Software Knowledge Capture and Acquisition: Tool Support for Agile Settings

Filipe Figueiredo Correia
ParadigmaXis
Avenida da Boavista, 1043
4100-129 Porto, Portugal
Faculdade de Engenharia,
Universidade do Porto
Rua Dr. Roberto Frias, s/n
4200-465 Porto, Portugal
filipe.correia@fe.up.pt

Ademar Aguiar
INESC Porto,
Departamento de Engenharia Informática,
Faculdade de Engenharia,
Universidade do Porto
Rua Dr. Roberto Frias, s/n
4200-465 Porto, Portugal
ademar.aguiar@fe.up.pt

***Abstract*—Knowledge plays a key role in software development, and the effectiveness of how it is captured into artifacts, and acquired by other team members, is of crucial importance to a project's success. The life-cycle of knowledge in software development is derived from the adopted artifacts, practices and tools. These axes are here reviewed from a knowledge capture and acquisition perspective, and several open research issues are identified.**

The present work is being carried out in the context of the author's doctoral research. The research objectives are derived from the presented open issues, and a research strategy is outlined. Some preliminary results are also presented.

***Index Terms*—Software Engineering Tools and Methods**

I. Introduction

During software development activities, developers resource to mental models of the program that they are building, by inspecting project artifacts, by direct contact with other developers, and by using knowledge gained from previous experiences. This is a costly process, and the mental models that exist at a given moment may be easily forgotten as time passes, and the developers' attention is diverted to other tasks and program locations. If this knowledge is not captured and shared, it may be lost, and thus have to be reacquired the next time it is needed, with the overhead that entails.

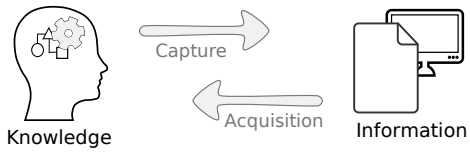

Fig. 1. Knowledge Capture and Acquisition

Although the topics of knowledge capture and acquisition have been addressed before, we will start by clarifying some concepts for the purpose of this document, as they have evolved over the years, and are not always consensual [1]. **Knowledge Capture**, or *representation*, is seen as the process of recording knowledge in a medium, and, doing so, transforming and encoding it as **information**. On the other hand, **Knowledge Acquisition** is the process through which a human actor gains knowledge, that is, the process of learning and understanding information. Figure 1 depicts this duality.

Research in the cognitive science domain has dealt with several knowledge-related issues. It has allowed to identify different types of knowledge and strategies through which they are acquired, with results useful on different domains of human activity, including that of software development [2].

Developing software is a knowledge-intensive activity, heavily relying on the creation and evolution of knowledge. The difficult part of the process of developing software is not the production of programming language statements *per se*, but the discovery of the knowledge that will allow the developers to build it. This is essentially an activity of learning, or acquiring knowledge [3], in that **artifacts**, in which knowledge manifests itself, go through different levels of formality: from unstructured information, as textual documents and verbal communication, to instructions to be executed by computers.

Software processes appear as a means of achieving increased quality and productivity on the development of software, defining a set of rules and trying to make the course of events more predictable and repeatable.

The best process for a given software project is very dependent on it's specific context. Although there are a number of pre-established development processes, they may be seen as starting points, or sets of practices that are known to work well together. When put into use, processes are adjusted to fit the particular context at hand, or built up from a minimal set of core practices, that grows according to the project's needs [4]. As part of a knowledge-intensive activity, these **practices** are ways of directly supporting the capture and the acquisition of knowledge.

Software development relies on the use of **tools** to support both the handling of the different kinds of artifacts, and the

carrying out of development practices. They are a means of capturing knowledge, and of manipulating the information derived therefrom.

The purpose of this section was to introduce the topics of knowledge capture and acquisition in software development, and Section II will further review them and present current approaches. Section III describes the main objectives of the proposed research, and how these objectives are to be reached. Some preliminary work is then presented in Section IV, and the envisioned work plan is described in Section V. Some concluding remarks are made in Section VI.

