Available online at www.sciencedirect.com

SCIENCE DIRECT

Int. J. Human-Computer Studies 63 (2005) 607–627

International Journal of Human-Computer Studies

www.elsevier.com/locate/ijhcs

Evaluation of integrated software development environments: Challenges and results from three empirical studies

Rex Bryan Kline[a,b,*], Ahmed Seffah[b]

[a] *Psychology Department, Concordia University, 7141 Sherbrooke St. W. Montreal, Quebec, Canada, H4B 1R6*

[b] *Human-Centered Software Engineering Group, Computer Science and Software Engineering Department, Concordia University, 1455 DeMaisonneuve Blvd. W. Montreal, Quebec, Canada, H3G 1M8*

Received 30 July 2004; received in revised form 22 March 2005; accepted 20 May 2005
Available online 22 July 2005
Communicated by D. Boehm-Davis

Abstract

Evidence shows that integrated development environments (IDEs) are too often functionality-oriented and difficult to use, learn, and master. This article describes challenges in the design of usable IDEs and in the evaluation of the usability of such tools. It also presents the results of three different empirical studies of IDE usability. Different methods are sequentially applied across the empirical studies in order to identify increasingly specific kinds of usability problems that developers face in their use of IDEs. The results of these studies suggest several problems in IDE user interfaces with the representation of functionalities and artifacts, such as reusable program components. We conclude by making recommendations for the design of IDE user interfaces with better affordances, which may ameliorate some of most serious usability problems and help to create more human-centric software development environments.

© 2005 Elsevier Ltd. All rights reserved.

Keywords: Software development environment; Integrated development environment (IDE); CASE tools; Usability; User interfaces; User-centered design

*Corresponding author. Psychology Department, Concordia University, 7141 Sherbrooke St. W. Montreal, Quebec, Canada, H4B 1R6. Tel.: 514 848 2424x7556; fax: 514 848 4523.

E-mail addresses: rbkline@vax2.concordia.ca (R.B. Kline), seffah@cs.concordia.ca (A. Seffah).

1071-5819/$ - see front matter © 2005 Elsevier Ltd. All rights reserved.
doi:10.1016/j.ijhcs.2005.05.002

1. Introduction

This article is concerned with the usability of a type of computer-assisted software engineering (CASE) tool known as the integrated development environment (IDE). Other terms, such as integrated design environment, integrated debugging environment, or integrated CASE environment, are also used to refer to IDEs. Briefly, an IDE is computer software that generally consists of a source code editor, a compiler or interpreter (or both), build-automation tools, and a debugger. Many IDEs also have a version control system or tools for building a graphical user interface (GUI). Examples of IDEs for Java programming include Forte (Sun Microsystems), VisualAge for Java (IBM), JBuilder (Borland), Visual Café (Symantec), and Visual J# (Microsoft), and examples for C++ programming include Visual C++ (Microsoft) and C++ Builder (Borland). Ideally, use of an IDE should result in greater productivity compared with the use of multiple single-purpose CASE tools for program development, such as a text editor and a separate compiler (e.g., NEdit and GCC for C++ programming in Unix). Whether this ideal is realized in practice is considered next.

In the IEEE's Software Engineering Body of Knowledge (SWEBOK), Carrington (2004) specified the goals that a software development environment (which includes IDEs) should meet. Such environments should (1) reduce the cognitive load on the developer; (2) free the developer to concentrate on the creative aspects of the process; (3) reduce any administrative load associated with applying a programming method manually; and (4) make the development process more systematic.

Unfortunately, there is evidence that these goals are often not met in practice and that poor IDE usability is a culprit.[1] For example, IDEs are described by many developers as difficult to learn and use (Seffah and Rilling, 2001). Printed documentation for developers, online help, and related training materials are often presented in language that is precise but esoteric and difficult to understand (Seffah, 1999). This is one of the major factors that contribute to rejection of CASE tools (IDEs or otherwise) by developers (Standish Group, 1994). Lundell and Lings (2004) argued that the continuing gap between increasing features on the one hand and actual feature usage on the other contributes to overly optimistic expectations on the part of developers and management about an IDE's relative advantage. These unrealistic expectations may increase the likelihood that an IDE winds up unused (i.e., it is rejected). This concern is also consistent with the observation that some of

[1]We emphasize that IDE *usability* is not synonymous with IDE *usefulness*. For example, the ISO/IEC 9126–1 (2001a) standard defined usability as a software quality attribute that can be decomposed into five different factors, including understandability, learnability, operability, attractiveness, and usability compliance (with published style guides or conventions for user interfaces). The ISO/IEC 9216–4 (2001b) standard defined the related concept of *quality in use* as a kind of higher-order software quality attribute that can be decomposed into three factors, effectiveness (i.e., usefulness), productivity, and safety. Some software cost estimation methods, such as COCOMO II (Boehm et al., 2000), take account of the impact of the software development environment on these areas. However, a cost estimate associated with a particular suite of CASE tools could be rendered inaccurate if poor usability compromised effectiveness, productivity, or even safety.

the features most heavily advertised by IDE vendors may be viewed as only somewhat useful by developers (Maccari and Riva, 2000).

Kemerer (1992) reported that about 70% of CASE tools for program development (not necessarily IDEs) were not being used one year after their purchase and only about 5% were widely used but not to capacity. In a survey of software development companies, Iivari (1996) found that the single most important factor in predicting software development tools usage was perceived voluntariness. Specifically, if developers believed that tool use was up to them, they tended *not* to use the tool. There are also concerns that the representation of software artifacts, such as classes in an object-oriented programming environment, often does not facilitate program comprehension. For example, developer productivity may not be significantly improved when using visual instead of textual representations (Blackwell and Green, 1999). All of these results indicate that, in general, IDEs and related tools have not had the positive effects on developer productivity and software quality anticipated in the early 1980s. This is unfortunate because the cost of adopting an IDE is not insignificant: It can be as high as about $20,000 (US) per developer after all product and training costs are considered (Lending and Chervany, 1998).

The next two sections consider special problems in the design of usable IDEs and the evaluation of IDE usability.

