Collaborative Software Engineering: An Annotated Bibliography

Carl Cook

Technial Report TR-02/04 (Revision 0.2), May 2005
(First published June 2004)

Software Engineering & Visualisation Group,
Department of Computer Science and Software Engineering,
University of Canterbury, Private Bag 4800,
Christchurch, New Zealand
`carl@cosc.canterbury.ac.nz`

Abstract

This work is intended to be a useful starting point for those researching the field of Collaborative Software Engineering (CSE). We list research articles related to current CSE tools, models and discussions, as well as relevant papers in related fields. This paper does not present a taxonomy of CSE research, rather it is simply a categorised, annotated listing of articles that are likely to be of use to CSE researchers during their preliminary investigations.

Please note:

- This bibliography is periodically updated to incorporate new or overlooked papers related to Collaborative Software Engineering. If you would like to nominate a paper for inclusion within this bibliography, please contact the author.

Contents

1 Introduction

Collaborative software engineering (CSE) is a fast growing area of Computer Science. While CSE has been researched in detail for decades, the discipline also draws on many other related areas of Computer Science, each with their own dedicated area of research activity. There are many exhaustive surveys on these related areas; this bibliography presents selected papers from these related areas that are pertinent to CSE, as well as listing papers core to CSE frameworks and tools. This bibliography presents a categorised listing of all such papers, accompanied by annotations that describe the content of the papers and their relevance to the field of CSE.

1.1 What is Collaborative Software Engineering?

Almost all modern software engineering methodologies involve several distinct groups of people working on many artifacts with different types of tools to produce multiple versions of software products. Software engineering is unavoidably collaborative, and CSE looks at ways of lending computer-based support for programmer and tool communication, management of artifacts, and coordination of tasks.

Research into CSE is progressing rapidly. Driving factors include the advent of industrial strength open source IDEs, a solidification of standards for distributed computing, significant advances in processing speeds and memory capacities, and more powerful, interoperable programming languages. Reliable high-speed networking reduces the boundaries between remote developers, programming frameworks such as .Net and J2EE provide inexpensive access to rich information related to any given software project, and new collaborative features can be incorporated into IDEs through open APIs.

Despite these recent technological advances, it is unlikely that one monolithic system will be developed that solves all the current challenges in supporting CSE. Instead, specific tools have been designed to support specific software engineering tasks or to provide specific collaborative enhancements over conventional tools, and this is likely to be the direction that CSE research follows for some time to come. Given the need to provide more collaborative support to software engineers than what is currently available through conventional development tools, the minimum list of considerations is:

Location Are the developers in a face-to-face and constantly co-located setting, or are they distributed throughout several departments or organisations?

Time Will developers typically work at the same time, different times, or a combination of both possibilities?

Task type What type of software engineering tasks are to be supported? Typical tasks likely to be supported in CASE tools include requirements gathering, analysis, system design, implementation, validation and verification.

Task complexity What is the level of complexity for the typical task? Are third party components used? Is legacy code employed? Is there extensive use of external libraries? What is the size of the typical task in terms of files, classes/modules and lines of code?

Task process Which development methodologies should be supported? Classical closed-source SDLC? eXtreme Programming? Open source development (where there are often no time pressures, minimal documentation and heavy moderation of changes)?

Artifact management How are artifacts controlled? Is a pessimistic locking scheme employed, an optimistic locking scheme, or is realtime artifact sharing possible? If so, what floor control policies are in place to manage collaboration?

Group size What number of people are likely to be supported? What is the maximum number of people likely to be working closely together on the same subset of artifacts within the project?

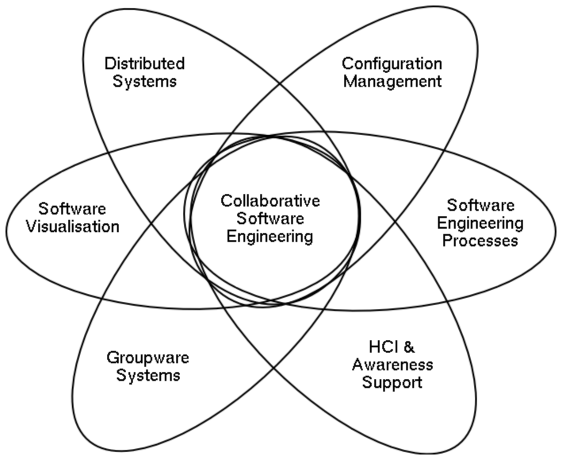
Group culture What are the ability levels of each developer? Should a mix of abilities be supported? Does a culture exist within the team where certain informal processes are likely to be followed, such as posting code update notifications to a mailing list?

Extensibility Should the tool support extensibility/customisation, different languages, and/or different views of the same artifact?

Implementation How should the tool monitor developer activity and analyse program changes?

Usability How should the tool deliver feedback to the user? Will such feedback be embraced by the user or seen as a hindrance?

After considering the above points, it is apparent that for a successful implementation of CSE tools, a knowledge of several related disciplines of Computer Science is required. Whilst an understanding of all aspects of software engineering is compulsory, other topics that must be addressed are human factors and usability, software configuration management, distributed systems, groupware and computer-supported collaborative work, and software visualation. As presented in the following figure, CSE forms an intersection of these six areas. After presenting the literature directly relevant to CSE, this bibliography will discuss each of these related areas in turn.



1.2 Purpose of this Document

This article is intended to be used as an introduction to CSE, CSE tools and related fields. It provides most of the main research articles related to CSE, and many other useful related papers can be found from these key articles. The main thrust of this article is to provide a useful teaching and learning resource for initial investigations into CSE.

This article does not present a major taxonomy of CSE research, nor does it attempt to classify CSE literature in a new or novel way. It is purely a bibliographic listing of important work classified into subcategories of CSE and related fields, with supporting annotations.

2 Existing CSE Tools

Software engineering encompasses a wide range of tasks ranging from requirements outlining to code debugging, and researchers have developed prototype CSE tools for each conceivable task. Subsequently, many developmental CSE tools and frameworks exist today, and support for real-time collaborative operation is becoming commonplace within this class of tools. At the time of writing, a handful of commercial CSE tools also exist for relatively simple software engineering tasks, and the research world is constantly publishing new and novel architectures, tools and perspectives on collaborative software engineering.

This bibliography classifies the literature on CSE tools and frameworks into five sections: management tools, design tools, development tools, inspection tools and discussion papers. For each section, an overview of the research area is presented, followed by a listing of key research articles.

2.1 Management Tools

The main objective of management tools is to support the organisation of software engineering workspaces. Such tools might allow the indexing and searching of source code and other artifacts but typically will not support direct editing of files, shared or otherwise. Management tools typically do not support design or modelling concepts within software engineering; rather their principle concern is the management of users and artifacts within a software engineering project. Management tools of these types are used to keep track of large artifact bases, groups of engineers, and interactions between artifacts and engineers.

Publications

[1] Pauline Wilcox, Craig Russell, Mike Smith, Alan Smith, Rob Pooley, Lachlan MacKinnon, Rick Dewar, and David Weiss. A CORBA-oriented approach to heterogeneous tool integration; OPHELIA. In *ESEC/FSE Workshop on Tool-Integration in System Development*, pages 1–5. Helsinki, Finland, 2003.

Annotation: This paper describes OPHELIA, a framework to integrate heterogeneous software engineering tools. Several papers have been published that describe the large research effort of OPHELIA, but this relatively short paper suitably covers the main aspects. The main thrust of the paper, in describing the framework, is to claim that many insights will be discovered by the ongoing work of integrating standalone tools into collaborative environments.

As described in this paper, the OPHELIA architecture creates global views of artifacts, and allows relationships to be defined between any project elements via a component called Traceplough. Additionally, event driven metrics are also supported, with updates broadcasted to all relevant listeners. In this paper, the OPHELIA architecture is illustrated by a brief example of integrating the ArgoUML tool, however very little is written about the benefits this provides.

[2] J. Altmann and R. Weinreich. An Environment for Cooperative Software Development: Realization and Implications. In *Proceedings of the Hawaii International Conference On System Sciences.* Kona, Hawaii, January 1997.

Annotation: This paper describes an architecture for collaborative software engineering where the emphasis is focused on minimising the number of constraints placed on the users of the system. As described within, two main components exist to facilitate distributed annotation of source code, bug tracking, and scheduling of tasks: a workspace manager and a cooperation manager. Many screenshots are presented to illustrate the system, and excellent background and architectural description sections are given, with compelling and useful references.

Unfortunately, no mention of system evaluations are made, either qualitative or quantitative. Whilst the architecture presented is very compressive system with rich user interfaces for communication and coordination, the question on whether or not it is genuinely usable remains unanswered. Regardless, this is a valuable article due to the comprehensive discussion of the architecture's construction.

[3] Stephen E. Dossick and Gail E. Kaiser. CHIME: A Metadata-Based Distributed Software Development Environment. In *Proceedings of the 7th European Engineering Conference held jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 464–475. Springer-Verlag, 1999. ISBN 3-540-66538-2.

Annotation: CHIME, as presented in this paper, is a metadata-based virtual environment for visualising and managing artefacts within a software engineering project. Artifacts are held within their original repositories, such as source code control systems, databases, and bug-tracking tools; CHIME is used as a navigation tool amongst these artifacts via generated virtual environments. Whilst leaving the artifacts in their originating repositories, CHIME manages the storage of annotations and chat sessions held between users. CHIME supports collaboration by displaying the avatar of users within the virtual world; once two or more avatars are inspecting artifacts within the same location, a collaborative chat session is enabled.

The CHIME architecture is interesting in its separation of spaces and views; the architecture is well defined and described within this document. This paper presents collaborative software engineering research from both the fields of development and visualisation; accordingly it is useful as a survey paper as well. A major drawback of this paper is the absence of visualisation examples, which is a central theme to this research project.

[4] Ian Gorton, Igor Hawryszkiewycz, Kenny Ragoonaden, Charles Chung, and Shijian Lu Guneet. Groupware Support Tools for Collaborative Software Engineering. In *Proceedings of the 30th Hawaii International Conference on System Sciences*, volume 2, pages 157–166. Maui, Hawaii, January 1997.

Annotation: This paper introduces an environment for the development and testing of software within highly distributed teams. The system, named GWSE, is based upon Lotus Notes because of the access to a flexible data repository. GWSE has facilities for configuration management and project management; these facilities are demonstrated in detail with the assistance of numerous screenshots and discussions.

This paper also introduces the concept of 'Gain Effect Exploitation', where the hours lost due to incomplete deliverables were calculated in a field trial. This is interesting as it illustrates some of the issued related to the isolation caused by traditional software engineering tools. The paper could benefit from having more references to support the stated requirements of the architecture, but it is still a very informative paper that describes one approach for the management of large distributed groups of developers.

2.2 Design Tools

Design tools within the field of CSE have some or all focus on supporting collaboration during the design of software engineering artifacts. Tools within this category typically support the design of relatively simple and low-detailed artifacts such as class diagrams and Class-Responsibility-Collaborators (CRC) diagrams. Other UML-based diagrams such as state transition diagrams and use-case diagrams appear too complex to be supported by CSE tools, although a few commercial implementations of such tools have been recently released, such as Poseidon for UML Enterprise Edition. CSE design tools also appear to focus more on workflow, communication and coordination rather than on artifact development and progression to implementation artifacts such as source code.

Publications

[1] Hurwitz Report. Collaborative UML Development. White Paper, CanyonBlue Incorporated, November 2001. URL `http://www.canyonblue.com/whitepapers.htm`.

Annotation: Konesa, initially known as Cittera, is presented in this white paper as the first UML tool to support real-time collaborative modeling. There are no published refereed papers that describe Konesa, but the tool's website is a useful source of reference information that may not have been covered within this white paper. As described in this paper, Konesa supports user communication via chat and shared whiteboard facilities. Additionally, a bulletin board service stores all user correspondence and changes to the model; these events can be searched as a form of model inspection. Konesa also supports change-tracking to provide graphical representations of changes over time per user. Finally, Konesa also has the concept of module ownership; owners can review changes before accepting or rejecting them.

This white paper also provides a quality discussion about the evolution of software engineering from stand-alone single user projects to real-time collaborative distributed development of large-scale systems. Unfortunately, the white paper only gives a brief outline of the features of Konesa—without any serious attention to providing proof of concepts or literature-based support. Additionally, there are no published papers with further information relating to Konesa as of yet. Finally, and whilst not mentioned in this white paper, the organisation that develops Konesa is part of the Eclipse consortium, and develops proprietary Konesa plugins for the Eclipse framework. For a description of Eclipse, please see Section 3.1.4 on page 20.

[2] Marko Boger, Thorsten Sturm, Erich Schildhauer, and Elizabeth Graham. *Poseidon for UML User Guide.* Gentleware AG, 2002. URL `http://www.gentleware.com/support/documentation.php4`.

Annotation: Poseidon is a commercial version of the ArgoUML modeling tool. The research related to ArgoUML is well published—please see Section 3.1.4 on page 20. There are no formally refereed papers on Poseidon, however many have been published that cite the product. Apart from being one of the most popular UML modeling tools in current practice, the enterprise edition of Poseidon has been designed for use in highly collaborative environments.

This paper represents the user guide for Poseidon, providing a solid overview of the system's architecture and plugin infrastructure. The Poseidon website provides further information relating specifically to the enterprise edition's collaborative features, including Jabber-based instant messaging and coarse-grain version control.

[3] Nicholas Graham, Hugh Stewart, Authur Ryman, Reza Kopaee, and Rittu Rasouli. A World-Wide-Web Architecture for Collaborative Software Design. In *Software Technol-*

ogy and Engineering Practice, pages 22–32. Pittsburgh, Pennsylvania, August 1999.

Annotation: Rosetta is a light-weight tool for web-based collaborative development of UML design diagrams. This paper introduces the Rosetta architecture, describes the types of documents that can be developed using Rosetta, and then explains how the architecture supports collaborative development. The paper also gives details on evaluation of Rosetta, and discusses some of the experiences in using the Internet as the platform for all Rosetta-based applications.

This is a well written paper that describes an architecture currently being used within a commercial environment. This paper gives insights related to the design and deployment of a simple yet useful collaborative and distributed software engineering tool. This paper also provides many useful references in the introduction and background sections relating to the need for such tools, and the problems pertaining to distributed collaboration.

[4] Neville Churcher and Carl Cerecke. GroupCRC: Exploring CSCW Support for Software Engineering. In *Proceedings of the 4th Australasian Conference on Computer-Human Interaction.* IEEE Computer Society Press, Hamilton, New Zealand, November 1996.