II. State-of-the-Art

Knowledge capture and acquisition are not trivial tasks, and several tools have been devised to help developers in this regard [5]. In practice, they are aimed mostly at supporting development artifacts and practices, but how to make these tools efficient from a knowledge standpoint is not always clear. In the remaining of this section we will present some of software development's artifacts, practices, and tools, from a knowledge capture and acquisition perspective.

A. Software artifacts

A software system's knowledge is important in different ways, and to different people, depending on which knowledge is recorded and how is this knowledge organized. The maintenance of software artifacts' understandability may be achieved by complementing or making more accessible the information enclosed on those artifacts, which may be of different kinds, including source code, models, graphic-design resources, a working product, etc. They are all by-products of the software development process, but don't always carry by themselves all the necessary information in order to be easily understood and reused.

1) Documentation: Software documentation plays an important role in the acquisition of knowledge. It is an effective way to capture unstructured knowledge, to share it between team members and to preserve it for future use. Despite this, it is still common practice to see the creation of documentation as a low priority activity, or even one that is not worthwhile to carry out, due to the effort its creation and maintenance may imply.

Part of the complexity of producing software documentation comes from the fact that several forces are at stake when choosing which information to record [6], [7], [8].

Different writers and audiences. Documentation is produced and used by participants with different roles in the software development process, each possessing different levels of knowledge, and having different knowledge needs.

Different subjects. Documentation is commonly made of textual descriptions but usually also addresses other artifacts, further describing them or using them to support more elaborate descriptions. A given subject may be covered along different facets, and along different levels of abstraction.

Different notations. Different kinds of information are better communicated with different notations. Some kinds of information, a textual description may be the most appropriated, but there are also those better conveyed by using diagrams, or source code.

Different forms. According to the specific context at hands, one can conceive and structure a document in a way that it's most effective. There are recurring types of document structures, that address typical documentation structuring needs: scenarios, design patterns and pattern languages, system overviews, user manuals, tutorials, contextual help, frequently asked questions, cookbooks, recipes, hooks and motifs, among others.

Different alternatives have been considered over the years to support software documentation. Approaches like Literate Programming, Elucidative Programming, Code Annotations and wikis, although not in universal use, are some of the most effective. Some tools to support these approaches will be presented on Section II-C.

2) Models: The creation of abstractions is a recurring solution in the conception of software, as they allow developers to focus on the design of software being built, instead of the implementation details.

Model Driven Engineering (MDE) goes beyond the use of models as a way to abstract technology-related issues, and sees them as a way of abstracting business domains [9]. MDE approaches provide several benefits, including increased reuse, fewer bugs, shorter time-to-market and systems that are simpler to understand [10].

However, the creation and consumption of models brings with it several challenges [11].

Level of abstraction. Several modeling languages allow the creation of models, among which the Unified Modeling Language (UML) is one of the most successful. It allows to model software systems from different perspectives, according to a defined meta-model. However, it is sometimes needed to resort to other types of models, that are more specific to a given domain. Finding the appropriate level of abstraction may not be easy, if it is to accommodate all the required information for that domain.

Consistency. Some approaches see the use of models and other artifacts as independent, while other approaches take further advantage of models by generating other artifacts from them, or interpreting them at runtime [12]. The later ones have the advantage of avoiding the maintenance of multiple artifacts that record same knowledge, or have parts is common, both easing the processes of recording and understanding those artifacts.

Skills. Finding people with the right set of skills may be difficult, as it requires knowledge in domain analysis, metamodelling, generator/interpreter development and architecture. Such profile may be more difficult to find, and their knowledge is sometimes difficult to pass on to other team members.

3) Source code: Code is usually seen as the primary software development artifact. Even though other approaches tend to focus on alternative kinds of artifact as the primary ones (MDE focuses on models, LP focuses on documentation, etc), source code is still needed to instruct machines what to do. In fact, some argue that source code is the only artifact that one can depend on, as it *doesn't lie* — if inconsistent with other artifacts, source code is the artifact to go to for the actual program behaviour.

