer is the first layer in the pyramid and forms its base. It depicts the basic level of automated collaboration support provided by tools. Previous to the tools in this layer, collaboration was managed manually. The tools at this layer were promoted by the type of organization and product structure prevalent at that time. From the organizational and product structure point of view, tools primarily needed to support collaboration for small, “flat” teams working with structured programming languages.

Based on the aforementioned criteria, tools at this layer provide basic automated support for developers to communicate with each other, access and modify a common set of artifacts, and enable managers to allocate and monitor tasks. These tools form the transition from “no automated support” for collaboration to the “minimal automated support” necessary to allow a team to function.

4.1 Communication

Communication is one of the key factors in coordinating the activities of team members and has a big impact on how successful the team can be [123, 33]. Traditionally, face-to-face meetings were the typical means of communication in a team, with reports and memos being used for archival purposes.

As personal computers became popular, face-to-face communication was increasingly replaced by electronic communication, primarily email. Email was widely adopted as it allowed developers to asynchronously communicate over distances, refer to previous replies, reply to a group, record communication with little extra effort, and so on. Email has largely become popular because of its low learning curve and the ability to resolve problems without having to schedule meetings. Currently, there are a number of email clients, both commercial and open source [104, 142].

In addition to email, the open source community relies heavily on news groups and discussion forums for their communication needs. While email is more directive and chiefly used for point-to-point communications, news groups and discussion forums are more open and allow interested people to subscribe to the list [26, 66]. Discussions in these forums usually start when someone in the list posts a question or an interesting solution, that is then followed by discussion among other subscribers. Often times, complex solutions are attained in these discussion forums without members ever meeting face-to-face or sending personal email.

The open source community is an excellent example of a software development community that primarily depends on the tools in the `functional` layer for their collaboration support. Most of the team members never even meet each other face-to-face. The core development teams in open source projects are relatively small

[181] and use email or discussion forums for most of their communication needs. Successful projects produced by the open source community have demonstrated that groups can successfully collaborate by relying solely on the tools in `functional` layer [154].

4.2 Artifact Management

The basic support for managing artifacts in a collaborative software development environment involves: access to a common set of artifacts, private workspaces that allow developers to work uninterrupted on their tasks, and version control. Software configuration management (SCM) systems provide exactly this kind of functionality [215, 183]. Developers can check-out artifacts, in which they are interested, from a central repository into a private workspace. There, they can make changes without any interference from other developers. Once the changes are complete, they can check-in the artifact back into the repository, making the artifacts and the changes available to the rest of the team.

SCM systems also version artifacts such that changes can be made incrementally and rolled back should the need arise. The SCM repository creates a new version (a new identifier) for the artifact when it is checked-in. Versions are not directly accessible, but have to be checked out from the repository. This process ensures that artifacts in the repository cannot be changed directly and that changed artifacts are always given a new version number. SCM systems thus support the development process by maintaining the software artifacts, recording the history of the artifacts, providing a stable working context for changing the artifacts, and allowing a team to coordinate their changes to its artifacts [76].

The check-out/check-in model of SCM systems, as characterized by systems, such as SCCS [187], RCS [214], and DSEE [131], is mainly pessimistic in nature. In this model, once an artifact has been checked-out by a developer, it is locked by the repository and is not available for modifications by others, until the changes are checked back into the repository by the original developer and the lock is released by the repository. Pessimistic SCM systems ensure that there are no conflicting changes to the same artifact, but only ensure this by severely limiting the amount of parallel work.[2]

SCM systems were the first applications that handled artifact management and based on their success they have become the de facto system for managing software artifacts, especially code [74]. Some systems, such as BSCW [6] and Orbits [143], that manage artifacts by leveraging repositories. However, these tools are not as popular or widely used as SCM systems. We discuss some of these systems later, based on their advanced functionality in addition to basic artifact management.

[2]RCS does allow merging (rcsmerge [214]), but this is a functionality that we will discuss in Section 6.

4.3 Task Management

In the `functional` layer, task management involves task allocation and task monitoring. Managers allocate tasks based on expertise of developers and in a manner to avoid duplication of work or conflicting efforts [67]. Once tasks are assigned, managers continuously monitor the progress of team members to ensure that individual tasks are executed on time, to coordinate the activities of team members where responsibilities overlap, and to keep the overall development effort on schedule.

In the past this was a purely manual activity, with schedules on paper, meetings to assign tasks, status meetings for monitoring progress, and so on. The tools in the `functional` layer of the pyramid provide managers with rudimentary task management support. Early versions of nowadays well established project management tools like Milos [85], Autoplan [62], and MS-project [35] provided basic project planning and scheduling tools.

At this layer, we also include tools such as, email, basic SCM systems, and rudimentary bug tracking systems as they too help in scheduling and monitoring task assignments. Besides its critical role in communication among individual developers, email has come to play a central role in task management. It has been found that senior managers spend a majority of their time answering email [66, 18]. Email has become a popular management tool as it facilitates managerial activities like: assigning tasks, scheduling meetings with the help of calendering systems that track when a developer is busy, and placing reminders for themselves by flagging important email or with to-do lists [16].

Pessimistic SCM systems (e.g., RCS [214], SCCS [187]) and bug tracking systems (e.g., Mantis [144], Bugzilla [30]) serve as coordination tools for project management by allowing managers to monitor changes to artifacts. For example, a manager can query a SCM repository to detect which software artifacts are currently locked by which developers, bug trackers can be used to detect which bug reports have been closed and which are still open, and so on. Pessimistic SCM systems can also prevent task duplication considerably, as in this model only one developer can work on an artifact at any given time. This ensures that developers are always working with the latest changes in the repository and therefore will not duplicate efforts that have already been coded. As stated before, the open source community is a successful example of software development teams that mainly depend on email, SCM systems and bug trackers for their collaboration needs.

5 The Defined Layer

The tools discussed in the previous layer, the `functional` layer, provide only rudimentary automated support for collaborative groups. They are sufficient for teams that are small, have minimum overlapping of tasks, and low interdependencies in functionalities. The structure of the teams changed as the field of software engineering matured and demand for software burgeoned, with different disciplines relying on software for their applications (e.g., medical community, automobile industry, telecommunication industry). Teams became larger in size and more structured in nature. Moreover, in order to stay competitive and reduce the development time and cost, parallel development was encouraged.

The tools at the `functional` layer can not handle these changed collaboration needs. Well-defined software processes are needed to coordinate the activities of members in a large team. The second layer in the pyramid, the `defined` layer, historically builds on the `functional` layer and includes tools that support such well-defined software processes to help in collaboration. The tools at this layer prescribe defined steps that advise developers which steps they are required to take and who to communicate which artifact with.

The birth of object oriented programming, in some ways, accelerated parallel development, as programs could now be broken down into smaller modules with different developers working on each module. Concepts like encapsulation allow modularization with well-defined interfaces that can then be integrated back [172].

Tools at the `defined` layer are characterized by their support for larger teams with clearly defined processes. An interesting observation regarding the systems discussed at this layer is that they primarily focus on task management. This is in accordance with the theme of this layer: prescribing defined coordination steps to help the team in effectively collaborating.

5.1 Communication

Email as the sole communication medium for large teams is inadequate, given the overwhelming number of email that are sent in a typical team. In such teams, email users quickly become inundated with the volume of email they receive per day and often end up scanning only the subject headers [86, 17]. Many times, users have to retrieve previously discarded email, email that seemed unimportant then. Retrieving old email, that may have been important, remains a daunting task in spite of the sort mechanisms and filters provided by email clients [142].

Communication that is recorded along with the artifact is easier to retrieve at a later stage than when archived separately (as in email). Artifacts such as project schedules and bug reports play a crucial role in coordinating the development activities [30]. Artifacts playing an important role in coordination is depicted by the slow migration of the communication strand into the artifact management strand at this layer (See Figure 4). Moreover communication records that are tightly associated with artifacts, such as bug reports [30] or check-in comments in SCM systems, allow better access mechanisms since artifact storage usually leverages access and query mechanisms of underlying database systems.

In addition to one-to-one communication, meetings are the next most common communication medium, but scheduling face-to-face meetings to bring the entire team together in a room often proves to be difficult. It is now a common practice to either teleconference or video-conference with team members who cannot be physically present in the same room [105, 108]. Some companies even have mobile units with conferencing facilities that allow developers to attend meetings when they are travelling. In addition to the tele/video conferences, users can use the internet to remotely login and use streaming video and audio to participate in meetings. Designers at the Jet Propulsion Laboratories frequently use web based conference calls to participate in design meetings with their design teams that are geographically separated [145].

We see that, at this layer, the technology has shifted from just email and informal communication conventions to communication that is recorded along with the artifacts as supported by process based environments.

5.2 Artifact Management

As discussed earlier, parallel development became necessary to reduce overall development time, which in turn made concurrent access to artifacts necessary. Coordinating parallel access to multiple artifacts is too complicated to handle manually. Teams need automated support to keep track of: which developers have access to which artifacts, which developer is working on which artifacts, which versions have been created that need to be integrated, and so on.

Parallel development can be either synchronous or asynchronous in nature. Groupware applications support synchronous editing, the majority of which deal with collaborative editing. These collaborative editors allow multiple users to simultaneously access and edit documents (e.g., text documents, software code, design drawings). To take a few examples, GROVE is a textual multi-user outlining tool [71]; ShrEdit is a multi-user text editor [150]; DistEdit is a toolkit for implementing distributed group editors [125]; and Flesce is a toolkit for shared software coding [60]. These editors ensure the consistency of simultaneous changes either by using locks or by ordering the editing events. Most of these editors also support shared views and shared telepointers (MMM [22], GroupSketch [87]). Shared views allow different users to see a part of the document in exactly the same manner as the other user using the WYSIWIS (What You See Is What I See) metaphor [203]. Shared telepointers allow multiple cursors, one for each user, which are shown at all sites and updated in real time. The user interfaces of these applications are tightly coupled such that the views of the users are updated to reflect the action of every user. Some of these systems also provide support for speech and communication (GroupKit - a real time conferencing toolkit [188]).

SCM systems support asynchronous editing (primarily software code), where developers edit artifacts in their private workspace and then synchronize their changes with the repository [215]. SCM systems like CVS [20], Telelogic CM/Synergy [213], and Rational ClearCase [3] are optimistic in nature and allow concurrent changes to a common set of artifacts. Unlike the pessimistic SCM systems discussed in the previous section (Section 4.3), optimistic SCM systems do not place locks on artifacts when they are checked-out. The repository allows a developer to check-out an already checked-out artifact as long as the changes are later synchronized in the repository. In the optimistic model, developers who complete their changes first have the opportunity to check-in their code. The next person who tries to check-in has to ensure that their changes integrate with the latest version in the repository. Typically, this is supported by automated merge facilities [151]. Some SCM systems also can provide access rights to developers such that different developers have different privileges based on their role in the project [76, 51].

Overall, the level of support has risen from providing basic support in artifact management to managing parallel development and means of integrating these changes. The tools at the `functional` layer provided the basic infrastructure using which developers could access and modify artifacts that were stored in central repository. Tools at the `defined` layer provide enhanced support in artifact management and allow both synchronous and asynchronous parallel development.

5.3 Task Management

Task management is at the heart of this layer. Task management is where one really manages the steps in the process from creating project schedules to coordinating activities of developers. Historically there have been two approaches: workflow and process engineering. Both these approaches break the development process into steps and prescribe the computation required at each step and the coordination protocol between the steps. While initially they seem similar they are really complementary to each other. Workflow focuses on the unit of work that flows between the steps [165], whereas process engineering focuses on the development steps [171, 173].

5.3.1 Workflow

The concepts and technologies involved in workflow systems evolved from work in the 1970s on office information systems. Workflow systems are mainly useful for domains that have standard procedures, for example office automation or inventory control. A workflow system is built upon a workflow model that describes the characteristics of the target system. The workflow model characterizes the target system by focusing on the critical characteristics of the system while ignoring the non critical ones.

The characteristics that a model considers critical depends to a large extent on the purpose of the model. For example, a model of a purchasing procedure created for teaching new employees the current procedure will focus on describing how authorizations are obtained, how people in purchasing interact with vendors, and so on. On the other hand, if the model is to be used to analyze the staffing requirements of the purchasing department, characteristics such as the distribution of purchase requests, and the amount of time required to identify a vendor would be critical [165].

The workflow model divides the work procedure in an organization into discrete steps with explicit specifications of what actions are to be taken at which step and how the unit of work flows through the different steps. A workflow language that describes the model, thus, contains constructs to define a set of steps to represent units of work. It does so with two languages, a sequential computation language [206, 135, 228] that provides an interpretation for each step, and a coordination language (E-nets [164], ICNs [70], Petri nets [119]) that defines how the unit of work flows among the steps.

A workflow model is created in a workflow modeling system such as IBM Flowmark [155, 135], Filenet Visual Workflo [68], or PIF [134]). These modelling systems are editing environments that provide facilities for creating and browsing a representation model (describing how a particular step is accomplished for the benefit of human users), for applying algorithms to an analysis model (quantifying the various aspects of a step's execution), and for collaborative interaction and information archival for design models (capturing requirements, constraints, relationships, and algorithms for implementing an individual component).

Once a workflow model has been formulated it needs to be enacted, an operation in which the model is encoded into a set of directives that will be executed either by humans or computers in a workflow management system. Workflow enactment systems establish the order in which steps should be executed for each work unit and identify the software modules that would implement each step in a workflow management system (e.g., FlowPath [31], FlowMark [135, 155], and InConcert [146]).

5.3.2 Process-centered Software Engineering Environments

Process-centered Software Engineering Environments (PSEEs) are environments that provide a process model (a representation of the process) to support development activities. These environments support the process designer in analyzing the existing process, designing a process model, and enacting the model to create directives that are then executed by the team and the computer system.

The heart of the process environment is the process model, as it is the model that determines the accuracy of the process, its flexibility, and whether the process can deal with exceptions. The process model describes the process to be used to carry out a development task, the roles and responsibilities of developers, and the interaction of tools needed to complete the tasks. Process models are usually expressed in a formal notation, so that they can support process analysis, process simulation, and process enactment. Different paradigms exist that different process modeling languages follow, such as state oriented notations (Petri nets [177]), rule-based languages [77], or logic languages [77].

Once the process model is defined, the environment enacts the model and provides a variety of services, such as, among others, automation of routine tasks, invocation and control of software development tools, and enforcement of mandatory rules and practices. Several PSEEs are currently available, each with a slightly different focus. We illustrate a few representative environments here.

- *OIKOS*: OIKOS is a research project [160] whose main goal is to ease the construction of a PSEE. It is an environment to specify, design, and implement PSEEs. It also helps in the comprehension and documentation of software processes. It has been developed in Prolog [40] and is built in two separate languages, Limbo [4] is for the requirement specification and Pate [5] for its implementation.

- *EPOS* [42] has chiefly been designed to provide flexible and evolving process assistance to multiple software developers involved in software development and maintenance. Its process model is expressed in SPELL [43], an object-oriented, concurrent, and reflexive modelling language. It has its own versioned database called EPOS-DB which is used to store the process models.

- The *SPADE* project [12] was created as an environment for software process analysis, design, and enactment. SPADE adopts extended Petri-nets [177, 161] and augments them with specific object-oriented constructs to support product modelling.