Annotation: This paper describes GroupCRC, a multiuser implementation of a tool to facilitate the Class, Responsibility, Collaborator (CRC) object oriented analysis technique. The purpose of this paper is to demonstrate how CSCW technologies support and enhance the ability of groups to accomplish shared goals. For the unfamiliar, the paper also provides a useful description of the CRC process.

The paper describes the roles of CASE tools, the roles of CSCW applications, and then illustrates why the two types of systems have different priorities. The main research effort of this paper is to demonstrate how enhanced collaboration and maintainance of document structures can be achieved simultaneously.

[5] Josef Altmann and Gustav Pomberger. Cooperative Software Development: Concepts, Model and Tools. In *Proceedings of the 30th Conference on Technology of Object-Oriented Languages and Systems*, pages 194–209. California, USA, August 1999.

Annotation: This paper introduces a model for the organisational support of collaborative software engineering. A solid discussion of the role of collaboration is provided, and an interesting model to map the domain of software engineering is proposed, based on collaboration, coordination, and communication. The model supports two main views: processes and products; the paper then presents a rather comprehensive suit of tools in support of the model.

This paper is six years old now, and no follow-up systems or evaluations have been published to our knowledge. Rather strong claims for improved productivity and quality are also made, but no specific goals defined. There are also no specific references to planned user evaluations—yet such evaluations are surely essential due to the expansive user interfaces and system interactions.

2.3 Development tools

The task of designing and implementing successful Cse development tools is difficult; not only is a knowledge of collaborative software engineering required, but the implementors must also be able to program concurrent distributed systems and be competent of designing complex yet usable user interfaces. Subsequently, there are many prototype Cse development tools but they are typically designed for only a single task or programming language. Such development tools are also quite trivial when compared to commercial-scale IDEs, therefore, Cse development tools still have a significant advance to make before they will have a serious impact on mainstream software engineering. Similarly, there are very few frameworks that allow the rapid development and extension of Cse tools and the migration of single user tools to being collaborative. Instead, the majority of Cse research activity appears to be based around the development of 'throw-away' prototype tools.

There are many tools available which support real-time modelling, design and management of software. Development tools, however, are typically based upon conventional software engineering tools and technologies: as developers check in or check out source code from a central repository, users are alerted to possible conflicts. Only a few real-time editing and diagramming tools exist where the conventional model of copy-modify-merge is replaced with fully synchronous file sharing.

Publications

[1] Uwe M. Borghoff and Gunnar Teege. Application of Collaborative Editing to Software-Engineering. In *SIGSOFT Software Engineering Notes*, volume 18, pages 56–64. ACM, July 1993.

Annotation: This paper presents IRIS, a collaborative editor for software engineering artifacts. Written ten years ago, this is another paper where the importance is for its historical contribution to the field of collaborative software engineering, rather than the relevance of IRIS today. IRIS, as described within the paper, is a collaborative system for the editing and management of well structured documents, specifically source code. Collaborative features include dynamic voting and notification of changes to all current users, and IRIS uses a replicated data architecture to allow concurrent editing of artifacts.

This paper provides a good survey of similar systems at the time of writing, and presents a very detailed and informative description of the IRIS architecture. Unfortunately, the paper lacks a discussion of an evaluation of the system, justifications of design choices, and illustratory examples. However, since IRIS was one of the first collaborative tools for software engineering, this paper is still essential reading.

[2] Prasun Dewan and John Riedl. Toward Computer-Supported Concurrent Software Engineering. *IEEE Computer*, 26(1):17–27, 1993.

Annotation: This is an early paper within the field of collaborative software engineering, but it is still very relevant and informative today. The references throughout the paper are of well respected publications, therefore the paper in itself is a good survey. The paper was well cited the year following publication, but perhaps surprisingly, it has not received much attention since then.

FLECSE is an architecture comprised of a collection of multiuser tools: editors, inspection tools, code repositories, and debuggers. The architecture also provides multimedia support. As well as describing the FLECSE architecture, a good definition of collaborative software engineering is provided, which includes the key concepts of tools, concepts, life cycles, integration, and sharing. An observation admitted by the authors is that this paper lacks arguments to support why the proposed tools are genuinely useful. This paper is still very useful however, and as the authors state, it should give readers an idea of the benefits that collaborative technology can bring to software engineering.

[3] Carl Cook and Neville Churcher. An Extensible Framework for Collaborative Software Engineering. In Deeber Azada, editor, *Proceedings of the Tenth Asia-Pacific Software Engineering Conference*, pages 290–299. IEEE Computer Society, Chiang Mai, Thailand, December 2003.

Annotation: This is a self-citation to the paper that describes the CAISE collaborative software architecture. As described in this paper, CAISE is an extensible architecture that supports the integration of software engineering tools. The storage and synchronous sharing of software artifacts is managed by CAISE, and changes to artifacts by client tools update the underlying model of the software, as maintained by the CAISE server. Any artifact and programming language can be supported by CAISE; the more formally defined the grammar of the language, the finer the level of modeling.

This paper describes the motivation for such a framework, highlights related work, and then illustrates the key concepts via an example set of collaborative tools.

[4] Anita Sarma and Andr van der Hoek. Palantr: Coordinating Distributed Workspaces. In *26th Annual International Computer Software and Applications Conference*. IEEE, Oxford, England, August 2002.

Annotation: Palantir is a framework for the realtime notification of software engineering events within a distributed group of developers. It is based upon a SCM system, and monitors check-ins to produce change impact and severity analysis reports. Such reports are delivered to end users, where they are visualised in real time.

After providing an example scenario of usage, this paper gives implementation details of Palantir, followed by a brief listing of related work. In the conclusion the paper asserts that more development of Palantir is required, including case studies to investigate the effectiveness of the framework.

[5] Carl Cook, Neville Churcher, and Warwick Irwin. Towards Synchronous Collaborative Software Engineering. In *Proceedings of the Eleventh Asia-Pacific Software Engineering Conference*, pages 230–239. IEEE Computer Society, Busan, Korea, December 2004.

Annotation: This paper is an update to the original description of the CAISE CSE framework. The paper asserts a definition of CSE, and then goes on to discuss the collaborative spectrum of tools and processes. Prior to detailing the latest version of the collaborative architecture, the paper discusses the factors that influence collaboration within software development, and the types of feedback that users require when developing software. Another key point of this paper is the identification of the real-time logical area of critical code for each specific development task—a set of source files which is typically larger than what current single user tools support.

After describing the latest version of CAISE, the paper presents new tools developed for the architecture. The paper demonstrates how these tools provide the types of feedback required by collaborative development teams, and shows how further tools can be developed. The paper concludes by detailing future work, which includes the identification of patterns of collaboration, and analysis of realtime user activity.

[6] Till Schummer. Lost and Found in Software Space. In *34th Annual Hawaii International Conference on System Sciences.* Maui, Hawaii, January 2001.

Annotation: This paper presents TUKAN, a system for the collaborative editing of SmallTalk software projects. TUKAN is built upon the Orwell configuration management section (see Section 3.3 on page 29), and provides a collaborative class browser and editor for each programmer to use as they inspect and develop their code base.

This paper provides an insightful usage scenario to describe the key features of TUKAN, with a good balance of screenshots, discussions, and formal definitions. Of particular interest to CSE researchers are the spatial models, where dynamically updated representations for presence awareness and user locations are presented. Apart from describing the TUKAN architecture, this paper gives an interesting discusses the role of user awareness and program comprehension in order to justify TUKAN's design. Whilst limited to a single interface for the SmallTalk language, the discussion of the TUKAN architecture within this paper is comprehensive.

[7] Li-Te Cheng, Cleidson R. B. de Souza, Susanne Hupfer, John Patterson, and Steven Ross. Building Collaboration into IDEs. In *ACM Queue*, volume 1, pages 40–50. ACM, January 2004.

Annotation: The Jazz architecture is a Java-specific collaborative development extension to the Eclipse integrated platform—see Section 3.1.4 on page 20 for information on Eclipse. After a detailed discussion of need for and challenges of integrating collaboration within an IDE, this paper provide several simple examples and proof of concepts, and also gives a detailed description of Jazz's features. One point to note is that Jazz is designed for use in office environments to facilitate a higher level of collaboration and communication between developers; in this light, Jazz is not necessarily suitable for highly distributed settings.

There are several earlier papers that introducing Jazz in more detail, but this paper is a polished summary of the researchers' work to date. The paper provides a useful 'Lessons Learned' section, and also details some recent publications and research projects.

[8] W. R. Bischofberger, C. F. Kleinferchner, and K. U. Matzel. Evolving a Programming

Environment Into a Cooperative Software Engineering Environment. In *Proceedings of the International Conference on Software Engineering*, pages 95–10. McGraw-Hill, New Delhi, India, February 1995.

Annotation: Sniff is a project developed for industrial collaborative software engineering. From the prototype came a commercial version called Sniff+. This paper discusses the mechanisms for collaboration within Sniff+, such as its parser-based source code analysis, class visualisation system, and communication facilities. Additionally, the paper discusses a research project that is a continuation of the original sniff research named BeyondSniff, which integrates additional tools into the Sniff environment.

Whilst this paper is low on references and arguments to support the design of the architecture, the paper is valuable for its description of a novel, commercial collaborative system that is still popular today. The paper provides many screenshots of the Sniff+ tools, and presents a good discussion on the merits of both pessimistic and optimistic artifact control.

[9] C. M. Werner, M.S. Mangan, L. Murta, R. Pinheiro, M. Matoso, R. Braga, and M. R. S Borges. OdysseyShare: an Environment for Collaborative Component-Based Development. In *The IEEE International Conference on Information Reuse and Integration*, pages 61–68. Las Vegas, USA, October 2003.

Annotation: OdysseyShare is a set of tools to support the collaborative development of component-based systems. For a given domain, the OdysseyShare environment provides support for the development, editing, and reverse-engineering of software artifacts—including source files, binary components, and diagrams. The environment is supported by a workflow engine, and a distributed component library.

In this paper, a collaborative class diagrammer is presented, with a focus on two real-time collaborative widgets: a 'radar view' and telepointers. The general goal is to provide realtime support for a team of software engineers by increasing the level of intra-team awareness and communication. Little mention is made to the type of tasks that OdysseyShare supports, and it is also unclear what sized terms the system is designed for, the types of languages supported, and the general capabilities of the collaborative tools provided. Regardless, the workflow-based approach is interesting, and there are many useful references within the paper.

[10] Scott Lewis. Eclipse Communication Framework. Internet Homepage, April 2005. URL `http://www.eclipse.org/ecf/goals.html`.

Annotation: The Eclipse Communication Framework (ECF) is a new research initiative that was demonstrated at EcipseCon 2005. Once released, the ECF will provide a platform to rapidly develop eclipse based collaborative tools, including software engineering tools based upon the Eclipse Modelling Framework (EMF). While no official papers have been published yet, the above URL provides the first glimpse of what the ECF promises to be.

There are three core components to the ECF: and API for sharing the eclipse project model, APIs for supporting communication and collaboration, and the Rich Client Platform (RCP) for building eclipse-based tools. The article given here provides links to the first prototype tools developed under the ECF, such as a shared text editor and shared graphing program. Whilst relatively unheard of at time of writing, ECF promises a great deal in terms of increasing the degree of collaboration within software engineering, and it is gaining attention and support from within and outside the eclipse community.

2.4 Inspection Tools

Inspection tools within the field of CSE typically support one of two functions: allowing users to collaboratively inspect code and designs as a group, or allowing single users to inspect code and designs that have been collaboratively developed. Inspection tools differ from management tools in that their key role is the inspection and investigation of software engineering artifacts for the benefit of future development and refinement, as opposed to management tools that are more concerned with group coordination and artifact control.

Publications

[1] Vahid Mashayekhi, Chris Feuller, and John Riedl. CAIS: Collaborative Asynchronous Inspection of Software. *Proceedings of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering*, 19(5): 21–34, December 1994.

Annotation: Whilst ten years old now, this paper describes an interesting tool for the collaborative inspection of software systems named CAIS, which was built using Lotus Notes and Suite. The system does not support shared browsing of software artifacts; rather it facilitates asynchronous discussions between users as they inspect software systems. Although the CAIS architecture itself is not groundbreaking in terms of support for collaborative software engineering, the evaluation as presented in this paper is very informative in terms of user behavior when inspecting code.

The paper is well referenced throughout, and has thorough sections on related work and lessons learnt. The evaluation concluded that meetings take longer in CAIS than in face to face, and that many meetings require synchronous interaction—something that CAIS can not provide. More faults, however, were reported in the software when using CAIS than without. Another interesting point raised in this paper was that users preferred less detailed and structured meetings if given the choice.

[2] Michael Stein, John Riedl, Soren J. Harner, and Vahid Mashayekhi. A Case Study of Distributed, Asynchronous Software Inspection. In *Journal of Collaborative Software Engineering*, pages 107–117. ACM, Boston, MA, USA, May 1997.

Annotation: This paper, which presents the collaborative software inspection tool named AISA, has quality references throughout. The AISA system, as described within, is a web-based advance on the previous systems of CSI and CAIS. AISA allows voting between users, asynchronous meetings, and structured fault tracking. AISA also allows the graphical display of the domain being inspected as an interface for inspection. It should be noted however that AISA only supports asynchronous meetings and feedback; real-time collaboration is not possible with this system.

In the background sections, this paper includes solid theoretical underpinnings for the proposed system. In the evaluation section, results were reported from the measurement of fault corrections resulting from inspections in an industrial setting. The paper concludes that people enjoy using the system, but the system delay is noticeable over the intercontinental link. As a positive aspect, a proportion of users continued using the tool after the trial, even with the seven hour time difference between the two groups of users. As a final point, the authors emphasise that the tool is not intended to replace face to face meetings, but to support existing synchronous activities.

[3] Robert Lougher and Tom Rodden. Supporting Long-term Collaboration in Software Maintainance. In Simon Kaplan, editor, *Proceedings of the Conference on Organizational Computing Systems*, pages 228–238. ACM, Milpitas, California, USA, November 1993.

Annotation: Maintainance of software systems is progressively becoming a highly collaborative task—especially as system sizes increase. This paper presents an architecture that supports long term collaboration between maintainers by capturing annotations of the design rationale. Using this architecture, maintainers of the software system can add structured comments to units of code, draw contextual diagrams, and search the change-log for comments as the system evolves.

The paper provides an excellent discussion of "documentation by annotation". There are also many good formal discussions in this paper to justify the design of the system, including a section on the understanding of collaboration in software maintainance. The paper also gives numerous examples of the system illustrated by screenshots, and there are additional sample scenarios to show usage of the architecture. An interesting point raised within the paper is that this system is different in its ability to support documentation in a parallel thread to the main artifacts—this is essential due to the relatively unstructured task of software maintainance.