2. Challenges in the design of usable IDEs

There are several reasons why it is difficult to design usable IDEs. These tools are typically very complex computer applications with large numbers of different kinds of functionalities. These functionalities often appear in quite different modes or contexts that include source code editing, dynamic debugging, file linking and management, and the viewing of relations between structural elements of the application being developed (i.e., program comprehension), among others. Each context may require somewhat different functionality representation or user interface organization. For example, it may be beneficial to design the basic user interface for a source code editor to resemble that of a standard word processor, which capitalizes on the developer's familiarity with this type of general office productivity tool for manipulating text. In contrast, a reusable components browser may need an advanced user interface that exhibits different visualizations to facilitate program comprehension. This requires a much higher level of integration of textual and graphical representation of program elements, such as classes, packages, or data structures. It is no simple task to design a user interface for an integrated application that is equally consistent and usable across many different contexts or modes.

It is also more difficult to apply certain design principles from the human–computer interaction (HCI) area to a domain like IDEs, which should support both the cognitive processes of individual developers and collaboration among members of a development team. For example, Norman (1990) described some general principles, including affordances, visual constraints, and mapping, that can be applied to the design of everyday objects, such as cameras or telephones. These design principles

concern the association between functionalities and how they are represented in some type of control mechanism or user interface. Affordances concern whether the perceived properties of an object indicate how that object can be used. An everyday example of poor affordance would be a door equipped with both a handle and a push plate. This combination of features would not give a clear indication as to which action to take. Clear affordances help users know what to do and how to do it; they also help them steer them away from making mistakes and encourage awareness of alternatives, shortcuts, and the like.

Constraints and mapping concern limitations on the number of possible features and the conceptual relations between them. There are different types of constraints including physical (e.g., components that can only be assembled in the correct way), cultural (e.g., red lights placed at the rear of automobiles), and semantic (e.g., a sets of words makes sense only if presented in a certain sequence). All the examples just listed can also be understood from the perspective of affordance. Mapping defines the set of possible relations between program artifacts such as classes and functionalities. In the HCI area, it is generally believed that well designed, perceivable user interface affordances in computer applications can directly enhance usability and learnability through reduction of the need for instructions, user manuals, and on-line help and support (e.g., Preece et al., 1994).

The problem is that software is not a real object—it is inherently an abstraction without physical properties that can be readily mapped to a control mechanism or user interface. Indeed, one could easily argue that software artifacts and relations, such as classes and inheritance, have more in common with hypothetical constructs than with real-world objects. Thus, it is more difficult to apply design principles from "real thing design" when the product itself, in this case an IDE, is an abstraction (see also Brooks, 1987). However, the empirical results described later suggest that some key usability problems of IDEs are related to poor use of visual constraints, mapping, and affordances in the user interfaces of such tools. It is argued later that more consistent application of these design principles—however challenging it may be to do so—has the potential to minimize some of the more serious usability problems of IDEs. Similar kinds of issues have been described concerning visualizations of software artifacts (e.g., Knight and Munro, 2000).

3. Challenges in the study of IDE usability

Budgen and Thomson (2003) noted that because complex CASE tools, such as IDEs, typically offer a large number of functionalities, it is virtually impossible to study the usability of all of them in a single investigation. These same authors also argued that an integrative approach is needed to understand IDE usability. This is because in order to effectively evaluate the usability of an IDE, one must consider characteristics of programmers (e.g., background, level of training) and those of computer systems (e.g., the user interface) together with the application domain. That is, Budgen and Thomson (2003) basically assumed that

the usability of IDEs are emergent properties of tool, developer, and domain characteristics. Ramage and Bennett (1998) took a similar view of software maintenance, which they proposed is affected by not only the source code and skill of the maintenance programmers, but also by other factors including management and business processes. This view is consistent with a basic principle of user-centered design that understanding both the user and how that user interacts with a computer tool in a particular context is just as important as the understanding the tool itself. That is, the emphasis is on the evaluation of the functionality of computer tools and the degree of the match between those functionalities and the needs of users.

Budgen and Thomson (2003) also argued that the evaluation of the usability of complex CASE tools requires a framework that integrates research methods and measurement theory from both computer science and the behavioral sciences. Methods for usability evaluation include interviews, task analysis, direct observation, questionnaires, and heuristic evaluation, among others (see Barnum, 2002). The use of multiple measurement methods is needed to deal with the potential problem of common method variance. Suppose that the suitability of a user interface is evaluated with three different questionnaires and it is found that their scores are highly correlated, which in this context means that scores from the same person tend to maintain their rank order across the different questionnaires. This result could have occurred because of some artifact in the questionnaire method, such as a response bias, rather than the result of the measurement of common underlying usability constructs. The idea that ones needs to use multiple measurement methods is not new in software engineering. For example, it is generally understood that multiple methods of testing are necessary in order to detect defects in software (e.g., Juristo et al., 2004).

Another problem is the paucity of measures of user satisfaction that are specific to software developers. Perhaps most measures of user satisfaction are questionnaires, and there are many such questionnaires for users of software products like general office productivity tools or Web sites. Unfortunately, the psychometric properties of perhaps most of these user satisfaction questionnaires are poor (LaLomia and Sidowski, 1991). For example, some questionnaires have only a single scale that assesses global satisfaction. That is, they are unidimensional measures, but user satisfaction is undoubtedly a multidimensional construct (e.g., Lewis, 2002). Most of these questionnaires also lack standardization samples and associated norms, which means that there is little basis to interpret the meaning of lower versus higher scores. Suppose that a user satisfaction questionnaire has 20 items where each item is scored as "1" if the response is positive or as "0" otherwise. An individual's score on the questionnaire is 12. What does this result mean? Does it indicate low, adequate, or high satisfaction? Without a comparison group, it is often difficult to say. The multidimensional user satisfaction questionnaire used in one of the empirical studies described later has a normative sample, but it is not comprised of software developers. However, we are not aware of any user satisfaction questionnaire normed in a sample of software developers, which is a very specialized and expert group. It should be noted that several multidimensional user satisfaction

questionnaires with good psychometric characteristics have been constructed for end users of information systems in business settings (Zviran and Erlich, 2003). Some of the most widely used of these questionnaires in this area includes ones by Bailey and Pearson (1983), Doll and Torkzadeth (1988), and Chin et al. (1988). However, these questionnaires were not intended specifically for software developers unless they happen to also use information systems computer programs.

4. Empirical studies of IDE usability

This section describes the sequential application of four different methods over three empirical studies in order to identify increasingly specific kinds of IDE usability problems. The results of earlier studies also influenced the selection of outcome measures for later studies. In Study 1, unstructured interviews were conducted with developers about their experiences using IDEs for Java programming. In Study 2, the methods of heuristic evaluation and psychometric assessment were applied in samples of experienced and novice programmers who used the same IDE for C++ programming. In Study 3, developers-in-training who were more or less experienced with the same C++ IDE were observed in the laboratory as they tried to solve common types of programming tasks.

