Understanding Frameworks Collaboratively : Tool Requirements

Nuno Flores
Departamento de Engenharia Informática
Faculdade de Engenharia da Universidade do Porto
Porto, Portugal
e-mail: nuno.flores@fe.up.pt

Ademar Aguiar
INESC Porto, DEI
Faculdade de Engenharia da Universidade do Porto
Porto, Portugal
e-mail: ademar.aguiar@fe.up.pt

Abstract — **Software development is a social activity. Teams of developers join together to coordinate their efforts to produce software systems. This effort encompasses the development of a shared understanding surrounding multiple artifacts throughout the process. Frameworks are a powerful technique for large-scale reuse, but their complexity often makes them hard to understand and learn how to use. Developers resort to their colleagues for help and insight, at the expense of time and intrusion, as documentation is often outdated and incomplete. This paper presents a study on the state-of-the art on program comprehension, framework understanding and collaborative software environments, proposing a set of requirements for developing tools to improve the understanding of frameworks in a collaborative way.**

Keywords- Frameworks; Understanding; Collaborative; Tools; Requirements

I. INTRODUCTION

As software systems evolve in size and complexity, frameworks are becoming increasingly more important in many kinds of applications, in different technologies (object-orientation and recently aspect-orientation too), in new domains, and in different contexts: industry, academia, and single organizations.

Frameworks are a powerful technique for large-scale reuse that helps developers to improve quality and to reduce costs and time-to-market. However, before being able to reuse a framework effectively, developers have to invest considerable effort on understanding it. Especially for first time users, frameworks can become difficult to learn, mainly because its design is often very complex and hard to communicate, due to its abstractness, incompleteness, superfluous flexibility, and obscurity.

Understanding a piece of software is an important activity of software development, with a big social emphasis. Advances in global software development are leading to teams continuously becoming more and more distributed. A software development project requires people to collaborate. Trends toward distributed development, extensible IDEs, and social software influence makers of development tools to consider how to better assist the social aspects of development.

Learning how to use a framework deals with understanding its components, from its purpose and high-level architecture to its source code. Understanding framework means understanding software. The program comprehension community addresses this, in a broad sense. The social aspects of software development encompass the concerns of the collaborative software development research areas.

This paper outlines a set of requirements that should be tackled in order to develop tools to improve framework understanding using a collaborative approach.

Sections II to IV present a state-of-the art survey on the main domain areas dealt by this paper, namely Program Comprehension, Framework Understanding and Collaborative Software Development Environments. Section V points out open issues in those domains, converging to the key research questions in section VI. Section VII presents the solution approach and lists the set of requirements proposed by the authors. The paper concludes in section VIII.

These findings are part of an on-going research work [1].

II. PROGRAM COMPREHENSION

Program comprehension research can be characterized by both the theories that provide rich explanations about how programmers comprehend software as well as the tools that are used to assist in comprehension tasks.

Since the time of the first software engineering workshop [99], challenges in understanding programs became too familiar. As such, the field of program comprehension as a research discipline has evolved considerably. The main goal of the community is to build an understanding of these challenges, with the ultimate objective of developing more effective tools and methods that supports them [129].

This research has been rich and diversified, with various shifts in paradigms and research cultures during the last decades.

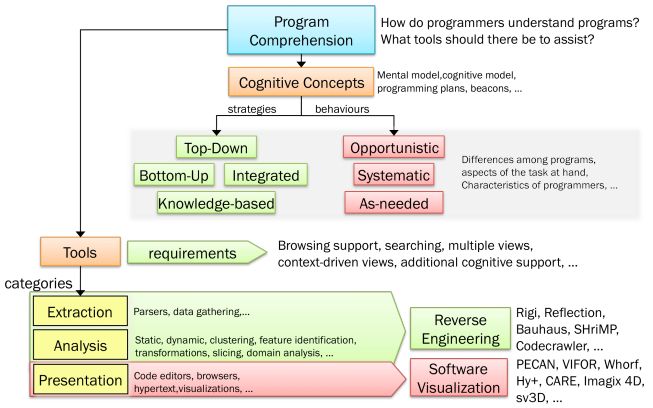

Figure 1 - Program Comprehension topics

A multitude of differences in program characteristics, programmer ability and software tasks have led to many diverse theories, research methods and tools.

Consequently, there is a wide variety of theories that provide rich explanations of how programmers understand programs and can provide advice on how program comprehension tools and methods may be improved.

In this section, an overview of existing comprehension theories, models and methods is presented, as an attempt to create a landscape of program comprehension research and possibly trends for future work directions. An overall depiction of the main topics can be seen in Figure 1.

A. Cognitive theories and models

At first, experiments were done without theoretical frameworks to guide the evaluations, and thus it was neither possible to understand nor to explain to others why one approach might be superior to other approaches [32].

As a lack of theories was being recognized as problematic, methods and theories were borrowed from other areas of research, such as text comprehension, problem solving and education. These theoretical underpinnings led to the development of cognitive theories about how programmers understand programs and ways of building supporting tools. These theories brought rich explanations of behaviors that would lead to more efficient processes and methods as well as improved education procedures [64].

1) Concepts

A **mental model** describes a developer's mental representation of the program to be understood whereas a **cognitive model** describes the cognitive processes and temporary information structures in the programmer's head that are used to form the mental model. **Cognitive support** assists cognitive tasks such as thinking or reasoning [145].

Programming plans are generic fragments of code that represent typical scenarios in programming. For example, a sorting program will contain a loop, which compares two numbers in each iteration, or visiting a structure will have a loop going through all its elements [128].

Beacons are recognizable, familiar features in the code that act as cues to the presence of certain structures [21]. **Rules of programming** discourse capture the conventions of programming, such as coding standards and algorithm implementations [128].

Then there are strategies and behaviors. Behaviors are ways of changing from one strategy to another.

2) Top-down comprehension strategy

Two main theories emerged that support a top-down comprehension strategy. Brooks [21] suggested that programmers understand a completed program in a top-down way where the comprehension process relies on reconstructing knowledge about the application domain and mapping that to the source code. The process starts with a hypothesis about the general nature of the program. This initial hypothesis is then refined in a hierarchical fashion by forming secondary hypothesis. These are then refined and evaluated in a depth-first manner, whose verification (or rejection) depends heavily on the absence or presence of beacons.

Soloway and Ehrlich [128] observed that top-down understanding is used when the code or type of code is familiar. They observed that expert programmers use beacons, programming plans and rules of programming discourse to decompose goals and plans into lower-level plans. They noted that delocalized plans complicate program comprehension.

3) Bottom-up comprehension strategy

The bottom-up theory of program comprehension proposed by Shneiderman and Mayer [119] assumes that programmers first read code statements and then mentally chunk or group these statements into higher levels abstractions. These abstractions (chunks) are aggregated further until a high-level understanding of the program is attained. The authors differentiate between syntactic and semantic knowledge of programs: syntactic knowledge is language dependent and concerns the statements and basic units in a program; semantic knowledge is language independent and is built in progressive layers until a mental model is formed, which describes the application domain.

Similarly, Pennington [102] also observed programmers using a bottom-up strategy initially gathering statement and control-flow information. These micro-structures (statements, control constructs and relationships) were chunked and cross-referenced by macro-structures (text structure abstractions) to form a program model. A subsequent situation model was formed, also bottom-up, using application-domain knowledge to produce a hierarchy of data-flow and functional abstractions (the program goal hierarchy).

4) Knowledge-based strategies

Littman et al [84] observed that programmers use either a systematic approach, reading the code in detail and tracing through control and data-flow, or they use an "as-needed approach, focusing only on the code related to the task at hand. Subjects using a systematic strategy acquired both static knowledge (information about the structure of the program) and causal knowledge (interactions between components in the program when it is executed). This enabled them to form a mental model of the program, however, those using the as-needed approach only acquired

static knowledge resulting in a weaker mental model of how the program worked. More errors occurred since the programmers failed to recognize casual interactions between components in the program.

Soloway et al. [127] combined these two theories as macro-strategies aimed at understanding the software at a more global level. In the systematic macro-strategy, the programmer traces the flow of the whole program and performs simulations as all of the code and documentation is read. However, this strategy is less feasible for large programs. In the more commonly used as-needed macro-strategy, the programmer looks at only what they think is relevant. However, more mistakes could occur since important interactions might be overlooked.

5) Integrated strategies

Von Mayrhauser and Vans [88] combined the top-down, bottom-up, and knowledge-based approaches into a single metamodel. In their experiments, they observed that some programmers frequently switched between all three strategies. They formulated an integrated metamodel where understanding is built concurrently at several levels of abstractions by freely switching between the three types of comprehension strategies.

The model consists of four major components. The first three components describe the comprehension processes used to create mental representations at various levels of abstraction and the fourth component describes the knowledge base needed to perform a comprehension process:

- The top-down (domain) model is usually invoked and developed using an as-needed strategy, when the programming language or code is familiar. It incorporates domain knowledge as a starting point for formulating hypotheses.
- The program model may be invoked when the code and application is completely unfamiliar. The program model is a control-flow abstraction.
- The situation model describes data-flow and functional abstraction in the program. It may be developed after partial program model is formed using systematic or opportunistic strategies.
- The knowledge base consists of information needed to build these three cognitive models. It represents the programmer's current knowledge and is used to store new and inferred knowledge.

6) Factors affecting comprehension strategies

The general opinion most researchers realize is that certain factors will influence the comprehension strategy adopted by a programmer [130] [140]. These factors also explain the apparently wide variation in the comprehension strategies discussed above. The variations are primarily due to:

- Differences among programs,
- Aspects of the task at hand, and
- Varied characteristics of programmers.

To evaluate how programmers understand programs, these factors must be considered [130]. These are further explored in section 2.1.1.

With experience, programmers "know" which strategy is the most effective for the given program and task. A change of strategy may be needed because of some anomaly of the program or the requested task. Program understanding tools should enhance or ease the programmer's preferred strategies, rather impose a fixed strategy may not always be suitable.

B. Program and programmers trends

Both program and programmer influence a comprehension strategy choice by their inherent and varied characteristics. Additionally, this choice also depends of the task at hand. This section debates these issues giving an insight on the subject, available studies and trends for future research.

1) Program characteristics

Programs that are carefully designed and well documented will be easier to understand change or reuse in the future. Pennington's experiments showed that the choice of language as an effect on comprehension processes [102[104]. For instance, COBOL programmers consistently fared better at answering question related to data-flow than FORTRAN programmers, while these would fare better at control-flow questions than their counterparts.

Object-oriented (OO) programs are often seen as a more natural fit to problems in real world because of "is-a" and "is-part-of" relationships in a class hierarchy and structure, but others argue that objects do not always map easily to real world problems [32]. In OO programs, abstractions are achieved through encapsulation and polymorphism. Message passing is used for communication between class methods and hence programming plans are dispersed (i.e., scattered) throughout classes.

2) Program trends

As new techniques and programming paradigms emerge and evolve, the comprehension process must shift to embrace these changes. New characteristics on both program and programming approaches seem to produce new trends for comprehension research. A few follow [129]:

a) Distributed applications.

Along with web-based applications, both are becoming more prevalent with technologies such as .NET, J2EE and web services. One programming challenge that is occurring now and is likely to increase is the combination of different paradigms in distributed applications, e.g., a client side script sends XML to a server application (which currently evolved to the AJAX [51] technology).

b) Higher levels of abstraction.

Visual composition languages for business applications are also on the increase. As the level of abstraction increases, comprehension challenges are shifting from code understanding to more abstract concepts.

c) Aspect-oriented programming.

The introduction of aspects [76] as a construct to manage scattered concerns (delocalized plans) in a program has created new excitement in the software engineering community. Aspects have been shown to be effective for managing many programming concerns, such as logging and security. However, it is not clear how aspects written by

others will improve program understanding, especially in the long term. More empirical work is needed to validate the assumed benefits of aspects.

d) Improved software engineering practices.

The more informed processes that are used for developing software today will hopefully lead to software that is easier to comprehend in the future. Component-based software systems are currently being designed using familiar design patterns [49][25] and other conventions. Future software may have traceability links to requirements and improved documentation such as formal program specifications. Also, future software may have autonomic properties, where the software self-heals and adapts as its environment changes – thus in some cases reducing time spent on maintenance.

e) Diverse sources of information.

The program comprehension community, until quite recently, mostly focused on how static and dynamic analyses of source code, in conjunction with documentation, could facilitate program comprehension. Modern software integrated development environments, such as the Eclipse Java development environment [36], NetBeans or Visual Studio [93], also manage other kinds of information such as bug tracking, test cases and version control. This information, combined with human activity information such as emails and instant messages, will be more readily available to support analysis in program comprehension. Domain information should also be more accessible due to model driven development and the semantic web.

3) Programmer individual characteristics

There are many individual characteristics that will impact how a programmer tackles a comprehension task. These differences also impact the requirements for a supporting tool. There is a huge disparity in programmer ability and creativity, which cannot be measured simply by their experience.