[4] Carl Cook and Neville Churcher. Modelling and Measuring Collaborative Software Engineering. In Vladimir Estivill-Castro, editor, *Proceedings of ACSC2005: Twenty-Eighth Australasian Computer Science Conference*, volume 38 of *Conferences in Research and Practice in Information Technology*, pages 267–277. ACS, Newcastle, Australia, January 2005. 25% acceptance rate.

Annotation: This paper discusses the modelling, analysis and visualisation of fine-grained activity data gathered from collaborative software engineering tools as they work on a shared project. Through the use of tree maps, several distinct patterns of collaboration were apparent from a short period of activity between four co-located programmers. Through analysis of the event data such as file modifications, chat messages, and model change events, it was possible to identify times when users worked on concentrated code fragments, navigated around several files within a short period of time, experienced difficulties in compiling, and spent large amounts of time communicating with others via the messaging utility. By having access to fine-grained information related to user activity, it is also possible to produce simple visualisations of the intersections of artifacts and the users that modified them. It is intended that such visualisations can provide a greater understanding of the fine-grained actions of developers.

As a separate thread in this paper, the concept of heuristic evaluations for CSE tools is introduced. It is intended that these heuristics are used to constantly refine CSE tools so that they adhere to principles of good design and fundamental requirements of CSE.

[5] Jon Froehlich and Paul Dourish. Unifying Artifacts and Activities in a Visual Tool for Distributed Software Development Teams. In *6th International Conference on Software Engineering (ICSE'04)*, pages 387–396. IEEE, Edinburgh, Scotland, United Kingdom, May 2004.

Annotation: This article presents Augur, a comprehensive visualisation tool for inspecting and exploring software development activity. Augur consists of a data gathering architecture based on the semantic analysis of source code repositories and a set of visualation tools. These tools allow developers to monitoring their activity and explore the distribution of their combined activities over time and artifacts.

This article gives four interesting case studies of developers that used Augur as a supplement to their regular software engineering tools. The feedback suggests that Augur is both valuable and meaningful as a feedback tool. After presenting the design and implementation details of augur, the paper provides a sound section on related work. The paper concludes by asserting that developers were assisted in their work through the combined views of artifacts and user activity.

2.5 Discussion Papers

These papers present general discussions and surveys on aspects of CSE, rather than presenting a tool that performs a specific task or fulfills a certain role. Within the field of CSE research there are many discussion papers related to conceptual frameworks rather than actual implementations, and these papers provide useful insights for any researcher considering the design or development of new CSE tools. This section also presents papers related to the social protocols of collaboration, including open source development practices.

Publications

[1] Uwe Busbach. Activity Coordination in Decentralized Working Environments. In Alan Dix and Russell Beale, editors, *Remote Cooperation: CSCW Issues for Mobile and Tele-workers*, chapter 8. Springer/BCS, September 1996.

Annotation: This paper discusses the perils and pitfalls of asynchronous collaboration within highly distributed environments; topics explored include activity coordination and conflict resolution. An activity coordination tool named Task Manager is also presented in this paper, which is used to illustrate the assertions made within. One of the most important points raised in this paper is the trade-off between uninhibited collaboration and consistency of data objects.

[2] Stan Jarzabek and Riri Huang. The Case for User-Centered CASE Tools. *Communications of the ACM*, 41(8):93–99, 1998.

Annotation: This paper is another good discussion of what researchers should be aware of when selecting or designing CASE tools, and analyses the recent state-of-the-art. Similar to other CASE tool discussion papers, it does not refer specifically to multi-user tools, but the discussion is certainly applicable for researchers who are developing shared software engineering tools. This paper is a discussion document on why tools should become more user-centric; no attempt is made to study users or evaluate tools.

The paper raises the interesting point that the market for CASE tools is increasing, but such tools are still not widely utilised. For CASE tools to be more widely used, the paper argues that such tools need to move away from being method and process based, and begin supporting programmers dynamically, based on what they are developing, and the level of programmer expertise. The paper continues by suggesting that we need to first take a look at what the behaviour of programmers is, and then build tools around the models of system development. This is a light paper to read, but worthwhile for the interesting discussion on design considerations for CASE tools.

[3] J. Iivari. Why are CASE Tools Not Used? In *Communications of the ACM*, volume 39, pages 94–103. ACM Press, October 1996.

Annotation: This paper provides a comprehensive study and discussion of problems with CASE tools, centered upon a statistical analysis of a survey of CASE tool users. The data was collected in 1993; whilst the tools under investigation are no longer in use today, the lessons learnt are still relevant reading. A key point of this paper is that much research into CASE tool usage is based on description rather than identifying theoretical underpinnings or explaining of factors; therefore this paper performed a major survey to illustrate and validate the theoretical arguments within.

The paper gives a conceptual model and hypothesis about drivers for CASE adoption, and then surveyed 35 organisations. Whilst this is a worthy research effort, only 1 of the 105 individual respondents was employed primarily as a programmer. This suggests that the results are relevant for analysis and design tasks rather than programming duties. The paper places importance on its statistical results to illustrate the factors related to CASE tool adoption, but it is worth considering that it may be dangerous to infer causality for software engineering in the large based solely on this survey. Regardless, the suggestions based from observation within this paper, such as stronger support from management for CASE tools, are well worth noting. Additionally, many references to papers describing the adoption of CASE tools, from viewpoints of both theory and practice.

[4] Iris Vessey and Ajay Paul Sravanapudi. CASE Tools as Collaborative Support Technologies. In *Communications of the ACM*, volume 38, pages 83–95. ACM Press, January 1995.

Annotation: CASE tools are designed to aid software developers, and this paper points out that most CASE tools are not collaborative. Accordingly, the potential benefits of collaboration-aware CASE tools are investigated in this paper, and in doing so represents one of the first exploratory investigations into collaborative CASE tools.

This paper begins with an in-depth discussion about the difference between coordination and collaboration, and the sometimes opposing objectives they support. As an evaluation, this paper examined four CASE tools for systems analysis, and ranked each of them in terms of control, information sharing, monitoring, and cooperation. Whilst the study appears to be minimal in evaluating 'cooperation', the analysis of 'coordination' features is useful. The paper concludes with a comprehensive discussion on the implications of CASE tool design for developers and researchers.

[5] G. Booch and A. W. Brown. Collaborative Development Environments. In Marvin Zelkowitz, editor, *Advances In Computers*, volume 59. Academic Press, August 2003.

Annotation: This article represents an excellent discussion on the current state of research into collaborative development environments (CDEs). Key to the article is a

vivid metaphor—points of friction. The article discusses the points of friction within current collaborative development, and suggests ways to create a frictionless surface.

The article starts with an excellent opening discussion on the role of developer, and the dynamics of software. Useful discussions then follow related to CDE features, with categorization into coordination, collaboration, and community building. After a discussion on the evolution of CDEs, and the presentation of a conceptual model of CDEs, the article finishes with a five-step guide to prepare organisations for CDE adoption.

Included in this article is an excellent and up-to-date survey of CDE systems, with useful screenshots from various applications. A useful taxonomy of CDEs is then given, categorising CDEs into the non-software domain, asset management, information services, infrastructure, community, and software development. Apart from being a very informative and insightful read, the article also gives many useful references.

[6] Yunwen Ye and Kouichi Kishida. Toward an Understanding of the Motivation of Open Source Software Developers. In *Proceedings of the International Conference on Software Engineering (ICSE2003)*. Portland, Oregon, May 3-10 2003.

Annotation: This paper leads an interesting discussion as to what might motivate people to contribute to open source software projects. A case study is presented based upon the GIMP project, one of the most popular open source projects in existence. Using learning theory, this paper suggests that Legitimate Peripheral Participation (LPP) might be one of the driving forces behind contributions to open source projects, where users feel legitimised for their efforts and further their own software engineering knowledge.

This paper provides a good discussion of the nature of open source development: the economics, the social structure and the dynamics of change. It presents the complicated field of open source software development from a new analytical viewpoint, providing practical guidelines for the creation, management and development of OS projects.

[7] Anita Sarma. A Survey of Collaborative Tools in Software Development. Technical Report UCI-ISR-05-03, Donald Bren School of Information and Computer Sciences, University of California, Irvine, March 2005.

Annotation: This document presents an excellent taxonomy of CSE tools and related papers. The related work section provides several conceptual models that categorise CSE in various ways, such as space and time, model domains, and software engineering processes. The survey then introduces a well-designed classification framework, which pairs level of coordination support against three elements of collaboration: communication, artifact management and task management. The remainder of the survey then categorises and discusses a massive 228 CSE papers according to classification framework, and concludes with a discussion of observations made during the classification process.

[8] Carl Gutwin, Reagan Penner, and Kevin Schneider. Group awareness in distributed software development. In *CSCW '04: Proceedings of the 2004 ACM Conference on Computer Supported Cooperative Work*, pages 72–81. ACM Press, New York, NY, USA, 2004. ISBN 1-58113-810-5.

Annotation: This paper leads an interesting discussion as to how distributed open source developers maintain awareness of other users. The paper presents the findings of a process that involved interviewing developers, reading communications, and examining artifacts and repositories for three large and successful open source projects. The paper explains that developers do need to maintain an awareness in two senses: a general project awareness and a closer awareness of others that are working on highly related code.

The investigation shows that developers build up a mental model of the project and its associated developers, and this model is maintained through an ethic of regularly reading and posting to the development mailing lists and chat servers. An interesting characteristic was revealed upon inspection of the projects' artifacts—large numbers of strongly partitioned files exist, which suggests that the projects are in parts maintained by single developers only. Other key findings to come out of this study is that many developers feel that they require little additional tool support due to the success of their manual coordination efforts. Accordingly, given the option of visualisation tools and proactive tools such as CVS edit, many programmers responded negatively. It appears that with well run development groups that have clearly established work ethics, such as those presented in this paper, problems in duplicated and conflicting efforts are rare and easily resolved. As stated in the paper, the only time when problems do occur and have a significant impact is when the open source teams are placed under time pressures.

[9] Frederick P Brooks Jr. *The Mythical Man Month: Essays on Software Engineering.* Addison-Wesley, second edition, 1995. ISBN 0-201-83595.

Annotation: This famous work is essential reading for any computer science researcher. For the CSE researcher in particular, it provides many great discussions about how projects are constructed in teams, and makes many observations related to team dynamics and leadership within software projects. For those unaccustomed to large-scale industrial software development, this book gives great insight into the perils and pitfalls of collaborative software development.

3 Fields Related to CSE Research

This section does not represent an exhaustive collection of all papers related to the associative fields of collaborative software engineering. Rather, it represents all the papers that we found useful as background research into the construction of an architecture and associated tools for collaborative real-time software engineering. As comprehensive surveys already exist for each discipline in its own right, the following sections are focused primarily on relevance to supporting collaborative software engineering.

The following sections discuss the areas of Computer Science identified within the Venn diagram of figure 1.1. These area are: Human Factors, Source Code Management Systems, Distributed Systems, Software Visualisation and Metrics, Groupware and Computer Supported Collaborative Work, and Software Engineering Processes. We do not claim that CSE research is at the core of each related discipline, but we do assert that each of the above areas are key in supporting and enabling successful CSE tools and frameworks.

3.1 Software Engineering Processes

Software engineering is a very practical area within the discipline of Computer Science. As such, the theories related to software engineering are often produced empirically by observation of practicing software engineers and the induction of facts, rather than by deduction, proofs and foresights conceived in research laboratories. This is not to say that research papers related to software engineering are of less significance than papers in other fields of Computer Science; we merely observe that the software engineering industry is driven by demand, and the 'best ideas' and practices are often developed in the field and then analysed and documented, as opposed to all ideas being conceived within academia and progressing to the practicing industry.

Due to the large range of tasks within software engineering and the considerable amount of variance in methodologies, team dynamics and duties to be performed, it is difficult to perfectly map what happens within software engineering projects to a set of rules and processes. Consequently, many of the highly cited articles related to software engineering are chapters within books that document entire processes, not scientific conference and journal papers.

This section presents key articles related to three core methodologies of software engineering: classical methodologies, object oriented methodologies and agile methods. The final listing of articles is for software engineering tool support, providing details on some of the most popular tools used by software engineers today. It is intended that this section provides the reader with an understanding of contemporary software engineering practices and supporting tools.

3.1.1 Classical Methodologies

After the 'software crisis' of the 1980s, the practice of software engineering matured considerably with the birth of formal software engineering processes and methodologies. Structured, controlled ways of developing software were introduced, such as top-down and bottom-up development, the waterfall, spiral and chaos models, and data-driven development. Common to all of these methodologies were requirements analysis, design, implementation, testing and maintenance. The books given in this section provide an overview of these methodologies, working examples and case studies.

Publications

[1] Ian Summerville. *Software Engineering.* Addison Wesley, Reading, MA, 6th edition, 2000. ISBN 020139815X.

Annotation: In terms of the Software Development Lifecycle, this is a classic textbook. Whilst several chapters are dedicated to object oriented design, the content focuses mainly on the SDLC methodology of component-based software engineering—namely specific and isolated stages of development. This textbook is very comprehensive, covering everything from system requirements gathering to formal specification with Z schemas. In particular to collaborative software engineering, this text integrates content on CASE tools throughout the book, and also includes specific chapters on group working, configuration management, and large-scale project management. This is a classic software engineering textbook, but it also very valuable for those wanting more of an insight into group development.

[2] Steven C. McConnell. *Code Complete: A Practical Handbook of Software Construction.* Microsoft Press, Redmond, Washington, 2nd edition, June 2004.

Annotation: Ten years after the first edition, this popular 'practitioner's guide' to software engineering has been revised and released. This book is not heavy on the academic aspects of software engineering; rather it offers great advice on the implementation and design issues that programmers face on a daily basis. This book is respected for its advice on everything from testing to using bug databases to coding conventions.

Whilst most software engineering researchers are not necessarily interested in every chapter of this 900-page book, it is an excellent guide for those wish to gain more of an insight into the 'real-world' complications of software development. Code Complete noticeably stays away from most software engineering methodologies, but does have a chapter specifically dedicated to XP-styled refactoring. As an insight into real-world and large-scale development, and as an up-to-date discourse of the systems and processes that are being used within the software engineering industry today, this book is well worth examining.

3.1.2 Object Oriented Methodologies

Object oriented design and programming is a methodology that enforces modular programming and data hiding through encapsulation. This methodology specifically encourages software reuse via inheritance and polymorphism, where general-purpose components can be extended by 'differences only' to provide a suitable solution rather than 'reinventing the wheel'.

There have been two key peaks of popularity for object oriented development. The first was in the 1970s when the SmallTalk language was introduced. Subsequent to this, object orientation became popular again in the 1990s through the advent of C++ and then Java. Object oriented design and implementation is now the de facto standard within the software engineering industry and is supported by recently introduced languages such as C#, and Python.