- *Arcadia* [212] was built with the goal to create an environment that was tightly integrated yet flexible and extensible enough to support experimentation with alternate software processes and tools. The environment is comprised of two complementary parts. The variant part consists of process programs and the tools and objects used and defined by these programs. The other, fixed part, or infrastructure, supports the creation, execution, and changes to the constituents to the variant part. The infrastructure part is composed of a process programming language and interpreter (Appl/A [103]), object management system (PGraphite [224], Cactis [114]), and user interface management system (Chiron [227]).

In addition to a well defined coordination process, managing changes to software artifacts is an integral component in software development [215]. The importance of SCM has been widely recognized by the software development community ("Indeed, SCM is one of the few successful applications of automated process support" [74]). Its importance is also reflected in particular in the Capability Maturity Model (CMM) developed by the Software Engineering Institute (SEI) [175]. CMM defines levels of maturity in order to assess software development processes in organizations. Here SCM is seen as one of the key elements for moving from "initial" (undefined process) to "repeatable" (project management and quality assurance being the other two).

5.3.3 Summary

To summarize, workflow and process environments have changed the picture as with respect to the previous layer from bare bones support in task management to sophisticated systems that prescribe the development steps and the coordination protocol required. Tools at the `functional` layer provided only rudimentary support in task scheduling and task assignment. Compared to them, the tools at the `defined` layer are sophisticated system that prescribe the ideal development process for an organization and provide directives to enact it.

6 Proactive

While tools at the `defined` layer are useful and effective in breaking the development process into smaller steps and coordinating developers' activities between the steps, developers using these tools soon realized that these systems made the development process very rigid. As a result of the prescribed rules of coordination, developers no longer had any flexibility in the development process. To work around these strict rules, developers often cheated the system by creating their own ad-hoc coordination processes. For instance, it was found in a study [55] at NASA/Ames, that developers followed an informal convention where in they sent email to the developers' mailing list before checking-in their code. This email described their changes and its impact on others thereby preparing others for the change. In another study [89], it was found that developers in a software development company used the information about who has currently checked-out what artifacts to coordinate their activities outside of the provided and otherwise used SCM system. A flexible coordination infrastructure was thus needed, one that recommends the desired coordination steps, but is flexible enough to allow developers to change the process if necessary.

Tools at the third layer of the pyramid, the `proactive` layer, satisfy this need and allow developers to be proactive by giving them more control over the development process that they follow. These tools primarily allow developers to obtain information regarding the state of the project to help them detect potential conflicts. The critical advance in tools at this layer is that developers are no longer bound by strict rules, but have the flexibility to take decisions to change the process if necessary.

6.1 Communication

In accordance with the theme of this layer, the communication tools at this layer allow developers to be more proactive. They achieve this by: (1) allowing developers to communicate with their colleagues in real-time, such that they no longer have to wait for the other party to reply, and (2) by allowing developers to monitor changes to their project space, such that they are informed when other developers change artifacts of interest.

6.1.1 Synchronous Communication

In asynchronous communication, such as email, users do not have any control over when their colleagues would reply. Sometimes email discussions can stretch over days, with each party asking questions and waiting for answers. This waiting time can be eliminated with synchronous communication, such as Instant Messaging (IM). Here, similar to email, users have to initiate a conversation and wait for the other party to respond, but once the person responds, users can immediately start conversing in real-time. Since IM is akin to having a conversation, it is much easier and faster to discuss the issues involved with a particular problem and resolve it. Compared to email, IM is much more informal, which allows users to quickly draft messages and not worry about formalities [162, 26].

Users generally configure the IM environments to indicate whether it is an appropriate time for others to approach them or not [162]. Most IM environments provide a standard set of status messages, such as *away*, *idle*, and *busy* for this purpose. Some environments (e.g., Yahoo) allow users to personalize the status messages to make them better suited to the current situation (e.g., *in a meeting*, *on the phone*). Researchers are currently investigating the effect of embedding emotions in IM environments [121, 184], such that users can better express their mood, which in turn would reflect their emotional availability.

Instant messaging has been widely used for informal communication over the internet, but only recently has been adopted by businesses as acceptable means of communication [104]. Researchers are now studying the effects of IM on employee productivity [104] and interruption management techniques to help developers manage the continuous interruption caused by IM [56, 50]. IM environments also allow users to archive their IM conversations that can then be later investigated. Currently IM archives are not widely used yet, but it is a resource that might be tapped in the future. JAZZ [36], an awareness tool, is integrated with an IDE that allows developers to start chats from the IDE and in the context of the code that they are involved in. JAZZ then annotates the part of the code that was discussed with the relevant chat message.

6.1.2 User Initiated Artifact Monitoring

The tools at the previous layer mostly coordinated development activities by isolating developers in private workspaces and synchronizing their changes at specific synchronization points. Even though this isolation was needed, it often lead to conflicting changes. Developers thus felt the need to break this isolation and be aware of development activities around them, such that they could detect potential conflicts before they become large and difficult to resolve.

Tools at this layer allow developers to monitor the artifacts that they are interested in, but only if they explicitly register interest (tools at the next layer can provide semi-automated monitoring, see Section 7.1.2). In the event of a change to an artifact, the tool notifies those developers who have registered interest in the artifact. Being notified of changes as they are occurring allows developers to gain an overall understanding of their project, with respect to what changes are being made to which artifacts and by which developer. This awareness helps developers understand others' changes and how that might affect them. Even though these tools require manual registering of interest they are an advancement over tools at the previous layer, where developers had no flexibility and were prescribed what steps to follow, when to follow, and how to follow.

Since Software Configuration Management (SCM) systems continuously log which developer has accessed or modified which artifact in the repository, these systems are well suited for providing capabilities for monitoring artifacts. Whereas SCM systems of the previous generation (tools at the `defined` layer) allowed developers to access information regarding which developers are changing which artifacts by manually querying the repository, SCM tools at this layer (e.g., Coven [37], CVS-watch [20]), allow developers to obtain this information at real-time by registering their interest on specific artifacts with the repository. Clearly, registering interest in artifacts is an advantage only if it is easy to set the monitors or is a one-time effort, otherwise it amounts to expensive user effort and will never get adopted by the development community [92]. An interesting observation is that the next layer, the `passive` layer, is the epitome of this, because in the passive layer we place tools that aim to have no set up time whatsoever.

Most of the tools that build on SCM systems monitor artifacts that are stored in the repository. Developers thus know when an artifact has been checked-out for modification and when changes have been completed. However, they have no idea of what changes are taking place in the workspace. Being aware of changes as they are taking place in the workspace can be more important and timely in avoiding conflicts, than when changes have already been made [97]. Only recently has research in SCM started investigating monitoring changes at the workspace level [195, 170]. There are a number of CSCW applications that provide workspace awareness, but they usually present this information in a passive format and we place those tools in the next layer.

Some systems, like COOP/Orm [140, 139] and State treemaps [157], do not support notification of changes, but allow developers to monitor changes to artifacts by presenting them, as part of their regular interaction, with a view of parallel activity in an easy to understand graphical format. We notice that these notifications of changes are currently a very active mechanism in SCM. We could add the CSCW tools that have similar features, but it turns out that most of these tools tend to be more passive in nature and we therefore will discuss them in the next layer. For example, in BSCW [6] there are query mechanisms akin to the SCM tools, but it also provides some automated visualizations that display the information when a user browses the repository.

6.1.3 Summary

To summarize, tools at the previous layer provided the process infrastructure that defined the coordination protocols that developers were required to follow (who to coordinate with and when). Developers were required to strictly follow these prescribed steps. We note that the tools at this layer allow developers to be flexible and represent the intertwining of the strands. Developers can be more flexible in the way they handled their development process by using the communication tools of this layer (e.g., IM and artifact change monitors). Developers no longer have to depend on asynchronous communication media, but can directly chat with their colleagues to quickly resolve problems. Another instance of this flexibility is that developers can break the isolation promoted by workspaces and monitor development activities in their project. Even though tools at this layer require users to explicitly set up the monitors and actively investigate the notifications, they are still an advancement from the isolation enforced by tools at the previous layer.

6.2 Artifact Management

Tools that support artifact management at the `proactive` layer allow developers to avoid conflicts either by versioning artifacts at a lower granularity (smaller than the source file level) or by allowing developers to share intermediate changes. In addition to these mechanisms, which help developers avoid conflicts, tools at this layer provide sophisticated conflict resolution support to help developers resolve conflicts that cannot be avoided. We once again focus our discussion on SCM tools, as SCM systems have been the forerunners in artifact management ("Product management forms the core functionality of SCM systems" [74]).

6.2.1 Fine-Grained Versioning

SCM systems of the previous generation (e.g., RCS [214], SCCS [187], CVS [20]) version artifacts at the "source file" level, the file being the smallest unit in the repository. But versioning at the file level often is not the best solution [139, 74]. In pessimistic SCM systems (RCS, SCCS) artifacts are locked to avoid conflicting changes to the same artifact. Since the lock is applied to the entire source file, developers who intend to make changes to non-conflicting sections of the file have to wait for the lock on the file to be released. In large teams, waiting for locks can quickly become a bottleneck, increasing the time and cost of development. Even though optimistic SCM systems (CVS [20], Subversion [216]) do not suffer from bottlenecks due to locking, versioning at the file level still makes it fairly cumbersome to extract change histories of smaller units (e.g., a particular method or function) from the version history of the source file containing the method/funtion.

Fine-grained versioning avoids these bottlenecks and difficulties in allowing access to smaller software constituents. Most software artifacts can be broken down into smaller units. For example, software programs are composed of functions that can be further broken down into declarations and statements; text documents are composed of sections and paragraphs; and so on. Software changes usually affect more than one file [176], but "it is often the case that a change affects only a small part of a document" [139]. Owing to encapsulation of concerns in methods, changes to software artifacts are usually contained in smaller logical units, and most of the times developers only change a specific method and do not touch other parts of the program [140]. Empirical data collected by Perry et al. further supports this observation [176].

Fine-grained versioning of artifacts takes advantage of this characteristic and allows multiple developers to change different parts of the same artifact (e.g., methods, collections of statements). Systems such as Coven [37], COOP/Orm [139], and Desert/Poem [182] provide fine-grained versioning. Each of these systems version and manage configurations directly in terms of functions, methods, or classes in the source code with the goal to increase software reuse and collaborative programming [136].

6.2.2 Conflict Management

Task allocation can seldom ensure that task responsibilities do not overlap. This overlapping of task responsibilities, more often than not, results in conflicting changes. Tools in areas such as CSCW and Workflow attempt to avoid conflicts by locking artifacts. SCM systems that allow parallel development attempt to alleviate this problem by supporting fine-grained versioning and providing awareness of parallel activities. However, conflicting changes are still very much a reality and organizations lose time and money in resolving them [176]. Considerable research on conflict detection and resolution has been done in the area of SCM [151, 44], since developers are required to merge their changes and resolve any conflicts before they can check-in their artifacts into the repository. These merge tools can be classified along two dimensions, the ancestry of the artifact considered and the semantic level at which merging is performed.

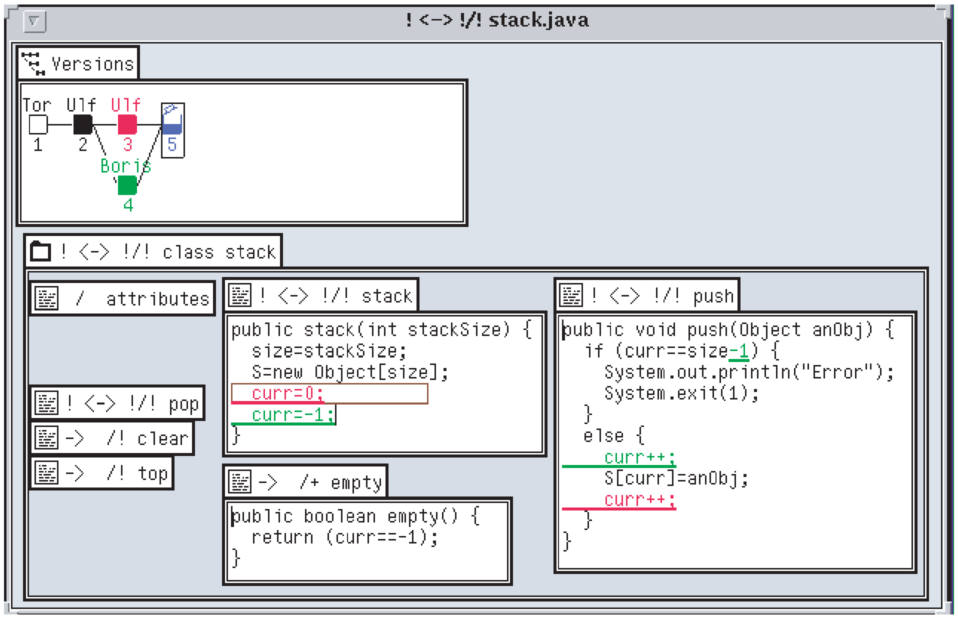
Three main merging techniques, based on whether the tool considers the parent of the version or not, exist. *Raw merging* simply applies a set of changes in a different context. For example, consider a change c_2 that was originally performed independently of a previous change c_1 to an artifact. In raw merging, c_2 is later combined with c_1 to produce a new version. Source Code Control System (SCCS [187]) supports raw merging. *Two-way merging* compares two versions of an artifact a_1 and a_2 and merges them into a single version a_3. It displays the differences to the user, who then has to select the appropriate alternative incase of a conflict. A two-way merge tool can merely detect differences, and cannot resolve them automatically (Unix diff [116]). *Three-way merging* compares two versions of an artifact in a similar fashion to two-way merging, but consults a common baseline if a difference is detected. If a change has been applied in only one version, the change is incorporated automatically, otherwise a conflict is detected, which then needs to be resolved manually. Compared to two-way merging, this resolves a number of 'supposed' conflicts automatically (false positives) [151]. Aide-de-Camp [118] offers a three-way merge tool in addition to raw merging.

Crosscutting the various ancestry-based merge techniques is the semantic level at which merging is applied [44]. *Textual merging* considers software artifacts merely as text files (or binary files). The most common approach is *line-based merging*, where each line is considered an indivisible unit (rcsmerge [214]). This approach can detect when lines have been added or deleted, but cannot handle two parallel modifications to the same line very well. It has been found in an industrial case study that three-way, line-based merge tools are most popular and can resolve 90 percent of changed files without any need for user intervention [176, 130]. *Syntactic merging* [29] is more powerful than textual merging as it recognizes the syntax of the programming language, which in turn allows them to perform intelligent decisions. Syntactic merging ignores conflicts due to differences in formatting or commenting and guarantees the syntactical correctness of the merge result. However, syntactic merging is not trivial and has been implemented by only a few research prototypes [29, 222]. *Semantic merging* builds on syntactic merging and takes the semantics of the software program into account [21, 23, 111]. Semantic merge tools perform sophisticated analyses in order to detect conflicts between changes. The exact definition of a semantic conflict is a hard problem to solve and the merge algorithms developed so far are applicable to only simple programming languages. In addition to the basic merging techniques discussed above researchers are also investigating structural, state-based, change-based and operation-based merge techniques. Mens [151] provides a detailed discussion of these aforementioned techniques.