4.1. Study 1 (unstructured interviews)

4.1.1. Method and participants

Seffah and Rilling (2001) interviewed developers who used Java IDEs, such as Forte (Sun Microsystems), VisualAge for Java (IBM), JBuilder (Borland), Visual Café (Symantec), or Visual J++ (Microsoft). To limit the scope of the survey, all interviews focused on the developer's experiences with GUI builders embedded in such tools. A total of 15 software developers from five different companies participated in these interviews. Three developers from each company had one of the following backgrounds: (1) inexperienced users (including new hires) who had used a Java IDE for less than three months; (2) occasional users, such as team leaders or senior system architects, who used Java IDEs so infrequently that they may forget how to complete certain tasks; and (3) experienced Java IDE users who may still need to learn how to use new (for them) functionalities, such as how to effectively use Java Beans.

4.1.2. Results

During a 2-h recorded interview, three participants from each company (all present at the same time) were asked to give their overall impression of the Java IDEs they used, give their top five usability problems encountered, and demonstrate these problems to the interviewer through typical situations (if possible). The interaction among the participants often raised additional issues and common problems with their use of Java IDEs. The following selected exchange among

developers with different levels of experience highlight some of their difficulties in using IDEs:

> *Inexperienced user:* There are so many buttons, menus, and windows that I do not necessarily need. I'm very busy and I have no time to close, resize, and move this large mosaic of windows. More than this, I cannot understand what the system is doing.
>
> *Experienced user:* I'm familiar with different Java IDEs, and each one has its own operating mode—there is no standard. Developers should understand and master this mode, but it is time consuming. If you haven't time, use a text editor and the JDK compiler. You have a better chance of becoming productive in a short time.
>
> *Interviewer:* How much time is required to master the operating mode—how the system works?
>
> *Experienced user:* A few months, a few weeks, a few days or a few hours? It's so much and we haven't got the time for training, reading manuals, or supporting junior developers. We have to deliver, but we can't learn while developing. Integrating learning in the IDE can reduce training time and costs.
>
> *Occasional user:* Well, one of my problems is that I cannot build a mapping between what is in the different windows and my program's overall structure. The system breaks up the mental picture of a program into different windows with no clear relationships between them. I need something that can help me see the program's big picture (as it is in my mind) while I'm focusing on a specific piece of code. This is the kind of support that is needed.
>
> *Interviewer:* Do you think that this functionality doesn't exist?
>
> *Occasional user:* It may exist, but I'm sure that it would take time to master, which makes it no longer useful.

Upon completion of all the interviews, we compiled and compared the comments gathered at the five companies. The top five usability problems in terms of frequency across all developers are listed next:

1. Relevant program artifacts, such as classes and interfaces, and IDE functionalities are not always visible on-screen when they are needed. Likewise, irrelevant or rarely needed artifacts and functionalities compete with the relevant ones and diminish their relative visibility.
2. The memory load is too high. For example, developers should not be obliged to remember functionalities or program artifacts across different dialogs or windows.
3. The IDE does not speak the developer's language in error messages, dialogs, etc. Words, phrases, and concepts with which developers are generally familiar should be used instead of system-oriented vocabulary.

4. The developer is not always kept informed about the status of the IDE or the application being constructed. That is, feedback is too often not contextualized.
5. The IDE does not do enough to prevent mistakes, even obvious ones. Even better than clear error messages is an environment that prevents mistakes from occurring in the first place.

The kinds of usability problems just described—visibility, operability, consistency, error handling, and high memory load—are well known in the HCI area. They have been previously described for Web sites or office productivity tools, but these results suggest that these usability issues are significant for integrated CASE tools, too. It is argued later that these and other results described next are in part consequences of poor affordances in how functionalities are represented in the user interfaces of many IDEs.

4.2. Study 2 (Heuristic evaluation and psychometric assessment)

4.2.1. Methods and participants

Kline et al. (2002) used the methods of heuristic evaluation and psychometric assessment with questionnaires to evaluate a widely used IDE for C++ programming, Visual C++ (Version 6, Microsoft). The idea of heuristic evaluation is quite straightforward: It is an inspection method where usability experts evaluate a user interface against a set of accepted principles, or heuristics. Each expert makes a list of usability problems that he or she should prioritize according to judged seriousness. Heuristic evaluation is related to the technique of software inspection where external experts check system representations, such as the requirements document, design artifacts, or source code. However, there is a significant challenge in applying heuristic evaluation to the study of IDEs: Usability experts, who tend to have backgrounds in the behavioral sciences, may not also be experts in programming; therefore, their ability to evaluate the usability of an IDE may be limited. We deal with this problem by giving to a total of seven professional C++ programmers the definitions of the heuristics adapted from Nielsen (1994) and presented in Table 1 before asking them to inspect the user interface of an IDE with which they were very familiar, Visual C++. The programmers had relatively similar backgrounds and demographic characteristics: All but one were male, most were in their late twenties or early thirties, and they had on average about 10 years of general experience with computers and about 5 years of professional programming experience.

A multidimensional user satisfaction questionnaire was also administered to the seven professional C++ programmers. The particular questionnaire used was the 50-item Software Usability Measurement Inventory (SUMI; Kirakowski and Corbett, 1993). The SUMI assesses the five separate aspects of user satisfaction described in Table 2 and also includes a global satisfaction scale. The psychometric properties of these scales are generally adequate. For example, internal consistency reliabilities coefficients range from about .70 for the Control scale to about .90 for the Global scale (Kirakowski and Corbett, 1993). A distinguishing characteristic of the SUMI is that it has a relatively large standardization sample of about 1000 cases. Through norms established within this sample, user ratings can be described in a standardized

Table 1
Definitions of heuristics

Heuristic	Definition
Visibility	Informs user about program status
Real-world match	Speaks developer's language (e.g., dialogs are understandable)
Control	User feels in control of what they are developing (the application), and not just controlling the IDE
Consistency	Coherent, predictable system behavior and appearance
Error handling	Clear error messages and recovery
Recognition, not recall	Minimizes memory load
Flexibility	Suits users of varying experience
Minimalist design	Shows only relevant information
Relevant help	Easy to search and find information

Note. Adapted from Nielsen (1994).