In her work [143], Vessey presents an exploratory study to investigate expert and novice's debugging processes. She classified programmers as expert or novice based on their ability to chunk effectively. She notes that experts used breadth-first approaches and at the same time were able to adopt a system view of the problem area, whereas novices used breadth-first and depth-first approaches but were unable to think in system terms.

Détienne [32] also notes that experts make more use of external devices as memory aids. Experts tend to reason about programs according to both functional and object-oriented relationships and consider the algorithm, whereas novices tend to focus on objects.

4) Programmer trends

As with everything else, programmers also adapt and evolve, trying to accompany the paradigm shifts and new trends in their development environment. Relevant issues are [129]:

a) Program comprehension everywhere.

The need to use computers and software intersects every walk of life. Programming, and hence program comprehension, is no longer a niche activity. Scientists and knowledge workers in many walks of life have to use and customize software to help them do science or other work. Scientists from diverse fields, such as forestry, astronomy or medical science, are using and developing sophisticated software without a formal education in computer science. Consequently, there is a need for techniques to assist in non-expert and end-user program comprehension. Fortunately, there is much work on this area (especially at conferences such as Visual Languages and the PPIG group, where they investigate how comprehension can be improved through tool support for spreadsheet and other end user applications.

b) Sophisticated users.

Currently, advanced visual interfaces are not often used in development environments. A large concern by many tool designers is that these advanced visual interfaces require complex user interactions. However, tomorrow's programmers will be more familiar with game software and other media that displays information rapidly and requires sophisticated user controls. Consequently, the next generation of users will have more skill at interpreting information presented visually and at manipulating and learning how to use complex controls.

c) Globally distributed teams.

Advances in communication technologies have enabled globally distributed collaborations in software development. Distributed open source development is having an impact on industry. Some notable examples are Linux and Eclipse. Some research has been conducted studying collaborative processes in open-source projects [94] [58] [52], but more research is needed to study how distributed collaborations impact comprehension.

C. Tools for Program Comprehension

Understanding a software program is often a difficult process because of missing, inconsistent, or even too much information. The source code often becomes the sole arbiter of how the system works. The field of program comprehension research has resulted in many diverse tools to assist in program comprehension. When developing such tools, experts bring knowledge from other fields of research as Software Visualization and Reverse Engineering as means to answer the researched requirements. This section provides insight over the studies made to improve tool development to assist on program comprehension.

1) Tool requirements studies

Which features should an ideal tool have to efficiently support program comprehension? Needless to say that these tools will only play a supporting role to other software engineer tasks, such as design, development, maintenance, and (re) documentation.

There are mainly two ways of conducting studies to discover effective features to support program comprehension: an empirical approach by observing programmers trying to understand programs and an approach based on personal experience and intuition. Given the variability in comprehension settings, both approaches contribute to answering this complex question.

As such, several studies already conducted by several authors revealed a number of tool requirements, as follows.

Biggerstaff [15] notes that one of the main difficulties in understanding comes from mapping what is in the code to the software requirements – he terms this the concept assignment problem. Although automated techniques can help locate programming concepts and features, it is challenging to automatically detect human oriented concepts. The user may need to indicate a starting point and then use slicing techniques to find a related code. It may also be possible for an intelligent agent (that has domain knowledge) to scan the code and search for candidate start points. From his research prototypes he found that queries, graphical views and hypertext were important tool features.

Von Mayrhauser and Vans [89], from their research on the Integrated Metamodel, make an explicit recommendation for tool support for reverse engineering. They determined basic information needs according to cognitive tasks and suggested the following tool capabilities to meet those needs:

- Top-down model: online documents with keyword search across documents; pruning of call tree based on specific categories; smart differencing features; history of browsed locations; and entity fan-in.
- Situation model: provide a complete list of domain sources including non-code related sources; and visual representation of major domain function.
- Program model: Pop-up declarations; online cross-reference reports and function count.

Singer and Lethbridge [123] also observed the work practices of software engineers. They explored the activities of a single engineer, a group of engineers, and considered company-wide tool usage statistics. Their study led to the requirements for a tool that was implemented and successfully adopted by the company. Specifically they suggested tool features to support "just-in-time comprehension of source-code". They noted that engineers after working on a specific part of the program quickly forget details when they move to a new location. This forces them to rediscover information at a later time. They suggest that tools need the following features to support rediscovery:

- Search capabilities so that the user can search for code artifacts by name or by pattern matching.
- Capabilities to display all relevant attributes of the items retrieved as well as relationships among items.
- Features to keep track of searches and problem-solving sessions, to support the navigation of a persistent history.

Erdös and Sneed [41] designed a tool to support maintenance following many years of experience in the maintenance and reengineering industry. They proposed that the following seven questions needed to be answered for a programmer to maintain a program that is only partially understood:

- Where is a particular subroutine/procedure invoked?
- What are the arguments and results of a function?
- How does control flow reach a particular location?
- Where is a particular variable set, used or queried?
- Where is a particular variable declared?
- Where is a particular data object accessed?
- What are the inputs and outputs of a module?

Other attempts to capture tool requirements were made that involved observation of programmers performing different tasks.

Murray and Lethbridge [98] observed software professionals using a mixed approach combining elements from specific methods used in software engineering empirical research and a sociological qualitative research called "ground theory". From this approach, they were able to develop the basis for a theory of the ways people think when explaining and comprehending software, which they called "cognitive patterns". These patterns can then be applied to further empirical observatory studies as a roadmap to capture programmer behaviour.

Zayour [150] proposes a methodology for assessing cognitive requirements and adoption success for reverse engineering tools, from which he concludes five main rules of thumb: (1) A clear and realistic definition of the problem space to be targeted is a must; (2) direct observation of the targeted user is required to form a realistic perception of users problems and tasks; (3) Tool designers should document their perception of the user's problems and tasks; (4) When determining the success of a tool, cognitive load is a more important indicator to measure than elapsed time (because it affects adoptability more) and (5) design should be aimed at satisfying cognitive requirements and thus should be guided by cognitive principles.

Work by other authors included recall tests to evaluate the ability to answer questions regarding a piece of code programmers studied for a limited period of time [102]. Subjective ratings [120] have been used recently to measure different levels of comprehension. Additionally, other studies may ask subjects to label or group different code members based on the similarity of their functionalities [113]. Soloway and Erlich [128] asked programmers to fill blank lines and complete unfinished programs on paper in an unfamiliar source code without providing specifications about the program's use or functionality. Similarly, Bertholf et al. [14] asked novice developers to complete incomplete literal programs on paper. Additional techniques to measure program comprehension involved completing incomplete call graphs, modifying existing code, report a bug, or separate source code from two different algorithms [121].

From this research and derived from cognitive theories, Storey [129] extracts and synthesizes several tool requirements:

a) Browsing support.

The top-down process requires browsing from high-level abstractions or concepts to lower-level details, taking advantage of beacons in the code; bottom-up comprehension requires following control-flow and data-flow links, both novices and experts can benefit from tools that support breadth-first and depth-first browsing; and the Integrated Metamodel suggests that switching between top-down and bottom-up browsing should be supported. Flexible browsing support also will help to offset the challenges from delocalized plans.

b) Searching.

Tool support is needed when looking for code snippets by analogy and for iterative searching. Also inquiry episodes should be supported by allowing the programmer to query on the role of a variable, function, etc.

c) Multiple views.

Programming environments should provide different ways of visualizing programs. One view could show the message call graph providing insight into the programming plans, while another view could show a representation of the classes and relationship between the to show an object-centric or data-centric view of the program. These orthogonal views, if easily accessible, can facilitate comprehension, especially when combined.

d) Context-drive views.

The size of the program and another program metrics will influence which view is the preferred one to show a programmer browsing the code for the first time. For example, in an object-oriented program, it is usually preferable to show the inheritance hierarchy as the initial view. However, if the inheritance hierarchy is flat, it may be more appropriate to show a call graph as the default view.

e) Additional cognitive support.

Experts need external devices and scratchpads to support their cognitive tasks, whereas novices need pedagogical support to help them access information about the programming language and the corresponding domain.

2) Tool development

Programming comprehension tools can be roughly grouped into three categories [139]:

- Extraction tools include parsers and data gathering tools.
- Analysis tools do static and dynamic analysis to support activities such as clustering, concept assignment, feature identification, transformations, domain analysis, slicing and metrics calculation.
- Presentation tools include code editors, browsers, hypertext and visualizations. They are strongly linked to research in software visualization.

Integrated software development and reverse engineering environments will usually have some features form each category. The set of features they support is usually determined by the purpose for the resulting tool or by the focus of the research. As such, two majors areas that relate to this issue are Software Visualization and Reverse Engineering.

a) Software Visualization

Software visualization tools and browsing tools provide information that is useful for program understanding.

These tools use graphical and textual representations for the navigation, analysis and presentation of software information to increase understanding. Mixed results have been reported through the literature on the role of text and graphics for program comprehension. While Green and Petre [53] observed that text was faster than graphics for experimental program comprehension tasks, Scanlan [117] reported an improvement using graphical visualizations when comparing textual algorithms and structured flowcharts. Petre [104] attributes the difficulty in understanding program visualizations to the fact that graphical representations have fewer navigational cues, namely secondary notations, when compared to program text: source code implies a serial inspection strategy. Moreover, she observed that experienced readers tend to use parallel textual and graphical information whenever available to assist their comprehension process: they use text as a main source to guide their understanding of graphical representation.

Several software visualization tools show animations to teach widely used algorithms and data structures [22] [125] [131]. Another class of tools shows dynamic execution of programs for debugging, profiling and for understanding run-time behavior [68] [115]. Other software visualization tools mainly focus on showing textual representations, some of which may be pretty printed to increase understanding [9] [63] or use hypertext in an effort to improve the navigability of the software [104]. Typography plays a significant role in the usefulness of these textual visualizations.

Many tools present relevant information in the form of a graph where nodes represent software objects and arcs show the relations between the objects. This method is used by PECAN [102], Rigi [96], VIFOR [107], Whorf [19], CARE [83], Hy+ [91] and Imagix 4D [67]. Other tools use additional pretty printing techniques or other diagrams to show structures or information about the software. For example, the GRASP tool uses a control structure diagram to display control constructs, control paths and the overall structure of programming units [130].

b) Reverse Engineering

Reverse Engineering concerns how to extract relevant knowledge from source code and present it in a way that facilitates comprehension. Several studies conducted in the past have proposed solutions on how to overcome caveats in the program comprehension process. Maryhauser and Vans [89], Singer and Lethbridge [123] and Zayour [150] have given their insight on how to address tool development for reverse engineering of useful information to assist on program understanding (seen on section 2.1). K.Wong also discusses reverse engineering tool features [149]. He specifically mentions the benefits of using a “notebook” to support ongoing comprehension.

Usually, the reverse engineering tools and techniques associated to program comprehension are bundled into broader development environments where other types of tools also co-exist.

It is possible to examine each of these environments and to recover the motivation for the features they provide by tracing back to the cognitive theories. For example, the Rigi system [96] has support for multiple views, cross-referencing and queries to support bottom-up comprehension. The Reflection tool [97] has support for the top-down approach through hypothesis generation and verification. The Bauhaus tool [38] has features to support clustering (identification of components) and concept analysis. The SHriMP tool [132] provides navigation support for the Integrated Metamodel, i.e, frequent switching between strategies. And the

Codecrawler tool [79] uses visualization of metrics to support understanding of an unfamiliar system and to identify bottlenecks and other architectural features.

All these tools combine reverse engineering tasks with software visualization techniques to improve program comprehension on different levels of abstraction, gathering information recovered or simply mined together into user-friendly viewed chunks of valuable data for the programmer.

3) Tool trends

The forthcoming breakthroughs in tool technology seem promising as research and evaluation methods and theories become more relevant to end-users doing programming-like tasks. Therefore, directions in tool evolution appear to follow several guidelines presented next [129].

a) Faster tool innovations.

The use of frameworks as an underlying technology for software tools is leading to faster tool innovations, as less time needs to be spent reinventing the wheel. A prime example of how frameworks can improve tool development is the Eclipse platform [36]. Eclipse was specifically designed with the goal of creating reusable components, which would be shared across different tools. The research community benefits from this approach in several ways. Firstly, they are able to spend more time writing new and innovative features as they can reuse the core underlying features offered by Eclipse and its plug-ins; and secondly, researchers can evaluate their prototypes in more ecologically valid ways as they can compare their new features against existing industrial tools.

b) Mix 'n match tools.

Given a suite of tools that all plug in to the same framework, together with a standard exchange format (such as GXL), researchers will be able to more easily try different combinations of tools to meet their research needs. This should result in increased collaborations and more relevant research results. Such integrations will also lead to improved accessibility to repositories of information related to the software including code, documentation, analysis results, domain information and human activity information. Integrated tools will also lead to fewer disruptions for programmers.

c) Recommenders and search.