This listing again provides references to definitive books on object oriented methodologies which are considered by many to be essential reading for all software engineers. This listing provides references to overviews of object orientated programming and development, as well as best practices for practical system development.

Publications

[1] Bernd Bruegge and Allen Dutoit. *Object-Oriented Software Engineering.* Prentice-Hall, New Jersey, 2000. ISBN 0-13-489725-0.

Annotation: As a good example of a textbook for object oriented software engineering, this book by Bruegge and Dutoit is worth examining. All examples within the text are based upon the UML notation, and the extensive use of design patterns are made. Apart from being a popular text for object oriented software engineering, this book also makes specific references to distributed and group development.

[2] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional Computing Series. Addison Wesley, 1995. ISBN 0201633612.

Annotation: This books constitutes the seminal publication on design patterns; it is highly cited in all modern contexts of software engineering. Design Patterns—known informally as the Gang-of-Four book—provides a concise taxonomy of design patterns for software construction, categorised into creational, structural, and behavioral patterns.

The Gang-of-Four book is an excellent reference guide for common patterns, and also has well written introductory chapters demonstrating how design patterns solve design problems. An extensive case study is used for illustratory purposes. Examples for every pattern are provided in either C++, SmallTalk, or both.

This book sits next to nearly every software architect, and is compulsory reading for every CS student. Every pattern is consistently documented in terms of intent, purpose, motivation, known uses, implementation notes, and related patterns. Whilst again this book is not specifically written for collaborative software engineering, it is very relevant in terms of implementation considerations.

[3] John Vlissides. *Pattern Hatching: Design Patterns Applied.* Addison Wesley, Reading, MA, 1998. ISBN 0-201-43293-5.

Annotation: To supplement the 'Gang of Four' book [2], this short paperback article has been produced. It has many insightful comments and code examples on the implications of applying design patterns. No new patterns are introduced; rather it is just a best-practices guide for programmers and system architects. This book does not make specific reference to addressing aspects of collaborative software engineering, but as a general guide for modern-day programming, it is essential reading. It is also useful for wish to gain insight into some 'real-world' issues and solutions when designing and implementing modern computer software.

[4] Marin Fowler. *UML Distilled: A Brief Guide To The Standard Object Modeling Language.* Object Technology Series. Addison Wesley, Reading, MA, 3rd edition, 2004.

Annotation: This is the latest edition of the popular UML distilled title, and represents a significant rewrite of the previous edition. UML Distilled covers UML version 2.0—the latest OMG UML standard.

This book is more than just a reference to UML–it is also a practical guide for usage. There are plenty of examples and explanations, along with a few code snippets for illustratory purposes. Additionally, this book mentions how UML integrates with specific development processes such as waterfall and RUP, and a section on integrating UML with patterns. The book also has detailed discussions on when and where to use UML.

UML Distilled is a modern-day classic software engineering book—again not specifically for the collaborative software engineering researcher, but it is essential reading for any practitioner wishing to design large-scale contemporary applications.

3.1.3 Agile Methods

Agile methods is a collective term given to a relatively new software engineering approach. Agile methods encourages software development that deals effectively with issues such as changing requirements and fluctuations in resourcing levels. It encourages practices that emphasise rotation of duties and stakeholder involvement. Agile methods also promote the iterative development of small yet functional pieces of the total project. This is in considerable contrast with the waterfall process described in section 3.1.1.

This listing provides details on agile methods, including pair programming and the eXtreme Programming process. Pair programming is particularly important to CSE research due to its inherently collaborative nature. All other agile method approaches are also important to CSE, due to incorporating teamwork as a cornerstone of the methodology.

Publications

[1] Alistair Cockburn. *Agile Software Development*. Addison-Wesley, 1st edition, December 2001.

Annotation: The terms 'Agile Methods' and 'Agile Development' are becoming increasingly common within the software engineering literature. As the agile methods banner is relatively new, little literature exists within the research community. This book, however, provides a comprehensive overview of the agile development methodology, and is well cited within both the industry and research communities.

Surprisingly enough, the book is tends to be focused on group communication rather than methodologies. The book does not necessarily attempt to 'sell' agile methods as the one true software engineering religion; rather it provides a detailed and somewhat exhaustive description of what constitutes agile methods, what they can be applied to, and where they originated from. The book also provides many excellent illustrations of peer programming environments, challenging scenarios, and the like.

This book is an excellent resource for the collaborative software engineering researcher, in particular the sections relating to communication within and between teams, co-location vs distributed environments, artifact types and life-cycles, and forms of communication.

[2] Laurie Williams, Robert R. Kessler, Ward Cunningham, and Ron Jeffries. Strengthening the Case for Pair Programming. In *IEEE Software*, volume 17, pages 19–25. July 2000.

Annotation: Pair programming, an integral component of XP, is becoming common practice within development teams of all sizes. This paper presents some empirical evidence that the practice of pair programming has its advantages.

The paper starts with a comprehensive high-level description of pair programming, and then describes a new experiment to assert the claims of that peer programming is not significantly slower than conventional approaches, reduces bugs, and subjects enjoy it more. The study involved 41 university students, and concluded that the pairs worked 40-50 percent faster than individuals, with slightly higher test cases passed.

Unfortunately, the paper is weak on the documentation of the experimental design, however the anecdotal comments made by students, and the subsequent discussion by the authors make for useful reading. Regardless of the experiment writeup, this paper provides a solid description of XP, including the history of pair programming. The anecdotal comments, such as programmers enjoyed peer programming more, and felt more confident about their coding due to the presence of an observer, make this paper an important one in the story of collaborative software engineering.

[3] Hans Gallis, Erik Arisholm, and Tore Dyb. An Initial Framework for Research on Pair Programming. In *International Symposium on Empirical Software Engineering*, pages 132–142. Rome, Italy, October 2003.

Annotation: This article examines the claims made about the reported benefits of pair programming. After a complete yet concise overview of pair programming, the paper surveys and discusses six recent and well-cited papers that empirically evaluated pair programming. In the context of this survey, a framework is proposed to assist ongoing qualitative and quantitative empirical research on peer programming. The framework is described by three main aspects: independent variables such as programmer collaboration, dependent variables such as development time and cost, and context variables such as the programmers being evaluated and the task to be completed.

The majority of this paper discusses and describes the framework, and while this provides an informative insight into the issues surrounding peer programming, no evaluation of the framework is made. Regardless, the discussion of peer programming, and the degree of examination of peer programming evaluation techniques makes this paper recommended reading for all researchers of collaborative software engineering. Additionally, the paper has many great references to leading related work.

[4] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, Reading, MA, 1st edition, October 1999.

Annotation: Extreme Programming (XP) is a collaboration-centric software engineering methodology. Whilst literature related to XP is limited—most probably due to the very nature of XP itself, this book appears to be the authoritative background material for XP. This book discusses the roots and philosophies of XP, and outlines the motivating factors for the use of XP, including a discussion on why previous yet accepted methodologies often fail. After explaining the problems that XP endeavors to solve, this book outlines the concrete methodology for practicing and adopting XP. This book is recommended reading for any collaborative software engineering researchers who want to learn about the theoretical underpinnings of XP, including pair programming, code walk-throughs, and the 'planning game'.

3.1.4 Tool Support

There is a wide range of tools available to the software engineer. At one end of the spectrum, well designed simple tools have always enjoyed widespread use and popularity. At the other end of the spectrum, massively complicated, monolithic and expensive Computer Aided Software Engineering (CASE) tools that incorporate code editors, diagrammers, project management components and debuggers are used exclusively by some corporations, but have not gained wide spread acceptance elsewhere.

The listing presented here provides articles on tools that have been successful in commercial and open source development groups alike. All of the tools presented in this listing are single user tools, with the exception that they can integrate with code repositories such as CVS. It is important for potential Cse researchers to be aware of these tools, including their capabilities and limitations—it is from these tools that we can envisage how collaboration can be supported in future generations of development.

Publications

[1] Christopher Garrett. Software Modeling Introduction: What Do You Need from a Modeling Tool? White Paper, Borland Software Corporation, 28 May 2003.

Annotation: As the title suggests, this white paper presents the core features that a modeling tool should require. The author does not cite any supporting publications; therefore this paper should only be used as an introduction into the Together Control Center IDE. An interesting aspect of this paper is that even though there are numerous screenshots of the Together IDE, the paper fails to name the system—the screenshots are only there to support the paper's arguments about what an IDE should provide. Despite its name, Together does not possess any specific collaborative properties, but it has been a very popular and extensible IDE for many years. Multiple language support, and support for all UML diagrams, design patterns, refactoring, reverse engineering, reporting, and round trip engineering are all features of this tool. Together also has support for source code management modules. As an aside, Together is widely regarded as the largest Java application in existence, but the system's main claim is that it is the first tool that supports 'continuously synchronised' models and code—otherwise known as *roundtrip engineering*.

This white paper provides an excellent discussion relating to the role of models within software engineering, and makes sensible analogies between software engineers and other professions. The paper also argues that tools such as Together help bridge the communication gap apparent amongst developers. Even though this is just a white paper, it is written in an unbiased manner. Its introduction provides an excellent context for the remained of the material, and the example screenshots are useful to researchers as well as users of IDEs.

[2] Jason E. Robbins, David M. Hilbert, and David F. Redmiles. Argo: A Design Environment for Evolving Software Architectures. In *Journal of Collaborative Software Engineering*. ACM, Boston, MA, USA, May 1997.

Annotation: ArgoUML, as described in this short paper, was one of the first tools to support analysis continually during the evolution of the model, not just at the end of each completed stage. ArgoUML provides feedback by critiquing the model and then appending notes to a 'todo' list, with the appropriate problems highlighted within the

model. This short paper is restricted to introducing the second conceptual version of ArgoUML, and presenting a few of the arguments why continuous analysis is useful to the modeling of software systems.

The version of ArgoUML presented in this paper was constructed in Java; the first version was written in SmallTalk and demonstrated in ICSE-17. Whilst this is a short paper, there is a very detailed and informative thesis written by Robbins to read as background information if required. Furthermore, the commercial and collaborative version of the product has extensive user documentation—see Section 2.2 on page 7.

[3] IBM Rational Rose XDE Developer. White Paper, April 2004. URL `http://www3.software.ibm.com/ibmdl/pub/software/rational/web/datasheet%s/version6/xde.pdf`.

Annotation: XDE, whilst now owned and maintained by IBM, previously existed as Rational Corporation's main set of tools to support the Rational Software Development Process. Since being taken over by IBM, the set of tools has been ported to exist within several popular IDEs, including Eclipse, WebSphere, and Visual Studio. It can also run in standalone mode. This article presents a brief summary of the main components provided by XDE, and gives details on the different environments that they can run in.

XDE is another family of products to support the complete lifecycle of software engineering. In particular, XDE supports an extensive range of modeling, including UML, data modeling, and IDE specific models. In terms of development, it supports most popular design patterns, software reuse, and template generation. It also support an array of runtime testing and debugging tools. For researchers of collaborative software engineering, this article gives brief details on XDE's collaborative features, including configuration management, multiple software models, and merge support. The XDE toolset is particularly interesting in its ability to support numerous source code repository systems including Microsoft's Source Safe, IBM's Clear Case, and the generic SCC interface.

[4] Eclipse Platform Technical Overview Version 2.1. White Paper, Object Technology International Incorporated, February 2003. URL `http://www.eclipse.org/articles/`.

Annotation: Eclipse is currently the largest research and development platform for IDEs and related tools. Whilst it has no published refereed research papers describing its architecture, a wealth of technical articles and white papers are available from the official web site, including the paper listed here. It is important to include the Eclipse platform within this bibliography due to its large following, including nearly all the major players in the software engineering research community.

This white paper describes the overall architecture of the Eclipse platform—an open source, open community project for the development and extension of software tools based on a commercial-strength underlying architecture. Currently, plugins for Eclipse that form a Java IDE are used by approximately half of the world's Java programmers, and many companies have developed third party extensions for Eclipse via the plugins API—including various plugins to support collaborative software engineering. Additionally, since the advent of Eclipse, many of the main IDEs have introduced compatibility frameworks not just to allow interoperability with Eclipse, but for interoperability with each other.

For an overview of the Eclipse IDE, the plugins API, or the Eclipse consortium of developers, this white paper is the most suitable single reference. It includes detailed information related to the key components of Eclipse, along with a useful case study of how the a java development tool is implemented in Eclipse. Eclipse is a platform well worth considering for researchers of collaborative software engineering; many collaborative features have already been introduced by third-party developers.

[5] Visual Studio Developer Center, Microsoft Corporation, July 2003. URL `http://msdn.microsoft.com/vstudio`.

Annotation: Visual Studio is a large player in the IDE market, and the URL supplied in the above reference is the authoritative source for numerous white-papers and

other technical documents. Within industry, Visual Studio is one of the most commonly used software engineering tools, and it is also the IDE of choice for many computer science laboratories—within both the research and education sectors. Visual Studio has well-tuned support for code editing, including class library browsing, and extensive code completion capabilities. It also has plugin-based version control and configuration management support.

Interestingly enough, Visual Studio is not actually 'visual'. To date, it still does not support round-trip engineering for code and diagrams; it only allows specific types of diagrams to be imported from Microsoft Visio and translated to skeleton code. Recently however, Together has provided a plugin for Visual Studio, allowing Together's round-trip modeling capabilities to be utilised within Visual Studio. It is important however to emphasis that Visual Studio by itself is not a design environment—it only supports the editing of code. Additionally, very little support for software metrics or unit testing is provided. It does, however, have excellent facilities for memory leak detection, and the debugger is second to none.

[6] WebSphere Studio Application Developer Version 5.1.1. White Paper, January 2004. URL `www-306.ibm.com/software/integration/wsadie/library`.

Annotation: After the retirement of the VisualAge suite of development tools, IBM's main commercial IDE environment is WebSphere Studio. WebSphere is current hosted within the Eclipse environment to provide the User Interface components, but underlying this is an array of design and runtime support tools.

This white-paper contains many useful screen shots of various parts of the IDE, and describes the main components of WebSphere in a logical objective manner. The white-paper also provides a comprehensive specification listing, which details the features currently supported by WebSphere—the main categories are Web, Web Services, Database, and Team Development. For the collaborative software engineering researcher, this paper is well worth reading to obtain a brief overview of the teamwork features supported by WebSphere, including its support for Rational's ClearCase configuration manager (see Section 3.3).

3.2 Groupware and Cscw

Groupware and Computer Supported Collaborative Work (CSCW) is an area that clearly lends itself to Cse tools and research. Through groupware technologies, user interfaces for simple tools can be replicated and shared in real time by multiple users, and such technology can be directly applied to assist the development of multiuser software engineering tools.