Table 2
Content scales of the Software Usability Measurement Inventory (SUMI)

Scale	Definition (whether users...)	No. of Items
Efficiency	View the program as efficient	10
Affect	Enjoy using the program	10
Helpfulness	Feel the program assists in its use	10
Control	Feel they are in control of the program	10
Learnability	Believe the program is easy to learn	10
Global	Overall satisfaction	25

metric. Specifically, standard scores on the SUMI are expressed in a scale where higher scores indicate greater satisfaction, the average is 50, and the standard deviation is 10. For example, a score of 60 is higher than about 85% of the scores in the SUMI normative sample and may indicate relatively high user satisfaction. In contrast, a score of 40 is higher than only about 15% of the scores in the normative sample and may suggest user dissatisfaction. The SUMI scoring program, SUMISCO, also identifies individual questionnaire items where the respondents report statistically lower levels of satisfaction compared to the normative sample. Inspection of these critical items may identify specific kinds of usability problems.

The people that comprise that SUMI normative sample are not specifically software development professionals. Instead, they are end users of general computer office productivity tools, such as word processors and spreadsheet programs, in business settings. Consequently, SUMI standard scores (and associated percentiles) are not relative to software development professionals. As noted earlier, though, we are not aware of any multidimensional user satisfaction questionnaire where software developers only make up the normative sample much less developers who specifically use IDEs. Also, aspects of user satisfaction measured by the SUMI, such as efficiency or learnability (see Table 2), are just as relevant in their use of IDEs as for more general business users of office productivity software.

In order to compare ratings of the experienced C++ programmers on the SUMI with those of less experienced users of the same IDE, the SUMI was also administered to 100 computer science students who were enrolled in an advanced course in C++ programming where Visual C++ was used in course computer laboratories. The average age of the students was 25.5 years (standard deviation = 5.6 years), and 71% were men and 29% were women. A total of 43% of the students reported using the Visual C++ for at least one year, 25% for 6–12 months, 18% for 3–6 months, and only 14% for less than 3 months. A total of 80% reported using computers for any purpose (e.g., for electronic mail or word processing) for at least 3 years. The participation of the students in this study was entirely voluntary and had no bearing on their course standing. To reinforce this perception, all questionnaires were completed anonymously, and students who declined not to participate could not be individually identified. Only one student declined to complete the questionnaire.

4.2.2. Results

The experienced developers rated the user interface of Visual C++ against the nine heuristics listed in Table 1. Their rating form also provided room for general comments about usability problems that did not correspond to any of the heuristics. Review of the comments by the experienced developers generally indicated few usability problems for the control, flexibility, and minimalist design areas. Other areas did not fare so well. A notable example is error handling, for which many problems were noted including ambiguous messages and poor system error representation. Experienced developers who used the rated C++ IDE for testing generally also complained of insufficient information about recovering from errors. Concerns about inadequate feedback from the C++ IDE were also noted for the visibility heuristic. For example, the experienced users reported that the IDE may just simply stop responding in some situations when there is an error in the source code. Other concerns were expressed about the C++ IDEs real world match: Some of its dialogs use specific technical terminology or are so ambiguously worded such that even experienced developers had difficulty understanding them.

Presented in Fig. 1 is the median SUMI satisfaction profile with 95% confidence intervals for the seven experienced developers. With one exception, these scores suggest about average overall levels of satisfaction with the rated C++ IDE. The exception is learnability, for which the median score of about 40 is about full standard deviation below what is typical in the SUMI's normative sample (50). That is, the C++ IDE was described as generally difficult to learn. Presented in the top part of Table 3 are examples of critical SUMI items rated as more problematic than expected compared with the test's normative sample. For several of these items, the developers describe the C++ IDE as being difficult to learn or as having unhelpful documentation. Other critical items mention possible problems with program stability and difficulty re-starting the program, while others note that the on-screen information is sometimes not very useful. It is noteworthy that many of these same problems were found in the heuristic evaluation with these same participants.

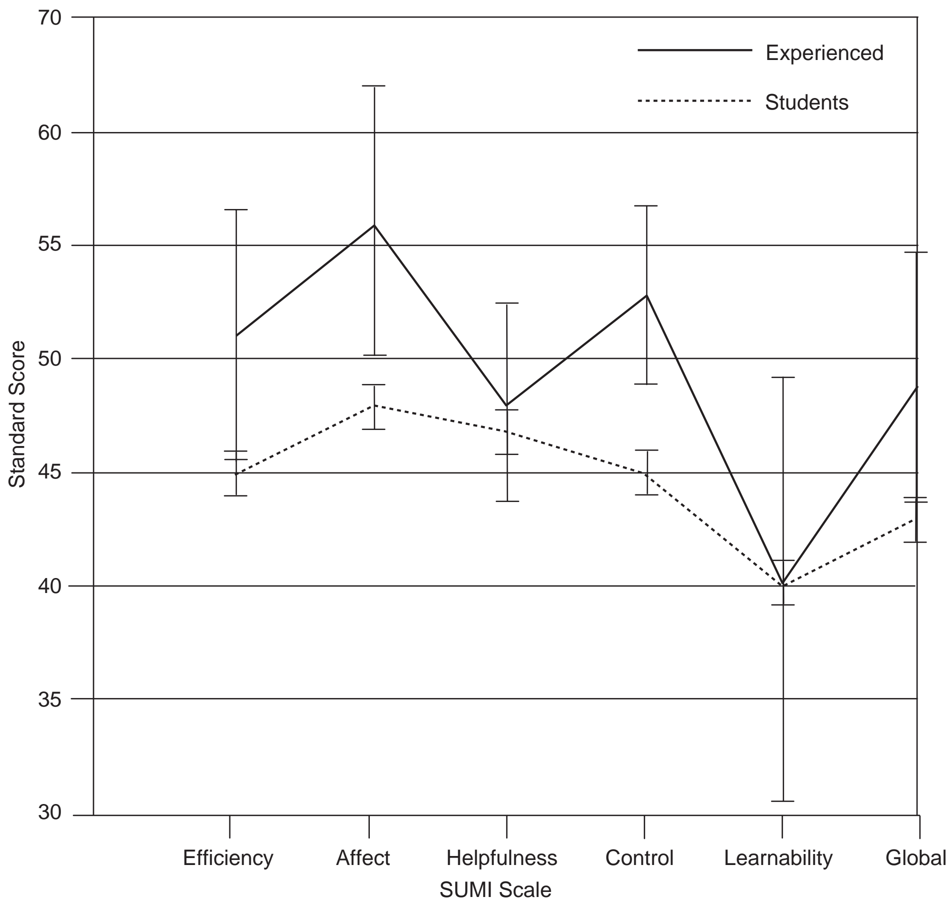


Fig. 1. Median profiles with 95% confidence intervals for experienced programmers ($n_1 = 7$) and students ($n_2 = 100$) on the Software Usability Measurement Inventory (SUMI) concerning the same C++ integrated development environment. *Note*: Higher scores indicate greater satisfaction.