Software engineering tools, especially those developed in research, are increasingly leveraging advances in intelligent user interfaces (e.g., tools with some domain or user knowledge). Recommender systems are being proposed to guide navigation in software spaces. Examples of such systems include Mylar [74] and NavTracks [124]. Mylar, (now called MyLyn) uses a degree of interest model to filter non-relevant files from the file explorer and other views in Eclipse. NavTracks provides recommendations of which files are related to the currently selected files. Deline et al. also discuss a system to improve navigation [31]. The FEAT tool suggests using concern graphs (explicitly created by the programmer) to improve navigation efficiency and enhance comprehension [114]. Search technologies, such as Google, show much promise at improving search for relevant components, code snippets and related code. The Hipikat tool [30] recommends relevant software artifacts based on the developer's current project context and development history. The Prospector system recommends relevant code snippets [86]. It combines a search engine with the content assist in Eclipse to help programmers use complex APIs. Although new, this work shows much promise and it is expected to improve navigation in large systems while reducing the barriers to reuse components from large libraries.

d) Adaptive interfaces.

Software tools typically have many features, which may be overwhelming not only for novice users, but also for expert users. This information overload could be reduced through the use of adaptive interfaces. The idea is that the user interface can be tailored automatically, i.e., will self-adapt, to suit different kind of users and tasks. Adaptive interfaces are now common in Windows applications such as Word. Eclipse has several novice views (such as Gild [132] and Penumbra) and Visual Studio has the Express configuration for new users. However, neither of these mainstream tools currently have the ability to adapt nor even to be easily manually adapted to the continuum of novice to expert users.

e) Visualizations.

These have been subject of much research over the past ten to twenty years. Many visualizations, and in particular graph-based visualizations, have been proposed to support comprehensions tasks, some of then already referred in section 2.2.2. Other examples include Seesoft [11], Bloom [111], Landscape views [103], and sv3D [87]. Graph visualization is used in many advanced commercial tools such as Klocwork, Imagix4D and Together. UML Diagrams are also commonplace in mainstream development tools. One challenge with visualizing software is scale and knowing at what level of abstraction details should be shown, as well as selecting which view to show. More details about the user's task combined with metrics describing the program's characteristics (such as inheritance depth) will improve how visualizations are currently presented to the user. A recommender system could suggest relevant views as a starting point. Bull proposes the notion of model-driven visualization [24]. He suggests creating a tool for tool designers and expert users that recommends useful views based on characteristics of the model and the data.

f) Collaborative support.

As software teams increase in size and become more distributed, collaborative tools to support distributed software development activities are more crucial. In research, there are several collaborative software engineering tools being developed such as Jazz and Augur [66] [47]. There are also some collaborative software engineering tools deployed in industry, such as CollabNet, but they tend to have simple tool features to support communication and collaboration, such as version control, email and instant messaging. Current industrial tools lack more advanced collaborative features such as shared editors, and research falls short on providing empirical work to improve these

tools. Another area for research that may prove useful is the use of large screen displays to support co-located comprehension. O'Reilly et al. [101] propose a war room command console to share visualizations for team coordination. There are other research ideas in the CSCW (computer supported collaborative work) field that could be applied to program comprehension.

g) Domain and pedagogical support.

The need to support domain experts that lack formal computer science training will necessarily result in more domain-specific languages and tools. Non-experts will also need more cognitive scaffolding to help them learn new tools, languages and domains more rapidly. Pedagogical support, such as providing examples by analogy, will likely be an integral part of the future software tools. The work discussed above on recommending code examples is also suggested at helping novices and software immigrants (i.e., programmers to a new project). Results from the empirical work also suggest that there is a need for tools to help programmers learn a new language. Technologies such as TXL [29] can play a role in helping a user see examples of how code constructs in one language would appear in a new language.

III. FRAMEWORK UNDERSTANDING

Program comprehension covers a wide range of sub-areas when it comes to comprehend programs. When we say programs, we mean software artefacts: constructs built upon source-code. A framework can be considered one of such artefacts and, due to its importance and growing adherence by the software community, spawned and research area of its own. Framework understanding deals with understanding and learning about a framework for usage, implementation, and evolution.

Object-oriented frameworks are a powerful form of reuse but they can be difficult to understand and reuse correctly. They are promoted as having the potential to provide the benefits of large-scale reuse [49] [25] [43]. While practical evidence does suggest that framework usage can increase reusability and decrease development effort [95], experience has identified a number of issues that complicate framework application and limit potential benefits [18]. One of the major challenges is effective framework understanding – a specialized kind of program comprehension.

Over the past decade a large range of candidate documentation techniques has been proposed to support framework understanding, including design patterns [29], pattern languages [70], example-based learning [118], cookbooks [78], hooks [48] and exemplars [50].

However, the lack of investigation of these techniques and their impact in framework understanding, together with the lack of insight into problems that limit the comprehension and reuse of software frameworks, spurred a few studies, which identified some concerns and bases for future research in the field. The next section will briefly address some of these studies, and, afterwards, a brief review of some existing tools and approaches to aid in framework understanding and reuse. An overall depiction of the main ideas behind framework understanding is shown in Figure 2.

A. Reuse and comprehension issues

There is a considerable quantity of literature into framework domain, but little of it deals with the identification of reuse problems or evaluation of strategies to support the framework developer as a whole. There are tools that address topics under the realm of framework building, design recovery and documentation, but none clearly emphasizes or studies the overall symptoms behind ineffective framework reuse, and thus hindering a framework's main goal.

Fayad and Schmidt [43] claimed that different alternatives could improve framework understandability:

- Refining the framework's internal design.
- Using methods that can ensure a successful development and usage of frameworks.
- Adhering to standards for framework development, adaptation, and integration.
- Producing comprehensible framework documentation.

These guidelines are mainly preventive and don't focus on the issue of reusability, being general advices. Nevertheless, they can be relevant as rules of thumb for framework development and maintenance.

Butler, Keller and Milli [26] describe a taxonomy of framework documentation primitives that appear to address reusability issues. They describe six primitives, which emphasize the need for information about class interfaces and communication protocols between classes.

Johnson [70] identifies three important areas for framework documentation to address – purpose, how to use and design. He argues that the purpose of the framework and its constituent parts should be communicated so that developers may select the correct parts for a task. While knowledge of how those parts are expected to operate allows them to be employed correctly and a description of the underlying design provides developers with an understanding of how to adapt and extend the framework in a manner consistent with existing structure.

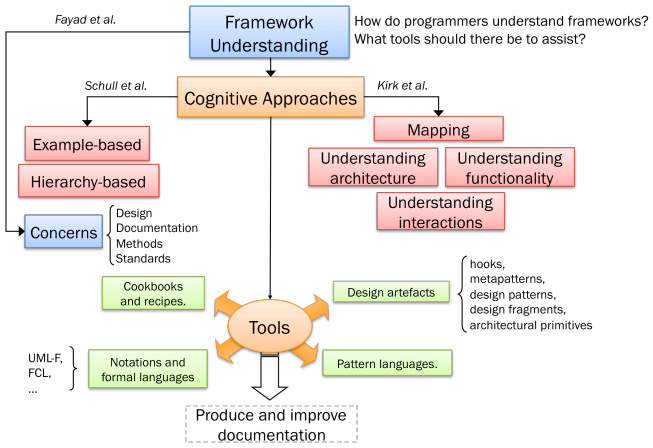

Figure 2 - Framework Understanding topics

Shull et al. [118] presents an evaluation of the role that examples play in framework reuse. Their study compared two approaches to framework reading and eventual documentation, and example-based approach and a hierarchical-based approach. Their results suggested that examples are an effective learning strategy, especially for those beginning to learn a framework. They also identify potential problems with an example-based approach: finding the small pieces of required functionality in larger examples, inconsistent organization and structure of examples and lack of design choice rationale in example documentation. They also discuss the possibility that developers become too reliant on examples and do not understand the system at a sufficient level of detail to implement effectively from scratch when necessary.

Kirk et al. [75] conducted a research, through observation of both novice and experienced re-users, where they identified four fundamental problems of framework reuse:

- Mapping identifies the problem on translating an abstract, conceptual solution into a concrete implementation, which reuses the existing structures within the framework. Such problems were often expressed as "what should I use to represent…?" or "How do I express…?"
- Understanding functionality describes problems understanding what specific parts of the framework actually do. Manifestations of this problem included "How does … work?", "Where … does happen?" or "Where is … defined/created/called?"
- Understanding interactions focuses on problems concerning the communication between classes in the framework ("What happen if …?" or "Where should I put …?"). Such problems are significant because of hidden or subtle dependencies within the framework that may cause failures to occur elsewhere as the result of a wrongly positioned modification.
- Understanding the framework architecture is the problem of making modifications without giving appropriate consideration to the high-level architectural qualities of the framework. Such alterations might have no short-term effects but ultimately lead to the framework losing its flexibility.

From these problems, the authors experimented applying two known solutions they deemed the most suited to address these issues: pattern languages and micro-architectures. Their results showed that the pattern language provided some support for mapping problems, particularly for those with no experience of the framework, by introducing key framework concepts and providing examples of framework use. However, it was clear that previous experience dominated the explicit use of the pattern language, as well as being an inhibitor to other forms of documentation as its immediacy often precludes consideration of alternative solutions.

Although the micro-architectures, used to help develop and understanding of the key interactions within the framework, seemed relatively ineffective, it is the authors' belief that documentation of this kind is necessary to address these problems in particular.

B. Tools to assist framework understanding

As for program comprehension tools, the same line of thinking applies for framework understanding tools. Both subjects share the same problems and trends, yet some framework specific issues may be addressed when devising aids to framework learning and understanding.

The past and present research in the field focus on topics that range from uncovering design artifacts to representing processes and behaviors that might help using the framework. Mostly, the proposals converge to producing and enhancing existing documentation with adequate information that can be mined and represented using different formats (recipes, cookbooks), languages (patterns, beacons, idioms) and notations (textual, graphical, UML, formal languages, etc.). Next, a brief summary of these proposals is presented. The categorization used emerged from its most relevant technique, yet several use mixed approaches combining several techniques to optimize their results.

1) Cookbooks

Confronting the challenge of communicating how to use the Model-View-Controller framework in Smalltalk-80, Krasner and Pope [78] built an 18-page cookbook that explained the purpose, structure, and implementation of the MVC framework. This cookbook was designed to be read from beginning to end by programmers and could also be used as a reference but every recipe did not follow a consistent structure nor was it suitable for parsing by automatic tools.

The Framework EDitor / JavaFrames project [59] [60] [61] has developed a language for modelling design patterns and tools that act as smarter cookbooks, guiding programmers step-by-step to use a framework. With the 2.0 release of JavaFrames, many of these tools work within the Eclipse IDE. Their language allows expression of structural constraints and the tool can check conformance with the structural constraints. Code can be generated that conforms to the patterns definition, optionally including default implementations of method bodies. Specific patterns can be

related to general patterns; for example a specific use of the Observer pattern in a particular framework can be connected to a general definition of the Observer pattern.

2) Design Artifacts

Ralph Johnson seems to be the first to suggest documenting frameworks using patterns [70]. He notes that the typical user of framework documentation wants to use the framework to solve typical problems, but also that cookbooks do not help the most advanced users [71]. Patterns can be used both to describe a framework's design as well as how it is commonly used. He argues that the framework documentation should describe the purpose of the framework, how to use the framework, and the detailed design of the framework. After presenting some graduate student with his initial set of patterns for HotDraw [20], he realized that a pattern isolated from examples is hard to comprehend.

Froelich et al.'s hooks [48] focus on documenting the way a framework is used, not the design of the framework. They are similar in intent to cookbook recipes but are more structured in their natural language. The elements listed are: name, requirement, type, area, uses, participants, changes, constraints, and comments. The instructions for framework users (the changes section) read a bit like pseudo code but are natural languages and do not appear to be parsable by tools.

Design patterns themselves can be decomposed into more primitive elements [106]. Pree calls these primitive elements metapatterns and catalogues several of them with example usage. He proposes a simple process for developing frameworks where identified points of variabilility are implemented with an appropriate metapattern, enabling the framework user to provide an appropriate implementation.

From the declarative metaprogramming group from Vrije University, Tourwé and Mens [141] [142] use Pree's metapatterns to document framework hotspots and define transformations for each framework and design patterns. Framework instances (plug-ins) can be evolved (or created) by application of the transformations. The tool uses SOUL, a prolog-like logic language. The validation was done on the HotDraw framework by specifying the metapatterns, patterns and transformations needed. The validation uncovered design flaws in HotDraw, despite its widespread use, along with some false positives. The declarative metaprogramming approach to modeling framework hotspots appears to have significant up-front investment before payoff in order to provide its guarantees about correct use of the framework. It may additionally assume a higher level of accuracy or correctness in frameworks than will commonly be found in practice. The authors comment that their approach specifically avoids design patterns in favor of metapatterns because there could be many design patterns. While this makes their technique generally applicable and composable, it will be difficult to add pattern-specific semantics and behavior checking to their approach.