Researchers in areas other than traditional SCM have also investigated conflict resolution. We briefly describe some of the proposed solutions here. Asklund et al. [9] in COOP/Orm use the structure of the program in its visualizations to help users make merge decisions. The tool automatically performs a three-way, line based merge and the user needs to interact with the tool only for conflicts that it cannot automatically resolve. In such cases the tool helps users make merge decisions by presenting them with a version graph of which developer has changed which artifacts. The artifacts in the graph that have conflicts are highlighted (Figure 6, top panel). The editing panel of the tool annotates artifacts that have merge conflicts with "$< - >$" (the parent window is showing a conflict in `stack.java`) and displays changes from both the files. Changes that are conflicting are color coded. For example the bottom middle panel displays the conflict in the code for `stack.java`, and shows two variations for the assignment statement for the variable `curr`. The user then has to choose the changes that should be preserved.

Molli et al. investigate synchronization problems arising when changes are executed in parallel. They have produced a number of prototypes (e.g., COO [156] and divergence metric [158]) that attempt to coordinate parallel activity. The COO[3] environment supports synchronization of development activities by encapsulating

[3] *COO* stands for cooperation and coordination in the software process



```
public stack(int stackSize) {
  size=stackSize;
  S=new Object[size];
  curr=0;
  curr=-1;
}
```

```
public boolean empty() {
  return (curr==-1);
}
```

```
public void push(Object anObj) {
  if (curr==size-1) {
    System.out.println("Error");
    System.exit(1);
  }
  else {
    curr++;
    S[curr]=anObj;
    curr++;
  }
}
```

Figure 6: COOP/Orm merge UI [9].

the activities as COO transactions. COO transactions relax the isolation property enforced by traditional ACID[4] transactions while maintaining the other properties. The COO environment allows developers to share intermediate results. In COO each developer works on their local copy of the artifact, makes changes and can place an intermediate version of the artifact in the repository that can then be used by another developer. COO creates a final stable version of the artifact based on the order in which the transactions were performed [156]. In other research, the authors investigate the divergence that can occur in an artifact when developers concurrently change the same artifact. They calculate this divergence based on the operations that have been made to the artifact (operational transformation algorithms) and present the calculated divergence metrics as graphs. The aim of the tool is to enable users to better understand the divergence that has emerged as an indication of the effort involved in its resolution, which needs to be resolved.

6.2.3 Summary

Compared to the previous layer in our pyramid, artifact management has advanced from just providing the infrastructure for allowing parallel development to providing automated support for integrating parallel changes. The focus of the tools at the previous layer was to create protocols that would allow multiple developers to access and modify artifacts synchronously or asynchronously. If the protocol allowed an artifact to be modified simultaneously by two or more developers, it was the respon-

[4] *ACID*: Atomic, Consistent, Isolation, and Durable

sibility of the developer to integrate the parallel changes. This is problematic, as it was found in a study that the amount of parallel work is proportional to the number of conflicts in a project [176]. Tools at this layer therefore focused on helping developers to deal with conflicts arising out parallel development.

6.3 Task Management

The tools at this layer support task management mainly by drawing upon the extensive knowledge base that is already available in the repository archives, such as change logs, design documents, check-in comments, and so on. The tools analyze this information and display the results as visualizations (e.g., which developer has changed which artifact, how many lines of code have been changed by a particular developer) or provide recommendations to developers (e.g., which developer can help me, which artifacts are important for the task at hand, who has worked on similar projects).

A thing to note is that these tools are useful to both managers and developers. In fact many of the tools that we discuss here have been designed more for the developer than the manager (e.g., Hipikat, a tool that allows new developers to be productive faster). Another observation is that these tools support task management by mining information from archived artifacts, which is a clear example of the blending of artifact management and task management strands of the pyramid.

6.3.1 Visualizing Software Evolution

Managers need to maintain a comprehensive view of the status of their projects along with the interdependencies between the software modules, so that they can make informed decisions. Managers usually make these kind of decisions based on their expert judgement, but as software systems become more complex in nature and larger in size, keeping track of the software and its components becomes more difficult.

Research in information visualization aims to represent software systems, their evolution, and interdependencies between its constituents in an easy to understand graphical format, such that users can better understand the project space and thereby make informed decisions. Some of these visualizations concentrate at the code-level (Augur [82], Figure 7), while others take a broader outlook and visualize the software system at the structural level (Evolution Matrix [129], Figure 8). Some systems are currently exploring displaying these visualizations at a central place using multiple monitors [169]. Storey et al. [205] have surveyed such visualizations that support awareness of development activities. We briefly describe a few such systems here.

SeeSoft [69] and Augur [82] use the software code maintained in a SCM repository to display the number of lines that a particular developer has changed in a particular file. Figure 7 illustrates Augur's multi-pane interface, with the central pane displaying lines of code that are color coded to represent changes by different developers (lines of code changed by a particular developer are colored in a single color). This metric helps managers comprehend the contribution of each developer, the evolution of the system, and existence of code decay. When code gets old (e.g., legacy systems), eventually there are parts of code and issues involved with them that people forget. As the code keeps getting modified, its architecture degrades with bits of code never being used or being duplicated. Code decay represents this degradation of code that makes subsequent maintenance of the system or addition of new functionalities a difficult task.

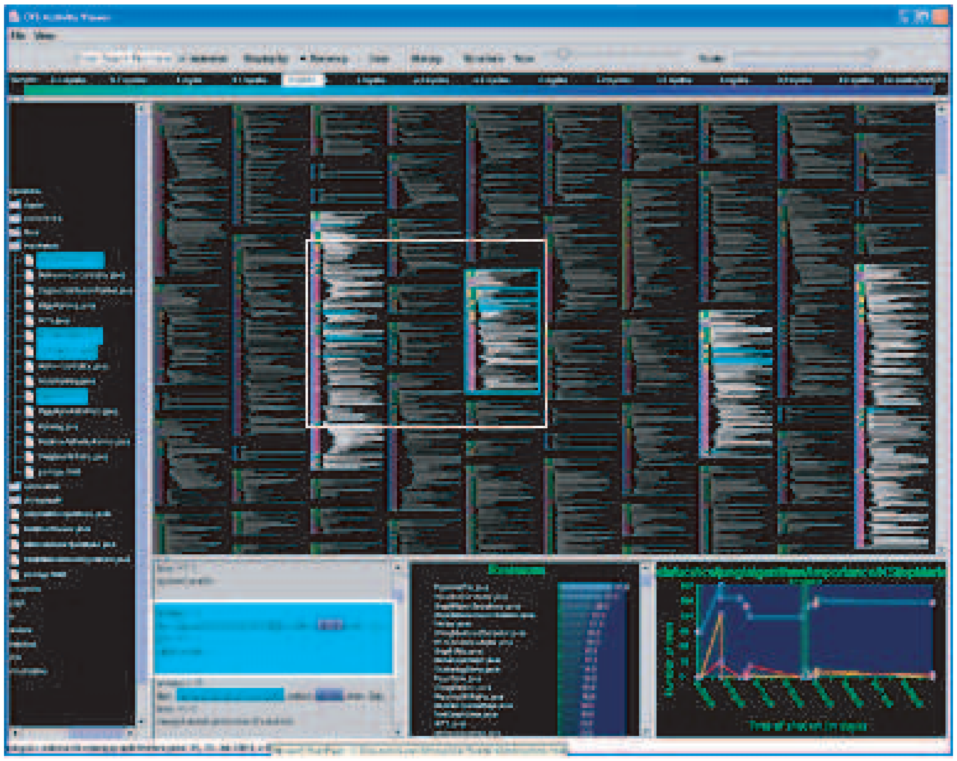


Figure 7: Multi-Pane Interface of Augur [82].

Hill and Hollan modified the ZMACS editor to create the Edit Wear and Read Wear tool [110]. This tool displays documents with scroll bars placed in the margins. Scroll bars indicate how many times a particular line has been read or edited. Additional information regarding the authors who have modified the document is also presented as meta-data along with the scroll bars. This system can therefore graphically answer which sections in a document are most stable (changing the slowest) or unstable (being edited rapidly).

Evolution Matrix [129] focus on the architecture of the system to represent the evolution of the system. The Evolution Matrix considers classes that have been added, modified, or deleted in different versions and creates a 2-D matrix, where each cell in the matrix represents a class. Figure 8 shows the code structure of MooseFinder [204] as visualized by the tool. The rows in the matrix represent the number of classes and the columns the versions of the artifact. The evolution of the system (e.g., when have classes been added to the system, when the system has been stagnating) can be easily understood by simply looking at the evolution matrix.

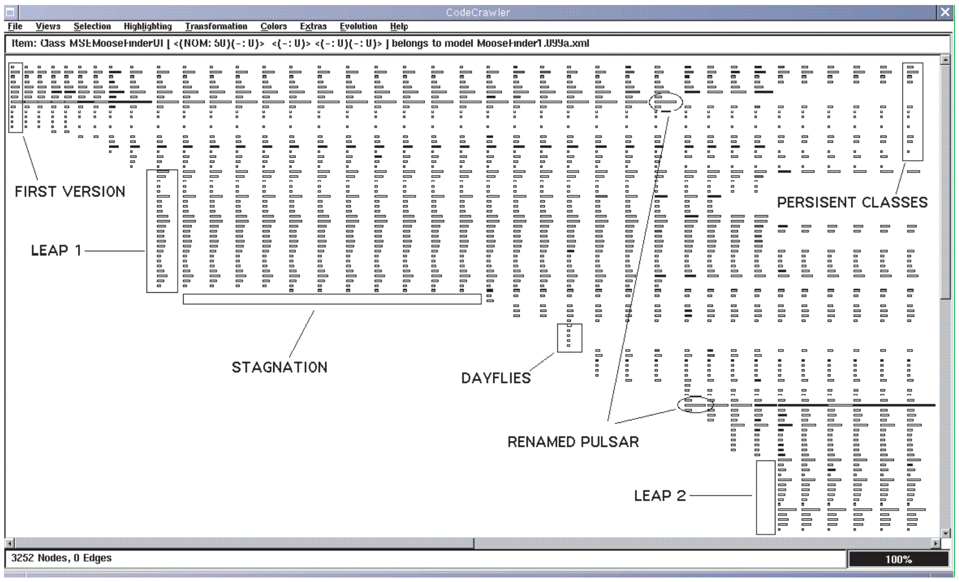

CodeCrawler
File Views Selection Highlighting Transformation Colors Extras Evolution Help
Item: Class MSEMooseFinderUI | <(NOM: 50)(-: 0)> <(-: 0)> <(-: 0)(-: 0)> | belongs to model MooseFinder1.099a.xml
FIRST VERSION
LEAP 1
STAGNATION
DAYFLIES
RENAMED PULSAR
PERSISTENT CLASSES
LEAP 2
3252 Nodes, 0 Edges
100%

Figure 8: Evolution Matrix Showing Code Structure of MooseFinder [129].

SeeSys [11] is a visualization system that uses state treemaps [120] to display software code. The state treemap visualization maps hierarchical information of directories and files onto a rectangular 2-D display. In this method, nodes whose attributes are of more importance are given a larger display area. SeeSys further uses color and animation (zoom in and zoom out) to show historical changes to the system.

6.3.2 Organizational Memory

New developers need to understand the design, the architecture, and any existing code of the system before they can become productive and start coding. Most companies assign an experienced developer as a mentor to the new developer. The mentor guides the new developer in understanding the design constraints of the system and any other relevant information that is necessary to understand the current task. Initially, the new developer requires more help from their mentors but as the developer gains experience these interactions reduce. It has been found in a study [108] that developers are more inclined to help their colleagues who are collocated than colleagues who are separated [91, 107]. Therefore, when teams are distributed in nature (e.g., virtual teams), new developers cannot always get the advantage of having a mentor. In cases were new developers are on their own, they have to spend a considerable amount of time getting acquainted with their project and organization before they can be productive [46]. In this section, we discuss tools that utilize the product and organization information already existing in the software archives to help developers be up-to-date with their task. New developers can use these tools to recommend the correct design practices, experts in a domain, required artifacts for a given task, and so forth.

Software critics are used in design environments (ArgoUML [185], ArchStudio [52]) to help developers maintain the design specifications of the software system. Software critics are active agents that support decision-making by continuously and pessimistically analyzing partial designs or architectures. Critics analyze the changes made by developers and compare these changes with the architecture of the system. If the design violates any constraints it immediately notifies the developer. Since critics are continuous and pessimistic in nature, these environments allow the critics to be switched off so that developers are not distracted in their design exercise. Developers can start the critics to check their entire design once they have completed their task.

Recommendation systems can be used for recommending either domain experts (e.g., Expertise Recommender [148], Expertise Browser [153]) or artifacts (e.g., Hipikat [46, 47], Codebroker [226]) that are essential for the current task. Artifact recommenders help newcomers become productive quicker by recommending existing artifacts from the development set that are relevant to the task at hand. For example, Hipikat [46] uses two techniques to help newcomers. First, the tool infers links between artifacts that may have been apparent at one time to members of the development team but that went unrecorded. Second, using these links the tool acts as a mentor and suggests relevant information that would help the newcomer with their task.

Expertise recommenders on the other hand use data from SCM systems to locate people with desired expertise. They help locate developers who have expertise in a particular domain or have worked extensively in a specific product. The Expertise Browser [153], for example, quantifies the past experience of developers and presents the result to the user. This enables a manager to easily distinguish a developer who has only worked briefly in a particular area from someone who has extensive experience. In addition to locating developer expertise, the tool also helps managers discover expertise profiles of organizations.

6.3.3 Social Navigation

Social relationships are a strong factor in determining who collaborates with whom. Social networks are used by groupware designers to provide a means of visualizing existing and potential interaction in organizational settings [147]. Social networks should ideally make software development environments more sensitive to social situations and guide users toward effective collaborations.

Social networks typically represent groups of people and the connection among them. A common approach is to use social network visualization as an overview of group participation or group membership. Social network visualizations (e.g., Conversation Map [192], PeCo [166], Contact Map [163]) mainly analyze the communication medium among developers. Conversation Map [192] provides a content visualization by analyzing the message content and displays the network of participants. Ogata et al. [166] use PeCo to facilitate finding the person with whom to collaborate by mining and analyzing email exchanges among individuals. Contact Map [163] is a personal communication and contact management system. It mines the email exchanges and allows the user to arrange their contacts based on people with whom they interact more regularly or deem important.

6.3.4 Group Decision Support Systems

Managers spend a considerable amount of their time in meetings. When starting a new project or if a project is behind schedule managers meet to discuss the potential problems that the group faces and generate potential solutions to solve them. These meetings usually involve multiple stakeholders who have to agree on the solution and the way it would be implemented. The members in the meeting may not always be collocated which makes the decision making process even harder. Group Decision Support Systems (GDSS) [126] provide automated support that help managers in their decision making process.

GDSS combine computing, communication, and decision support technologies to facilitate formulation and solution of unstructured problems by a group of people [58]. GDSS improve the process of group decision making by removing common communication barriers, providing techniques for structuring decision analysis, and systematically directing the pattern, timing, and content of the discussion. Technological advancements, such as electronic boardrooms, broadband local area networks, teleconferencing, and decision support software, have spurred research in this area. The availability of advanced GDSS coupled with the fact that managers have to participate in meetings while they are away has made decision-related meetings more frequent and more effective [112].