The median SUMI satisfaction profile for the 100 students is also shown in Fig. 1. The 95% confidence intervals for the students are narrower than for the experienced developers because of the much larger group size for the former. The pairs of confidence intervals for the two groups do not overlap for the Affect scale and the Control scale of the SUMI. Thus, the medians for these two scales are statistically different at the .05 level. However, the power of statistical tests that compare the medians for the two groups is expected here to be low because of the small size of the group of experienced developers ($n = 7$). Thus, it is surprising that *any* statistical differences were found. In general, the students expressed less satisfaction with the same C++ IDE rated by the experienced developers. Specifically, the median scores for the novice developers were about average (50) in the areas of affect and program helpfulness. However, the below-average score of about 40 on the learnability scale is

Table 3
Examples of critical items on the Software Usability Measurement Inventory (SUMI) for experienced and novice developers

Experienced developers
There is too much to read before you can use some elementary features
I will never learn to use all that is offered in this software
I feel safer if I use only a few familiar commands
I feel in command of this software when I am using it (disagree)
It takes too long to learn the software commands
I keep having to go back to look at the guides
Students
There is never enough information on the screen when it's needed
It takes too long to learn the CASE tool features
Learning to operate this software initially is full of problems
Learning how to use new functions is difficult
I feel safer if I use only a few familiar commands
I prefer to stick to the facilities that I know best

comparable to that of the experienced developers on the same scale (see Fig. 1). Thus, both groups described the C++ IDE as being relatively difficult to learn. The students also reported additional problems in the areas of global satisfaction, control, and efficiency. Presented in the bottom part of Table 3 are examples of critical SUMI items for the students. Specific problems reported more often than expected compared with the test's norms include the lack of helpful information on the screen when it is needed, difficulties learning new functions, trouble moving data in and out of the program, and disruption of a preferred way of working.

It is not surprising that experienced developers reported greater overall satisfaction with the rated IDE for C++ than the students. Of greater interest are the areas of agreement across the two groups: Both experienced developers and students alike described the C++ IDE as difficult to learn and also mentioned concerns about lack of understandable on-screen information. Thus, relatively poor tool learnability and visibility were mentioned regardless of experience level. These points of agreement between the students and professional developers are also consistent with results of other studies of programmer perceptions of IDEs cited earlier (Section 1). These results also indicate that more novice users have some unique usability concerns compared with more experienced users of the same IDE, especially in the areas of control and efficiency. This suggests that learning resources, such as user manuals or online help systems, aimed at very experienced developers may not fully meet the needs of less experienced developers.

4.3. Study 3 (laboratory observation and the cognitive walkthrough)

4.3.1. Method and participants

Even more specific behavioral information about usability can be obtained through direct observation of how users interact with a computer program than

through unstructured interviews, heuristic evaluation, or questionnaires. Naghshin et al. (2003) observed in a laboratory setting a total of six developers-in-training as they worked with the same IDE evaluated in Study 2 (Visual C++) to carry out a set of common programming tasks. The participants were all computer science graduate students with similar academic backgrounds in C++ programming in particular and object-oriented programming in general. They were all currently involved in C++ programming during their graduate studies or work in industry, but at two different levels of experience. A total of three participants had extensive prior experience with the C++ IDE and thus could be considered as more expert users. The other three participants were less experienced users of the same IDE. This is mainly because they had been using other CASE tools besides Visual C++ (not necessarily IDEs) for C++ programming.

All participants were asked to carry out five different tasks common to the use of basically any IDE, including (1) creation of a project, (2) editing of source code, (3) compilation of the whole application or part of it, (4) debugging, and (5) archiving the generated application, related code source, and documentation. Efforts were made to ensure that these five tasks did not require specific knowledge of the C++ programming language per se. Each participant was asked to carry out these tasks using the C++ IDE to work with a sample program that concerned an elevator simulation. It consisted of about 1000 lines of code stored in 11 different files. There were six major classes in the sample C++ program for things like controlling the state of the elevator and responding to requests for service. For example, the participants were asked to find the source of a known "bug" (e.g., the elevator would not travel from the first to the second floor), correct it, and recompile the modified program. The participants were also asked to describe any problems encountered while attempting to carry out the tasks. This aspect of the laboratory observations is similar to idea of the cognitive walkthrough, which is a usability review technique. Briefly, it involves asking users to demonstrate how they would perform certain tasks by mentally "stepping through" a user interface. User comments are then analysed concerning learning and effectiveness, such as how well the user interface supports learning by doing or a particular task; see Wharton et al. (1994) for more information.

Each participant was observed individually while he or she worked alone at a computer in one room while a researcher monitored the session from another computer located in a separate room. Each session lasted about 1 h. The room for the participants was equipped with a video camera for recording the participant's use of the keyboard and mouse. The Adobe Premier application was used to import the images of keyboard and mouse interaction from the camera via a USB port. The Camtasia Studio application was used to record and edit these video images. Both applications just mentioned were controlled within the Timbuktu Pro Enterprise Edition Multi-Platform Remote Control application. It also displayed the participant's screen on the display of the researcher's computer and recorded all the participant's interactions (e.g., keystrokes, mouse clicks) with the C++ IDE. The Timbuktu application also supported transfer of files between two different computers through an interconnection network and also communication by

instant message, text chat, or voice intercom. These capabilities allowed the participant to ask a question of the researcher without interrupting the observation session.

The following types of interactions with the C+ + IDE were automatically saved in a Microsoft Excel file for each participant: (1) Use of a pull-down menu (i.e., the menu bar) to select a functionality; (2) use of an icon (i.e., the tool bar) to select an option; (3) interaction with a particular IDE window (e.g., code editor window); (4) interaction with a dialog box; (5) use of a program or operating system service (e.g., in Windows) outside the IDE; (6) use of a right mouse click to display a menu of properties or options; (7) interaction with IDE visualizations (e.g., for classes); and (8) use of a keyboard shortcut (e.g., Crtl+c for copy). Each participant was also asked to "think out loud" while conducting each task, especially to explain any particular difficulty in using the C+ + IDE. All such statements were also recorded.