JFREEDOM [44] is a design recovery tool that discovers metapatterns in a framework or software system. It relies on Tourwé's formal definition of metapatterns and uses JQuery, a logic inference-engine, to search the code for instances of these metapatterns. It then recommends possible GoF [49] design pattern instances based on its found metapatterns. Other design pattern recovery tools exist and a brief review of each one can be found in [44]. Design pattern recovery is, by itself, a research field where a community recently formed to combine efforts.

Bruch et al. [23] propose the use of data mining techniques to extract reuse patterns from existing framework instantiations. Based on these patterns, suggestions about other relevant parts of the framework are presented to novice users in a context-dependent manner. They built FrUiT, an Eclipse plug-in that implements the approach and, yet at an early stage, already presents several benefits: relying on expert-written framework instantiations, there is no need to create special artifacts such as documentation or code snippets; using data mining, significant reuse rules are extracted, only concerning how to use the framework; and the tool makes automatic context search relieving developers from searching for rules explicitly.

Fairbanks et al. [42] present a pattern language based on the notion of design fragment. A design fragment is a pattern that encodes a conventional solution to how a programmer interacts with a framework to accomplish a certain goal. It provides the programmer with a "smart flashlight" to help him/her understand the framework, illuminating only those parts of the framework he/she needs to understand for the task at hand. They use XML to express these patterns, so that automation tools are a step away. They have analyzed the 20 Java applets provided by Sun and came up with a catalogue of design fragments, which evaluated against other 36 applets from the internet proved that those design fragments were common and recurrent. Design fragments gives programmers immediate benefit through tool-based conformance and long-term benefit through expression of design intent.

Zdun and Avgeriou [151] propose to remedy the problem of modeling architectural patterns through identifying and representing a number of architectural primitives that can act as the participants in the solution that patterns convey. According to the authors, these "primitives" are the fundamental modeling elements in representing a pattern and also they are the smallest units that make sense at the architectural level of abstraction (e.g., specialized components, connectors, ports, interfaces). Their approach relies on the assumption that architectural patterns contain a number of architectural primitives that are recurring participants in several other patterns. They chose UML as the preferred notation to represent the primitives and pretend to formalize the definitions using OCL.

3) Notations and formal languages

A UML profile is a restricted set of UML markup along with new notations and semantics [46]. Fontoura et al. present the UML-F profile that provides UML stereotypes and tags for annotating UML diagrams to encode framework constraints. Methods and attributes in both framework and user code can be marked up with boxes (grey, white, half-and-half, and a diagonal slash) that indicate the method/attribute's participation in superclass-defined template patterns. A grey-box indicates newly defined or

completely overridden superclass method. A white box indicate inherited and not redefined, a half-and-half indicates refined but call to super(), and a slashed box indicates an abstract superclass method.

The Fixed, Adapt-static, and Adapt-dyn tags annotate the framework and constrain how users can subclass. Template and Hook tags annotate framework and user code to document template methods. Stereotypes for Pree's metapatterns (like unification and separation variants) are present, as are predefined tags for the GoF patterns. Recipes for framework use are present in a format very similar to that of design patterns but there is no explicit representation of the solution versus the framework. The recipe encodes a list of steps for programmer to perform.

The Framework Constraint Language (FCL) [65] applies the ideas from Richard Helms object oriented contracts [62] to frameworks. Much like Riehle's role models [112], FCLs specify the interface between the framework and the user code such that the specification describes all legal uses of the framework. The researchers raise the metaphor of FCL as framework-specific typing rules and validate their approach by applying it to Microsoft Foundation Classes, historically one of the most widely used frameworks. The language has a number of built-in predicates and logical operators and is designed to operate on the parse tree of the user's code.

C. Trends

Not as developed as program comprehension, framework understanding research still has room for expansion, and future work is needed to address existing open issues. It shares the same trends as program comprehension, yet it has its own issues. Reuse problems must be better addressed by documentation or tool support if frameworks are to be widely adopted. There are still significant and stimulant challenges:

1) Pattern languages.

While developing pattern languages for framework documentation, some issues have to be addressed such as identifying the expertise necessary to create effective pattern languages, how to identify the framework domain problems that should be the basis of patterns in the pattern language, how to best describe patterns, and what inter-pattern relationships should be included.

2) Widen context domain research.

There is a clear need to investigate the prevalence of framework understanding problems in industrial context frameworks. Industry and academia have to join efforts to ascertain the impact frameworks learning problems have in large-scale software development environments, so that adequate solution may be searched for.

3) Integrated environments.

With the advent of pluggable and extensible software development environments (like, Eclipse), tools for assisting on framework understanding tend to be integrated into these self-sustainable platforms, producing solutions that are multi-faceted and present different and varied approaches to accommodate different user needs. The combination and personalization of these tools, offer flexibility to adjust the environments to the specific needs of particular users in particular tasks.

IV. COLLABORATIVE SOFTWARE ENVIRONMENTS

Software projects usually involve a team or multiple teams that have to work together. For some time now, there has been a concern on how to coordinate these teams of developers to be able to efficiently work together. Research areas such as Groupware and Computer-Supported Collaborative Work rose to address collaboration supported by software. The Collaborative Software Engineering domain deals with collaboration within the software development process. The next sections address these research areas in further detail.

A. Groupware and CSCW

Many credit Peter and Trudy Johnson-Lenz for coining the term "groupware" in 1978. They defined it as: "intentional group processes plus software to support them". This definition, however, was not widely accepted as it has narrowed the scope of group work to a set of processes.

Another attempt to provide a definition came from Johansen [69]: "Groupware… a generic term for specialized computer aids that are designed for the use of collaborative work groups. Typically, these groups are small project-oriented teams that have important task and tight deadlines. Groupware can involve software hardware, services, and/or group process support". This definition also didn't take, as it would exclude categories of products that were not designed specifically for supporting work groups, like email or shared databases. Besides that, it also focuses on small teams, which is also restrictive.

To broaden the scope, Ellis et al. [39] proposed to define groupware as: "computer-based systems that support groups of people engaged in a common task (or goal) and that provide an interface to a shared environment". Although less restrictive, this definition was considered too broad. Despite excluding multi-user systems (such as time-sharing systems where users don't share the same goal), it would include shared database systems. Many argue that these systems cannot be considered groupware because they provide the illusion that every user has independent access, alas, they are not "group-aware."

In general, as Grudin points out in [57] groupware means different things to different people. According to Nunamaker et al. [100], groupware is defined as "any technology specifically used to make a group more productive". Coleman states [28], "Groupware is an umbrella term for the technologies that support person-to-person collaboration; groupware can be anything from email to electronic meeting systems to workflow". These definitions although quite broad capture almost all the products and projects that are identified as groupware.

The common denominator in all the above definitions is the notion of group work. Groupware is designed to support teams of people working together. As such, groupware provides a new focus in software technology from human – computer to human – human interaction. Human interactions have three key elements: communication, collaboration and

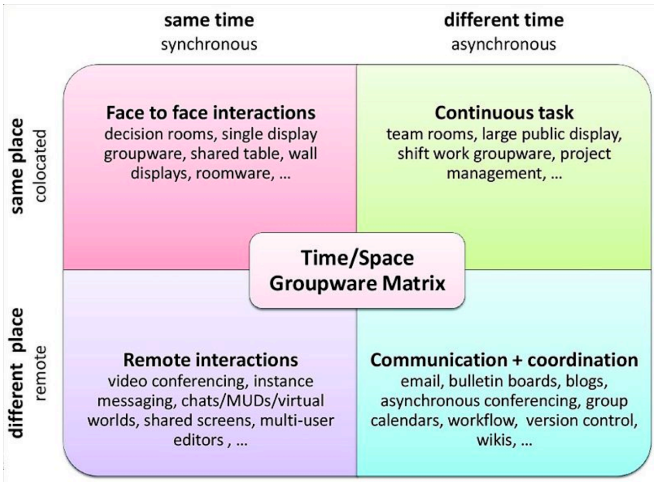

Figure 3 - Groupware Matrix (extracted from [69][10])

coordination. The goal of groupware is to assist groups in communicating, in collaborating and in coordinating their activities [39], and has been focusing on these issues for years.

The fact that most groupware tools failed to be widely adopted made clear the need for a better understanding of how groups of people work together. A new research area emerged called: "Computer-Supported Collaborative Work (CSCW)".

Iren Greif of MIT and Paul Cashman of Digital Equipment Corporation, who organized a workshop in 1984 for people interested in how groups work, coined the term CSCW. Since then, this new field attracted a lot of interest. Amongst the various definitions, Wilson's seems to have captured the scope of CSCW [148]: "CSCW [is] a generic term, which combines the understanding of the way people work in groups with the enabling technologies of computer networking, and associated hardware, software, services and techniques." Greenberg [54] adds: "CSCW is the scientific discipline that motivates and validates groupware design. It is the study and theory of how people work together, and how computer and related technologies affect group behavior."

CSCW collects researchers from a variety of specializations – computer science, cognitive science, psychology, sociology, anthropology, ethnography, management, and information systems – each contributing a different perspective and methodology for acquiring knowledge of groups and for suggesting how the group's work could be supported.

CSCW led to a better understanding of groups and made clear that group relationships are not based only on communication, collaboration and co-ordination. As pointed out by Kling [77]: "In practice, many working relationships can be multivalent with and mix elements of co-operation, conflict, conviviality, competition, collaboration, commitment, caution, control, coercion, co-ordination and combat."

CSCW researchers that design and build systems try to address core concepts in novel ways. These concepts have largely been derived through the analysis of systems designed by researchers in the CSCW community, or through studies of existing systems and the most addressed are:

- *Awareness*. Individuals working together need to be able to gain some level of shared knowledge about each other's activities [33].
- *Articulation work*. Cooperating individuals must somehow be able to partition work into units, divide it amongst themselves and, after the work is performed, reintegrate it [126].
- *Appropriation (or tailorability)*. How an individual or group adapts a technology to their own particular situation; the technology may appropriate in a manner completely unintended by the designers [34].

However, the complexity of the domain makes it difficult to produce conclusive results. The success of CSCW systems is often so contingent on the peculiarities of the social context that it is hard to generalize. Consequently, CSCW systems that are based on the design of successful ones may fail to be appropriated in other seemingly similar contexts for a variety of reasons that are nearly impossible to identify a priori [56].

In [2], Ackerman describe CSCW's main intellectual contribution has the effort to close the social-technical gap between what we know we must support socially and what we can support technically. He states that systems lack nuance, flexibility and ambiguity, clearly properties inherent to Human activity. Therefore, the social aspects must be taken into account when designing systems for these to be increasingly effective.

In [109], Weber et al. contributed with a taxonomy that defines and describes criteria for identifying CSCW systems and serves as a basis for defining CSCW system requirements. The criteria are divided into three major groups:

- *Application*. From an application viewpoint, certain tasks are generically present in many scenarios, from general-purpose tasks such as brainstorming, note taking and shared agenda features to more dedicated domains where there is the need for tailored tools. To the user, a CSCW system appears complete only when specialized and generic tools are integrated.
- *Functional*. A CSCW system relates functional features with the social aspects of teamwork. Each functionality has an impact on the work behavior and efficiency of the entire group using the system. Issues such as interaction, coordination, distribution, user-specific reactions, visualization and data hiding must be taken into consideration. However, the psychological, social, and cultural processes active within groups of collaborators are the real keys to the acceptance and success of CSCW Systems.
- *Technical*. This criteria comprises hardware, software and network support. It divides the architecture of a CSCW system into four classes of classes or features: (1) input, (2) output, (3) application, and (4) data. Each can be centralized or replicated.

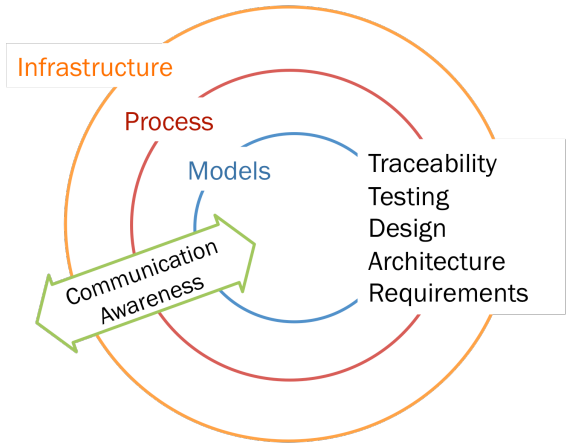

Figure 4 - Collaborative Software Engineering Model

For all of these groups, concerns such as flexibility, transparency, collaboration and sharing are addressed and guidelines for supporting them are presented.

Another approach to conceptualizing groupware and a CSCW system, states that its context can be considered along two dimensions: first, whether collaboration is co-located or geographically distributed, and second, whether individuals collaborate synchronously (same time) or asynchronously (not depending on others to be around at the same time). This approach can be seen in Figure 3 and was first introduced by Johansen [69] in 1988, also appearing in [10].