6.3.5 Summary

In conclusion, task management at the previous layer focused on encoding the development process such that the system could generate the coordination protocol that is required for the team. Effort was mainly concerned with correctly analyzing the current process and enacting it. Exceptions to the prescribed rules were not well handled. The focus of the tools at this layer is to complement the existing process infrastructure by providing further information about development activities, such that developers and managers alike can monitor the project space. The goal of these tools is to make developers aware of development activities around them such that they can detect potential conflicts and take steps to avoid them. In addition to providing information of development activities, the tools mine the already existing organization memory [1, 2] to help developers in their task. These tools mainly help developers by recommending the correct design practices, experts in the field, or artifacts required for tasks that the developer is involved in.

7 Passive

The tools at the previous layer allow developers to monitor development activities around them and be proactive by resolving potential conflicting situations before they become real. However, these tools require explicit user involvement in setting up the monitors, investigating notifications (usually sent via email), or understanding the visualizations [205]. Moreover, barring a few, most of these tools are stand alone applications and require users to switch contexts from their development task to the visualizations. The tools at the `seamless` layer focus on reducing this load on users by providing awareness to users in a passive format, such that they are not distracted from their tasks. The tools further reduce the overload on users by providing filtering mechanisms that allow developers to view only relevant information with minimal or no configuration requirement.

The advance in Integrated Development Environments (IDE) facilitated the integration of awareness widgets into the IDE, thus reducing the context switch required in monitoring activities. For example, tools such as Palantír [194] and JAZZ [36] use the Eclipse plug-in development features [39] to create passive workspace awareness embedded in the IDE.

In a study at a commercial development site, Sawyer and Guinan [196] investigated the development practices of forty teams to assess the effects of production methods and social processes on software product quality and team performance. Findings of the study indicate that production methods, such as use of software methodologies and automated development tools, provide no explanation for the variance in either software product quality or the team performance. Social processes on the other hand, such as the level of informal coordination and communication, the ability to resolve intragroup conflict, and the degree of supportiveness in the team, can account for 25 percent of the variations in software product quality. Heath and Luff, in their seminal study [102, 101], investigated how collocated team members coordinate their actions by monitoring the physical cues and actions of their colleagues. Developers generally subconsciously monitor the ambient noises (e.g., people arriving or leaving, phone ringing, conversations in the hallway) in the environment to coordinate their actions [80].

Teams that are distributed cannot take advantage of these environmental cues making communication and coordination even harder [108]. However, distributed development has become the norm in technology companies. Organizations frequently operate teams that are geographically separated to leverage expertise located elsewhere, spread development across time zones, and reduce development costs [67, 106]. This new development practice exacerbates the already existing problems in communication and coordination [28], with developers having to work across different time zones, languages, and cultures [167, 107]. Researchers in collaborative development have started investigating ways to provide collocation benefits to distributed teams [72]. These awareness tools mainly focus on providing awareness of user presence and activities to help distributed users collaborate; other areas are still to be addressed.

The tools in the fourth layer of the pyramid, the `passive` layer, mainly focus in providing awareness of co-developers and their development activities in a non-intrusive, relevant, and timely manner. The tools at this layer provide further evidence of the three strands slowly but definitely merging into each other to provide tools a well-rounded approach to collaboration.

7.1 Communication

Tools at the previous layer enabled developers to be proactive by allowing them to monitor development activities and communicate at real time. However, users had to be actively involved while using these tools. Tools at this layer attempt to reduce the user effort involved in collaboration. The awareness tools at this layer provide information of parallel activities in a passive format (Tukan [198], Elvin [80]) so as to reduce the effort and context switch that users have to undertake to monitor the changes to artifacts. Some of these awareness widgets (JAZZ [36], Palantir [194]) are integrated with the development environment to further help developers avoid the context switch between their development task and monitoring others' activities. Notifications services (Siena [34], Yancees [199, 200]) allow developers to filter out information in which they are interested thereby reducing the information overload.

7.1.1 Awareness

Awareness can be defined as "an understanding of the activities of others, which provides a context for your own activity" [64]. This context allows developers to ensure that their contributions are relevant to the activity of the group as a whole. Research in promoting awareness in shared work has been widely recognized in the CSCW community [80] and currently is being investigated by researchers in software engineering.

Awareness applications can be broadly classified into three categories based on the type of awareness that they provide [178]. *Task-oriented awareness* focuses on activities performed to achieve a specific shared task. This kind of awareness is usually provided by information on the state of an artifact in a shared workspace or changes to it. This information can then be used to coordinate activities of developers around the shared object (GroupDesk [83], Tukan [198]). *Social awareness* presents information about the presence and activity of people in a shared environment (Portholes [65], Social Awareness@work [217]). Systems focusing on social awareness provide information about who is present in a shared environment and about the activities the users are currently engaged in. The difference between task-oriented and social awareness is the shared context. The shared context for a task-oriented awareness system is established by the artifact that is part of a collaborative process, whereas for social awareness it is the environment that is inhabited by the users. The third approach, *room based awareness*, integrates the aforementioned approaches. In systems following this approach, virtual rooms provide a shared location for the organization and collection of task-oriented objects (TeamRooms [189], Diva [201]). For example, users who collaborate on a particular task share a room and information of someone working on a shared artifact contributes to task-oriented awareness. Whereas, the mere presence of the person in the shared environment represents social awareness.

An interesting observation is that tools that provide awareness are difficult to distinguish based on the strand in the pyramid that they support best, because these tools largely encompass all the three strands in some way or another. For instance, the same tools can be used to support collaboration in communication, artifact management, task management or a combination thereof. This illustrates the fact that, as research has matured, tools have taken a broader scope in supporting collaboration. However, tools at this layer have not blended the three strands completely yet, but lay the foundation for the tools at the next layer (the `seamless` layer) to do so.

Task-Oriented Awareness: The chief purpose of tools that provide task-oriented awareness is to enable developers to have an idea of the development activities of others so that they can coordinate their tasks accordingly. The amount of awareness information that is required for a particular task is based on the nature of the task in which the user is involved. For example, synchronous editing requires the users to be aware of every mouse-click and key-stroke of their co-authors, whereas the awareness details required for asynchronous or semi-synchronous applications are much more coarse (e.g., a developer has finished working on an artifact, a user has started work). Semi-synchronous applications allow users to work in both synchronous and asynchronous mode and therefore the awareness information required for these applications is dependent on the mode of collaboration in which they are currently engaged. For example, a developer might code in asynchronous mode when they need to concentrate on their task, but work in synchronous mode while debugging to obtain help from their colleagues. There are a number of applications that support each of these modes of collaboration (GroupDesk [83], ShrEdit [150], SUITE [59], [88]) and a few that allow developers to change their modes (Tukan [198], Sepia [100]). Tools that allow users to change their mode of collaboration are discussed in Section 7.3.

Awareness tools can also be categorized based on the manner in which they present awareness information. Some attach this information along with the artifacts, which the users then need to investigate (already discussed in the `proactive` layer), while others present this information in a passive unobtrusive display [197]. There are a number of approaches that tools have investigated to disseminate information in an effective but unobtrusive manner. The most common approach is to provide awareness widgets that are small and stay in the background, but change their appearances to draw the attention of the user (desouza99, fritzpatrick02). Some tools emulate the development interface with which the user is familiar to embed awareness information (Groupdesk [83] uses the desktop metaphor, PoliAwac [202] uses the windows explorer interface), while others embed this information directly in the development environment (JAZZ [36], Palantír [194]). Many tools are investigating media outside the desktop. Some of these media includes separate display screens, multiple monitor displays, 3D display of artifacts, ambient devices (ambient globes), and ambient noises (whirring of the fan, thud of mail dropping) [80, 178, 169].

Traditionally, CSCW applications have focused on providing workspace awareness (gutwin96, GroupSketch [87], Quilt [78]) while SCM based tools have focused on repository based awareness (Cover [99], State TreeMaps [157]). This was so, because the nature of work in the two communities differed, with CSCW applications typically supporting synchronous activities and SCM tools supporting asynchronous activities. Currently this division is getting erased with some CSCW applications investigating repository based information [6, 143], while SCM based tools are exploring workspace awareness [97, 194, 170]. A thing to note is that repository based tools are better suited to provide a history of past actions [198, 194] than tools that support synchronous activity and workspace awareness.

Social Awareness: Collocated teams members are often aware of activities around them by subconsciously monitoring cues from the environment (e.g., conversation in the hallways, the humming of printers, people passing by offices). Presence awareness tools support distributed teams in taking advantage of these environmental cues by representing activities of developers in the organization, both distributed and remote, in a digital format. Tools such as Portholes [65] and Polyscope [25] use video technology to capture activities in public areas and offices and present them as regularly-updated digitized video images on workstations. Glance [210], another video monitoring tool, follows a slightly different metaphor and provides the electronic analog of strolling down a hallway and intentionally glancing into people's office.

Use of video technology raises privacy concerns among users (see Section 7.3.1). To avoid privacy issues of users, some tools do not use video technology. These tools monitor user activities and presence by monitoring activities, such as when developers are working on their computer, the kind of activity in which the developer is involved (read/edit), whether the developer is present in a shared area (physical or digital), and so on (Work rhythms [15], awareness monitors [32, 113]). In addition to monitoring the presence of users to ascertain when is the best time to contact a developer, these tools also help in community building via a shared space [81]. A more popular and lower bandwidth solution to presence awareness are IM messages (Section 6.1.1). JAZZ [36], a collaborative development environment, enhances the information that IMs currently provide by adding context information about the task in which the user is currently involved, making the application more suitable for development teams.

Room-based Awareness: In collocated software development teams physical conversation spaces are indispensable to conduct meetings, review design documents, resolve conflicts, disseminate information or just converse informally [53]. Organizations usually designate a permanent shared space to be used by the team, serving as a meeting room, work area, a place to store documents that are needed by the teams projects, and more generally, as a focus for communication within the group [189, 201]. Traditional team rooms have relied on the physical proximity of the team members and their easy access to the room. However, as teams get increasingly distributed and chances for face-to-face discussions reduce, team members turn to virtual team spaces for their communication needs [117].

Research in CSCW has resulted in virtual rooms [189, 201] that support the team room concept for virtual teams. Using these systems, teams that are distributed can still get a feeling of belonging to the group, monitor which artifact has been changed by which developer, be able to access team related documents, or start electronic discussions with team members. BCSW [6] provides similar kinds of information (which developer has last viewed an artifact, which artifact has been viewed the most, who is currently working on shared artifacts) but in a web-based format. Some room-based systems allow developers to interact with the artifacts and other developers in the shared room in a 3D environment allowing developers to experience the benefits of collocation [63, 27]. The main drawback of such room-based awareness systems is that they are non-contextual. These systems are created as separate applications and developers have to switch from their development environment to view activities in these rooms. Moreover, when developers belong to multiple rooms it becomes difficult to monitor activities in different rooms or search for a particular artifact in a particular room.

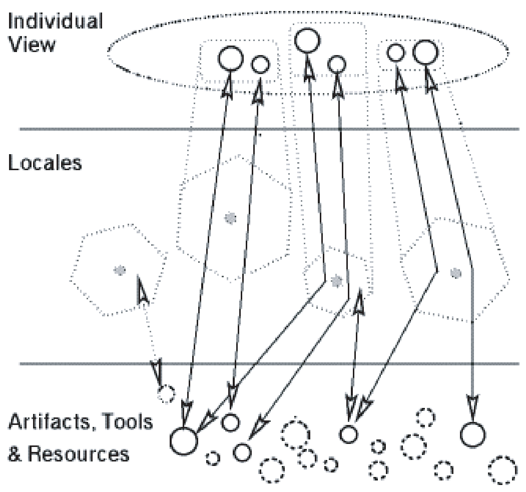


Figure 9: The Three-Layer Model of Orbit [143].

Researchers have started investigating making team rooms better by avoiding the aforementioned drawbacks [117]. Orbit [143, 79], a collaborative environment, is based on the notion of locales, a conceptual place where groups of people come together to work on a shared activity. Orbits implements locales by using a three-layer model (Figure 9). The bottom layer contains artifacts, tools and resources to be used by the organization. The middle layer defines locales based on specific collaborative activities. Each locale is then mapped to the artifacts, resources, and tools in the bottom layer that are relevant to it. The top layer signifies the individual layer, where individuals can belong to more than one locale and access artifacts with respect to the locale that they are currently working in.

7.1.2 Event Notification Systems

One of the primary responsibilities of awareness tools is to notify users of events in which they are interested (an artifact getting modified, a developer exploring the shared space, a discussion in the shared space). A drawback to the awareness tools is the amount and complexity of information that a user is forced to process to take advantage of the benefits of awareness. When users are not able to process this large amount of information they simply ignore it, sometimes ignoring important information. Traditionally, users set filters to streamline the information to suit their needs [17, 18], but it is not the perfect solution, since users have to manually set the filters up and the filters are not flexible enough.

Event notification services work based on the publish-subscribe model, which helps reduce the information overload on the recipients while making information distribution easier for the sender [138]. In order to leverage this advantage many awareness tool builders have started using event notification services for their information distribution needs, instead of creating their own event distribution network. A secondary, but important benefit of using notification services, is that they allow information from disparate sources to be integrated [54]. There are a number of notification services currently available [180, 190]. Here, we discuss some characteristics of notification services that are important for creating collaborative tools.

- *Subscription type*: whether the subscription matching is content based or channel based. Content-based subscription [80, 34] matches the content of the event with the subscription expression and forwards only those events that satisfy the expressions. Channel-based subscriptions [168, 209], on the other hand have predefined named topics or channels that must be specified by both the producer and the consumer.

- *Pull vs Push*: whether it is a pull-based or a push-based mechanism. In the pull-based mechanism [138, 122], clients poll the event server and retrieve matching events, whereas in the push-based mechanism [80, 34] events are sent to clients as soon as they occur.

- *Persistency*: whether the service stores the events persistently for later retrieval [138, 122] or loses the events once they have been delivered.
- *Federation*: whether they work in a federation of multiple servers [80, 34] to increase the load bearing capabilities and provide wide-area scalability or work with a single server [122].
- *Infrastructure*: whether they are strongly coupled with a collaborative tool [174] or provide a notification infrastructure [178]. Researchers are currently investigating making notification infrastructures more versatile [200].

Awareness tools have built on different notification services based on which characteristic is important for the specific tool. For example, federation of servers is suitable for large applications, content-based subscription matching provides more flexibility, and so on.

7.1.3 Summary

To summarize, tools at the previous layer allowed developers to be proactive, but this required active user involvement either when communicating with their colleagues (IM conversation) or for monitoring development activities of others (explicitly set up the monitors, interpret the visualizations). The tools at this layer concentrate on reducing the user effort in acquiring the information and processing it. The tools present awareness information in a passive unobtrusive format, reduce the information overload that developers have to face, and allow developers the flexibility to configure the amount and type of information that they want to view.