Conventional applications constructed through groupware are typically shared whiteboard editors, chat facilities, and map and graph browsers. The Cse researcher must be aware, however, that for complicated applications such as software engineering tools, groupware has its limitations that are of particular concern for Cse tools.

In this section, we present general literature related to groupware as well as specific classes of groupware including programmable toolkits, desktop systems and constructed applications of groupware. This section then presents articles that discuss the limitations of groupware, which is pertinent to any researcher considering the construction of a Csetool.

3.2.1 General Literature

The listing given here presents essential background reading for Cse researchers. A comprehensive survey and taxonomy of groupware technology is given along with articles that provide guidance for the application of groupware.

Publications

[1] Saul Greenberg. The 1988 Conference on Computer-Supported Cooperative Work: Trip Report. In *SIGCHI Bulletin*, volume 20 of *5*, pages 49–55, July 1989. Also published in Canadian Artificial Intelligence, **19**, April 1989.

Annotation: This report is Saul Greenberg's writeup of the 2nd Annual ACM conference on CSCW. From this introduction to CSCW, Saul worked on many projects, and soon became recognised as a leader within the field of groupware and collaboration. This report represents a snapshot of the state-of-the-art of CSCW in 1988, where ideas based upon remote collaboration and information sharing were just being conceived.

The report outlines each of the sessions presented in the conference, including synchronous communication, evaluation of systems, enabling environments, and practical system development experiences. One warning within this paper was that researchers must remember to work on is likely to be relevant and useful to the general community, given the broad spectrum of possibilities with CSCW.

[2] W. Greg Phillips. Architectures for Synchronous Groupware. Technical report, Department of Computing and Information Science, Queen's University, Ontario, Canada, May 1999.

Annotation: A comprehensive survey of synchronous groupware systems. The survey discusses over 100 architectures that were presented in the 1990s. Each system is described by one of three views—reference models, architectural styles, or distribution architectures. For distribution architectures, a new descriptive framework named Interlace is used. This survey is an excellent starting point for any background information relating to real-time collaboration architectures.

[3] Prasun Dewan. An Integrated Approach to Designing and Evaluating Collaborative Applications and Infrastructures. *Computer Supported Cooperative Work*, 10(2):75–111, January 2001.

Annotation: This paper introduces a research plan for the design and evaluation of collaborative systems. Whilst the research plan of system decomposition and evaluation appears to be well formed, the paper is more notable for its comprehensive survey of collaborative frameworks and applications—this survey was required to illustrate the suitability of the evaluation method amongst an array of CSCW research projects. This paper surveys many groupware systems including GroupKit and XTV, but also presents collaborative software engineering tools such as FLEECE and Suite. Please see Section 2 on page 5 for a discussion of such tools.

3.2.2 Programmable Toolkits

Programmable groupware toolkits are used to rapidly construct CSCW applications from common multiuser components. Essentially, programmable groupware toolkits are all that developers require to facilitate communication between a group of distributed applications unless highly specific networking or collaborative features is to be supported.

Typical components within groupware toolkits includes shared text editors, chat facilities, sketch-pads, voting facilities and mechanisms to support group membership, connection and disconnection. Some programmable toolkits also provide comprehensive floor control policies; for the remainder, all concurrency control must be implemented by hand.

A multitude of programmable toolkits exist today. The articles presented in this listing describe some of the more prominent systems, and each article describes how to construct applications from the given toolkit. For the CSE researcher, it is useful to determine what facilities and components are available from the current range of programmable toolkits, which in turn provides the scope of CSCW possibilities for CSE tools. Any collaborative facilities outside this scope will more than likely have to be created by the tool designer.

Publications

[1] Mark Roseman and Saul Greenberg. Building Real Time Groupware with GroupKit, A Groupware Toolkit. *ACM Transactions on Computer-Human Interaction*, 3(1):66–106, March 1996.

Annotation: GroupKit, a toolkit to allow the development of collaborative applications, is arguably the most notable research output from within the CSCW field. This paper provides a comprehensive overview of the second, and vastly improved version of GroupKit. The first chapter provides a brief introduction to the GroupKit architecture, and also presents the core requirements for any groupware toolkit. Aside from the discussion and an extensive related work section at the end of the paper, the remainder of this article provides a user guide for developers of collaborative applications.

Whilst detailed enough to illustrate programming concepts, the user guide is rich in

observations and research-related comments; the experiences shared in the development of this toolkit are well worth noting, regardless of whether the reader intends to use the toolkit or not.

[2] Bela Ban. Design and Implementation of a Reliable Group Communication Toolkit for Java, September 1998. URL `http://citeseer.nj.nec.com/ban98design.html`. Cornell University.

Annotation: This paper presents the design and implementation of JavaGroups, a toolkit for group communication for Java applications that is closely based upon the Horus group communications system. The motivation for such a system is that very few group communication mechanisms are available for the Java language; this toolkit intends to provide object-oriented components that are relatively easy to integrate with existing Java applications.

JavaGroups focuses on reliability, using the concept of redundancy to cater for components that could fail within a group of communication processes. As well as presenting the system architecture, this paper also identifies and discusses the design patterns within JavaGroups, which may be valuable to both users of the system, and any researchers of other group communication toolkits. This paper is essential reading for any researcher who wishes to design an object oriented group communication framework, or any associated applications.

[3] Rich Burridge. *Java Shared Data Toolkit User Guide*, October 1999. URL `java.sun.com/products/java-media/jsdt`.

Annotation: This manual provides an overview of the Java Shared Data Toolkit (JSDT). The first chapter presents the need for a toolkit that supports highly interactive applications, and abstractly describes how the JSDT is suitable for most collaborative programming needs. The second chapter outlines the key components of the toolkit, and explains abstractions made to support the notion of users, sessions, and data. The remainder of the manual provides a user guide, outlining keys classes and example usages. The JSDT is a useful and very comprehensive tool for designing collaborative applications, and this manual is the definitive source of information for researchers and developers.

[4] H. Abdel-Wahab, B. Kvande, O. Kim, and J.P. Favreau. An Internet Collaborative Environment for Sharing Java Applications. In *Proceedings of the Fifth IEEE Workshop on Future Trends of Distributed Computing Systems*. Tunis, Tunisia, October 1997.

Annotation: This paper describes a Java package named JCE which allows single-user applications to be adapted to multi-user versions, with minimal effort. The Java Collaborative Environment, as this paper describes, overrides the existing Java Abstract Windows Toolkit (AWT) with multiuser components. Given these specialised components, a session manager, and a framework that captures, distributes, and remotely reproduces user input events, the JCE allows simple applications to be shared collaboratively.

This paper presents an example of a whiteboard that can be replicated on any number of remote systems, and given this example, the session manager is also presented, along with a discussion of JCE's floor control policies. One considerable drawback of the system is the need to rewrite the AWT components upon every new version release from Sun, and no literature has been found that suggests JCE still exists now that the AWT has been superseded by the Swing toolkit.

[5] E. James Whitehead Jr. and Yaron Y. Goland. WebDAV: A Network Protocol for Remote Collaborative Authoring on the Web. In *Proceedings of the Sixth European Conference on Computer Supported Collaborative Work*, pages 291–310. Copenhagen, Denmark, September 1999.

Annotation: This article presents a high-level overview of the aim, purpose, and mechanisms of the WebDAV protocol. WebDAV was proposed by the Internet Engineering Task Force as a mechanism to support the continual editing of web resources such as web pages and images. Traditionally, such resources were only downloaded, but the WebDAV protocol defines a set of standards to allow continual refinement and alteration from any number of authorised users. WebDAV employs only course-grain file locking and simple ftp-like protocols for

the obtaining and uploading of resources; because of this, many non-collaborative applications have enjoyed an easy migration path to collaboratively maintain web servers.

WebDAV has experienced widespread adaptation—the apache web server and Microsoft's suite of Office tools both support the full WebDAV protocol. Recently, Adobe has released its latest version of the Acrobat PDF reader tool—with WebDAV support it now allows the collaborative editing of annotations to online PDF documents. Whilst WebDAV is only a protocol for web-based handling of resources, its existence at the network-layer allows any set of tools to integrate with no additional third-party components. This suggests that use of the WebDAV protocol is well worth considering for any collaborative tool or framework that is client/server based.

[6] Till Schummer. COAST: An Open Source Framework to Build Synchronous Groupware with Smalltalk, German National Research Center for Information Technology, October 2002. URL `http://www.opencoast.org/documentation`.

Annotation: This is a draft chapter that discusses the COAST framework for a yet to be published collaborative SmallTalk book. Whilst the seminal paper on COAST was published in the proceedings of the 1996 ACM conference on CSCW (pages 30–38), this book chapter presents more detailed and updated material on the COAST framework—a toolkit for providing collaboration support for SmallTalk applications.

The paper starts with an outline of the requirements for a groupware framework, which are very-much inline with those specified in the initial GroupKit [1] paper. The paper then presents an example application—that of a collaborative UML editing tool, using a related-WYSIWIS style. The paper also discusses the underlying COAST architecture, including details about its model-view-controller design, and the concept of the COAST domain model. During this description, the paper presents a large and informative 'behind the scenes' section, discussing many of the real issues that face developers of collaborative applications and support tools. The paper concludes by showing some excellent example applications that use the COAST framework.

[7] Jason Hill and Carl Gutwin. Awareness Support in a Groupware Widget Toolkit. In *Proceedings of the International ACM SIGGROUP Conference on Supporting Group Work*, pages 256–267. ACM Press, Sanibel Island, Florida, USA, November 2003.

Annotation: This paper presents MAUI—a Java toolkit for the construction of collaboration-aware user interfaces and applications. The paper describes the components of the MAUI toolkit, including multi-user replacements for the standard Swing UI components, and specialised components such as telepointers. All components within the toolkit are packaged as Java Beans, allowing easy integration with IDEs; the paper presents an example application being constructed via the JBuilder development environment. An interesting feature of MAUI, and something new to the field of groupware, is that of both design-time run-time customisation of the multiuser widgets.

The paper gives several screen-shots well worth examining, descriptions, and discussions of the MAUI widgets, and the run-time and design-time custom dialogs. The system architecture is also described in detail, and the paper concludes by presenting an example MAUI-based application in the form of a shared web browser. This paper introduces an excellent new toolkit for the construction of multiuser Java applications, and should be read by any developer considering the construction of new collaborative tools.

3.2.3 Desktop Systems

Desktop CSCW systems are designed to support general workflow and collaboration without the need for customisation or special tool development. They are monolithic systems that allow common applications, documents and data sets to be shared, often in real-time. Such systems are useful for most general workflow contexts such as document editing and project scheduling, but do not lend themselves directly to most software engineering tasks. Desktop systems do however provide a good illustration of how CSCW technologies can be applied to facilitate computerised structured teamwork.

Many desktop CSCW systems exist; this listing provides articles of such systems that have a strong research theme. The range of desktop CSCW systems presented in this listing is varied from simple chat applications such as NetMeeting and JAMM to full artifact and workflow management such as CVW and Lotus Notes.

Publications

[1] Windows NetMeeting, Microsoft Corporation, February 2004. URL `http://www.microsoft.com/windows/netmeeting/Authors`.

Annotation: This is a reference to the official documentation for Windows Netmeeting. Whilst no cited publications are dedicated to NetMeeting, this tool is arguably the most popular piece of commercial collaboration software ever built. Netmeeting allows single-user applications to be replicated transparently between several users, with the original instance of the application acting as the server. NetMeeting also supports instant audio, video, and text-based chat, along with a simple shared whiteboard utility. This reference also includes a link to the NetMeeting API documentation, where Windows-based programs can integrate and embed NetMeeting services.

[2] James Begole, Mary Beth Rosson, and Clifford A. Shaffer. Flexible Collaboration Transparency: Supporting Worker Independence in Replicated Application-Sharing Systems. *Transactions on Computer-Human Interaction*, 6(2):95–132, June 1999.

Annotation: This article presents an enhanced application sharing system named JAMM - Java Applets Made Multiuser. The authors argue that conventional application sharing systems, where collaboration is transparent to the developers and users of the shared applications, are inefficient and lack key support for groupware principles such as concurrency control and group awareness. Additionally, examples are given to strengthen these claims. The authors then introduce the collaboration-transparent JAMM framework, where features previously available only within specialised, collaboration-aware applications are provided, such as multiuser widgets and awareness mechanisms.

The JAMM architecture operates by replacing conventional user interface components with multiuser versions at runtime; this article demonstrates the system using an example of a document editor that is converted at runtime to support concurrent editing, telepointers, and a ‘radar view’ of all users currently editing the document.

A pleasing aspect of this article is the well-described evaluation section, where results suggest that JAMM is faster for task completion and preferable to the NetMeeting application sharing system for several simple text-editing trials that varied in degree of required collaboration.

[3] Peter Spellman, Jane Mosier, Lucy Deus, and Jay Carlson. Collaborative Virtual Workspace. In *Proceedings of the International ACM SIGGROUP Conference on Supporting Group Work*, pages 197–203. ACM Press, NY, Phoenix, AZ, November 1997.

Annotation: This paper introduces the Collaborative Virtual Workspace - a ”placed-based” system for supporting both synchronous and asynchronous collaborative tools. The authors argue that a merger of session-based tools such as netmeeting [1] and document-based tools such as Lotus Notes [4] is required, because realistic collaboration requires both solid support for real-time communication as well as document management.

This paper describes the CVW architecture, and details the lessons learnt from its initial deployment. CVW uses the rooms metaphor to separate different groups of users, and provides video and audio conferencing, shared whiteboards, and a document tracking utility. Roles for users are loosely defined, and searching of a collaborative workspace for people and documents is possible. Whist the CVW framework is now several years old, this paper is well worth reading for a detailed example of a large-scale collaboration manager.

[4] Berthold Reinwald and C. Mohan. Structured Workflow Management with Lotus Notes Release 4. In *Proceedings of the 41st IEEE Computer Society International Conference (CompCon)*, pages

451–457. Santa Clara, California, February 1996. URL `http://citeseer.nj.nec.com/reinwald96structured.html`.

Annotation: This paper describes how to implement customised workflow management using Lotus Notes Release 4. Whilst at time of writing the seminal reference for the Lotus Notes architecture is the *Lotus Notes Developer's Guide, Version 4.0*, Cambridge, MA, the paper referenced here also gives a good description of the Lotus Notes architecture, and this paper has been peer reviewed.