As the research continues, both groupware and CSCW fields still face challenges. The current trends evolve mostly in the following directions:

1) Mobile technologies.

With the emergence of new mobile technologies and the increasing connectivity users enjoy, the importance of having light, easy-to-use and accessible groupware features is growing.

2) Web 2.0.

With the advent of concepts of the so-called second-generation web or "Web 2.0", collaboration and contextual-connectivity become even more present in our day-to-day activities. From blogs to wikis, social software is booming and its capabilities should be harnessed to improve group work.

3) Strong commercial interest.

Major commercial competitors such as Microsoft, Google, IBM, amongst others, are releasing solutions into the market at an increasing rate. This must come as an incentive to continue researching into these ever-increasing fields of interest.

4) Delocalization of groups.

Teams and groups are becoming more and more delocalized. Work stops at one side of the planet and starts contiguously on the other side. Communication and synchronism become critical for a adequate and effective flow of work.

B. Collaborative Software Engineering

Software engineering projects are inherently cooperative, requiring many software engineers to coordinate their efforts to produce a large software system [146]. As such, this effort encompasses the development of a shared understanding surrounding multiple artefacts, each embodying its own model, over the entire development process. Figure 4 depicts that effective communication and awareness are crosscutting concerns across, not only the phases of software development but its models, process and infrastructure.

Collaboration techniques in software engineering have evolved to address our limitations: humans are slow and error-prone, especially when working at high-levels of abstraction; our natural language is expressive but ambiguous; our memory skips the details of large projects and we can't keep track of what everyone is doing.

Software engineering collaboration has multiple goals spanning the entire lifecycle of development:

a) Establish the scope and capabilities of a project.

Engineers must work with the users and stakeholders of a software project to describe what it should do at both a high level, and at the level of detailed requirements. How this collaboration takes place can have profound impact on a project, ranging from the up-front negotiation of the waterfall model, to the iterative style of evolutionary prototyping [90].

b) Converge towards a final architecture and design.

System architects and designers must negotiate, create alliances, and engage domain experts to ensure convergence on a single system architecture and design [55].

c) Manage dependencies among activities, artefacts, and organizations.

This encompasses a wide range of collaborative activities, including typical management of subdividing work into tasks, ordering them, monitoring, assessing, and controlling the plan of activities [85].

d) Reduce dependencies amongst engineers.

An important mechanism for managing dependencies is to reduce them where possible, thereby reducing the need for collaboration. Defining per-developer workspaces helps reducing dependencies in development time.

e) Identify, record and resolve errors.

Errors and ambiguities exist in all software artefacts, and many approaches have been developed to find and record them. Collaborative techniques such as inspections, reviews, beta testing and bug tracking assist on mitigating these problems and tracking the quality of the software.

f) Record organizational memory.

In any long running collaborative project, people may join and leave. Part of the work of collaboration is recording what people know, so that project participants can learn this knowledge now, and in the future [3]. SCM change logs are one form of organizational memory in software projects, as are project repositories of documentation. Process models also record organizational memory, describing best practices for how to develop software.

Collaboration in software engineering can be unstructured, where occasional and sporadic informal conversations occur concerning a piece of software anywhere in the project's lifecycle. It can also be structured, where the focus goes to various formal and semi-formal artefacts (requirement specifications, architecture diagrams, UML diagrams, source-code, bug reports, etc.) Software engineering collaboration can thus be understood as artefact-based, or model-based collaboration, where the focus of activity is on the production of new models, the creation of shared meaning around the models, and elimination of error and ambiguity within the models. Without the structure and semantics provided by the model, it would be more difficult to recognize differences in understanding among collaborators.

This focus on model-oriented collaboration embedded within a larger process is what distinguishes collaboration research in software engineering from broader collaboration research, which tends to address artefact-neutral coordination technologies and toolkits.

Software engineers have developed a wide range of model-oriented technologies to support collaborative work on their projects. These technologies span the entire lifecycle, including collaborative requirements tools [16][136], collaborative UML diagram creation, software configuration management systems and bug tracking systems [137].

Process modelling and enactment systems have been created to help manage the entire lifecycle, supporting managers and developers in assignment of work, monitoring current progress, and improving processes [17] [81]. In the commercial sphere, there are many examples of project management software, including Microsoft Project [92] and Rational Method Composer [108]. Several efforts have created standard interfaces or repositories for software project artefacts, including WebDAV/DeltaV [35][147] and PCTE [144]. Web-based integrated development environments serve to integrate a range of model-based (SCM, bug tracking systems) and unstructured (discussion list, web pages) collaboration technologies.

1) Tools, environments and infrastructure

Tool support developed specifically for collaboration in software engineering falls into four broad categories:

a) Model-based collaboration tools.

Software engineering involves the creation of multiple artifacts. These range from the end product and the source code to all the models, diagrams and specifications that cover all the phases of the software development process. Each artifact has its own semantics, with a variable degree of formality, and creating them is an inherently collaborative activity. Systems designed to support the collaborative creation and editing of specific artifacts are really supporting the creation of specific models, and hence support the model-based collaboration. Collaboration tools exist to suppose the creation of every kind of model found in typical software engineering practice.

b) Process centred collaboration.

A software process model structures steps, roles and artifacts to create during software development. Typically, engineers reduce the amount of overhead coordination to initiate the project, tackling more quickly with the project at hand, rather than negotiating the entire project structure. Overtime, as experience grows, the net effect is to reduce the amount of coordination work required within a project by regularizing points of collaboration, as well as to increase predictability of future activity. Process centered software development environments have facilities for writing software process models in a process modeling language, then executing these models in the context of the environment. For example, the environment can manage the assignment of tasks to engineers, monitor their completion, and automatically invoke appropriate tools. Some examples of such systems are Arcadia [72], Oz [12], Marvel [13], ConversationBuilder [73], and Endeavours [17].

c) Collaboration awareness.

Software engineering is a human-driven and human-intensive activity. Most medium- to large-scale projects involve multiple software developers that may or may not be co-located. In recent years, there has been much work in developing collaborative development environments that provide support for coordination and communication during software development [66]. A key issue in any collaborative is awareness, or "knowing what is going on" [40]. More precisely, awareness is "an understanding of the activities of others, which provides a context for [one's] own activity" [33]. Awareness encompasses knowing who else is working on the project, what they are doing, which artifacts they are or were manipulating, and how their work may impact other work. In distributed collaborative work, maintaining awareness is considerably more difficult. Research areas ranging from software visualization to reverse engineering have been developing tools and techniques to provide awareness during software development. Seesoft [37], Palantir [116], Lighthouse [122] and Jazz [66] are but a few. A more extensive survey and comparison study can be found at [134].

d) Collaboration infrastructure.

Various infrastructure technologies make it possible for engineers to work collaboratively. Software tool integration technologies make it possible for software tools to coordinate their work. Major forms of tool integration include data integration (ensuring that tools can exchange data), control integration (ensuring that tools are aware of activities of other tools and can take action based on that knowledge). For example, nowadays, most IDEs know when a source-file is saved after editing and store it on a central repository (data integration) or SCM, then automatically call the proper compiler (control integration). Tools like Eclipse, Visual Studio, Marvel and WebDAV already implement these behaviors. Whether through calling other external tools based on the context of the task or coordinating between integrated tools, these environments already bring a sustainable collaboration between engineers and theirs development tasks.

2) Trends and future research directions

There are still several areas to be addressed for improving collaboration in software engineering, which may reveal the future trends on this domain of expertise.

a) Integrating web and desktop environments.

The migration of development tools to the web is increasing, now that the user interface is becoming more sophisticated (thanks to AJAX and its overall adoption) and the processing power of browsers is higher. UML and source code editing are no longer relegated only to desktop applications, whereas in the past, the web could not support such features. Despite this trend, there is a longstanding practice surrounding the use of integrated development environments (Visual Studio, Eclipse, JBuilder, etc.), which are not going to be displaced by completely web-based environments. Instead, future projects are likely to adopt a mixture of web-based and desktop tools, for which interfacing open standards between the desktop IDE's and the web-based services should be created. Although not an easy task, these open standards would allow a more seamless interaction with the complex information a software project creates.

b) Broader participation in design.

Currently, software customers are engaged in the development process during requirements elicitation, but then become not so engaged for the requirements analysis, design and coding phases, only to reconnect again for the final phase of testing. Broadened participation by customers in the requirements analysis, design, coding and early testing phases would keep them engaged during these middle stages, allowing a more actively assurance that their direct needs are met. By increasing the participation of the direct end users, software engineers can reduce the risk that the final product does not meet the needs of customer organizations. Surely, a balance between completely open-sourced projects and a fine-grained proprietary closed-source model available for the customer to refine has to be made. Nevertheless, a participatory development model would allow customers a better tailoring of the software to their needs. The trend toward providing support for distributed development teams in a wide range of development tools makes a broader engagement possible. Open source SCM tools like Subversion, as well as web-based requirements tools and problem tracking tools make it possible to coordinate globally distributed teams.

c) Capturing rationale argumentation.

One of the strongest design criteria used in software engineering is design for change, which inherently involves making predictions about the future. As a result, the design process is not just and engineer making rational decisions from a set of facts, but instead is a predictive process in which multiple engineers argue over current facts and future potentials. Architecture and design are argumentative processes in which engineers resolve differences of prediction and interpretation to develop models of a software system's structure. Since only one vision will prevail, the process of architecture and design is simultaneously cooperative and competitive. Providing collaborative tools to support engineers in the recording and visualization of architecture and design argumentation structures would do a better job of capturing the nuances and tradeoffs involved in creating large systems. They would also better convey the assumptions that went into a particular decision, making it easier for succeeding engineers to know when they can safely change a system's design.

d) Using novel communication and presence technologies.

Software engineers have a long track record of integrating new communication technologies into their development processes. Email, instant messaging and web-based applications are very commonly used in today's projects to coordinate work and be aware of whether other developers are currently active (present). As a result, engineers would be expected to adopt emerging communication and presence technologies if they offer advantages over current tools. For instance, networked collaborative 3d game worlds are such an emerging technology that spawned "software immersion environments". Second Life is an example of using such a 3D world to develop software, as their team uses its own platform to do so. There is a range of research issues inherent to the use of 3D virtual environments as a collaboration infrastructure, for example, how to synchronize physical and virtual worlds. Ultimately, the utility of adopting a 3D virtual world needs careful examination, as the benefits of the technology need to clearly exceed the costs. It is currently very unclear that this is true.

e) Improved assessment of collaboration technology.

Assessing the impact of the introduction of new technology into a project is difficult, and usually subjective. Estimation in software development is a difficult task, which hinders the objective assessment of collaboration technology. Without the uncovering of the pros and cons of specific collaboration tools, forward progress in the field of software collaboration support tools is hard to measure. There is a lack of studies how already introduced tools (instant messaging, Internet-aware SCM tools, email, bug tracking systems, etc.) that quantify the benefits received from using these collaboration tools. Developing improved methods for assessing the impact of collaboration tools would boost research in these areas by increasing confidence in positive results, and making it easier to convince teams to adopt new technologies.

V. OPEN ISSUES

Program Comprehension deals with understanding programs and software artefacts. Framework Understanding focuses on a specific kind of software artefact: a framework. This understanding is often made resorting only to information on the artefact itself and accompanying documentation. More and more, software is developed collaboratively. Can this "collaboration" help in framework understanding?

From the state-of-the-art review, a number of open research issues arise. An insight of the most relevant ones follows, as a means to focus the reader to the intended scope:

a) Frameworks are often hard to understand and use.

The difficulty of understanding frameworks is a serious inhibitor of effective framework reuse. This is mainly due to framework designs being usually very complex, and thus hard to communicate. The framework design is: (1) very abstract, to factor out commonality; (2) incomplete, requiring additional classes to create a working application; (3) more flexible than needed by the application at hand; (4) obscure, in the sense that it usually hides existing dependencies and interactions between classes. The learning curve becomes steep, requiring a considerable amount of effort to understand and learn how to use a framework.

b) Framework documentation is often outdated and inaccurate.

Good documentation significantly improves the process of learning and understanding new frameworks. By guiding users on the customization process and by explicitly showing the framework design principles and details, effective documentation contributes to make frameworks easy to reuse. Despite these reasons, framework documentation is still regarded as of low importance within the framework development process. Most commonly during maintenance or evolution phases, documentation is used to assist on these tasks but its update is often discarded or neglected. Moreover, it is still hard, costly and tiresome to define and write good quality documentation for a framework. Good documentation should be easy to use, support different audiences and provides multiple views through different types of documents and notations. The difficulty of producing contents for these requirements may hinder its applicability and demotes its importance within the development process.

c) Programmers (both experts and novices) recurrently tackle with understanding problems.