Despite this, the knowledge that gave origin to the source code is very weakly recorded within this form. It exists only implicitly, rather than explicitly, being therefore difficult, if even possible, for the developers to reconstruct the original mental model at a later time from source code alone.

B. Practices

Agile processes, like *Extreme Programming* [13], *Scrum* [14] and *Crystal Clear*, have emerged from the need for *lighter* methodologies in the development of software, as opposed to traditional *heavyweight* ones. In this context, *agility* refers to the adaptiveness towards change, and to the incremental development of value, which is part of the essence of these processes.

The set of practices that make up a process vary. They define behaviors to be taken by team members, having in mind an intended outcome, in what concerns factors like project planning, design, coding, testing and team work. These behaviors are ultimately about manipulating artifacts and communicating information between team members — which can be reduced to a set of knowledge-related activities, including it's capture and acquisition. The three practices bellow are of common use in agile processes, and are here detailed as to exemplify how the capture and acquisition of knowledge may be related to the response to change and to the set of adopted practices.

Refactoring. Technique that consists on the improvement of a system's internal structure without affecting it's external behavior. It plays a part in improving knowledge acquisition, by encouraging developers to capture knowledge in a simpler and clearer way.

Pair Programming. This practice is about two programmers working together, with the use of only one computer, with one keyboard. While one of them types in the code, the other continuously reviews it and considers the strategic direction of the work. Pair programming addresses the issue that captured knowledge never reflects the original mental-model accurately, and that it may quickly become outdated. It addresses it by *avoiding* knowledge capture, and relying on a stronger communication between the two team members that are pairing.

Test Driven Development. Technique in which unit tests are developed prior to developing the code that will be tested, using very short development iterations. It induces the programmer to think in the implementation more thoroughly. The captured knowledge, as both the implementation and the tests, will thus better convey the desired behavior for the system.

C. Tools

Different approaches and tool sets to software documentation are presented below.

Literate Programing. The ultimate objective of Literate Programming (LP) is to make computer programs that are comprehensible by human beings [15], by switching the focus traditionally given to source code artifacts, to documentation artifacts. The fundamental idea behind LP is that, when writing programs, one should not instruct a computer what to do, but rather explain to human beings what the computer will do.

This approach is supported by a set of tools that allow the production of literate programs, by describing pieces of the program at the same time they are developed, and connecting them as a *web* of related ideas. The result is a unified document, combining several fragments (*chunks*) of source code and documentation, disposed not as a set of assorted blocks of information, but following a line of reasoning. Contents are arranged in the order in which they are written, so that it may be better read and understood [16], [17]. This is an advantage towards readability, when comparing to the approach of having source code organized according to its own structure.

The original LP approach as evolved since it's conception, and several variants have since appeared. Namely, Reverse Literate Programming [18], Literate Modeling [19], Theme-based Literate Programming [17] and Elucidative Programing [20] are some of the most proeminent.

Code annotations. This technique is inspired in LP, in that documentation is obtained from an unified representation of textual descriptions and source code. However, it is also fundamentally different from LP, as textual descriptions exist in the form of source code comments. This means that the unified representation of textual descriptions and source code is itself valid and compilable source code, avoiding an additional step of extracting the source code for compilation. When comparing to LP, it is also important to highlight that writing code annotations is not a document-oriented way of creating documentation, but rather a source code oriented one. As such, it misses one of the main benefits of LP, which is the possibility of reordering documentation according to an intended *psychological arrangement*.

Code annotations are primarily used for creating API documentation, and don't address all the issues that LP tries to address. Having said this, it has shown to be quite successful in this niche, and has helped increasing the awareness on the need for documentation, and showed how can it enhance knowledge acquisition. The widespread use of this approach as been much the merit of Javadoc [21], a tool supporting this functionality for the Java language, but similar tools have since appeared for different languages and environments.