This paper focuses on implementation of tools for workflow management, however, the background information and reference list provides a comprehensive overview of the Lotus Notes architecture. The paper discusses the Notes database, the address book, multimedia documents, customisable forms, event-based agents, and user views. Lotus Notes, as described in this article, is a typical example of a workflow management framework, where artifacts are routed to specific users based upon the defined processes and roles. A handful of collaborative software tools have been constructed using Lotus Notes, including the GWSE data repository and the CAIS software inspection tool (see Section 2.4 on page 11 and Section 2.1 on page 5 respectively).

[5] Ludwin Fuchs, Steven E. Poltrock, and Ingrid Wetzel. TeamSpace: An Environment for Team Articulation Work and Virtual Meetings. In DEXA *Workshop*, pages 527–531, 2001.

Annotation: This article presents TeamSpace, a collaborative workspace system that supports general development teams. Within this article, screen-shots of the TeamSpace system show web interfaces for group discussions, shared project management utilities, and an artifact tracking system. TeamSpace allows general objects to be shared such as text documents, meeting schedules, and presentations. A general pool of formally-specified information for each development team is also shared, including a breakdown of each task, status reports, testing plans, and work artifacts such as documents. Task are broken down to small single-person components, and the TeamSpace architecture monitors the progress of completion, and is also used to facilitate group communication.

This article also provides an objective discussion, relating TeamSpace to other workspace collaboration systems within the field of research. The paper concludes by proposing enhancements to TeamSpace that will allow it to systematically monitor changes in artifacts and notify all related users.

[6] W. Appelt. WWW Based Collaboration with the BSCW System. In *Proceedings of the 26th Conference on Current Trends in Theory and Practice of Informatics*, number 1725 in Lecture Notes in Computer Science Series, pages 66–78. Springer-Verlag, Milovy, Czech Republic, November 1999.

Annotation: BSCW (Basic Support for Collaborative Work) was introduced in 1995 at the first web-based groupware system. Since its introduction, many research and commercial projects have used BSCW to provide asynchronous and synchronous collaboration between participating users, using the shared-workspace metaphor. BSCW is still being developed today—the last version release was in February 2004. This paper presents a concise overview of the more recent versions of BSCW.

The paper describes the general goals, implementation, and features of BSCW, including several application screen shots and minor examples of usage. The paper addresses the full range of BSCW features, including the event model, user authentication, search facilities, and discussion management. Only a brief mention of related work and the theoretical underpinnings for BSCW is made, but the overview is very informative.

[7] Ward Cunningham. Wikiwikiweb front page, April 2005. URL `http://c2.com/cgi/wiki?WikiWikiWeb`.

Annotation: Wiki is a collective term used to describe websites that are editable by all subscribed users, and are described by its inventor as"The simplest online database that could ever work". There have been many implementations of discussion servers that allow collaborative editing of pages, but Wikis have been immensely popular with computer scientists since their introduction in the mid 1990s. An important feature of

Wikis is their policy of allowing any user to add new pages, and to automatically link pages both from within the Wiki and other external Wikis and web resources. Wikis do not allow synchronous editing of pages, but do allow CVS-based page locking for controlling multiple concurrent edits. Through the use of a CVS back-end, the full change history of an evolving wiki is accessible.

The URL given in this listing is the front page for the wiki wiki web. From here there are many links to frequently asked questions and answers, wiki editing tutorials, and discussion documents. Wikis are already used for collaboration in software development, and will undoubtedly play a large part in future Cse tools in one form or another. The idea of computer-supported editing and automatic management of hypertext documents, as provided by Wikis, is likely to be popular with the next generation of groupware technologies.

3.2.4 Constructed Applications

This listing presents articles which describe the implementation of applications constructed with groupware toolkits. These papers are intended to give the Cse developer insight into the advantages and disadvantages of using groupware, assisting in the design and implementation decisions of Cse tool mechanisms and user interfaces.

For additional papers of this nature, section 2 presents a paper describing GroupARC, a collaborative tool built from the GroupKit toolkit. The GroupKit toolkit, as presented in section 3.2.2, also includes comments from developers who have used the toolkit to construct collaborative applications.

Publications

[1] Neville Churcher and Clare Churcher. Realtime Conferencing in GIS. *Transactions in GIS*, 3(1):23–30, March 1999.

Annotation: This article discusses how CSCW has been applied to Geographical Informations Systems (GIS). A system for collaborative GIS browsing, named GroupARC, was developed using the GroupKit multiuser toolkit, and this paper describes the lessons learned from the development of this system. GroupARC allows the display of spatial information, typically maps or other rich data sets, in a relaxed-WYSIWIS manner. This system is a comprehensive example of a tool that successfully uses the GroupKit toolkit; browsing this paper will give the reader a solid perspective on the task of constructing non-trivial collaborative applications using groupware technologies.

[2] Donald Cox and Saul Greenberg. Supporting Collaborative Interpretation in Distributed Groupware. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work*, pages 289–298. ACM Press, Philadelphia, PA, December 2000.

Annotation: This paper presents the PreSS—a system for supporting distributed collaborative interpretation within a spatial visual workspace, where collaborative interpretation is defined as the process of categorising a set of many fragments of information into a coherent set of meaning descriptions. Whilst collaborative interpretation is not directly related to collaborative software engineering, this paper is valuable for its description of the implementation and evaluation of a complex system that accommodates the social dynamics of its users.

PreSS was developed using GroupKit, and this paper presents numerous screenshots giving the reader an appreciation of the types of interfaces that are possible to construct with this toolkit. The evaluation section is also valuable, in particular the points raised relating to issues workspace awareness between users.

3.2.5 Limitations of CSCW

Despite the original claims and promises made in CSCW and groupware literature, it is still a daunting task to implement Cse tools. CSCW technology is based on the support of unstructured and transient documents that have little or no semantic relationship to other artifacts. This is converse to the foundations of software engineering: highly structured, evolving documents that have vast interdependencies and long lifetimes.

The literature surrounding CSCW originally spoke of solving all problems related to computerised collaborative work, suggesting that

CSE would be trivial to implement once technology had naturally progressed. Unfortunately, twenty years on from those claims, we are still no further to having a 'silver bullet' technology that facilitates collaborative software engineering or any other complicated application. Accordingly, the literature listed here discusses the failings of CSCW, and makes reassessments of the useful scope of CSCW technology.

Publications

[1] Jonathan Grudin. Why CSCW Applications Fail: Problems in the Design and Evaluation of Organizational Interfaces. In D. Marca and G. Bock, editors, *Groupware: Software for Computer-Supported Cooperative Work*, pages 552–560. IEEE Press, Los Alamitos, CA, 1992.

Annotation: This is an excellent and well cited article written when research into CSCW was in its infancy. Grudin showed great insight when he pointed out in this article that many CSCW applications will fail given the current approach into CSCW application development. The key problems identified are: the disparity between the people making the effort to build the system and the people who use the system, the poor intuitions of developers for building multi-user applications, and the extreme difficulty in analysing and evaluating such complex applications. This paper gives a detailed discussion on each of these problems, and presents four case studies of applications that have already failed due to some or all of these problems.

The conclusion to this paper suggests that we must have a better understanding of how groups function and evolve before attempting to develop an application to automate the collaboration. This paper also include very useful appendices, where discussions related to different types of collaborative applications are presented. The appendices conclude by stating that CSCW applications should not cause considerable disruption to users and their roles if they are to be introduced and utilised successfully.

[2] Jonathan Grudin. Groupware and social dynamics: Eight challenges for developers. In *Communications of the ACM*, volume 37 of *1*, pages 92–105. ACM Press, January 1994.

Annotation: This is another high quality discussion document from Grudin related to the challenges and perils of CSCW development. This article can be viewed, in part, as an updated of Grudin's earlier paper "Why CSCW applications fail" [1], but it also provides some examples of CSCW successes. After providing a brief history of groupware, this paper described eight problem areas—much of which builds upon the problems presented previously. The conclusion gives advice on how to build successful CSCW applications, including extending existing systems rather than writing applications from scratch, finding niches where existing groupware applications succeed, and ensuring that the groupware systems benefit all users of the system, not just people in specific roles.

3.3 Source Code Configuration Management

Source Code Configuration Management (SCM) systems are core to software engineering. Such systems enable the versioning, branching and management of software engineering artifacts to ease the burden of producing multiple versions of software products derived from the efforts of potentially hundreds of developers. Even for single user projects, the benefits gained from SCMs including the ability to roll-back changes makes the use of such systems warranted and valuable. SCMs also attempt to keep source files coordinated as they evolve by allowing regular check-ins and project builds. Without such facilities, it is possible for individual coding efforts to skew the project into several separate and hard-to-consolidate directions.

SCMs address the fact that many people may be working on the same code base, and that often several developers will want to work on the same source file. To facilitate controlled file sharing, two schemes are typically employed: file locking or file coping and subsequent merging. As explained by the various articles in this section, both approaches have their advantages and problems. Regardless of the issues surrounding the use of SCMs, such systems are a fundamental component for most software engineering tools, both collaborative and single user.

This section provides references to articles about conventional SCMs, as well as articles on the latest versions of SCMs that have some major advantages over their predecessors. This section also provides some CSE specific SCM systems as well. The articles in this listing are essential reading for CSE researchers—it is imperative that an understanding of the challenges of concurrent development is obtained before considering the development of CSE tools. Additionally, social protocols dictate the use, effectiveness and impact of SCM systems; this aspect of SCM use is discussed within several of the articles presented here.

Publications

[1] Walter F. Tichy. RCS — A System for Version Control. *Software — Practice and Experience*, 15(7):637–654, 1985.

Annotation: This is the main paper published to describe RCS (a Revision Control System), one of the first globally-popular SCM systems. While readers could also look at *"The Source Code Control System", M. J. Rochkind, IEEE Transactions on Software Engineering, December 1975* as the first ever published SCM system, this paper by Tichy is probably the most relevant article related to the first generation of modern day SCM systems. Even this paper is very old now, but it is an important read for those wishing to understand the basis of how SCM systems works today.

This paper represents both a design document and a user manual for the RCS system. It presents the concept of branching in detail, and also discusses how the Make utility and RCS can be integrated to manage software engineering projects. Other topics included in this paper include a discussion about the fundamentals of delta storage, algorithms used to computing file deltas within RCS and details of locking in RCS. The paper then provides statistics of Purdue's RCS research repository, and concludes with a survey of other SCM tools available in the year of publication.

[2] Brian Berliner. CVS II: Parallelizing Software Development. In *Proceedings of the USENIX Winter 1990 Technical Conference*, pages 341–352. USENIX Association, Berkeley, CA, 1990.

Annotation: Surprisingly, there are not a lot of academic papers on CVS. This paper, however, is well cited and gives a good historical account of how CVS was conceived and developed. CVS is a front-end that sits atop RCS, and provides a copy-modify-merge optimistic scheme of version control, as opposed to the lock-modify-unlock mechanism employed by RCS and SCCS. CVS was a natural progression from these first generation of SCM systems, and gained immediate widespread use. In the decade after the publication of this article, CVS has been enhanced considerably, with server-based CVS being one of the most significant advances.

This paper gives a basic overview of the CVS architecture, showing how optimistic locking operates, and how the files can be patched upon concurrent modification. The paper also shows how CVS can be used to branch between revisions, using two versions of a kernel as an example. This paper gives some statistics on file changes for the Prisma kernel, showing how many files and lines of codes have changed, for example. Finally, the paper makes brief mention of CVS's performance. This is an old paper now, but it is useful in understanding how today's revision control systems work, and where such technologies originated from.

[3] Open Source Development Network. SourceForge.net Home Page. Internet URL, Open Source Technology Group, July 2003. URL `http://sourceforge.net/docman/display_doc.php?docid=6025&group_id=1/`.

Annotation: There are no definitive academic papers that describe source forge, an open-source code repository, but as it is the world's largest source code repository is it well worth mentioning within this annotated bibliography. The article given here provides a brief overview of source forge, including a description of what open source software is, and where to obtain further information on source forge.

Source forge is essentially a web-based interface for a CVS repository, but in addition, each project has a home page, a bug tracking database, a documentation area, usage statistics, public forums and mailing lists. At the time of writing, there were 100,000 projects with 1,000,000 users in the source forge repository. The most popular project,

GAIM, has had to date 13,000 file commits, 3000 file adds, and the web pages within the project have been accessed 34 million times in the last four years. The basic version of source forge is free to use via the main source forge servers, or you can locally install an enterprise edition that provides better security and integration options from a commercial vendor.

[4] Ben Collins-Sussman, Brian W. Fitzpatrick, and C. Michael Pilato. *Version Control with Subversion.* O'Reilly Media, 1 edition, June 2004. URL `http://svnbook.red-bean.com/en/1.0/svn-book.pdf`.

Annotation: Subversion is becoming accepted worldwide as the replacement for CVS. Subversion is essentially a rewrite of CVS, with the same underlying repository model, but without the bugs and design flaws. Major advantages over CVS are: it incorporates directories and symbolic links as well as files in the repository, and as a result, it can now handles file renames and deletes as modification events rather than treating these actions as the end of a revision branch. Subversion also handles atomic commits, which means that either all changes are accepted, or none at all. This prevents problems that can occur when only a subset of updated files are checked back into the repository. Subversion can also handle several different supporting network layers such as Webdav and secure shell (SSH).

This book is the official documentation for subversion. It is available free on line, and a hard copy is available for purchase from O'Reilly Media. This is a very well written book that provides three main aspects: it gives an excellent overview of the subversion system, it provides a clear and concise user manual, and it also gives a lot of material for the further development of subversion and associated tools. For articles related to subversion that have more academic content, the best place to look is actually papers related to CVS, as subversion is based nearly completely on CVS.

[5] Mark C. Chu-Carroll and Sara Sprenkle. Coven: Brewing Better Collaboration through Software Configuration Management. In *SIGSOFT '00/FSE-8: Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 88–97. ACM Press, New York, NY, USA, 2000. ISBN 1-58113-205-0.

Annotation: This paper introduces Coven, a collaboration-centric SCM that supports multiple organisational views of software artifacts, driven by a query-based repository The paper argues that people use SCM to work as a team on a subset of a project, and Coven's hierarchical structure is based upon this observation. Coven allows soft-locking of artifact fragments, where users can break the lock if they really wish to concurrently modify an artifact. This results in notification to all involved parties. This paper states that Coven does not introduce any new features to SCM, but does combine many useful features of existing systems to provide a SCM system suitable for use within collaborative software engineering tools.

Aside from describing the Coven architecture, this paper gives realistic scenarios of collaborative development, and shows how Coven is used within such a setting. The paper also provides a solid section on related work, and concludes with directions for future work such as building a user interface for collaborative development and studying real patterns of collaboration under Coven.

[6] Cleidson R. B. de Souza, David F. Redmiles, Gloria Mark, John Penix, and Maarten Sierhuis. Management of Interdependencies in Collaborative Software Development. In *International Symposium on Empirical Software Engineering (ISESE)*, pages 294–303. IEEE Computer Society, Rome, Italy, September 2003. ISBN 0-7695-2002-2.