4.3.2. *Results*

The total number of different types of interactions with the C+ + IDE were tallied for each participant. These totals were then averaged within each group of participants, the more novice users and the more experienced users of the C+ + IDE. Averages for each group were converted to pie graphs that show the relative composition of types of program interactions for each group. The pie graph for the more experienced users is presented in Fig. 2(a), and the one for the more novice users is presented in Fig. 2(b). Differences in interaction patterns between more novice and more experienced users of the C+ + IDE are summarized next:

1. In Visual C+ + (and many other IDEs), the main functionalities are represented in the tool bars as icons as well as in the program menus. Theoretically, the tool bars should be used more often than the menus because the former are more

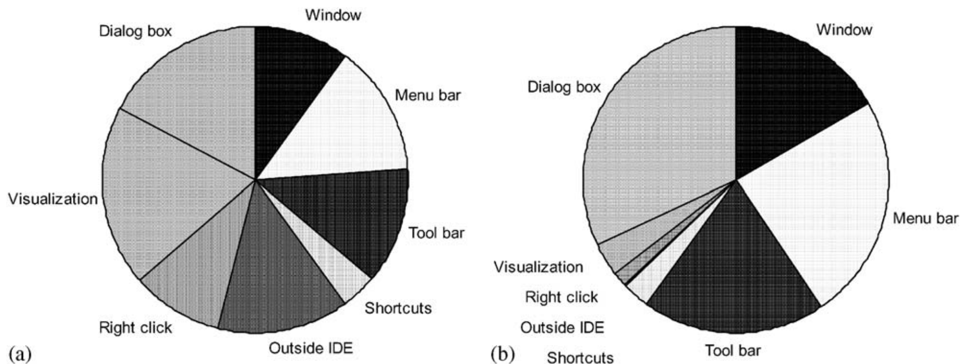


Fig. 2. Overall composition of interactions for more experienced users (a) versus more novice users (b) of the same C+ + integrated development environment (IDE).

quickly accessible. However, the more novice participants did not always understand that the same functionality was available through either a tool bar icon or a menu option. Moreover, when faced with an error message, the more novice participants often tried the same functionality through both the menu and tool bars.
2. The use of a right mouse click to display properties or options was performed mainly by the more experienced participants, which made the interaction faster.
3. Almost no one used a keyboard shortcut except for obvious ones that are generic shortcuts in many Windows applications (e.g., Ctrl+c for copy). However, the participants were observed for only about an hour. They may have started other keyboard shortcuts that are more specific to the C++ IDE for some repetitive tasks if the session were longer.

The number and type of errors for each participant as they attempted to edit, compile, debug, and archive the sample program with Microsoft Visual C++ were also recorded. Review of these records and comments of participants recorded during the test session indicated the most common usability problems in terms of frequency that are listed next:

1. All more novice participants started their tasks by simply opening the main program file without specifying a project. When the modified source code is later compiled, fatal errors related to the linking of files occur because no project was defined. This is an obvious kind of "beginner's mistake," but it was not prevented by the C++ IDE.
2. Mistakes related to link errors were not obvious to the less experienced participants. They had difficulty understanding the nature of the problem, and they tended to try different paths to solving the problem (e.g., compiling then building then running) without much success.
3. As mentioned, some functionalities are represented in two different ways (e.g., menu option or tool bar icon). This double representation seemed to make the user interface overly complex for the more novice participants. It was mentioned that these participants did not always understand the fact that such dual representations concerned the same functionality until they tried both.
4. Both more novice and more experienced participants had trouble understanding error messages during code compilation. All participants found the language of these messages to be overly technical and confusing.
5. All participants complained about the lack of a way to view the structure of the program in a hierarchical way. The absence of this type of view made the sample C++ elevator program more difficult to understand.
6. All participants struggled with limited archival capabilities in the C++ IDE, such as the ability to rename a file or change to a different folder. Participants who tried to perform these tasks outside of the C++ IDE eventually encountered file linking errors.

5. Summary and limitations of the empirical studies

Presented in Table 4 is a summary of IDE usability problems identified across the three empirical studies. The method used to identify each problem is also indicated in the table. Some kinds of identified usability problems were unique to a particular method or study. For example, user complaints about the on-screen presence of irrelevant program artifacts were identified mainly in the unstructured interviews with developers (Study 1), and user complaints that the IDE did not assist in program comprehension were identified mainly with the observational method (Study 3). That some identified usability problems are specific to a particular method is not surprising because not all different methods can capture all the same kind of problems. Other kinds of usability problems were identified with more than one method. This includes the complaint of poor information about IDE status, which was identified with interviews, questionnaires, and heuristic evaluation (Study 1 and Study 2). Another example is the complaint of the excessive use of technical jargon (i.e., real world match) in IDE dialogs, which was identified with interviews, heuristic evaluation, and laboratory observation (all three studies).

It is encouraging that the kinds of usability problems identified with different methods across the three empirical studies are quite similar, which suggests convergent validity. It is also encouraging that these results correspond with those reported by other researchers who have evaluated the usability of CASE tools for program development (e.g., Iivari, 1996; Kemerer, 1992; Lending and Chervany, 1998) and for other phases of the software lifecycle, such as program comprehension and maintenance (e.g., Müller et al., 2000). This consistent pattern of results suggests

Table 4
Summary of identified usability problems by study and method

Usability problem	Study and methods			
	Study 1	Study 2		Study 3
	Interviews	Heuristic[a]	Questionnaire	Lab observation
Relevant artifacts not visible	✓		✓[b]	
Irrelevant artifacts clutter screen	✓			
High memory load	✓			
Too much technical jargon	✓	✓		✓
Ambiguous program status	✓	✓	✓	
Does not help to prevent mistakes	✓			✓
Poor error handling		✓		✓
Difficult to learn	✓		✓	
User does not feel in Control			✓[b]	
User views as inefficient			✓[b]	
Unclear representation of functionalities				✓
Does not aid program comprehension				✓

[a]Experienced developers only.
[b]Novice developers only.

that the kinds of usability problems identified in the three empirical studies may be representative of more general issues of usability and learnability problems associated with different kinds of CASE tools.