Wikis. Although our interest is on the use of wikis in

the context of software development, they have a wide scope of application. They can be generically defined as web-based systems aimed at the collaborative writing of documents.

Some wiki engines have specifically been conceived for the domain of software development, and provide additional features for the integration of software artifacts. This is done, not by copying them to the wiki environment, but by linking or inlining the appropriate artifact [22], [23]. Unlike wiki-links between wiki pages, these references carry a special semantics, according to the kind of artifact being linked (code fragment, UML model, issue-tracking record, etc).

These previous experiences have shown wikis to possess characteristics that grant several benefits in the area of software documentation [22], [24], [25]. They show that wikis allow an eased access to both technical and not technical people; promote participation of the entire project team in the documentation process; improve overall team communication; allow integration of heterogeneous types of content while maintaining the information structured; and may be extended to allow integration in IDEs and in the entire development process.

Knowledge captured as models is richer in structure than knowledge captured as documentation, which makes the requirements for modeling tools more demanding. Models have also more diverse goals — for example, they may be used as documentation, but it may make sense also to use them as executable artifacts — requiring modeling tools to be integrated with source code creation tools, as well as with document creation tools.

Platforms such as the Eclipse Modeling Framework (EMF) and Microsoft Oslo, provide visual tools, domain specific languages and model transformation mechanisms, from within integrated development environments.

As to source code, there are many tools supporting its creation. Integrated Development Environments (IDEs), such as Eclipse and Visual Studio provide this capability, while also allowing additional mechanisms. Among others, they provide mechanisms for code navigation and visualization [26], [27], as well as a quicker expression of the user's intents with features like Code-completion and Refactoring.

III. Research Objectives and Approach

Existing approaches to knowledge capture and acquisition in the context of software development still leave space open for improvement. This research will focus on addressing the following identified issues.

Collaboration. Software systems are usually products of the work of several people. The collaboration towards the same objective brings the need for constant communication and coordination among team members, which is not easy to fulfil in an efficient way, specially as teams increase in size, and become geographically distributed.

There are several tools, following distinct approaches, that already help making collaborative development tasks more efficient [28], [24], [26], [29], allowing several players to contribute to the creation of a given artifact. However, the social interactions that don't directly translate into the creation of an artifact are usually left out of the scope of these tools. By further supporting the interactions between the different actors, subsequent knowledge capture and acquisition may be substantially improved.

Consistency. Whether on an initial conception phase or on a maintenance phase, most software systems evolve constantly, which means that several existing artifacts will also be changing. Keeping them in-sync is a difficult issue, especially when unstructured information is involved, such as textual documentation. The value of the existing artifacts, from a knowledge acquisition standpoint, directly depends on their ability to convey accurate information.

Contents Integration. Integrating contents means that instead of dealing with heterogeneous artifacts independently, the implicit relations that exist between them are made explicit. Current approaches store these relations and allow to navigate them, or use *verisimilitude* [30], which consists on having several artifacts physically close to one another [8], [31], [24].

These mechanisms support both knowledge capture and acquisition to some degree, but still require a considerable effort to be made by the information authors and consumers.

Reuse. Good productivity gains can be achieved by reusing source code artifacts, namely by employing components, frameworks, libraries, and other techniques, but reuse may play an equally important role with other artifacts too [32]. The reuse techniques used for source code artifacts are not always directly applicable, but some concepts from the object-oriented paradigm, such as inheritance and information hiding, have shown to be of some use concerning documentation [33], [34]. However, it is still not clear how to support such concepts in modern environments, without over-complexifying tools and placing additional barriers to effective knowledge capture.