Annotation: This article presents an eight-week observation of a group of 31 colocated software engineers working on a suite of tools within the NASA/Ames Research Centre. The code base consisted of 1 million lines of code, and the engineers were comprised of a development team and a testing team. The paper examined the activities of the software engineers to discover how interdependencies were managed from the viewpoints of both formal and informal software engineering methodologies.

The paper gives an interesting discussion on the realities of industrial software

development, in particular how the informal approaches are utilised to address shortcomings in tool support for the more formal methods. For example, email was used whenever a file is checked back in to alert others to possible conflicts. Whilst this was not part of the formal software development process, it was necessary to employ email, due to no other notification mechanism in place. By comparing the formal methodologies with the actualities of team-based development, this paper gives useful insight as to where further advancements can be made for computer supported collaborative software engineering. This paper also provides many useful references to previous case studies and discussion papers.

[7] Dave Thomas and Kent Johnson. Orwell: A Configuration Management System for Team Programming. In *Proceedings of Object-Oriented Programming Systems, Languages, and Applications (SIGPLAN)*, pages 135–141. San Diego, CA, 1988.

Annotation: This paper presents Orwell, a configuration management tool for multi-person Smalltalk projects. Orwell supports the sharing of source code and object code, and also provides version control. The authors claim that whist they only describe the implementation of a tool for multi-person Smalltalk programming, many of the design ideas can be easily adapted for other languages. Orwell defines classes as the unit of ownership with a Smalltalk project, and projects can be developed with or without the presence of a central server.

Whilst Orwell has been placed in the configuration management section of this bibliography, it could also be placed in the section related to existing Cse tools, as it provides many of the services that a collaborative software engineering tool requires, such as file sharing, revision control, and class library browsing. The more recent Tukan system (see Section 2.3 on page 8) uses Orwell as the underlying mechanism for its collaborative services.

3.4 Human Computer Interaction

A vast amount of Human Computer Interaction (HCI) research has been published over the last few decades. Whilst virtually all HCI research is relevant to Cse, this section contains a selection of papers that are directly related to the design of Cse tools and frameworks. There are certainly many other HCI papers that investigate team work, coordination of artifacts, and supporting awareness and collaboration, but these are not mentioned here for the sake of brevity. The papers presented in this section lead interesting discussions on the usefulness and effectiveness of collaboration within software engineering processes and tools.

HCI papers typically present micro-level evaluations where problems are isolated into a most simplistic form. This is a scientifically correct and well-accepted practice, but unfortunately for the purposes of Cseresearch, such research can be too trivial for software engineering purposes—gainly research into software engineering requires an acceptance that tools are complex and artifacts are numerous, and that simplification can yield results that are not significant in 'real world' terms. Accordingly, most HCI papers that address Cse only assess programming within isolated environments such as spreadsheeting tools. Therefore, the Cse researcher needs to be aware that HCI studies may not necessarily scale to realistic software engineering scenarios—often the papers are useful for general guidance only.

Of the papers presented in this section, many give detailed information on user trials and tool evaluations related to program understanding, task completion rates and levels of user satisfaction under collaborative settings. For the Cse researcher, such papers are useful in terms of providing general insights about patterns of collaboration in software engineering. These papers are also useful for providing examples of well-formed scientific evaluations.

Publications

[1] M. A. Storey, K. Wong, and H. A. Müller. How Do Program Understanding Tools Affect How Programmers Understand Programs? *Science of Computer Programming*, 36(2–3):183–207, 2000.

Annotation: This paper presents an evaluation of the effectiveness of software engineering tools to reducing cognitive overheads and improve program comprehension. The paper includes a detailed discussion of SNiFF+, a well researched and contrasted motlier development environment. SNiFF's collaborative features not used in this study, but this paper does provide an example of a well-planned evaluation to explore how tools affect programmers understandings. To learn more about SNiFF, its collaborative features are presented in section 2.3.

A user evaluation of 30 subjects is described where a range of tools was provided. The task for users was to identify units of code or to answer questions related to the structure and semantics of a non-trivial program. Users were free to use whichever tools they pleased, and they were also allowed to use whatever features of the individual tools that suited. After describing the experimental design, many detailed discussions are presented on the findings, with numerous observations given. The main outcomes of this study were that the tools did appear to enhance users comprehension strategies, or in other words, the exploratory features of the tools were utilised. In some cases, however, information overloading was observed. Other findings of the evaluation were that users felt the dependency visualisations were useful for program comprehension, as too was the ability to switch between high and low level views.

Apart from a well described and insightful evaluation, this paper provides an excellent range of references for further reading into program comprehension tools. For CSE researchers, this paper is worth reading for details as to what users do when they are navigating around source files within a complex program. The paper also provides a well-formed template for further evaluations of this type, which is useful for future CSE tool evaluations.

[2] J. W. Atwood, M. Burnett, R. Walpole, E. M. Wilcox, and S. Yang. Steering Programs Via Time Travel. In *IEEE Symposium on Visual Languages*, pages 4–11. Boulder, CO, September 1996.

Annotation: This paper introduces the dimension of time for supporting the development of software. Whilst the topic is not directly related to collaboration, it is useful for investigating how future CSE tools could support new modes of development, particularly from the aspect of private work. The paper uses the Forms/3 spreadsheet interface as the example programming environment, and illustrates how programmers can move back and forward in time to view different runtime program states, which the authors argue can assist in debugging and program development.

This article has a lot of content about visual programming languages. It also discusses a platform different to most software engineering tools, where runtime inspection and design time coding are not usually combined as they are in spreadsheets. Regardless of these differences, however, this paper is very useful to CSE researchers in terms of project exploration, problem isolation and development visualisation. The paper suggests possibilities that could be built into future CSE tools, and shows approaches to support private workspaces within a global project.

[3] Joanna DeFranco-Tommarello, Fadi P. Deek, Starr Roxanne Hiltz, Julian P. Keenan, and Cesar Perez. Collaborative Software Development: Experimental Results. In *Proceedings of the 36th Hawaii International Conference on System Sciences (HICSS'03).* Big Island, Hawaii, January 2003.

Annotation: This article presents one of the very few papers that details a structured experimental design for a CSE evaluation. A study was undertaken to assess whether a collaborative system could help subjects understand programming problems better and devise superior solutions. 174 subjects were tested over a three week period, using combinations of a collaborative system, groupware, and control configurations. Two judges assessed the approach and solution of each group, with results suggesting most outcomes were the same regardless of the tool mode used.

This paper is an important read due to its well written description of a thoroughly planned CSE evaluation, rather than because of its findings. Of the few significant results, however, it was shown that under collaborative conditions, participants did manage to understand the programming problem more thoroughly and were also judged to have planned their solutions better.

[4] Bonnie A. Nardi and James R. Miller. An Ethnographic Study of Distributed Problem

Solving in Spreadsheet Development. In *Proceedings of the Conference on Computer Supported Cooperative Work*, pages 197 – 208. Los Angeles, CA, October 1990.

Annotation: This paper presents a refined version of the original article "Twinkling Lights and Nested Loops". The basis of this paper is a discussion into the collaborative design and development of spreadsheets, which is of a high relevance to CSE research. The paper gives details on case studies and user comments related to the development of spreadsheets within organisations, and shows how multiple users cooperate in designing, developing and debugging them. An interesting observation made in this article is that the users generally have a high success rate in developing functional spreadsheets, even though spreadsheets were never intended to support collaborative development.

This paper observes that cooperative spreadsheet development within the investigated organisations was spontaneous and casual, where users employed existing informal social networks to initiate collaboration. The study also observed that many different levels of programming ability go into the development of single spreadsheets, spreadsheets are complicated with complex control structures, macros, and graphs, yet the design and development of spreadsheets is still a relatively easy process for most users.

This paper represents a very interesting read for all user interface designers: it shows a success story for collaborative development, makes some very informative observations, and provides guidelines for the future development of groupware and collaborative systems. For CSE researchers, it provides an interesting story of how socially driven collaboration can lead to the successful development of complex computational artifacts.

[5] E. M. Wilcox, J. W. Atwood, M. Burnett, J. J. Cadiz, and C. R. Cook. Does Continuous Visual Feedback Aid Debugging in Direct-Manipulation Programming Systems? In *Proceedings of CHI 97: Human Factors in Computing Systems*. Atlanta, GA, 22-27 March 1997.

Annotation: This paper does not directly related to user collaboration, but does provide an interesting and well structured evaluation into continuous visual feedback, something very important to CSE research. The study used the Forms/3 Visual Programming Language (VPL) to study 29 students, half of which were used as a control group. The evaluation showed that whilst continuous feedback did not significantly help in general, there were specific tasks where it did. The study also showed that subjects made significantly more changes when working with live continuous visual feedback. Another important observation was that three factors appeared to control the results: type of bug, type of user and type of problem.

This is a good article to read prior to designing evaluations of CSE systems as it gives a detailed illustration of a well structured and carried out evaluation. This article also gives an interesting discussion and insight into the merits of continuous visual feedback, a feature that most CSE systems aim to provide.

3.5 Distributed Systems

In terms of providing facilities for interprocess communication, groupware technology can offer basic distributed communication support. For the carrying of application specific data, or for where more complex and efficient systems are to be supported, then a distributed systems technology may be the only answer to support CSE tools.

Distributed systems aim to make the boundaries between computers invisible, a term often referred to as global computing. Distributed systems provide facilities to support client/server, pair-to-pair, and grid computing architectures. For the development of CSE tools, distributed systems allow tools to communicate with each other, send and receive custom data, access peripheral servers, and make calls to remote functions and methods.

Distributed systems technology has made significant advances in the last few years, particularly with the introduction of .net, J2EE, SOAP, and web services, all of which are explained in the annotations within this section. Due to these advances, it is possible to implement very comprehensive collaborative features within CSE tools, with functions more advanced than those typically supported in conventional groupware toolkits. Additionally, with the advent of the Internet and wireless networking, the physical boundaries of computer networks are diminishing, allowing CSE tools to be supported far further than just the local network.

This section presents many of the key distributed systems papers and specifications. For the CSE researcher, the references here provide overviews, descriptions and tutorials of the major distributed systems architectures and protocols. By becoming familiar with these articles, the CSE researcher should have enough understanding of distributed systems technologies to understand the challenges and goals of this area of research, and to make the correct architectural selection for CSE tool development.

Publications

[1] David Chappell. *Understanding .NET*. Independent Technology Guides. Addison Wesley, 1st edition, May 2002.

Annotation: There are now hundreds of books and articles on Microsoft's .net architecture. The book given here presents an easy to read lightweight tutorial on the fundamentals of the architecture along with many useful references for more in-depth reading. The .net framework is a collection and refinement of the previous generation of tools and technologies for the Windows platform of operating systems. Whilst impossible to describe in a single paragraph, it can be viewed as a way of allowing programs written in any language to communicate, even in a distributed setting.

Any distributed application development on the Windows platform will now require the use of the .net framework. Through web services, soap, xml, a common language runtime model and network centric APIs it is possible to rapidly develop complex distributed applications such as CSE tools, and this book is an excellent starting point to introduce and explain all the related technologies and languages.

[2] Jim Keogh. *J2EE: The Complete Reference*. McGraw Hill/Osborne, California, USA, 1st edition, 2002.

Annotation: Java 2 Enterprise Edition (J2EE) can be viewed as Sun's alternative to Microsoft's .net framework, although it can be shown historically that J2EE has been around for longer than .net. J2EE contains a vast range and number of Java technologies to support complex commercial applications and services. Such enterprise technologies include web services, databases, interconnectivity facilities, xml and servlets. The book listed here presents each of these technologies in turn, giving comprehensive explanations and tutorials.

CSE tools will require the use of distributed systems technology, and the services of J2EE are attractive. Java 2 Standard Edition (J2SE) has many built-in features that would be useful to CSE tool developers: platform independence, vendor support, type reflection and network-centric language features. With J2EE, developers also have the support of database systems, web services and interconnectivity APIs. Accordingly, this book is well worth reading for the CSE researcher who wants to know the design possibilities and implementation details of complex distributed applications.

[3] Tim Bray, Jean Paoli, C M Sperberg-McQueen, Eve Maler, and Francois Yergeau. Extensible Markup Language 1.0 Third Edition. Technical report, W3C Consortium, February 2004. URL `http://www.w3.org/TR/REC-xml`.

Annotation: In terms of distributed systems, XML is becoming a leading technology for vendor-independent transfer of application data between processes. The reference provided here is the official XML specification recommendation of the World Wide Web Consortium (W3C). The main purpose of this document is to provide technical details for vendors that implement XML-compliant technologies, however, many other useful sections are provided giving the background of XML, usage examples, and references to many other useful articles for further reading.

There are many books that give detailed tutorials on the use of XML, in fact, most of the books given in this section's listing have chapters on XML. For the CSE researcher, however, the article presented here is an excellent starting point for learning about the structure and purpose of XML—a technology core to many modern distributed systems.

[4] Nilo Mitra. SOAP Version 1.2 Part 0: Primer. Technical report, W3C Consortium, June 2003. URL `http://www.w3.org/TR/soap12-part0`.

Annotation: This article presents a simple overview of the Simple Object Access Protocol (SOAP). SOAP is employed as a lightweight protocol for exchanging well-defined information in a distributed environment, and is a key driver for the success of web services. Web services are components that provide client applications with programmatic access to databases and other resources, which effectively replaces server-side web page access to remote servers with more intelligent alternatives.

As described in this article, SOAP messages, which provides the transportation backbone for web service communication, are defined entirely in XML. Key to the success of SOAP in providing a globally-adopted distributed messaging framework is its vendor, technology, and language neutrality.

SOAP provides a way to communicate between distributed applications in a manner that is simpler than other interprocess communication methods such as IIOP and DCom. Additionally, it has gained support from main industry cooperations such as Microsoft, IBM and Sun Microsystems. SOAP was designed to operate over HTTP, which gives it the benefit of not requiring special firewall configurations. For the CSE researcher, an understanding of web services, and more specifically, SOAP, is essential if tools are to be implemented that are compatible with J2EE or the .net framework. Accordingly, the primer listed here is a suitable starting point for code examples and tutorials related to SOAP and web services.

[5] Emerald Chung, Yennun Huang, Shalini Yajnik, Deron Liang, Joanne Shih, Chung-Yih Wang, and Yi-Min Wang. DCOM and CORBA Side by Side, Step By Step, and Layer by Layer. In *C++ Report*. Journal of Object-Oriented Programming, January 1998.