7.2 Artifact Management

Conflict resolution is a time consuming and tedious effort that reduces the productivity of teams [176]. Researchers have investigated different methodologies to help reduce conflicts ranging from programming paradigms (separation of concerns, encapsulation of functionalities) [172] to conflict detection tools [36, 129, 6]. However, despite these tools and methodologies, conflicts are still very much a reality in commercial software development. Indeed there are a slew of conflict resolution and merge tools that are currently available (Section 6.2.2). It is a well known fact that the earlier a conflict is detected the less expensive it is, therefore researchers have focused on creating sophisticated tools that help detect and resolve conflicts before they become large and difficult to resolve. Tools at the previous layer enabled developers to understand and visualize the changes to artifacts, such that, based on their experience and domain knowledge, they could identify potential conflicts and take steps to avoid them. Tools at this layer analyze the changes to an artifact and its effect on other artifacts based on program analysis. These tools, thus, aim to provide more accurate conflict detection and at the same time reduce the user effort required.

7.2.1 Change Analysis

Software change impact analysis helps users to understand and implement changes in large systems by a providing detailed examination of the consequences of changes in software [8, 7]. A major goal of impact analysis is to identify the software artifacts that would be affected by a proposed change. Developers can then use this information to evaluate and implement the proposed changes, potentially in a way that causes less impact.

Dependency analysis and traceability analysis are the two main schools of thought that constitute impact analysis. Dependency analysis [137] involves examining detailed dependency relationships among program entities (e.g., variables, logic modules, methods). It provides a detailed evaluation of dependencies at the code level, but does not provide any support for other artifacts created in the software life cycle such as requirement specifications, design documents, test suites, and so on.

Traceability analysis [38] examines dependency relationships among all artifacts that are created in the software's life cycle. For instance, it can relate requirement specifications with associated design components or components in the architecture to software code. Although traceability covers many of the relationships among artifacts that a software repository stores, these relationships typically are not very detailed.

The aforementioned impact analysis methods are not new and have been extensively used in program comprehension. However, these techniques have largely not been used by collaborative tools. Research in collaboration has only recently started investigating the impact of changes made by a developer on another [198], sarma03]. These tools attempt to inform a developer of the effect of others' change on their current task. Being aware of this effect, developers can take appropriate actions to coordinate their activities with others to avoid any conflicting situations. A secondary benefit of the impact analysis is that it helps developers identify the people that they need to collaborate with.

7.2.2 Artifact relationships

As discussed in Section 7.1.1, developers need to have constant awareness of the activities related to a shared product while collaborating. These activities are mainly related to the parts of the product on which a developer is directly working, but a developer may also need to be informed about activities related to other parts of the product that are somehow related to “their” parts. Researchers are increasingly concentrating on the product model for providing awareness to developers regarding which artifacts are of interest to the developer, which artifacts have an effect on the current task, which other developers are working on artifacts that are relevant, and so on. This model is particularly useful in a distributed development context because the product is normally the main focus of work for the developers and the shared product is something on which the developers need to collaborate.

Systems that use product models specifically to aid collaboration use the inherent relationship between the artifacts to interpret the impact of changes. These models mainly represent the artifacts (classes or methods) in the software project as a semantic network [198]. Each artifact in this network is either mapped to a node in a graph or connected to other artifacts based on a relationship model [75]. When a developer creates a new artifact or makes changes to an existing one, the tool scans the artifact for possible relations to other artifacts in the graph to predict the impact of the change. These tools usually allow a developer to define the set of artifacts in which they are interested (called the *focus* of the developer’s current interest). Artifacts that are related to this set of artifacts create the shared interest space (*nimbus*). These tools, thus, use the developers’ *focus* and *nimbus* to predict changes in which the developer should be interested.

The relationships between artifacts can be mined either using the aforementioned product models or by using dependency analysis to create *social callgraphs* [53]. Developers can then navigate these social callgraphs to identify developers who are most likely to affect them and with whom they should probably coordinate their activities.

Task based SCM systems, unlike popular systems (e.g., CVS [20], Subversion [216], Visual Source Safe [193]) that version individual artifacts, track the modified artifacts that are related to a particular task (e.g., CCC/Harvest [10], CM/Synergy [223], Rational ClearCase [211]). Developers in these SCM systems select the particular task that they are working on, the SCM system then uses that information to track all the changes that have been made by the developer. On finishing their task, the developer can simply check-in the task and the SCM system ensures that all the relevant artifacts are checked-in. Managers and developers can thus depend on the SCM system for information, such as, which artifacts were associated with which tasks, which set of changes have been implemented, which changes have been used in the latest build, and so on.

7.2.3 Summary

We note that tools at this layer attempt to reduce conflicts in collaborative development by investigating the relationships among artifacts. Tools at the previous layer showed which artifact is being changed by which developer and it was the responsibility of the developer to leverage their experience and domain knowledge to detect potential conflicts. Tools at this layer analyze the changes to artifacts and their effects, thereby reducing the user effort. A secondary benefit of these tools is that they allow developers to better understand the structure of their product and identify colleagues whom they should collaborate with.

7.3 Task Management

Informal communication is an important part of coordination [196]. Distributed teams lose this ability and its corresponding cues that sometimes lead to delays or creates problems in development [108]. Awareness tools attempt to recreate the social cues in an organization by presenting users with presence awareness, awareness of activities of others, and providing virtual shared rooms where distributed developers can interact. These systems serve the purpose of communication tools and can be used for task management. We note that the three strands of the pyramid merge here as tools take a broader perspective and combine more than one activity. For example, the awareness tools that we discuss in Section 7.1.1, serve the purpose of communication, but can be and are used by developers and managers for task management.

In this section we discuss research that improves the awareness tools already discussed. In particular, we discuss research that addresses the privacy concerns in presence awareness tools, interruptions that distract developers, and the different modes of collaboration that users would like to be involved in based on their current tasks.

7.3.1 Privacy Concerns

Collocated teams often coordinate their activities based on informal communication (coffee hour discussion, supervisors looking over the shoulder to check progress), which is lost in distributed development. Unfortunately, it has also been found that developers who are geographically separated are less likely to collaborate and help each other [128, 108]. To enable distributed developers to take advantage of this informal communication, researchers have created tools that provide presence awareness in organizations (Section 7.1.1).

A primary concern among users of such systems is the tradeoff between access to presence data for legitimate uses, and concerns about privacy [84, 115]. Some of the issues with privacy are camera shyness, threat of surveillance, loss of control over privacy, lack of support of awareness of audience, and so on. Researchers are investigating policies, such as reciprocity and ownership of data, to alleviate privacy concerns. For example, users are not keen on sharing their information with strangers, but are more comfortable with people whose information they can view [84, 104]. Systems that use video technology address privacy concerns of users by allowing them to host their own pictures, place cameras in neutral positions, blur users images, and so on [132].

7.3.2 Interruption Management

Developers have to manage a number of distractions (colleague dropping by, telephone ringing, IM message), which slow and sometimes introduce errors in complex tasks in which developers are involved [186]. In collocated teams, social cues help determine whether a colleague can be interrupted or not (a developer may close the door to their office when they need to work in privacy). In a distributed development environment developers can use the presence awareness tools to create social cues that inform others about their availability. However, these tools fail to avoid interruptions from automated external agents, such as critics, source control monitors, bug-database monitors, and so on. Filtering interruption by creating rules does not always work either, as some interruptions may be useful and help in getting a task accomplished better or faster. Based on the benefits of the interruption, users may assign higher priority to some interruptions than others. This priority may also change based on the task in which the user is currently involved.

Researchers have studied the way managers and developers handle interruptions, and they have found that there is never a good time for interruptions, since interruptions invariably disrupt the current task [49, 113]. This observation implies that interruption management systems should not queue possible interruptions for the ideal time, but send them at the best relative time based on the work patterns of the user. Interruption management tools, thus, need to incorporate a certain level of social process to ease the challenge of limited attention of the users [56, 149].

7.3.3 Changing Modes of Collaboration

The amount of collaboration that developers need varies from working in isolation to tightly coupled collaboration mode, based on the current task in which the developer is involved [98]. For example, a developer might want to work in isolation and not be interrupted while implementing a complex task that requires intense concentration. However, once the developer has finished implementing the task, they may want to collaborate with their colleague to debug the program.

The tools following the formal process-based approach required the users to work in isolation and then synchronize their changes with the repository. The tools following the informal awareness-based approach supported the other extreme in collaboration (synchronous editors) [218]. Researchers have realized that the mode of collaboration depends on the task in which the developer is currently involved [98, 60] and may lie at any point in between the two extremes. Tools following this approach typically allow a user to work in three distinct modes, namely *Synchronous*, *Asynchronous*, and *Multi-Synchronous* (SAMS environment [159], FLECSE [60]). Tukan [198] further decomposes its available modes to create nine distinct modes of collaboration in which developers may be involved. All these tools however require the user to explicitly set and change the mode of collaboration. The tools at the next layer aim to reduce the user effort involved in changing the modes and automatically alter the mode based on the current task that the user is involved in.

7.3.4 Summary

The tools at the `proactive` layer supported both managers and developers alike in managing task by visualizing the development activities in the project space and providing organizational memory. However, the tools at that layer required users to be actively involved in using them. The awareness tools at this layer are an enhancement of the tools at the previous layer as they require less user involvement. The tools at this layer also address concerns that arise from using awareness tools, such as, privacy issues, distractions caused by interruptions, and the need for different modes of collaboration. These tools have succeeded in bringing presence awareness and task-oriented awareness to distributed development, but still are very much a work in progress. These tools have not yet been able to completely imitate the social cues in an organization, or be easy to install and use either (and may never be able to do so).

8 Seamless

Collaboration in software development is a challenging task [127] that is further magnified with large teams [48] and geographical separation [33]. Traditionally, software engineering and CSCW have taken two very different approaches to collaboration. Software engineering research advocate dividing the development process into distinct steps and prescribing the coordination protocols among the steps [13, 171], while CSCW researchers have recognized the importance of awareness and environmental factors in collaboration [208, 64, 218]. Both these approaches have produced tools that are extensively used in software development [74, 26], but they also suffer from drawbacks. Formal process-based systems are scalable to large teams, but are rigid and do not handle exceptions to the process very well [171]. Informal awareness-based approaches, on the other hand, are user centric, but are not scalable and overwhelm users with excessive amounts of information [113]. Researchers are currently investigating combining both these approaches so as to build on their strengths, while avoiding their drawbacks.

There are different ways in which both approaches can be integrated. Some researchers are investigating integrating the collaborative features typically provided by CSCW applications into software engineering tools [61, 194]. While others are trying to understand the development process with respect to the activities and artifacts in which developers are involved, creating software product and supporting them [220, 221].

Strubing [207] in his study found three other activities that developers consistently carry out beyond coding: "organizing their working space and process, representing and communicating design decisions and ideas, and communicating and negotiating with various stakeholders". However, coding has traditionally been considered the most important activity of a developer in software engineering. As a result, tool builders have focused on creating better programming languages and environments that facilitate coding, while ignoring other activities. In the recent past, environments that support collaborative activities and software artifacts produced during the software life cycle have become popular.

Researchers following this approach are investigating environments that either support different types of software artifacts or integrate tools that do so [211, 182, 152]. Using these environments developers usually partition their views to interact with different types of artifacts (e.g., system specification, design document, code). Modifying parts of a system specification in one tool can introduce inconsistencies with related parts of the system specified in other tools, between specifications shared by different developers, or even cause inconsistencies within the same tool. In a complex system involving multiple developers and development tools, developers are often unaware of the introduction, or even existence of such inconsistencies [96]. Mechanisms for detecting these kinds of inconsistencies and for informing developers are currently being researched [152, 94].

Booch and Brown [24] propose Collaborative Development Environments (CDE), a virtual space wherein all the stakeholders of a project, collocated or distributed, can collaborate by negotiating, brainstorming, sharing knowledge, and in essence working together to create an executable deliverable and its supporting artifacts. A similar approach by van der Hoek et al. advocate continuous coordination, a paradigm where "humans must not and cannot have their method of collaboration dictated, but should be supported flexibly with both the tools and the information to coordinate themselves and collaborate in their activities as they see fit" [218].

The aforementioned approaches exemplify the trend in research to provide a flexible unified environment that allows the user to effortlessly collaborate with additional stakeholders by using different artifacts, while providing them the flexibility to choose the level of collaboration support that is required. We observe that the three strands in the pyramid have begun to strongly blend at this layer to create a unified environment that begins to seamlessly supports communication, artifact management, and task management.

We acknowledge that collaboration is a difficult task and integrated environments do not solve all the collaboration problems, but it definitely is a step in the right direction. We also recognize that problems in global software development, such as differing time zones, language and culture barriers have yet to be investigated. We have left the top of the pyramid open to signify the need and room for further research.

9 Observations

In this paper we present a survey that takes a different perspective from the existing surveys in classifying collaborative tools. Our framework considers the user effort that is required to collaborate effectively − a critical component in deciding whether a tool is adopted or not [92]. For the purposes of this paper, we define user effort as the attention and time that a user has to expend in using the tool to communicate and coordinate their activities.

Our classification framework is in the form of a pyramid, with five layers and three strands (Figure 4). The layers in the pyramid are arranged vertically and are: (1) `functional`, (2) `defined`, (3) `proactive`, (4) `passive`, and (5) `seamless`. Tools that are at a higher layer in the pyramid provide more sophisticated automated support, thereby lowering the user effort required in using these tools. Crosscutting the layers of the pyramid are three strands that we believe are the critical needs of collaboration. These strands are: *communication*, *artifact management*, and *task management*.

We note that there is a natural ordering of the collaborative tools based on the user effort required in operating them. Our framework highlights this ordering by placing them in successive layers of the pyramid. For example, tools that facilitate communication started by allowing teams to communicate by using electronic mail in the `functional` layer, which was an advancement over the prevalent office memos. The user effort that was required in determining when, who, and how to send communication was reduced in the `defined` layer with tools archiving communication along with artifacts and sending certain notes automatically (e.g., "`foo.c` has been checked out and is ready for testing"). The third layer of the pyramid, the `proactive` layer, allowed developers to be proactive by allowing them to communicate asynchronously and monitor changes to artifacts. The tools at the next layer, the `passive` layer, reduced the effort required to set the monitors and investigate the changes by providing awareness of activities and co-developers in a passive unobtrusive manner. Tools at the `seamless` layer take a step further and attempt to provide a seamless environment that would make communication smooth and effortless.

Tools that support task management reflect a similar ordering. Task management has moved from ad hoc management techniques using tools at the `functional` layer to well-defined coordination and computation steps prescribed by tools at the `defined` layer. The undesirable rigidity enforced by tools at the `defined` layer is broken by tools at the next layer. By using tools at the `proactive` layer users can be proactive and have the flexibility to change the development process if needed. The tools at this layer further help users in their task by leveraging the existing information and history of past activities. The next layer, the `passive` layer, further enhances task management by providing collocation benefits and awareness to distributed teams. The final layer of the pyramid, `seamless`, attempts to create a continuous and smooth coordination infrastructure that allows both developers and managers to effectively manage their tasks.