Environments. Modern IDEs reflect some good and commonly used practices, namely, they are integrated with test frameworks, refactoring tools, bug-tracking systems, and version control systems, among others. This integration of tools under the same environment is a step closer of seamless supporting an entire software development process. Among the advantages of an integrated environment, one can find:

- Developers are subjected to less *mode-switching*, as they will be required to use a lower number of tools, which are made available in a consistent way.
- It becomes easier to switch between underlying tools with similar objectives, as it exists a common abstraction over similar operations. An example of this

is the support given by IDEs to different programming languages, or the support given to different source code versioning systems.
  - By using an integrated environment some tasks can be made less repetitive, as a simple command, issued by the user, can transparently automate several underlying tools.

  Although desktop-based IDEs are still the environments of choice of most developers, following web-based approaches is becoming increasingly popular with some types of tools. Namely, issue-tracking systems and documentation systems, have prospered as web-based applications. Although IDEs also frequently provide support for these tools, not all are easily integrable on both desktop and web environments [28], fragmenting the development environment.

  The trend seems to be towards the web and the use of rich user interfaces, and web-based IDEs may even come to fully replace desktop-based ones. However, current solutions are required to take into account the coexistence of these distinct realities. This poses an interesting challenge from a knowledge capture and acquisition perspective, as information becomes scattered and difficult to combine, cross-reference and made consistent.

According to these issues, the following research questions may be outlined:

- How may knowledge capture be improved?
  - How to identify the knowledge that is not captured but should be?
  - How to know the appropriate level of detail to capture?
  - Which strategies can be used to capture knowledge in a non-intrusive way?
- How may knowledge acquisition be improved?
  - How to identify which recorded information would fulfill a certain knowledge gap?

It is the author's thesis statement that **the presented issues may be addressed by providing an approach to knowledge capture and acquisition, along with supporting tools, improving the quantity and quality of recorded knowledge and leveraging the support for software understanding**.

More specifically, the main objectives of this research will be to:

- Improve knowledge capture by leading developers to record structurally rich information.
- Given the continuous change information goes through, support its incremental maintenance and evolution, so that the value of captured knowledge may be kept.
- Improve knowledge acquisition by providing developers with the information they need, given an identified knowledge gap.

Several authors point out that the most adequate research strategy is one that combines several methods, according to the specific research at hand. Known research methods have been grouped into four general categories [35], [36]. As described by Zelkowitz and Wallace [35]:

> **Scientific methods.** *Scientists develop a theory to explain a phenomenon; they propose a hypothesis and then test alternative variations of the hypothesis. As they do so, they collect data to verify or refute the claims of the hypothesis.*
>
> **Engineering methods.** *Engineers develop and test a solution to a hypothesis. Based upon the results of the test, they improve the solution until it requires no further improvement.*
>
> **Empirical methods.** *A statistical method is proposed as a means to validate a given hypothesis. Unlike the scientific method, there may not be a formal model or theory describing the hypothesis. Data is collected to verify the hypothesis.*
>
> **Analytical methods.** *A formal theory is developed, and results derived from that theory can be compared with empirical observations.*

Different kinds of methods will coexist throughout the phases of this doctoral work, namely, **empirical methods** will be used to identify both knowledge capture and acquisition patterns, **scientific methods** will be applied for deriving theoretical models from these real-world observations, **engineering methods** used for developing concrete tools and **empirical methods** will be again applied in the final part of the research, to validate some parts of the achieved results.

IV. Current Work and Preliminary Results

Preliminary research has been done by the authors on the conception of Weaki: a wiki-based prototype solution to structuring software documentation in such way that is non-intrusive for authors, and leverages knowledge acquisition by allowing the increasing awareness of team members towards each other and the created information structure.

A new metric was devised to assess the compliance between a given documentation content and its associated structure, laying the grounds for a mechanism to support consistency maintenance.

Evolution is taken into account by supporting incremental formalization of contents, and there are direct advantages in terms of collaboration, by building upon the fundamental principles of wikis.

V. Work Plan and Implications

A further study of both existing tools, practices and artifacts will be conducted, in order to identify recurrent knowledge capture and acquisition patterns from existing approaches, as well as related benefits and liabilities. Theoretical models will be derived from these observations, and will be the base for the development of new tools.