Annotation: This article presents a comprehensive overview of DCOM and CORBA. These technologies are now somewhat obsolete in terms of distributed computing, but it is valuable for the CSE researcher to have a basic understanding of their architectural structure. A basic knowledge of object request broker systems such as DCOM and CORBA is also very useful in helping understand the design and motivation for modern frameworks such as J2EE and .net.

This article does not present performance statistics, rather is represents a comparison of the two frameworks. It discusses three aspects of the frameworks: the basic programming architecture, the remoting architecture and the wire protocol architecture. To make a comparison, the article presents a simple application, and illustrates how it is implemented in both architectures. This article provides excellent references to the applicable specifications from within the case studies, and also includes other useful reference articles as well.

[6] Bobby Krupxzak, Kenneth L. Calvert, and Mostafa H. Ammar. Implementing Communication Protocols in Java. In *Communications Magazine*, pages 93–98. IEEE, October 1998.

Annotation: This paper discusses the performance and implementation details of network protocols implemented in the Java programming language. Java is a network centric programming language, and this paper provides useful insight related to the customisation of networking for distributed tools, which is an important topic for CSE researchers.

An excellent discussion is provided on how to customize existing network protocols and implement custom protocols when the conventional ones are not suitable. This article shows that tool implementors have a large degree of flexibility in implementing customised networking capabilities, which is useful reading for CSE researchers considering the design of tools with custom networking demands. The paper also is one of the very few to incorporate a discussion of network security issues and best practices.

[7] Java(TM) Message Service Specification Final Release 1.1, Sun Microsystems,

March 2002. URL `http://java.sun.com/products/jms/docs.html`.

Annotation: The Java Messaging System (JMS) represents the low-level distributed communication API of the J2EE framework. The document presented here gives the definitive description of JMS and its API. JMS appears to be a reimplementation of the Java Shared Data Toolkit (see section 3.2.2), but with major enhancements to support both push-subscribe asynchronous messaging and point-to-point synchronous communication.

For the CSE developer this messaging framework is very useful; it provides mechanisms to implement both central server and tool-to-tool communication. The document presented here gives several coding examples, API details and architectural descriptions. This is not, however, a tutorial document. It is instead a good starting point for learning the basic concepts of the JMS architecture and also servers as a definitive specification document.

[8] RMI Architecture and Functional Specification, Sun Microsystems, 2002. URL `ftp://ftp.java.sun.com/docs/j2se1.4/rmi-spec-1.4.pdf`.

Annotation: This document introduces Java's Remote Method Invocation (RMI) framework. RMI allows Java applications to call methods in remote processes, and any type of return value can be marshalled back to the calling application. RMI allows the development of fast, complex networking functions between two or more participating applications, which makes it candidate technology for many CSE tools. RMI is however a complicated framework, and requires considerable coding and setup effort. It is also normally restricted by firewalls unless it runs in HTTP proxy mode.

For distributed client/server applications, SOAP is fast becoming the number one technology. If customized application-to-application data transmission and method invocation is required however, then RMI is still an effective mechanism. The article listed here provides a complete overview of RMI, coding examples, and installation and execution instructions. It also discusses the wire protocol and garbage collection mechanisms for those who require specialised information on the RMI architecture.

[9] *JINI Technology Architectural Overview*, Sun Microsystems, January 1999. URL `www.sun.com/jini/whitepapers/architecture.html`.

Annotation: The JINI framework is an architecture that allows network protocols and services to be developed, published and used in an arbitrary manner. It is a complicated, expansive architecture, but essentially it provides users and applications with access to networked devices through Java APIs. Additionally, it allows custom code to be written and installed automatically within a distributed system, modifying the network or networked services. Key to the framework is the concept of service discovery, where network devices are made available for sharing upon detection.

This paper talks about the goals, infrastructure and discovery mechanisms of JINI. Simple example usage scenarios and links to other useful references are provided. JINI is a valuable mechanism for the CSE researcher; it may well be a very suitable technology for supporting complex network functions and services in a very easy manner. This document provides a good starting point for those wishing to learn about the JINI framework; it provides a concise overview of the general concepts, making good sense of a very complicated and expansive technology.

[10] Carl Cook and Neville Churcher. A Pure-Java Group Communication Framework. Technical Report TR-COSC 02/03, Department of Computer Science and Software Engineering, University of Canterbury, Christchurch, New Zealand, July 2003.

Annotation: This article describes a Pure-Java simplification of the Java Shared Data Toolkit. Called caise.messaging, this framework provides simple asynchronous communication capabilities between a group of distributed processes with an event-based delivery method. The paper listed here describes caise.messaging's design and implementation, and provides a user manual with code examples.

There are many similar systems to

caise.messaging such as JGroups and Tribe. For CSE researchers, however, the caise.messaging system provides a simple way for applications to join a meeting and send and receive data, without the need for RMI, XML, or any system configuration. The caise.messaging system has been used as the distributed systems component for several applications, and gives CSE researchers insight into how customised group communication can be implemented.

3.6 Software Engineering Metrics and Visualisation

The field of software metrics and visualisation concentrates on the analysis and extraction of useful metrics from software projects. Findings are then presented back to engineers in a way that is useful and minimises information overloading. For CSE research, the field of software metrics and visualisation is immediately interesting: the functions of metrics gathering and reporting are a core function of most CSE tools. Additionally, many CSE tools now employ visualisations are their main mode of feedback, or provide visualisations as supplementary information.

Software metrics and visualisation is useful to software engineers as a independent area of research. As software engineering becomes more collaborative, however, visualisations may become even more important to the developers of CSE tools. The additional dimension of multiple users and their interactions with artifacts over time provides richer information to the software team, but the correct modes of visualisation for CSE tools are yet to be fully explored.

The field of software metrics and visualisation may well be worth further investigation for CSE researchers; metric generation by itself can be very useful to support the monitoring of user activity and team progress. The papers listed in this section provide basic reading for the CSE researcher; for the interested reader, the papers provided here also give references to other relevant metrics papers.

Publications

[1] Thomas Ball and Stephen G. Eick. Software Visualization in the Large. *IEEE Computer*, 29(4):33–43, 1996.

Annotation: This paper gives a comprehensive motivation section on software visualisation for improving program comprehension. While this paper is now dated, it gives the CSE researcher a good background on the goals of software visualisation, and presents some typical visualisation tasks and outputs.

The paper presents four visualation representations: line, pixel, file summary and hierarchical. Through five case studies, new visualisation techniques are shown to address program comprehension. Techniques include code age listings, program comparisons, highlighting of deep nesting levels and runtime code coverage graphs. Key to all visualisations is the concept of *focus + context*, where a global overview is provided with local detail. The paper concludes with an interesting discussion on the problems facing software visualisation researchers, such as information overloading and graph layout processing times.

This paper gives the CSE researcher many references to key articles on related work, examples of prototype visualisations, and provides insight for potential CSE visualisations.

[2] Neville Churcher, Lachlan Keown, and Warwick Irwin. Virtual Worlds for Software Visualisation. In A. Quigley, editor, *Software Visualisation Workshop (SoftVis99)*, pages 9–16. 1999.

Annotation: This paper presents visualisation of software engineering data and introduces a framework to produce such visualisations for Virtual Reality Markup Language (VRML) viewers. The paper discusses the software visualisation issues of information overload, highly skewed distributions of data, and the importance of preserving outliers within large data sets. The paper then discusses the use of VRML to represent such datasets in some novel ways such as 3D treemaps, contrasting inheritance trees of different modelling techniques, and visualisations where large distributions of data are displayed in immersive worlds.

This paper is simple to comprehend yet gives considerable insight into the complexities related to the visualisation of real-world software engineering data. A framework is presented to transform and visualised such

data in VRML, providing insight into possible CSE visualisations. This paper also discusses the problems associated with generating and displaying software engineering data, and such problems are applicable to CSE researchers as well.

[3] M. Gogolla, O. Radfelder, and M. Richters. Towards Three-Dimensional Representation and Animation of UML Diagrams. In *UML 99—Beyond the Standard*, pages 489–502, 1999.

Annotation: This paper shows a variety of Unified Modeling Language (UML) diagrams represented in three-dimensional visualisations. As a point of inspiration for possible CSE visualisations, this paper makes good reading. The paper also introduces the possibility of using animations to display UML diagrams such as sequence diagrams. Even from a static, paper representation, the use of animations appear intuitive as a mechanism to illustrate diagrams that contain the elements of time and sequence.

The use of animation and 3D visualisations to show the affects of change over time is well suited for application within the field of CSE. This paper provides a good overview of 3D representations of diagrams and the potential for animated sequences to assist information display, and presents a range of possibilities for further study.

[4] Warwick Irwin and Neville Churcher. Object Oriented Metrics: Precision Tools and Configurable Visualisations. In *9th International Software Metrics Symposium*. Sydney, Australia, September 2003.

Annotation: This paper addresses the importance of extracting the correct metrics from programs through rigorous semantic analysis of source code. The paper demonstrates the difficulties and inconsistencies in extracting metrics from source files using standard compiler generation techniques, and presents a framework for obtaining metrics from standard grammars free from manual manipulation. The paper discusses a transparent pipeline which uses a standard grammar and source files as the input, which then leads to XML-based parse trees and a full semantic model of the project. The paper concludes by presenting novel VRML visualisations resulting from custom rendering of a large application's semantic model.

This paper leads an interesting discussion into the difficulties of extracting correct information from source files based on their language's formal specification. For the CSE researcher, this is an interesting aspect, as correct and full information is required to support non-trivial CSE tools.

[5] Hideki Koike and Hui-Chu Chu. How Does 3-D Visualization Work in Software Engineering?: Empirical Study of a 3-D Version/Module Visualization System. In *ICSE '98: Proceedings of the 20th International Conference on Software Engineering*, pages 516–519. IEEE Computer Society, Washington, DC, USA, 1998. ISBN 0-8186-8368-6.

Annotation: This short paper presents a tool that visualises an RCS repository and allows users to execute RCS commands via the interface. Code repositories typically have numerous artifacts, branches and dependencies, which makes them and ideal candidate for visualation. This paper presents a simple VRML visualisation of RCS repositories and details an evaluation of the task completion rates when subjects used the visual interface compared to the standard RCS command line. Rather unsurprisingly, the subjects task completion rates were considerably lower when using the GUI-based system rather than the standard command line interface.

The evaluation presented in this paper is perhaps not highly significant to the progress of software engineering, but the paper is useful for CSE researchers in that it clearly shows the potential of software engineering visualisations and visual interfaces.

[6] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of Test Information to Assist Fault Localization. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 467–477. ACM Press, New York, NY, USA, 2002. ISBN 1-58113-472-X.

Annotation: This paper presents a framework that analyses unit tests of source code, and visualises statements in the code that are highly correlated with failed executions. After demonstrating the way that the erroneous code correlations are generated, the paper presents a system called

Tarantula, which displays a complete source code listing shaded according to areas of interest. Evaluation of the system showed that the correlation technique employed was highly effective for identifying erroneous statements.

This paper provides an excellent example of visualising local detail within a global context. Developers of CSE systems can learn from this paper in terms of information layout and effective use of colour to assist in visualisations of collaborative software engineering information.

[7] Andrian Marcus, Louis Feng, and Jonathan I. Maletic. Comprehension of Software Analysis Data Using 3D Visualization, May 2003.

Annotation: To assist comprehension of large-scale programs, this paper introduces a tool called source viewer 3D (sv3D). Sv3D extends many previous software visualisation and program comprehension tools by providing three dimensional visualisations of the same data. An interesting observation made in this paper is that while large distributions of data are easier to visualise in a three dimensional representation, the problem of occlusion is significant. To alleviate this issue, techniques for elevation and transparency are introduced, minimising the risk of obscuring distant data points.

This paper presents effective visualisation techniques for complex programs in a three dimensional space. It provides insight into the task of rendering real-world programs to a displayable form, and the problems that three dimensional visualisations can bring. For the CSE researcher, this is another paper that provides valuable guidance related to processing and visualising large volumes of complex software engineering data.

4 Concluding Remarks

The field of CSE is rapidly expanding. The demand for computer-supported collaborative software engineering tools steadily grows as the fields of interprocess communication, groupware and configuration management continue to mature. This, coupled with recent technological advances in IDEs and programming languages gives the CSE researcher plenty of scope for significant advances within his or her area of interest.

CSE tools may take many forms and functions, and there is a lot of freedom and possible alternatives for the developers of such tools. A basic understanding of all the fields that constitute CSE is important however, and that was the motivation for compiling the bibliography presented here. The listings provided in this bibliography are intended to provide the CSE researcher with a solid understanding of the most recent advances within the field, and a brief summary of key papers in all fields related to CSE.

4.1 Future Directions for CSE

As the related fields of research progress, so too will CSE. At present, the technology available to CSE researchers enables the construction of many new and powerful tools. Research never stops, however, and the following points make suggestions in the related fields for the further advancement of CSE tools and research.

Configuration Management Code repositories could become synchronous, possibly automatically attempting to merge new commits to the main build with individual user's working copies of files. Built-in collaborative code editors could also be supplied with code repository systems for peer programmers and others who do not wish to work on separate copies of project files.

CSCW and Groupware Groupware toolkits have matured to be useful in minor applications, or as a compliment to major applications. There is, however, a need to address wide area network latencies as networking delays are often inevitable. Additionally, users are already are becoming more distributed by moving out of local area networks and into more error-prone internet-based virtual private networks.

Software Engineering Processes A question yet to be asked within Software Engineering is that of how the development process will change given realtime CSE tools. At present, we only know software engineering methodologies that work around the mechanisms of code repositories, but perhaps more effective ways of development are possible as tools become more collaborative. Similarly, further research into the patterns of collaboration would be useful to fully expose the behaviour within current software engineering teams.

Software Visualisations and Metrics There are many possibilities for further research into software metrics and visualisation as it relates to CSE. With the dimension of individual user activity over time, and ability of realtime CSE tools to capture programmer activity to a high degree of accuracy and fine level of granularity, the potential for new metrics and types of visualisations and animations is unbounded. Virtual and augmented reality visualisations for CSE data are still only in the early stages of investigation.

Human Factors Further human factors research into CSE could have a major impact on the future directions of CSE tools. Many current CSE tools offer large volumes of feedback, both immediate and periodically, yet little research has addressed how useful feedback information is, or the correct ways to deliver such information to users.

Distributed Systems Much research is going into distributed systems at present. Given frameworks such as .Net and J2EE, it is now possible to access objects and invoke methods on remote machines as if they were within the same process. There is still a high degree of programmer overhead in doing so however, and for CSE tools, completely invisible boundaries between remote processes would reduce the programming effort considerably. At present, for example, it is difficult to add a listener for an event in a remote process—to do so requires a substantial coding effort. Eventually, research may allow a group of separate processes to share data and invoke methods natively.

Author Index