Every time a software developer needs to re-use a piece of code, whether it's a snippet, class, library or framework, she goes over the entire cognitive process of analyzing, understanding and capturing the relevant information she needs. Depending on the purpose of the task at hand (learning, teaching, communicating, using), the format (quality, clarity, structure, abstraction level, etc.) of the code, and the experience of the programmer (expert or novice) the understanding process may go through various approaches (top-down, bottom-up, etc.), not always leading to the desired outcome in a straight forward manner. Choosing the adequate understanding process should not be difficult, and changing from one to another should be feasible without much overhead.

d) The process of understanding a framework is not properly dealt with.

The palette of tools available to the framework learner scarcely deals with specific aspects of framework understanding. Without questioning its local and highly focused solutions, each tool aids in a specific aspect, whether capturing high-level design artifacts, browsing the code for hot-spots, or helping on producing sustainable output formats. Alas, the framework user has to navigate through a plethora of tools trying to figure out where the relevant information might be.

e) Different tools provide sparse results with variable quality.

By itself, each tool has its own problems and limitations, thus producing quality-questionable results. For instance, many of the problems design recovery (reverse engineering) tools have, tend to converge to selection of results (elimination of false positives) and semantic overlapping (same result can have several meanings). With such discrepancy amongst results, it becomes difficult to ascertain tool efficiency and compare results regarding precision and recall.

f) Collective knowledge of the development team is often not harnessed at its best.

Software development is a highly social process. It has been perceived that, when trying to understand a piece of code, developers turn first to the code itself and, when that fails, to their social network, that is, the team. This behavior, not only happens during code understanding, but also throughout the whole understanding process. Nevertheless, it is not easy to go for the team. Firstly, it is not clear who to address for clarification, for there is a lack of awareness of what other members of the team are doing or how do they relate to the work done. Secondly, the fields of expertise are not clear or stated, leading to wasteful interruptions of the wrong people. Thirdly and most often, the team or the experts are not available for consulting or rebuke their fellow colleagues due to interruption. Interrupted developers lose track of parts of their mental model, resulting in laborious reconstruction or bugs and discouraging more frequent interruptions.

g) Implicit developers' knowledge is not captured and shared as effectively as it could be if well supported.

Developers go to great lengths to create and maintain rich mental models of design and code that are rarely permanently recorded. Very often, developers, without referencing written material, can talk in detail about their product's architecture, how the architecture is implemented, who owns what parts, the history of the code, to-dos, wish-lists, and meta-information about the code. For the most part this knowledge is never written down, except in transient forms such as sketches on a whiteboard. The bottom-line problem here is that "Lots of [useful] information is kept in peoples' heads" [80].

VI. KEY RESEARCH QUESTIONS

From the open issues presented before, a few research questions revolve around a major question that is considered central to this research work:

1) How to improve framework understanding?

a) What kind of information do developers try to capture first? What makes them decide?

b) What are the actual goals of the framework learner?

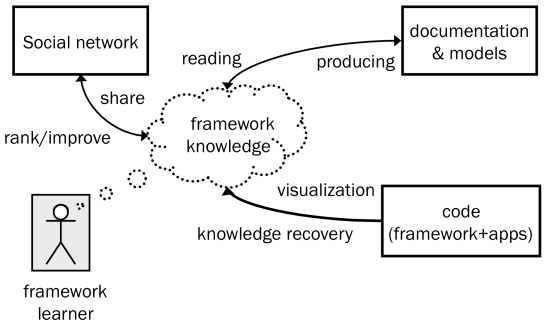

Figure 6 - Framework learning environment

c) Are there any typical and repeated behaviours developers apply when trying to learn how to use a framework?

d) How can tools assist the learning process?

e) What kind of information is presented to framework learners that they mostly look for? What do they look for that isn't there?

f) What is missing from existing development environments to assist on framework understanding?

In this paper, the authors address mainly questions *d)* and *f)*, and believe that to improve framework understanding, tools should be collaborative and specific knowledge should be captured and presented to the developers. The next section will address how they intend to pursue that.

VII. IMPROVING THROUGH COLLABORATION

Teams collaborate to develop software. But not all of the relevant knowledge is recorded for later use. Developers tackle recurrently with understanding and learning issues, especially if teams rotate their members often. Team members take tacit knowledge with them that decays with time and that proves useful later on. That knowledge, if permanent and available, could save time when dealing again with the system. What if that knowledge could be shared with other developers, novice or expert? The idea is to make that knowledge available within the development environment. Therefore, (re) learning about the system (frameworks, in this case) should benefit from knowing how its was learnt in the first place.

A. *Supporting the learning process*

Learning how to use a framework is not a trivial task. The learner is usually engaged in a *process* composed of a series of activities. This process has *best practices* that can be followed to improve its outcome. These practices could be actively applied and improved having *tools* to support them. These three levels are detailed next and depicted in Figure 6.

1) Process

In this particular domain, there is range of activities that may characterize the developer's behavior while trying to understand a framework. These activities may fall into three categories:

a) Code

Figure 5 - Learning environment support levels

"Where is all that we need to know"(?) The problem is that what we need to know is not explicitly in front of us. Furthermore, frameworks make it particularly difficult to find what we need to know. As an example, recovering design knowledge implicit in the code is a recurring practice to help clarify the framework's structure and purpose. The questions reside on what kind of design artifacts, to what kind of audience, and how to store and present the results so that they are useful.

b) Documentation.

When the developer wants to learn how to use a framework (or any reusable software artifact, for that matter), she goes for the documentation, if it exists. But, is there always documentation? And is that documentation clear, well suited and complete? Does it have all the answers? There are known ways of producing good documentation for frameworks [6][7][8]. The issue is nurturing the developers to easily produce and access that documentation, even during the learning process.

c) Social network.

When all else fails, the developer loses her self-sufficiency as a learner and resorts to her "contacts", that is, strong candidates to bear knowledge that might help her. Call it team, peers, social network, buddies or any other term, there is knowledge that one can't find anywhere else but on people's minds. It is called intrinsic knowledge. Getting this knowledge is intrusive. There should be ways of harnessing this knowledge without such intrusiveness.

Putting it short, a framework learner looks at the code, reads the documentation, visualizes information and asks her colleagues for help, as going through a learning process of understanding how to use the framework. Figure 5 (extracted from [5]) depicts this scenario.

2) Best Practices

Associated with the learning process, there is a series of good practices on how to deal with each stage of the learning process. These are presented in [45]. A learning environment should support and nurture these practices.

3) Supporting Tools

Depending on several factors (learner's experience, existing artifacts, learning goal, etc.) the learning process to undertake may resort to different practices and paths. What works for some, might not work for others, and may even vary between frameworks. Novices and experts will take different paths.

Yet, in a truly collaborative environment, where, at first, there is no distinction between who is expert and who is

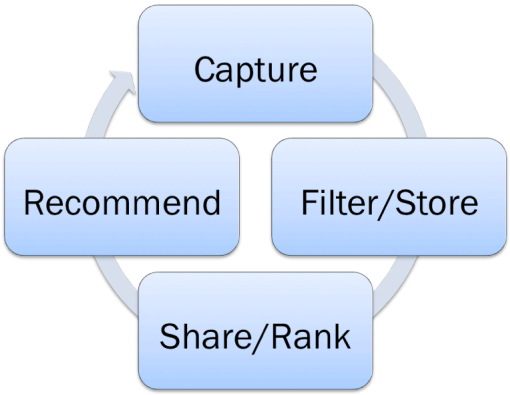

Figure 7 - Supporting steps to improve the learning process

novice, sharing experiences and advising the global community proves useful [135]. The importance given to an advice or counsel is measured by its actual applicability. You became experienced and expert by giving valid and helpful feedback into the community.

By supporting this sharing of knowledge, the learners may benefit from their collective intelligence, thus improving their own learning processes. Therefore, the supporting tools should be prepared to capture this knowledge, share it and assist other learners in their tasks.

B. *Tool requirements*

Teams turn into communities mainly due to high member rotation, high project preemption and the widely spread of frameworks. The strength of this community relies mainly on its ability to withhold valuable knowledge, filtering out what its not important. The issue is providing an effective infrastructure to share this information amongst its members without too much effort and to allow a "natural" selection of what is actually valuable.

Providing such an environment would have the following requirements:

a) Seamless integration into a IDE.

Tools and features to support the learning process should be available within the development environment as a means to enforce usage, without disrupting the normal way a developer works. When presented with possible solutions, it should be straightforward how to proceed within the IDE to apply those solutions.

b) Non-intrusive / non-interruptive.

Ideally, capturing the developer's intrinsic knowledge should be implicit. That is, the developer should not be asked to explicitly provide any information regarding that knowledge to the system. In practice, a satisfactory solution would be to notify the system we are trying to learn how to do some task and signal reaching that goal. Bottom-line, it should be as non-intrusive as possible.

c) Context-aware.

The tools should be aware of the context where the developer is learning and provide information that makes sense within that context.

d) Web-aware.

Not only should the environment seek knowledge within its own boundaries (its knowledge-base), it should also be prepared to go to the web in a contextual manner.

e) Descriptive, not prescriptive.

The system should not tell the developer how to proceed, but instead should give possible directions on how to solve the task at hand.

f) Shared knowledge-base.

The environment should store and share all the relevant knowledge that helps the framework learning process. Not only the documentation artifacts and source-code, but the captured knowledge that helps guiding the developer throughout the process.

g) Learning Path

Different developers learn in different ways. The environment should be able to deal with the learning profile of its users, considering aspects such as visualization of information and easy personalization of contents.

The learning process would be supported relying on a four-step cycle shown in Figure 7. The purpose would be to capture the learning steps taken by the learner. Whether she looks at the code first, goes for documentation, explores certain artifacts, and recovers others, until she reaches a satisfying conclusion. This path would then be recorded, stored and shared. "Sharing" means that other learners may reuse it or get assistance through it to guide them on their own learning path. If the shared knowledge really helped them, then they should rank it or improve it. As the collected knowledge keeps improving (through sharing, usage and ranking), the best learning strategies will be recommended to recurrent learners and thus improving their learning process.

Candidate existing environments are Eclipse, Jazz Team Concert and Visual Studio due to their extensible nature and pluggable architecture. These environments have a notion of context or process-awareness, yet they miss the learning context. The idea would be to insert the notion of a learning process and provide tools to assist in that process, supporting the steps depicted in Figure 7. The tools should be built as plug-ins to the collaborative environment, introducing a learning context and accompanying the learner throughout the process.

VIII. CONCLUSION

Frameworks are good software artifacts for reuse. Nevertheless they are complex, thus hard to learn. Most of the tools that may help in this task don't encompass the social nature of software development. In distress, learners tend to look for help at their colleagues, often disrupting their work. Supporting the social side of software development by raising awareness and capturing intrinsic knowledge helps improving the learning of software, namely, frameworks.

A set of requirements for tools to harness framework learning knowledge and assisting in the process of learning should: allow for seamless integration into an IDE; be non intrusive or interruptive; be context and web aware; be

descriptive instead of prescriptive; share a common knowledge base of evolving learning knowledge and capture the learning path taken by developers.

Providing a collaborative environment where learning knowledge can be captured, shared, ranked and recommended to recurrent learners, both expert and novice, in a non-intrusive way, aims at improving framework understanding.

REFERENCES

[1] N.Flores, "Patterns and Tools for improving Framework Understanding: a Collaborative Approach", SEDES Doctoral Symposium, ICSEA - The Fourth International Conference on Software Engineering Advances, Porto, Portugal, September 2009.

[2] M. Ackerman, (2000). "The Intellectual Challenge of CSCW: The gap between social requirements and technical feasibility". Human-Computer Interaction 15: 179–203

[3] M. S. Ackerman and D. W. McDonald (2000), "Collaborative Support for Informal Information in Collective Memory Systems", in Information Systems Frontiers, vol. 2, o. 3/4, pp. 333-347, 2000.

[4] Adoption-Centric Software Engineering Project site. Computer Science Department at University of Victoria, Canada URL:http://www.acse.cs.uvic.ca/index.html. Accessed at 30-06-2010

[5] A.Aguiar, "Framework Documentation – A Minimalist Approach", PhD Thesis, FEUP September 2003.

[6] A.Aguiar and G.David, "Patterns for Documenting Frameworks – Part III", PLoP'2006, Portland, Oregon, USA, October 2006.

[7] A.Aguiar and G.David, "Patterns for Documenting Frameworks – Part II", EuroPLoP'2006, Irsee, Germany, July 2006.

[8] A.Aguiar and G.David, "Patterns for Documenting Frameworks – Part I", VikingPLoP'2005, Helsinki, Finland, September 2005.

[9] R.Baecker and A.Marcus, "Human Factors and Typography for More Readable Programs". ACM Press, Addison-Wesley Publishing Company, 1990

[10] R.M. Baecker et al., (1995). "Readings in human-computer interaction: toward the year 2000". Morgan Kaufmann Publishers.

[11] T. Ball and S.G. Eick, "Software visualization in the large", IEEE Computer, 29, 4, pp.33-43, 1996

[12] I. Z. Ben-Shaul (1994), "Oz: A Decentralized Process Centered Environment (PhD Thesis)," in Department of Computer Science: Columbia University, Dec 1994.