Figure 2 depicts an overview of the research plan, with the envisioned timeframe.

Weaki is a first step in this direction, and will continue to evolve to accommodate additional functionality.

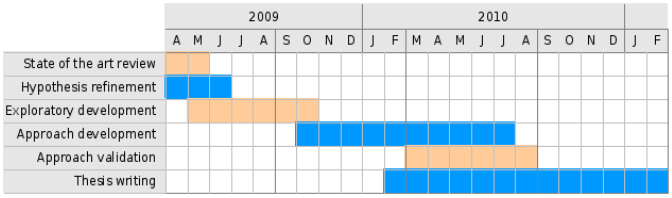

Fig. 2. Workplan overview

VI. Conclusions

Despite its importance, several difficulties exist in the process of recording knowledge and learning from knowledge that was previously recorded.

Some of the existing artifacts, practices and tools that play a part in this process were reviewed, and some open issues identified for research, namely: *Collaboration*, *Consistency*, *Integration of Contents*, *Reuse* and *Environments*. A thesis statement was also established, along with a set of research questions.

The presented open issues were taken into account in the creation of Weaki, but are still to be focus of further research and validation in real world scenarios.

We expect this research to provide a solid contribution both to the quality of the captured knowledge on a software project context, and the way this recorded knowledge is learned and used by project actors.

Acknowledgment

We would like to thank ParadigmaXis, S.A. for sponsoring this research.

References

[1] km forum, "KM forum discussion archives - knowledge vs information," http://www.km-forum.org/t000008.htm, 1996. [Online]. Available: http://www.km-forum.org/t000008.htm

[2] P. N. Robillard, "The role of knowledge in software development," *Commun. ACM*, vol. 42, no. 1, pp. 87–92, 1999.

[3] P. G. Armour, "The five orders of ignorance," *Commun. ACM*, vol. 43, no. 10, pp. 17–20, 2000.

[4] B. Boehm and R. Turner, "Observations on balancing discipline and agility," in *Agile Development Conference, 2003. ADC 2003. Proceedings of the*, 2003, pp. 32–39.

[5] M. Storey, F. D. Fracchia, and H. A. Mller, "Cognitive design elements to support the construction of a mental model during software exploration," *J. Syst. Softw.*, vol. 44, no. 3, pp. 171–185, 1999.

[6] A. Aguiar, "A minimalist approach to framework documentation," Ph.D. dissertation, Faculdade de Engenharia da Universidade do Porto, Sep. 2003.

[7] G. Butler, R. K. Keller, and H. Mili, "A framework for framework documentation," *ACM Comput. Surv.*, vol. 32, p. 15, 2000.

[8] J. Hartmann, S. Huang, and S. Tilley, "Documenting software systems with views ii: an integrated approach based on XML." Sante Fe, New Mexico, USA: ACM, 2001, pp. 237–246.

[9] D. Schmidt and D. Schmidt, "Guest editor's introduction: Model-Driven engineering," *Computer*, vol. 39, no. 2, pp. 25–31, 2006.

[10] D. Riehle, S. Fraleigh, D. Bucka-Lassen, and N. Omorogbe, "The architecture of a UML virtual machine," in *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*. Tampa Bay, FL, USA: ACM, 2001, pp. 327–341.

[11] T. Stahl and M. Voelter, *Model-Driven Software Development: Technology, Engineering, Management*, 1st ed. Wiley, May 2006.

[12] F. Correia and H. Ferreira, "Trends on adaptive object model research," in *Proceedings of the Doctoral Symposium on Informatics Engineering 2008*. Porto, Portugal: FEUP, 2008.

[13] K. Beck and C. Andres, *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2004.

[14] L. Rising and N. Janoff, "The scrum software development process for small teams," *IEEE Softw.*, vol. 17, no. 4, pp. 32, 26, Jul. 2000.

[15] D. E. Knuth, "Literate programming," *Comput. J.*, vol. 27, pp. 97–111, 1984.