Some possible limitations of the empirical studies just reviewed are now noted. There are many IDEs for various application domains or programming languages, probably dozens or even hundreds of different such programming environments by now. We were able to study the usability of only a relatively small number of such tools, including some Java development environments for building GUIs in Study 1 and Microsoft Visual C++ in Study 2 and Study 3. Our samples were generally small and not randomly selected; therefore, it is unknown just how representative they were in terms of developers who regularly use IDEs. This population is probably quite diverse, and we are planning studies that attempt to identify in much larger samples specific developer types and the kinds of the problems each type experiences in IDE use. It seems plausible that developers with different educational or professional backgrounds may have somewhat different needs concerning user interfaces in IDEs. We are currently conducting a series of new studies that are concerned with this problem.

6. From user problems to design recommendations

In this section, we synthesize the results of the three empirical studies in terms of the design principles described earlier (Section 2). Specifically, we propose that (1) poor user interface affordances and constraints and mapping seem to be a major source of the usability problems identified in our studies and related works, and (2) these design principles have the potential, if correctly applied, to minimize some of these problems.

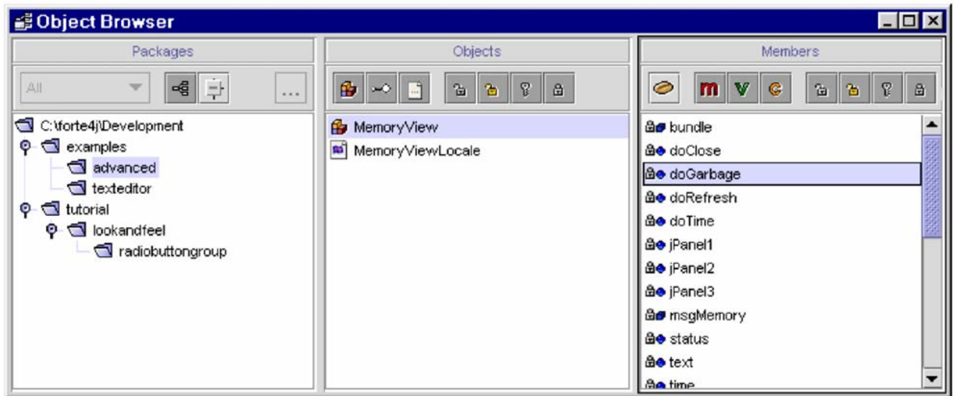
First, we identify three levels where affordance should be improved for IDEs. Without necessarily classifying them in this manner, Blackwell and Green (1999) and others have also made reference to the problem of affordances within the framework of software development environments and visual programming. The first level concerns the affordance of controls, including icons, menus, and especially menu bars. The control's action should be consistent with the representation (abstraction) of that control. The HCI literature on this type of affordance is quite large, and several solutions are possible (see St. Amant, 1999). This being said, we believe that the affordance of controls in IDEs should enable the developer to understand the type of action, how it is implemented, and its effects. The second level concerns reusable components, such as Beans in Java IDEs. These components are generally represented by icons that are very similar to those used to represent controls. Not only do we believe that reusable components require a more distinct representation, we also feel that its affordance should help the developer understand how to manipulate and combine reusable components without being forced to read a very long properties list, as is now the case in some Java IDEs.

The third level concerns textual or graphical representations of programs. With textual representations, different colors or icons may be used to distinguish among program entities such as data structures, classes, and so on. For example, one of the

Java IDEs analysed in Study 1 (Section 4.1) uses a symbol beside a class member to indicate its type (protected, public, etc.) in addition to color-coding the class name to distinguish it from the other program structures. Many developers complained of mental overload created by this double representation. They also said that they frequently needed to add a comment to clarify this notation. We also find that some graphical representations intended to reduce the mental load of the developer sometimes have just the opposite effect. This observation does not question the potential usefulness of graphical representations nor does it discount the potential benefits of software visualization techniques (e.g., Ball and Eick, 1996). Instead, it simply draws attention to the need to study their affordance.

Next, we consider the representation of functionalities in IDEs, which is sometimes characterized by poor use of visual constraints. This makes for poor mapping and ambiguous affordances. Nowhere is this problem more apparent than in tool bars, which represent specific functionalities as icons. These icons may be distributed across several different tool bars in such a way that the mapping between them and program artifacts is unclear. This problem was identified many times by both more experienced and less experienced developers across the empirical studies described earlier. Two examples are briefly considered. Presented in Fig. 3(a) is a

(a)



(b)

Fig. 3. Examples of tool bar organization that show poor constraints and mapping (a) versus a clearer mapping that reduces mental load (b).

screenshot from part of the main window in a Java IDE. A total of over 50 different functionalities are distributed seemingly randomly across the multiple tool bars accessible from this point in the user interface. Even the experienced developers we interviewed in Study 1 (Section 4.1) rarely made use of these tool bars. A counterexample is presented in Fig. 3(b), which shows a class browser with tool bars organized to reflect the object to which the functionalities apply. We observed that this tool bar was frequently used and also seemed to reduce the developer's mental load.

7. Conclusion

In this article, we highlighted the challenges of designing usable IDEs and evaluating the usability of such tools. We also applied different methods to identify increasingly specific kinds of usability problems that developers face in using IDEs, and related the results to design recommendations for improving the user interfaces of IDEs. The kinds of usability problems identified in this work and in related studies (e.g., Iivari, 1996; Kemerer, 1992; Lending and Chervany, 1998) are of obvious concern to practitioners because it seems that they are not deriving full benefit from use of IDEs compared with use of a combination of single-purpose tools, such as a text editor and a compiler.

It is one thing to identify the kinds of usability problems experienced by programmers in their use of IDEs and quite another to know why such problems occur and what to do about them. We argued that the kinds of usability problems observed here and in other studies are in part the consequences of poor affordances in IDE user interfaces, especially in the representation of tools and reusable program components. As discussed by Norman (1990) and others, clear affordances can make program functionalities more accessible and thus more human-centric. Specifically, developers should be provided with IDEs that offer functionalities in more rational, less visually complex formats that reinforce the relation between a specific functionality and the software artifacts on which that functionality acts. For reasons cited earlier (Section 2), it may be more difficult to design user interfaces for IDEs that respect these principles compared with other kinds of computer tools, but it is essential to do so. The usability problems and principles considered here could be incorporated into design guidelines—or perhaps as anti-patterns (e.g., van Welie et al., 2000)—for CASE tools developers. The explicit representation of these issues in such guidelines may help to close the fundamental gap between current functionality-centric IDEs and eventual human-centric IDEs.