We note that the changes in the functionality of the tools have gradually occurred over time, but that layer development has not been strictly historical. Sometimes, research has jumped a level (as in the case of Portholes [65], see section 7.1.1) and sometimes research has returned to a lower level to spark evolution at a higher level (as in research in social navigation, see section 6.3.3, where research on email allowed subsequent development of recommendation systems [163, 166]). Another interesting observation is that the requirements in collaboration have been significantly impacted by environmental factors and experience from usage of existing tools.

The strands in the pyramid slowly but surely intertwine with each other until they finally blend together, lending the shape of a pyramid to the framework. We note that lower layer tools generally focused on a specific strand, but as they evolved they took a broader approach encompassing more than one strand. This reflects the maturity of the field and the fact that research in different areas have started collaborating to create tools with a broader perspective.

We recognize that the manner in which collaboration is carried out depends to a large extent on environmental factors and development practices. For example, use of interfaces as part of object-oriented programming promoted stronger separation of concerns, which in turn facilitated distributed development. Distributed development, on the other hand, created the need and promoted research in collaborative tools that facilitate communication and coordination over distances. There are a number of problems in collaboration that have yet to be resolved (e.g., difficulties like collaborating across different time zones and cultures caused by global software development). We have left the top of the pyramid open to signify future research. We do not know if the `seamless` layer will be the last layer or if it will split into additional layers, and is subject to future research.

A secondary benefit of our framework is that users can relate tools in different areas (e.g., workflow, CSCW, SCM) under a single classification framework as our framework is independent of the methodology that a tool follows, and is based on the functionalities of the tool and the user effort required in using these tools.

10 Conclusions

Software development is not just an engineering effort, it also inherently involves human interactions. Moreover, the collaboration needs of software developers are not static and evolve over time. These needs are shaped by environmental factors and prevalent development practices. For example, the collaboration needs of a team vary based on the nature of the product (e.g., large government contract, in-house development, commercial shrink wrap software); the programming paradigm used (e.g., procedural languages, object-oriented programming, aspect-oriented programming); the organizational structure (e.g., collocated, distributed, open-source); and the amount of automation available to users.

Tool builders have recognized the collaboration needs of developers and are striving to incorporate collaboration features in their tool sets. However, collaboration support for software development is nontrivial and difficult to achieve by merely adding collaborative features to existing functionalities. Unfortunately, this is the approach that the majority of tool builders have taken. Tools produced by the software engineering community are typically built from a decidedly software engineering perspective, with the collaborative aspects of the tools based on bits and pieces of ideas borrowed from the CSCW community. For example, sophisticated IDEs provide awareness through the simple addition of awareness widgets. The reverse holds true as well – the CSCW community produces development tools that, while highly sophisticated in terms of collaboration mechanisms, have only marginal support for software-specific aspects, such as life cycle support, large group coordination, and diverse artifacts.

To create intrinsic collaboration support for software development, researchers need to reevaluate the traditional development practices to design development tools from the ground up to truly support collaboration. This step requires researchers in both CSCW and software engineering to work together to leverage the vast experiences in both fields. Fortunately, researchers have started to realize this and are proposing seamless and flexible environments that provide continuous coordination. In the meantime, plug-in based development environments are gaining popularity as they allow developers to find the right mix of tools that best suit their needs.

Our survey provides a framework that classifies tools based on the user effort required and the type of collaboration support provided. Our framework, thus, breaks the unseen boundaries between different research areas and allows one to relate tools in different areas. We hope that understanding the existing functionalities and relating them to each other will pave the way for a deeper understanding of collaboration needs in software development and ultimately lead to newer and better tools.

References

[1] M.S. Ackerman and C.A. Halverson. Considering an organization's memory. In *1998 ACM conference on Computer supported cooperative work*, pages 39–48, Seattle, Washington, 1998.

[2] M.S. Ackerman and C.A. Halverson. Reexamining organizational memory. *Communications of the ACM*, 43(1):58–64, 2000.

[3] L. Allen, G. Fernandez, K. Kane, D. Leblang, D. Minard, and J. Posner. Clearcase multisite: Supporting geographically-distributed software development. In *International Workshop on Software Configuration Management: ICSE SCM-4 and SCM-5 Workshops Selected Papers*, pages 194–214, 1995.

[4] V. Ambriola, G.A. Cignoni, and C. Montangero. Enacting software processes in oikos. In *4th ACM SIGSOFT Symposium on Software Developement Environments (SDE4)*, pages 183–192, Irvine, 1990.

[5] V. Ambriola, R. Conradi, and A. Fuggetta. Assessing process-centered software engineering environments. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6(3):283–328, 1997.

[6] W. Appelt. Www based collaboration with the bscw system. In *Conference on Current Trends in Theory and Informatics*, pages 66–78, 1999.

[7] R.S. Arnold. The year 2000 problem: Impact, strategies and tools. Technical report, Software Evolution Technology, Inc. Tech. Report, February 1996.

[8] R.S. Arnold and S. A. Bohner. Impact analysis - towards a framework for comparison. In *Proceedings of the Conference on Software Maintenance*, pages 292 – 301, 1993.

[9] U. Asklund and B. Magnusson. Support for consistent merge. In *Proceedings of SCM-10, 10th International Workshop on Software Configuration Management:New Practices, New Challanges and New Boundaries.*, pages 27–32, Toronto, Ontario, Canada, 2001.

[10] Computer Associates. Product solutions: http://www3.ca.com/products/, 2005.

[11] M.J. Baker and S. G. Eick. Space-filling software visualization. *Journal of Visual Languages and Computing*, 6(2):119–133, 1995.

[12] Fuggetta A. Ghezzi C. Bandinelli, S. and L. Lavazza. Spade: An environment for software process analysis, design, and enactment. In *Software process modelling and technology*, pages 223 – 247. 1994.

[13] P. Barthelmess and K.M. Anderson. A view of software development environments based on activity theory. *Computer Supported Cooperative Work*, 11(1-2):13–37, 2002.

[14] D. Beard, Humm A. Banks D. Nair A. Murugappan, P., and Y.-P. Shan. A visual calendar for scheduling group meetings. In *3rd Conference on Computer Supported Cooperative Work*, pages 279–290. ACM Press, NY, 1990.

[15] J. Begole, J.C. Tang, R. B. Smith, and N. Yankelovich. Work rhythms: analyzing visualizations of awareness histories of distributed groups. In *2002 ACM conference on Computer supported cooperative work*, pages 334–343, New Orleans, Louisiana, 2002.

[16] V. Bellotti, B. Dalal, N. Good, P. Flynn, D.G. Bobrow, and N. Ducheneaut. What a to-do: studies of task management towards the design of a personal task list manager. In *Conference on Human Factors in Computing Systems*, pages 735–742, Vienna, Austria, 2004.

[17] V. Bellotti, N. Ducheneaut, M. Howard, and I. Smith. Email-centric task management and its relationship with overload, 2002.

[18] V. Bellotti, N. Ducheneaut, M. Howard, and I. Smith. Taking email to task: the design and evaluation of a task management centered email tool. In *Human Factors in Computing Systems, SESSION: Integrating tools and tasks*, pages 345–352, Ft. Lauderdale, Florida, USA, 2003. ACM Press.

[19] I.Z. Ben-Shaul, G.T. Heineman, S.S. Popovich, P.D. Skopp, A.Z. Tong, and G. Valetoo. Integrating groupware and process technologies in the oz environment. In *Ninth International Software Process Workshop: The Role of Humans in the Process*, pages 114–116, Airlie VA, 1994. IEEE Computer Society Press.

[20] B. Berliner. Cvs ii: Parallelizing software development. In *USENIX Winter 1990 Technical Conference*, pages 341–352, 1990.

[21] V. Berzins. Software merge: Semantics of combining changes to programs. *ACM Transactions, Programming Languages and Systems*, 16(6):1875–1903, 1994.

[22] E.A. Bier and S. Freeman. Mmm: A user interface architecture for shared editors on a single screen. In *ACM Symposium on User Interface Software and Technology*, pages 79–86, 1991.

[23] D. Binkley, S. Horwitz, and T. Reps. Program integration for languages with procedure class. *ACM Transactions, Software Engineering and Methodology*, 4(1):3–35, 1995.

[24] G. Booch and A. Brown. Collaborative development environments. *Advances in Computers*, 59, 2003.

[25] A. Borning and M. Travers. Two approaches to casual interaction over computer and video networks. In *SIGCHI conference on Human factors in computing systems: Reaching through technology*, pages 13–19, New Orleans, Louisiana, 1991.

[26] E. Bradner, W. Kellogg, and T. Erickson. The adoption and use of 'babble': A field study of chat in the workplace. In *the Sixth European conference on Computer supported cooperative work*, pages 139–158, Copenhagen, Denmark, 1999. Kluwer Academic Publishers.

[27] S. Brave, H. Ishii, and A. Dahley. Tangible interfaces for remote collaboration and communication. In *1998 ACM conference on Computer supported cooperative work*, pages 169–178, Seattle, Washington, USA, 1998.

[28] F. P. Brooks Jr. The mythical man-month. *Datamation*, 20(12):44–52, 1974.

[29] J. Buffenbarger. Syntactic software merging. In *Software Configuration Management: ICSE SCM-4 and SCM-5 Workshops Selected Papers*, pages 153–172, 1995.

[30] Bugzilla. http://www.bugzilla.org.

[31] S.A. Bull. Introduction to flowpath. Technical report, May 1992.

[32] J.J. Cadiz, S.R. Fussell, R. E. Kraut, J.F. Lerch, and W.L. Scherlis. Awareness monitor: A coordination tool for asynchronous, distributed work teams. Technical report, Unpublished manuscript, Carnegie Mellon University, 1999.

[33] E. Carmel. *Global Software Teams: Collaborating Across Borders and Time Zones.* Prentice-Hall: Englewood Cliffs NJ, 1st edition edition, 1999.

[34] A. Carzaniga, D.S. Rosenblum, and A.L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 2001. ACM Trans. Comp. Sys.

[35] C.S. Chatfield and J.D. Johnson. *Microsoft Project 2000 Step by Step.* Microsoft Press;, bk and cd-rom edition edition, 2000.

[36] Li-Te Cheng, S. Hupfer, S. Ross, and J. Patterson. Jazzing up eclipse with collaborative tools. In *18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications / Eclipse Technology Exchange Workshop*, pages 102–103, Anaheim, CA, 2003.

[37] M.C. Chu-Carroll and S. Sprenkle. Coven: Brewing better collaboration through software configuration management. In *Eighth International Symposium on Foundations of Software Engineering*, pages 88–97, 2000.

[38] A. Cimitile, F. Lanubile, and G. Visaggio. Traceability based on design decisions. In *Proceedings of International Conference on Software Maintenance*, pages 309–317, Orlando, FL, 1992.

[39] E. Clayberg and D. Rubel. *Eclipse: Building Commercial-Quality Plug-ins.* Eclipse Series. Addison-Wesley Professional, 2004.

[40] W.F. Clocksin and C.S. Mellish. *Programming in Prolog.* Springer-Verlag, NY, USA, 3rd edition, 1987.

[41] J. Conklin and M.L. Begeman. gibis: A hypertext tool for exploratory policy discussion. In *the Conference on Computer Supported Cooperative Work (CSCW '88)*, pages 140–152. ACM Press, NY, 1988.

[42] R. Conradi, M. Hagaseth, J-O. Larsen, M.N. Nguyn, B.P. Munch, P.H. Westby, W. Zhu, M.L Jaccheri, and C. Liu. Epos: object-oriented cooperative process modelling. In *Software process modelling and technology*, pages 33 – 70. 1994.

[43] R. Conradi, M.L. Jaccheri, C. Mazzi, M.N. Nguyen, and A. Aarsten. Design, use and implementation of spell, a language for software process modelling and evolution. In *Second European Workshop on Software Process Technology*, pages 167–177, 1992.

[44] R. Conradi and B. Westfechtel. Version models for software configuration management. *ACM Computing Surveys*, 30(2):232–282, 1998.

[45] W.B. Croft and L.S. Lefkowitz. Using a planner to support office work. In *ACM Conference on Office Information Systems*, pages 55–62. ACN, NY, 1988.

[46] D. Cubranic, G. Murphy, and K Booth. Hipikat: A developer's recommender. In *OOPSLA*, 2002.

[47] D. Cubranic and Gail C. Murphy. Hipikat: Recommending pertinent software artifacts. In *International Conference on Software Engineering*, pages 408 – 418, Portland, Oregon, 2003.

[48] B. Curtis, H. Krasner., and N. Iscoe. A field study of the software design process for large systems. *Communications of the ACM*, 31(11):1268–1287, 1988.

[49] E. Cutrell, M. Czerwinski, and E. Horvitz. Notification, disruption, and memory: Effects of messaging interruptions on memory and performance. In *Interact*, Tokyo, Japan, 2001.

[50] M. Czerwinski, E. Cutrell, and E. Horvitz. Instant messaging and interruption: Influence of task type on perform. In *OZCHI 2000 Conference*, pages 356–361, 2000.

[51] S. Dart. Concepts in configuration management systems. In *Third International Workshop on Software Configuration Management*, pages 1–18, 1991.

[52] E. Dashofy, A. van der Hoek, and R. Taylor. A comprehensive approach for the development of modular software architecture description languages. *In ACM Transactions on Software Engineering and Methodology, to appear.*, 2005.

[53] C. R. B. de Souza, P. Dourish, D. Redmiles, S. Quirk, and E. Trainer. From technical dependencies to social dependencies. In *Workshop in Social Networks for Design and Analysis: Using Network Information in CSCW during the ACM Conference on Computer-Supported Cooperative Work*, Chicago, IL, 2004.

[54] C.R.B. de Souza, S.D. Basaveswara, and D Redmiles. Lessons learned using notification servers to support application awareness. In *Meeting of the Human Computer Interaction Consortium (HCIC 2002 Frasier, CO)*, 2002.

[55] C.R.B. de Souza, D.F. Redmiles, G. Mark, J. Penix, and M. Sierhuis. Management of interdependencies in collaborative software development. In *ACM-IEEE International Symposium on Empirical Software Engineering (ISESE 2003),*, pages 294–303, 2003.

[56] Y. Dekel and S. Ross. Eclipse as a platform for research on interruption management in software development. In *Eclipse Technology Exchange Workshop at OOPSLA'04*, Vancouver, BC, Canada,, 2004.

[57] A.R. Dennis, F.G. Joey, L.M. Jessup, J.F. Nunamaker, and D.R. Vogel. Information technology to support electronic meetings. *Management Information Systems*, 12(4):591–619, Dec 1988.

[58] G. DeSanctis and R. B. Gallupe. A foundation for the study of group decision support systems. *Management Science*, 33(5):589–609, 1987.

[59] P. Dewan and R. Choudhary. A high-level and flexible framework for implementing multi-user interfaces. *ACM Transactions on Information Systems*, 10(4):345–380, 1992.

[60] P. Dewan and J. Riedl. Toward computer-supported concurrent software engineering. *IEEE Computer*, 26(1):17–27, 1993.

[61] E. Di Nitto and A. Fuggetta. Integrating process technology and cscw. In *4th European Workshop on Software Process Technology*, volume 413, pages 154–161, 1995.

[62] Diget. Autoplan. 2004.

[63] S.E. Dossick and G.E. Kaiser. A metadata-based distributed software development environment. In *Seventh European Software Engineering Conference together with the Seventh ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, pages 464–475, 1999.