[16] D. Knuth, "The WEB system of structured documentation," 1983.

[17] A. Kacofegitis and N. Churcher, "Theme-based literate programming," in *Software Engineering Conference, 2002. Ninth Asia-Pacific*, 2002, pp. 549–557.

[18] M. Knasmueller, "Reverse literate programming." Dundee: Johannes Kepler Universitat Linz, 1996.

[19] J. Arlow, W. Emmerich, and J. Quinn, "Literate modelling - capturing business knowledge with the UML," J. Bézivin and P.-A. Muller, Eds., vol. 1618. Springer, 1999, pp. 189–199.

[20] K. NØrmark, "Requirements for an elucidative programming environment," in *Program Comprehension, 2000. Proceedings. IWPC 2000. 8th International Workshop on*, 2000, pp. 119–128.

[21] L. Friendly, "The design of distributed hyperlinked programming documentation," in *IWHD'95*, Montpellier, France, 1995.

[22] A. Aguiar, G. David, and M. Padilha, "XSDoc: an extensible wiki-based infrastructure for framework documentation," in *Jornadas de Ingeniería del Software y Bases de Datos*, Alicante, Oct. 2003.

[23] Edgewall Software, "The trac project — integrated scm & project management." [Online]. Available: http://trac.edgewall.org/

[24] A. Aguiar and G. David, "WikiWiki weaving heterogeneous software artifacts." San Diego, California: ACM, 2005, pp. 67–74.

[25] D. Thomas, "Programming with models - modeling with code — the role of models in software development," *Journal of Object Technology*, vol. 5, pp. 15–19, Nov. 2006. [Online]. Available: http://www.jot.fm/issues/issue_2006_11/column2

[26] M.-A. Storey, L.-T. Cheng, I. Bull, and P. Rigby, "Shared waypoints and social tagging to support collaboration in software development." Banff, Alberta, Canada: ACM, 2006, pp. 195–198.

[27] M. D. Storey, D. ubrani, and D. M. German, "On the use of visualization to support awareness of human activities in software development: a survey and a framework," in *Proceedings of the 2005 ACM symposium on Software visualization*. St. Louis, Missouri: ACM, 2005, pp. 193–202.

[28] M.-A. Storey, "Theories, methods and tools in program comprehension: past, present and future," in *Program Comprehension, 2005. IWPC 2005. Proceedings. 13th International Workshop on*, 2005, pp. 181–191.

[29] W. Xiao, C. Chi, and M. Yang, "On-line collaborative software development via wiki." Montreal, Quebec, Canada: ACM, 2007, pp. 177–183.

[30] V. Pieterse, D. G. Kourie, and A. Boake, "A case for contemporary literate programming." Stellenbosch, Western Cape, South Africa: South African Institute for Computer Scientists and Information Technologists, 2004, pp. 2–9.

[31] K. M. Anderson, S. A. Sherba, and W. V. Lepthien, "Towards large-scale information integration." Orlando, Florida: ACM, 2002, pp. 524–534.

[32] K. Haramundanis and L. Rowland, "Experience paper: a content reuse documentation design experience." El Paso, Texas, USA: ACM, 2007, pp. 229–233.

[33] B. Childs and J. Sametinger, "Literate programming and documentation reuse," in *Software Reuse, 1996., Proceedings Fourth International Conference on*, 1996, pp. 205–214.

[34] J. Sametinger, "Object-oriented documentation," *SIGDOC Asterisk J. Comput. Doc.*, vol. 18, pp. 3–14, 1994.

[35] M. Zelkowitz and D. Wallace, "Experimental models for validating technology," *Computer*, vol. 31, pp. 23–31, 1998.

[36] W. F. Tichy, N. Habermann, and L. Prechelt, "Summary of the dagstuhl workshop on future directions in software engineering: February 17-21, 1992, schloß dagstuhl," *SIGSOFT Softw. Eng. Notes*, vol. 18, pp. 35–48, 1993.