References

Bailey, J.E., Pearson, S.W., 1983. Development of a tool for measuring and analyzing computer user satisfaction. Management Science 29, 530–545.

Ball, T., Eick, S., 1996. Software visualization in the large. Computers 29, 33–43.

Barnum, C.M., 2002. Usability Testing And Research. Allyn & Bacon, Boston.

Blackwell, A.F., Green, T.R.G., 1999. Does metaphor increase visual language usability? In: Proceedings of the IEEE Symposium on Visual Languages. IEEE Computer Society, Los Alamitos, pp. 246–253.

Boehm, B.W., Abts, C., Brown, W., Chulani, S., Clark, B.F., et al., 2000. Software cost estimation with COCOMO II. Prentice-Hall, New York.

Brooks Jr., F.P., 1987. No Silver Bullet: Essence and Accidents of Software Engineering. IEEE Computer 20, 10–19.

Budgen, D., Thomson, M., 2003. CASE tool evaluation: experiences from an empirical study. Journal of Systems and Software 67, 55–75.

Carrington, D., 2004. Software engineering tools and methods. In: Bourque, P., Dupuis, R. (Eds.), Guide to the Software Engineering Body of Knowledge (SWEBOK). IEEE Computer Society, Los Alamitos pp. 10–1–10-10.

Chin, J.P., Diehl, V.A., Norman, K.L., 1988. Development of an instrument measuring user satisfaction of the human–computer interface. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. ACM Press, New York, pp. 213–218.

Doll, W.J., Torkzadeth, G., 1988. The measurement of end-user computing satisfaction. MIS Quarterly 12, 258–274.

Iivari, J., 1996. Why are CASE tools not used? Communication of the ACM 39, 94–103.

International Organization for Standardization/International Electrotechnical Commission, 2001a. ISO/IEC 9126-1 Standard-Software Engineering-Product Quality-Part 1: Quality Model. Author, Geneva.

International Organization for Standardization/International Electrotechnical Commission, 2001b. ISO/IEC 9126-4 Standard-Software Engineering-Product Quality-Part 4: Quality in Use Metrics. Author, Geneva.

Juristo, N., Moreno, A.M., Vegas, S., 2004. Reviewing 25 years of testing technique experiments. Empirical Software Engineering 9, 7–44.

Kemerer, C.F., 1992. How the learning curve affects CASE tool adoption. IEEE Software 9, 23–38.

Kirakowski, J., Corbett, M., 1993. SUMI: The Software Usability Measurement Inventory. British Journal of Educational Technology 24, 210–212.

Kline, R., Seffah, A., Javahery, H., Donayee, M., Rilling, J., 2002. Quantifying developer experiences via heuristic and psychometric evaluation. In: Proceedings of the IEEE 2002 Symposia on Human Centric Computing Languages and Environments. IEEE Computer Society, Los Alamitos, pp. 34–36.

Knight, C., Munro, M., 2000. Mindless visualizations. In: Emiliani, P.L., Stephanidis, C. (Eds.), Proceedings of the Sixth ERCIM Workshop User Interfaces For All, pp. 1–14. Retrieved from http://ui4all.ics.forth.gr/UI4ALL-2000/files/Long_papers/Knight.pdf

LaLomia, M.J., Sidowski, J.B., 1991. Measurements of computer attitudes: A review. International Journal of Human-Computer Interaction 3, 171–197.

Lending, D., Chervany, N.L., 1998. The use of CASE tools. Communication of the ACM 41, 49–58.

Lewis, J.R., 2002. Psychometric evaluation of the PSSUQ using data from five years of usability studies. International Journal of Human-Computer Interaction 14, 463–488.

Lundell, B., Lings, B., 2004. Changing perceptions of CASE technology. Journal of Systems and Software 72, 271–280.

Maccari, A., Riva, C., 2000. Empirical evaluation of CASE tools usage at Nokia. Empirical Software Engineering 5, 287–299.

Müller, H.A., Jahnke, J.H., Smith, D.B., Storey, M.-A., Tilley, S.R., Wong, K., 2000. Reverse engineering: A roadmap. In: Finkelstein, A. (Ed.), The Future of Software Engineering. ACM Press, New York, pp. 47–60.

Naghshin, R., Seffah, A., Kline, R., 2003. Cognitive walkthrough + personae = an empirical infrastructure for modeling software developers. In: Hosking, J., Cox, P. (Eds.), Proceedings of the 2003 IEEE Symposium on Human Centric Computing Languages and Environments. IEEE Computer Society, Auckland, pp. 239–241.

Nielsen, J., 1994. Heuristic evaluation. In: Nielsen, J., Mack, R.L. (Eds.), Usability Inspection Methods. Wiley, New York, pp. 25–62.

Norman, D.A., 1990. The Design of Everyday Things. Doubleday, New York.

Preece, J., Rogers, Y., Sharp, H., Benyon, D., Holland, S., Carey, T., 1994. Human–Computer Interaction. Addison-Wesley, Boston.

Ramage, M., Bennett, K., 1998. Maintaining maintainability. In: Khoshgoftaar, T.M., Bennett, K. (Eds.), Proceedings of the International Conference on Software Maintenance. IEEE Computer Society, Los Alamitos, pp. 16–20.

Seffah, A., 1999. Training developers in critical skills. IEEE Software 16, 66–70.

Seffah, A., Rilling, J., 2001. Investigating the relationship between usability and conceptual gaps for human-centric CASE tools. In: Proceedings of the IEEE Symposia on Human-Centric Computing Languages and Environments. IEEE Computer Society, Los Alamitos, pp. 226–231.

St. Amant, R., 1999. User interface affordances in a planning representation. Human-Computer Interaction 14, 317–354.

Standish Group, 1994. Chaos report. Retrieved from http://www.standishgroup.com/sample_research/index.php

Wharton, C., Rieman, J., Lewis, C., Polson, P., 1994. The cognitive walkthrough method: A practitioner's guide. In: Nielsen, J., Mack, R.L. (Eds.), Usability Inspection Methods. Wiley, New York, pp. 105–141.

van Welie, M., van der Veer, G.C., Eliëns, A., 2000. Patterns as tools for user interface design. In: Proceedings of the International Workshop on Tools for Working with Guidelines. Springer, London, pp. 313–324.

Zviran, M., Erlich, Z., 2003. Measuring user satisfaction: Review and implications. Communications of the Association for Information Systems 12, 81–103.