[64] P. Dourish and V. Bellotti. Awareness and coordination in shared workspaces. In *ACM Conference on Computer-Supported Cooperative Work*, pages 107–114, Monterey, California, USA, 1992.

[65] P. Dourish and S. Bly. Portholes: Supporting awareness in a distributed work group. In *Human Factors in Computing Systems, CHI'92*, pages 541–547, New York, 1992.

[66] N. Ducheneaut and V. Bellotti. E-mail as habitat: an exploration of embedded personal information management. *interactions*, Volume 8(Issue 5):30 – 38, 2001.

[67] C. Ebert and P. De Neve. Surviving global software development. *IEEE Software*, 18(2):62–69, 2001.

[68] D.A. Edwards and M.S. McKendry. Exploiting read-mostly workloads in filenet file system. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 58–70, 1989.

[69] S. G. Eick, J.L. Steffen, and Jr. Summer, E.E. Seesoft-a tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering, Special issue on software measurement principles, techniques, and environments*, 18(11):957–958, 1992.

[70] C.A. Ellis. Information control nets: A mathematical model of office information flow. In ACM Press, editor, *In Proceedings of the ACM Conference on Simulation, Measurement and Modeling of Computer Systems*, pages 225–240, Boulder, Colorado, 1979.

[71] C.A. Ellis, S.J. Gibbs, and G.L. Rein. Design and use of a group editor. In *Engineering for Human Computer Interaction*, pages 13–25, North Holland/Elsevier, Amsterdam, 1990.

[72] C.A. Ellis, S.J. Gibbs, and G.L. Rein. Groupware—some issues and experiences. *communications of the ACM*, 34(1):38–58, 1999.

[73] C.A. Ellis and N. Naffah. Design of office information systems. 1987.

[74] J. Estublier, D. Leblang, A. van der Hoek, R. Conradi, G. Clemm, W. F. Tichy, and D.W. Weber. Impact of software engineering research on the practice of software configuration management. *to appear (ACM Transactions of Software Engineering and Methodology)*, 2005.

[75] B. A. Farshchian. Integrating geographically distributed development teams through increased product awareness. *Information Systems Journal*, 26(3):123–141, 2001.

[76] P.H. Feiler. Configuration management models in commercial environments. Technical Report SEI-91-TR-07, Software Engineering Institute, Carnegie Mellon University, 1991.

[77] A. Finkelstein, G. Kramer, and B. Nuseibeh. *Software Process Modeling and Technology.* Research Studies Press. 1994.

[78] R.S. Fish, R. E. Kraut, and M.D.P. Leland. Quilt: a collaborative tool for cooperative writing. *ACM SIGOIS Bulletin*, 9(2-3):30–37, April 1988.

[79] G. Fitzpatrick, S. Kaplan, and T. Mansfield. Physical spaces, virtual places and social worlds: A study of work in the virtual. In *Computer-Supported Collaborative Work '96*, pages 334–343, 1996.

[80] G. Fitzpatrick, S. Kaplan, T. Mansfield, D. Arnold, and B. Segall. Supporting public availability and accessibility with elvin: Experiences and reflections. *Computer Supported Cooperative Work*, 2002.

[81] G. Fitzpatrick, T. Mansfield, S. Kaplan, D. Arnold, T. Phelps, and B. Segall. Augmenting the workaday world with elvin. In *Sixth European Conference on Computer Supported Cooperative Work*, pages 431–451, 1999.

[82] J. Froehlich and P. Dourish. Unifying artifacts and activities in a visual tool for distributed software development teams. In *Proceedings of the International Conference on Software Engineering*, pages 387–396, Edinburgh, UK, 2004.

[83] L. Fuchs, Pankoke-Babatz U., and W. Prinz. Supporting cooperative awareness with local event mechanism: The group desk system. In *European Computer Supported Cooperative Work (ECSCW'95)*, pages 247–262. Kluwer, Dordrecht, 1995.

[84] P. Godefroid, J. Herbsleb, L.J. Jagadeesany, and D. Li. Ensuring privacy in presence awareness: an automated verification approach. In *Proceedings of the 2000 ACM conference on Computer supported cooperative work*, pages 59–68, Philadelphia, Pennsylvania, USA, 2000. ACM Press.

[85] S. Goldmann, J. Mnch, and H. Holz. Milos: A model of interleaved planning, scheduling, and enactment. In *Web-Proceedings of the 2nd Workshop on Software Engineering over the Internet*, Los Angeles, CA, 1999.

[86] A. Graveline, C. Geisler, and M. Danchak. Teaming together apart: emergent patterns of media use in collaboration at a distance. In *18th annual ACM international conference on Computer documentation: technology and teamwork*, pages 381–393, Cambridge,Massachusetts, USA, 2000. IEEE Educational Activities Department.

[87] S. Greenberg and R. Bohnet. Groupsketch: A multi-user sketchpad for geographically-distributed small groups. *In Proceedings of Graphics Interface*, 1991.

[88] S. Greenberg and M. Roseman. Groupware toolkits for synchronous work. In *Computer-Supported Cooperative Work (Trends in Software)*, volume 7, pages 135–168, 1999.

[89] R.E. Grinter. Using a configuration management tool to coordinate software development. In *Conference on Organizational Computing Systems*, pages 168–177, 1995.

[90] R.E. Grinter. Supporting articulation work using software configuration management systems. *Computer Supported Cooperative Work*, 5(4):447–465, 1996.

[91] R.E. Grinter, J.D. Herbsleb, and D.E. Perry. The geography of coordination: Dealing with distance in r&d work. In *ACM Conference on Supporting Group Work (GROUP 99)*, pages 306–315, Phoenix, AZ, 1999.

[92] J. Grudin. Why cscw applications fail: problems in the design and evaluation of organization of organizational interfaces. In *Proceedings of the 1988 ACM conference on Computer-supported cooperative work*, pages 85–93, Portland, Oregon, 1988.

[93] J. Grudin. Cscw: History and focus. *IEEE Computer*, 27(5):19–27, 1994.

[94] J.C. Grundy and J.G. Hosking. Constructing integrated software development environments with mviews. *Interactional Journal Applied Software Technology*, 2(3/4):133–160, 1996.

[95] J.C. Grundy and J.G. Hosking. Serendipity: integrated environment support for process modelling, enactment and work coordination. *Automated Software Engineering*, 5(1):27–60, 1998.

[96] J.C. Grundy, J.G. Hosking, and W.B. Mugridge. Inconsistency management for multi-view software development environments. *IEEE Transactions on Software Engineering: Special Issue on Managing Inconsistency in Software Development*, Vol. 24(No. 11), 1998.

[97] C. Gutwin and S. Greenberg. Workspace awareness for groupware. In *CHI'96 Conference Companion on Human Factors in Computing Systems*, pages 208–209, 1996.

[98] A. Haake. Facilitating orientation in shared hypermedia workspaces. In *International ACM SIGGROUP Conference on Supporting Group Work*, pages 365–374, 1999.

[99] A. Haake and J.M. Haake. Take cover: Exploiting version support in cooperative systems. In *INTERCHI'93*, pages 406–413, 1993.

[100] J.M. Haake and B. Wilson. Supporting collaborative writing of hyperdocuments in sepia. In *1992 ACM conference on Computer-supported cooperative work*, pages 138–146, Toronto, Ontario, Canada, 1992. ACM Press.

[101] C. Heath, M. Jirotka, P. Luff, and J. Hindmarsh. Unpacking collaboration: the interactional organisation of trading in a city dealing room. *Computer Supported Cooperative Work*, 3(2):147–165, 1994.

[102] C. Heath and P. Luff. Collaboration and control: Crisis management and multimedia technology in london underground line control rooms. *Computer Supported Cooperative Work*, 1(12):69–94 rooms, 1992.

[103] D. Heimbigner, L. Osterweil, and S. Sutton. Appl/a: A language for managing relations among software objects and processes. Technical report, University of Colorado, Department of Computer Science, Boulder, Colorado, 1987.

[104] J. Herbsleb, D.L. Atkins, B.G. Boyer, M. Handel, and T. A. Finholt. Introducing instant messaging and chat in the workplace. In *Proceedings of the SIGCHI conference on Human factors in computing systems: Changing our world, changing ourselves*, pages 171–178, Minneapolis, Minnesota, USA, 2002.

[105] J. Herbsleb and R. E. Grinter. Splitting the organization and integrating the code: Conway's law revisited. In *Proceedings of the 21st international conference on Software engineering*, pages 85–95, Los Angeles, CA, USA, 1999.

[106] J. Herbsleb and D. Moitra. Global software development. *IEEE Software*, 18(2):16–20, 2001.

[107] J. D. Herbsleb and R. E. Grinter. Architectures, coordination, and distance: Conway's law and beyond. *IEEE Software*, pages 63–70, 1999.

[108] J.D. Herbsleb, A. Mockus, T. A. Finholt, and R.E. Grinter. Distance, dependencies, and delay in a global collaboration. In *Proceedings of the 2000 ACM conference on Computer supported cooperative work*, pages 319–328, Philadelphia, PA, 2000.

[109] J. Hill and C. Gutwin. Awareness support in a groupware widget toolkit. In *Proceedings of the 2003 international ACM SIGGROUP conference on Supporting group work*, pages 258–267, Sanibel Islands, Florida, USA, 2003. ACM Press.

[110] W. C. Hill and et al. Edit wear and read wear. In *Proceedings of the ACM SIGCHI Conference on Human Factors and Computing Systems*, pages 3–9, 1992.

[111] S. Horwitz, Prins. J., and Reps. T . Integrating non-interfering versions of programs. *ACM Transactions, Programming Languages and Systems*, 11(3):345–387, 1989.

[112] G. P. Huber. Issues in the design of group support systems. *MIS Quarterly*, September 1984.

[113] J.M. Hudson, J. Christensen, W.A. Kellogg, and T. Erickson. "i'd be overwhelmed, but it's just one more thing to do": availability and interruption in research management. In *Proceedings of the SIGCHI conference on Human factors in computing systems: Changing our world, changing ourselves*, pages 97–104, Minneapolis, Minnesota, 2002.

[114] S. Hudson and R. King. The cactis project: Database support for software environments. *IEEE Transactions on Software Engineering*, 14(6):709–719, 1988.

[115] S.E. Hudson and I. Smith. Techniques for addressing fundamental privacy disruption tradeoffs in awareness support systems. In *ACM, Computer-Supported Cooperative Work (CSCW96)*, Boston, Massachusetts, USA, 1996.

[116] J.W. Hunt and T.G. Szymanski. A fast algorithm for computing longest common subsequences. *Communications of the ACM*, 20(5):350–353, 1977.

[117] S. Hupfer, L.-T. Cheng, S. Ross, and J. Patterson. Reinventing team spaces for a collaborative development environment. In *"Beyond Threaded Conversation" Workshop, in CHI 2005 (to appear)*, 2005.

[118] Software Maintenance $ Development Systems Inc. Aide de camp product overview. Technical report, 1994 1994.

[119] K. Jensen. Colored petri nets: Basic concepts, analysis methods, and practical use. *Springer-Verlag*, 3:265, 1997.

[120] B. Johnson. Treeviz: Treemap visualization of hierarchically structured information. In John Bennett Penny Bauersfeld and Gene Lynch, editors, *Proceedings of the Conference on Human Factors in Computing Systems*, pages 369–372, Monterey, California, USA, 1992. ACM Press.

[121] R. E. Kaliouby and P. Robinson. Faim: integrating automated facial affect analysis in instant messaging. In *Proceedings of the 9th international conference on Intelligent user interface*, pages 244 – 246, Funchal, Madeira, Portugal, 2004.

[122] M. Kantor and D. Redmiles. Creating an infrastructure for ubiquitous awareness. In *Eighth IFIP TC 13 Conference on Human-Computer Interaction (INTERACT 2001),*, pages 431–438, Tokyo, Japan, 2001.

[123] D. W. Karolak. *Global software development: managing virtual teams and environments*. IEEE Computer Society, 1999.

[124] D. Kirsch. The context of work. *Human-Computer Interaction*, Vol 16:305–322, 2001.

[125] M.J. Knister and A. Prakash. Distedit: a distributed toolkit for supporting multiple group editors. In *Proceedings of the 1990 ACM conference on Computer-supported cooperative work*, Los Angeles, CA, 1990.

[126] K. Kraemer and J. L. King. Computer-based systems for cooperative work and group decision making. *ACM Computing Surveys (CSUR)*, 20(2):115–146, 1988.

[127] R.E. Krant and L.A. Streeter. Coordination in software development. *Communications of the ACM*, 38(3):69–81, 1995.

[128] R. Kraut, C. Egido, and J. Galegher. Patterns of contact and communication in scientific research collaboration. In *Proceedings of the 1988 ACM conference on Computer-supported cooperative work*, pages 1–12, Portland, Oregon, 1988.

[129] M. Lanza. The evolution matrix: Recovering software evolution using software visualization techniques. In *2001 International Workshop on the Principles of Software Evolution*, pages 28–33, 2001.

[130] D. Leblang. *The CM challenge: configuration management that works*, volume 2 of Trends in Software of *In Configuration management*. John Wiley $ Sons, Inc., New York,, 1995.

[131] D.B. Leblang and G.D. McLean. Configuration management for large-scale software development efforts. In *Proc. Workshop Software Eng. Environments for Programming-in-the-Large*, pages 122–127, 1985.

[132] A. Lee, A. Girgensohn, and K. Schlueter. Nynex portholes: initial user reactions and redesign implications. In *Proceedings of the international ACM SIGGROUP conference on Supporting group work: the integration challenge: the integration challenge*, pages 385–394, 1997.

[133] J. Lee. Sibyl: A qualitative decision management system. *Artificial Intelligence at MIT: Expanding Frontiers*, 1, 1990.

[134] J. Lee, G. Yost, and PIF Working Group. The pif process interchange format and framework. Technical report, Technical Report, University of Hawaii Information and Computer Science Department, February 1996.

[135] F. Leymann and W. Altenhuber. Managing business process as an information resource. *IBM Systems Journal*, 33(2), 1994.

[136] Y-J. lin and S.P. Reiss. Configuration management with logical structures. In *Proceedings of the 18th international conference on Software engineering*, pages 298–307, Berlin, Germany, 1996. IEEE Computer Society.

[137] J.P. Loyall and S.A. Mathisen. Using dependency analysis to support the software maintenance process. In *Proceedings of International Conference of Software Maintenance*, pages 282–291, Montral, Quebec, Canada, 1993.

[138] L. Lvstrand. Being selectively aware with the khronika system. In *In Proceedings of the European Conference on Computer Supported Cooperative Work, ECSCW(91)*, pages 265–278, Amsterdam, NL, 1991. ACM Press, New York.

[139] B. Magnusson and U. Asklund. Fine grained version control of configurations in coop/orm. In *Sixth International Workshop on Software Configuration Management*, volume 1167, pages 31–48, 1996.

[140] B. Magnusson, U. Asklund, and S. Minr. Fine-grained revision control for collaborative software development. In *Proceedings of ACM SIGSOFT '93: Symposium on Foundations of Software Engineering*, pages 33–41, Los Angeles, CA, USA, 1993.