[13] I. Z. Ben-Shaul, G. E. Kaiser, and G. T. Heineman (1992), "An Architecture for Multi-user Software Development Environments," in ACM SIGSOFT 92: 5th Symposium on Software Development Environments, Tyson's Corner, Virginia, 1992, pp. 149-158.

[14] C.F. Bertholf and J.Scholtz, "Program Comprehension of Literate Programs by Novice Programmers", Empirical Studies of Programmers: 5th Workshop, 1993

[15] T.J.Biggerstaff, B.W Mitbander, and D.Webster, "The concept assignment problem in program understanding", Proceedings of the 15th International conference on Software Engineering, pp.482-498, 1993.

[16] B. Boehm and A. Egyed (1998), "Software Requirements Negotiation: Some Lessons Learned", in the 20th International Conference on Software Engineering (ICSE'98), Japan, 1998, pp.503-507

[17] G. A. Bolcer and R. N. Taylor (1996), "Endeavors: a Process System Integration Infrastructure," in 4th International Conference on the Software Process (ICSP'96), Brighton, UK, 1996, pp. 76-89.

[18] J.Bosch, P.Molin, M.Mattsson, and P.O. Bengtsson, "Framework – Problems and Experiences" Building Application Frameworks, M.Fayad, D.Schmidt, R.Johnson, Wiley, 1999.

[19] M.S.K.Brade, M.Guzdial, and E.Soloway, "Whorf: A visualization tool for software maintenance". Proceedings 1992 IEEE Workshop on Visual Languages, pp.148-154, 1992

[20] J.M.Brant, "HOTDRAW", MsC Thesis, University of Illinois, 1995

[21] R. Brooks, "Towards a theory of the comprehension of computer programs", International Journal of Man-Machine Studies, pp. 543-554, vol.18, 1983.

[22] M.H.Brown and R.Brooks, "ZEUS: A system for algorithm animation and multi-view editing". Proceedings of the IEEE 1991 Workshop on Visual Languages, pp.4-9, 1991

[23] M.Bruch, T.Schäfer, and M.Mezini, "FrUiT: IDE Support for Framework Understanding", OOPSLA Eclipse Technology Exchange, 2006

[24] R.I.Bull and M-A. Storey, "Towards Visualization Support for the Eclipse Modeling Framework", A Research-Industry Technology Exchange at EclipseCon, 2005.

[25] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. "Pattern oriented software architecture - a system of patterns". John Wiley and Sons, 1996.

[26] G.Butler, R.Keller, and H.Milli, "A Framework for Framework Documentation", Computing Surveys, special Symposium Issue on Object-Oriented .Application Framework, ACM, 2000.

[27] T.Cheng, S.Hupfer, S.Ross, and J.Patterson (2007). "Social Software Development Environments", Dr.Dobb's Journal. Janury 11th , 2007.

[28] K. Coleman (1995), "Groupware technology and Applications", Prentice Hall PTR 1995

[29] J.R. Cordy, T.R. Dean, A.J.Malton, and K.A. Schneider, "Source Transformation in Software Engineering using the TXL Transformation System", Journal of Information and Software Technology, vol(44)13, pp. 827-837, 2002.

[30] D.Cubranic, G.C. Murphy, J.Singer, and K.S.Booth, " Hipikat: A project memory for software development", IEEE Transactions on Software Engineering 31, 6 (Jun. 2005), pp. 446-465.

[31] R. DeLine, A.Khella, M.Czerwinski, and G.Robertson, "Towards Understanding Programs through Wear-based Filtering", Softvis, 2005

[32] F. Détienne, "Software Design – Cognitive Aspects", Springer Practitioner Series 2001.

[33] P. Dourish and V. Bellotti, (1992). "Awareness and coordination in shared workspaces". Proceedings of the 1992 ACM conference on Computer-supported cooperative work: 107-114, ACM Press New York, NY, USA.

[34] P. Dourish, (2003). "The Appropriation of Interactive Technologies: Some Lessons from Placeless Documents". *Computer Supported Cooperative Work* **12**: 465–490. Kluwer Academic Publishers.

[35] L. Dusseault (2003), WebDAV: Next-Generation Collaborative Web Authoring, Prentice Hall PTR, 2003.

[36] Eclipse project site. URL: http://www.eclipse.org. Accessed at 30-06-2010.

[37] S. G. Eick, J. L. Steffen and, E. E. Sumner Jr. (1992) "SeeSoft – a tool for visualizing line oriented software statistics." IEEE Transactions on Software Engineering 28, 4, 396-412.

[38] T.Eisenbarth, R. Koschke, and D. Simon, "Aiding Program Comprehension by Static and Dynamic Feature Analysis", Proceedings of the IEEE International Conference on Software Maintenance, 2001

[39] C. A. Ellis, S. J. Gibbs, and G. L. Rein (1993), "Groupware some issues and experiences". In: Baecker, Ronald M. Readings in groupware and computer-supported cooperative work. San Francisco: Morgan Kaufmann, 1993. p. 9-28.

[40] M. Endsley (1995), "Toward a theory of situation awareness in dynamic systems". Human Factors 37, 1, 32-64.

[41] K. Erdös, and H.M. Sneed, "Partial Comprehension of Complex Programs (enough to perform maintenance)", Proceedings of the 6th

International Workshop on Program Comprehension, pp. 98-105, 1998.

[42] G.Fairbanks, D.Garlan and W. Scherlis, "Design Fragments Make Using Frameworks Easier", OOPSLA 2006 .

[43] M.Fayad, D.Schimdt, and R.Johnson, "Building Application Frameworks", Wiley 1999.

[44] N.Flores and A.Aguiar, "JFREEDOM: a Reverse Engineering Tool to Recover Framework Design", Proceedings of the 1st International Workshop on Object-Oriented Reengineering, ECOOP'05, 2005.

[45] N. Flores and A.Aguiar, (2008) "Patterns for Framework Understanding", in Proc. of th 15th Pattern Languages of Programming Conference (PLoP'08).

[46] M.Fontoura, W.Pree, and B.Rumpe, "The UML Profile for Framework Architectures", Addison-Wesley Professional 2001.

[47] J. Froehlich and P. Dourich, "Unifying artifacts and activities in a visual tool for distributed software development teams", Proceedings of the 26th International Conference on Software Engineering, pp. 387-396, 2004.

[48] G. Froehlich, H.Hoover, L.Lui, and P.Sorenson, "Hooking into Object-Oriented Application Frameworks", Proceedings of the 19th International Conference on Software Engineering, pp.491-501, 1997

[49] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. "Design Patterns — Elements of reusable object-oriented software". Addison-Wesley, 1995.

[50] D.Gangopadhyay and S.Mitra, "Understanding Frameworks by Exploration of Exemplars", Proceedings of CASE-95, IEEE Computer Society, pp. 90-99, 1995

[51] J.J.Garrett. "Ajax: A New Approach to Web Applications". Adaptive Path, February 18th 2005. URL: http://www.adaptivepath.com/publications/essays/archives/000385.php. Accessed at 30-06-2010.

[52] D. M. German, "Decentralized open source global software development, the GNOME experience," Journal of Software Process: Improvement and Practice, vol. 8,no. 4, pp. 201–215, 2004.

[53] T.R.G. Green and M. Petre, "When Visual Programs are Harder to Read than Textual Programs", Human-Computer Interaction: Tasks and Organization, Proceedings (ECCE)-6 (6th European Conference Cognitive Ergonomics), 1992

[54] S. Greenberg (1991), "Computer-supported Co-operative Work and Groupware", Academic Press Ltd., London, 1991.

[55] R. Grinter, (1999), "Systems Architecture: Product Designing and Social Engineering," in ACM Conference on Work Activities Coordination and Collaboration (WACC'99), San Francisco, California, 1999, pp. 11-18.

[56] J. Grudin, (1988). "Why CSCW applications fail: problems in the design and evaluation of organization of organizational interfaces". Proceedings of the 1988 ACM conference on Computer-supported cooperative work: 85-93, ACM Press New York, NY, USA.

[57] J. Grudin, (1994), "Computer-Supported Co-operative Work: History and Focus", Computer, Vol. 27, No. 5, May 1994

[58] C. Gutwim, R.Penner, and K. Schneider, "Group Awareness in Distributed Software Development", ACM CSCW, pp. 72 – 81, 2004

[59] M.Hakala, J. Hautamäki, K.Koskimies, J.Paakki, A.Viljamaa, and J.Viljamaa, "Annotating reusable software architectures with specialization patterns", Proceedings of the Working IEEE/IFIPConference on Software Architecture (WICSA'01), pp. 171, 2001

[60] I.Hammouda and K.Koskimies, "A pattern-based j2ee application development environment", Nordic Journal of Computing 9(3), pp. 248-260, 2002

[61] J.Hannemann and G. Kiczales, "Design pattern implementation in java and aspectj", Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages and applications, pp.161-173, 2002.

[62] R.Helm, I.Holland, and D.Gangopadhyay, "Contracts: specifying behavioral compositions in object-oriented systems", Proceedings of the European conference on object-oriented programming (ECOOP'90), pp. 169-180, 1990

[63] T. Hendrix, J.H. Cross II, L. Barowski, and K.Mathias. "Tool support for reverse engineering multi-lingual software". Proceedings of the 4th Working Conference on Reverse Engineering (WCRE'97), pp. 136-143, 1997

[64] L. Hohmann, "Journey of the Software Professional: The Sociology of Software Development", 1996.

[65] D.Hou and H.J. Hoover, "Towards specifying constraints for object-oriented frameworks", Proceedings of the 2001 conference of the Centre for Advanced Studies on Collaborative research, page 5, IBM Press, 2001.

[66] S. Hupfer, L.-T Cheng, S. Ross, and J. Patterson, "Introducing collaboration into an application development environment", Proceedings of the ACM Conference on Computer Supported Cooperative Work, pp. 444-454, 2004.

[67] Imagix 4D. Imagix Corporation. URL: http://www.imagix .com/index.html.

[68] S.Isoda, T.Shimomura, and Y. Ono, "VIPS: A visual debugger", IEEE Software, May 1987

[69] R. Johansen (1988) "Groupware: Computer Support for Business Teams" The Free Press.

[70] R.Johnson, "Documenting Frameworks using Patterns", Proceedings of the OOPSLA'92, SIGPLAN notices, 27(10), pp.63-76. 1992

[71] R.Johnson, "Components, framework, patterns. SIGSOFT Software Engineering Notes, 22(3), pp. 10-17, 1997

[72] R. Kadia (1992), "Issues Encountered in Building a Flexible Software Development Environment," in ACM SIGSOFT 92: 5th Symposium on Software Development Environments, Tyson's Corner, Virginia, 1992, pp. 169-180.

[73] S. M. Kaplan, W. J. Tolone, A. M. Carroll, D. P. Bogia, and C. Bignoli (1992), "Supporting Collaborative Software Development with ConversationBuilder," in ACM SIGSOFT 92: 5th Symposium on Software Development Environments, Tyson's Corner, Virginia, 1992, pp. 11-20.

[74] M.Kersten and G.Murphy, "Mylar: a degree-of-interest model for IDE's", International Conference on Aspect Oriented Software Development, pp.159-168, 2005

[75] D.Kirk, M.Roper, and M.Wood, "Identifying and Addressing Problems in Framework Reuse", Proceedings of the 13th International Workshop on Program Comprehension (IPWC'05), pp. 77-86, 2005

[76] G. Kizcales, J.Lamping, A.Mendhekar, C.Maeda, C.V.Lopes, J-M Loingtier, and J. Irwin. "Aspect Oriented Programming", Proceedings of the European Conference on Object-Oriented Programming (ECOOP), June 1997.

[77] R. Kling (1991) "Co-operation, Co-ordination and Control in Computer-Supported Work", CACM, vol. 34, no. 12, December 1991.

[78] G.E.Krasner, S.T.Pope,"A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80", Journal of Object-Oriented Programming 1,3, pp. 26-49, 1988

[79] M.Lanza and S.Ducasse, "A Categorization of Classes based on Visualization of their Internal Structure: the Class Blueprint".Proceedings for OOPSLA 2001, pp.300-311, ACM Press, 2001

[80] T.D.LaToza, G.Venolia, and R.DeLine (2006), "Maintaining Mental Models: A Study of Developer Working Habits". Proc. of the International Conference of Software Engineering (ICSE'06), Shanghai, China.

[81] B. S. Lerner, L. J. Osterweil, Stanley M. Sutton Jr., and A. Wise (1998), "Programming Process Coordination in Little-JIL Toward the Harmonious Functioning of Parts for Effective Results," in European Workshop on Software Process Technology, 1998.