[141] T. W. Malone and K. Crowston. The interdisciplinary study of coordination. *ACM Computing Surveys (CSUR)*, 26(1):87–119, 1994.

[142] T. W. Malone, K.R. Grant, F.A. Turbak, S.A. Brobst, and M.D. Cohen. Intelligent information-sharing systems. *Communications of the ACM*, (30):390–402, 1987.

[143] T. Mansfield, S. Kaplan, G. Fitzpatrick, P. Phelps, M. Fitzpatrick, and R.N. Taylor. Toward locales: supporting collaboration with orbit. *Journal of Information and Software Technology*, 41(6):367–382, 1999.

[144] Mantis. http://www.mantisbt.org/.

[145] G. Mark. Extreme collaboration. *Communications of the ACM*, 45(6):89–93, 2002.

[146] D.R. McCarthy and S.K. Sarin. Workflow and transactions in inconcert. *IEEE Data Engineering*, 16(2):53–56, 1993.

[147] D.W. McDonald. Recommending collaboration with social networks: a comparative evaluation. In *Proceedings of the conference on Human factors in computing systems*, pages 593–600, Ft. Lauderdale, Fl, 2003.

[148] D.W. McDonald and M.S. Ackerman. Expertise recommender: a flexible recommendation system and architecture. In *Proceedings of the 2000 ACM conference on Computer supported cooperative work*, pages 231–140, Philadelphia, PA, USA, 2000. ACM Press.

[149] D.C. McFarlane. Comparison of four primary methods for coordinating the interruption of people in human-computer interactions. *Human-Computer Interaction*, 17(1):63–139, 2002.

[150] L.J. McGuffin and G. Olson. Shredit: A shared electronic workspace. Technical Report 45, Cognitive Science and Machine Intelligence Laboratory, Tech report: 45, University of Michigan, Ann Arbor, 1992.

[151] T. Mens. A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering*, 28(5):449–462, 2002.

[152] S. Meyers. Difficulties in integrating multiview development systems. *IEEE Software*, 8(1):49–57, 1991.

[153] A. Mockus and J. Herbsleb. Expertise browser: A quantitative approach to identifying expertise. In *2002 International Conference on Software Engineering*, 2002.

[154] Fielding R. Mockus, A. and J.D. Herbsleb. Two case studies of open source software development: Apache and mozilla. *ACM Transactions on Software Engineering and Methodology.*, 11(3):309–346, 2002.

[155] C. Mohan. Tutorial: State of the art in workflow management system research and products. In *A tutorial at the ACM SIGMOD International Conference on Management Data*, 1996.

[156] P. Molli. Coo-transactions: Supporting cooperative work. In R. Conradi, editor, *Proceedings of Seventh International Workshop on Software Configuration Management (SCM7)*, pages 128–141, Boston, MA, USA, 1997. Springer.

[157] P. Molli, H. Skaf-Molli, and C. Bouthier. State treemap: an awareness widget for multi-synchronous groupware. In *Seventh International Workshop on Groupware*, 2001.

[158] P. Molli, H. Skaf-Molli, and G. Oster. Divergence awareness for virtual team through the web. In *Integrated Design and Process Technology*, 2002.

[159] P. Molli, H. Skaf-Molli, G. Oster, and S. Jourdain. Sams: Synchronous, asynchronous, multisynchronous environments. In *Seventh International Conference on CSCW in Design*, Rio de Janeiro, Brazil, 2002.

[160] C. Montangero and V. Ambriola. Oikos: Constructing process centered sde's. In *Software Process Modelling and Technology*, pages 131 – 151. 1994.

[161] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[162] B. Nardi, S. Whittaker, and E. Bradner. Interaction and outeraction: Instant messaging in action. In *Computer Supported Cooperative Work*, Philadelphia, PA, 2000.

[163] B.A. Nardi, S. Whittaker, E. Isaacs, M. Creech, J. Johnson, and J. Hainsworth. Contactmap: Integrating communication and information through visualizing personal social networks. *Communications of the ACM*, 45(4):89–95, 2002.

[164] G. Nutt. *The Formulation and Application of Evaluation Nets.* Phd thesis, 1972.

[165] G. Nutt. The evolution towards flexible workflow systems. In *Distributed Systems Engineering*, volume 3(4), pages 276–294, 1996.

[166] H. Ogata, Y. Yano, N. Furugori, and Q. Jin. Computer supported social networking for augmenting cooperation. *Computer Supported Cooperative Work: The Journal of Collaborative Computing*, 10:189–209, 2001.

[167] G.M. Olson and J.S. Olson. Distance matters. *Human-Computer Interaction*, 15(2and3):139–178, 2000.

[168] OMG. *CORBACos: Notification Service Specification v1.0.1.* 2002.

[169] C. O'Reilly, D. Bustard, and P. Morrow. The war room command console (shared visualization for inclusive team coordination). In *To Appear (2nd ACM Symposium on Software Visualization)*, St. Louis, Missouri, USA, 2005.

[170] C. O'Reilly, P. Morrow, and D. Bustard. Improving conflict detection in optimistic concurrency control models. In *Proceedings of the Eleventh International Workshop on Software Configuration Management*, pages 191–205, Portland, Oregon, 2003.

[171] L. Osterweil. Software processes are software too. In *Proceedings of the 9th International Conference on Software Engineering*, pages 2–13, Monterey, CA, 1987.

[172] D.L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[173] D.L. Parnas and P.C. Clements. A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering, SE*, 12(2):251–257, 1986.

[174] J. Patterson, M. Day, and J. Kucan. Notification servers for synchronous groupware. In *Proceedings of the 1996 ACM conference on Computer supported cooperative work*, pages 122–129, Boston, Massachusetts, USA, 1996.

[175] M.C. Paulk, B. Curtis, M.B. Chrissis, and V.W. Weber. Capability maturity model, version 1.1. *IEEE Software*, 10(4):18–27, 1993.

[176] D.E. Perry, H.P. Siy, and L.G. Votta. Parallel changes in large-scale software development: An observational case study. *ACM Transactions on Software Engineering and Methodology*, 10(3):308–337, 2001.

[177] J.L. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice Hall, 1981.

[178] W. Prinz. Nessie: an awareness environment for cooperative settings. In *Sixth European conference on Computer supported cooperative work*, pages 391–410, Copenghagen, Denmark, 1999. Kluwer Academic Publishers.

[179] P. Procter. *Cambridge International Dictionary of English.* Cambridge University Press, 1995.

[180] D. Ramduny, A. Dix, and T. Rodden. Exploring the design space for notification servers. In *1998 ACM conference on Computer supported cooperative work*, pages 227–235, Seattle, Washington, USA, 1998.

[181] E.S. Raymond. *The Cathedral $ the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary.* O'Reilly,, 2001.

[182] S.P. Reiss. Simplifying data integration: The design of the desert software development environment. In *Proceedings of the 18th International Conference on Software Engineering*, pages 398–407, Berlin, Germany, 1996. IEEE Computer Society.

[183] W. Rigg, C. Burrows, and P. Ingram. *Configuration Management Tools.* PhD thesis, 1995.

[184] k. Rivera, N.J. Cooke, A.L. Rowe, and J.A. Bauhs. Conveying emotion in remote computer-mediated-communication. In *Conference companion on Human factors in computing systems*, pages 95–96, Boston, Massachusetts, USA, 1994. ACM Press.

[185] J. Robbins and D. Redmiles. Software architecture critics in the argo design environment. *Knowledge-Based Systems*, 11(1):47–60, 1998.

[186] T.J. Robertson, S. Prabhakararao, M. Burnett, C. Cook, J.R. Ruthruff, and L. Beckwith. Impact of interruption style on end-user debugging. In *2004 conference on Human factors in computing systems*, pages 287–294, Vienna, Austria, 2004. ACM Press.

[187] M.J. Rochkind. The source code control system. *IEEE Transactions on Software Engineering*, SE-1(4):364–370, 1975.

[188] M. Roseman and S. Greenberg. Groupkit: a groupware toolkit for building real-time conferencing applications. In *Proceedings of the 1992 ACM conference on Computer-supported cooperative work*, pages 43–50, Toronto, Ontario, Canada, 1992.

[189] M. Roseman and S. Greenberg. Teamrooms: Network places for collaboration. In *In Proceedings of ACM Computer Supported Cooperative Work*, pages 325–333, 1996.

[190] D.S. Rosenblum and A. Wolf. A design framework for internet-scale event observation and notification. In *Sixth European Software Engineering Conf./ACM SIGSOFT Fifth Symposium on the Foundations of Software Engineering*, pages 344–360, Zurich, Switzerland, 1997.

[191] P. Sachs. Transforming work: Collaboration, learning, and design. *Communications of the ACM*, 38(9):36–44, September 1995.

[192] W. Sack. Conversation map: A content-based usenet newsgroup browser. In *Proceedings of the International Conference on Intelligent User Interfaces*, pages 233–240, New Orleans, 2000. ACM Press.

[193] Visual Source Safe. http://msdn.microsoft.com/vstudio/previous/ssafe/.

[194] A. Sarma, Z. Noroozi, and A. van der Hoek. Palantr: Raising awareness among configuration management workspaces. In *Twentyfifth International Conference on Software Engineering*, pages 444–454, Portland, Oregon, USA, 2003.

[195] A. Sarma and A. van der Hoek. Visualizing parallel workspace activities. In *IASTED International Conference on Software Engineering and Applications (SEA)*, pages 435–440, Marina Del Rey, California, 2003.

[196] .S Sawyer and P.J. Guinan. Software development: Processes and performance. *IBM Systems Journal*, 37(4), 1998.

[197] K. Schmidt. The problem with 'awareness': Introductory remarks on 'awareness in cscw'. *Computer Supported Cooperative Work*, 11(3):285–298, 2002.

[198] T. Schmmer and J.M. Haake. Supporting distributed software development by modes of collaboration. In *Seventh European Conference on Computer Supported Cooperative Work*, pages 79–98, 2001.

[199] R. S. Silva Filho, C.R.B. de Souza, and D.F. Redmiles. The design of a configurable, extensible and dynamic notification service. In *Second International Workshop on Distributed Event-Based Systems (DEBS 2003), In conjunction with The ACM SIGMOD/PODS Conference*, San Diego, 2003.

[200] R.S. Silva Filho, C. R. B. de Souza, and D. Redmiles. Design and experiments with yancees, a versatile publish-subscirbe service. Technical report, TR-UCI-ISR-04-1, University of California, Irvine. Irvine, CA, April 2004.

[201] M. Sohlenkamp and G. Chwelos. Integrating communication, cooperation, and awareness: the diva virtual office environment. In *Proceedings of the 1994 ACM conference on Computer supported cooperative work*, pages 331–343, Chapel Hill, North Carolina, USA, 1994.

[202] M. Sohlenkamp, W . Prinz, and L. Fuchs. Poliawac: Design and evaluation of an awareness-enhanced groupware client. *AI and Society: Special issue on computer-supported cooperative*, 14(1):31 – 47, 2000.

[203] D.G. Stefik, M .and Bobrow, G. Foster, S. Lanning, and D. Tatar. Wysiwis revised: early experiences with multiuser interfaces. *ACM Transactions on Information Systems (TOIS)*, 5(2):147–167, 1987.

[204] L. Steiger. *Recovering the evolution of object oriented software systems using a flexible query engine.* PhD thesis, University of Bern, (Diploma thesis), 2001.

[205] M-A. D. Storey, D. Cubranic, and D.M. German. On the use of visualization to support awareness of human activities in software development: A survey and a framework. In *Proceedings of 2nd ACM Symposium on Software Visualization, 2005 (to appear)*, St. Loius, Missouri, USA, 2005.

[206] P.D. Stotts and R. Furuta. alphatrellis: A system for writing and browsing petri-net-based hypertext. In *Proceedings of the 10th International Conference on Application and Theory of Petri Nets*, pages 312–328, Bonn, Germany, 1989.

[207] J. Strubing. Designing the working process: What programmers do besides programming. *User-Centered Requirements for Software Engineering Environments*, 1994.

[208] L.A. Suchman. *Plans and Situated Actions: The Problem of Human-Machine Communication.* Cambridge University Press, New York, 1987.

[209] SunMicrosystems. *Java Message Service API.* 2003.

[210] J.C Tang and M. Rua. Montage: providing teleproximity for distributed groups. In *Proceedings of the SIGCHI conference on Human factors in computing systems: celebrating interdependence*, pages 37–43, Boston, MA, 1994.

[211] A. Tate and K. Wade. Simplifying development through activity-based change management. White paper, IBM Software Group, October 2004.

[212] R. Taylor, F. Belz, L. Clarke, R. Osterweil, L .and Selby, J. Wileden, A. Wolf, and M. Young. Foundations for the arcadia environment architecture. In *In Proceedings of the Software Engineering Symposium on Practical software development environments*, pages 1–13, 1988.

[213] Telelogic. Cm/synergy.

[214] W.F. Tichy. Rcs, a system for version control. *Software - Practice and Experience*, 15(7):637–654, 1985.

[215] W.F. Tichy. Tools for software configuration management. In *In Proc. of the Int. Workshop on Software Version and Configuration Control*, pages 1–20, Grassau, 1988.

[216] Tigris.org. Subversion.

[217] K. Tollmar, O. Sandor, and A. Schmer. Supporting social awareness @ work design and experience. In *Proceedings of the 1996 ACM conference on Computer supported cooperative work*, pages 298–307, Boston, Massachusetts, USA, 1996. ACM Press.

[218] A. van der Hoek and et.al. Continuous coordination: A new paradigm for collaborative software engineering tools. In *Proceedings of Workshop on WoDISEE*, Scotland, 2004.

[219] I. Vessey and A. P. Sravanapudi. Case tools as collaborative support technologies. *Communications of the ACM*, vol. 38:83–95, 1995.

[220] L. Votta. By the way, has anyone studied real programmers yet? In *Ninth International Software Process Workshop*, Reston, Virginia, 1994.

[221] G. Weinberg. *The Psychology of Computer Programming*. Dorset House Publishing, 1989.

[222] B. Westfechtel, B.P. Munch, and R. Conradi. A layered architecture for uniform version management. *IEEE Transactions on Software Engineering*, 27(12):1111–1133, 2001.

[223] D. Wiborg Weber. Benefits of task-based change management. White paper, Telelogic, September 2003.

[224] J.C. Wileden, A.L. Wolf, C.D. Fisher, and P.L. Tarr. Pgraphite: an experiment in persistent typed object management. *ACM SIGPLAN Notices*, 24(2):130–142, 1989.

[225] T. Winograd and F. Flores. *Understanding Computers and Cognition: A New Foundation for Design*. Addison-Wesley Professional, 1 edition, 1987.

[226] Y. Ye and G. Fischer. Information delivery in support of learning reusable software components on demand. In *7th international conference on Intelligent user interfaces*, pages 159–166, San Francisco, California, USA, 2002. ACM Press.

[227] M. Young, R.N. Taylor, and D. B. Troup. Software environment architectures and user interface facilities. *IEEE Transactions on Software Engineering*, 14(6):697–708, 1988.

[228] P. Zave and W. Schell. Salient features for an extensible specification language and its environment. *IEEE Transactions on Software Engineering*, 12:312–325, 1986.