[82] S.Letovsky, "Cognitive processes in program comprehension", Empirical Studies of Programmers, pp.58-79, 1986

[83] P.Linos, P.Aubet, L.Dumas, Y.Helleboid, P.Lejeune, and P.Tulula. "Visualizing program dependencies: An experimental study" Software-Practice and Experience, 24(4):387-403, 1994

[84] D.C Littman, J.Pinto, S.Letovsky, and E.Soloway, "Mental models and software maintenance", Empirical Studies of Programmers, pp. 80-98, 1986

[85] T. W. Malone and K. Crowston (1994), "The Interdisciplinary Study of Coordination," in ACM Computing Surveys (CSUR), vol 26, no 1, pp. 87-119, 1994.

[86] D. Mandelin, L. Xu, R. Bodik and D.Kimelman, "Mining Jungloids: Helping to Navigate the API Jungle", Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, pp 48-61, 2005.

[87] A. Marcus, L. Feng, J.I. Maletic, "Comprehension of Software Analysis Data Using 3D Visualization", Proceedings of the IEEE International Workshop on Program Comprehension (IWPC2003), pp.105-114, 2003

[88] A. von Maryhauser and A. Vans. "Program comprehension during software maintenance and evolution" IEEE Computer pp. 44-55, August 1995.

[89] A. von Maryhauser and A. Vans. "From code understanding needs to reverse engineering tool capabilities" Proceedings of CASE'93, pp. 230-239, 1993.

[90] S. McConnell (1996), "Lifecycle Planning," in Rapid Development: Taming Wild Software Schedules Redmond, WA: Microsoft Press, 1996.

[91] A.Mendelson and J.Sametinger. "Reverse Engineering by visualizing and querying", Software – Concepts and Tools, 16:170-182, 1995

[92] Microsoft Corporation (2007), "Microsoft Office Project Standard 2007 Product Guide," April 2006, Available at: http://download.microsoft.com/download/d/f/b/dfb0e645-3e5a-4fe8-8ea6-1c9e86d6139a/ProjectStandard2007ProductGuide.doc. Accessed at 30-06-2010

[93] Microsoft Visual Studio (2008) website. http://msdn.microsoft.com/en-us/vstudio/default.aspx. Accessed on 30-06-2010

[94] A.Mockus, R.Fielding, and J.D. Herbsleb, "Two Case Studies of Open Source Software Development: Apache and Mozilla", ACM Transactions of Software Engineering and Methodology, 11, 3, pp.309-346, 2002

[95] S.Moser and O. Nierstrasz, The Effect of Object-Oriented Frameworks on Productivity", IEEE Computer, pp.45-51, 1996

[96] H.Muller and K. Klashinsky. "Rigi – A system for programming-in-the-large", Proceedings of the 10th International Conference on Software Engineering (ICSE'10), pp.80-86, 1988.

[97] G.C. Murphy, D.Notkin, and K.Sullivan, "Software Reflexion Models: Bridging the Gap Between Source and High-Level Models", Proceedings of Foundations of Software Engineering, pp. 18-28, 1995.

[98] A.Murray and T.Lethbridge, "On Generating Cognitive Patterns of Software Comprehension", Proceedings of the 2005 conference of the Centre for Advanced Studies on Collaborative research, pp.200-211, 2005.

[99] NATO, Software Engineering Conference, Garmisch, Germany, 7-11 October 1968.

[100] J. F. Nunamaker, R. O. Briggs, and D. D. Mittleman (1995). "Electronic meeting systems: Ten years of lessons learned." In D. Coleman, & R. Khanna (Eds.), Groupware: Technology and Applications (pp. 149–192). Englewood Cliffs, NJ: Prentice-Hall, 1995.

[101] C. O'Reilly, D.Bustard, and P.Morrow, "The War Room Command Console (Shared Visualizations for Inclusive Team Coordination)", Softvis, 2005.

[102] N.Pennington, "Stimulus structures and mental representations in expert comprehension of computer programs", Cognitive Psychology, pp. 295-341, vol 19, 1987.

[103] D.A. Penny, "The Software Landscape: A Visual Formalism for Programming-in-the-Large", PhD Thesis, University of Toronto, 1992.

[104] M.Petre, A. Blackwell, and T.Green, "Cognitive questions in software visualization". Software Visualization: Programming as a Multi-Media Experience, pp. 453-480. MIT Press, 1997

[105] M.Petre, "Why looking isn't always seeing: readership skills and graphical programming", Communications of the ACM, vol. 38, pp. 33-44, 1995

[106] W.Pree, "Design Patterns for Object-Oriented Software Development", Addison-Wesley, 1994

[107] V.Rajlich, N.Damskinos, and P.Linos, "VIFOR: A tool for software maintenance". Software-Practice and Experience, 20(1):67-77, 1990

[108] Rational Software Corporation (2003), "Rational RequisitePro User's Guide," June 2003, http://www.ibm.com/developerworks/rational/library/content/03July/getstart/RP/ReqProMain.pdf. Accessed on 30-06-2010.

[109] W. Reinhard, J. Schweitzer, G. Volksen, and M. Weber, (1994) "CSCW tools: concepts and architectures," Computer , vol.27, no.5, pp.28-36, May 1994

[110] S.Reiss, "Pecan: Program development systems that support multiple views", IEEE Transactions on Software Engineering, SE-11(3): 276-285, 1985

[111] S.Reiss, "An overview of BLOOM", Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program analysis for software tools and engineering, pp.2-5, 2001.

[112] D.Riehle, "Framework Design: A Role Modelling Approach", PhD Thesis, Swiss Federal Institute of Technology, 2000

[113] R.S. Rist, "plans in programming: Definition, Demonstration, and Development", Empirical Studies of Programmers, 1st Workshop, 1986.

[114] M.P. Robillard and G.Murphy, "FEAT: A tool for locating, describing, and analyzing concerns in source code", Proceedings of the 25th International Conference on Software Engineering, pp. 822-823, 2003

[115] G.-C.Roman, K.C. Cox, C.D. Wilcox, and J.Y.Plun. "Pavane: A system for declarative visualization of concurrent computations". Techinical Report WUCS-91-26, Washington University, St. Louis, 1991.

[116] A. Sarma, Z. Noroozi, and A. van der Hoek (2003). "Palantír – raising awareness among configuration management workspaces". In Proc. Of the 25th International Conference on Software Engineering, 444-454

[117] D.A. Scanlan, "Structured flowcharts outperform pseudocode: An experimental comparison", IEEE Trans. Soft. Eng., 1989

[118] F.Schull, F.Lanubile, and V.Basil, "Investigating Reading Techniques for Object-Oriented Framework Learning", IEEE TSE, vol.26, nº.11, 2000

[119] B. Shneiderman and R.Mayer, "Syntatic/semantic interactions in programmer behavior: A model and experimental results". International Journal of Computer and Information Science, pp. 219-238, 8(3), 1979

[120] B. Shneiderman, "Measuring computer program quality and comprehension", InternationalJournal of Man-Machine Studies, vol. 9, pp. 465-478, 1977

[121] B. Shneiderman, R.Mayer, D.McKay, and P.Heller, "Experiment investigations of the utility of detailed flowcharts in programming", Communications of the ACM, vol. 20, pp. 373-381, 1977

[122] I. A. da Silva, P. H. Chen, C. V. der Westhuizen, R. M. Ripley and A. van der Hoek (2006)" Lighthouse: Coordination through Emerging Design".In Proc. of the 2006 OOPSLA workshop on eclipse technology eXchange.

[123] J. Singer, T.Lethbridge, N.Vinson, and N. Anquetil, "An Examination of Software Engineering Work Practices", Proceedings of CASCON'97, pp. 209-233, 1997

[124] J. Singer, R. Elves, and M-A. Storey, "NavTracks Demonstration: Supporting Navigation in Software Space", International Workshop on Program Comprehension, 2005

[125] P.Schorn, A. Brungger, and M. de Lorenzi, "The XYZ Geobench: Animation of geometric algorithms. Animations for Geometric Algorithms: A Video Review, Digital Systems Research Center, Palo Alto, California, 1992

[126] K. Schmidt and L. Bannon, (1992). "Taking CSCW seriously". Computer Supported Cooperative Work 1: 7–40.

[127] E. Soloway, J.Pinto, S.Letovsky, D.Littman, and R.Lampert, "Designing documentation to compensate for delocalized plans", Communication of the ACM, 31(11): 1259-1267, 1988.

[128] E. Soloway and K. Erlich, "Empirical studies of programming knowledge", IEEE Transactions on Software Engineering, pp. 595-609, SE-10(5), September 1984.

[129] M-A Storey, "Theories, Methods and Tools in Program Comprehension: Past, Present and Future." Proceedings of the 13th IEEE International Workshop on Program Comprehension (IWPC). St. Louis, MO, pp. 181-191, IEEE Computer Society Press 2005.

[130] M-A Storey, F.Fracchia, and H.Muller. "Cognitive design elements to support the construction of a mental model during software visualization". Proceedings of the 5th International Workshop on Program Comprehension (IWPC'97), Dearborn, Michigan, pp. 17-28, May, 1997

[131] M-A Storey, F.Fracchia, and S.Carpendale, "A top down approach to algorithm animation" Techincal Report CMPT 94-05, Simon Frasier University, Brunaby, B.C., Canada, 1994

[132] M-A Storey, "Designing a Software Exploration Tool Using a Cognitive Framework of Design Elements", Software Visualization, 2003.

[133] M-A Storey, J. Michaud, M. Mindel, M. Sanseverino, D. Damian, D,Myers, D.Geman and E.Hargreaves, "Improving the Usability of Eclipse for Novice Programmers", Eclipse Technology eXchange (eTX) Workshop at OOPSLA 2003.

[134] M-A. D. Storey, D. Cubranic, and D.M. German (2005), "On the Use of Visualization to Support Awareness of Human Activities in Software Development: A Survey and a Framework". In Proc. of the 2005 ACM symposium on Software Visualization, 193-202.

[135] J. Surowiecki, (2004) "The Wisdom of Crowds: Why the Many Are Smarter Than the Few and How Collective Wisdom Shapes Business, Economies, Societies and Nations", Anchor Publishing.

[136] Rational DOORS, IBM (2008), Nov. 12, 2008. Available at: http://www-01.ibm.com/software/awdtools/doors/productline/. Accessed at 30-06-2010

[137] The Bugzilla Team (2008) , "The Bugzilla Guide - 3.3 Development Release". Mar 5, 2008. Available at: http://www.bugzilla.org/docs/tip/en/pdf/Bugzilla-Guide.pdf. Accessed at 30-06-2010

[138] W.F. Tichy, N. Habermann, and L. Pretchelt (1993). "Summary of the dagstuhl workshop on future directions in software engineering: February 17-21, 1992, schlo dagstuhl. ACM SIGSOFT Software Engineering Notes, 18(1):35-48

[139] S.Tilley and D.B.Smith, "Coming Attractions in Program Understanding", Technical Report CMU/SEI-96-TR-019, 1996

[140] S.Tilley, S.Paul, and D.Smith. "Towards a framework for program understanding". WPC'96: 4th Workshop on Program Comprehension, Berlin, Germany, pp. 19-28, March 1996

[141] T.Tourwé, "Automated Support for Framework-Based Software Evolution", PhD Thesis, Vrije Universiteit, 2002

[142] T.Tourwé and T.Mens, "Automated Support for Framework-Based Software Evolution", Proceedings of the International Conference on Software Maintenance, page 148, 2003

[143] I.Vessey, "Expertise in debugging computer programs: A process analysis", International Journal of Man-Machine Studies, pp. 459-494, vol 23, 1985.

[144] L. Wakeman and J. Jowett (2005), "PCTE: The Standard for Open Repositories": Prentice Hall, 1993. International Conference on Software Engineering Research, Management and Applications (SERA'05), Mount Pleasant, Michigan, USA, 2005, pp. 86-93.

[145] A. Walenstein, "Observing and Measuring Cognitive Support: Steps Toward Systematic Tool Evaluation and Engineering", 11th International Workshop on Program Comprehension (IWPC'03), pp. 185-195, May 2003.

[146] J. Whitehead (2007). "Collaboration in Software Engineering: A Roadmap". In *2007 Future of Software Engineering* (May 23 - 25, 2007). International Conference on Software Engineering. IEEE Computer Society, Washington, DC, 214-225.

[147] E. J. Whitehead, Jr. and Y. Y. Goland, (1999) "WebDAV: A Network Protocol for Remote Collaborative Authoring on the Web," in 6th European Conference on Computer Supported Cooperative Work (ECSCW'99), Copenhagen, Denmark, 1999, pp. 291-310.

[148] P.Wilson, (1991). "Computer Supported Cooperative Work: An Introduction". Kluwer Academic Pub, 1991.

[149] K. Wong. "The Reverse Engineering Notebook", Ph.D Thesis, University of Victoria, 2000.

[150] I. Zayour and T.C. Lethbridge, "A Cognitive and User Centric Based Approach For Reverse Engineering Tool Design", Proceedings of the CASCON 2000.

[151] U.Zdun and P.Avgeriou, "Modelling Architectural Patterns Using Architectural Primitives", OOPSLA 2005

[152] M.V. Zelkowitz and D.R. Wallace (1998). "Experimental models for validating technology". IEEE Computer, 31(5):23-31